QUICK PIPING: A Fast, High-Level Model for Describing Processor Pipelines†

Christopher W. Milner and Jack W. Davidson

Department of Computer Science, University of Virginia
Charlottesville, Virginia 22904

{cmilner,jwd}@cs.virginia.edu

ABSTRACT

Responding to marketplace needs, today's embedded processors must feature a flexible core that allows easy modification with fast time to market. In this environment, embedded processors are increasingly reliant on flexible support tools. This paper presents one such tool, called *Quick Piping*, a new, high-level formalism for modeling processor pipelines. *Quick Piping* consists of three primary components that together provide an easy-to-build, reusable processor description:

- Pipeline graphs—a new high-level formalism for modeling processor pipelines,
- *pipe*—a companion domain-specific language for specifying a pipeline graph,
- *pipe miner*—a compiler specification generator for *pipe* descriptions. *pipe miner* processes a *pipe* description and produces a compiler specification that is used to build a compiler that reads the corresponding machine's instruction set and automatically generates resource vectors.

Despite their ubiquity and importance in achieving high performance in modern processors, pipelines—and improving the mechanisms for specifying their operation—have received little attention. Until now, *handwritten* resource vectors have served to specify information about a processor's pipeline and encode relevant information about each instruction's resource usage. Describing the complete set of resource vectors for a machine can be quite tedious and error prone, since it commonly must be developed by hand on an instruction-by-instruction basis.

With its use of pipeline graphs, the *pipe* language, and the *pipe miner* compiler specification generator, *Quick Piping* gives the embedded processor architect and compiler writer an intuitive high-level abstraction of pipelines, a language for specifying a pipeline, and a tool for automatically producing pipeline resource vectors. The resulting specifications are quick to develop, easy to understand, simple to modify and maintain, and can be automatically processed to produce the low-level information required by processor control units and instruction schedulers.

† This work is supported in part by National Science Foundation grant EIA-0072043.

LCTES'02-SCOPES'02, June 19-21, 2002, Berlin, Germany.

Category and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*compilers*, *optimization*; C.1.1 [**Computer Systems Organization**]: Other architecture styles—*pipeline processors*, *cellular architectures*; D.4.7 [**Operating Systems**]: Organization and Design—*real-time systems and embedded systems*

General Terms

Performance, Design, Languages

Keywords

Modeling of computer architecture, pipelines, embedded systems

1 INTRODUCTION

As embedded applications have grown in complexity, so have the processors designed for them. Most embedded processors now use pipelining to gain increased performance. In contrast to desktop microprocessor design, a market where there are typically few processor variants, in the embedded market there are usually many different variants of an embedded processor, each one designed to meet the needs of a particular application area. There are both instruction set variations and underlying microarchitecture variations (i.e., pipeline, number of functional units, and speed and capability of the functional units).

Because embedded processors employ pipelining to help achieve high performance, an important aspect of embedded processor construction is designing and implementing cost-effective, fault-free pipelines [17, 14]. Similarly, an important aspect of compiler construction for embedded processors is designing and implementing instruction schedulers that make effective use of these pipelines. To achieve their respective goals, computer architects and compiler writers rely on run-time and compile-time models of processor pipelines.

Typically a *resource vector* specifies information about when an instruction uses processor resources. A resource vector describes, in a compact tabular form, the resources used by an instruction as it moves through each stage of the pipeline [32, 1]. Processor control units employ resource vectors to control the flow of instructions through the pipeline to prevent overuse of resources. Recent instruction schedulers use automata constructed from resource vectors at compile-compile time to model the pipeline at compile time and produce instruction schedules that minimize the number of pipeline stalls [13, 26, 28, 3].

While resource vectors provide the relevant information about each instruction's pipeline resource usage, processor manufacturers usually do not provide resource vectors to end users. For example, a survey of the published documentation on 11 processors showed that only for one processor did a manufacturer supply resource vectors [13, 31, 16, 30]. As a result, resource vectors for a machine's complete instruction set are usually developed by hand on an instruction-by-instruction basis from informal descriptions of the operation of the pipeline.

Developing resource vectors by hand is tedious and error prone. Modern processors have large instruction set repertoires. While instructions can be grouped into classes (another potential error source), the number of instruction classes can be large. For example, the MIPS32 4K embedded processor core family requires 39 resource vectors to describe the pipeline resource usage of the instruction set [16, 34]. The resulting vectors are subject to logic, interpretation, and transcription errors. Worse, it is difficult to verify the correctness of the resulting vectors. Furthermore, the resulting vectors are hard to maintain and modify.

This paper details an alternative approach to describing processor pipelines and developing resource vectors, called *Quick Piping*. *Quick Piping* incorporates three critical components: a new, domain-specific language, called *pipe*, linked to a new, high-level model for describing pipelines, the *pipeline graph,* and a compiler specification generator called *pipe miner* that generates resource vectors automatically.

Using the pipeline graph model and *pipe, Quick Piping* allows the embedded architect and compiler writer to specify quickly and compactly the operation of the processor's pipeline. The language and the underlying model are sufficiently general that superscalar and multiple-issue processors can be specified. A *pipe* specification of a complex pipeline is typically less than a page of text.

The paper also describes the design and implementation of *Quick Piping's pipe miner* tool, the compiler specification generator for processing *pipe* specifications. *pipe miner* processes a *pipe* specification of a machine and produces a yacc grammar. The grammar, in turn, produces a compiler that processes the machine's instruction set and emits the corresponding low-level resource vectors. In addition, the *pipe miner*-generated compiler verifies that the *pipe* description is complete and consistent with the description of the machine's instruction set and that resource vectors for all instructions have been computed.

There are several advantages to using *Quick Piping* over conventional handwritten resource vectors. First, unlike low-level resource vectors, *pipe* specifications are simple to modify and maintain. This simplicity is due to the high-level of the underlying model. Resource vectors combine information about the instruction set and pipeline behavior. Consequently, if the embedded core requires a change in the instruction set or the structure of the pipeline, the resource vectors must be rewritten. *pipe* specifications, on the other hand, are independent of the instruction set and vice versa. As a result, a change in one need not affect the other.

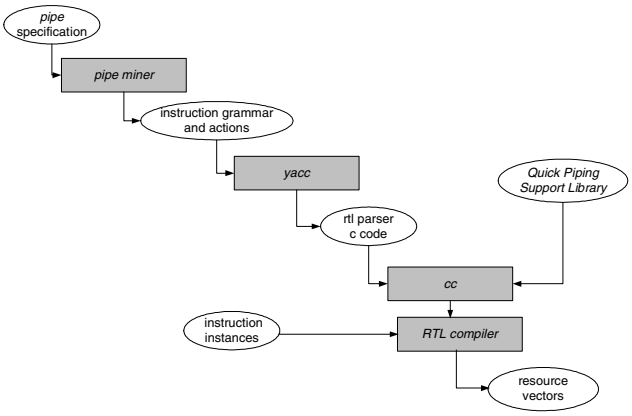

Figure 1: *Quick Piping* toolchain for producing resource vectors

Second, *pipe* specifications support composition whereas resource vectors provide no abstraction mechanism. Composition allows compiler writers and architects to describe complex pipelines to any level of detail with a consistent level of effort. Because of the tediousness of writing resource vectors by hand, the tendency is to favor simplicity over accuracy when developing resource vectors directly.

Third, the *pipe miner* compiler specification generator provides assurance that resource vectors for all instructions will be generated and that the specified pipeline contains the necessary resources to execute the instructions. When writing resource vectors by hand, it is very easy either to omit instructions or place an instruction in an incorrect category. In the fast-changing embedded environment, where instructions are added to customize the processor to the application, these errors are even more likely. Overall, *pipe* provides compiler writers and computer architects a simple, easy-to-use language for obtaining robust and complete resource vectors for a processor.

This paper has the following organization. Section 2 describes pipeline graphs, the underlying model of pipelines. Section 3 describes *pipe*. The *pipe* specification of the MIPS R3000 is used to illustrate the main features of *pipe*. The implementation of the *pipe miner* compiler specification generator is described in Section 4. Section 5 discusses related work and Section 6 provides a summary of *Quick Piping's* benefits.

2 PIPELINE GRAPHS

A review of dozens of processor reference manuals reveals that there are essentially three approaches for describing the operation of a pipeline. Text descriptions, by far, are the most common approach for describing pipeline operation. Typically the descriptions name the various stages and give a brief summary of the operation the stage performs. Table 1 reproduces the description of the MIPS32 4Kp processor core pipeline from the previously referenced datasheet. The tables are often augmented with text to explain exceptional conditions. For example, the following text is from the datasheet.

Stage	Name	Explanation
I	instruction fetch	An instruction is fetched from the instruction cache
E	execution	Operands are fetched from register file. ALU begins arithmetic operation for register to register operation ALU calculates data virtual address for load ALU determines if branch condition is true Instruction logic selects instruction address All multiplies start in this stage
M	memory fetch	ALU operation completes, data cache fetch performed for loads/stores, data cache look-up is performed, multiplies stall here for 31 clocks
A	Align	Aligner aligns load data to its word boundary, multiply/divide updates HI/LO, Mul operation makes result available.
W	writeback	For register-to-register or load instructions the instruction result is written back to the register file during the W stage.

Table 1: MIPS32 4Kp Processor Core Description.

The 4Kp core contains a multiply/divide unit (MDU) that contains a separate pipeline for multiply and divide operations. This pipeline operates in parallel with the integer unit (IU) pipeline and does not stall when the IU pipeline stalls.

Another approach for describing pipeline operation is to supply a diagram or schematic illustrating the operation of the pipeline. Such diagrams typically take two forms—an *instruction execution sequence diagram* or a *pipeline schematic*.

Figure 2 contains the instruction execution sequence diagram for the MIPS R6000. This type of diagram reads from left to right. It names the processor stages and usually contains some indication of the resources used by a stage. For example, Figure 2 shows that the **W** stage uses the register file (**RF**). While instruction execution sequence diagrams help one gain an intuitive understanding of the operation of the pipeline, they do not contain enough details to determine precisely the pipeline's operational characteristics.

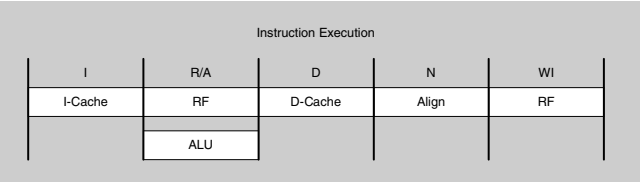

Figure 2: MIPS R6000 instruction execution sequence schematic.

Pipeline schematics, on the other hand, contain much greater detail. Figure 3 shows the pipeline schematic for the MIPS R3000. Pipeline schematics contain all the information one would need to understand the operation of the pipeline, but they are complex and require some effort to digest. Clearly, a pipeline schematic is not appropriate for quickly specifying pipeline behavior.

The third approach for describing pipeline operation is the resource vector. Table 2 gives the integer unit resource vectors for the MIPS R2000/R3000 architecture [28]. As we noted in the introduction, resource vectors are concise and convey the information needed by the instruction scheduler generators and automated design tools. For example, the `jal` entry of Table 2 shows that the jump and link instruction uses the memory resource **mem** at pipeline stage four and the register file and write back resources (**r_d** and **wb**) at stage five. Unfortunately, as noted earlier, most microprocessor manufacturers do not provide resource vectors. The resource vectors can be generated by hand by careful analysis of the pipeline schematic (if one is available). However, this is a time-consuming and error-prone task.

The resource vectors shown have been hand-developed and illustrate a disadvantage of *ad hoc* development. The author of this description hoped to integrate register allocation with instruction selection. Consequently, it was necessary to specify individual registers (e.g., the use of resource **r_31** for the `bgtzal` class). Such additions and tweaks make hand-generated resource vectors harder to maintain.

Our approach to developing the appropriate abstractions for describing processor pipelines has focused on evaluating existing descriptions. Cursory examination of resource vectors and the description approaches shows that the pipeline model should support the concepts of named resources and pipeline stages. The outstanding problem concerns modeling the dependencies between instructions.

Our key insight is the realization that a pipeline schematic specifies the flow of information (i.e., instructions) through the pipeline. Furthermore, the flow of a sequence of instructions through the pipeline is constrained by the connections between resources and the stage location of resources.[1] Flow of information (edges) between resources (nodes) is naturally represented by a graph.

Using a graph to model a pipeline is appealing for several reasons. First, efficient representations of graphs are well known. Second, efficient algorithms for analyzing and manipulating graphs are readily available. Third, a graph is a natural abstraction of a pipeline schematic, which is also a graph, albeit highly detailed.

Thus, our basic model of a pipeline is a directed, partitioned graph. Pipeline resources are nodes in the graph, and directed edges indicate flow of information (both data, address, and control) between resources. The nodes of the graph are partitioned into stages that correspond to the stages in the pipeline.

3 THE *PIPE* LANGUAGE

Fortunately, a language for specifying graphs can be fairly simple and intuitive. Further, there are several domain-specific languages (DSLs) for specifying graphs that provide good models for *pipe* [18]. We have drawn upon these DSLs in designing our language.

1. An instruction sequence is also constrained by the operation characteristics of the resource.

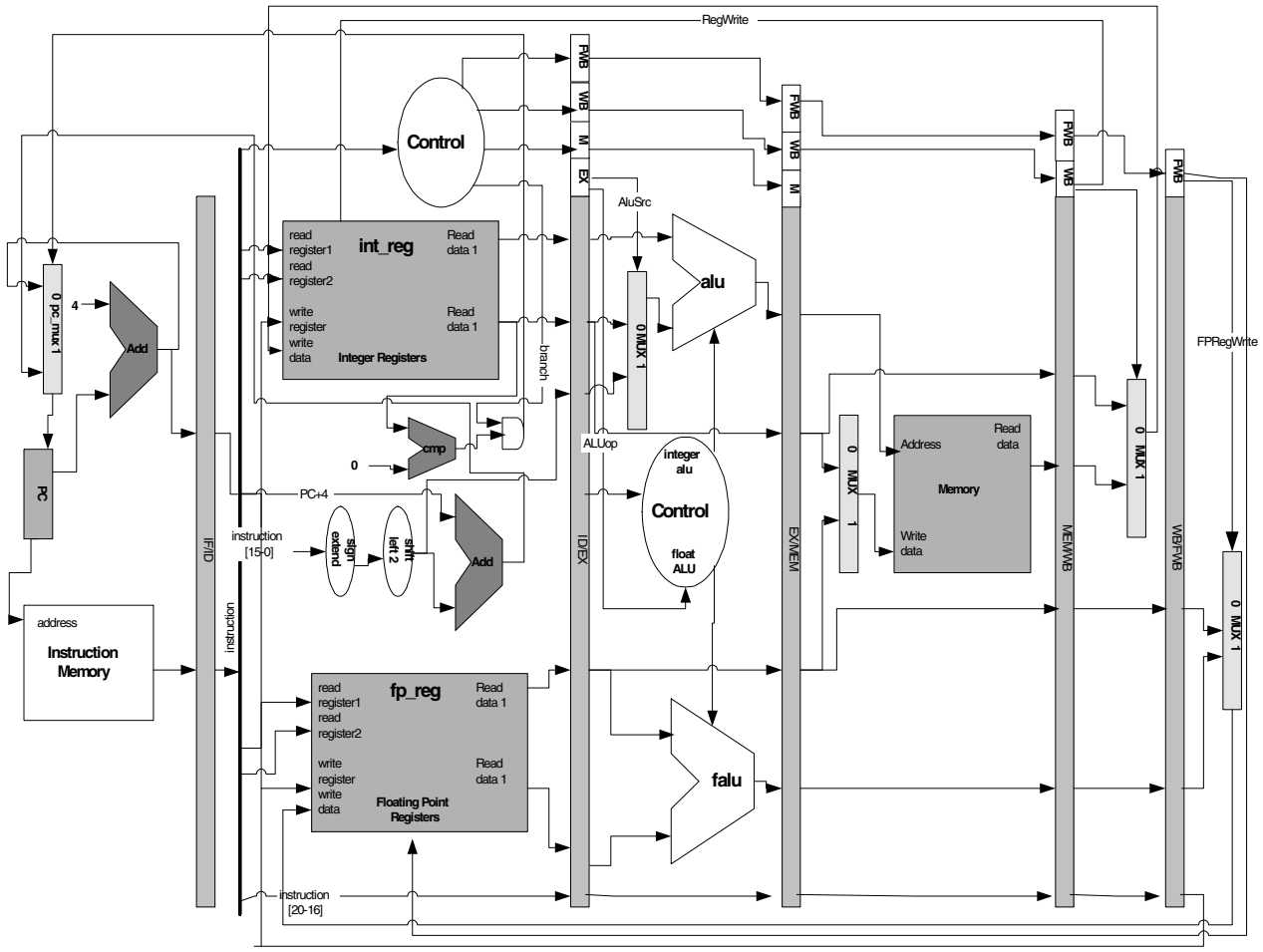

Figure 3: MIPS R3000 pipeline diagram.

Inst.Class	Resource usage
j	pc+ir rd+pc+epc alu mem wb
bgtzal	pc+ir rd+pc+epc alu r_31+mem wb
ctc1	pc+ir rd alu c+s+mem wb pwb
lwc1	pc+ir rd alu mem wb fwb+f_d
mtlo	pc+ir rd alu lo+mem wb
mthi	pc+ir rd alu mem+hi wb
arith	pc+ir rd alu mem r_d+wb
stores	pc+ir rd alu mem wb
loads	pc+ir rd alu r_t+mem wb
jal	pc+ir rd+pc+epc alu mem r_d+wb
mult	pc+ir rd lo+alu+hi lo+hi^11
div	pc+ir rd lo+alu+hi lo+hi^34

Table 2: MIPS R3000 integer unit resource vectors.

To illustrate *pipe*, we use the pipeline specification of the MIPS R3000 pipeline given previously in Figure 3. This pipeline includes a floating-point unit. The syntax of *pipe* is given using *Extended Backus-Naur Form* (*EBNF)* grammars [7].

3.1 Pipeline stages

A *pipe* specification has the following form:

PipeDecl → **pipeline** *name* **:**
stages: *StageDecls*
resources: *ResourceDecls*
connections: *ConnectionDecls*
StageDecls → *name* **:** *ResourceList* {*name* **:** *ResourceList* }
ResourceList → {*name* { **,** *name* }}

where the stages are an ordered list of the pipeline stages with a list of resources each stage contains. The pipe specification for the R3000 pipeline stages is:

```
stages
     IF: IR, PC, pc_adder, pc_mux
     DEC: int_reg,fp_reg,cmp, FCcmp, FC
     EXE: alu, falu, FC_WritePort
     MEM: Memory
     WB: WritePort
     FWB: FP_WritePort
```

For example, the stage **FWB** is the stage used by the floating-point unit to write back **falu** results to its register file.

3.2 Pipeline resources

The next section of a pipeline specification declares the resources. There are six types of resources: functional units, main memory, register files, registers, instruction registers, multiplexers and latches. Resources are distinguished by the operations they can perform and the types of connections they allow. For example, a multiplexer must have a control connection. These constraints are natural for architectural building blocks, but perhaps more importantly they also permit a *pipe* specification to be checked for consistency.

The syntax for declaring resources is:

ResourceDecl → *TypedResource { TypedResource}*
TypedResource →
  functional unit *name* **:** *Operation {* **,** *Operation}*
| **memory** *name* **:** *AccessSpec {* **,** *AccessSpec}*
| **register** *name* **{** **,** *name* **}**
| **register file** *name***:(r|f|d)** *{***,(r|f|d)** *}*
| **instruction register** *name* **:** *Field {* **,** *Field }*
| **mux** *name* **:**
| **latch** *name* **:**
Operation → *Operator { DelayOrRepeat }*
AccessSpec → **(B|W|L|D)** *{LoadStoreType}{ DelayOrRepeat}*
MemorySize → **(B|W|L|D)**
LoadStoreType → *{* **.l | .s** *}*
DelayOrRepeat → *(***^***Delay* | **||** *Repeat)*

The specification of the R3000's resources is:

```
resources:
   functional unit falu: '/.s'^12,'.d'^19,
      CV.s'^2 , CV.d^1,'*.s'^4,'*.d'^5,
      '.ps','.pd', '+.s'^2,'+.d'^2,'=',
      TR, MF, '+.s' , '+.d', '*.s', '*.d',
      ':.s', '<.s', '<.d', WF.s, WF.d,
      '-.s', '-.d', '-.ps','-.pd'
   functional unit alu: '+','*'^12,'&',
      '/'^35,'\\','=',MT,'-.p','~',':','`',
      'g','>','h','s','<','l','!','\'',
      '|','%','#','{','}','"','-','^'
   functional unit cmp: '<', ':', '`' ,'\'',
      '!', 'g', '>','h','s','l'
   functional unit FCcmp: ':','!'
   functional unit pc_adder: '+'

   memory Memory : B.s, W.s, L.s, D.s, B.l,
        W.l, L.l, D.l

   register PC , FC

   register file int_reg: r
   register file fp_reg: f,d

   instruction register IR : GLO, LOCAL,
      LBL, CON, FCON, '0', '1', '2', '3',
      '4', '5', '6','7','8','9'

   mux pc_mux:

   latch WritePort:
   latch FP_WritePort:
```

The specification of a functional unit includes the operations that the functional unit can perform. Some operations have been augmented with type information. For instance, a single-precision floating-point addition operator is listed for the floating-point ALU **falu** as **'+.s'**. This type information is used for consistency checking and to help match the description of the instruction to the resource vector.

When not explicitly given, operations are assumed to have unit latency (i.e., take a single clock cycle). However some operations have longer latencies (e.g., multiplication and division, floating-point operations, etc.). Operations can be annotated with **^** *Delay* to specify explicitly a latency. Similarly, operations can be annotated with **||** *Number* to specify explicitly a repeat count.

The specification of a memory resource includes the sizes of the memory locations that can be accessed and the type of access. Similarly, the integer and floating-point register files specify the data types that can be fetched. This information is necessary for automatically determining the resources an instruction uses. Latencies and repeat counts may be included in the specification. This mechanism can also be used to handle instructions with variable latency.

Multiplexers are used to select which of several inputs to output. In pipelines, multiplexers are used to select the appropriate input to functional units and to write into memory elements. Resource **pc_mux** is a multiplexer. It is used to determine whether to route the branch address of the **ID** stage adder to the **PC** (i.e., a conditional branch), or whether to use the incremented PC (contained in **PC**) to fetch the next instruction. This decision is made under control of the output of the **ID** stage adder. Notice the only input connection type information needed is that **pc_mux** is making a choice based on control input.

3.3 Pipeline connections

The connection section of a pipe program specifies how resources are connected together. The syntax for a connection section is:

ConnectionDecl → **connections:**
  data:
    ConnectionList
  address:
    ConnectionList
  control:
    ConnectionList

ConnectionList → *{ name* **->** *name }*

The following is the connection section for the R3000 pipeline.

```
data:
   int_reg -> cmp
   int_reg -> alu
   int_reg -> Memory
   fp_reg -> falu
   fp_reg -> Memory
   Memory -> WritePort
   Memory -> FP_WritePort
   WritePort -> int_reg
```

```
   FP_WritePort -> fp_reg
   falu -> FP_WritePort
   alu -> WritePort
   falu -> FC
   FC -> FCcmp
   PC -> pc_adder
   pc_adder -> pc_mux
   pc_mux -> PC
   IR -> pc_mux
address:
   alu -> Memory
control:
   FCcmp -> pc_mux
   cmp -> pc_mux
```

The use of distinguished connection types (data, address and control) allows *pipe miner* to check that the pipeline graph is well-formed.

The preceding 57 line specification is the complete *pipe* specification of the MIPS R3000 pipeline shown in Figure 3. Clearly, a *pipe* specification is simple to write. It can be written in an hour or so by examining the available documentation on the pipeline (i.e., an instruction sequence diagram, written descriptions, or a pipeline schematic).

3.4 Multiple instruction issue

Embedded core designers often use multiple instruction issue to boost processor performance. The MIPS64 5K family of embedded cores extends the MIPS32 4K family by (among other things) moving to 64 bits, adding floating-point processing and adding dual instruction capabilities [37, 35, 36].

Multiple instruction issue processors are modeled in *pipe* using two or more instruction register nodes. For example, Figure 4 shows a partial pipeline graph model of the SUN UltraSPARC [33]. The UltraSPARC has two integer ALUs, modeled by **`alu1`** and **`alu2`**, an integer multiplier, **`mul`**, and an integer divider, **`div`**. The processor has two floating-point/graphics operation ALUs, modeled by **`falu1`** and **`falu2`**. The UltraSPARC can issue up to four instructions per cycle. This is modeled by the array of instruction register nodes, **`IR[1-4]`**.

When using *n* instruction register nodes, the pipeline graph defaults to modeling a processor in which any one of *n* instruction registers may be used to issue any instruction. Often, processors have restrictions on the type of operation which may be issued from a particular slot. These restrictions may be specified using *issue constraints*.

Multi-issue processors may place constraints on the instructions that may be issued from a particular issue slot. VLIW processors are the most restrictive in this respect, with each slot in the instruction word reserved for specific operations while superscalar processors with out-of-order issue are the least restrictive.

An issue constraint section may be added to a *pipe* specification to express such restrictions on instruction order. The EBNF syntax for the issue constraint section is:

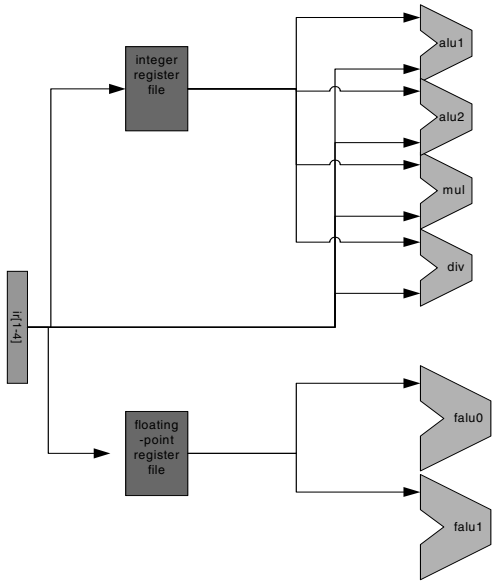

Figure 4: UltraSPARC instruction register nodes.

IssueConstraints → **`issue constraints :`**
Constraint { Constraint }

The issue constraint section is composed of one or more constraints. The EBNF syntax for constraints is:

Constraint → *InstructionRegisterName* **`issues to`**
Resource { , Resource }
| *InstructionRegisterName* **`issues`** *operation*
Operation { , Operation }

Constraints are expressed with instruction registers and one of two types of issue constraints. The first constraint names the operations that may be issued from an instruction register. The second constraint names the functional unit (or multiplexer) which will be used by the operations issued from the instruction register.

To illustrate the use of issue constraints, the UltraSPARC processor is used. The UltraSPARC processor can issue a group of up to four instructions per cycle. The following rules apply to the instruction groups:

- integer instructions can only issue from the first three slots,
- floating-point instructions can issue from any slot,
- only floating-point, NOPs or branch instructions may be issued from the fourth slot, and
- loads and stores can only issue from the first three slots.

The UltraSPARC processor is modeled with two integer functional units, **`alu0`** and **`alu1`**, two floating-point units, **`falu0`** and **`falu1`**, one load/store unit, **`memory`** and a multiplexer, **`pc_mux`**, for guarded assignments (branches).

The UltraSPARC instruction issuing constraints are specified as follows:

```
issue constraints:
   ir[0] issues to alu0, alu1, falu0,
      falu1, pc_mux, memory
   ir[1] issues to alu0, alu1, falu0,
      falu1, pc_mux, memory
   ir[2] issues to alu0, alu1, falu0,
      falu1, pc_mux, memory
   ir[3] issues to falu0,falu1,pc_mux
```

3.5 Embedded pipelines

In the pipeline schematic of Figure 3, the details of the operation of the floating-point pipeline were not shown. This omission is fairly common. Typically, floating-point pipelines are complex and consequently processor reference manuals often avoid discussing the details. However, manufacturers will publish floating-point operation latencies, hence the `^` annotation.

When details about the floating-point pipeline are provided, they are usually presented by giving a separate description of the floating-point pipeline. Again, this is to avoid complicating the schematic so much that it would be difficult to determine the operation of the pipeline.

When information about the operation of the floating-point pipeline is available, we would like to describe the pipeline so that we obtain the resource vectors that model the pipeline as accurately as possible. Drawing from the approach often used by manufacturers, *pipe* allows pipeline descriptions to be composed via embedding. That is, a pipeline description can contain another pipeline description and so on. This mechanism allows us to describe a complex floating-point pipeline separately, yet take its effect into account when constructing the resource vectors.

To illustrate, we consider floating-point division. The previous *pipe* specification gave the latency as 19 cycles. This is indeed the latency of the instruction, however, it was given by specifying that it used resource `falu` for 19 cycles. This is incorrect. In fact, the floating-point division instruction uses the `falu` for one cycle, a special division unit (`div`) for 15 cycles, and the `falu` again for three cycles for a total of 19 cycles. Specifying that the instruction uses the `falu` for all 19 cycles reduces the amount of concurrency. That is, another floating-point instructions could execute concurrently with the floating-point divide instruction (i.e., the ALU can be used while the divide instruction is proceeding through the division unit).

To describe the floating-point divide instruction accurately, we indicate that the operation is pipelined. The previous description of the `falu` resource is changed to be:

```
functional unit falu :
   '/.d'^pipelined,'/.s'^12, '==.s', CV^3,
   '*.s'^4, '*.d'^5,'+.s', '+.d'^2 ;
```

That is, we remove the latency and add the keyword `pipelined`. The keyword specifies that this operation is implemented by a pipeline.

The description of the pipeline is added at the end of the top-level *pipe* specification. The name of the pipeline is formed by concatenating the name of the containing resource (`falu`) with the operator name (`'/.d'`).

```
embedded pipeline falu:'/.d'
      stages
         1: falu
         2: div^15
         3: falu^3
```

With the addition of the above code, the *pipe miner* processor will produce the following resource vector for the floating-point divide instruction:

```
/.d   falu div^15 falu^3
```

The next section describes how a *pipe* specification is processed by *pipe miner* to produce resource vectors.

4 *PIPE MINER*

A common technique for implementing both prototypes and production implementations of DSL compilers is to compile the DSL to an existing high-level language. Many DSLs are compiled to C or C++ because of their high availability. For example, *lex* [19], *yacc* [15], *hancock*[6], among others, are compiled to C. Because our ultimate goal is to produce a compiler that will process instructions and produce resource vectors, we compile *pipe* specifications to a *yacc* grammar. The resulting *yacc* grammar is used to generate a compiler that processes the instruction set of the processor and produces resource vectors. Thus *pipe miner* is a program generator for a compiler-compiler.

4.1 Pipeline graph translation

Figure 5 illustrates the process of translating a *pipe* specification to an appropriate *yacc* grammar. *pipe miner* reads a *pipe* specification and for each embedded pipeline, and the top-level pipeline, builds a pipeline graph data structure. Building the pipeline graph data structure is straightforward.

After constructing the pipeline data structures, *pipe miner* performs semantic checks on the graph. It verifies that each graph is connected and that the graphs corresponding to embedded graphs are connected to the top-level graph. It also verifies that each resource has the proper number of inputs and the types are correct.

After verifying the pipeline graph, *pipe miner* processes the set of graphs in a bottom-up fashion—lower-level embedded graphs are processed before the graphs that contain them. Each pipeline graph is traversed breadth first. As *pipe miner* visits each node in the graph, it emits a grammar rule and action corresponding to the visited node. For example, the grammar fragment emitted for a functional unit resource has the form

```
alu_name : alu_in1 op alu_in2
         | op alu_in1
```

where `op` is the set of operations the alu can perform (which was given in the *pipe* specification). The productions for the

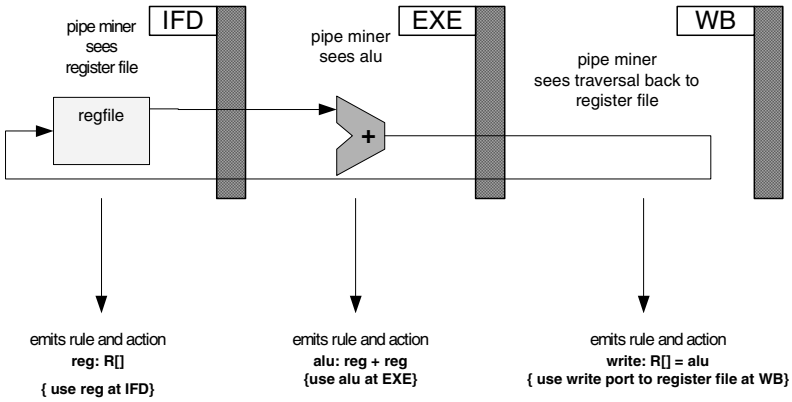

Figure 5: *pipe miner* compiler specification generator emitting *yacc* grammar rules and actions for a *pipe* specification.

nonterminals `alu_in1` and `alu_in2` would be produced when the resources that produced the inputs to the alu were processed.

For the example in Figure 5, *pipe miner* emits the grammar rule

```
alu :  reg '+' reg { rv[exe] = rv[exe]|alu;}
```

Other resources are processed in a similar manner. Embedded pipelines are handled by producing grammar chain rules that link the "start" symbol of the embedded pipeline to the grammar for the top-level pipeline. The start rule for the emitted grammar is simply all possible assignments to all storage resources. *yacc* processes the *pipe miner*-emitted grammar, and yields a parser that is linked with a support library to produce a compiler.

Compilation of instructions to produce resource vectors is shown in Figure 6. The compiler processes instructions expressed in register transfer lists (RTL) [4, 5]. The list of instructions to process is extracted automatically from our Computer System Description Language (CSDL) description of the instruction set[2]. As each instruction is parsed, the resources that the instruction uses are identified, and the appropriate semantic action is executed to place the resource in the correct stage of the resource vector.

There are several types of errors that can occur when parsing an instruction. The most common is not recognizing an RTL. This error is caused when the specified *pipe* cannot implement the instruction. This means that either the pipeline description is incomplete, or the parsed instruction was not a legal instruction. Because we use instructions generated automatically from a machine description, errors are usually due to an incomplete pipeline description.

4.2 Implementation

pipe miner is a literate program of approximately 4500 lines that converts to 2500 lines of C source code [29]. The implementation also includes a graphical editor called *pipelayer* for building pipeline graphs. *pipelayer* allows a user to draw the graph using a template of stencils representing pipeline resources. The tool produces a *pipe* specification which is then processed by *pipe miner*.

5 RELATED WORK

There has been some work on using graphs to model the data path of a processor. Leupers and Marwedel use similar graph modelling and tree rewriting techniques in the context of generating code selectors from hardware description languages [21, 20]. Whereas we allow the user to model the processor at a very high-level of abstraction, they present a very low-level and very general model. In their methodology, the user constructs a full high-level description language (HDL) model of the processor or application specific integrated circuit. The resulting netlist is traversed and register transfer templates, similar to our *yacc* rules, are generated.

Monahan and Brewer model the pipeline using a graph with latch, register, multiplexer and function unit nodes [25]. Using RT-level symbolic data path execution, they perform exact scheduling of hardware DFGs on a pre-existing data path.

Gupta and Önder's Architecture Description Language project seeks to develop microarchitecture descriptions for producing many systems tools, such as cycle-level simulators and assemblers [27]. In this sense, it is similar to our CSDL project. Their pipeline descriptions are procedural and not designed for analysis.

The Trimaran compiler infrastructure uses IMPACT's machine description, HMDES, to describe a processor's pipeline [11, 12, 10]. Using a Lisp-like notation that is similar to gcc's machine description, users create instruction categories and specify reservation tables for these categories. A HMDES specification is translated to a form that can be incorporated into the compiler. HMDES pipeline specifications are much lower level than *pipe* specifications. The low-level nature of Trimaran's description makes them harder to write, but it does afford users the full power of using reservation tables directly.

The HAWK microarchitecture specification language has been developed to describe, simulate and verify processors [24]. This

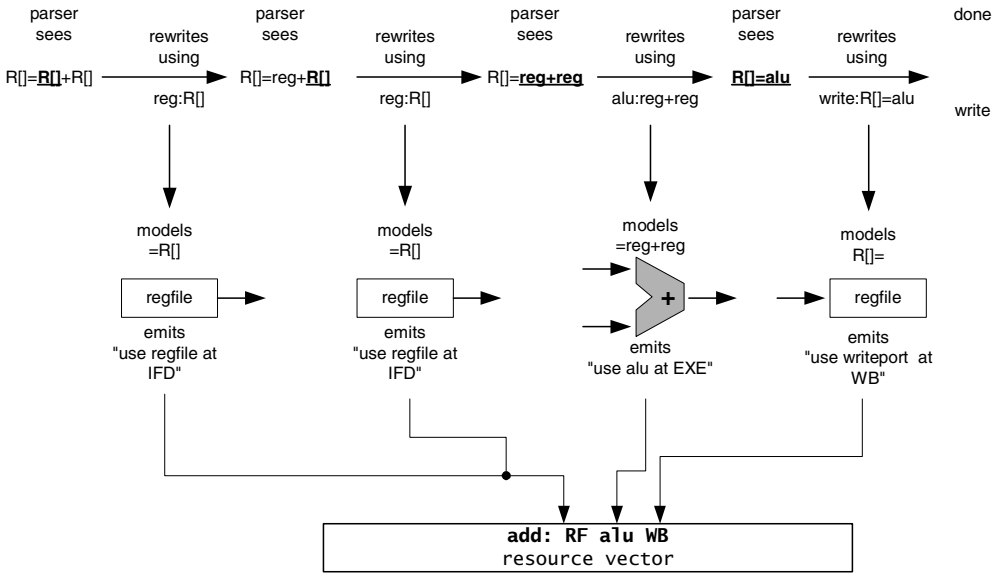

Figure 6: Compiling instructions to resource vectors.

work emphasizes verification and simulation, rather than system tool generation.

Several architectural description languages exist for system level design space exploration. These languages, such as Checkers [23], EXPRESSION [9, 8] and Mimola [22] are used in toolkits that contain a series of circuit design tools and program generators for automatically generating retargetable compilers and simulators. These systems generate compilers with their own instruction scheduling mechanisms based on the low-level description of the processor pipeline. Only EXPRESSION has a system for generating resource vectors from the descriptions.

6 SUMMARY

In the fast-paced embedded processor development environment, a flexible core is essential. A tool such as *Quick Piping* helps bolster flexibility and speed to market by focusing on automating resource vector generation and providing an abstract, easy-to-change way to describe processors.

Quick Piping provides multiple benefits to the architect:

- A processor pipeline model that uses a directed, partitioned graph for simple, intuitive high-level descriptions.
- A high-level pipeline model that greatly enables reuse and experimentation, in direct contrast to conventional resource vectors, which must be laboriously rewritten, instruction-by-instruction, and can be erroneously transcribed.
- The ability to describe complex pipelines at any level of detail desired.
- A unique mechanism within the model for handling issue constraints.
- *pipe,* a simple, intuitive domain-specific language that yields compact descriptions that correspond to the way compiler writers *think* about processors.
- A DSL compiler, *pipe miner*, that converts high-level *pipe* descriptions into low-level resource vectors automatically. Users get the benefit of thinking at a high level and implementing small modifications at that level that generate significant changes at the low-level.
- The marriage of pipeline description to instruction description. The *pipe miner* compiler actually guarantees this coupling occurs—an advancement beyond handwritten resource vectors where no such “linkage control” is possible.
- A high-level tool that, unlike resource vectors, automates description error identification, reducing time spent on debugging.

7 REFERENCES

[1] Jean-Loup Baer. *Computer Systems Architecture*. Computer Science Press, 11 Taft Court, Rockville, MD, USA, 1980.

[2] Mark W. Bailey. *CSDL: Reusable Computing System Descriptions for Retargetable Systems Software*. PhD thesis, University of Virginia, May 2000.

[3] Vasanth Bala and Norman Rubin. Efficient instruction scheduling using finite state automata. *International Journal of Parallel Programming*, 25(2):53–82, April 1997.

[4] C. G. Bell and A. Newell. *Computer Structures: Readings and Examples*. McGraw–Hill, New York, 1971.

[5] Manuel E. Benitez and Jack W. Davidson. The advantages of machine-dependent global optimization. *Lecture Notes in Computer Science*, 782:105–110, 1994.

[6] Dan Bonachea, Kathleen Fisher, Anne Rogers, and Frederick Smith. Hancock: A language for processing very large-scale data. In *Proceedings of the 2nd Conference on Domain-Specific Languages*, pages 163–176, Berkeley, CA, October 3–5 1999. USENIX Association.

[7] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. McGraw Hill, 1992.

[8] P. Grun, A. Halambi, N. Dutt, and A. Nicolau. RTGEN: An algorithm for automatic generation of reservation tables from architectural descriptions, 1999.

[9] Peter Grun and et al. Ashok Halambi. EXPRESSION: An ADL for system level design exploration. Technical Report 98-29, Department of Information and Computer Science, UC Irvine, Irvine, California, September 1998.

[10] John C. Gyllenhaal. A machine description language for compilation. Master's thesis, 1994.

[11] John C. Gyllenhaal, Wen mei W. Hwu, and B. Ramakrishna Rau. Hmdes version 2.0 specification. Technical report, University of Illinois at Urbana-Champaign, 1996.

[12] John C. Gyllenhaal, Wen mei W. Hwu, and B. Ramakrishna Rau. Optimization of machine descriptions for efficient use. *International Journal of Parallel Programming*, 26(4), 1998.

[13] Joe Heinrich. *MIPS R4000 Microprocessor User's Manual*. Prentice-Hall PTR, Upper Saddle River, NJ, USA, 1993.

[14] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach, 2nd ed.* Morgan Kaufmann Publishers Inc., Palo Alto, CA, 1995.

[15] S. C. Johnson. YACC: Yet Another Compiler Compiler. *Computing Science TR*, 32, 1975.

[16] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice Hall, 1992.

[17] P. Kogge. *The Architecture of Pipelined Computers*. McGraw Hill Book Company, New York, NY, 1981.

[18] Eleftherios Koutsofios. Editing graphs with *dotty*. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, USA, July 1994.

[19] M. E. Lesk. LEX - A lexical analyzer generator. *Computing Science TR*, 39, October 1975.

[20] R. Leupers. Retargetable code generation for digital signal processors, 1997.

[21] Rainer Leupers and Peter Marwedel. Retargetable Generation of Code Selectors from HDL Processor Models. *European Design and Test Conference*, 1997.

[22] P. Marwedel. The MIMOLA design system: Tools for the design of digital processors. In *ACM IEEE 21st Design Automation Conference*, pages 587–593, Los Angeles, Ca., USA, June 1984. IEEE Computer Society Press.

[23] Peter Marwedel and Gert Goossens. *Code Generation for Embedded Processors*. Kluver Academic Publishers, Boston, 1995.

[24] J. Matthews, J. Launchbury, and B. Cook. Microprocessor specification in Hawk, 1998.

[25] C. Monahan and F. Brewer. Symbolic modeling and evaluation of data paths. In *32nd Design Automation Conference Proc.*

[26] T. Müller. Employing Finite Automata for Resource Scheduling. In *Proceedings of the 26th Annual International Symposium on Microarchitecture (MICRO'93)*, pages 12–20, Los Alamitos, CA, USA, December 1993. IEEE Computer Society Press.

[27] Soner Önder and Rajiv Gupta. Automatic generation of microarchitecture simulators. In *Proceedings of the 1998 International Conference on Computer Languages*, pages 80–89. IEEE Computer Society Press, 1998.

[28] Todd A. Proebsting and Christopher W. Fraser. Detecting pipeline structural hazards quickly. In *Conference Record of POPL '94, 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 280–286, New York, NY, USA, 1994. ACM Press.

[29] Norman Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, September 1994.

[30] Richard L. Sites. *Alpha Architecture Reference Manual*. Digital Press and Prentice-Hall, 1992.

[31] SPARC International, Inc. *The SPARC Architecture Manual - Version 8*. Prentice-Hall, Upper Saddle River, NJ, USA, 1992.

[32] H. Stone. *High Performance Computer Architecture*. Addison-Wesley, New York, 1987.

[33] Sun Microelectronics. *UltraSPARC I&II*. Sun Microelectronics, 1997.

[34] MIPS Technologies. *MIPS32 4Kp Datasheet*. MIPS Technologies, 2001.

[35] MIPS Technologies. *MIPS64 5K Processor Core Family Software Users Manual*. MIPS Technologies, 2001.

[36] MIPS Technologies. *MIPS64 5Kc Processor Core Datasheet*. MIPS Technologies, 2001.

[37] MIPS Technologies. *MIPS64 5Kf Processor Core Datasheet*. MIPS Technologies, 2001.