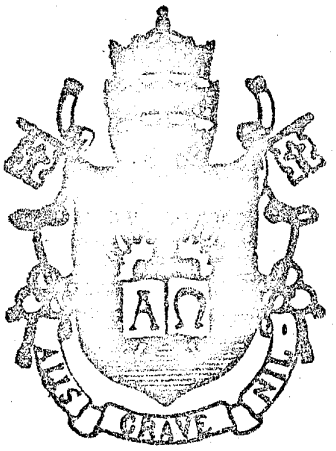

PUC

Série: Monografias em Ciência da Computação,
No. 14/90

LINGUAGENS PARA DESCRIÇÃO DE ARQUITETURA DE COMPUTADOR

Mariza A. S. Bigonha

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP-22453
RIO DE JANEIRO - BRASIL

PUC/RJ - DEPARTAMENTO DE INFORMÁTICA

Série: Monografias em Ciência da Computação, No. 14/90

Editor: Paulo A. S. Veloso Novembro/1990

Linguagens para Descrição de Arquitetura de Computadores[1]

Mariza Andrade da Silva Bigonha

[1]Trabalho apresentado como Qualificação ao Prof. Dr. José Lucas Mourão Rangel Netto

Parcialmente financiado pela CAPES/UFMG.

Responsável por publicações:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC RIO, Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453 - Rio de Janeiro, RJ
BRASIL

Tel.: (021) 529-9386 TELEX: 31078 FAX: (021) 274-4546
BITNET: userrtlc@lncc.bitnet

Abstract

The subject of this paper is to present a comparative study of different description languages for real computer architecture found in the literature. This paper begins with the presentation of some code generation and optimization systems that make use of machine architecture description. It also presents a brief description of the most important issues in the *MC68000, VAX* e *PDP-11* architectures with the objective to make easier to follow the examples in this paper. This work discusses the notation for register transfer used in the instructions description in one of the approaches presented here, and shows how this notation can be used to describe machine architecture. In addition, it is presented and evaluated some other notations used in the description of machines. The conclusion drawn from this work is that even though it is difficult to achieve consensus about which language is the most adequate, some characteristics of the language help in the selection process.

Keywords: Description languages for computers architecture, Machine descriptions, Code generation and optimization.

Sinopse

O trabalho apresentado nesta monografia é um estudo comparativo das diversas linguagens existentes para descrever arquiteturas de máquinas reais. Este artigo inicia com a apresentação de alguns sistemas de geração e/ou otimização de código que se utilizam de descrições de arquiteturas de máquinas alvo para atingirem seus objetivos. Apresenta uma descrição suscinta das principais características das arquiteturas *MC68000, VAX* e *PDP-11* afim de facilitar o entendimento dos exemplos contidos neste trabalho. Introduz uma notação para transferência de registradores utilizada para descrever as instruções em uma das abordagens apresentadas neste texto e mostra como arquiteturas de máquinas podem ser descritas nesta notação. Também são apresentadas, através de exemplos, algumas notações utilizadas nas especificações das máquinas, seguida de uma análise das mesmas. Desta análise, conclui-se que, muito embora seja difícil chegar-se a um consenso sobre qual linguagem é a mais adequada, certas características presentes ou ausentes em algumas linguagens auxiliam no processo de seleção.

Palavras-chave: Linguagens de descrição de arquitetura de computadores, descrições de máquina, geradores e otimizadores de código.

SUMÁRIO

1 Introdução

A idéia de projetar uma linguagem para descrever o conhecimento embutido nas arquiteturas de computadores faz parte da solução de um problema maior que é o projeto de um Gerador de Otimizador Local (GOL) baseado em técnicas oriundas da Inteligência Artificial. A solução deste problema incluirá, além da linguagem de descrição, objeto de estudo deste trabalho, o projeto de um Sistema Baseado em Conhecimento que a partir da descrição de uma arquitetura de máquina específica gere, automaticamente, regras que possibilitem dirigir um otimizador local; e, finalmente, o projeto de um otimizador local fixo e dirigido por regras.

A adequação da linguagem e do sistema propostos será verificada, posteriormente, através do projeto e implementação de um Gerador de Otimizadores Locais (GOL). A entrada do GOL será a descrição da máquina e a saída, o otimizador local dirigido por regras. Este otimizador local é fixo para qualquer arquitetura, sendo dirigido pelas regras geradas pelo GOL. A entrada para este otimizador será o código objeto, e o resultado deverá ser sequências de instruções otimizadas da máquina alvo. As Figuras 1 e 2 apresentam o diagrama completo do GOL e e utilização do produto obtido.

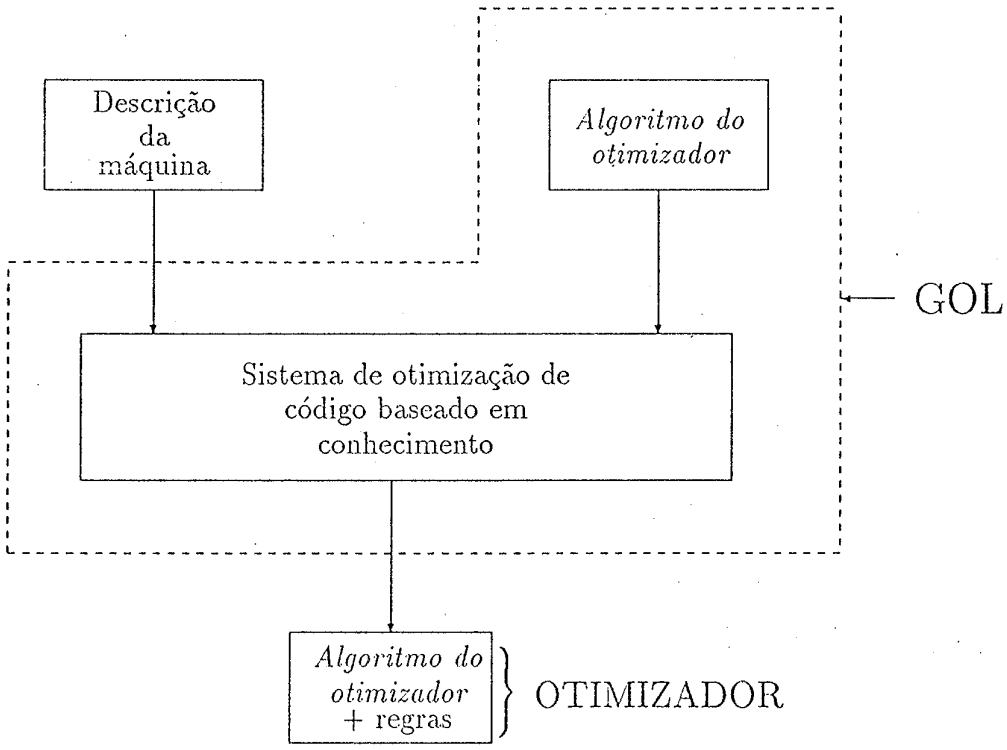

Figura 1: Sistema GOL.

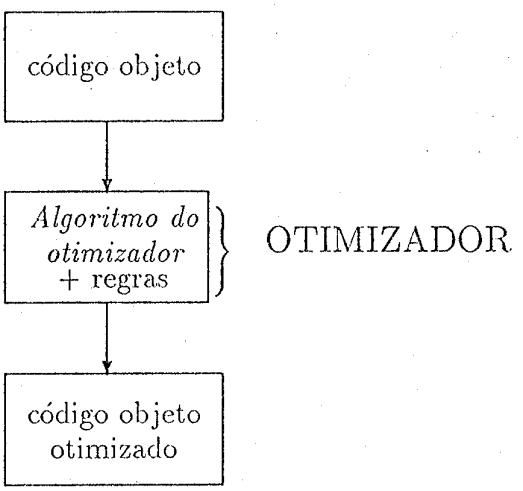

Figura 2: Uso do Otimizador

Faz parte do projeto GOL a avaliação da qualidade do código obtido, bem como as dificuldades advindas da metodologia adotada e da arquitetura da máquina alvo escolhida.

Em um compilador, automatizar as funções que dependem fundamentalmente de características da máquina alvo e de seu conjunto de instruções e gerar um bom código objeto não é uma tarefa simples dada a crescente proliferação de arquiteturas de máquinas, cada qual com características distintas. Existem arquiteturas com um, dois e três endereços; máquinas baseadas em registradores, máquinas baseadas em pilhas, etc. Mesmo entre as máquinas possuindo a mesma classe de arquitetura há grandes diferenças. Além dos conjuntos de instruções serem diferentes, cada arquitetura possui suas próprias idiossincrasias: certos registradores possuem uso restrito, por exemplo, divisão só pode ser efetuada em pares de registradores "par-ímpar"; algumas instruções ligam o código de condição, enquanto outras não. Tudo isto torna difícil decidir que instruções usar para produzir um bom código. No passado, a geração e otimização de código usavam intensamente a análise de caso para gerar sequências "ótimas"' de código.

Para que este problema seja formalmente tratado, torna-se necessário, entre outras providências, descrever formalmente arquiteturas de máquinas reais pois compiladores geram código para máquinas reais. Sabe-se, entretanto, que descrições formais de arquiteturas de máquinas reais tendem a ser complexas. Portanto, a proposta de uma única linguagem que atenda às diferentes características de arquitetura de máquina alvo existentes pode ser tão complexa quanto inadequada a aplicações reais devido aos motivos expostos no parágrafo anterior.

Procurando restringir o conjunto de instruções da linguagem descritiva, o projeto se torna mais viável e com possibilidade de ser bem sucedido. Isto é possível pois existem instruções

no repertório de instruções de uma máquina que nunca são geradas; existem outras instruções que tem pouca chance de serem substituídas durante o processo de otimização, como por exemplo, instruções de Entrada e Saída; movimentação de blocos; instruções de desvios entre subrotinas. Todas estas instruções, do ponto de vista do otimizador de código, podem ser omitidas da linguagem de descrição da arquitetura.

Além das dificuldades apontadas em relação as diferentes características das arquiteturas existentes, há uma série de questões que são levantadas referentes à linguagem e que devem ser avaliadas neste trabalho. Por exemplo: o que a linguagem deve prover, quais são suas restrições, propriedades, aplicabilidade, quais são as suas características, como por exemplo, o que vai ser processado, uma gramática, regras, quais os caminhos de solução possíveis em relação aos diferentes modos de endereçamento de uma dada arquitetura. Enfim, como sistematizar o conhecimento da máquina de uma forma clara e concisa.

A parte do problema que será abordado neste trabalho é um estudo comparativo das diversas linguagens existentes para descrever arquiteturas de máquinas reais e um possível desenvolvimento ou adaptação de uma destas linguagens estudadas para atender o objetivo de *GOL*. O desenvolvimento ou adaptação da linguagem não faz parte deste trabalho.

Este texto inicia com a apresentação de alguns sistemas de geração e/ou otimização de código que se utilizam de descrições de máquina para atingirem seus objetivos. Seção 2 apresenta uma descrição suscinta das principais características das arquiteturas *MC68000, VAX* e *PDP-11* afim de facilitar o entendimento dos exemplos contidos neste trabalho. Seção 3 introduz uma notação para transferência de registradores utilizada para descrever as instruções da abordagem mostrada na seção 1.1.5 e explica como as arquiteturas de máquinas podem ser descritas. Em seguida são apresentadas diversas abordagens de descrição através de exemplos. Seção 4 analisa estas abordagens e justifica porque as notações vistas nas seções 3.3 e 3.4.4 poderiam ser escolhidas para serem adaptadas para o sistema *GOL*. Seção 5 apresenta a conclusão e bibliografia.

1.1 Trabalhos Anteriores

Existem na literatura trabalhos elaborados nas áreas de geração e otimização de código que têm respostas para algumas das questões levantadas na seção anterior, por exemplo: [GLANVILLE 78], [FRASER 86], [FRASER 88], [FRASER 89], [WARFIELD 88], [CATTELL 78], [LEVERETT 80], [COSTA 90], [HENRY 89a], [HENRY 87], [HENRY 89b], [DAVIDSON 80],[DAVIDSON 81], [DAVIDSON 84a], [DAVIDSON 84], [DAVIDSON 87], [GANAPATHI 82], [KESSLER 84], [KESSLER 86], [GIEGERICH 83].

1.1.1 Abordagem de Glanville

Uma das mais promissoras abordagens referentes a geração automática de código é atribuída aos pesquisadores Robert S. Glanville e Susan L. Graham [GLANVILLE 78] e [GRAHAM 80]. A técnica descrita por eles fazem uso de um algoritmo dirigido por tabela que traduz um programa na forma de uma representação intermediária em linguagem de montagem para uma máquina alvo. As tabelas são construídas a partir da descrição do repertório de instruções da arquitetura da máquina alvo.

Construir tabelas para geração de código é similar á construção de tabelas para a análise sintática a partir de uma gramática. De fato, a teoria utilizada na análise sintática baseada em gramáticas livres do contexto foram utilizadas no desenvolvimento do algoritmo [AHO 73]. Como resultado, o algoritmo é rápido e de fácil entendimento.

O gerador de código produz um bom código local. Em sua dissertação de doutorado, "*A Machine Independent Algorithm for Code Generation and its use in Retargetable Compilers*", Glanville discute a implementação de geradores de código para duas máquinas, o *PDP-11* e o *IBM 370*. Muito embora estas máquinas possuam arquiteturas distintas, é difícil avaliar a habilidade do algoritmo de se adaptar a um universo maior de arquiteturas. Glanville supõe ter dificuldades com a adaptação de seu método com a família *CDC Cyber* devido a sua arquitetura "desajeitada".

Um problema mais sério aparece na descrição da máquina. Devido ao fato das tabelas serem construídas usando a técnica de análise sintática baseada em gramáticas livres do contexto, é necessário descrever cada instrução com todos os modos de endereçamento possível. Por exemplo, o *PDP-11* possui oito modos de endereçamento para uma palavra. Em algumas instruções de operando duplo, todo modo de endereçamento pode ser utilizado no fonte e no destino. Para uma instrução "*mov*" por exemplo, existem sessenta e quatro combinações de modos de endereçamento possíveis. Isto significa que uma descrição completa do *PDP-11* deve conter sessenta e quatro padrões somente para descrever a instrução "*mov*".

1.1.2 Abordagem de Ganapathi

O trabalho de Ganapathi [GANAPATHI 82] é similar a abordagem de Glanville. Enquanto Glanville se utiliza de gramáticas *LR*, Ganapathi usa gramáticas de atributos [KNUTH 68]. Na abordagem de Ganapathi, o algoritmo básico de Glanville é modificado para prover um processamento de atributos formalizado e um guia semântico automático. As principais extensões introduzidas estão relacionadas com a estrutura do gerador de código, atribuição de endereços, otimizações dependentes de máquina e redirecionamento. Atributos semânticos e atributos predicados são utilizados para atingir este objetivo. A máquina alvo é descrita usando gramática de atributos ao invés de gramáticas livre do contexto [AHO 73], e a geração de código é efetuada pelo analisador sintático com a

avaliação dos atributos.

Ganapathi produziu geradores de código para as arquiteturas *PDP-11* e *VAX-11/780.* Estas máquinas são tão similares que é impossível avaliar a habilidade de sua técnica para acomodar outras arquiteturas. A descrição da máquina é muito longa e difícil de ser entendida. A descrição de Ganapathi para o *PDP-11* possui sete páginas, enquanto a descrição de Glanville possui três páginas. Da mesma forma, a descrição de Ganapathi para o *VAX-11/780* é composta de onze páginas.

Ganapathi se utiliza dos atributos para efetuar várias otimizações dependentes de máquina. Estas otimizações são manualmente embutidas nas ações semânticas da descrição da máquina. Consequentemente, a qualidade do código gerado é bem melhor do que os métodos existentes, [JOHNSON 78] etc, até então.

1.1.3 Abordagem de Leverett

O projeto *PQCC* "*Production-Quality Compiler-Compiler*" [LEVERETT 80] é mais ambicioso que os dois trabalhos apresentados anteriormente. O objetivo de *PQCC* é automatizar todas as fases de construção de um compilador. O resultado prático deste objetivo é obter um sistema de geração de compiladores realmente automático. O projeto focalizou em duas áreas: na otimização e na geração de código.

A seleção do código de *PQCC* é efetuada pelo gerador de gerador de código desenvolvido por Cattell [CATTELL 78], [CATTELL 79] e [CATTELL 80]. A técnica de geração de código é similar a abordagem utilizada por Johnson [JOHNSON 78] no compilador C. Gabaritos (*templates*) são casados com a representação em forma de árvore do programa, entretanto, neste trabalho os gabaritos são gerados automaticamente a partir da descrição de máquina, o que não acontece com o método de Johnson.

Gabaritos são gerados usando métodos de pesquisa heurísticos, técnica oriunda de Inteligência Artificial. O uso de heurística não garante que a "melhor" sequência de instrução seja encontrada, ou ainda, que alguma sequência seja encontrada. Entretanto, na prática, heurística parece ser bem eficaz. As instruções na abordagem de Cattell são descritas como um conjunto de asserções que expressam a ação das instruções. Essas asserções são representadas como árvores, e uma notação semelhante a *LISP* é utilizada pelas árvores. Cattell relata em sua dissertação que produz 2000 instruções por segundo no *computador DEC-10.*

Várias etapas inerentes ao gerador de código não são abordadas no gerador de código de Cattell. Tais etapas como: alocação e atribuição a registradores, atribuições a temporários e a memória são tratados por outras fases do compilador. Isto espalha informações dependentes da arquitetura alvo ao longo do compilador. Trabalhos foram realizados pelo grupo do projeto *PQCC* para a construção de geradores que deduzam estas informações

dependentes de máquina da descrição da arquitetura [LEVERETT 80].

1.1.4 Abordagem de Costa

O trabalho de Costa [COSTA 90], 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. Seu sistema se assemelha a abordagem de Glanville apresentada na Seção 1.1.1.

Costa introduz três tipos de extensões à gramática de descrição da máquina alvo de Glanville. A primeira diz respeito à correspondência entre instruções e produções. Na abordagem de Glanville, cada produção é associada a uma única instrução. Em *AutoCode*, cada produção em uma descrição de máquina pode ser associada a várias instruções limitado apenas pela disponibilidade de memória da máquina onde o sistema esteja sendo executado. Esta extensão permite que instruções especiais sejam descritas com maior precisão. A segunda extensão diz respeito ao uso de operadores semânticos, eles foram introduzidos com o objetivo de efetuar intervenções semânticas em tempo de geração de código. A terceira e última extensão diz respeito às técnicas de fatoração gramatical. Dois tipos de fatoração foram incluídos: fatoração de modos de endereçamento e fatoração de operadores.

AutoCode é modularmente dividido em duas partes: a primeira parte refere-se a construção automática das tabelas que guiarão o algoritmo do gerador de código, e a segunda parte refere-se a geração de código propriamente dita.

A abordagem de Costa, entretanto, possui algumas características indesejáveis [COSTA 90]. A primeira delas é a dependência de máquina da representação intermediária emitida pelo *front-end*. Isto prejudica a portabilidade do compilador. A segunda diz respeito a inclusão de operadores semânticos na linguagem de descrição de arquitetura. Muito embora esta inclusão aumente o poder de expressão de uma descrição de máquina, por outro lado, aumenta também o tempo gasto para a geração de código, porque novas reduções devem ser efetuadas.

O esquema proposto por Costa pode mostrar-se ainda ineficiente, se aplicado a arquiteturas pouco simétricas. Modos de endereçamento e instruções complexas de máquinas especializadas, normalmente, criam um número indesejado de casos especiais a serem explorados pela gramática descritiva. Ademais, muitas arquiteturas possuem registradores cujo uso é limitado a operações exclusivas, e diversas instruções possuem efeitos colaterais difíceis de serem formalizados. Estes fatores podem comprometer a qualidade do código gerado.

1.1.5 Abordagem de Davidson

Davidson [DAVIDSON 81] descreve uma técnica para implementação rápida de compiladores de produção de boa qualidade, através da utilização de um otimizador local independente de máquina, denominado *PO*. Dado um programa em linguagem de montagem e uma descrição simbólica da máquina, *PO* simula pares de instruções adjacentes, e, quando possível, troca-as por uma única instrução equivalente. *PO* faz um passo para determinar o efeito de cada instrução, um segundo passo para juntar pares com o mesmo efeito e um terceiro para selecionar a instrução mais barata. *PO* se utiliza da descrição da máquina para verificar a validade do resultado obtido. Quando *PO* termina, nenhuma instrução, ou par de instruções adjacentes pode ser substituída por outra de menor custo. Esta completeza permite ao *PO* isentar os geradores de código de muitas análises de casos, como por exemplo, o gerador de código pode produzir somente sequências "*load/add-register*" e fiar-se em *PO* para, quando possível, descartá-los em favor de instruções "*add-memory, add-immediate*", ou incremento. Como resultado desta organização, PO é independente de máquina e pode ser descrito formalmente e concisamente.

A arquitetura de máquina para Davidson é descrita através de uma gramática para tradução dirigida por sintaxe entre a linguagem de montagem da máquina alvo e a transferência de registradores em *ISP*. A partir da descrição da máquina, um reconhecedor e um tradutor são construídos automaticamente. *PO* utiliza-se do reconhecedor para verificar se as transferências de registradores representam instruções válidas na máquina alvo. *PO* usa o tradutor para verter a representação interna das instruções para a linguagem de montagem da máquina alvo. Detalhes sobre a descrição da máquina são apresentados na seção 3.

As publicações de Davidson e Fraser [DAVIDSON 80] e [DAVIDSON 84] também elaboram sobre esta abordagem. [DAVIDSON 80] apresenta a primeira versão de *PO* e [DAVIDSON 84] descreve a implementação do compilador "YC", para a linguagem de programação Y [HANSON 81], que otimiza após a geração de código.

Muito embora os resultados obtidos no método proposto por [DAVIDSON 81] sejam bons, existem duas limitações inerentes em abordagens interpretativas como esta: a primeira refere-se ao número de instruções na janela do "peephole", e a segunda diz respeito a eficiência, já que o fato de efetuar todas as manipulações simbólicas em tempo de compilação faz com que *PO* se torne muito lento.

Para solucionar a segunda limitação, Davidson e Fraser estendem *PO* para gerar automaticamente padrões ou regras que descrevem as otimizações que devem ser efetuadas. Um conjunto fixo de regras é gerado em tempo de geração de compilador e carregado em um otimizador dirigido por regras, intitulado *HOP*. A descrição deste sistema não será abordado neste texto dado que o objetivo deste trabalho é apresentar as diversas formas de representação de descrição de máquina; detalhes sobre *HOP* podem ser encontrados em [DAVIDSON 84a] e [DAVIDSON 87]. A vantagem desta abordagem é que as regras

são derivadas automaticamente. A desvantagem é que essencialmente deve se construir dois compiladores, um que opera usando a descrição da máquina e um outro que usa as regras. O segundo, entretanto, é construído automaticamente uma vez que o primeiro é completado. O resultado final é sem dúvida um otimizador local rápido.

1.1.6 Abordagem de Robert R. Kessler:

Robert R. Kessler [KESSLER 84] descreve um sistema, denominado *PEEP*, que, ao invés de analisar as sequências de instruções que ocorrem durante a geração de código, analisa a descrição da máquina durante a construção do compilador. A técnica proposta por ele limita-se a descobrir instruções que sejam equivalentes a sequências de instruções de comprimento dois. Kessler utiliza-se da descrição da máquina para encontrar todas as otimizações possíveis. Os efeitos de um par de instrução são combinados, e a descrição de instruções é pesquisada para descobrir uma única, mais eficiente e que tenha o mesmo efeito que as duas instruções combinadas. *PEEP* se apresenta em duas partes, o gerador de tabelas *PEEP* propriamente dito, que efetua a análise da máquina alvo, e o otimizador *PEEP* que efetua as otimizações como especificadas pelas tabelas produzidas pelo gerador de tabelas.

Robert Kessler utiliza-se de uma linguagem baseada em *LISP* para descrever a arquitetura da máquina alvo. Esta proximidade de *LISP* facilita-lhe a expressão de construções, e proporciona uma grande flexibilidade na escrita das definições, além de permitir ao usuário escrever macros *LISP*, quando necessário. O usuário pode definir constantes, registradores, modos de endereçamento e instruções. A definição de constantes informa ao sistema a presença de constantes e provê seus valores. A definição de registradores define todos os registradores pertencentes a uma determinada classe, por exemplo, registrador *A* no *MC68000* e todos os seus sinônimos, isto é, por exemplo, registrador *A7* é também referido como registrador *ST*. Na maioria das arquiteturas, cada instrução permite vários modos de endereçamento para cada operando. A linguagem proposta por Kessler permite a definição de uma enumeração na especificação de cada um dos operandos, ou seja, os operandos das instruções são definidos como um conjunto de todos os modos de endereçamento possíveis. Esta enumeração é utilizada durante o casamento de padrões de instruções. Ao pesquisar por uma otimização, dois operandos podem ser interceptados para verificar a superposição de modos de endereçamentos. Se a interseção é vazia, a otimização é ignorada. As instruções são descritas provendo o seu formato de entrada, as equações semânticas definindo suas funções e o seu custo englobando tempo e espaço.

1.1.7 Abordagem de Giegerich

Giegerich [GIEGERICH 83] apresenta um método para formalizar arquiteturas de máquinas visando a derivação sistemática de otimizadores, onde a exatidão da otimização seja garantida.

A derivação de otimizadores para máquinas específicas tem início a partir da descrição da máquina alvo. Uma descrição formal da semântica do conjunto de instruções define de forma precisa os conceitos característicos a nível de código de máquina, tais como: os modos de endereçamento e instruções, os efeitos colaterais, a superposição de registradores ou células de memória. A notação adotada para a descrição do conjunto de instruções é adaptada de ISP [BELL 71].

A descrição da máquina é analisada dentro de uma abordagem formal. A análise deriva predicados e funções que irão testar e inferir informações sobre o fluxo de dados. Estes predicados esclarecem dependências de máquina, dependências de fluxo de dados e propriedades dependentes do contexto. Os predicados dependentes de máquina nas instruções e modos de endereçamento são avaliados em tempo de geração de compiladores. Os predicados dependentes de programa devem ser avaliados em tempo de geração de código. Durante a geração de código, uma instrução é comparada com outra e substituída se for vantajoso. O otimizador aplica várias transformações independentes de máquina explorando informações de fluxo de dados dependentes do programa. Estas transformações cobrem várias otimizações locais, incluindo eliminação de sub-expressões comuns e código redundante.

1.1.8 Abordagem de Peter B. Kessler:

Para aliviar alguns dos problemas inerentes a um modelo dirigido puramente pela sintaxe, Peter B. Kessler [KESSLER 86] sugere a inclusão de uma fase separada de transformação de código. Assim, construções de propósitos especiais são completamente removidas da descrição da máquina. E, ao invés de usar a composição "força bruta" para formar sequências de instruções, ele sugere a descoberta de idiotismos ou idiomatismos (*idioms*) pela decomposição. Uma sequência de instrução complexa é decomposta em uma sequência de instruções simples, e por este meio determina-se sequências de código ineficientes que podem ser substituídas por outras mais eficientes. Ou seja, esta técnica identifica restrições semânticas em uma sequência de instruções de tamanho arbitrário que a tornam equivalente a uma única instrução de propósito especial, denominada idiomatismo. A decomposição não é limitada a descobrir pares de instruções equivalentes; ela pode descobrir que uma instrução é equivalente a uma longa sequência de instruções. No pior caso, decomposição toma mais tempo que a composição, entretanto, na média, decomposição pode gastar menos tempo.

Suscintamente, a decomposição da descrição de instruções é feita da sequinte maneira: dada qualquer instrução, identifica-se todas as outras sequências de código que podem ser substituídas por aquela instrução. Este processo é repetido para cada instrução da máquina alvo, produzindo uma lista de todas as restrições de equivalência.

A maior contribuição desta técnica está no fato de que sequências de instruções podem ser estendidas para comprimentos arbitrários em uma tentativa de decompor uma instrução.

A complexidade do processo de análise é exponencial ao número de instruções da máquina alvo, o grau de exponenciação depende do comprimento das sequências equivalentes que foram encontradas [KESSLER 86]. Esta é uma propriedade importante, pois, quanto mais complexo for o conjunto de instruções mais tempo levará para analisar.

1.1.9 Abordagem de Fraser

Fraser [FRASER 89] descreve uma linguagem de programação para escrever geradores de código. A linguagem proposta por ele abrevia construções repetitivas, simplifica a codificação e torna os geradores de códigos menores e mais rápidos. Por exemplo, uma especificação para o VAX gasta 126 linhas, para o Motorola 68020 gasta 156 e para o MIPS R3000 são gastas 75 linhas. Sua técnica contrasta com os mais recentes métodos para geradores de código redirecionáveis, inclusive aqueles propostos pelo próprio autor: em primeiro lugar, os sistemas mais recentes aceitam descrições de máquina em uma representação não-procedural e produzem tabelas para serem interpretadas em tempo de compilação. Este sistema aceita uma representação compacta de um programa e emite um gerador de código *hard-coded*. A especificação do sistema possui um aspecto procedural, entretanto ela é menor que outras especificações. Em segundo lugar, os sistemas mais recentes utilizam-se de técnicas sofisticadas para gerar suas tabelas; em particular, este sistema usa um pré-processador, cuja operação é bem transparente. E, como nos analisadores sintáticos, qualquer um pode observar um analisador descendente recursivo e *"ver"* a gramática por trás, contudo é difícil *"ver"* qualquer padrão significativo em uma tabela LR. Em terceiro lugar, os sistemas mais recentes fiam-se em algoritmos de propósitos gerais cuja aplicação abrange um universo maior que a geração de código. Por exemplo, os sistemas de Graham-Granvile [AIGRAIN 84], [GANAPATHI 85] fiam-se em analisadores LR. O sistema de Twig e Burs [AHO 85] e [GRAHAM 88] fiam-se em avanços recentes em reconhecimento de padrão em árvores [CHASE 87]. Sistemas baseados em otimizadores locais redirecionáveis [DAVIDSON 84] baseiam-se em simulação simbólica. Em contraste, a técnica fundamental no sistema de Fraser é específica, correspondendo apenas a geração de código.

Programas são representados na linguagem para geração de código principalmente através de regras de reescrita. Algumas regras reescrevem o código intermediário em um código simples em linguagem de montagem. Outras regras otimizam localmente o resultado. A linguagem de regras representa cada instrução da máquina alvo com uma instrução em linguagem de montagem sob a forma de gabaritos *(templates)*. Por exemplo,

```
mov{b w l f d} y, z
{add sub mul div} {b w l f d}3 x,y,z
```

representa várias instruções VAX.

As regras de otimização nesta abordagem são escritas na mesma linguagem que as regras de geração de código, muito embora, os idiomatismos sejam um pouco diferentes. Regras de geração de código casam código intermediário e produzem código objeto, enquanto as regras de otimizações casam códigos objetos e produzem códigos objetos melhorados.

Cada especificação é compilada em um programa C, denominado *rewrite*, que aceita *dags* (grafos acíclicos dirigidos) anotados com código intermediário, e gera, otimiza e emite código para a máquina alvo. Os geradores de código são usados com um *front-end* para o *ANSI C*. Os compiladores resultantes emitem código similar ao *pcc1's* [JOHNSON 78], mas eles são executados duas vezes mais rápido.

É difícil avaliar a habilidade do método proposto, seu trabalho não apresenta exemplos para outras arquiteturas além do *VAX*, muito embora Fraser mencione o número de linhas gastas na especificação das arquiteturas do *MOTOROLA 68002* e do *MIPS R3000*.

Trabalhos nesta linha pesquisam novas máquinas alvo e codificações mais eficientes para as instruções. As regras poderiam ser mais simples, como por exemplo em [DAVIDSON 87], [DAVIDSON 89]. Davidson e Whalley [DAVIDSON 89] mostram que o sucesso do redirecionamento pode ser baseado em regras de reescrita bem mais simples com seu compilador *vpcc* que as regras proposta por Fraser, muito embora, a técnica deles requeira casamento de *string* em tempo de compilação. Pesquisas em andamento investigam geradores de código tão rápido como *rewrite* [FRASER 89] a partir de regras tão simples como aquelas utilizadas no *vpcc's* [DAVIDSON 89].

1.1.10 Abordagem de Fraser e Wendt

Fraser e Wendt [FRASER 86] propõem um compilador, onde o gerador de código e o otimizador local dependente de máquina estão coesamente integrados. Ambas as funções são efetuadas por um único sistema baseado na reescrita de regras (*rule rewriting*), que casa padrões e substitui novos textos por eles. Esta organização torna o compilador mais simples, rápido e mais capaz de produzir um bom código.

O projeto, proposto por eles, se inicia com um otimizador local dirigido por regras, denominado *HOP* [DAVIDSON 84a], e o generaliza, para também assumir as responsabilidades da geração de código. O compilador é redirecionável. As regras de geração de código são escritas manualmente, mas sua tarefa é simplificada pela ausência de análise de casos especiais. A necessidade de escrever estas regras é compensada pelo fato de a descrição da máquina [DAVIDSON 81] necessária ser pequena o suficiente para tornar o método descrito competitivo com os outros métodos de compiladores redirecionáveis existentes [CATTELL 80].

Um conjunto de regras gera código através da substituição do código intermediário por instruções da máquina alvo, representadas como transferência de registradores. Outro, o qual

é geralmente criado automaticamente em tempo de geração de compiladores (compiler-compiler time), otimiza este código tão logo ele seja produzido, substituindo instruções justapostas por uma única. Ainda outras regras traduzem as transferências de registradores otimizados para código em linguagem de montagem.

No sistema *HOP* quando um novo programa utiliza uma superposição de instruções que não foi vista quando as regras foram geradas em tempo de geração de compiladores *HOP* pode deixar de efetuar otimizações. Para corrigir esta dificuldade e tornar o sistema mais robusto, o sistema de Fraser e Wendt estende *HOP* sob dois aspectos. Em primeiro lugar, integra *HOP* e *PO* [DAVIDSON 81] de tal forma que as regras em *HOP* sejam estendidas incrementalmente invocando *PO* para gerar regras para substituir ou rejeitar justaposições de instruções anteriormente não vistas. Estas regras são geradas em tempo de compilação, mas nenhuma regra gerada anteriormente necessita ser produzida novamente. Portanto, o efeito é aproximadamente o mesmo da geração em tempo de geração de compiladores. Em segundo lugar, Fraser e Wendt estendem as regras de reescrita de tal forma que agora elas podem invocar rotinas "*built-in*" para efetuar operações que não podem ser implementadas convenientemente com o reconhecimento de padrão e substituições. As regras de reescrita deste sistema se assemelham as regras de *HOP*.

Outro trabalho recente de Fraser e Wendt [FRASER 88], também nesta linha, utiliza-se da notação para descrição da arquitetura de máquina delineada por [DAVIDSON 81].

1.1.11 Abordagem de Henry

Henry ([HENRY 87], [HENRY 89a], [HENRY 89b]) descreve um gerador de código redirecionável, denominado *CODEGEN*, projetado para substituir *pcc1*, o gerador de código da primeira versão do Compilador C [JOHNSON 78]. *CODEGEN*, desde 1980, tem sido utilizado para estudar a organização de geradores de código, as linguagens de especificações de geradores de código, os esquemas de transformações de árvores, os geradores de reconhecedores de padrões em árvores, a administração de registradores, os otimizadores locais e as técnicas utilizadas para descrever arquiteturas de máquinas.

CODEGEN é somente uma parte de um compilador completo, contendo um *front-end*, o próprio *CODEGEN*, um montador e um editor de linquedição.

CODEGEN consiste em quatro fases lógicas: o transformador de árvores, o seletor de instruções, o administrador de temporários e o formatador de instruções. A estrutura do gerador de código é análoga ao modelo utilizado em um compilador de um único passo dirigido por sintaxe. O transformador de árvores desenvolve o papel de um analisador léxico; o seletor de instruções desempenha o papel de um analisador sintático; e o administrador de temporários e o formatador de instruções (*EMIT*) desempenham o papel do atribuidor. O transformador de árvores lê árvores na forma intermediária e as converte em uma floresta de pequenas árvores expressas em uma notação um pouco diferente da

representação intermediária. O seletor de instruções é responsável pela determinação de sequências de instruções de máquina semânticamente equivalentes a uma árvore na forma intermediária. O administrador de temporários é responsável pela alocação e liberação de temporários, tanto em registradores como na memória principal. O formatador de instruções é responsável pelo agrupamento de fragmentos de instruções virtuais, convertendo-as em instruções da máquina alvo. O administrador de temporários e o formatador de instruções manipulam atributos descrevendo partes das instruções.

O seletor de instruções é redirecionado para outra máquina, substituindo-se a especificação das instruções da máquina alvo. A descrição das instruções é definida como uma enumeração de cinco-tuplas denominada *PPRACs* ou *regras*. Um conjunto de *PPRACs* em uma descrição de máquina é denotada por "*P*". Cada *PPRAC* modela a semântica de uma parte ou de toda instrução da máquina alvo. Uma *PPRAC* é constituída de um *padrão* (*pattern*), um *predicado booleano* (*boolean predicate*), uma *substituição* (*replacement*), uma *ação* (*action*) e um *custo* (*cost*).

Um *padrão* deve ser uma árvore e pode conter símbolos de substituição somente nas folhas. O *predicado* deve ser uma função booleana que descreve quais condições semânticas devem ser válidas antes que uma *PPRAC* possa casar. Se o *predicado* for omitido, assume-se verdadeiro. A *substituição* deve ser um símbolo não-terminal. A *ação* é um trecho de código na linguagem de implementação. Cada um dos C_i deve ser uma função retornando um número não negativo. O C_i modela o custo do i-ésimo recurso.

Uma *PPRAC* é definida da seguinte forma:

replacement $\rightarrow$ pattern *when* predicate = action *cost*(C_1 ... C_{ncost})

analizada como:

(replacement $\rightarrow$ pattern)(*when* predicate)=(action)(*cost*(C_1 ... C_{ncost}))

e lida da seguinte maneira: se um padrão casa, e se o predicado for verdadeiro, e se a *PPRAC* for selecionada, então a ação semântica é avaliada por seu efeito colateral, a porção da subárvore casada é reescrita com a substituição (*replacement*), e o custo incurrido é descrito pelo componente de custo.

1.1.12 Abordagem de Warfield e Bauer

Warfield e Bauer [WARFIELD 88] apresentam uma técnica que usa um Sistema Especialista com a missão de reconhecer que instruções podem ser otimizadas a partir da descrição da instruções da máquina alvo. Uma ferramenta denominada "Meta-level Representation System" (MRS) [RUSSELL 85] é utilizada na construção do sistema especialista, sendo sua característica principal a inclusão de um esquema de controle flexível utilizando ra-

ciocínio para-frente, raciocínio para trás e uma técnica denominada resolução [RICH 83] para provar teoremas, além da habilidade de representar o conhecimento sobre ele mesmo (metalevel knowledge). MRS descreve uma teoria como um conjunto de fatos em sua própria biblioteca.

Neste sistema são definidas três teorias: a teoria de descrição de instruções que descreve o que cada instrução faz em termos das proposições entendidas pelo MRS. A teoria de modos de endereçamento que descreve os modos de endereçamento e suas classes. Finalmente a teoria de otimização. Nesta teoria utilizando o raciocínio para trás e um "peephole" de duas instruções as regras tentam encontrar todas as otimizações possíveis usando certos critérios.

O otimizador de regras retorna uma lista de todas as otimizações possíveis e as condições que devem ser verdadeiras para que o otimizador funcione. Este resultado do Sistema Especialista é colocado na forma de regras e posto em uma teoria denominada Regras. Novas regras são criadas a partir daquelas que o otimizador retorna. O raciocínio para-frente é então usado para otimizar o código objeto utilizando estas regras. Cada linha do código objeto é lida como um fato na biblioteca após ter sido codificada no formato definido por MRS. Após duas instruções terem sido declaradas, MRS pesquisa, automaticamente, as regras para encontrar uma que case as duas instruções. Se esta regra é encontrada, MRS instala a nova instrução otimizada na biblioteca. Esta nova instrução pode ser recuperada da biblioteca para ser usada na substituição de duas instruções originais no código objeto.

A linguagem de descrição proposta por Warfield e Bauer baseia-se na sintaxe de LISP. Ela provê, além de facilidades para descrever instruções e modos de endereçamento, e suas respectivas classes, um mecanismo para definir regras.

Para redirecionar o Sistema Especialista proposto para outra máquina, basta substituir a descrição das instruções na teoria de instruções e a descrição dos modos de endereçamento na teoria de modos de endereçamento.

2 Descrição de Máquina

Esta seção descreve, suscintamente, os processadores *MC68000*, *VAX-11* e o *PDP-11*, suas arquiteturas e modos de endereçamentos. O objetivo desta seção é apresentar os aspectos relevantes destas arquiteturas, possibilitando um melhor entendimento das diversas instruções *add* utilizadas como base de comparação entre as distintas notações para descrição de arquitetura de máquinas, que serão apresentadas na seção 3.

2.1 MC68000

2.1.1 Instruction-set Processor

Os processadores da família *MC68000* [MOTOROLA 82] possuem os seguintes registradores:

- oito registradores de dados (32 *bits*), *D0-D7*, que podem operar com dados de tamanho 8-bit (*byte*), 16-bit (*word*) e 32-bit (*long word*);
- sete registradores *A0-A6* de endereço (*32 bits*). O registrador *A7* é usado como apontador de pilha e como base de endereço. Este registrador contém o apontador de pilha do modo usuário e o apontador de pilha de modo supervisor. Os oito registradores contidos neste item possuem metades individualmente endereçáveis, ou seja, podem ser usados com operações de endereço para *word* e *long word*;
- o registrador de *status* contém as máscaras de interrupção e também os códigos de condição: *overflow (V), zero (Z), negative (N), carry (C) e extend (X).* Este registrador é de *16 bits.*
- o registrador *program counter* é um registrador de *32 bits.*

Os dezessete registradores apresentados podem ser usados como registradores de índices.

2.1.2 Registradores

Os registradores de dados destinam-se a operações lógicas e aritméticas. Cada um destes registradores possui 32 *bits.* Operandos do tipo *byte* ocupam os 8 *bits* de mais baixa ordem, operandos do tipo *word* ocupam os 16 *bits* de mais baixa ordem e os operandos do tipo *long wordd* ocupam os 32 *bits.* O *bit* menos significativo é endereçado como o *bit* zero; o mais significativo é endereçado como *bit* 31.

Quando um registrador de dados é utilizado como operando destino ou operando fonte, somente a apropriada porção de mais baixa ordem é alterada, o restante da porção de mais alta ordem não é utilizada ou modificada.

Os registradores de endereço e o registrador apontador de pilha comportam endereços de 32 *bits.* Estes registradores não operam com operandos do tipo *byte.* Portanto, quando um registrador de endereço é utilizado como um operando fonte, tanto a palavra de mais baixa ordem ou a palavra longa é utilizada, dependendo apenas do tamanho da operação. Mas, quando um registrador de endereço é utilizado como um operando destino, o registrador inteiro é afetado, independente do tamanho da operação. Por exemplo, se o tamanho

de uma operação é uma palavra, os operandos são estendidos para 32 *bits* antes que a operação seja efetuada.

2.1.3 Endereçamento de Memória

O *MC68000* possui um barramento de endereço de 24-bit, o que o torna capaz de endereçar 16 *megabytes* de memória (16,777,216 bytes). Esta capacidade de endereçamento juntamente com a unidade de administração de memória permite que programas grandes e modulares sejam desenvolvidos sem recorrerem a incômoda técnica de paginação.

2.1.4 Modos de endereçamento

As instruções no *MC68000* contêm dois tipos de informações: o tipo da função a ser executada e a localização dos operandos sobre os quais a operação será efetuada. Os métodos utilizados para localizar os operandos são especificados nas instruções de uma das três formas:

Especificação do registrador: o número do registrador é fornecido pelo campo de registrador da instrução.

Endereço efetivo: uso de um dos modos de endereço efetivo. O endereçamento efetivo é composto de dois campos de 3-bits: o campo de modo e o campo do registrador. O valor contido no campo de modo seleciona os diferentes modos de endereçamento. O campo do registrador contém o número de um registrador.

Referência implícita: a definição de certas instruções implica no uso de registrador específico.

Os catorze modos de endereçamento disponíveis no *MC68000* estão distribuídos da seguinte forma:

- Modo de registrador direto
  - Registrador de dado direto
    O operando está no *registrador de dados* especificado pelo campo do registrador de endereçamento efetivo.
  - Registrador de endereço direto
    O operando está no *registrador de endereço* especificado no campo do registrador de endereçamento efetivo.

- Modo de endereçamento de memória

  - Endereçamento de registrador indireto
    O endereço do operando está no *registrador de endereço* especificado no campo do registrador.
  - Endereçamento de registrador indireto com pos-incremento
    O endereço do operando está no *registrador de endereço* especificado no campo do registrador. Após o uso do endereço do operando, ele é incrementado de 1, 2 ou 4, dependendo se o tamanho do operando é *byte, word ou long word.*
  - Endereçamento de registrador indireto com pré-decremento
    O endereço do operando está no *registrador de endereço* especificado no campo do registrador. Antes do uso do endereço do operando, ele é decrementado de 1, 2 ou 4, dependendo se o tamanho do operando é *byte, word ou long word.*
  - Endereçamento de registrador indireto com deslocamento
    Este modo de endereçamento requer a extensão de uma palavra. O endereço do operando é a soma do endereço contido no *registrador de endereço* e o deslocamento na palavra de extensão, estendido para 16 *bits* com propagação do *bit* do sinal.
  - Endereçamento de registrador indireto com índice
    O endereço do operando é a soma do endereço contido no *registrador de endereço*, o deslocamento, estendido com sinal, contido nos oito *bits* de mais baixa ordem da palavra de extensão e o conteúdo do registrador de índice.

- Modo de endereçamento especial
  Este modo de endereçamento usa o campo do registrador de endereço efetivo para especificar o modo de endereçamento especial ao invés do número do registrador.

  - Endereçamento absoluto curto
    Este modo requer uma palavra de extensão. O endereço do operando está na palavra de extensão.
  - Endereçamento absoluto longo
    Este modo requer duas palavras de extensão. O endereço do operando é obtido pela concatenação das palavras de extensão.

- Contador de programa relativo

  - Contador de programa com deslocamento
    Este modo requer uma palavra de extensão. O endereço do operando é a soma do endereço no contador de programa e o deslocamento de 16 *bits* estendido com sinal na palavra de extensão. O valor contido no contador de programa é o endereço da palavra de extensão.
  - Contador de programa com índice
    O endereço é a soma do endereço do contador de programa, o deslocamento *sign-extended* contido nos oito *bits* da palavra de extensão e o conteúdo do registrador de índice.

- Dado imediato
  Este modo requer uma ou duas palavras de extensão, dependendo do tamanho da operação.

  - Operação com byte: o operando está no *byte* de mais baixa ordem da palavra de extensão.
  - Operação com palavra: o operando é a palavra de extensão.
  - Operação com palavra longa: o operando está nas duas palavras de extensão.

2.1.5 Instruções

O conjunto de instruções do *MC68000* é dividido em 8 grupos funcionais, apresentados a seguir:

- Transferência de dados.
- Aritmética.
- Lógica
- Manipulação de bits (deslocamento e rotação).
- BCD (*Binary coded decimal*)
- Controle de programa.
- Controle do processador.

Instrução *add*

ADD

sintaxe:
ADD <ea>, Dn
ADD Dn,<ea>

operação:
(<Dn>) + (<ea>) → <Dn>
(<ea>) + (<Dn>) → <ea>

Descrição: Soma o operando fonte com o operando destino e armazena o resultado na localização do destino. O tamanho da operação pode ser *byte*, palavra ou palavra longa. O modo da instrução indica qual é o operando fonte e qual é o destino, assim como, o tamanho da operação. Esta instrução, como pode ser vista na operação, pode ser efetuada nas duas direções.

ADDA

sintaxe: ADDA <ea>, An (soma endereço)
operação: (fonte) + (destino) → destino
Descrição: Soma o operando fonte com o operando destino contido no registrador de endereço, e armazena o resultado no registrador de endereço. O tamanho da operação pode ser palavra ou palavra longa.

ADDI

sintaxe: ADDI #<dado>,<ea>
operação: Dado imediato + (destino) → destino
Descrição: Soma o dado imediato com o operando destino e armazena o resultado na localização do destino. O tamanho da operação pode ser *byte*, palavra ou palavra longa.

ADDQ

sintaxe: ADDQ #<dado>,<ea>
operação: Dado imediato + (destino) → destino
Descrição: Soma o dado imediato com o operando contido na localização do destino. O tamanho da operação pode ser *byte*, palavra ou palavra longa.

ADDX

sintaxe:
ADDX Dy,Dx
ADDX -(Ay), -(Ax)

operação: (fonte) + (destino) + X → destino

Descrição: Soma o operando fonte com o operando destino e *extended bit* e armazena o resultado na localização do destino. Os operandos podem ser endereçados de duas maneiras:

1. registrador de dados com registrador de dados: os operandos estão contidos nos registradores de dados especificados na instrução;
2. memória com memória: os operandos são endereçados com o modo de endereçamento predecremento usando o registrador de endereço especificado na instrução. O tamanho da operação pode ser *byte*, palavra ou palavra longa.

2.2 VAX-11

2.2.1 Instruction-set Processor

Os processadores da família *VAX* [VAX 82] oferecem dezesseis registradores de 32 *bits* de carater geral, que podem ser utilizados para armazenamento de temporários, como acumuladores, registradores de índices e registradores bases. Os registradores são distribuídos da seguinte forma: doze registradores *R0-R11* podem ser usados como registradores de

propósito geral. Os quatro restantes possuem significado especial dependendo da instrução que está sendo executada. Seus usos são apresentados na seção seguinte.

Um operando de uma instrução pode estar localizado na memória principal, em um registrador de caráter geral ou na própria instrução. O método através do qual a localização de um operando é especificado é denominado modo de endereçamento do operando.

2.2.2 Registradores

O sistema *VAX* provê 16 registradores gerais. Os registradores são denotados por "*Rn*" e "*R[n]*", onde *n* é um inteiro entre *0 e 15*. Certos registradores possuem uso específico e nomes especiais:

PC: o registrador *R15* é o contador de programa. Este registrador não é usado como temporário, acumulador ou registrador de índice.

SP: o registrador *R14* é o apontador de pilha.

FP: o registrador *R13* é o apontador de *Frame*. Na arquitetura *VAX* a execução de uma chamada de procedimento constrói uma estrutura de dados na pilha denominado *stack frame*. A função do registrador *FP* é carregar o endereço base da *stack frame* no momento que uma instrução *call* é executada.

AP: o registrador *R12* é o apontador de argumento. Na arquitetura *VAX* a execução de uma chamada de procedimento usa uma estrutura de dados denominada, lista de argumentos e necessita o registrador *AP* como o endereço base desta lista.

R6-R11: os registradores, *R6* até *R11*, não possuem significado especial para o *hardware* ou para o sistema operacional.

R0-R5: estes registradores são, geralmente, disponíveis para uso geral, mas também podem ser carregados com valores específicos por instruções, cuja execução deva ser interrompida, por exemplo, aritmética decimal etc.

2.2.3 Endereçamento de Memória

O espaço de endereçamento físico contido no controlador da memória é dividido em duas áreas: endereçamento físico de memória e endereçamento de entrada e saída. O endereçamento físico é de 24-bits, o que o torna capaz de endereçar 16 megabytes. Para acessar este espaço de endereçamento, o sistema possui dois tipos de operações: a operação de leitura e a operação de escrita.

2.2.4 Modos de endereçamento

Os modos de endereçamento do *VAX* podem ser divididos em duas categorias, modo de endereçamento geral e modo de endereçamento de desvio.

O modo de endereçamento geral classifica-se em:

- Modo registrador *Rn*
  Neste modo, o operando é o conteúdo do registrador *n*.
- Modo registrador indireto *(Rn)*
  O registrador contém o endereço do operando.
- Modo auto-incremento *(Rn)+*
  Rn contém o endereço do operando. Após o operando ser determinado o seu tamanho é somado ao conteúdo de *Rn* e substitui-se o conteúdo do mesmo pelo resultado obtido.
- Modo auto-incremento indireto *@(Rn)+*
  Rn contém o endereço de uma palavra longa, a qual é um apontador para o endereço do operando. Após o endereço do operando ser determinado, adiciona-se quatro ao conteúdo de *Rn* e o conteúdo de *Rn* é substituído pelo resultado obtido.
- Modo auto-decremento *-(Rn)*
  Neste modo de endereçamento, o tamanho do operando é subtraído do conteúdo do *Rn* e o conteúdo de *Rn* é substituído pelo resultado obtido.
- Modo literal *S↑#literal*
  Este modo de endereçamento especifica constantes inteiras.
- Modo deslocamento *D(Rn)*
  Neste modo de endereçamento, o deslocamento é adicionado ao conteúdo do registrador *Rn*, e o resultado é o endereço do operando. Este modo é equivalente ao modo de endereçamento indireto do *PDP-11*.
- Modo indireto com deslocamento *@D(Rn)*
  Neste modo de endereçamento, o deslocamento é somado ao conteúdo do registrador *Rn* selecionado, e o resultado é um endereço de palavra longa contendo o endereço do operando.
- Modo indexado *i[Rn]*
  Neste modo de endereçamento, o especificador de operando consiste em dois *bytes*, o especificador primário do operando e o especificador operando base. O endereço do especificador primário do operando é determinado primeiro multiplicando-se o conteúdo do registrador de índice *Rx* pelo tamanho do operando primário. Este valor é, então, somado ao endereço dado pelo especificador do operando base, e o resultado é o endereço do operando.

Existem disponíveis várias formas do modo de endereçamento de índices:

- Registrador de índice indireto *(Rn)[Rx]*
- Auto-incremento indexado *(Rn)+[Rx]*
- Imediato indexado *i#constant[Rx]*
- Auto-incremento indireto indexado *@(Rn)+[Rx]*
- Absoluto indexado *@#address[Rx]*
- Auto-decremento indexado *-(Rn)[Rx]*
- Deslocamento indexado de byte, palavra ou palavra longa
  B↑D(Rn)[Rx],
  W↑D(Rn)[Rx] e
  L↑D(Rn)[Rx] respectivamente.
- Deslocamento indexado indireto de byte, palavra ou palavra longa
  @B↑D(Rn)[Rx],
  @W↑D(Rn)[Rx] e *@L↑D(Rn)[Rx]* respectivamente.

- Endereçamento de desvio *A*
  Neste modo de endereçamento um deslocamento de um *byte*, ou uma palavra é estendido a 32 *bits*,com propagação de sinal, e somado ao conteúdo do *PC* atualizado. O conteúdo de *PC* atualizado é o endereço do primeiro *byte* após o especificador do operando.

2.2.5 Instruções

O conjunto de instruções do *VAX-11* é muito versátil. Ele inclui instruções de inteiros, decimal compactado, *strings* de caracteres, ponto flutuante, além de instruções especiais.

O *VAX-11* pode processar os seguintes tipos de dados: *bits* (até 32 *bits*), palavras (16 *bits*), palavra longa (*32 bits*), *quadwords (64 bits*).

Instrução ADD: A instrução *ADD* pode ser de dois ou três operandos e o seu mneumônico é formado pela palavra *ADD* seguida da letra inicial de um dos seguintes tipos de dados:
Byte;
Word;
Long-word;
F-floating, para ponto flutuante de precisão simples,
D-floating, para indicar uma operação de ponto flutuante de precisão dupla,
G-floqting, para dados de oito *bytes* contínuos e
H-floating, para dados de dezesseis *bytes* contínuos,
além do número de operandos.

ADD:

operação:
sum ← sum + addend; (2 operandos)
sum ← addend1 + addend2; (3 operandos)

código de condição:
N ← sum LSS 0;
Z ← sum EQL 0;
V ← *overflow*
C ← *carry* para o *bit* mais significativo (inteiro)
C ← 0 (ponto flutuante).

Descrição: No formato de dois operandos, o operando addend é somado ao operando *sum*, o qual é substituído para conter o resultado da operação. No formato de três operandos, o operando *addend* 1 é somando ao operando *addend* 2, e o operando *sum* é substituído pelo resultado da operação.

ADDB2: soma *byte* de dois operandos,

ADDB3: soma *byte* de três operandos,

ADDW2: soma *word* de dois operandos,

ADDW3: soma *word* de três operandos,

ADDL2: soma *long word* de dois operandos,

ADDL3: soma *long word* de três operandos,

ADDF2: soma *F-floating* de dois operandos,

ADDF3: soma *F-floating* de três operandos,

ADDD2: soma *D-floating* de dois operandos,

ADDD3: soma *D-floating* de três operandos,

ADDG2: soma *G-floating* de dois operandos,

ADDG3: soma *G-floating* de três operandos,

ADDH2: soma *H-floating* de dois operandos,

ADDH3: soma *H-floating* de três operandos.

2.3 PDP-11

2.3.1 Instruction-set Processor

Os processadores da família *PDP-11* oferecem 8 registradores de 16 *bits*, que podem ser utilizados como acumuladores, registradores de índices ou como apontadores de pilha para armazenamento de dados em temporários. Os registradores *R0-R5* são registradores de propósito geral. O registrador *R7* é utilizado como o endereço da próxima instrução a ser executada. O registrador *R6* é normalmente utilizado como apontador de pilha e indica a última posição ocupada.

O processador pode efetuar transferência de dados diretamente entre os dispositivos de entrada e saída e a memória. Ele possui endereçamento de operando simples e de operando duplo e opera com dados de tamanho *bytes (8 bits)* e palavras *(16 bits)*.

2.3.2 Endereçamento de Memória

Os endereços gerados pela unidade de processador central da família do *PDP-11* são de *18 bits* de endereçamento direto. Muito embora o comprimento da palavra seja de *16 bits*, o endereçamento lógico do *UNIBUS* e *CPU* possui *18 bits* de comprimento. Assim, enquanto a palavra do *PDP-11* pode endereçar *64K bytes* (*32K words*), a *CPU* e o *UNIBUS* podem endereçar *256K bytes (128K words)*. Estes dois *bits* extras de endereçamento lógico provêem uma forma básica para expandir referências de memória.

2.3.3 Modos de endereçamento

As instruções no *PDP-11* contêm três tipos de informações: a função a ser executada, um registrador de uso geral para localizar os operandos e um modo de endereçamento para especificar como o registrador selecionado será usado. Operações de dois operandos são efetuadas por instruções que especificam dois endereços. O primeiro operando é denominado operando fonte, e o segundo, operando destino. Os modos de endereçamento do *PDP-11* são:

- Endereçamento direto *Rn*
  - Registrador
    O registrador contém o operando.
  - Modo Auto-incremento *(Rn)+*
    O conteúdo do registrador é o endereço do operando. Soma-se ao seu conteúdo 1 para *byte*, 2 para *palavra* para endereçar a próxima localização sequencial.

- Modo auto-decremento *-(Rn)*
  O conteúdo do registrador selecionado é decrementado do tamanho do operando e então usado como o endereço do operando.
- Modo indexado *X(Rn)*
  O valor *X* é somado ao *(Rn)* para produzir o endereço do operando. *X* e *(Rn)* não são modificados.

- Modo de endereçamento indireto
  - Registrador indireto *@Rn* ou *(Rn)*
    O registrador contém o endereço do operando.
  - Modo Auto-incremento indireto *@(Rn)+*
    O registrador é utilizado primeiramente como um apontador para uma palavra contendo o endereço do operando, então incrementado sempre por 2, mesmo que seja uma instrução *byte*.
  - Modo auto-decremento indireto *@-(Rn)*
    O conteúdo do registrador selecionado é decrementado de 2, mesmo que a instrução seja *byte* e então utilizado como um apontador para uma palavra contendo o endereço do operando.
  - Modo indexado indireto *@X(Rn)*
    O valor *X* armazenado em uma palavra seguindo a instrução e *(Rn)* são somados e a soma é usada como um apontador para a palavra contendo o endereço do operando. *X* e *Rn* não são modificados.

- Registrador 7 - contador de programa como um registrador geral
  - Imediato *#n*
    O operando segue a instrução. Este modo é equivalente a usar o modo autoincremento com *PC*.
  - Absoluto *@#A*
    O endereço absoluto segue a instrução. Este modo é equivalente a usar o modo imediato indireto ou autoincremento indireto usando o *PC*.
  - Relativo *A*
    O endereço relativo, valor do índice, segue a instrução.
  - Relativo indireto *@A*
    O valor do índice, armazenado em uma palavra seguindo a instrução, é o endereço relativo para o endereço do operando.

- Registrador 6 - apontador de pilha como um registrador geral
  - Auto-decremento com registrador 6
    Empilha dado.
  - Auto-incremento com registrador 6
    Desempilha dado.

– Modo indexado com registrador 6
Permite acesso randômico aos elementos da pilha.

2.3.4 Instruções

O conjunto de instruções do *PDP-11* classificam-se em dois grupos: instruções de um operando e instruções de operando duplo.

ADD:

Operação:
(dst) ← (src) + (dst)

Código de condição:
N: liga se o resultado < 0; senão limpa.
Z: liga se o resultado = 0; senão limpa.
V: liga se houve *overflow* como resultado da operação; senão limpa.
C: liga se houve *carry* no *bit* mais significativo do resultado, senão limpa.

Descrição: Soma o operando fonte (src) ao operando destino (dst), e armazena o resultado no endereço do destino. O conteúdo original do destino é perdido mas o conteúdo do fonte não se altera.

ADD 20,R0: soma para o registrador.

ADD R1,XXX: soma para a memória.

ADD R1,R2: soma registrador com registrador.

ADD @#17750,XXX: soma memória com memória.

XXX é um mneumônico definido pelo programador para representar uma posição na memória.

3 Uma Linguagem para Descrição de Arquiteturas

A abordagem de Davidson [DAVIDSON 81] objeto das seções 3.1, 3.2 e 3.3 compreende uma notação para a descrição da sintaxe e efeito de cada instrução do conjunto de instruções para máquinas reais.

3.1 Conjunto de instruções do processador

Para trabalhar com computadores hoje a nível de código de máquina, o pesquisador sabe o que significa para ele familiarizar-se com conjuntos de instruções de novos processadores.

Este não é um desafio intelectual, contudo não é fácil. De um lado todos os processadores são semelhantes, existe um pequeno conjunto de funções no sentido algébrico (*operações aritméticas, lógicas e algumas mais*). Todas elas mantêm os operandos das operações em células de memória de vários tamanhos e provêem um número limitado de mecanismos para acessar estas células. Não há realmente grandes diferenças nas noções básicas necessárias para o entendimento de conjuntos de instruções tradicionais. A dificuldade encontrada ao estudar novos conjuntos de instruções está na maneira em que estes conceitos gerais são combinados; por exemplo como o armazenamento do processador é arranjado, como as operações podem endereçar seus operandos, e quais operações são combinadas para formar uma simples instrução.

Na abordagem de Davidson, os efeitos das instruções são descritos usando uma notação semelhante a *ISP* [BELL 71]. Por exemplo, a instrução da máquina *DEC-10*

`add 3,loc` é expressa em *ISP* como: `r[3] ← r[3]+m[loc]`

e significa que o conteúdo da posição de memória `loc` é somada ao registrador 3.

Muitas instruções de máquina possuem múltiplos efeitos. Tais instruções podem conter várias transferências de registradores. Por exemplo, a instrução

`aoje 5,L1` da máquina *DEC-10* é expressa em *ISP* da seguinte maneira:

`r[5] ← r[5]+1;PC ← if r[5]+1 = 0 then L1 else PC.`

A primeira transferência de registradores especifica que o registrador 5 é incrementado, a segunda especifica que se o resultado obtido é zero, então um desvio é efetuado para o rótulo `L1`, senão a execução continua com a próxima instrução. Note que o "`+1`" é necessário na segunda transferência de registradores porque toda transferência de registradores é efetuada em paralelo.

Além destas considerações, detalhes irrelevantes à descrição de máquina podem ser omitidos. Por exemplo, a instrução

`tst r1` do *PDP-11* é expressa em *ISP* como: `NZ` $\leftarrow$ $r[1]$? 0;

onde, `NZ` representa o registrador de código de condição. Nesta instrução, o conteúdo do registrador "`1`" é comparado com zero, e o registrador de código de condição é atualizado de acordo com o resultado da operação. Os geradores de código não necessitam saber como o registrador de código de condição representa o resultado da comparação, assim sendo, a semântica do operador "?" não precisa ser especificada na descrição da máquina.

3.2 Descrição de Máquina

Na abordagem de Davidson, a arquitetura de máquina é especificada por meio de uma gramática para tradução dirigida por sintaxe entre a linguagem de montagem da máquina alvo e a transferência de registradores. As produções nesta gramática são compostas de expressões e comandos simples envolvendo os registradores e células de memória da máquina alvo. A linguagem proposta provê ainda facilidades para definir não-terminais, a sintaxe em linguagem de montagem para cada não-terminal e a sintaxe para a correspondente transferência de registradores e modos de endereçamento.

A descrição de máquina é logicamente dividida em duas partes. A primeira parte descreve os modos de endereçamento da máquina, e a segunda parte descreve as instruções. Esta divisão proporciona um modo natural de descrever arquiteturas de máquinas. Os modos de endereçamento são descritos sem considerar as instruções que os usam. As operações de máquina e os modos de endereçamento são combinados para descrever o conjunto de instruções da máquina. A vantagem de estruturar o conhecimento sobre uma arquitetura desta forma produz como resultado, uma descrição de máquina concisa e legível.

No trabalho de Davidson, o reconhecedor e o tradutor mencionados na seção 1.1.5 são obtidos transformando-se a descrição de máquina em uma gramática para o gerador de analisadores léxicos, *LEX*, [AHO 86]. A partir da entrada, *LEX* gera subrotinas que implementam o reconhecedor e o tradutor. Como *LEX* reconhece somente expressões regulares, a máquina alvo deve ser descrita usando-se expressões regulares.

Muitas máquinas possuem componentes em suas instruções que são sensíveis ao contexto. Por exemplo, a instrução *add* do computador *DEC-10* pode ser descrita como:

`RG` ← `RG + M`, onde `RG` representa um registrador, e `M` representa uma posição de memória. Quando um não-terminal aparece duas vezes em uma produção os *strings* casados pelos padrões devem ser idênticos. Neste exemplo, as duas instâncias de `RG` devem casar o mesmo *string*.

3.3 Exemplo de uma descrição de máquina

Esta seção apresenta partes da descrição de máquina para o *PDP-11*.

```
RN      [0-7]+
XDENT (((“_” | “L”)[A-Za-z0-9_]+) | (-?[0-9]+)
IDENT  XDENT(“ ” [-+] “ ” XDENT)*
LABEL  “L” [L0-9]+
```

Esta parte da descrição define expressões regulares utilizadas ao longo da descrição dos modos de endereçamento. A primeira definição descreve um número de registrador: uma sequência de um ou mais dígitos entre 0 e 7. A segunda definição descreve um componente de um identificador. `IDENT` define um identificador: um `XDENT` seguido de zero ou mais ocorrências dos operadores "+" ou "-" seguindo por um `XDENT`. A última definição descreve rótulos. A seguir são definidos os modos de endereçamento.

RG	:= r[RN]	:= rRN
LB	:= LABEL	:= LABEL
	:= 0	:= $0
	:= 1	:= $1
ID	:= IDENT	:= $IDENT
WORD	:= m[IDENT]	:= IDENT
	:= m[r[RN] + IDENT]	:= IDENT(rRN)
	:= m[r[RN]++]	:= (rRN)+
	:= m[−−r[RN]]	:= -(rRN)
	:= m[r[RN]]	:= (rRN)
	:= m[m[r[RN]++]]	:= *(rRN)+
	:= m[m[−−r[RN]]]	:= *-(rRN)
	:= m[m[r[RN] + IDENT]]	:= *IDENT(rRN)
	:= m[m[IDENT]]	:= *IDENT
	:= m[m[r[RN]]]	:= *(rRN)
BT	:= b[IDENT]	:= IDENT
	:= b[r[RN] + IDENT]	:= IDENT(rRN)
	:= b[r[RN]++]	:= (rRN)+
	:= b[−−r[RN]]	:= -(rRN)
	:= b[r[RN]]	:= (rRN)
	:= b[m[r[RN]++]]	:= *(rRN)+
	:= b[m[−−r[RN]]]	:= *-(rRN)
	:= b[m[r[RN] + IDENT]]	:= *IDENT(rRN)
	:= b[m[IDENT]]	:= *IDENT
	:= b[m[r[RN]]]	:= *(rRN)
REL	:= ==	:= eq
	:= =	:= ne
	:= ≥	:= ge
	:= ≤	:= le
	:= <	:= lt
	:= >	:= gt
SO	:= <<	
NZ	:= NZ	
PC	:= PC	

Cada linha da definição acima possui três campos: o *token* retornado pelo reconhecedor se o padrão para a transferência de registradores casa, a sintaxe na forma *ISP* para o padrão, e a sintaxe em linguagem de montagem para o padrão. Por exemplo, a primeira linha acima define um registrador *r* seguido pelo número do registrador entre colchetes. A sintaxe em linguagem de montagem correspondente é *r* seguido pelo número do registrador. Quando um registrador é reconhecido, o *token RG* é retornado.

Se o primeiro campo for vazio, o *token* retornado é o *string* casado, como aparece na terceira linha da definição. O último campo correspondente a sintaxe em linguagem de montagem para o padrão também é opcional, como mostra a definição de `SO` (operador *shift*).

A parte final da definição de endereço agrupa os *tokens* e os caracteres simples em classes. Este agrupamento permite uma definição simples e concisa das instruções, pela combinação dos operadores e modos de endereçamentos semelhantes.

```
RG1  := RG
RG2  := RG
DSTW := RG | WORD
SRCW := RG | ID | WORD | 0 | 1
```

A sintaxe para a definição de instruções é:

instruction expression := *instruction definition*
instruction definition := *action*
| { [*test condition* :] *action*
[*test condition* :] *action*
...
}

Colchetes representam um campo opcional, e "|" separa as alternativas. Elipses ("...") representam repetição indefinida de itens; *instruction expression* é a representação em *ISP* da instrução; *action's* são executadas se o *test condition* a ela associada é verdadeiro; *test condition* é avaliado na ordem em que aparece. Uma *instruction expression* casa com o *string* de entrada se ela efetua todas as transferências de registradores requisitadas. Segue dois exemplos de definição de instruções para o *PDP-11*, instruções *cmp*, *asr* e *ash*:

```
NZ ← DSTW ? SRCW;  := cmp DSTW,SRCW
```

No exemplo abaixo, o *test condition* pode invocar procedimentos fornecidos pelo sistema ou pelo usuário. O *test condition* permite que *instruction expression* identifique duas instruções. Se a rotação (*shift*) é "-1", então a instrução *asr* casa, senão a instrução

mais geral *ash* casa. Isto é possível através do procedimento *strcmp*. Este procedimento compara dois *strings* e retorna verdadeiro se eles são iguais, caso contrário, retorna falso.

```
RG ← RG SO SRCW; NZ ← RG SO SRCW ? 0; := {
        !strcmp(SRCW, "-1") :  asr RG
                ash SRCW,RG }
```

3.4 Comparação com outros trabalhos

Os diversos trabalhos apresentados na seção 1.1 utilizam-se da descrição de máquina para obter a independência das arquiteturas de máquina. Uma vez que cada método depende fortemente destas descrições, vale a pena comparar e contrastar as várias abordagens de descrição. A instrução "add" do *PDP-11* é utilizada como base para a comparação. Entretanto, existem algumas abordagens que não fornecem informações suficientes para elaborar um exemplo nesta máquina, nestes casos, as arquiteturas a que a intrução "add" se refere são explicitamente indicadas antes do exemplo.

3.4.1 Abordagem de Glanville

Na abordagem de Glanville uma instrução *add* é descrita da seguinte maneira:

r.1 ::= (+r.1 r.2)	"add r.2,r.1";
r.1 ::= (+k.1 r.1)	"add $k.1,r.1";
r.1 ::= (+ ↑ k.1 r.1)	"add *k.1,r.1";
λ ::= (:= k.1 + ↑ k.1 r.1)	"add r.1, *k.1";
r.2 ::= (+ ↑ + k.1 r.1 r.2)	"add k.1(r.1), r.2";
λ ::= (:= + k.1 r.1 + ↑ + k.1 r.1 r.2)	"add r.2, k.1(r.1)";
λ ::= (:= + k.1 r.1 + ↑ + k.1 r.1 ↑ + k.2 r.2)	"add k.2(r.2), k.1(r.1)";
λ ::= (:= + k.1 r.1 + k.2 ↑ + k.1 r.1)	"add $k.2, k.1(r.1)";

`r.1` e `r.2` representam registradores; `k.1` e `k.2` representam constantes. ↑ representa um operador unário que recupera o valor da posição de memória endereçada por seu operando. A instrução na linguagem de montagem (*assembler*) que aparece à direita da produção corresponde a instrução a ser emitida se a produção casa com a entrada.

3.4.2 Abordagem de Ganapathi

Na abordagem de Ganapathi uma instrução *add* é descrita da seguinte maneira:

```
Word↑ → + Word↑a Word↑r Istemp(↓r) EMIT (↓'add' ↓a ↓r)
      → + Word↑a Word↑b GETTEMP(↓'word' ↑r)
            EMIT(↓'mov' ↓b ↓r)
            EMIT(↓'add' ↓a ↓r)
```

Todas as produções são da forma "*LHS* → *RHS*". O *LHS* é um único não-terminal, normalmente contendo atributos sintetizados. O *RHS* contêm terminais com atributos sintetizados, não-terminais com atributos sintetizados, predicados para retirada de ambiguidade (sublinhados) com atributos herdados e símbolos de ação (letras maiúsculas), possuindo atributos sintetizados e herdados. ↑ representa um atributo sintetizado que transmite informação acima na árvore; ↓ representa um atributo herdado, ele passa informação para baixo na árvore sintática. As variáveis `a, b e r` são variáveis de atributo. `Word` representa o modo de endereçamento baseado em palavra disponível no computador *PDP-11*. `Istemp` é um predicado responsável pela retirada de ambiguidade (*disambiguating predicate*), ele determina quando a produção corrente pode ser aplicada. `GETTMP` requisita um temporário, e `EMIT` gera a instrução em linguagem de montagem se a produção casa com a entrada.

3.4.3 Abordagem de Cattell

Na abordagem de Cattell instruções são sintaticamente representadas sob a forma de assertivas escritas em notação parentetizada no estilo *LISP*. A descrição da instrução *add* é:

```
(; (← $1:DST (+ $1:DST $2:SRC)) (← %N (LSS (+ $1:DST $2:SRC) 0))
   (← %Z (EQL (+ $1:DST $2:SRC) 0)) :: (EMIT [ADD 2 1 1]6 $2 $1)
```

As instruções são representadas como produções. O lado esquerdo é formado pelas asserções de entrada e saída que definem as ações das instruções. Os componentes do lado direito, a partir de `EMIT`, especificam o custo espaço/tempo, formato, mneumônico e uma lista de valores dos campos das instruções.

`DST` e `SRC` representam os modos de endereçamento com palavras disponíveis no computador *PDP-11*. Eles são descritos em uma parte separada da descrição. `$1` é usado para indicar que os registradores de destino e do operando devem ser o mesmo registrador. Ambos "`$1` e `$2`" são usados para referências posteriores na determinação dos campos reais, isto é, número dos registradores. "2" indica operação de dois operandos. Os dois "1's" representam o custo relativo ao espaço/tempo respectivamente.

3.4.4 Abordagem de Costa

Na abordagem de Costa, a instrução *add* pode ser descrita da seguinte maneira:

```
regra modoI    : modoI := k;                  FazDesloc k;
regra modoR    : modoR:= r;                   FazBase r;
regra modo1    : modo1 := + modoI modoR;      SomaDescrs modoI modoR;
regra modo2_1 : modo2 := ↑ modo1;             CopiaDescr modo1; IncNivel modo2;
regra modo2_2 : modo2 := modoI;               CopiaDescr modoI;
regra modo2_3 : modo2 := modoR;               CopiaDescr modoR;
regra modo3_1 : modo3 := ↑ modo2;             CopiaDescr modo2; IncNivel modo3;
regra modo3_2 : modo3 := modo2;               CopiaDescr modo2;
```

```
regra addword1 : r.1                 :=              word + modo3 r.1;
                 Emite 'add'         modo3 ',' r.1;  custo 2;
regra addword2 : r.1                 :=              word + r.1 modo3;
                 Emite 'add'         modo3 ',' r.1   custo 2;
regra addword3 : lambda              :=              word st modo3 word + ↑ modo3 modo2;
                 IncNivel modo3;
                 Emite 'add word'    modo2 ',' modo3; custo 2;
regra addword3 : lambda              :=              word st modo3 word + modo2 ↑ modo3;
                 IncNivel modo3;
                 Emite 'add word'    modo2 ',' modo3; custo 2;
```

`FazBase`, `FazDesloc`, `CopiaDescr`, `IncNivel` e `SomaDescrs` são operadores semânticos disponíveis na linguagem descritiva. `modo1`, `modo2`, `modo3`, `modoI` e `modoR` são não-terminais definidos para agrupar os diversos modos de endereçamento. `r.1` representa registrador; k representa as constantes. ↑ representa um operador unário que recupera o valor da posição de memória endereçada por seu operando.

3.4.5 Abordagem de Davidson

Na abordagem de Davidson [DAVIDSON 81], uma instrução *add* é descrita da seguinte maneira:

```
DSTW ← DSTW + SRCW; NZ ← DSTW + SRCW ? 0; := add SRCW, DSTW
```

DSTW e SRCW representam os modos de endereçamento com palavras disponíveis no computador *PDP-11*. NZ representa o registrador de código de condição. A instrução em linguagem de montagem que aparece a direita de ":=" é emitida se a expressão representando a instrução se casa com a entrada.

3.4.6 Abordagem de Robert R. Kessler

Na abordagem de Kessler, uma instrução *add* para a arquitetura *MC68000* é descrita da seguinte maneira:

```
((add.1 EA-All Dn)
(Make-Flags (setf Dn (+ Dn EA-All)) Dn * * * * *)
(+ 8 (EA-Time EA-All 4)) % Time & Space depend
(long EA-An)) % on Adrressing mode
```

`EA-All` representa todos os modos de endereçamento para a arquitetura *MC68000*, `EA` significa o modo de endereçamento efetivo. `EA-An` representa `EA` sem `An`. `Dn` representa o registrador de dados. Os "*" representam os códigos de condição, `X`, `N`, `Z`, `V` e `C` respectivamente. `Make-flags` é uma macro que a partir de uma expressão e uma lista de código de condições retorna uma expressão com todos os valores definidos.

3.4.7 Abordagem de Giegerich

Na abordagem de Giegerich, uma instrução *add* para a arquitetura *MC68000* é descrita da seguinte maneira:

ADD_1: `Dreg := Dreg + all, Cy := carry(Dreg + all),`
`Z := (Dreg + all = 0), N := (Dreg + all < 0) cost2.2;`

`Dreg` especifica o modo de endereçamento, neste caso, o registrador de dados; `all` representa a classe do operando que pode ser: `Dreg, Areg, disp, postinc, im32,...`, onde, `Areg` é o registrador de endereço, `disp` é o modo de endereçamento de palavra-dupla (base + deslocamento), `postinc` é o modo de endereçamento pós-incremento, e `im32` representa constantes. `Cy`, `Z` e `N` representam os códigos de condição, `Cy` define o ``carry'', `X` e `V` foram omitidos nesta descrição.

3.4.8 Abordagem de Peter B. Kessler:

Na abordagem de Peter B. Kessler, uma instrução de adição de palavra longa composta de três operandos `ADDL3`, para a arquitetura *VAX-11* é descrita da seguinte maneira:

(in-order
(←a-dest (+ a-src_2 a-src_1))
(any-order
(← cc-n (< a-dest 0))

(← cc-z (= a-dest 0))
(← cc-v (overflow a-dest))
(← cc-z(carry a-dest))))

Os operadores `in-order` e `any-order` representam a sequência da computação. "`cc's`" representam os códigos de condição que a arquitetura do *VAX-11* atualiza durante a execução da maioria das instruções aritméticas. A notação prefixada parentetizada é usada para mostrar como a árvore define a computação.

3.4.9 Abordagem de Fraser

Na abordagem de Fraser, uma instrução *add* é descrita da seguinte maneira:

.=="ADDI" .=%f%t3 %x,%y,%z
f = "add"
t = "1"
xm = "r%n" ym = "r%n" zm = "r%c"
yn = K0
xn = K1

A linguagem de regras possui dois operadores básicos: "==" testa, e "=" assinala. "`%f`" representa um operador binário, por exemplo, `add, sub, etc.`. "`%t`" representa o tipo do sufixo (`b, w, f, l, d`). "%y" e "%z" são os operadores gabaritos. "`yn`" e "`xn`" representam apontadores para os nodos que calculam os endereços dos operandos.

As regras podem ser abreviadas pela substituição das constantes *strings* pelos seus *placeholders* de tal forma que a regra acima definindo a instrução *add* poderia ser expressa da seguinte maneira:

.=="ADDI" .= "add13" "r%n,r%n,r%c"
yn = K0
xn = K1

Para interpretar um gabarito em linguagem de montagem, ou seja, para gerar uma saída, *placeholders* são substituídos com os valores dos campos correspondentes. Por exemplo, se *f* representa *add* e *t* representa *1* então o gabarito *(template) %f%t3* representa *add13.*

3.4.10 Abordagem de Henry

Na abordagem de Henry, uma instrução *add* é descrita da seguinte maneira:

```
Plus reg src'1
reg
cg("add2 $src'1,$reg", "$reg")
cost(1)
```

`Plus reg src'1` representa o padrão na forma prefixada; `reg` representa a substituição; `cg("add2 $src'1,$reg" "$reg")` representa a ação e `cost(1)` representa o custo. O predicado para esta instrução não foi especificado, portanto assuma-o verdadeiro.

3.4.11 Abordagem de Warfield

Na abordagem de Warfield uma instrução *add* para o *VAX* é descrita da seguinte maneira:

```
(instruction addb3)
(operation addb3 addition)
(left addb3 operand3 all-modes)
(right1 addb3 operand2 all-modes)
(right2 addb3 operand1 all-modes)
(set addb3 NZ)
(size addb3 1)
(space addb3 3)
(time addb3 3)
```

`all-modes` representa todos os modos de endereçamento, os três operandos desta instrução podem ser usados com qualquer modo de endereçamento. A proposição `set` indica que os bits do código de condição "N" e "Z" devem ser atualizados após a operação de adição. A proposição `size` representa o tamanho dos operandos, nesta operação "1" diz que `addb3` é uma instrução *byte*. As constantes especificadas nas proposições `space` e `time` de uma instrução são arbitrárias, mas devem ser consistentes com o restante das outras instruções presentes na descrição.

4 Avaliação

A descrição de máquina proposta por Davidson [DAVIDSON 81] é, de certa forma, mais fácil de ler que as outras apresentadas porque ela se utiliza da notação infixada ao invés da notação prefixada. A descrição de Ganapathi requer que o usuário implemente os

predicados de retirada de ambiguidade e as ações para cada máquina. Enquanto isto torna o método flexível no que tange as otimizações independentes da arquitetura da máquina alvo, complica a elaboração da descrição da mesma.

A descrição proposta por Glanville é substancialmente mais longa que as outras apresentadas devido aos modos de endereçamento não terem sido fatorados na definição das instruções. Este problema é resolvido na abordagem apresentada por Costa [COSTA 90]. A vantagem da técnica delineada por Costa em relação a Glanville reside, exatamente, na fatoração da gramática descritiva. Reduzindo a gramática também diminui o tamanho das tabelas geradas automaticamente pelo sistema. Contudo, o tempo gasto na geração de código pode aumentar devido as novas reduções introduzidas pela fatoração, constituindo assim uma desvantagem da abordagem de Costa.

Ambas as descrições, de Glanville e Ganapathi, são incompletas no sentido de que o teste e a atualização (*setting*) dos códigos de condição não são descritos. A mesma observação é válida para o método de Costa, se o objetivo é otimizar código. No caso da abordagem de Glanville, isto significa que nenhum código pode ser gerado ao efetuar uma adição unicamente pelo efeito colateral de alterar o código de condição. Ganapathi possui um mecanismo separado que vasculha por instruções que são utilizadas somente para ajustar o código de condição. Caso ele ajuste o código de condição exatamente como na instrução anterior, a instrução é suprimida. Como a descrição da instrução *add* não especifica explicitamente que a instrução ajusta o código de condição, supõe-se que esta informação deva vir de alguma outra fonte.

A descrição de máquina de Cattell é mais complexa que as outras apresentadas. A definição de máquina através de axiomas não é prática para uma variedade de instruções de máquina. Sua abordagem em termos de complexidade compara-se com a abordagem de Ganapathi, muito embora os métodos utilizados por ambos sejam diferentes.

As demais técnicas apresentadas na seção 1.1 [KESSLER 84], [KESSLER 86], [HENRY 87], [HENRY 89b], [HENRY 89a], [FRASER 89] não oferecem informações suficientes para poder avaliá-las, contudo, de todas estas abordagens, a que parece mais simples é aquela proposta por R. Kessler [KESSLER 84].

A descrição de Giegerich é baseada na formalização da semântica do conjunto de instruções. Ela compreende um modelo abstrato de conjunto de instruções de uma maneira geral, uma notação para a descrição de máquina real baseada em *ISP* [BELL 71], 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.

A abordagem utilizada por Warfield e Bauer 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 85] que está disponível somente em três máquinas, elas são: *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 técnica de descrição utilizada por Davidson provê um método simples, muito embora robusto, para descrever instruções de máquina. Detalhes irrelevantes da arquitetura da máquina podem ser omitidos, enquanto que aqueles complicados, mas necessários podem ser facilmente descritos.

Todas descrições de arquiteturas apresentadas neste trabalho, através de exemplos, possuem algumas características em comum. Em primeiro lugar, elas definem modos de endereçamento e instruções das arquiteturas de máquina, algumas de forma mais complicada, outras mais simples do ponto de vista de legibilidade. Em segundo lugar, quase todas utilizam a notação *ISP* ou adaptação dela [BELL 71] como base nas descrições. Isto, acredito, seja porque *ISP* provê um mecanismo preciso e sem ambiguidades para especificar arquiteturas de máquinas. Em terceiro lugar, quase todas processam algum tipo de gramática. Glanville e Costa usam gramáticas livres do contexto, Ganapathi utiliza gramáticas de atributos, Davidson usa expressão regular. Outros utilizam-se de regras, por exemplo, Fraser e Wendt, Henry, Warfield e Bauer. E finalmente quase todas fatoram de alguma forma os diferentes modos de endereçamento das arquiteturas em questão.

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. Outras informações, como por exemplo, mecanismos para atualizar o código de condição utilizado nas operações lógico-aritméticas tornam as notações mais poderosas.

A aplicabilidade dos métodos apresentados não varia muito, a maioria deles visa as arquiteturas *PDP-11, VAX-11, MOTOROLA 68000.* Por exemplo, Glanville discute a implementação de geradores de código para as máquinas *PDP-11* e *IBM-370.* Ganapathi produziu geradores de código para as arquiteturas *VAX-11, PDP-11* e *Intel 8086/8087.* Cattell discute a implementação de geradores de código para *IBM-360, PDP-10, PDP-11, Intel 8080, Motorola 6800* e *PDP-8.* Costa apresenta a implementação de um sistema de produção de geradores de código para o processador *Intel 8088*; sua técnica também é aplicável aos processadores *IBM-360* e *PDP-11.* Robert Kessler discute a implementação de *PEEP*, um otimizador local atualmente integrado no compilador *LISP* na arquitetura *Motorola 68000.* O trabalho de Giegerich é direcionado para as arquiteturas *MC86000* e *8086*, mas a maioria das otimizações são também relevantes para máquinas de grande porte. Peter Kessler descreve uma técnica para analisar descrições de máquina visando o uso de instruções de propósito especial. Sua ferramenta analisa duas arquiteturas, *VAX-11* e *MC68000.* A linguagem para escrever geradores de código proposta por Fraser [FRASER 89] se aplica nas arquiteturas *VAX* e *MC68020.* O compilador proposto por

Fraser e Wendt no qual o gerador de código e o otimizador estão coesamente integrados gera código para a arquitetura *VAX*. A máquina alvo utilizada por Warfield e Bauer também é o *VAX*. O método proposto por Davidson [DAVIDSON 81] *PO* é aplicável nas arquiteturas: *CDC Cyber 175, DEC-10, PDP-11* e no microprocessador *8080* da Intel. Estas máquinas representam arquiteturas bem diferentes. O *CDC Cyber 175* e o *DEC-10* são dois tipos distintos de computadores de grande porte. O *PDP-11* é um minicomputador de 16 *bits*, enquanto o *8080* é um microprocessador de 8 *bits*. Além destas arquiteturas, é possível usar *PO* em compiladores para arquiteturas não convencionais, por exemplo, processadores vetoriais com *pipelines*.

Nas abordagens apresentadas, nenhuma pesquisa foi realizada para explorar o paralelismo de *CPU*, a não ser [DAVIDSON 81]. Gerar código para arquiteturas não convencionais, como por exemplo processadores vetoriais e máquinas possuindo paralelismo, é difícil. Estudos realizados por Davidson sugerem que a inclusão em *PO* de informações adicionais sobre o processador através da descrição da máquina pode ser possível gerar código para o processador de maneira convencional. *PO* aplicaria transformações no código, produzindo um código que se aproveita das arquiteturas não convencionais. Por exemplo, nas máquinas com múltiplos processadores aritméticos, código poderia ser gerado usando somente um processador para cada instrução. *PO* pode substituir pares de instruções que utilizam processadores diferentes por uma única que usa ambos.

5 Conclusão

Este texto apresentou várias abordagens para geração e otimização de código que se utilizam de descrição de arquiteturas de máquinas alvo para atingirem seus objetivos. Foram mostrados também, através de exemplos, algumas notações utilizadas nas especificações das máquinas, seguida de uma análise das mesmas. Desta análise, conclui-se que, muito embora seja difícil chegar-se a um consenso sobre qual linguagem é a mais adequada, porque quase todas, basicamente, diferem somente na notação e terminologia adotada, tornando a seleção da "melhor" apenas uma questão de gosto. Certas características presentes ou ausentes em algumas linguagens apontadas na seção 4 auxiliam na seleção.

Resumidamente, uma linguagem para descrição de arquiteturas tendo em vista a geração automática de otimizadores de código deve, em princípio, possuir as seguintes propriedades:

1. possuir mecanismos para definição dos modos de endereçamento;
2. permitir definição completa da semântica das instruções, incluindo atualização de código de condição;
3. dispor de mecanismos de associação de custo;

4. ser concisa e com alto poder de expressão.

Entre as abordagens apresentadas, três delas, [DAVIDSON 81], [KESSLER 84] e [COSTA 90], destacam-se das demais, por sua simplicidade, pelo seu poder de expressão e pela forma concisa de descrever as arquiteturas de máquinas.

Os sistemas *PO* e a primeira versão de *PEEP* atribuídos a Davidson e Robert Kessler, respectivamente, foram desenvolvidos em paralelo, e baseiam-se na descrição de máquina para atingirem seus objetivos, entretanto, informações sobre a linguagem utilizada por Robert Kessler são mínimas, o que dificulta um possível uso ou adaptação de sua abordagem para ser utilizado no sistema *GOL*.

Em conclusão, os trabalhos de Costa e Davidson satisfazem os requisitos citados acima, exceto por:

1. A descrição de Davidson não dispõe de mecanismos para definir custos.
2. A descrição do repertório de instruções de Costa é incompleta porque não possui mecanismos para indicar operação com os códigos de condição.

O conhecimento do custo é necessário para dirigir o processo de otimização, e o efeito completo das instruções deve estar claramente especificado para que uma substituição por outra sequência de instrução proceda. Este é um requisito fundamental se o objetivo almejado é otimizar o código gerado.

Contudo, as abordagens de Costa e Davidson podem ser adaptadas, suprindo estas deficiências, para serem utilizadas em *GOL*. Não se justifica, portanto o projeto de uma nova linguagem.

Bibliografia

[AHO 73] AHO , A. V.,and ULLMAN, J. D., The Theory of parsing, translation and compiling, Vols. 1 e 2, Prentice-Hall, Englewood Cliffs, N.J. 1973.

[AHO 86] AHO, A. V., SETHI, R., and ULLMAN, J. D., COMPILER PRINCIPALS, TECHNIQUES AND TOOLS, Addison -Wesley Publishing Company, 1986.

[AHO 85] AHO A. V. and GANAPATHI, M. , *Efficient Tree Pattern Matching: An Aid to Code Generation*, Conf. Rec. 12^{th} ACM Symp. on Princ. of Programming Languages, Jan. 1985, 334-340.

[AIGRAIN 84] AIGRAIN, P. et alii,*Experience with a Graham-Glanville Code Generator*, Proceedings of the SIGPLAN'84 Symposium on Compiler Construction, June 1984, 13-24.

[BELL 71] BELL, C. G., & NEWELL, A., Computer Structures: Readings and Examples. McGraw-Hill, New York, 1971.

[CATTELL 78] CATTELL, R. G. G., *Formalization and Automatic Derivation of Code Generators*, PhD dissertation, Carnegie-Mellon University, Pittsburgh, Pa., April 1978.

[CATTELL 79] CATTELL, R. G. G., Newcomer, J. M., Leverett, B. W., *Code Generation in a Machine-Independent Compiler*, SIGPLAN Conference Compiler Construction, ACM, 1979.

[CATTELL 80] CATTELL, R. G. G., *Automatic Derivation of Code Generators from Machine Descriptions*, ACM Transactions on Programming Languages and Systems, Vol. 2, No. 2, April 1980.

[CHASE 87] CHASE, D. R., *An Improvement to Bottom-up Tree Patern Matching*, Conf. Rec. 14^{th} ACM Symp. on Princ. of Programming Languages, Jan. 1987, 168–177.

[COSTA 90] COSTA, Paulo Sávio da S., "*Um Gerador Automático de Geradores de Código*", Dissertação de Mestrado, Departamento de Informática PUC-RJ, Fevereiro 1990.

[DAVIDSON 80] DAVIDSON, Jack W., & FRASER, Christopher W., "*The Design and Application of a Retargetable Peephole Optimizer*", ACM Transactions on Programming Languages and Systems, Vol. 2, No. 2, April 1980.

[DAVIDSON 81] DAVIDSON, Jack W.,"*Simplifying Code Generation Through Peephole Optimization*", Ph.D dissertation, Department of Computer Science, The University of Arizona, Tucson, Arizona, December 1981.

[DAVIDSON 84] DAVIDSON, Jack W., & FRASER, Christopher W., *Code Selection through Object Code Optimization*, ACM Transactions on Programming Languages and Systems, Vol. 6, No. 4, October 1984.

[DAVIDSON 84a] DAVIDSON, Jack W., & FRASER, Christopher W., "*Automatic Generation of Peephole Optimization*", Proceedings of the ACM SIGPLAN, Symposium on Compiler Construction , SIGPLAN NOTICES, Vol. 19, No. 6, June 1984.

[DAVIDSON 87] DAVIDSON J. W. &.FRASER C.. W., *Automatic inference and fast interpretation of peephole optimization rules*, SOFTWARE- PRACTICE AND EXPERIENCE, 17, (801-812) , 1987.

[DAVIDSON 89] DAVIDSON J. W. & WHALLEY D. B., *Quick Compilers Using Peephole Optimization*, SOFTWARE-PRACTICE AND EXPERIENCE, VOL. 19(1), (79-97), JANUARY 1989.

[FRASER 86] FRASER, Christopher W. & WENDT, Alan L., "*Integrating Code Generation and Optimization*", ACM Sigplan Notices, 1986.

[FRASER 88] FRASER, Christopher W. & WENDT, Alan L., *Automatic Generation of Fast Optimizing Code Generators*, Proceedings of the SIGPLAN'88 Symposium on Compiler Construction, SIGPLAN Notices 23, 7(July 1988), 79-84.

[FRASER 89] FRASER, Christopher W., *A Language for Writing Code Generators*, SIGPLAN'89 Conference on Programming Language Design and Implementation, Portland, Oregon June 21–23, 1989.

[FRASER 77] FRASER C. W., *Automatic gencration of code generators*, PhD dissertation Comp. Science Dept. Yale University, New Haven, Conn. 1977.

[GRAHAM 80] GRAHAM, Susan L., *Table-Driven Code Generation*, COMPUTER 13, August 1980.

[GRAHAM 82] GRAHAM, Susan L., et alli, *An Experiment in Table Driven Code Generation*, ACM Sigplan Notices, Proceedings of the Sigplan'82 Symposium on Compiler Construction, Boston, Massachusetts June, 1982.

[GRAHAM 88] GRAHAM, S. L. and PELEGRI–LLOPART, E., *Optimal Code Generation for Expression Trees: An Application of BURS Theory*, Co nf. Rec. 15^{th} ACM Symp. on Princ. of Programming Languages, Jan. 1988, 294–308.

[GLANVILLE 78] GLANVILLE R. S. & GRAHAM S. L., *A new method for compiler code generation*, In Conference Record of the Annual ACM Symposium on Principles of Programming Languages (Tucson, Ariz. Jan. 23-25). ACM, New York, pp. 231–240, 1978.

[GANAPATHI 82] GANAPATHI, M. and FISCHER, C. N., *Description-Driven Code Generation using Attribute Grammars*, In: Proc. 9th POPL Conference, p. 108-119, ACM, January 1982.

[GANAPATHI 89] GANAPATHI, M. & MENDAL, G., *Issues in ADA Compiler Technology*, COMPUTER IEEE, February 1989, (52-60).

[GANAPATHI 85] GANAPATHI, M. and FISCHER, C. N.,*Affix Grammar Driven Code Gencration*, ACM Transaction Programming Language and Systems, 7, 4 Oct. 1985, 560-599.

[GANAPATHI 88] GANAPATHI, M. and FISCHER, C. N., "*Integrating Code Generation and Peephole Optimization*", Acta Informatica 25, 85-109 (1988).

[GIEGERICH 83] GIEGERICH, R., "*A Formal Framework for the Derivation of Machine Specific Optimizers*", ACM Translations on Programming Languages and Systems, Vol. 5, No. 3, July 1983 (478–498).

[HANSON 81] HANSON, D.R., *"The Y programming language"*, ACM Sigplan Notices, Proceedings of the 16,2 February 1981, 59-68.

[HENRY 87] HENRY, Robert R., "*Code Generation by Table Lookup*" , Technical Report # 87-07-07, Computer Science Department, FR-35 University of Washington, Seattle, WA 89195 USA, July 1987.

[HENRY 89b] HENRY, Robert R. and Damron, Peter C., "*Algorithms for Table-Driven Code Generators Using Tree-Pattern Matching*", Technical Report # 89-02-03, Computer Science Department, FR-35 University of Washington, Seattle, WA 89195 USA, February 1989.

[HENRY 89a] HENRY, Robert R. and Damron, Peter C., "*Performance of Table-Driven Code Generators Using Tree-Pattern Matching*" , Technical Report # 89-02-02, Computer Science Department, FR-35 University of Washington, Seattle, WA 89195 USA, February 1989.

[HOFFMANN 82] HOFFMANN C., & O'DONNELL M. J., *Pattern Matching in Trees"*, J. ACM 29, January 1982, 68-95.

[JOHNSON 78] JOHNSON, S. C., *A portable compiler: Theory and Practice*, In Proc. 5th ACM Symp. Principles of Programming Languages (Tucson, Ariz., Jan. 23-25), ACM New York, Jan. 1978, pp. 97- 104.

[KESSLER 84] KESSLER, Peter B., *Peep - An Architectural Description Driven Peephole Optimizer*, Proceedings of the ACM SIGPLAN, Symposium on Compiler Construction , SIGPLAN NOTICES, Vol. 19, No. 6, June 1984.

[KESSLER 86] KESSLER, Peter B., *Discovering Machine-Specific Code Improvements*, ACM Sigplan Notices, 1986.

[KNUTH 68] KNUTH, D. E., "*Semantics of context-free languages*", Math Systems and Theory 2,2, June 1968, 127-145.

[LEVERETT 79] LEVERETT, Bruce W., et alli, *An Overview of the Production-Quality Compiler-Compiler Project*, Carnegie-Mellon University Computer Science Technical Report 79-105, 1979.

[LEVERETT 80] LEVERETT, Bruce W., et alli, *An Overview of the Production-Quality Compiler-Compiler Project*, COMPUTER 13, August 1980.

[MOTOROLA 82] MOTOROLA Edition, 16-Bit Microprocessor, USER'S MANUAL, third edition, 1982.

[PDP11 73] PDP11, Processor Handbook, Ed. DIGITAL Equipment Corporation, 1973.

[RICH 83] RICH, Elaine, Artificial Intelligence, McGraw-Hill Book Company, N. Y., 1983.

[RUSSELL 85] RUSSELL, Stuart, *"The Compleat Guide to MRS"*, Stanford Knowledge Systems Laboratory, Stanford University Report no. KSL-85-12, June 1985.

[VAX 82] VAX, Hardware Handbook, Architecture Handbook, Ed. DIGITAL Equipment Corporation, 1982.

[WARFIELD 88] WARFIELD, Jay W. & BAUER, Henry R., "*An Expert System for a Retargetable Peephole Optimizer*", ACM Sigplan Notices, Vol. 23, No. 10, 1988.