# Linguagens para Descrição de Arquiteturas de Computadores

Mariza Andrade da Silva Bigonha[1]
José Lucas Mourão Rangel Netto[2]

**RESUMO**

Arquiteturas mais modernas de computadores motivam pesquisas por técnicas mais eficazes de implementação de compiladores capazes de gerar código de alta qualidade. As arquiteturas superescalares têm conjuntos reduzidos de instruções, cuja combinação para execução eficiente deve ser feita pelos "back-ends" dos compiladores usualmente de maior complexidade que seus correspondentes para arquiteturas CISC. Os objetivos deste trabalho são o estudo das características de linguagens formais de descrição de arquiteturas, apropriadas para uso com geradores de geradores de código, e o projeto e a validação semântica de uma linguagem de descrição de arquiteturas apropriada para uso com o gerador de geradores de código projetado.

**ABSTRACT**

Modern computer architectures lead to research looking for better techniques for effective implementation of compilers which must be able to produce high-quality code. In the special case of superscalar architectures we have reduced instruction sets, and instructions must be combined to warrant efficient execution by the compilers, which have more complex back-ends than their CISC counterparts. Objectives of this work are the study of the characteristics of formal languages for the description of architectures, meant for use with code generator generators and the project, and the semantic validation of an architecture description language adequate for use with the generator system.

## 1 Introdução

Qualquer discussão sobre geração de código deve considerar a arquitetura alvo e, isso introduz a necessidade de um mecanismo adequado para especificá-la. As linguagens de descrição

[1] Mariza Andrade da Silva Bigonha é mestre em Ciência da Computação pela UFMG e doutora em Informática: Ciência da Computação pela Pontifícia Universidade Católica do Rio de Janeiro. É analista de sistemas do Departamento de Ciência da Computação da UFMG. Áreas de interesse: linguagens de programação, compiladores, arquitetura.
E-mail:mariza@dcc.ufmg.br

[2] José Lucas Mourão Rangel Netto é mestre em Engenharia Elétrica pela COPPE/UFRJ e doutor em Ciência da Computação pela Univ. of Wisconsin, E.U.A. É professor do Instituto de Matemática da UFRJ e Professor Associado do Dep. de Informática da PUC/RJ. Áreas de interesse: engenharia de software, compiladores, linguagens de programação, teoria da computação.
E-mail: rangel@inf.puc-rio.br

de arquiteturas (LDA) desempenham este papel. Muito embora sejam encontrados na literatura sistemas geradores de geradores de código que as utilizam, as linguagens existentes não descrevem os principais processadores e, quando o fazem, não são capazes de especificar suas características mais complexas. Uma das qualidades mais almejadas em uma linguagem para descrição de arquitetura para um sistema redirecionável é a completeza. Completeza significa que a linguagem é capaz de modelar todas as máquinas existentes dentro de uma classe particular.

Nas arquiteturas tradicionais, o conjunto de instruções é caracterizado por sua complexidade, valhendo-lhes o nome CISC (*Complex Instruction Set Computers*). Nas arquiteturas mais recentes o conjunto de instruções é caracterizado pela simplicidade, cabendo-lhes o nome de RISC (*Reduced Instruction Set Computers*). Muito embora exista um grande volume de publicações relacionadas com as arquiteturas CISC e RISC, ainda não foi desenvolvida uma teoria bem fundamentada, como a que existe para *front-ends* de compiladores, sobre a qual possam se basear os projetistas de *back-ends*, especialmente aqueles voltados para arquiteturas superescalares. As arquiteturas superescalares são uma evolução dos processadores RISC. Estas arquiteturas possuem várias características em comum. As principais são a habilidade de executar mais de uma instrução por ciclo e a incorporação de várias unidades funcionais que podem operar em paralelo. A maior vantagem desta última característica é a habilidade de explorar o paralelismo em nível de instrução, pela execução simultânea de instruções em unidades individuais. Percebe-se, inclusive, que há discordância entre pesquisadores da área em relação a determinados enfoques. Por exemplo, do ponto de vista das linguagens de descrição de arquiteturas, existem duas correntes. Uma delas defende o princípio de que devem ser incorporadas às linguagens primitivas para auxiliar o escalonamento de instruções[3] [Wall and Powell, 1987, Bradlee et al., 1991]. A outra corrente defende o princípio de que estas primitivas não precisam ser incorporadas explicitamente nas linguagens de descrição de arquiteturas para que o escalonamento de instruções seja feito de forma eficaz [Benitez and Davidson, 1988, Stallman, 1989, Benitez and Davidson, 1991].

Uma das motivações para esta pesquisa advém do fato de que não existe hoje um sistema baseado na descrição da arquitetura de máquina que incorpore, de forma satisfatória, restrições sobre o escalonamento, ou mesmo que construa um escalonador de instruções a partir da descrição da máquina. Assim sendo, linguagens de descrição de arquiteturas caracterizam um problema que ainda não tem uma solução satisfatória. Consideramos que uma linguagem deste tipo tendo em vista a geração de código deve, em princípio, possuir as seguintes propriedades: (1) incorporar mecanismos para definição dos modos de endereçamento; (2) permitir definição completa da semântica das instruções, incluindo mecanismos para atualização do código de condição; (3) ser concisa e com alto poder de expressão [Bigonha, 1990]. As questões envolvendo informações sobre as características inerentes dos processadores superescalares ainda merecem estudos mais profundos. Em particular, as seguintes questões devem ser levantadas em relação à linguagem de descrição de arquiteturas proposta para esta nova gama de processadores:

- avaliar a necessidade de incorporar na linguagem de descrição de máquina facilidades

[3] Escalonamento de instruções é uma técnica de *software* que rearranja seqüências de código durante a compilação com o objetivo de reduzir possíveis atrasos de execução.

para descrever, além das instruções e conjuntos de registradores, as unidades funcionais, os estágios da *pipeline* e outras propriedades do escalonamento como por exemplo, a latência e custo das instruções etc.;
- incorporar informações sobre a geração e otimização de código nesta linguagem;
- identificar classes de arquiteturas que podem ser descritas com a nova linguagem de descrição de máquina.

Um dos objetivos deste trabalho é apresentar soluções para estas questões. Precisamente, pretendemos demonstrar a viabilidade de definir uma linguagem de descrição de máquina concisa que seja capaz, contudo, de especificar diferentes famílias de processadores superescalares. Para demonstrar que alcançamos nosso objetivo, propusemos e definimos formalmente a sintaxe e semântica de uma linguagem para descrever o conhecimento embutido nas arquiteturas de computadores e projetamos um gerador de código para a arquitetura SPARC [SUN Microsystems, 1987, SUN Microsystems, 1989], escolhida como exemplo para demonstrar o poder de expressão da linguagem desenvolvida.

## 2 As Famílias de LDAs

Sucintamente, apresentamos um estudo comparativo das diversas linguagens existentes para descrever arquiteturas de máquinas CISCs, tendo em vista o projeto de uma nova LDA para arquiteturas superescalares. Iniciamos apresentando os principais trabalhos dentro de quatro famílias de geradores de código (veja Figura 1), encontrados atualmente na literatura, que se utilizam de linguagens de descrição de arquiteturas para obter o resultado almejado.

Glanville e Graham [Graham and Glanville, 1978, Graham et al., 1982] construíram o primeiro sistema redirecionável de gerador de código. Seu sistema tem como entrada a descrição da máquina em forma de gramática e produz como resultado um gerador de código que utiliza técnicas de análise sintática LR para efetuar o casamento de padrão com a linguagem intermediária. As produções da gramática são compostas de padrões, que englobam os símbolos terminais e não-terminais, e de um lado esquerdo que é formado por um símbolo não-terminal. Ações associadas às produções dirigem a geração do código de máquina.

O sistema CODEGEN de Robert Henry [Aigrain et al., 1984, Henry and Damron, 1989a, Henry and Damron, 1989b] é um sucessor do sistema de Graham-Glanville. Esse sistema tem como entrada PPRACs (padrão, predicado, substituição (*replacement*), ação, custo) e produz uma árvore de reconhecimento de padrão. PPRACs se assemelham às produções do sistema de Graham-Glanville, mas são acrescidas de predicados e custos para dirigir o casamento (*matching*).

O trabalho de Paulo Costa [Costa, 1990], intitulado *AutoCode*, é um sistema de produção de geradores de código baseado em reconhecimento de padrões e dirigido por tabelas geradas automaticamente, a partir de uma descrição da arquitetura da máquina alvo na forma de gramática livre do contexto. Seu sistema se assemelha à abordagem de Glanville e Graham.

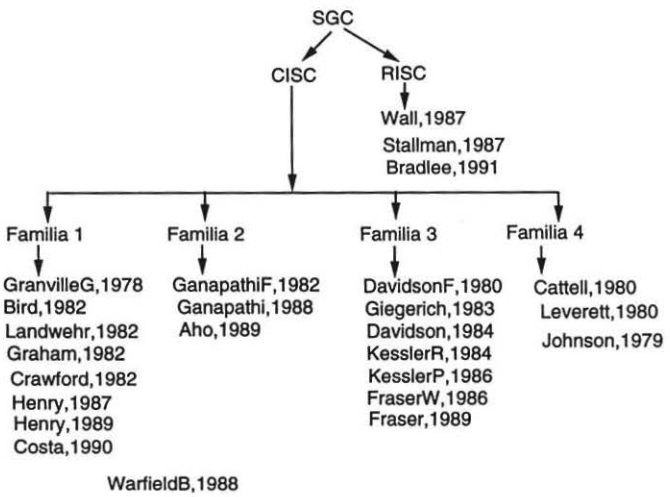


Figura 1: Uso de LDAs em Sistemas de Geração de Código.

Ganapathi e Fischer utilizam gramática de atributos para produzir geradores de código [Ganapathi and Fischer, 1992, Ganapathi and Fischer, 1985]. Para auxiliar a descrição das restrições da arquitetura de máquina, atributos e predicados são associados às produções da gramática. Esta abordagem não enfatiza muito a sintaxe como o sistema de Graham-Glanville. As vantagens atribuídas ao sistema de Ganapathi e Fischer são o tamanho reduzido das tabelas do analisador e a facilidade de capturar os efeitos colaterais. A desvantagem é que o projetista do compilador deve utilizar duas técnicas: gramática e regras de atributos. Um descendente deste sistema [Aho et al., 1989] utiliza programação dinâmica[4] com o algoritmo de reescrita de árvore para gerar código [Aho et al., 1989].

A técnica de descrição utilizada por Davidson [Davidson and Fraser, 1980] provê um método simples, muito embora robusto, para descrever instruções de máquina. A arquitetura é descrita através de uma gramática para tradução dirigida por sintaxe entre a linguagem de montagem da máquina alvo e a transferência de registradores. Detalhes irrelevantes da arquitetura da máquina podem ser omitidos, enquanto que aqueles complicados, mas necessários, podem ser facilmente descritos.

A descrição de Giegerich [Giegerich, 1983] é baseada na formalização da semântica do conjunto de instruções. Ela compreende um modelo abstrato de conjunto de instruções de uma maneira

[4] O princípio do algoritmo de programação dinâmica é particionar o problema da geração de código "ótimo" para uma expressão em sub-problemas para a geração de código "ótimo" para as sub-expressões da expressão dada. Exemplo, dada a expressão $E$ da forma $E_1 + E_2$, um "ótimo" programa para $E$ é formado pela combinação de "ótimos" programas para $E_1$ e $E_2$, em qualquer ordem, seguida de código para avaliar "+"

geral, uma notação para a descrição de máquina real, baseada em *ISP* [Bell and Newell, 1971], além da semântica para tal descrição, que é apresentada em termos do modelo abstrato. Este enfoque resulta em uma descrição de instruções simples, entretanto a forma utilizada para descrever os modos de endereçamento é complicada e trabalhosa.

O projeto do PQCC (*Production-Quality Compiler-Compiler*) é um descendente do compilador Bliss-11 de Wulf [Wulf et al., 1975]. Este projeto foi muito ambicioso: PQCC tentou incluir uma gama de detalhes muito minuciosa na linguagem para descrever a arquitetura da máquina alvo, de tal forma que compromissos entre espaço e tempo na otimização, seleção de código e alocação de registradores pudessem ser examinados. Devido ao fato de ter tentado incorporar muitos detalhes da máquina alvo, o sistema ficou difícil de ser utilizado.

A abordagem utilizada por Warfield e Bauer [Warfield and Bauer, 1988] para descrever arquiteturas de máquinas é compacta, preenche todos os requisitos necessários em uma linguagem de descrição, inclusive mecanismos para definir regras. Entretanto, depende de uma ferramenta, *MRS* [Russell, 1985], que está disponível somente em três máquinas: *DEC-20* rodando *MacLISP*, *VAX* rodando *FranzLISP* sobre o *UNIX* de Berkeley e "*Symbolics LISP machine LM-2/3600*", o que torna o seu uso inviável.

A tabela a seguir mostra uma avaliação destes métodos.

| | descrição máquina | método usado | redirecionamento |
|---|---|---|---|
| GLANVILLE | muito extensa incompleta | gramáticas livres do contexto LR | difícil avaliar dependência das linguagens:fonte e descritiva |
| GANAPATHI | longa, difícil e incompleta | gramáticas de atributos | difícil avaliar arquiteturas semelhantes |
| CATTELL | mais complexa, equipara-se a Ganapathi em complexidade | definição na forma de asserções de entrada e saída associada a cada instr. | distribuição de etapas em outras partes do compilador |
| COSTA | muito longa | gramáticas livres do contexto | dependência de máquina da repres. intermediária embutida no frontend |
| DAVIDSON | fácil de ler, flexível | expressão regular gramáticas livre contexto-lex,yacc | mais fácil |
| WARFIELD | compacta, mas depende de MRS | regras | difícil avaliar |
| GIEGERICH | simples porem forma utilizada para descrever modos de endereçamento complicada, trabalhosa | adaptação de ISP | restringe somente a MC86000, 8086 |
| HENRY | razoável | enumeração de cinco tuplas PPRACS ou regras | difícil avaliar |

Para especificar as características dos processadores RISC, linguagens de descrição de arquiteturas com as propriedades descritas até agora não são suficientes. Uma linguagem para

estas máquinas deve possuir mecanismos que lhe permitam tratar o escalonamento de instruções de forma satisfatória. Os sistemas que se aplicam a estas arquiteturas são poucos (veja Figura 1), contudo eles incluem informações sobre o escalonamento de instruções de alguma forma. O sistema Mahler de Wall e Powell [Wall and Powell, 1987] utiliza informações sobre escalonamento, mas não possui uma linguagem de descrição de arquitetura explícita. Mahler utiliza uma linguagem de montagem para uma máquina virtual sem *pipeline* e com um número infinito de registradores. Detalhes sobre escalonamento estão contidos dentro do tradutor de Mahler e, portanto, uma substituição na arquitetura de máquina implica em substituições manuais no tradutor. O compilador GNU C [Stallman, 1989] é considerado um compilador redirecionável bem sucedido. GNU utiliza descrições interpretadas de máquina, relembrando o sistema PO de Davidson e Fraser. Muito embora GNU C tenha sido transportado para várias arquiteturas RISCs, a descrição de máquina não contém informações sobre escalonamento. Tiemann escreveu um escalonador para GNU C [Tiemann, 1989] onde latências dependentes da arquitetura alvo e informações sobre recursos computacionais são encapsulados. Entretanto, ele não indica em seu trabalho a forma do encapsulamento. Bradlee [Bradlee et al., 1991] projetou o protótipo de um sistema de geradores de código. Este sistema é o único que possui uma linguagem de descrição de máquina embutida, entretanto as primitivas de sua linguagem descrevem apenas os componentes básicos dos processadores RISC. Ela não modela várias das características típicas das arquiteturas superescalares, como por exemplo: o esquema de prioridade do barramento destino usado em algumas arquiteturas para conter o uso de recursos; o mecanismo de janela de registradores; os efeitos colaterais das instruções, como o acionamento do registrador de condição e, principalmente o despacho múltiplo de instruções.

Analisando estes sistemas, concluimos que é difícil chegar a um consenso sobre qual linguagem é a mais adequada, porque quase todas diferem basicamente, apenas na notação e terminologia adotada, tornando a seleção da "melhor" apenas uma questão de gosto. Contudo certas características presentes ou ausentes em algumas linguagens auxiliam na seleção. Todas elas possuem primitivas para modelar apenas características comuns aos processadores de uma determinada classe de arquitetura. Por exemplo, a linguagem de descrição ISP (*Instruction Set Processor*) [Bell and Newell, 1971] tem sido utilizada nas descrições de arquiteturas CISC. Entretanto, ela não é apropriada para processadores RISC, porque contém mais detalhes do que realmente é necessário em algumas áreas, enquanto pouco detalhe em outras. Para o gerador de código de uma máquina RISC, uma descrição detalhada do formato da palavra de estado do programa não é a questão mais crucial, sendo, por exemplo, o conhecimento das propriedades de escalonamento de cada instrução mais importante. Árvores e gramáticas também têm sido utilizadas pelos compiladores redirecionáveis para CISCs para descrever as arquiteturas de máquinas. Vários sistemas produzem sofisticados métodos de seleção de instruções para CISCs. Até esta data, a única linguagem de descrição existente projetada por [Bradlee et al., 1991] para atender máquinas RISC é deficiente para esta classe de processadores. Como mostrado nesta seção, ela só modela características básicas dos processadores RISCs.

## 3 A LDA para Arquiteturas Superescalares

Considerando que uma linguagem de descrição de máquina serve como veículo para especificar o comportamento de uma máquina alvo, é importante que a mesma possua mecanismos para definir o máximo de informações possível sobre uma dada arquitetura. Dependendo da aplicabilidade da descrição de máquina, é fundamental que ela seja capaz de especificar outras características da máquina, além de seu conjunto de instruções e modos de endereçamento. Por exemplo, a inclusão de facilidade para definir custo relativo a tempo e espaço é característica importante, quando o objetivo é otimizar o código gerado. O conhecimento do custo é necessário para dirigir o processo de otimização. Outras informações, como por exemplo, mecanismos para atualizar o código de condição utilizado nas operações lógico-aritméticas, tornam as notações mais poderosas.

A LDA para a especificação de arquiteturas de máquinas superescalares que apresentamos neste trabalho leva em consideração as questões levantadas na Seção 1 e incorpora, de forma que consideramos satisfatória, mecanismos para definição dos modos de endereçamento, facilidades para descrever as unidades funcionais, os estágios do *pipeline* e outras propriedades para o escalonamento; permite definição completa da semântica das instruções; incorpora informações sobre a geração e otimização de código; identifica as classes de arquiteturas que podem ser descritas e é concisa e possui alto poder de expressão. Ela é uma adaptação da linguagem proposta por Bradlee [Bradlee et al., 1991], incorpora as características básicas desta linguagem, mas inclui novas facilidades, amplia seu leque de abrangência e a torna mais robusta. Ela é capaz de especificar os pontos abaixo relacionados necessários para descrever arquiteturas superescalares.

1. Características gerais das arquiteturas:
   - Registradores.
   - Estágios de *pipeline*.
   - Unidades funcionais.
   - Memória.
2. Modelo da máquina virtual.
   - Definição do uso dos registradores.
   - Passagem de parâmetros.
3. Lista de Instruções:
   - Instruções padrão para cada instrução de máquina deve ser especificado:
     - Mnemônico da instrução.
     - Restrições de tipo dos operandos.
     - Semântica da instrução.
     - Recursos de *pipeline* necessários.
     - Outras propriedades de escalonamento, como por exemplo. latência, custo e número de *slots* associado às operações.
   - Instruções especiais.
4. Detalhes específicos de arquitetura:

- Transformações necessárias para auxiliar no mapeamento da linguagem intermediária com o conjunto de instruções da máquina alvo.
- Especificações de latência especiais.

## 3.1 Estrutura da LDA

Para maior clareza, a LDA é composta de três seções: seção de declarações, seção com as características da máquina alvo e seção de instruções. A especificação das declarações deve necessariamente iniciar com a palavra "**declarations** {" e finalizar com "}" . Nesta seção são declarados os registradores, os recursos da máquina, as constantes, o tamanho da memória disponível entre outras informações. A seção **vm** especifica as características da máquina alvo. Esta seção inicializa com a palavra "**virtual_machine**" seguida do símbolo "{" e finaliza com o símbolo "}" . A seção de instruções introduz a descrição de instruções da máquina, instruções especiais e transformações necessárias para casar padrões da linguagem intermediária com padrões da linguagem da máquina alvo. Esta seção inicializa com a palavra chave "**instructions**" seguida do símbolo "{" e finaliza com o símbolo "}" .

### 3.1.1 Seção de Declarações

Esta seção consiste em uma lista de definições que especificam as características da arquitetura alvo. Para cada definição existe uma palavra iniciada com "%" que a identifica. A Figura 2 mostra um trecho da descrição da seção de declarações. São declaradas nesta seção:

- Definição de registradores: conjuntos de registradores disjuntos para operações de ponto flutuante e inteiros ou um conjunto de registradores compartilhado entre estes dois tipos de operações.
- Sobreposição de registradores: através da diretiva %equiv, o projetista do compilador descreve a sobreposição do conjunto de registradores. Esta facilidade pode ser usada em arquiteturas que possuem um conjunto de registradores compartilhado para as operações com inteiros e ponto-flutuante.
- Recursos da máquina: declara os recursos computacionais da máquina, por exemplo, estágios da *pipeline*.
- Constantes: define um nome para representar um valor dentro de um intervalo.
- Memória disponível: define um arranjo contendo as localizações de memória.
- Definição de Rótulos: define um nome para representar um rótulo com endereço relativo.
- Declaração de Relógios: define um nome para ser um relógio. Este relógio acompanha o processamento de sub-operações durante o escalonamento temporal[5] em arquiteturas que possuem *pipeline de avanço explícito*[6]

[5] Escalonamento temporal é o processo de acompanhar a colocação dos resultados das sub-operações em registradores temporários.

[6] *Pipeline de avanço explícito* é uma *pipeline* que retém seu estado até que instruções de um conjunto em particular sejam executadas.

- Definição de Classes: define classe para ser o conjunto de elementos previamente definidos.
- Associação de Instruções a Relógios: define um nome que representa uma instrução como um elemento pertencente a um conjunto classe.

```
declarations
{
  %reg r[0:31] (charlintlshortlpointer);
  %reg i[0:7]  (charlintlshortlpointer);
  %reg f[0:31] (floatlint);
  %reg d[0:15] (double);
  %clock clk-cc;
  %reg cc(int; clk-cc) +temporal;

  %equiv i[0]    r[24];

  /* Integer Unit Stages  (CY7C601) */
       %resource IF;  /* instruction fetch */
       %resource ID;  /* decode */

  %memory m[0:4294967295];

  %def uns-const[65536:2147483647]  +relocatable;
   %label rel-lab[-4194304:4194303]  +relative
}
```

Figura 2: Trecho da Seção de Declarações para uma dada Arquitetura

Informações coletadas das diretivas da seção de declarações são colocadas em uma matriz e posteriormente utilizadas nas rotinas para construir o DAG de código[7], para gerar código, para efetuar reconhecimento de padrões, para alocar registradores, para fazer transformações e para escalonar instruções.

### 3.1.2 Seção de Definição da Máquina Alvo

A seção (`vm`) descreve o modelo de execução da máquina à qual o código gerado deve corresponder. Seus objetivos são, basicamente: especificar as convenções das chamadas de procedimento; definir o uso dos registradores. Por exemplo, na SPARC e na maioria das arquiteturas, durante a entrada de um procedimento, um registrador, o apontador de *frame*, é estabelecido para apontar para a base do registro de ativação corrente na pilha. Todas as referências a registradores locais utilizam índices a partir dele, de modo que os registradores locais mantêm

[7] DAG de código é a estrutura de dados usada no processo de escalonamento de instruções. Representa-se nesta estrutura de dados os blocos básicos[8] em que foi dividido o programa.

os mesmos deslocamentos, mesmo se o apontador de pilha variar durante a execução do procedimento. Na máquina `MIPS` não existe um apontador de *frame*. Os registradores locais são endereçados relativamente ao `sp`. De posse destas informações e algumas outras apresentadas nesta seção é, portanto, possível descrever o funcionamento da máquina alvo. A Figura 3 mostra um trecho da especificação da máquina alvo para uma determinada arquitetura. As declarações e definições permitidas nesta seção são:

- Declaração de pilhas: %sp define o registrador usado como apontador de pilha.
- Definição do registro de ativação: %fp define o registrador usado como apontador do registro de ativação.
- Definição da área global: %gp representa o apontador global e a área usada.

```
virtual-machine
{
    %general (charlshortlintlpointer) i;
    %general (charlshortlintlpointer) r;
    %general (float) f;
    %general (double) d;

    %hard r[0] 0;

    %sp r[14] +up;
    %fp r[30] +down;
    %gp r[2] 65536;

    %allocable r[1,3:13,15:29,31];
    %calleesave r[2,14,30];
    %callersave r[15:23];

    %arg (int) r[8] 1; /* out registers */
    %arg (int) r[15] 6;

    %par (int) r[24] 1; /* in registers */
    %par (int) r[29] 6;

    %retaddr r[31];

    %result r[8] (int);
}
```

Figura 3: Trecho da Especificação da Máquina Alvo para uma dada Arquitetura

- Declaração de registradores de propósito geral: %general indica os registradores de propósito geral para um dado tipo.
- Declaração dos registradores alocáveis: define através da diretiva %allocable os registradores que podem ser usados pelo alocador de registradores.

- Declaração de registradores com valores pré-definidos: %hard define registradores que contêm valores especificados pelo *hardware* da arquitetura em questão.
- Definição dos registradores preservados: define registradores com a diretiva %calleesave que devem ser preservados pelo procedimento chamado. Na arquitetura SPARC corresponde aos registradores (*out*) declarados pela diretiva %arg.
- Definição dos registradores preservados pelo procedimento chamador: define registradores com a diretiva %callersave que devem ser preservados pelo procedimento chamador. Na arquitetura SPARC corresponde aos registradores (*in*) declarados pela diretiva %par.
- Definição de argumentos específicos: define os argumentos do tipo especificado em %reg.
- Definição de parâmetros específicos: define os parâmetros do tipo especificado em %reg.
- Definição do resultado da operação: %result define o registrador que contém o resultado de acordo com o tipo especificado.
- Definição do endereço de retorno: %retaddr define o registrador que contém o endereço de retorno.
- Definição do método de avaliação de argumento: a idéia da diretiva associada a este item é poder especificar quando expressões devem ser avaliadas ou não. A diretiva usada neste caso é %evalargs.

As informações coletadas das diretivas %calleesave, %callersave, %allocable, %args, %par, %result e %equiv são colocadas em uma matriz para serem usadas na geração de código, durante o escalonamento de instruções, na construção do DAG de interferência[9] de registradores e essencialmente na passagem de parâmetros. As demais diretivas são utilizadas durante a execução da máquina alvo.

### 3.1.3 Seção de Definição de Instruções

O objetivo desta seção é descrever as instruções da máquina alvo e suas funções. A seção de definição de instruções é composta de dois tipos de instruções: as instruções comuns e as instruções especiais. Cada diretiva %instr descreve uma instrução em cinco partes. A Figura 4 apresenta a sintaxe de uma instrução. A primeira parte especifica o mnemônico e os operandos da instrução. A segunda parte, entre parênteses, especifica, opcionalmente, as restrições de tipos dos operandos. A terceira parte, entre chaves, define a semântica da instrução. A quarta parte, entre colchetes, define os recursos utilizados pela instrução. A quinta parte, entre parênteses, define informações sobre o custo e os ciclos gastos pela instrução, etc.

Incluem-se na categoria de instruções especiais as instruções para movimentação entre registradores, as declarações de instruções sem operação, as declarações das instruções "glue" e as definições de instruções para especificar uma nova latência. As instruções especiais auxiliam

[9] DAG de interferência é a estrutura de dados usada no processo de alocação de registradores.

```
%instr  fstoi  f, f  ($1 == float & $2 == int)
   {$2 = int($1); }
   [IF; ID; 5*IE; IW; ]
   (1,5,0)
```

Figura 4: Exemplo de uma Instrução em LDA

a compilação. Informações coletadas da diretiva %glue especificam transformações dependentes da máquina alvo. Elas são colocadas em uma tabela e posteriormente usadas durante o reconhecimento de padrão. A Figura 5 mostra um trecho da descrição da seção de declaração de instruções. A partir das informações coletadas nas três seções que compõem a LDA são derivadas várias tabelas. As mais importantes são (1) a tabela contendo os recursos utilizados pelas instruções cujo conteúdo da tabela de recursos é usado essencialmente na rotina básica de escalonamento de instruções, para verificar os conflitos existentes e agrupar instruções; (2) a tabela com informações sobre as instruções. Informações contidas nesta tabela são utilizadas durante a fase de inicializações do compilador e em várias funções do gerador de código. Por exemplo, nas rotinas de escalonamento de instruções, na rotina básica do escalonamento, na construção do DAG de código, na construção do DAG de interferência de registradores, na geração de código, durante a coloração do grafo de interferência, no reconhecimento de padrões, etc.

## 3.2 Escopo de Aplicação de LDA

Completeza é uma qualidade desejável em uma linguagem para descrição de arquitetura para sistemas redirecionáveis. LDA ainda não cobre toda a classe das máquinas superescalares, contudo, ela é capaz de modelar a maior parte das características inerentes destas arquiteturas, abrangendo as características básicas das máquinas RISC. Para ter uma idéia de sua aplicabilidade apresentamos algumas das características mais comuns às arquiteturas superescalares, comercialmente disponíveis. As diretivas de LDA são suficientes para descrever os seguintes aspectos das arquiteturas superescalares:

**Conjunto de registradores compartilhados:**
Na especificação da arquitetura em questão, o projetista do compilador pode declarar vários conjuntos de registradores, bem como registradores com propósitos específicos. Para isto ele tem à sua disposição a diretiva %reg.

**Pares de registradores:**
O projetista do compilador pode declarar através da diretiva %reg um conjunto de registradores para representar o par e indicar que este conjunto se sobrepõe a outro conjunto de registradores utilizando a diretiva de especificação %equiv. As funções de escape, definidas pelo projetista, permitem que se tenha acesso a metade de registradores.

```
instructions
{
 %instr  cmp  r, r, r[0]   (;  clk-cc)
   {cc = ($1 :: $2); }
   [IF; ID; IE; IW;]
   (1, 1, 0)

 %instr be  #rel-lab
    {if cc == 0
          goto $1;}
    [IF; ID; IE;]
    (1, 1, 1)

 %move  mov  i, r
    {$2 = $1;}
    [IF; ID; IE; IW;]
    (1, 1, 0)

  %glue  r,  #uns-const13          (int)
       { ($! == $2) ==> (($1 :: $2) == 0);}
}
```

Figura 5: Trecho da Especificação de Instruções em uma dada Arquitetura

**Janela de registradores:**
A janela de registradores, visível por uma função em particular, é um subconjunto do conjunto total de registradores. O projetista do compilador usa as diretivas %arg e %par para especificar separadamente os argumentos e parâmetros, o que modela a renomeação de registradores, e, utiliza as diretivas %calleesave e %callersave para modelar o salvamento dos registradores especificados.

**Registradores com valores pré-definidos por *hardware*:**
Os registradores com valores pré-definidos por *hardware* são especificados pela diretiva %hard.

**Dependência Estrutural:**
O projetista do compilador especifica as dependências de recursos ou dependências estruturais explicitamente durante a especificação de cada instrução da arquitetura em questão. Durante o escalonamento de instruções verificam-se estes recursos. Isto evita atrasos por dependência estrutural. Os recursos devem ter sido préviamente definidos na seção de declarações através da diretiva %resource.

**Esquema de prioridade por *hardware* para contenção de recursos:**
O projetista do compilador especifica as prioridades dos recursos associando valores numéricos aos recursos pertinentes na seção de declarações e a instrução indica sua pri-

oridade usando o recurso. A prioridade do recurso é verificada durante o escalonamento de instruções.

**Carga de ponto flutuante de precisão dupla usando duas instruções:**
A facilidade das funções de escape permite ao projetista do compilador escrever uma função em linguagem C para gerar uma seqüência de duas instruções, com cada uma delas tendo acesso a uma metade de um registrador de precisão dupla. As duas instruções de carga são então escalonadas como instruções separadas.

**Desvios com *delay slots*:**
Durante a especificação de cada instrução da máquina pertinente o projetista do compilador indica o número de instruções que devem ser colocadas no código para que a instrução de desvio seja executada sem causar atrasos na *pipeline*. Com esta informação *no-ops* são inseridas no código escalonado.

**Desvios com *delay slots* executados condicionalmente:**
O projetista do compilador pode especificar um *delay slot* com valor negativo na diretiva da instrução para indicar que a instrução no *slot* só é executada se o desvio ocorrer.

**Latência de operação dependente do destino da instrução:**
O projetista do compilador especifica um par de instruções com restrições e uma nova latência de operação por meio da diretiva %aux. Caso as restrições sejam satisfeitas, substitui-se o valor do rótulo no vértice do DAG de código entre as duas instruções pelo valor da nova latência.

***Pipelines* com avanço explícito:**
O projetista do compilador pode especificar esta característica da arquitetura `i860` utilizando-se das diretivas %reg, temporal, %clock, %element e %class.

**Registradores de código de condição:**
O registrador de código de condição pode ser declarado a parte como um registrador temporal utilizando as diretivas %reg e temporal. As diretivas das instruções indicam o resultado da comparação utilizando este registrador. Porém, diretivas separadas devem ser especificadas na descrição da máquina para as instruções que ligam o código de condição como um efeito colateral, por exemplo, subtração.

## 4 Ambiente da LDA

A LDA apresentada neste trabalho faz parte de um sistema gerador de geradores de código para arquiteturas superescalares (GGCO). O GGCO recebe como entrada a especificação de uma arquitetura de máquina em LDA e gera uma série de tabelas e funções que representam o resultado do processamento das informações extraídas das principais diretivas de LDA. Estas tabelas e funções constituem a parte dependente da arquitetura em análise.

Este sistema ainda não está totalmente implementado, contudo a especificação da parte dependente de arquitetura está formalmente definida em [Bigonha, 1994]. A definição formal mostra o mapeamento da descrição de arquitetura de máquina para o seu respectivo gerador de código. O resultado final deste mapeamento é um trecho de programa em linguagem C que constitui a parte dependente de máquina do gerador de código para o processador descrito.

## 5 Conclusões

Com o objetivo de projetar uma linguagem para descrição de arquitetura capaz de tratar as construções próprias dos processadores comercialmente disponíveis foi realizado um estudo sobre as linguagens existentes visando definir suas diretivas. Chegamos a conclusão de que deveriam ser incorporada à linguagem primitivas para auxiliar o escalonamento de instruções, como por exemplo, facilidades para descrever as unidades funcionais, os estágios da *pipeline*, informações sobre a geração e otimização de código, além daquelas diretivas para descrever as instruções e conjuntos de registradores.

Dentre os processadores comercialmente disponíveis, procuramos cobrir a classe de arquitetura de computadores denominada superescalar, subtende-se aqui, também os processadores RISCs. Consideramos como base para a especificação da linguagem para descrição de arquiteturas os processadores SPARC. Como resultado deste estudo, projetamos um gerador de geradores de código otimizado que inclui uma linguagem para descrever tais arquiteturas. Esta linguagem, LDA, possui além das primitivas necessárias à descrição de características gerais destes processadores, mecanismos para modelar aquelas peculiaridades de algumas arquiteturas superescalares, como por exemplo, diretivas para manipular janelas de registradores, *pipelines* com avanço explícito, latências de tempo de execução, latências auxiliares, recursos, etc. Enfim, LDA oferece vários mecanismos que auxiliam o processo de escalonamento de instruções. Mediante esta ferramenta, o projetista de compilador pode especificar o gerador de código para a máquina que deseja obter, bastando para isso que ele descreva as características do processador nesta linguagem. Estas informações são obtidas diretamente de manuais de usuário.

O projeto de LDA faz parte de um trabalho [Bigonha, 1994] no qual apresentamos um sistema de geração de código otimizado e definimos formalmente a a sintaxe e semântica de uma linguagem para descrever arquiteturas de computadores, cujos principais resultados obtidos foram: (1) projeto e validação semântica de uma linguagem de descrição de arquiteturas, apropriada para uso com o gerador de geradores de código projetado; (2) identificação dos atributos mais relevantes que devem ser considerados em uma LDA tendo em vista arquiteturas superescalares; (3) especificação da parte dependente de máquina do processador SPARC para seu respectivo gerador de código para demonstrar o poder de expressão da linguagem LDA proposta.

## Referências

[Aho et al., 1989] Aho, A. V., Mahadevan, G., and Tjiang, S. W. K. (1989). Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4).

[Aigrain et al., 1984] Aigrain, P. et al. (1984). Experience with a graham-glanville code generator. In *Proceedings of the Sigplan'84 Symposium on Compiler Construction*, pages 13–24. ACM Sigplan Notices 19(6).

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

[Benitez and Davidson, 1988] Benitez, M. E. and Davidson, J. W. (1988). A portable global optimizer and linker. *ACM Sigplan Notices*, 23(7).

[Benitez and Davidson, 1991] Benitez, M. E. and Davidson, J. W. (1991). Code generation for streaming: an access/execute mechanism. In *ASPLOS-IV Proceedings - Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara California.

[Bigonha, 1990] Bigonha, M. A. S. (1990). Linguagens para descrição de arquiteturas de computadores. Série de Monografias em Ciência da Computação 13/90, PUC-RJ, Departamento de Informática.

[Bigonha, 1994] Bigonha, M. A. S. (1994). *Otimização de Código em Máquinas Superescalares.* PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática-PUC/RJ.

[Bradlee et al., 1991] Bradlee, D. G., Eggers, S. J., and Henry, R. R. (1991). The marion system for retargetable instruction scheduling. In *ACM Sigplan Conference on Programming Language Design and Implementation.* ACM Sigplan Notices 26(7).

[Costa, 1990] Costa, P. S. S. (1990). Um gerador automático de geradores de código. Master's thesis, Pontifícia Universidade Católica do Rio de Janeiro.

[Davidson and Fraser, 1980] Davidson, J. W. and Fraser, C. W. (1980). The design and application of a retargetable peephole optimizer. *ACM Transactions on Programming Languages and Systems*, 2(2).

[Ganapathi and Fischer, 1985] Ganapathi, M. and Fischer, C. N. (1985). Affix grammar driven code generation. *ACM Transaction Programming Language and Systems*, 7(4):560–599.

[Ganapathi and Fischer, 1992] Ganapathi, M. and Fischer, C. N. (1992). Description-driven code generation using attribute grammars. In *Proceedings of the 9th POPL Conference*, pages 108–119.

[Giegerich, 1983] Giegerich, R. (1983). A formal framework for the derivation of machine specific optimizers. *ACM Translations on Programming Languages and Systems*, 5(3):478–498.

[Graham et al., 1982] Graham, S. L. et al. (1982). An experiment in table driven code generation. In *ACM Proceedings of the Sigplan'82 Symposium on Compiler Construction*, Boston Massachusetts. ACM Sigplan Notices 17(6).

[Graham and Glanville, 1978] Graham, S. L. and Glanville, R. S. (1978). A new method for compiler code generation. In *In Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, pages 231–240, Tucson Arizona.

[Henry and Damron, 1989a] Henry, R. R. and Damron, P. C. (1989a). Algorithms for table-driven code generators using tree-pattern matching. Technical Report # 89-02-03, Computer Science Department, University of Washington, FR-35 Seattle, WA 89195 USA.

[Henry and Damron, 1989b] Henry, R. R. and Damron, P. C. (1989b). Performance of table-driven code, generators using tree-pattern matching. Technical Report # 89-02-02, Computer Science Department, University of Washington, FR-35 Seattle WA 89195 USA.

[Russell, 1985] Russell, S. (1985). The compleat guide to mrs. Technical Report KSL-85-12, Stanford Knowledge Systems Laboratory, Stanford University.

[Stallman, 1989] Stallman, R. M. (1989). Using and porting GNU C. Free Software Foundation Incorporation. Cambridge Massachusetts.

[SUN Microsystems, 1987] SUN Microsystems, I. (1987). *The SPARC Architectural Manual.* Mountain View, California.

[SUN Microsystems, 1989] SUN Microsystems, I. (1989). *The SPARC Architectural Manual.* Version 8, Part No. 800-1399-09.

[Tiemann, 1989] Tiemann, M. D. (1989). The gnu instruction scheduler. Class Report CS 343, Stanford University.

[Wall and Powell, 1987] Wall, D. W. and Powell, M. L. (1987). The mahler experience: Using an intermediate language as the machine description. In *ASPLOS-II Proceedings - Second International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, California.

[Warfield and Bauer, 1988] Warfield, J. W. and Bauer, H. R. (1988). An expert system for a retargetable peephole optimizer. *ACM Sigplan Notices*, 23(10).

[Wulf et al., 1975] Wulf, W. A., Johnson, R., Weinstock, C., Hobbs, S., and Geschke, C. (1975). *The Design of an Optimizing Compiler.* American Elsevier.