Hardware/Software Resolutions for Pipeline Hazards in Instruction Set Processors

Ing-Jer Huang and Alvin M. Despain

CENG Technical Report 92-18

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213)740-6006

Technical report, 1992

Hardware/Software Resolutions for Pipeline Hazards in Instruction Set Processors

Ing-Jer Huang and Alvin Despain
Advanced Computer Architecture Laboratory
Department of Electrical Engineering - Systems
University of Southern California

Abstract

This report presents a hardware/software concurrent engineering approach to the resolution of pipeline hazards. In automatic pipeline synthesis, a highly pipelined instruction set processor might not be free of pipeline hazards which necessitate software and/or hardware solutions. The register-related pipeline hazards (RPHs) are caused by various types of inter-instruction dependencies. The conventional taxonomy for dependencies (data, anti-data, output) is insufficient for differentiating dependencies in a pipeline structure. We propose an extended taxonomy consisting of nine classes <forward/backward/stationary, data/anti-data/output> for the analysis of RPHs. Hardware and software resolutions are then provided for each class of dependency, including forwarding/duplicate registers in hardware and up/down instruction reordering in the compiler back-end. The combination of hardware and software resolutions can be selected with respect to the characteristics of application benchmarks, in order to balance the performance/cost tradeoff.

1. Introduction

Designing high performance pipelined instruction set processors requires a careful balance between design choices in both hardware and software. The goals of our automation efforts in Advanced Computer Architecture Laboratory are twofold: first, we want to develop a systematic method of constructing both hardware and software components of pipelined ISPs from abstract, sequential behavioral description [1],[3]; secondly, we attempt to exploit the synergy between hardware and software in order th achieve higher performance and better cost-effectiveness for pipelined ISPs. We will addresses the second issue in this report.

This report presents a hardware/software concurrent engineering approach to the resolution of pipeline hazards in instruction set processors (ISPs). The pipeline hazard is the major hurdle in pipeline synthesis. It degrades the pipeline performance and complicates the interaction between micro-architectures and compiler back-ends. We first investigate the complication of hardware and software.

Pipelining is an effective way to improve the throughputs of ISPs by overlapping the computation of consecutive instructions. However, both hardware's and software's aspects of the systems may be affected or modified by pipelining. In hardware, micro-operations with higher degree of concurrency result in faster instruction firing rate at the costs of some extra hardware and more complex control such as duplicate functional units and bypassing buses. The relative timing of micro-operations is modified such that their external behavior may be different from the expected behavior of the original sequential semantics. The 'delayed load' in today's pipelined RISC processors is a typical example: the consumer instruction immediately following the producer 'load' will not be able to use the loaded data since they won't be available until several cycles later; therefore, the consumer of the data has to be scheduled several cycles after the producer, leaving some NO-OP slots between the producer/consumer pair. Thus the sequential semantics of instruction execution is not preserved in the pipelined RISC processors.

The incompatibility between pipelined and sequential implementations calls for the modification of the software to compensate for such difference. Moreover, the software may be modified to take advantage of the hardware's pipeline structure. For example, modern compilers for pipelined instruction set processors have the explicit knowledge of the pipeline structure. A proper amount of NO-OPs is inserted between some particular dependent instruction pairs such as instructions after 'delayed load' or 'delayed branch', in order to preserve the desired behavior. The compilers can optimize the performance of the compiled codes by reordering instructions to eliminate the NO-OP slots. On the other hand, the limitation of compilers and the characteristics of application benchmarks may in turn pose their impact on the hardware pipeline structure. For example, the small average size of basic blocks in application benchmarks and the complexity of instruction reordering in a compiler back-end may void a highly pipelined hardware design: most of the stages are executing NO-OPs since the compiler is unable to find useful instructions to be scheduled into the NO-OP slots.

The driving force behind such hardware and software interactions is the pipeline hazard caused by the desire to highly pipeline the ISP. A *pipeline hazard* happens when the previous instructions do not release resources in time for the next instruction which depend on the same resources such that the next instruction is prevented from executing during its designated cycle. Pipeline hazards are characterized into three categories: *structural, data*, and *control* [4]. They are caused by the conflict use of functional units, registers, and program counter, respectively. In fact, the control hazard is a special case of data hazards with the datum being the program counter. In this report we will focus on the register-related pipeline hazards (RPHs)[1] (data and control hazards).

In this report we will address the problem of RPH resolution in pipeline synthesis. First, we will present an extended taxonomy of inter-instruction dependencies for the analysis of RPHs. RPHs are caused by various types of inter-instruction dependencies. However, while trying to relate RPHs to these dependencies, we found that the conventional taxonomy of dependencies, i.e., data, anti-data, and output dependency, is insufficient to encapsulate the relative timing of register accesses in pipeline stages. Therefore, we further differentiate each class of dependency into forward, backward, and stationary dependencies.

1. The structural hazards exist where the functional units have multi-cycle execution time but are not fully pipelined [4]. This problem can be bypassed in a synthesis environment where this type of functional units is not supported.

Secondly, we will provide hardware and software resolutions for each type of dependency. These resolution techniques include inserting forwarding/duplicate registers in hardware, and generating compilation guides (instruction reordering) to compiler back-ends. The performance/ cost tradeoff of these techniques will be measured with respect to a set of application benchmarks.

These techniques have been integrated into Piper, our pipeline synthesis system for instruction set processors. Piper automatically generates pipelined RTL designs and interfaces to compiler back-ends, from abstract behavioral descriptions of ISPs [1]. In this report, Piper will serve as the platform for discussion and demonstration of these techniques.

The rest of this report is organized as follows. Section 2 reviews the related work. Section 3 overviews the structure of Piper. Section 4 lists the extended taxonomy of inter-instruction dependencies and their relation to pipeline hazards. The hardware and software resolution strategies are outlined in Section 5. The consideration of tradeoffs between hardware and software resolution is discussed in Section 6. Section 7 shows an example of applying above techniques to synthesize a simple pipelined ISP. We conclude this report with a report on current status and future direction in Section 8.

2. Related work

Most of the existing pipeline synthesis techniques, such as SEHWA [5], HAL [6], CATHEDRA II [7] and PLS [8], are developed for DSP applications. These synthesis techniques avoid pipeline hazards by limiting the degree of pipelining: the pipelined machines generated should not have a *pipeline latency* (the number of clock cycles between the firing of consecutive computing tasks) less than the *minimal achievable latency* (MAL) [8] which is the minimal number of clock cycles before the execution of the next task, in order to prevent pipeline hazards. Pipelined instruction set processors synthesized with these techniques will suffer from the lower degree of pipelining. For example, the MIPS R2000 would have to fire an instruction every two cycles (MAL=2), resulting in three pipeline stages as opposed to five, if it were synthesized with these technique [1].

ASPD improves the achievable degree of pipelining, at one instruction per cycle, with an enhanced percolation scheduling algorithm [9]. ASPD deals with the pipeline hazards by flushing the pipeline. As soon as an instruction which may cause hazards is decoded, the pipeline is flushed, no matter whether it really causes hazards or not. For example, whenever a ‘delayed load’ is decoded, the pipeline is flushed, regardless of whether the succeeding instructions depend on the ‘delayed load’. This approach leaves no room for compilers to utilize the delay slots.

While these systems deal with hardware only, the PEAS project addresses the hardware/software concurrent engineering for instruction set processors [10]. It automatically generates a micro-architecture and software components such as compilers and simulators from a set of application benchmarks. A proper set of instruction is first selected from a pre-defined super set according to the benchmarks. A parameterized micro-architecture is then specialized. A compiler and a simulator are also customized from their super sets (the GNU’s C compiler and simulators). The difference between Piper and PEAS is that PEAS currently does not handle pipelined processors, and deals with a super set of pre-defined instructions, while Piper does not select instructions but accepts any given instruction set written in a subset of Prolog [1] and generates pipelined processors.

3. Overview of Piper: a high level synthesis system for pipelined instruction set processors and compiler back-ends

Piper and its pre-processor, Fiper, serve as the behavioral domain of ADAS, a full-range design automation system for micro-processors [3]. They translate the abstract specification of an instruction set architecture (ISA), written in a subset of Prolog, into pipelined register-transfer level designs consisting of data paths and control paths. Piper also generates an interface to the compiler back-end (reorderer), and measures its time and space complexities. The benchmark characteristics which is obtained through simulation analysis is applied to evaluate the quality of designs.

Figure 1 illustrates the conceptual structure of the Piper system. Fiper serves as the front end of Piper. It translates the ISA specification in Prolog into a sequential abstract finite state machine.

Piper takes the output of Fiper and performs the following tasks: (1). pipeline scheduling; (2) pipeline hazard resolution; (3). resource allocation.

The first phase, pipeline scheduling, assigns micro-operations into pipeline stages. Pipeline hazards may be introduced by the pipeline scheduler in a highly pipelining case. These hazards are resolved in the second phase by a combination of hardware and software resolution strategies. This is accomplished in two steps: analysis of inter-instruction dependencies, and application of resolution strategies. In this phase Piper generates a reorder table consisting of reordering constraints which instruct the compiler back-end (reorderer) to properly organize the codes for the pipelined machine synthesized. At the last phase, the hardware resources are allocated, producing a pipelined RTL level design.

Design decisions made by the pipeline scheduling and hazard resolution affect the performance and cost of hardware and the time/space complexities of the compiler back-end. Therefore, in addition to the design engine, there are a set of estimators and application benchmarks for the estimation of those effects.

4. The analysis of pipeline hazards

RPHs are caused by inter-instruction dependencies. To resolve a hazard, we have to determine the type of dependency it involves and choose an appropriate resolution strategy. We first provide a taxonomy of inter-instruction dependencies which consists of nine types, derived from the cross products of <forward/backward/stationary> and <data/anti-data/output>.

For simplicity, we will assume a pipelined machine with stage time of one cycle throughout this section. With this assumption, a micro-operation of an instruction executed at its C'th cycle belongs to the C'th stage of the pipeline. The generalized case of stage time being multiple cycles will be discussed in Section 5.

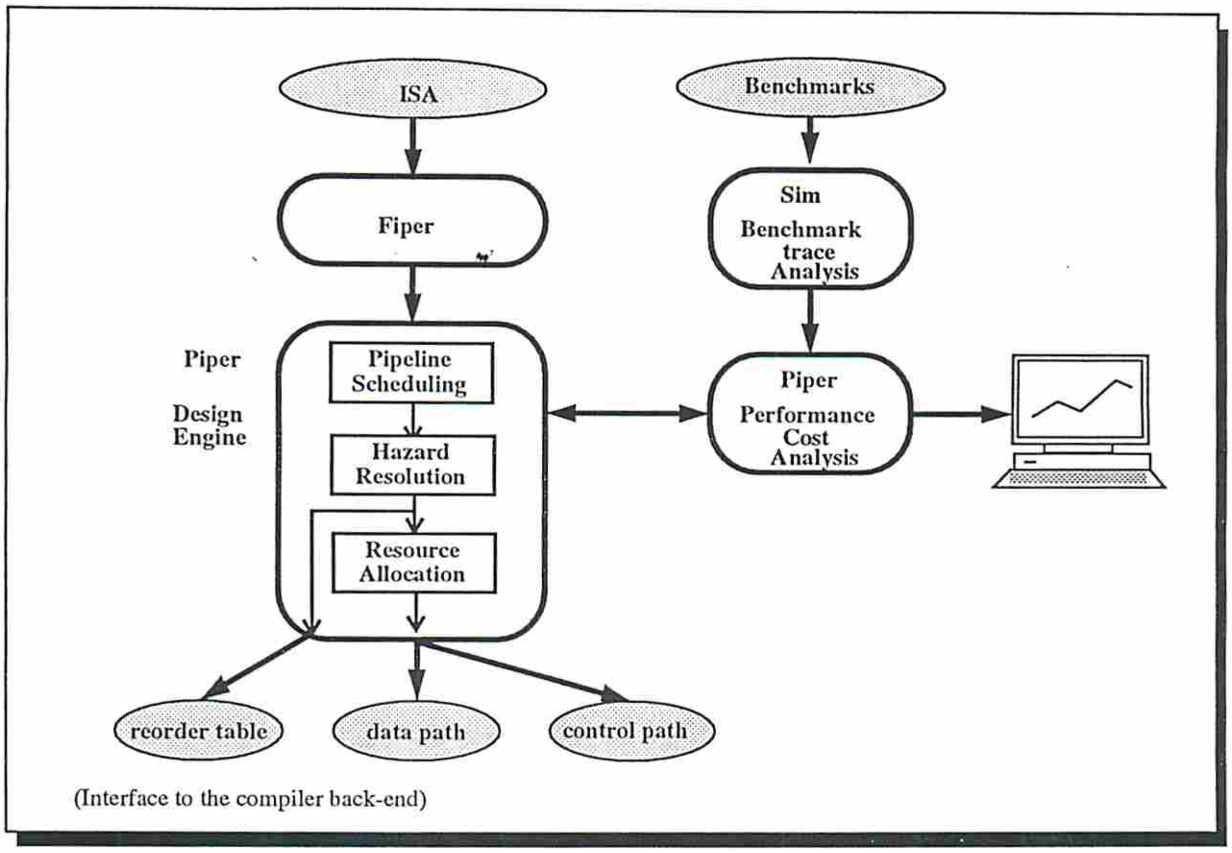

Figure 1 The structure of Piper system

4.1 An extended taxonomy of inter-instruction dependencies in pipelined ISPs

An inter-instruction dependency in a pipeline structure can be described in terms of the ternary tuple (P, R_p, R_s). R_p and R_s are the register access patterns (read/write) of the preceding and succeeding instruction, respectively. The conventional taxonomy of dependencies is encapsulated by this pair of parameters: data (R_p =write, R_s =read; or, read after write), anti-data (R_p =read, R_s =write; or, write after read), and output (R_p =write, R_s =write; or, write after write) dependency. On the other hand, P describes the relative position of register accesses in a pipeline structure of a dependent instruction pair. The possible values of P are: forward, backward, and stationary. Thus P provides a classification of dependencies from a pipeline structure's point of view. Figure 2 shows the relative positions for the register accesses of preceding and succeeding instructions. Suppose the instruction `instA` accesses register x at C_a'th cycle (at C_a'th stage), and the instruction `instB` accesses register x at C_b'th cycle (at C_b'th stage), with $C_a < C_b$. Now we are ready define the first two classes of dependencies with respect to C_a and C_b and the precedence of instruction pairs: forward and backward dependency.

A *forward dependency* happens when a preceding instruction accesses a register at an earlier stage and a succeeding instruction accesses the same register at a latter stage such as the `instA-instB` pair (`instA` followed by `instB`); a *backward dependency* happens when a preceding instruction accesses a register at a latter stage and a succeeding instruction accesses the same register at an earlier stage such as the `instB-instA` pair (`instB` followed by `instA`). Note that both forward and backward dependencies potentially co-exist in the hardware for any pair of

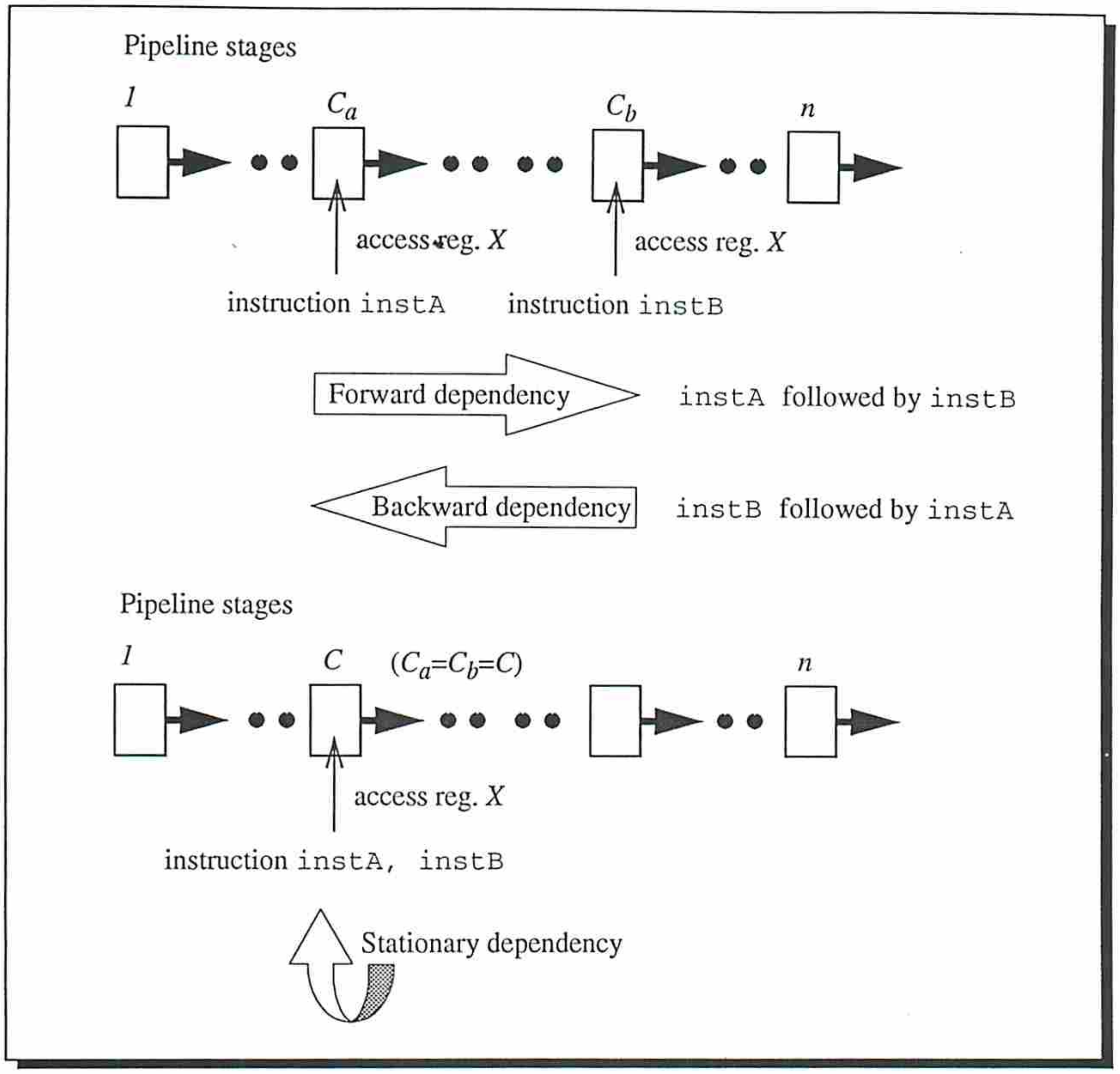

Figure 2 Forward, backward and stationary dependencies and pipeline stages

instructions which access the same registers at different cycles (except the read-read case). The actual direction of the dependency (forward or backward) in an application program is determined by the relative precedence relationship of the instruction pair; for example, the `instA-instB` pair has a forward dependency and the `instB-instA` pair has a backward dependency.

The third class of dependency is *stationary dependency* which happens when the preceding and succeeding instructions access the same register at the same cycle ($C_a = C_b$).

After defining the forward, backward and stationary dependencies, now we are ready to further refine dependencies into nine types. By applying the conventional taxonomy to each class of dependency, we have forward data, forward anti-data, forward output, backward data, backward anti-data, backward output, stationary data, stationary anti-data, and stationary output dependency. Table 1 summarizes these nine types of dependencies.

The preceding instruction access register x at cycle C_p The succeeding instruction access register x at cycle C_s	Types of register accesses		
	Read after Write (R_p=write, R_s=read)	Write after Read (R_p=read, R_s=write)	Write after Write (R_p=write, R_s=write)
$C_p < C_s$ (P=forward)	forward data dependency	forward anti-data dependency	forward output dependency
$C_p > C_s$ (P=backward)	backward data dependency	backward anti-data dependency	backward output dependency
$C_p = C_s$ (P=stationary)	stationary data dependency	stationary anti-data dependency	stationary output dependency

Table 1 Inter-instruction dependencies for pipelined instruction set processors

4.2 Pipeline hazards and inter-instruction dependencies

Not all classes of dependencies cause pipeline hazards. A backward dependency definitely causes a pipeline hazard because when the succeeding instruction reaches the stage where it accesses a register, the preceding instruction hasn't arrived at the stage (a latter stage) where it accesses the same register (as the `instB-instA` pair in Figure 2). For example, a 'delayed load' instruction immediately followed by another instruction which use the loaded datum is a pipeline hazard which involves a backward dependency.

A forward dependency does not cause any pipeline hazard. The `instA-instB` pair in Figure 2 is an example of a forward dependency. Instruction `instA` accesses register x at stage C_a, leaving enough time ($C_b - C_a + 1$ cycles) for the succeeding instruction `instB` to reach stage C_b to access X. However, even though forward dependency does not cause any pipeline hazard, a proper handle of it may eliminate another backward dependency (for example, the `instB-instA` pair in Figure 2). We will further explain this issue in Section 5.

The stationary dependency is the most stand-along class of dependency. It does not cause any pipeline hazard, nor does it interfere with other classes of dependencies. The succeeding instruction can access the same register in the next cycle right after the preceding instruction's access. Instructions exhibiting stationary dependencies never access the same register simultaneously. There is no delay slot required for instruction pairs with stationary dependencies. Therefore, the stationary dependency is the most desired way of handling inter-instruction dependency in the pipeline synthesis. This transcribes to a design goal in the scheduling phase of pipeline synthesis which requires accesses to the same registers in different instructions be scheduled to the same cycle. However, this goal might not achievable with respect to other design constraints. For example, aligning register accesses to the same cycle may effectively lengthen the critical path. When a stationary dependency can not be preserved, forward and backward dependencies occur, which necessitate some forms of resolution to ensure the proper behavior.

In the following section, we will present the hardware and software resolutions for pipeline hazards caused by forward or backward dependencies.

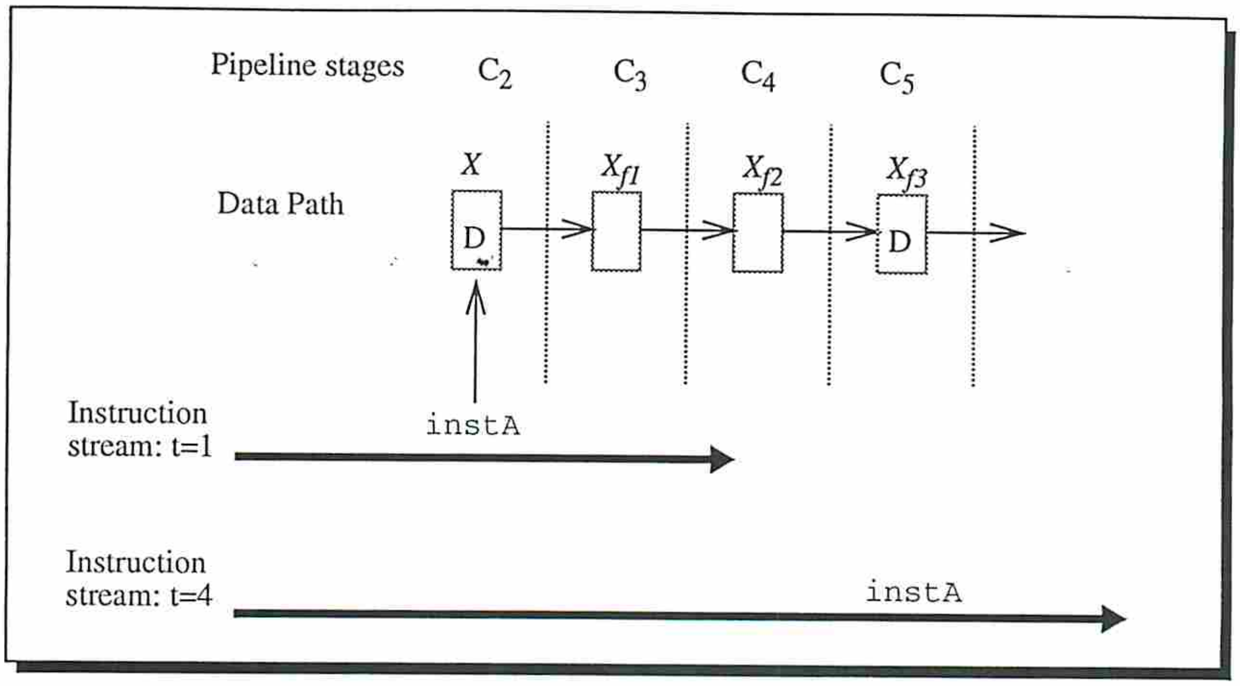

Figure 3 Hardware resolution: forwarding registers (X_{f1}, X_{f2}, X_{f3})

5. Hardware and software resolution strategies

We first overview the general hardware/software resolution strategies, and then associate them with various types of pipeline hazards.

5.1 General hardware and software resolution strategies

5.1.1 Hardware resolution

For some of register-related pipeline hazards, the most straightforward way to resolve them in hardware is to use additional registers. Two types of additional registers can be employed: forwarding and duplicate registers.

Forwarding registers carry the data along with the instruction stream in the pipeline. In Figure 3 a datum *D* is set to register *X* by instruction `instA` in stage C_2, and then is forwarded in the pipeline via forwarding registers. The datum moves along the pipeline synchronously with `instA`. The advantage of forwarding is that as soon as the current datum of register *X* is forwarded to next stage, *X* is free for next datum. This is analogous to adding extra latches (delays) in pipeline synthesis for DSP applications.

Duplicate registers release the burden of temporary registers. In Figure 4 (a) a temporary register *T* connects two sources S_1 and S_2 and two destinations D_1 and D_2. There are four possible connection patterns. All connections are mutually exclusive, with *Y* being the bottleneck of the data traffic. Suppose that the real connections to be established by *T* are S_1->D_1 and S_2->D_2, and better performance will be achieved when these two connections can be made concurrently. Then

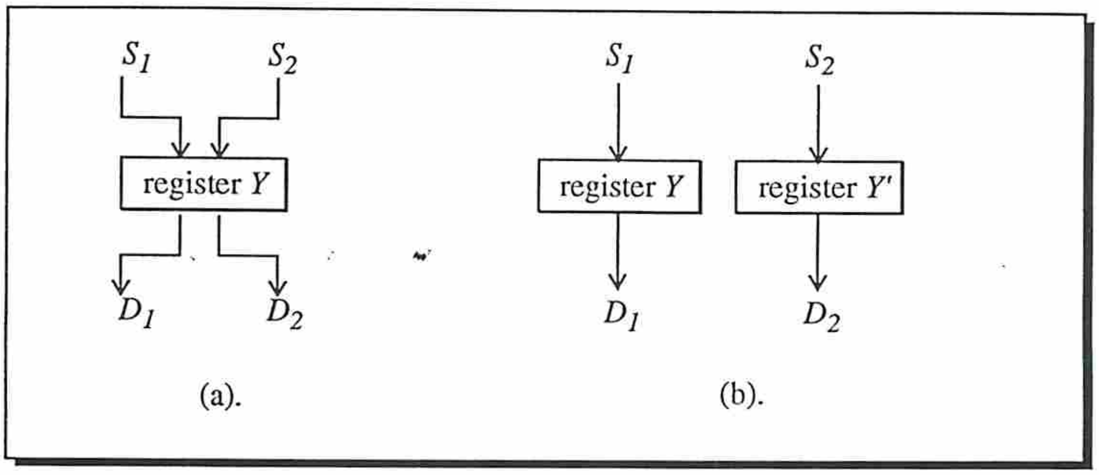

Figure 4 Hardware resolution: duplicate registers

adding a duplicate register Y' to create two data paths will ease the traffic and improve the performance as shown in Figure 4 (b).

5.1.2 Software resolution

The major technique used in software (compiler back-end) to resolve pipeline hazards is instruction reordering. As described in Section 1, the desired behavior of an instruction stream may be distorted due to the change in the relative timing of micro-operations, caused by pipelining. *Instruction reordering* restores the desired sequential semantics of an instruction stream by reordering the sequence of instructions.

There are two directions in reordering: up and down reordering. In Figure 5 (a) is a piece of sequential codes where instruction `instB` follows and depends on `instA`. The cases (b) and (c) of Figure 5 are the reordered codes for some pipeline structures. In case (b) the instruction `instB` is moved up and ahead of `instA`, whereas in case (c) `instB` is moved down and apart from `instA`. There are usually constraints (windows) about these movements: W_{up} being the maximal numbers of slots `instB` can be moved ahead of `instA` and W_{down} being the minimal number of slots `instB` has to be moved down from `instA`. For the example in Figure 5, W_{up} and W_{down} are three and two, respectively.

5.2 Inter-instruction dependencies and their applicable resolutions

In this subsection we will provide applicable hardware and/or software resolutions to pipeline hazards caused by forward or backward dependencies. For ease of discussion, we will use the same pipeline architecture and instruction pairs in Figure 2 as an example, assuming stage time of one cycle. The generalized case where a pipeline stage takes multiple cycles will be provided in Section 5.2.3.

5.2.1 Forward dependencies

We first discuss forward dependencies. A forward dependency is the case of `instA-instB` pair (`instA` followed by `instB`) (Figure 2). As mentioned in Section 4.2, a forward dependency

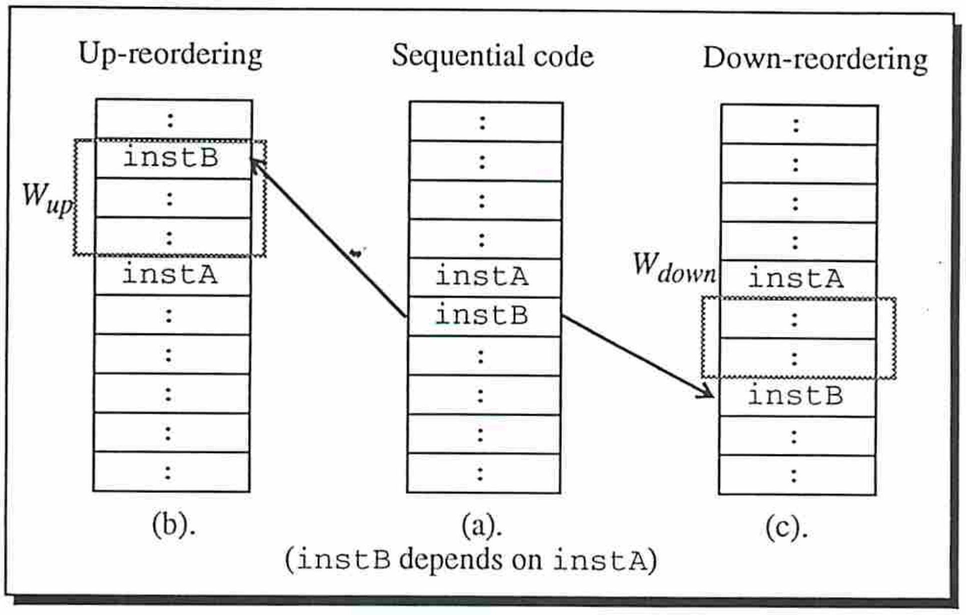

Figure 5 Software resolution: instruction reordering

does not cause pipeline hazards; however, some resolutions can be optionally applied to improve the system performance.

Forward data dependency

The forward data dependency is the case where `instA` writes to register *X* and `instB` reads *X* as shown in Figure 6. There are two ways to resolve this type of dependency: forwarding registers or up-reordering.

First, one forward register per stage can be allocated to stages C_a to C_{b-1} (total of C_b-C_a-1 forward registers) (Figure 6 (b)). A side effect of this approach is that the related backward anti-data dependency (`instB-instA` pair) is automatically resolved. This approach is preferable in the case where both `instA-instB` and `instB-instA` pairs happen very frequently in the application programs, at the cost of additional registers in the data path.

Secondly, instead of hardware resolution, one can choose to optionally move instruction `instB` ahead of `instA` (up-reordering) at most C_b-C_a-1 slots (cycles) as shown in Figure 6 (c). This has the advantage of hiding the possible delay slots associated with the instruction `instB` (for example, `instB` is a ‘delayed load’ instruction), at the cost of longer compilation time in the compiler back-end.

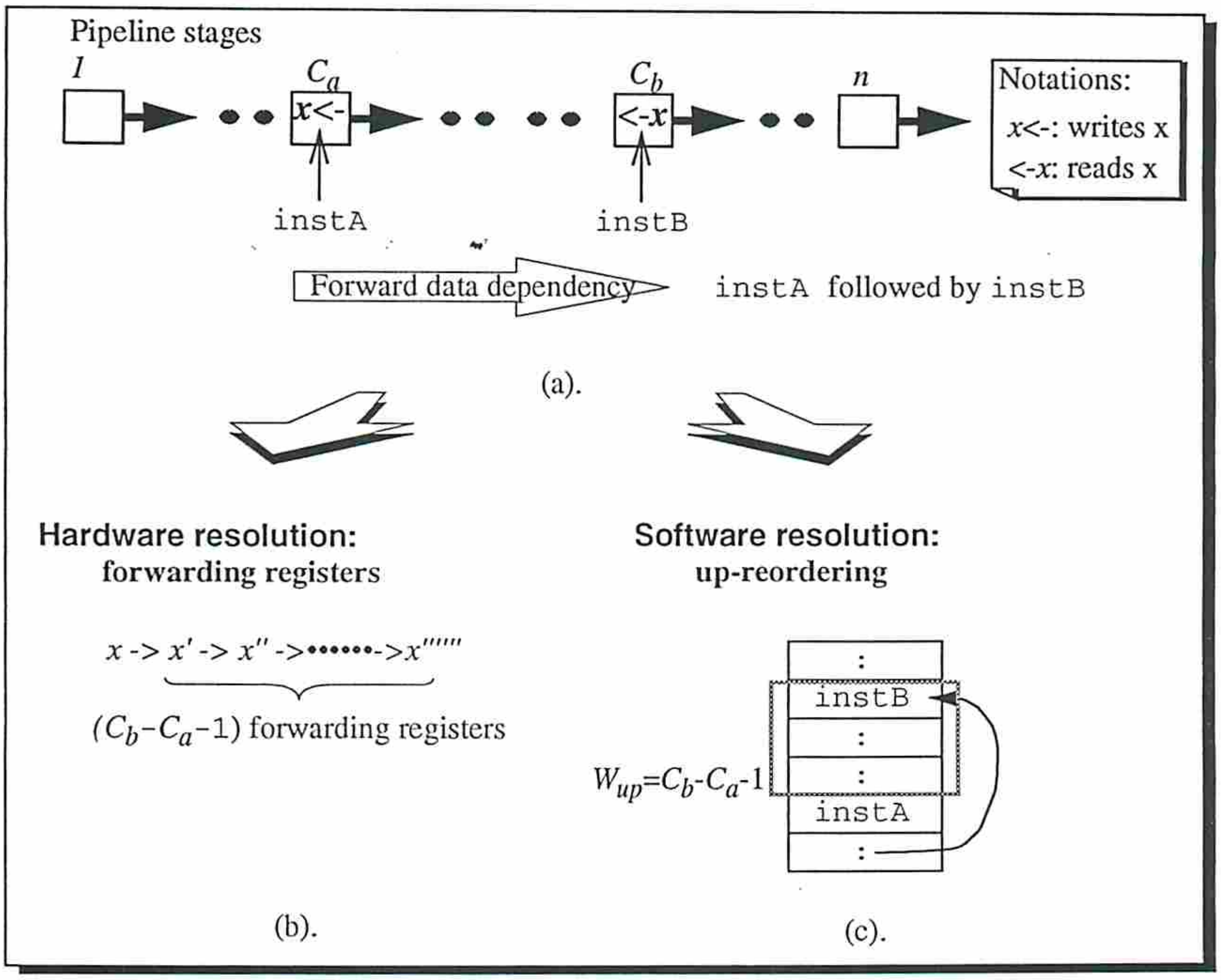

Figure 6 Forward data dependency and its resolutions

Forward anti-data dependency

The forward anti-data dependency is the case where `instA` reads register X and `instB` writes to X as shown in Figure 7. The applicable resolution is the up-reordering in the compiler back-end as described above.

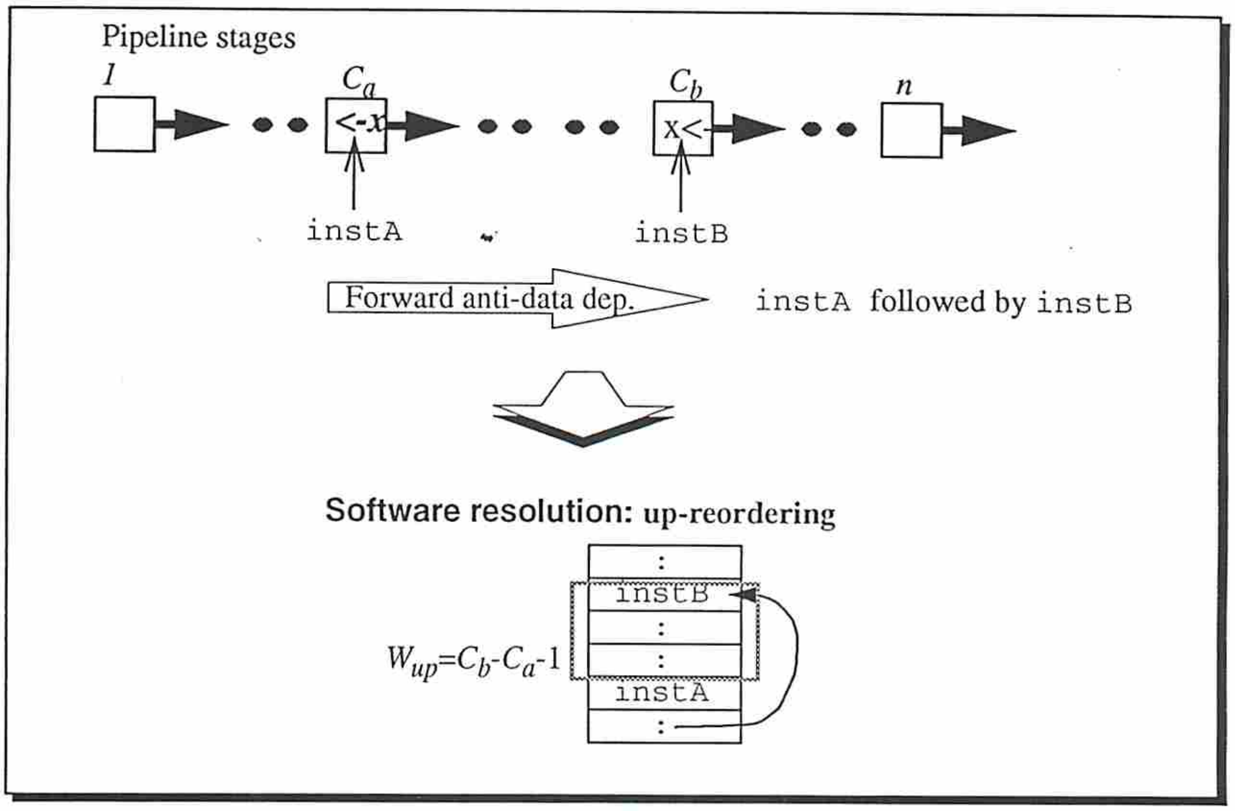

Figure 7 Forward anti-data dependency and its resolution

Forward output dependency

The forward output dependency is the case where both `instA` and `instB` write to register *X* as shown in Figure 8. There are two ways of resolving this type of dependency: duplicate registers or up-reordering.

Duplicate registers eliminate the forward output dependency such that `instA` and `instB` become independent instructions. The side effect is that the backward output dependency (`instB-instA` pair) is also eliminated as well. However, this hardware resolution can be applied only when there is no circular dependency with respect to the register involved.

Figure 9 shows an example of a circular dependency. The instruction pair `instA-instC` exhibits forward output dependency with respect to register *X*. However, the dependencies with respect to *X* exhibited by other instruction pairs `instB-instC` and `instC-instA` close a dependency loop `instA-instB-instC-instA` such that the attempt to allocate a duplicate register *X'* to `instA` will require `instB` to access *X'* which will in turn require `instC` to access *X'*, resulting in that register *X* becomes obsolete and a new forward output dependency (with respect to *X'*) is created for the `instA-instC` pair which invalidates the previous effort to eliminate the forward output dependency (with respect to *X*) for the `instA-instC` pair.

The circular dependency checking can be applied in two scopes. It can be applied against the pipeline structure as in the example in Figure 9. This is a conservative approach. On the other

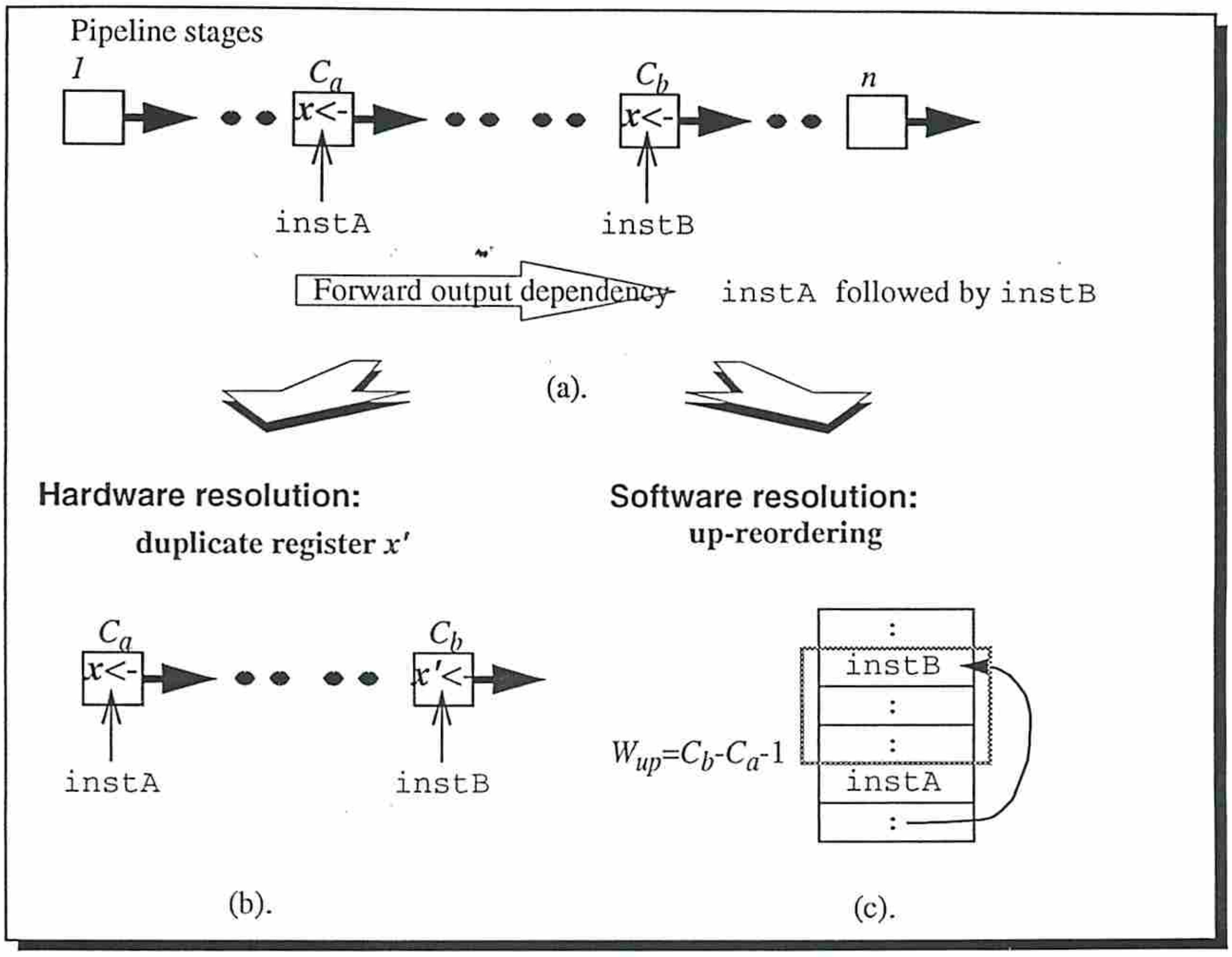

Figure 8 Forward output dependency and its resolutions

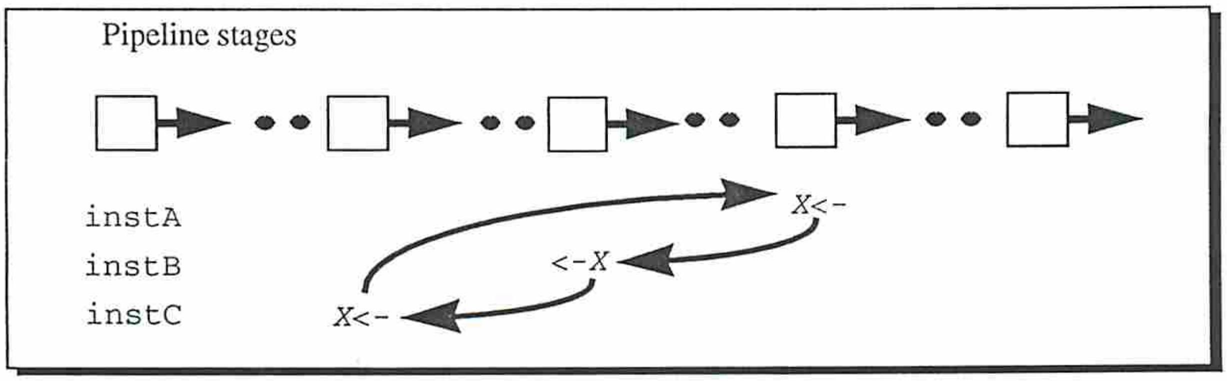

Figure 9 Circular dependency

hand, in an application specific environment, the circular dependency check can be applied against the application programs since some circular dependency exists in the pipeline structure might not exist in the applications. For example, if the `instA-instB` dependency never exists in the application programs, then the dependency loop is broken, which validates the insertion of duplicate register X' in an attempt to eliminate the `instA-instC` dependency.

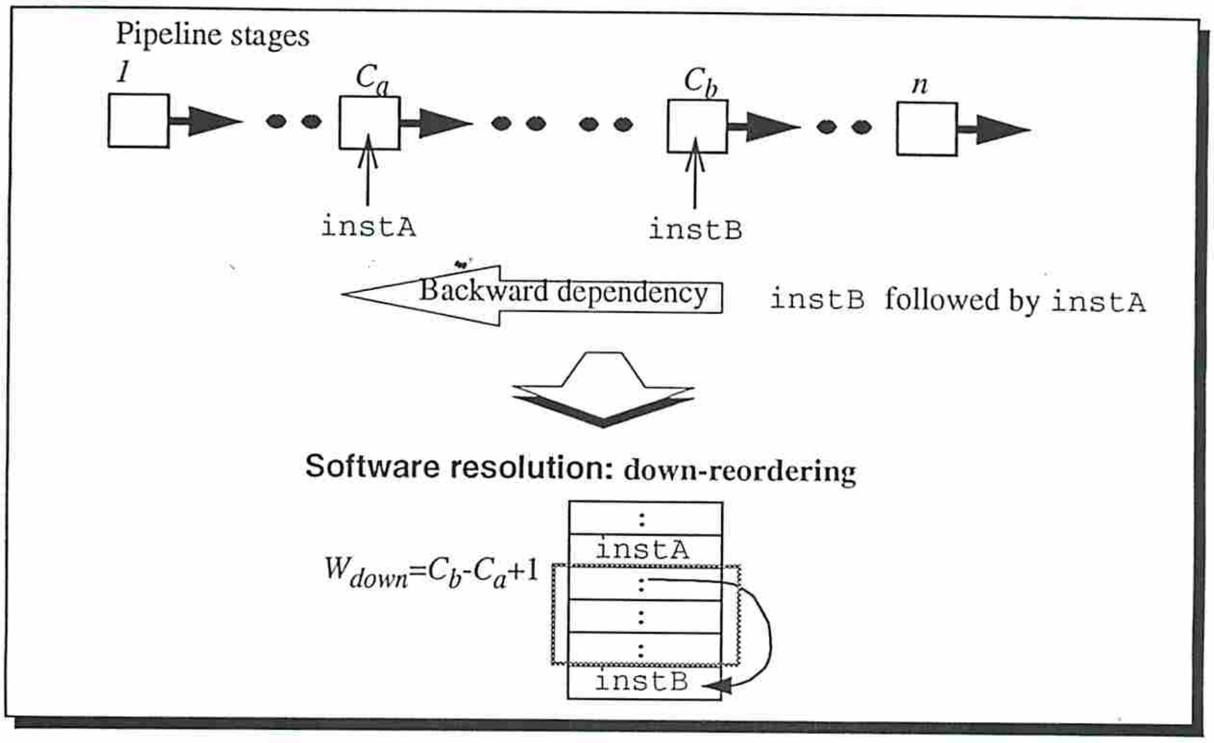

Figure 10 Backward dependencies and their resolution

Forward output dependencies can also be resolved by the optional up-reordering in the compiler back-end as described previously.

5.2.2 Backward dependencies

A backward dependency is the case of `instB-instA` pair (`instB` followed by `instA`). There is a software resolution (down-reordering) available for all backward dependencies. The dependent instruction `instA` has to be moved away, downward from its predecessor `instB` for at least C_b-C_a+1 slots (W_{down}=C_b-C_a+1) to ensure that `instA` access the target register after `instB`'s access (Figure 10). The compiler back-end can fill in these slots with `NO-OP`s or reorder some other independent instructions into these slots.

The backward output dependency can be also resolved in hardware with the same technique, duplicate registers, as in the case of forward output dependency (Figure 10 (b)). Circular dependency check has to be performed before a duplicate register can be inserted. Unfortunately, there is no hardware resolution for backward data/anti-data dependencies.

5.2.3 Summary and extension

Here we summarize the resolution strategies for inter-instruction dependencies in Table 2. In this table we extend the resolution strategies for pipeline machines with multi-cycle stage time: every instruction spends *S* cycles in a stage before it advances to next stage. Therefore, a micro-operation executed at the *C'th* cycle of its execution path will be executed at the `mod(`*C/S*`)+1`'th cycle of the $\lceil C/S \rceil$'th stage. In this table it is assumed that `instA` and `instB` access register *X* at the C_a'th and C_b'th cycles of their execution paths, respectively. Please note that the constant M in the rows of anti-data dependencies is used to adjust for the case of master-slaved registers.

Inter-instruction dependency	Hardware resolution	Software resolution
Forward data (instA-instB)	• Forward registers total number of forward registers: $\lceil (C_b - C_a)/S \rceil - 1$	• Side effect of h/w resolution: backward anti-data dependency (instB-instA) is resolved
	• N/A	• Optional up-reordering: $W_{up} = \lceil (C_b - C_a)/S \rceil - 1$
Forward anti-data (instA-instB)	• N/A	• Optional up-reordering $W_{up} = \lceil (C_b - C_a + M^{*})/S \rceil - 1$
Forward output (instA-instB)	• Duplicate register	• Side effect of h/w resolution: backward output dependency (instB-instA) is resolved
	• N/A	• Optional up-reordering: $W_{up} = \lceil (C_b - C_a)/S \rceil - 1$
Backward data (instB-instA)	• N/A	• Down-reordering $W_{down} = \lfloor (C_b - C_a)/S \rfloor + 1$
Backward anti-data (instB-instA)	• N/A	• Down-reordering $W_{down} = \lfloor (C_b - C_a - M)/S \rfloor + 1$
Backward output (instB-instA)	• Duplicate register	• Side effect of h/w resolution: forward output dependency (instA-instB) is resolved
	• N/A	• Down-reordering $W_{down} = \lfloor (C_b - C_a)/S \rfloor + 1$

Table 2 Hardware/Software resolution strategies for inter-instruction dependencies

*. M= 1 for master-slaved registers; M=0 otherwise.

6. Consideration of hardware/software tradeoffs

The hardware and software resolutions provide different performance/cost tradeoffs. In an application specific environment, the tradeoff can be tuned towards the characteristics of the application benchmarks.

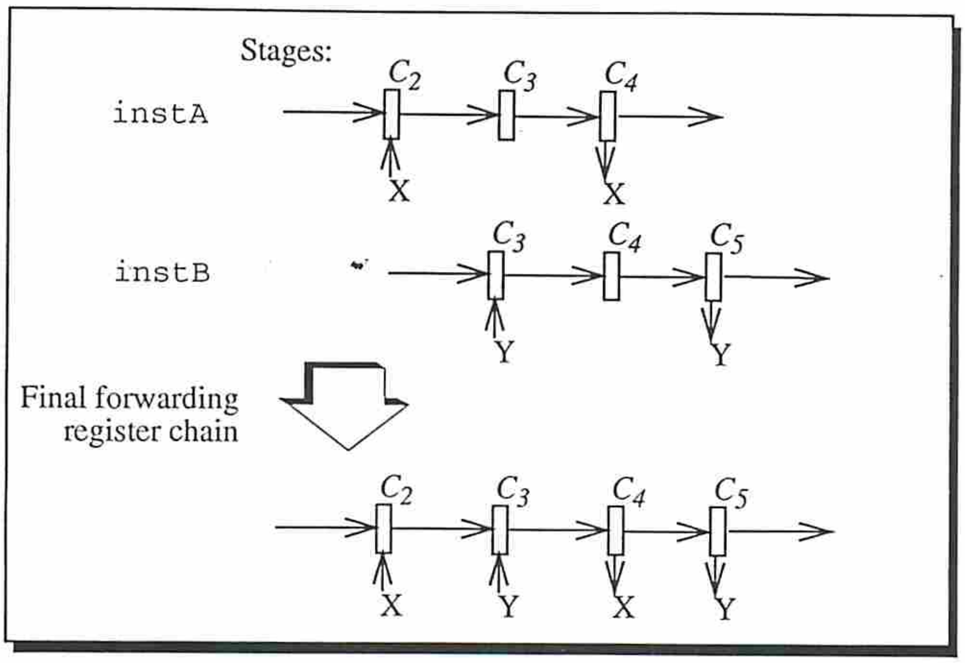

Figure 11 Sharing forwarding registers

6.1 Performance and cost with respect to hardware

A forwarding register in a stage can be shared by all forward data-dependent instruction pairs whose forward data dependencies span across this stage, regardless of the target registers they depend on. For example, suppose the forward data-dependent instruction pairs `instA-instB` (depending on register *X*) and `instC-instD` (depending on register *Y*) require forwarding registers from stage C_2 to stage C_4 and from C_3 to C_5, respectively, as shown in Figure 11. Then the final data path would have a total of four forwarding registers, with forwarding registers in stages C_3 and C_4 shared by the *X* and *Y* forwarding paths. Therefore, the utilization of a forwarding register of a stage is the summation of frequencies of the forward data dependencies in the benchmarks which span across the same stage. In this sense, forwarding registers are more desired in the data path for application benchmarks exhibiting larger amount of forward data dependencies across the same pipeline stages.

Forwarding registers improve the performance in that they eliminate pipeline hazards caused by backward anti-data dependencies with respect to the same instruction pairs. The compiler back-end also benefits from them since less amount of reordering has to be done. On the other hand, the existence of forwarding registers prevents the up-reordering of the dependent instructions such as `instB` and `instD` in the previous example. This may be an undesirable result if `instB` and `instD` are instructions of 'delayed load' type (involving a backward data dependency with a instruction following them) since they will degrade the performance unless they are scheduled earlier. Therefore, the decision of whether to employ forwarding registers depends heavily on the relative frequencies of dependent instruction pairs in the application benchmarks.

The use of duplicate registers is more restricted, compared with the use of forwarding registers. A duplicate register can not be shared unless it is for writing to the same register at the same stage. The performance advantage of duplicate registers is that they remove both forward and backward output dependencies. Therefore, duplicate registers are feasible only when there is a high frequency of output dependent instruction pairs in benchmarks and the two register writes are separated by many pipeline stages (large performance penalty). However, duplicate registers may change the instruction semantics if they are applied to architected registers instead of temporary registers.

6.2 Performance and cost with respect to software

The compilation complexity of the compiler back-end for the pipelined ISP generated depends on the number of up/down reordering constraints, as well as the reorder distances (W_{up} and W_{down}). Generally speaking, the larger number of reordering constraints and longer reorder distances, the more time the compiler back-end spends to compile the codes. We model this relationship by asking the compiler writer to express the time/space complexities in terms of those factors. For example, the time complexity of some compiler back-end may be expressed as $L \bullet E \bullet D_{max}^{2}$ where L is the average basic block size, E is the number of down-reordering constraints, and D_{max} is the maximal reorder distance. Now we can compare the complexity of the compiler back-end for various pipelined implementations by substituting their associated L, E, and D_{max} values into the expression.

The code expansion (static/dynamic) due to the inserted `NO-OPs` by the compiler back-end can be approximated through the analysis of application benchmarks. The characteristics derived from the analysis includes instruction pair frequencies and instruction level parallelism. For example, suppose that the dynamic frequency of instruction pair `instA-instB` is 10%, the down-reordering constraint requires that `instA` and `instB` be separated by at least five slots (W_{down}=5), and the instruction level parallelism is two, then the average code expansion due to the inserted `NO-OPs` between `instA` and `instB` will be roughly 30% of the original dynamic code size((5-2)•10%).

Finally, for each pipelined processor p, the run time performance with respect to a benchmark can be approximated by the following equations:

$$Exp = \sum_{j} Dist_{p_j} \cdot (1 - C_j) \cdot R_j \cdot A + A$$

$$S_p = \frac{A \cdot M}{L_p \cdot Exp}$$

$$T_p = \frac{A}{L_p \cdot Exp}$$

where E_{xp} is the dynamic code expansion due to the insertion of `NO-OP`*s*, S_p is the speedup of pipelined processor p with respect to non-pipelined processor, and T_p is the throughput of pipelined processor p.

$Dist_{pj}$ is the number of `NO-OPs` (in the worst case) to be inserted between a consecutive instruction pattern j for pipelined processor p, R_j is frequency of the consecutive instruction pattern j, L_p is the pipeline latency of pipelined processor p, M is the instruction cycle time in a non-pipelined processor, A is the total number of instructions in the simulation trace, and C_j is the compression factor for consecutive instruction pattern j which means that, empirically, the percentage of `NO-OPs` which can be deleted by reordering instructions. C_j can be empirically approximated by *(instruction level parallelism* -1*)/Distpj*, and can be set to zero for the worst case analysis.

The product *AM* is the total number of cycles when the benchmark is executed on the non-pipelined processor while the product L_pE_{xp} is the total number of cycles executed on the pipelined processor *p*.

For more details please refer to [1] and [2].

7. Example

In this section we illustrate these resolution techniques with a synthesis example. *SM1* is an accumulator-based instruction set processor with nine instructions including ‘add’, ‘and’, ‘sub’, ‘shr’ (shift right), ‘store’, ‘load’, ‘jump’, ‘brn’ (branch on negative) and ‘nop’. For its instruction set specification, please refer to [3]. This machine has a critical path of six clock cycles, and a minimal achievable latency of five. Piper returned six possible pipeline latencies ranging from one to six. The pipeline latency of six is a sequential implementation. Any pipelined implementation with a pipeline latency less than five would have some pipeline hazards. The highly pipelined one has six stages and fetches an instruction every cycle. This implementation has 178 inter-instruction dependencies. Therefore, the maximal performance speedup (w.r.t. non-pipelined implementation) of this implementation is six, but the average speedup of an application will be degraded due to pipeline hazards. We will focus how the resolution techniques are applied to pipeline hazards of this six-stage machine.

Figure 12 shows the pipeline schedule of *SM1* (the input to the pipeline hazard resolution phase of Piper): a finite state representation of the machine and the pipeline stages. Only states related to instruction fetch and instructions ‘add’, ‘store’, ‘load’ and ‘brn’ are shown here. Bubbles represent states. The state in stage three is the decode state. States in stages four, five and six are conditionally executed, according to the opcode. The contents of bubbles are register accesses we are interested in this discussion. Empty bubbles contain RTLs which are not of interest here. The thick bi-directional arcs are forward/backward inter-instruction dependencies associated with those register accesses. These arcs are labelled as `PC-1`, `MAR-1`, `MAR-2`, etc. There are fourteen dependencies in the figure such as the forward anti-data dependency of `PC`, backward data dependency of `PC` (shown as the bi-directional arc `PC-1`), etc.

Figure 13 shows the instruction pair analysis for the benchmark `concat` which we will use to evaluate the various resolution strategies. The first field is the instruction pair: [preceding instruction, succeeding instruction]. The second field is the frequency with which the associated instruction pair exists in the dynamic execution trace. The average instruction level parallelism for `concat` is about one since most of the instructions access the accumulator. This observation implies that hardware resolutions may be preferable over software resolutions since there will be very few independent instructions which can be reordered into the `NO-OP` slots.

We will use the analytical models developed in Section 6.2 to evaluate the effective speedups (w.r.t. the non-pipelined design) and relative time complexities of the reorder (compiler back-end) for pipelined designs synthesized with various resolution strategies.

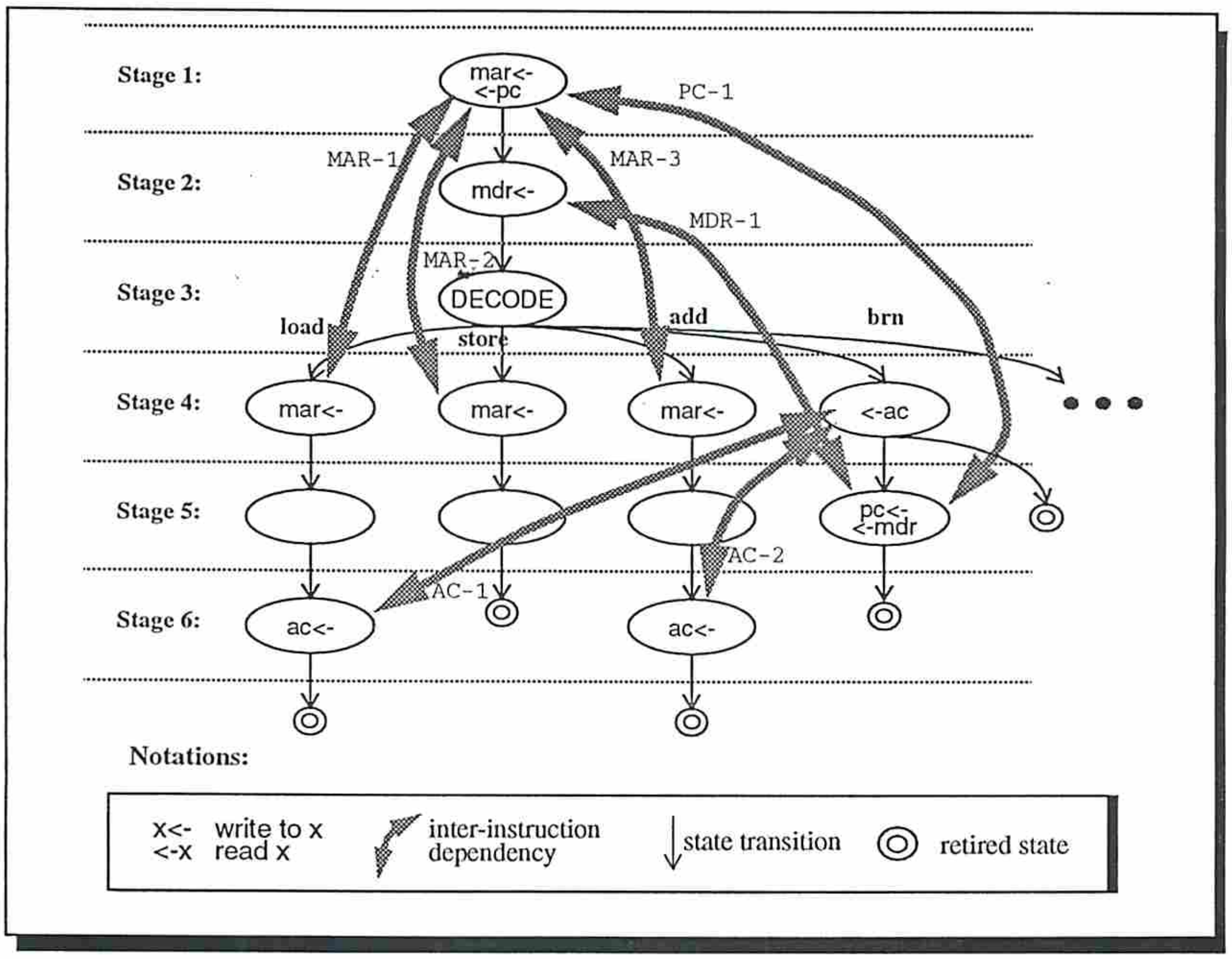

Figure 12 Pipeline stages and state transitions for instructions: load, store, add, brn

instruction_pattern_count([add,and],0.03).	instruction_pattern_count([load,add],0.14).
instruction_pattern_count([add,brn],0.03).	instruction_pattern_count([load,and],0.02).
instruction_pattern_count([add,store],0.13).	instruction_pattern_count([load,load],0.02).
instruction_pattern_count([and,store],0.06).	instruction_pattern_count([load,store],0.21).
instruction_pattern_count([brn,jump],0.01).	instruction_pattern_count([store,add],0.02).
instruction_pattern_count([brn,load],0.02).	instruction_pattern_count([store,jump],0.03).
instruction_pattern_count([jump,load],0.04).	instruction_pattern_count([store,load],0.24).

Figure 13 Instruction pattern analysis for a benchmark `concat`

Now let's investigate the resolutions to these dependencies. First, let's consider all-software resolution. Piper found sixteen up-reordering and down-reordering constraints, respectively. In Table 3 lists the reorder constraints derived from the dependencies `PC-1`, `AC-1`, and `MDR-1`. Note that some of the reorder constraints can be covered by others because a dependent instruction pair may contain more than one inter-instruction dependency. For example, reorder constraints derived for the instruction pair (*allInstructions, brn*) from dependency `MDR-1` can be covered by the ones derived from PC-1. The performance and cost of the design generated with all-software resolution is listed in Table 4 under column 'Design 1'.

Dependent pair f=forward, b=backward	Reorder direction	Preceding instruction	Succeeding instruction	Reorder distance (W_{up}, W_{down})
PC-1:f	up	all instructions	brn	4
PC-1:b	down	brn	all instructions	5
AC-1:f	up	brn	load	2
AC-1:b	down	load	brn	2
MDR-1:f	up	all instructions	brn	2
MDR-1:b	down	brn	all instructions	2

Table 3 Some reorder constraints for six-staged SM1

We can add a duplicate register for mar to resolve its output dependencies between stage one and three which involves instructions 'add', 'and', 'load' and 'store'. These patterns exist in our benchmark with a total frequency of 92%. This hardware resolution removes four up/down reordering constraints, respectively, and improves the performance by 87% (1.43 to 2.67 in speedup) and reduces the reordering complexity by 19% (1.31 to 1.06 in relative time complexity). This is shown in column 'Design 2' of Table 4. The speedup improvement is very significant since for every 'add', 'and', 'load', and 'store' in the execution trace, one 'nop' has to be inserted to avoid the output conflict of mar, if it were not duplicated. (mar is a special register. A duplicate of it requires a two-port memory support).

A further hardware resolution involves the forward data dependency MDR-1. The reorder constraints by dependency MDR-1 require that two 'nop' be inserted after 'brn'. A forwarding register chain consisting of two registers can be allocated in stage three and four such that 'brn' can carry its own copy of mdr along the pipeline until it reaches stage five where it accesses mdr, leaving stage two immediately available for next instruction. This strategy allows other instructions to be reordered into the delay slots of 'brn', which further improves the speedup to 2.84 and reduces the relative time complexity of reorderer to 1, as shown in column 'Design 3' of Table 4.

One observation about the reorder constraints for various designs of *SM1* is that the maximal reorder distance remains as five while the number of constraints decreases as more hardware resources are invested. This is because that the maximal reorder distance in this example is derived from the backward data dependency PC-1 between stage one and five which does not have applicable hardware resolution. Therefore, the decrease in the time complexity of the compiler back-end benefits solely from the decrease in the number of constraints as hardware increases.

In summary, we have demonstrated the RPH resolution of *SM1* with strategies ranging from all software to some combinations of software and hardware. We have shown the estimated cost and performance for each possible design. Designers can select one appropriate design based upon the application domain and the design goal.

	Design 1	Design 2	Design 3
Registers (h/w resolution) <type, target register, number> `dg` = duplicate register `fg` = forward register		<dg,mar,1>	<dg,mar,1> <fg,mdr,2>
Reorder constraint (s/w resolution) <direction, # of constraint, maximal reorder distance> u=up, d=down	<d, 16, 5> <u, 16, 4>	<d, 12, 5> <u, 12, 4>	<d, 11, 5> <u, 11, 4>
Estimated speedup of the given benchmark (w.r.t. non-pipelined)	1.43	2.67	2.84
Relative time complexity of the reorderer (compiler back-end)	1.31	1.06	1

Table 4 Designs with various combinations of hardware/software resolutions

8. Concluding remarks

We have proposed an extended taxonomy of inter-instruction dependencies for the analysis of register-related pipeline hazards in instruction set processors, and then presented hardware and software resolutions for the hazards. These resolutions include forwarding/duplicate registers in hardware, and up/down instruction reordering in software (compiler back-end). The register-related pipeline hazards are resolved according to the types of the inter-instruction dependencies they involve. In an application specific environment, the combination of hardware and software resolutions can be tuned towards the characteristics of the application benchmarks. These techniques have been integrated into our pipeline synthesis system Piper for instruction set processors in two phases: analysis of inter-instruction dependencies, and application of resolution strategies. A pipelined micro-architecture and an interface (reorder table) to the compiler back-end are generated as the outputs of Piper.

Current limitations include: (1). we are not able to resolve structural pipeline hazards since Piper does not handle multi-cycle non-fully-pipelined functional units (only single-cycle and fully-pipelined functional units are supported). (2). All various pipelined designs for an instruction set specification assume the same clock cycle length which is not realistic in a real design and may distort the performance estimation. (3). Better dependency analysis and resolution strategies for instructions involving register files are yet to be developed since the actual register access patterns can not be determined from the instruction set specification. (4). Better designs can be obtained if the analysis of pipeline hazards is fed back to the pipeline scheduling phase. (5). Finer controls between hardware and software such as the architectural support for delayed branch with annulling slots are to be investigated. These limitations will be our focus in future work.

Reference

[1] Ing-Jer Huang and Alvin Despain, "High Level Synthesis of Pipelined Instruction Set Processors and Back-End Compilers," *Proc. of 29th DAC*, June, 1992

[2] Ing-Jer Huang "Piper: an Integrated Design Automation System for Instruction Set Processors," *USC Technical Report*, in preparation

[3] Iksoo Pyo, et al., "Application-Driven Design Automation for Microprocessor Design," *Proc. of 29th DAC*, June, 1992

[4] John L. Hennessy and David A. Patterson, "Computer Architecture A Quantitative Approach," *Morgan Kaufmann Publishers*, pp. 257-278, 1990

[5] Nohbyung Park and Alice Parker, "Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications," *Trans. on CAD, Vol. 7, No. 3*, March 1988

[6] Pierre B. Paulin and John P. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's," *IEEE Transactions on Computer-Aided Design, Vol. 8, No. 6,* June 1989

[7] Gert Goossens, Jan Rabaey, Joos Vandewalle and Hugo De Man, "An Efficient Microcode Compiler for Application Specific DSP Processors," *IEEE Transactions on Computer-Aided Design, Vol. 9, No. 9,* September 1990

[8] Cheng-Tsung Hwang et al., "Scheduling for Functional Pipelining and Loop Winding", *Proc. of the 28th Design Automation Conference,* 1991

[9] Mauricio Breternitz Jr. and John Paul Shen, "Architecture Synthesis of High-Performance Application-Specific Processors", *Proceedings Design Automation Conference,* 1990

[10] Masaharu Imai, Alauddin Alomary et al., "An Integer Programming Approach to Instruction Implementation Method Selection Problem," *Proc. of Euro-DAC*, 1992

[11] Peter M. Kogge, *The Architecture of Pipelined Computers*, MacGraw-Hill, 1981