# PIPELINE DESCRIPTIONS FOR RETARGETABLE COMPILERS: A Decoupled Approach

A

Proposal

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

---

*Christopher W. Milner*

*June 1998*

## Abstract

A good optimizing compiler must have detailed information about the target processor’s execution pipeline in order to generate and schedule code with high levels of instruction-level parallelism. Current state-of-the-art pipeline descriptions are tediously constructed on an instruction-by-instruction basis. These descriptions often fail to capture important instruction scheduling constraints so artificial resources are introduced to enforce these constraints. The result is a pipeline description that is difficult to maintain and reuse; retargeting the compiler means retargeting each instruction and rethinking the purpose of each artificial resource.

To address the above problems, the proposed research will develop a new, powerful approach for describing modern instruction pipelines by separating the pipeline description from the instruction set description. The proposed approach uses a graphical description of the pipeline and an accompanying annotation language to describe the relevant behavior of the machine’s execution pipeline. Using the descriptions of the pipeline and an existing description technique for instruction sets, it will be possible to generate instruction scheduler information automatically. Furthermore, this decoupling of the pipeline description from the instruction set description eases the burden of retargeting the compiler as new instruction set extensions and new pipeline implementations appear.

# PIPELINE DESCRIPTIONS FOR RETARGETABLE COMPILERS:
# A Decoupled Approach

## 1 Introduction

Modern processors use a variety of techniques, such as superpipelining and multiple instruction issue, to exploit instruction level parallelism. A good optimizing compiler for this type of processor must have an instruction scheduler to produce high performance code: studies have shown that even processors with dynamic scheduling perform better with scheduled codes [LPU95, DH96]. Currently, instruction schedulers are produced in one of two ways. Schedulers are either written in a systems programming language to enforce *ad hoc* rules, or automatically generated from a pipeline description. As scheduling rules are becoming more complex, compiler writers are abandoning the hand-coded approach in favor of generating scheduling information from a pipeline specification.

There are problems with using pipeline descriptions to generate instruction schedulers. First, the descriptions are written on an instruction-by-instruction basis. Modifying these descriptions is a tedious and error prone process but is required when a new implementation of a processor is introduced. Second, there are no clear rules about how to write these descriptions. This can lead to schedulers that permit overconstrained or underconstrained schedules.

At first blush, the process of writing descriptions is straightforward: each instruction is described in terms of the pipeline resources used and the specific cycles in which the resources are used. However, the pipeline descriptions that result from this exercise are rarely easy to decipher. Because all instructions or instruction classes must be specified, there is a tendency on the part of the compiler writer to use the sparest of descriptions. When revisiting the specification, it is often unclear why one resource was used when another was not. Also, when the architecture is modified, such as when a new pipeline stage is introduced, the specification for each instruction must be carefully rewritten to reflect the change in the underlying implementation.

Writing pipeline descriptions that result in correct schedulers is not a well delineated cookbook exercise. In practice, a first cut at describing a pipeline will result in an instruction scheduler that does not fully capture resource and ordering constraints. To remedy these shortcomings, the pipeline description is often augmented with artificial resources. Artificial resources come with their own set of problems. Artificial resources can make the description confusing as they do not correspond directly to resources in the pipeline. Also, there are no clear rules about how to introduce resources to add a desired constraint. The trial and error method of introducing resources is used to bring about changes in constraints enforced by the scheduler. There is always the danger of adding resources that introduce unintended constraints that are very subtle to detect.

To summarize, pipeline descriptions are tedious to write and maintain and they force the compiler writer to reason about resource interactions, rather than scheduling constraints. What is needed is new way to describe pipelines and a more direct way to express ordering constraints for instruction schedules.

The thesis of this research is that the description of execution pipeline is too closely interwoven with the instruction set description of the machine: this research will decouple the pipeline description from the instruction set description. The *annotated pipeline graph* is proposed to extend resource vector pipeline descriptions to address the aforementioned shortcomings. The pipeline graph descriptions will resemble pipeline diagrams in architecture manuals. In addition, an annotative language of regular expressions will be developed; these annotations will specify scheduling constraints not reflected in the pipeline graph. With this research we will develop the analysis and algorithms necessary to generate instruction scheduler components automatically from the graph diagram. This approach is designed to encourage intuitive descriptions of pipelines that are easy to write, easy to modify for purposes of experimentation and precise enough to model current implementations of processors.

## 2 Instruction Schedulers

The purpose of an instruction scheduler is to arrange a sequence of instructions in such a way as to minimize execution time while preserving program correctness. Obstacles to this purpose include the delay of instruction execution due to dependencies on data availability (data hazards), delays due to dependencies on instruction availability in the presence of branches (control hazards) and delays due to microarchitectural resource availability (structural hazards). The instruction scheduler generally calls upon a contention query module (CQM) to assist in determining the presence of structural, control and data hazards.

The CQM generally works in one of two ways. In the first method, the instruction scheduler calls the CQM to test each new instruction to be scheduled. The CQM looks back over a list of already scheduled instructions and checks for a conflict with the new instruction. This method usually relies upon hand-coded rules to maintain pipeline constraints. For instance, the *vpo* compiler contains a single hand-coded rule for the SPARC processor that prevents scheduling a conditional branch on the cycle immediately following a floating-point compare [BD94]. While this method is sufficient for simpler pipelines it consumes greater amounts of time as the pipeline constraints increase. Furthermore, this portion of the scheduler must be rewritten with each new implementation of the pipeline.

The alternative method of checking for conflicts involves modeling the actual resource usages of the instructions as they progress through the pipeline. When two or more instructions try to use the same resource in the same cycle a structural hazard is detected and the offending schedule must be changed. This type of pipeline model is usually generated automatically from a pipeline description.

Automatic generation of pipeline modeling tools is very appealing and widely used. As the proposed research will build upon this type of model, the next section is devoted to explaining pipeline descriptions, how scheduling components can be generated from them and the shortcomings of these components.

## 3 Pipeline Modeling via Resource Vectors

The pipeline model depends on each instruction or instruction class being represented by a resource vector. The resource vector indicates all microarchitectural resources used by the instruction and the cycle in which the resource is used. The pipeline itself is represented by a *reservation table* which is a vector composed of all resources used by instructions occupying the pipeline. As the cycles lapse the old instructions are shifted out of the reservation table and new instruction vectors are composed with the new pipeline configuration.

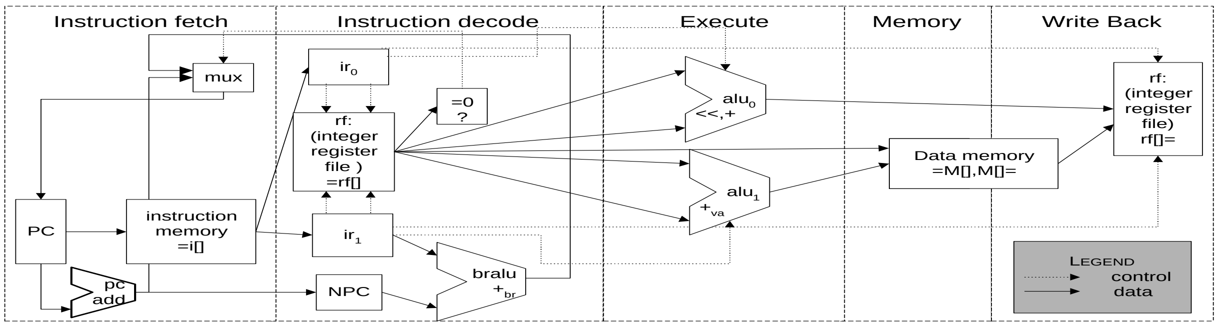


**FIGURE 1: Pipeline diagram for the PIPE processor**[a]

a. This figure shows control lines (dotted lines) as well as data lines. Part of the research will be to determine if the control lines are really needed.

Consider the simple five stage dual-issue processor, called PIPE, depicted in Figure 1. This type of pipeline diagram is representative of those found in manufacturer-supplied architecture manuals. The PIPE processor has two ALUs. $\mathsf{alu}_0$ is labeled with the operations it can perform, such as addition and shifting. $\mathsf{alu}_1$ is used exclusively for generating virtual addresses for loads and stores. This processor has two instruction registers labeled $\mathsf{ir}_0$ and $\mathsf{ir}_1$ to indicate that two instructions may be issued simultaneously. The control path from $\mathsf{ir}_0$ to $\mathsf{alu}_0$ indicates that all `add` and `sll` operations must be issued from the first instruction slot in a group of two instructions. Similarly, all loads, stores and branches must be issued from the second instruction slot. The diagram is labeled with the five pipeline stages: instruction fetch, instruction decode, execution, memory and write back. PIPE is a toy processor and has only five instructions, as shown in Table 1.

| **instruction** | **description** |
|---|---|
| add $r_x,r_y,r_z$ | # r[z] <- r[x] + r[y] |
| sll $r_x,r_y,r_z$ | # r[z] <- r[x] shift logical left r[y] |
| ld $r_x,r_y,r_z$ | # r[z] <- M[r[x]+r[y]] |
| st $r_x,r_y,r_z$ | # M[r[x]+r[y]] <- r[z] |
| bez $r_x$,LBL | # PC <- (r[x]==0):NPC+LBL:NPC |

**TABLE 1: Instruction set for the PIPE processor**

The first step in building a pipeline model is to derive resource vectors for each of the instructions. This is done by examining each stage in the pipeline and listing all resources used by the instruction in the stages. Some resources may be accessed simultaneously by multiple instructions: these are called *counted resources* [BR97]. The PC (and all resources in the instruction fetch stage) and the register file fall in this class. To simplify the discussion we avoid counted resources and focus on resources that are used by a single instruction per cycle. We return to counted resources in Section 3.2.

| **instruction** | **instruction fetch (cycle 0)** | **instruction decode (cycle 1)** | **execute (cycle 2)** | **memory (cycle 3)** | **write back (cycle 4)** |
|---|---|---|---|---|---|
| add | - | $\text{ir}_0$ | $\text{alu}_0$ | | |
| sll | - | $\text{ir}_0$ | $\text{alu}_0$ | | |
| bez | - | $\text{ir}_1$+bralu | | | |
| ldst (ld/st) | - | $\text{ir}_1$ | $\text{alu}_1$ | M | M |

**TABLE 2: Resource vectors for PIPE processor instructions**

The resource vector for the `add` instruction is constructed in the following manner. In the instruction fetch stage, all resources are ignored because they are counted resources; in the instruction decode stage, $\mathsf{ir}_0$ is used and the register file is ignored because it is counted; in the execution stage, $\mathsf{alu}_0$ is used; and in the write-back stage, the register file is written (but ignored). The resource vectors for the five PIPE instructions are shown in Table 2. Note that `ld` and `st` use the same resources in identical patterns. For this reason they have been combined into the `ldst` instruction class.

A reservation table is used to model the processor pipeline. The reservation table is initialized to an empty state to represent an empty pipeline. An instruction may be legally scheduled if its resource vector can be composed (a UNION operation) with the reservation table without oversubscribing a resource. A scheduler for a multi-issue processor will try to schedule as many instructions in the cycle as possible. Once no additional instructions can be issued, the reservation table is in a “cycle-advancing” state. To simulate the advance of the clock the leftmost set of resources is removed and all other resources are shifted forward. This corresponds to instructions advancing in the pipeline.

| instruction | cycle 1 | cycle 2 | cycle 3 | cycle 4 | comment |
|---|---|---|---|---|---|
| empty reservation table | Ø | Ø | Ø | Ø | |
| `add` | $ir_0$ | $alu_0$ | Ø | Ø | |
| new reservation table | $ir_0$ | $alu_0$ | Ø | Ø | empty + `add` |
| `ldst` | $ir_1$ | $alu_1$ | M | M | |
| new reservation table | $ir_0+ir_1$ | $alu_0+alu_1$ | M | M | pipeline+`ldst` |
| *No more instructions can be scheduled in this cycle, advance the cycle by shifting the reservation table* | | | | | |
| shifted reservation table | $alu_0+alu_1$ | M | M | Ø | |
| `sll` | $ir_0$ | $alu_0$ | Ø | Ø | |
| new reservation table | $ir_0+alu_0+alu_1$ | $alu_0$+M | M | Ø | pipeline+`sll` |
| `add` | $ir_0$ | $alu_0$ | Ø | Ø | |
| new reservation table | ***$ir_0$ hazard*** | ***$alu_0$ hazard*** | M | | not legal |

**TABLE 3: Scheduling [add, `ldst`, `sll`, add] on a dual-issue pipeline using reservation table and resource vectors.**

The reservation table in Table 3 demonstrates scheduling an `add`, a `ldst`, an `sll` and a concluding `add` on an empty pipeline. The final `add` instruction causes a hazard on $ir_0$ and $alu_0$ and so it cannot be legally scheduled.

There are two major problems with using the reservation table method to check for pipeline contention. First, it can be slow. Each time a query is made to the CQM, a bitwise comparison on several long words must be performed. When a “cycle-advancing” state is reached in the reservation table, more bitwise operations must be performed to advance the simulation of the instructions in the pipeline. Second, it can consume a great deal of space. Each resource vector requires *#resources* × *length of longest pipeline in stages* bits to model the instruction. The MIPS R3000 processor has 23 modeled resources and a division instruction that may take up to 36 cycles; each resource vector would require 828 bits! For more advanced scheduling it is sometimes necessary to save the state of the pipeline after each instruction is scheduled. To move an already scheduled instruction, such as when attempting code motion, the state of the pipeline after instruction *I* is scheduled must be known so that instruction *I*’s resources may be unsubscribed as the instruction is unscheduled and moved.

To address these concerns an approach based on finite-state automata can be used. The key insight into building a scheduling FSA is that what is wanted is a tool to reason about strings in the language of structural hazard-free instruction schedules [Mül93]. An FSA is built at compile-compile time that considers all possible schedules of instructions. Once the FSA is built, the resource vectors and the reservation table can be discarded. Using an FSA to recognize legal instruction schedules is fast. To check the legality of scheduling an instruction, a table lookup of the sort `FSA[state][instruction]` is made: bitwise OR’ing of the resource vector and reservation table is no longer required because it has been done at compile-compile time. The next section outlines the process for building instruction scheduling FSA.

## 3.1 Finite-State Automata for Scheduling

Constructing the scheduling FSA directly from the reservation tables is inefficient. Each state in the FSA corresponds to a reservation table with the initial state corresponding to the empty reservation table. To start, a resource vector for an instruction is composed with the empty reservation table. If the instruction does not cause hazards with the state, then a new state is produced and the transition from the initial state to the new state is labeled with the instruction. This process is repeated for all instructions. If all of the instructions cause hazards with the state, then the reservation table for the state is shifted left to simulate the advancement of a cycle. This construction continues until no more states can be produced. This procedure is slow because collisions between instructions are rediscovered as each new state is produced.

To overcome this inefficiency a data structure known as a *collision matrix* (CM) is computed to determine all instruction interaction hazards and when they happen. The collision matrix can be viewed as a function:

$$\text{CM: } \textit{instruction} \times \textit{instruction} \times \textit{cycles} \rightarrow \textit{boolean.}$$

Given two instructions, $inst_1$ and $inst_2$ and an interval, *intv*, the CM will specify whether there is a hazard when $inst_1$ and $inst_2$ are scheduled *intv* cycles apart. Once the CM is built, an FSA can be constructed by using the CM to determine all instructions which have a legal transition out of a particular state in the FSA.

### Constructing the collision matrix

The collision matrix entry for instructions, $inst_1$, $inst_2$ and interval, *intv* is set to `1` if issuing $inst_2$ *intv* cycles after issuing $inst_1$ results in a structural hazard. The collision matrix is computed as follows [PF94]:

$$CM[inst_1, inst_2, intv] = \begin{cases} 1 & \exists x \text{ such that } inst_1\left[x + intv\right] \cap inst_2\left[x\right] \neq \varnothing \\ 0 & otherwise \end{cases}$$

For instance, to calculate the CM entries for instruction $inst_1$= `add` and $inst_2$=`add`, as in Table 4, start by composing the resource vector for `add` with the reservation table of an empty pipeline. Call the resulting

| instruction | cycle 1 | cycle 2 | cycle 3 | cycle 4 | comment |
|---|---|---|---|---|---|
| `add` in pipeline<br>second `add` | $ir_0$<br>$ir_0$ | $alu_0$<br>$alu_0$ | | | |
| new reservation table | ***$ir_0$ hazard*** | ***$alu_0$ hazard*** | | | `CM[add,add,0]=1` |
| *Remove second add from the reservation table. Shift the reservation table to simulate advance of a cycle.* | | | | | |
| shifted vector<br>`add` | $alu_0$<br>$ir_0$ | <br>$alu_0$ | | | |
| new reservation table | $ir_0$+$alu_0$ | $alu_0$ | | | `CM[add,add,1]=0` |

**TABLE 4: Constructing the collision matrix CM[`add`, `add`,`time`].** Conflict arises when two `add`s are scheduled in same cycle. No such conflict occurs if two `add`s are scheduled one cycle away.

reservation table $RT_{orig}$. To determine if a conflict occurs at *intv*=0, initiate a second `add` instruction by composing its resource vector with $RT_{orig}$. Since hazards arise, `CM[add,add,0]=1.` To determine if a conflict occurs at *intv*=1, shift $RT_{orig}$ by one cycle, and try to initiate a new `add` instruction. In this case, no conflict arises, so `CM[add,add,1]=0`. This procedure of shifting $RT_{orig}$ by *intv* and composing resource vectors with it continues until $RT_{orig}$ shifted by *intv* is empty. Table 5 lists the conflict matrices for all instructions in the PIPE processor.

| Instruction | | `CM[A,B,intv]` | | | | Instruction | | `CM[A,B,intv]` | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| A | B | `intv=0` | `1` | `2` | `3` | A | B | `intv=0` | `1` | `2` | `3` |
| `add` | `add` | 1 | 0 | 0 | 0 | `ldst` | `add` | 0 | 0 | 0 | 0 |
| | `sll` | 1 | 0 | 0 | 0 | | `sll` | 0 | 0 | 0 | 0 |
| | `bez` | 0 | 0 | 0 | 0 | | `bez` | 1 | 0 | 0 | 0 |
| | `ldst` | 0 | 0 | 0 | 0 | | `ldst` | 1 | 1 | 0 | 0 |
| `sll` | `add` | 1 | 0 | 0 | 0 | `bez` | `add` | 0 | 0 | 0 | 0 |
| | `sll` | 1 | 0 | 0 | 0 | | `sll` | 0 | 0 | 0 | 0 |
| | `bez` | 0 | 0 | 0 | 0 | | `bez` | 1 | 0 | 0 | 0 |
| | `ldst` | 0 | 0 | 0 | 0 | | `ldst` | 1 | 0 | 0 | 0 |

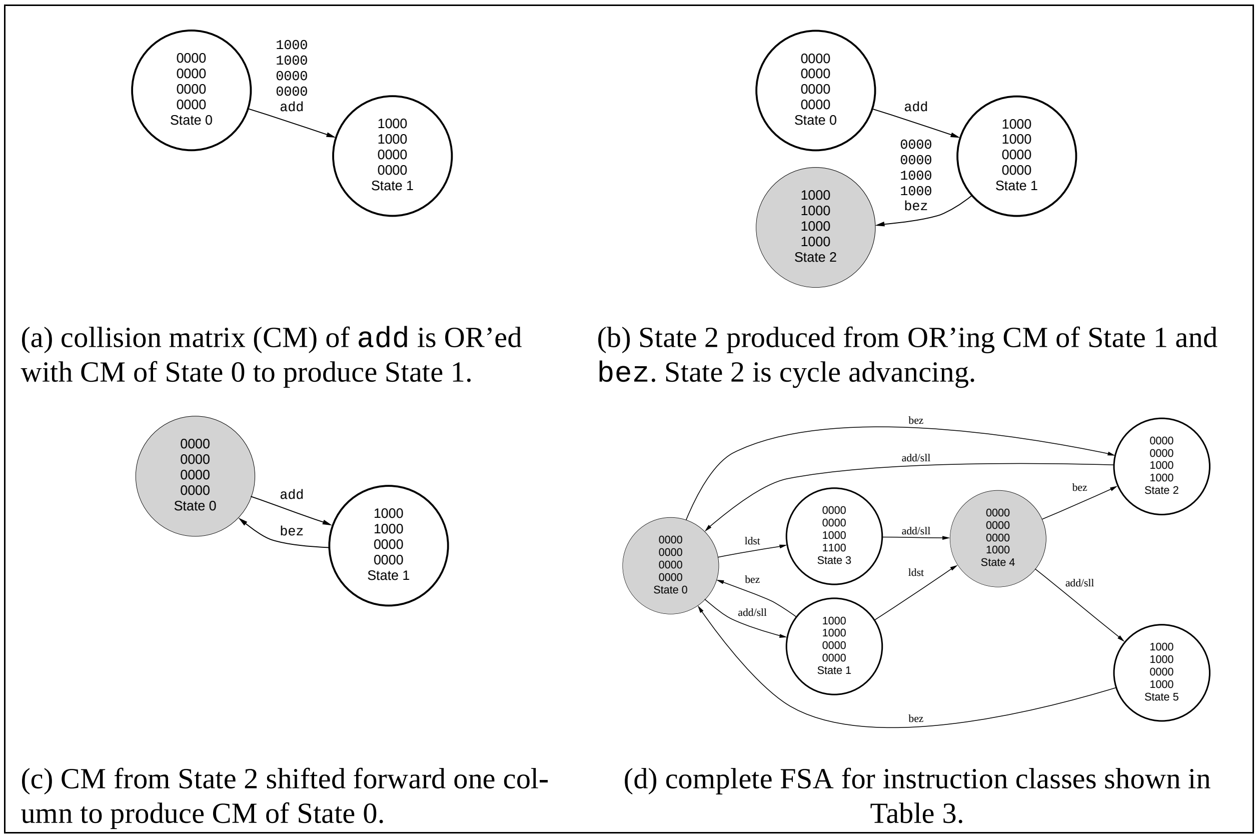
**TABLE 5: Collision Matrix for all instructions of PIPE processor**

### Constructing the scheduling FSA

Constructing a scheduling FSA based on the collision matrix is similar to constructing one from reservation tables; in this case, each state in the FSA corresponds to a conflict matrix. The first state in the FSA is constructed by starting with a CM for an empty pipe (a zero in every position for all instructions and all cycles). The collision matrix for an instruction, such as `add,` is composed (OR'ed) with that of the empty pipeline. Figure 2a demonstrates this composition. This new matrix represents a new state in the pipeline. A transition from State 0 to State 1 is labeled with the instruction that produced the new state.

The 1's in the first column of State 1 indicate that neither an `add` instruction nor an `sll` instruction may be issued from this state, since either would result in a conflict: only `ldst` and `bez` may be issued from this state. Composing the collision matrix for the `bez` instruction with the matrix for State 1 results in a new State 2 (Figure 2b). 1's in the whole first column indicate that no instructions may be issued from this state. This is known as a "cycle-advancing" state and is indicated by a gray shading over the state. To model the advancing of a state, the collision matrix is shifted forward by a column. The resulting collision matrix is now composed of all 0's. Because this is just State 0 a transition from State 1 to State 0 is labeled with the `bez` instruction and State 0 is marked as cycle advancing (Figure 2c). State 2 is deleted.

The process of generating new states and labeling transitions continues until no new states can be generated. The constructed finite-state automaton recognizes the language of sequences of instructions which are free of structural hazards.[1] The constructed FSA recognizes legal instruction sequences in basic blocks.


0000 0000 0000 0000 State 0
1000 1000 0000 0000 add
1000 1000 0000 0000 State 1
0000 0000 0000 0000 State 0
add
1000 1000 0000 0000 State 1
0000 0000 1000 1000 bez
1000 1000 1000 1000 State 2
0000 0000 0000 0000 State 0
add
bez
1000 1000 0000 0000 State 1
bez
add/sll
0000 0000 1000 1000 State 2
bez
0000 0000 1000 1100 State 3
add/sll
0000 0000 0000 1000 State 4
ldst
0000 0000 0000 0000 State 0
ldst
bez
add/sll
1000 1000 0000 0000 State 1
add/sll
1000 1000 0000 1000 State 5
bez

(a) collision matrix (CM) of `add` is OR'ed with CM of State 0 to produce State 1.

(b) State 2 produced from OR'ing CM of State 1 and `bez`. State 2 is cycle advancing.

(c) CM from State 2 shifted forward one column to produce CM of State 0.

(d) complete FSA for instruction classes shown in Table 3.

**FIGURE 2: Construction of a pipeline modeling finite-state automaton.**

## 3.2 Extensions to the scheduling FSA

Modern processors often have multiple copies of identical resources, such as adders, shifters, ALUs, etc., and resources which allow multiple simultaneous accesses, such as multiported register files. These sorts of

1. Because `add` and `sll` have the same collision matrix, they result in the same transitions from state to state. The FSA in Figure 2d reflects these instructions classes.

functional units are called *counted resources* because the scheduler behaves as if there is a counter associated with each of them.

Counted resources can be modeled by multiple resource vectors [BR97]. If an instruction can use one of two identical resources, then a separate resource vector for each of the two resources will be written for the instruction. For instance, suppose the PIPE processor is extended, as in Figure 3, so that any

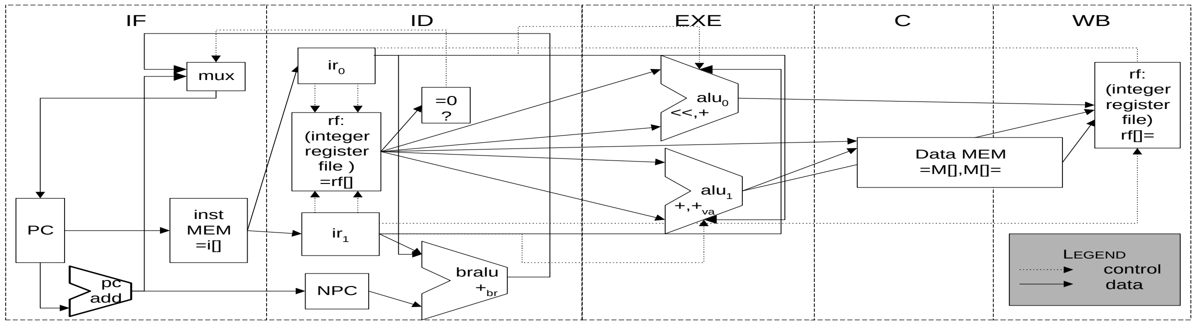


**FIGURE 3: Modified PIPE processor.** This processor can issue any instruction from any slot and has the capability of issuing two `adds` simultaneously.

instruction may be issued from any instruction register and $\text{alu}_1$ is changed so that it can execute `add` instructions. The `add` instruction for this processor will have four resource vectors to reflect the different combinations of counted resources. These four resource vectors are shown in Table 6. To avoid duplication

| instruction | instruction decode stage | execute stage | memory stage | write back stage |
|---|---|---|---|---|
| $\texttt{add}_1$ | $\text{ir}_0$ | $\text{alu}_0$ | | |
| $\texttt{add}_2$ | $\text{ir}_0$ | $\text{alu}_1$ | | |
| $\texttt{add}_2$ | $\text{ir}_1$ | $\text{alu}_0$ | | |
| $\texttt{add}_4$ | $\text{ir}_1$ | $\text{alu}_1$ | | |

**TABLE 6: Multiple resource vectors for add instruction on the modified PIPE processor.**

we adopt a shorthand notation for multiple resource vectors. To denote a counted resource *res* with two copies, we write $res_{[0\text{-}1]}$. The resource vectors for the extended PIPE processor are shown in Table 7.

| instruction | instruction decode stage | execute stage | memory stage | write back stage |
|---|---|---|---|---|
| `add` | $\text{ir}_{[0\text{-}1]}$ | $\text{alu}_{[0\text{-}1]}$ | | |
| `sll` | $\text{ir}_{[0\text{-}1]}$ | $\text{alu}_0$ | | |
| `bez` | $\text{ir}_{[0\text{-}1]}$+bralu | | | |
| `ldst` | $\text{ir}_{[0\text{-}1]}$ | $\text{alu}_1$ | M | M |

**TABLE 7: Resource vectors for the modified PIPE processor.** The [0-1] notation indicates a nondeterministic “choose one” semantics.

From these multiple resource vectors a nondeterministic FSA (NFSA) can be built. The NFSA is then converted into a deterministic FSA by standard subset construction techniques [HU79]. The original degrees of freedom represented by the nondeterministic transitions in the NFSA allow the scheduler to “select” the right resource at the right time. For instance, the instruction sequence `[add, sll]` is legal because the scheduler selects the resource $\text{alu}_1$ for instruction `add` so that instruction `sll` can use $\text{alu}_0$.

## 3.3 Shortcomings of Resource Vector Schedulers

While there is a certain elegance in using resource vectors to generate scheduling components, this approach can become quite cumbersome when writing realistic compilers for realistic processors. Instruction-by-instruction descriptions give little sense of the structure of the pipeline. Additionally, the “cut and paste” mentality that arises from the repetition of specifying each instruction makes it easy to omit some crucial resource that differentiates one instruction from another.

For purposes of architectural experimentation these descriptions are very burdensome. If new instructions are proposed for a processor, the pipeline description gives little information about whether such an instruction can be supported by the current pipeline implementation. Conversely, if the pipeline implementation changes, then each instruction resource vector must be inspected to reflect the change in stages. This is not always simply a matter of moving resources from one stage to another.

| instruction | instruction decode | execute | memory |
|---|---|---|---|
| `add` | $ir_{[0-1]}$ | $alu_{[0-1]}$ | |
| `sll` | $ir_{[0-1]}$ | $alu_0$ | |
| `bez` | $ir_{[0-1]}$+bralu | | |
| `ldst` | $ir_{[0-1]}$ | $alu_1$ | M |
| a) Unconstrained resource vector | | | |

| instruction | instruction decode | execute | mem-ory |
|---|---|---|---|
| `add(slot0)` | $ir_0$ | $alu_0$ | |
| `add(slot1)` | $ir_1$ | $alu_{[0-1]}$ | |
| `sll` | $ir_{[0-1]}$ | $alu_0$ | |
| `bez` | $ir_{[0-1]}$+bralu | | |
| `ldst` | $ir_{[0-1]}$ | $alu_1$ | M |
| b) Constraint added that an `add` may not be issued before a `sll` in the same cycle | | | |

**TABLE 8: Unconstrained and constrained resource vectors for the modified PIPE processor.**

Often artificial resources are added to a resource vector to introduce ordering constraints in the scheduling FSA. For example, suppose the issue logic of the modified PIPE processor prohibits issuing an `add` before an `sll` in the same cycle.[1] This restriction can be expressed for the case of two instruction issue, as indicated in Table 8b, but the mechanism for enforcing this constraint is subtle. If the pipeline is extended to issue three instructions per cycle, this mechanism no longer enforces the “no `add` before an `sll` in the same cycle” constraint. Notice, too, that the counted resource notation no longer completely applies to the `add` instruction and that it requires several resource vectors to be explicitly spelled out.

Generation of scheduling components from resource vector pipeline descriptions is a very powerful technology for retargetable compilers. The current state-of-the-art supports scheduling for multiple instructions per cycle and for multiple functional units. The pitfalls of using resource vector pipeline descriptions are:

1. The machinery for maintaining instruction-by-instruction descriptions is cumbersome and error-prone;
2. The descriptions do not support experimentation with either the instruction set architecture or the architectural implementation;
3. The descriptions can contain subtle “logic tricks” that help introduce ordering constraints in the FSA; these constraints may be hard to maintain should the pipeline implementation change.

The purpose of this research is to address these three shortcomings of resource vector pipeline descriptions.

# 4 Proposed Solution - Annotated Pipeline Graphs

The central thesis of this research is that pipeline descriptions will be easier to use, more accurate and more amenable to experimentation if the pipeline description is decoupled from the instruction set architecture description.This research will separate the pipeline description from the instruction set architecture by

1. This example is taken from the issue logic of the UltraSPARC I&II[Sun97].

developing the *annotated pipeline graph*. The annotated pipeline graph is similar in appearance to diagrams found in architecture manuals in that it depicts the functional units in the processor, the operations performed by the units and the stages in which these functional units are used. Regular expressions are used to annotate the graph with additional scheduling constraints. From the annotated pipeline graph and an existing instruction set description, an FSA can be generated for inclusion in an instruction scheduler.

## 4.1 Pipeline Graphs

The pipeline graph for the extended PIPE processor, shown in Figure 3, is representative of the proposed pipeline graph. The graph is partitioned into pipeline stages. Each stage is labeled with the name of the stage and each stage contains diagrams of functional units that are used in the stage. Each functional unit is named and contains a list of operations that the functional unit executes. Functional units are connected by means of control flow and data flow edges.

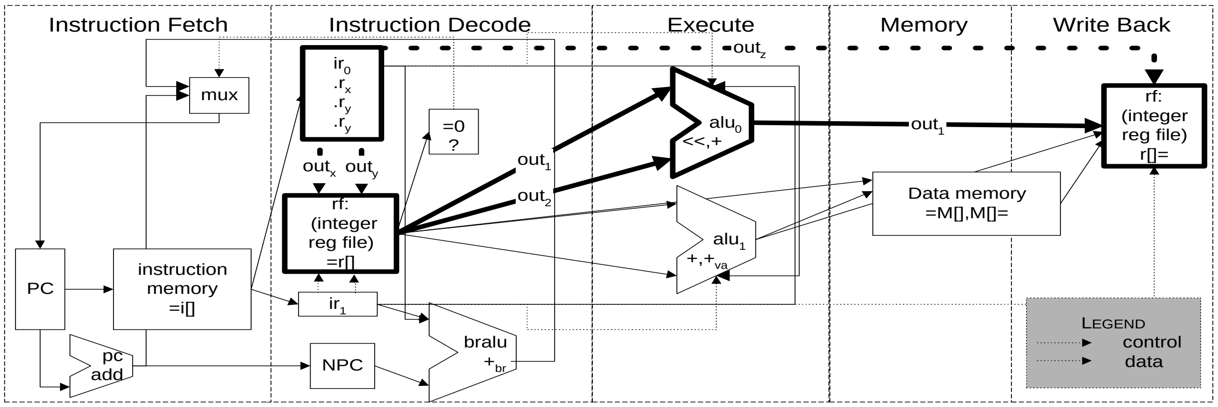


**FIGURE 4: Pipeline graph highlighting functional units, control flow and data flow of `sll` instruction through the pipeline.** Several of the control and data lines have been labeled. Additionally, $ir_0$ has been labeled with several data field operands. These include the operands $.r_x$, $.r_y$, and $.r_z$ (to determine source and destination registers).

The task of generating resource vectors from this pipeline graph can be carried out in a mechanical way. Pick an instruction, e.g. `sll`, and identify the functional units it uses in each stage of the pipeline. Figure 4 shows one path (heavy line) of the `sll` instruction through the pipeline.[1] For those stages in which more than one unit is used, collect all of the units by joining them with the "+" operator. For instance, in the instruction decode stage, the instruction uses instruction slot $ir_0$ and the register file rf. The only ALU capable of `sll` operations is $alu_0$, so the execution stage portion of the resource vector is: $alu_0$. The completed resource vector for `sll` is shown in Table 9.

| instruction | inst fetch stage | inst decode stage | exec stage | mem stage | write back stage |
|---|---|---|---|---|---|
| `sll` | N/A | $ir_0$+rf | $alu_0$ | | rf |

**TABLE 9: Resource vector for the `sll` instruction.**

To perform this process automatically, what is required is a method to discover the functional units that an instruction uses and the control and data paths required to carry out the execution of the instruction. The next section outlines a process for automatically discovering the resource vector from the pipeline graph.

1. The instruction fetch stage and associated resources are ignored to simplify the example.

## 4.2 Derivation of Resource Vectors from the Pipeline Graph

This research will apply "instruction selection" compiler technology to the problem of automatically generating resource vectors from pipeline graphs. The approach taken here will be to take existing tree-based descriptions of instructions and match patterns derived from the pipeline graph against them. As a pattern is matched and a tree rewritten, resources uses are generated based on the matched pattern. In the following description we will use tree pattern matching technology similar to `iburg` [FHP92], but there are many similar rewriting technologies which could be applied to the problem should `iburg` not prove flexible enough.

This research will use instruction descriptions based on register transfer lists (RTLs). RTLs are a machine independent way of representing the changes to the instruction set architecture (ISA) that occur as a result of executing an instruction. Register references appear as `r[n]`, while memory references appear as `M[addr]`. Table 10 lists RTLs for all of the PIPE processor's instructions.

| Register Transfer List | ISA instruction | comment |
|---|---|---|
| `r[z] <- r[x] + r[y]` | add $r_x, r_y, r_z$ | add |
| `r[z] <- r[x] << r[y]` | sll $r_x, r_y, r_z$ | shift logical left |
| `r[z] <- M[r[x]+r[y]]` | ld $r_x, r_y, r_z$ | load |
| `M[r[x]+r[y]] <- r[z]` | st $r_x, r_y, r_z$ | store |
| `(r[x]==0):PC=NPC+LBL:NPC` | bez $r_x$,LBL | branch on r[x]==0 |

**TABLE 10: RTL's for the PIPE processor instructions.**

To perform the pattern matching it is convenient to have the RTLs in tree form, known as TRTL. Figure 5 shows TRTLs for the PIPE processor. TRTLs can be derived from the machine description of the *vpo* compiler. With TRTLs, the nodes are either terminal operands or interior operators. The interior operators evaluate their children. These operators include accessing the register file and memory file, addition, shifting and addition for branching and addressing memory. The terminal operands are constants, such as register numbers, constant numerals or labels.

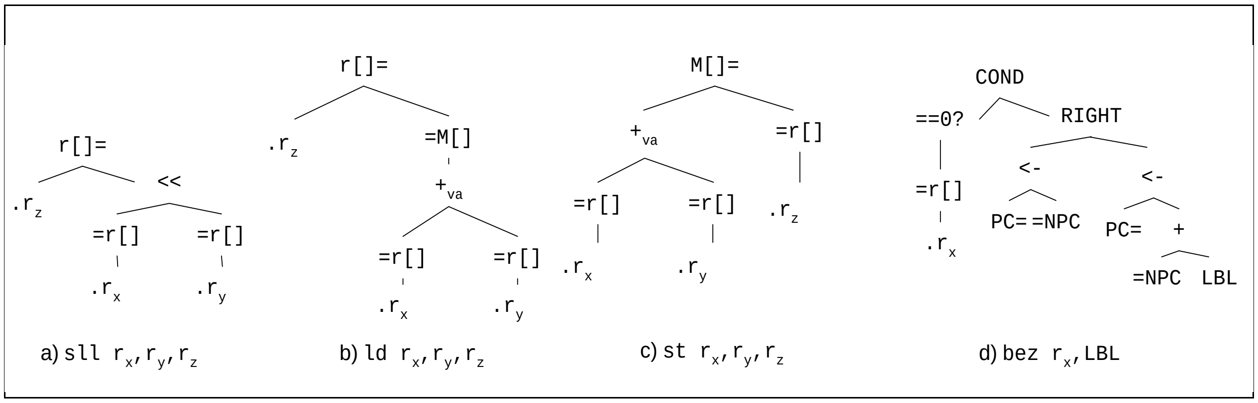


**FIGURE 5: Tree RTLs corresponding to RTLs in Table 6 for the PIPE processor.** The `sll` and `add` trees are very similar. The .r notation indicates a register field of an opcode.

Tree patterns are derived from the functional units and their operations, the data paths and control paths. TRTLs have a specific shape that we can exploit. Interior nodes are operators that have one or more children and terminal nodes have zero children. The following algorithm will generate tree patterns for operator nodes. It works by scanning the pipeline graph for functional units and generating patterns for all of the operations that the functional units perform.

```
Input.    A collection of tree RTLs (TRTL) and the pipeline graph
Output. A collection of tree matching patterns and their associated actions.
Method. First build the nonterminal (interior node) tree matching patterns, then build the tree matching patterns for
terminals.
1.   For each  operation, op, specified in the TRTL do the following.
       for each  functional unit, fun_unit, specified in the pipeline graph with the operation label op
             for each stage, stg,  in which fun_unit is labeled with op
              if  the functional unit fun_unit has any outputs {
                   for each output, out,  from fun_unit
                            for each  combination of inputs, $input_0$..$input_n$, into fun_unit
                               generate pattern    fun_unit.out:   op($input_0$..$input_n$)
                               generate action     res_vec[stg]  := res_vec[stg] ∪ fun_unit
             } else, if the functional unit has no outputs in a stage {
                            for each combination of inputs, $input_0$..$input_n$, into fun_unit
                               generate pattern    fun_unit:    op($input_0$..$input_n$)
                               generate action     res_vec[stg]  := res_vec[stg] ∪ fun_unit
             }
2. For each  terminal, term, specified in the TRTL do the following.
       for each  functional unit, fun_unit, that is labeled with term
             for each stage, stg,  in which term  labels fun_unit
                   for each output, out,  from fun_unit
                               generate pattern        fun_unit.out:     term
                               generate action         res_vec[stg]  := res_vec[stg] ∪ fun_unit
```

**ALGORITHM 1: Generating tree matching patterns from TRLT and the pipeline graph**

The algorithm generates patterns and actions that are to be applied when the pattern is matched. To illustrate the use of this algorithm, it is applied to the highlighted portion of the pipeline graph in Figure 4 and the TRTL in Figure 5 to yield the interior and terminal node patterns shown in Table 11.

| # | Pattern | Action |
|---|---|---|
| 1 | `rf:` `r[]=(`$alu_0$`.`$out_1$`,`$ir_0$`.`$out_z$`)` | *res_vec[WB] =res_vec[WB]* ∪ ***rf*** |
| 2 | $alu_0$`.`$out_1$`:` `<<(rf.`$out_1$`,rf.`$out_2$`)` | *res_vec[EXE] =res_vec[EXE]* ∪ ***$alu_0$*** |
| 3 | `rf.`$out_1$`:` `=r[](`$ir_0$`.`$out_x$`)` | *res_vec[ID] =res_vec[ID]* ∪ ***rf*** |
| 4 | `rf.`$out_2$`:` `=r[](`$ir_0$`.`$out_y$`)` | *res_vec[ID] =res_vec[ID]* ∪ ***rf*** |
| 5 | $ir_0$`.`$out_x$`:` `.`$r_x$ | *res_vec[ID] =res_vec[ID]* ∪ ***$ir_0$*** |
| 6 | $ir_0$`.`$out_y$`:` `.`$r_y$ | *res_vec[ID] =res_vec[ID]* ∪ ***$ir_0$*** |
| 7 | $ir_0$`.`$out_z$`:` `.`$r_z$ | *res_vec[ID] =res_vec[ID]* ∪ ***$ir_0$*** |

**TABLE 11: Interior and terminal node patterns for instruction `sll`.** The actions indicate which resources should be added to the resource vector (res_vec) and the stages in which they should be inserted. Note: spurious patterns, such as those indicating inputs to functional units from multiple instruction registers, have been elided.

Tree pattern matching algorithms are applied to the tree RTL and the pipeline graph patterns. The pattern matcher rewrites the tree and generates resources for the resource vector. When a “cover” for the instruction tree is found the process is complete.  Figure 6 illustrates the process of matching pipeline patterns against the RTL tree for the `sll` instruction. Table 12 shows the portion of the resource vector that is generated by this matching.

Obviously the patterns in Table 11 are not complete. For instance, there are several questions left unanswered: how are resources such as instruction memory accounted for that don’t directly match an RTL

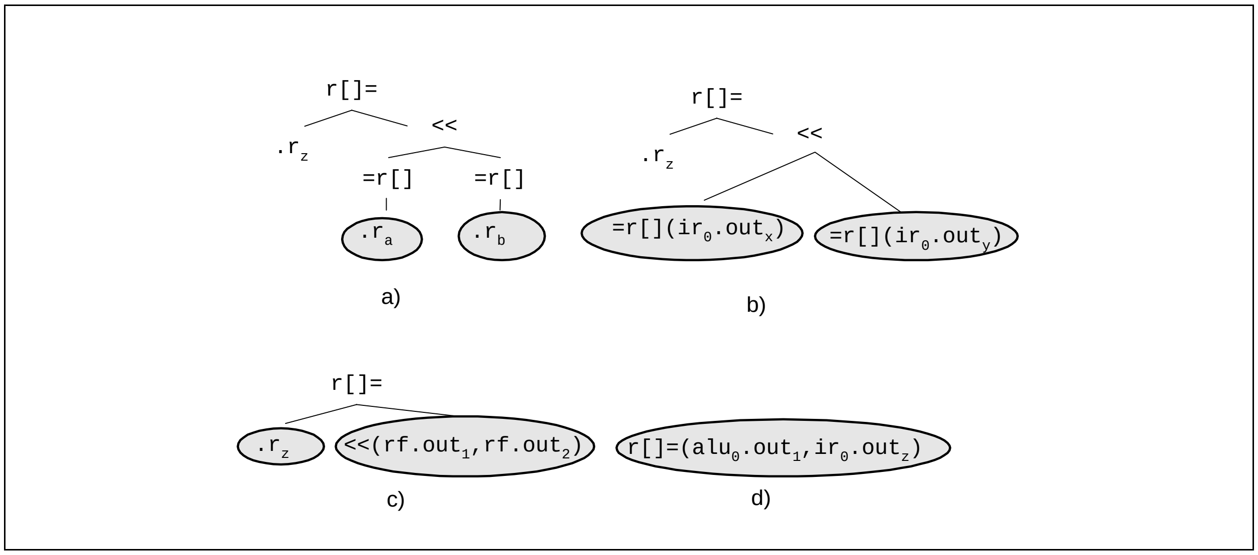


**FIGURE 6: Tree pattern matching against tree RTL for `sll`.** Shaded ovals indicate the tree pattern being matched and rewritten. a) Patterns #5 and #6 are matched and replaced - add instruction register $ir_0$ to resource vector (res_vec) in instruction decode stage. b) Patterns #3 and #4 are matched and replaced - add the register file (rf) to res_vec in the instruction decode stage. c) Patterns #7 and #2 are replaced - add ir0 to res_vec in instruction decode stage (redundant) and add $alu_0$ to res_vec in the execute stage. d) Pattern #1 is replaced - add rf to res_vec in the write back stage.

| instruction | inst fetch stage | inst decode stage | exec stage | mem stage | write back stage |
|---|---|---|---|---|---|
| `sll` | N/A | rf+$ir_0$ | $alu_0$ | | rf |

**TABLE 12: Resource vector for the `sll` instruction.** This resource vector (res_vec) has been derived from the pipeline graph.

operator? What sort of discipline or interface must be developed so that the RTL specifier and the pipeline specifier refer to the same operations and resources? How are branch and conditional execution instructions handled?

While there are significant portions of this research still undone, it does represent a realistic approach to automatically generating resource vectors from pipeline graphs.

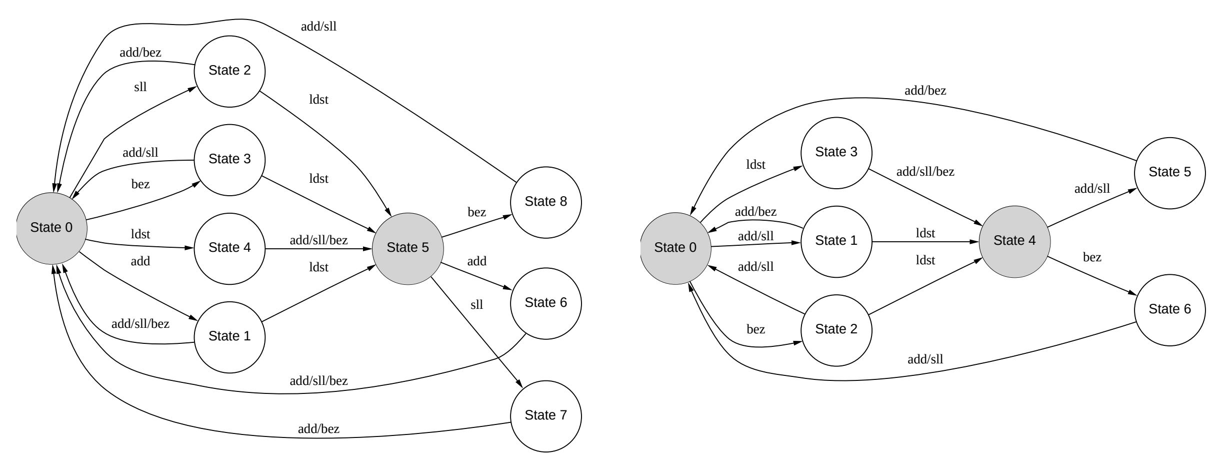
## 4.3 Annotation Language

This section describes a mechanism for annotating pipeline graphs with additional instruction scheduling constraints. The pipeline graph and the annotations are combined at compile-compile time to produce an instruction scheduling component that accurately models the processor pipeline.

To build the pipeline graph the compiler writer consults an architecture manual with information about the numbers and kinds of functional units, pipeline stages, register files and the like. The instruction scheduler components are then generated from this specification. Often, however, there are scheduling constraints that arise from properties of the pipeline that are not apparent from the manufacturer's pipeline specification. These constraints are often given in the form of "guidelines" for code generation. Such guidelines specify forbidden instruction sequences and the penalties for using them.

To accommodate these additional instruction scheduling constraints, we propose an annotation language based on regular expressions. The constraints can be expressed in terms of acceptable or forbidden sequences of instructions. In addition to the usual expressions, the annotative language will include the means to express set intersection(**&**), difference (**-**), union (**|**), concatenation, repeated concatenation(**^**) and Kleene star(*****) [MMR98].

To illustrate, consider a scheduling constraint found in the UltraSPARC II processor [Sun97]. This processor penalizes the execution of schedules containing an `add` followed by an `sll` instruction in the same group. As mentioned in Section 4.1.1, this constraint is hard to model using resource vectors. Using regular expressions, this constraint is the difference of the existing language and the sequence `[add sll]`.



(a) FSM$_{pipe}$ before constraints are considered (b) Composed FSM: FSM$_{pipe}$ - [add sll]

**FIGURE 7: FSM for extended PIPE pipeline graph from Figure 5 on page 9.**

The pipeline graph and the regular expression annotations are combined in the following way. First, the pipeline graph is converted to resource vectors and an FSM for recognizing legal instruction sequences is constructed: call this machine FSM$_{pipe}$. Second, an FSM is generated for each annotation regular expression: these are called FSM$_{RE[0\text{-}N]}$. Third, each annotation FSM$_{RE}$ is composed with FSM$_{pipe}$. Figure 7 shows a finite-state machine for recognizing legal instruction sequences for the extended PIPE pipeline shown in Figure 3. The new composed FSM preserves the accepting state semantics of the original FSM while combining states where possible. For instance, states 1 and 2 from FSM$_{pipe}$ have been combined into one, state 1, in the new composed machine FSM$_{pipe}$ - `[add sll]`. Similarly, states 6 and 7 have been combined to produce state 5 in the new machine.

## 5 Related Work

The Marion system is one of the first retargetable compilers to incorporate information about the pipeline in the machine description [BHE91, Bra91]. Each instruction is accompanied by a resource vector description. Any modification of the pipeline would require a rewrite of each instruction.

Davidson, Shar, Thomas and Patel introduced the use of automata for scheduling processor pipelines. Their work uses a 3D FSA [DSTP75]. If instruction *i* can be issued *c* cycles after state *n* without causing a hazard, then a new state would be returned by the FSA. Müller developed the idea of using the FSA to assist in instruction scheduling [Mül93]. In his paper, Müller gives a postpass method for minimizing the FSA. Proebsting and Fraser construct the minimal FSA directly [PF94].

Bala and Rubin extend the FSA to accommodate processors with multiple instruction issue capabilities [BR97]. They demonstrate how the FSA can be extended to schedule instructions while avoiding hazards which reach across basic block boundaries. They combine a forward and a reverse FSA to assist in global code motion.

Eichenberger and Davidson propose reduced machine descriptions [ED96]. In this scheme, instructions are fully described and no attempt is made to hand optimize the descriptions. The machine description is reduced and a synthesized machine description is produced that preserves all of the scheduling constraints of the original description, but requiring only 20-90% of the space of the original. They argue that this reduction in space requirements make using the reservation table at compile time feasible and fast. They also argue that the automata required for Bala and Rubin type schedulers can get very large unless the compiler writer intervenes to reduce the FSA by hand.

Gyllenhaal, Hwu and Rau propose AND/OR-trees for storing the resource vectors for machines with multiple functional resource units [GmWHR96]. The data structure helps the CQM prune away many bad scheduling attempts. They, like Eichenberger and Davidson, claim that CQM components based on finite-state automata do not support advanced scheduling techniques such as iterative modulo scheduling.

Recently, Gupta and Önder have proposed the *Architecture Description Language* (ADL) for specifying processor microarchitectures [GÖ98]: these specifications are used to generate cycle level simulators and assemblers. Their research is similar to the proposed research in that they attempt to separate the microarchitecture description from the ISA description but they do not completely decouple the two descriptions; their approach is to put those actions common to all instructions in the microarchitecture description and put the instruction-specific actions in the ISA description.

## 6 Research Agenda and Expected Contributions

### 6.1 Research Agenda

The research agenda for the proposed dissertation has five elements:

1. Determine how the pipeline graphs and their components will appear, what features and constructs the annotation language will include and how to integrate the annotations into the pipeline graph specification. With respect to the pipeline graph, these questions need to be answered:

    - How will the components be specified? Multicycle operations are different from fully pipelined operations; how can the two be differentiated? Some multicycle operations prevent the dispatch of any subsequent instructions until they complete; how can this be indicated? When is a result ready to be used and by what functional units?
    - What resources can be ignored? There must be a mechanism for marking functional units “ignorable”.

    The annotation language must be fully specified. Its design raises these questions:

    - What do the instruction sequences look like? What is the notation for indicating groups of instructions in a cycle?
    - Is it useful to give a weight to a sequence of instructions that indicates its desirability? Is there some general mechanism and interface for returning information beyond the simple predicate “this is a legal instruction sequence”?
    - The method of integrating the annotations into the pipeline graph needs to be resolved. How can that be done in a natural and useful way?

    The graphic interface requires work but there are several programmable graph manipulation tools available, such as VISIT and Lefty [DK91]; this portion of the research is not expected to take very long to resolve.

2. Develop the analysis to produce resource vectors from the pipeline graphs and RTLs. Work has begun on this goal, as indicated in Section 4.2, but several subgoals remain:

    - refine the algorithm for generating tree patterns from the graph.
    - modify `iburg` to generate resource vectors from the tree covers.

3. Develop the analysis to compose the resource FSM with the annotation FSM.

- define language of instruction sequences, cycles, breaks in cycles.
- develop a parser for the language that recognizes annotations, produces FSM and composes original FSM with annotation FSM.

4. Integrate the resulting scheduling component into *vpo*.
5. Describe several representative pipeline implementations including superscalar processor(s), an out-of-order processor and a CISC processor.

## 6.2 Expected Contributions and Artifacts

**Contributions**

The proposed research will make several contributions to the compiling community. This work is the first to separate the pipeline description from the instruction set. Because of this decoupling, the compiler writer will be able to describe the processor pipeline in an intuitive, integrated way; the tool will be responsible for matching each instruction with its appropriate pipeline resource usage. In addition to the tool, another contribution will be the analysis required to apply code generation techniques to the problem of reestablishing this match between instruction and pipeline implementation.

Another contribution will be the introduction of an annotation language based on regular expressions to directly specify needed scheduling constraints. This annotation language will allow the compiler writer to refine the pipeline description in a natural way instead of using awkward synthetic resources. We will develop the analysis required to merge the annotations with the pipeline graph to produce instruction scheduler components.

**Artifacts**

As a practical contribution, the WYSIWYG tool will add structure to the pipeline description process by guiding the user through the description, offering a library of often used components and disallowing certain illegal or unlikely connections and components.

From the annotated graph we will produce scheduler components such as contention query modules. We will implement an instruction scheduler which uses the produced module and integrate this with the *vpo* compiler. We will test the portability of the pipeline description tool by targeting several disparate processors. Subject to availability, we will target both direct issue machines and dynamically scheduled processors. We will measure the performance increase gained by using the more accurate pipeline descriptions.

## 7 Summary

In spite of architectural trends towards dynamic scheduling, processors continue to rely on compilers to get peak performance. Compilers, in turn, rely on ever more detailed machine descriptions to produce good instruction schedules. Our experience with microarchitecture level machine descriptions indicates that there is room for improvement. Currently, machine descriptions are based on resource vectors and are written on an instruction-by-instruction basis. This method is tedious and repetitious — it does not take advantage of the structure of the pipeline and is too closely tied to the instruction set description of the processor.

To address the above problems, the proposed research will develop a new, powerful approach for describing modern instruction pipelines by separating the pipeline description from the instruction set description. The proposed approach uses a graphical description of the pipeline and an accompanying annotation language to describe the relevant behavior of a machine's execution pipeline. Using the descriptions of the pipelines and an existing description technique for instruction sets, it will be possible to generate instruction scheduler information automatically. Furthermore, this decoupling of the pipeline description from the instruction description will ease the task of retargeting the compiler as new instruction set extensions and new pipeline implementations appear.

## References

[BD94] Manuel E. Benitez and Jack W. Davidson. Target-specific global code improvement: Principles and applications. Technical Report CS-94-42, Department of Computer Science, University of Virginia, November 4 1994.

[BHE91] David G. Bradlee, Robert R. Henry, and Susan J. Eggers. The Marion system for retargetable instruction scheduling. In Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation, volume 26, pages 229–240, Toronto, Ontario, Canada, June 1991.

[BR97] Bala and Rubin. Efficient instruction scheduling using finite state automata. IJPP: International Journal of Parallel Programming, 25, 1997.

[Bra91] David Gordon Bradlee. Retargetable instruction scheduling for pipelined processors. Technical Report TR 91-08-07, University of Washington, 1991.

[DH96] David A. Dunn and Wei-Chung Hsu. Instruction scheduling for the HP PA-8000. In Proceedings of the 29th Annual International Symposium on Microarchitecture, pages 298–307, Paris, France, December 1996. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[DK91] D. Dobkin and E. Koutsofios. LEFTY: A two-view editor for technical pictures. In Proceedings of Graphics Interface '91, pages 68–76, June 1991.

[DSTP75] E. S. Davidson, L. E. Shar, A. T. Thomas, and J. H. Patel. Effective control for pipelined computers. In COMPCON 75, pages 181–184, NY, 1975. IEEE.

[ED96] Alexandre E. Eichenberger and Edward S. Davidson. A reduced multipipeline machine description that preserves scheduling constraints. In Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation, pages 12–22, Philadelphia, Pennsylvania, 21– May 1996.

[FHP92] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a simple, efficient code generator generator. ACM Letters on Programming Languages and Systems, 1(3):213–226, September 1992.

[GmWHR96] John C. Gyllenhaal, Wen mei W. Hwu, and B. Ramakrishna Rau. Optimization of machine descriptions for efficient use. In Proceedings of the 29th Annual International Symposium on Microarchitecture, pages 349–358, Paris, France, December 2–4, 1996. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[GÖ98] R. Gupta and S. Önder. Automatic generation of microarchitecture simulators. In Proceedings of the IEEE International Conference on Computer Languages, May 1998.

[HU79] J. Hopcroft and J. Ullman. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, N. Reading, MA, 1979.

[LPU95] A. Leung, K. Palem, and C. Ungureanu. Run-time versus compile-time instruction scheduling in superscalar (RISC) processors: Performance and tradeoffs. Technical report, New York University, Computer Science, 1995.

[MMR98] Fernando C. N. Pereira Mehryar Mohri and Michael Riley. A rational design for a weighted finite-state transducer library. In Lecture Notes in Computer Science, page 1. Lecture Notes in Computer Science (to appear), Springer Verlag, 1998.

[Mül93] T. Müller. Employing finite automata for resource scheduling. In Proceedings of the 26th Annual International Symposium on Microarchitecture (MICRO'93), volume 24 of SIG-MICRO Newsletter, pages 12–20, Los Alamitos, CA, USA, December 1993. IEEE Computer Society Press.

[PF94] T. A. Proebsting and C. W. Fraser. Detecting pipeline structural hazards quickly. In ACM, editor, Proceedings of 21st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pages 280–286, New York, NY, USA, 1994. ACM Press.

[Sun97] Sun Microsystems. UltraSPARC I&II User's Manual. Sun Microsystems, Mountain View, CA, USA, 1997.