PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL

FACULDADE DE INFORMÁTICA

CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

COMPILADOR CONFIGURÁVEL PARA PROCESSADORES EMBARCADOS

Prof. Fernando Gehm Moraes

Orientador

Adriano Soli Rodrigues da Roza
Carlos Eduardo Dutra Brum
Leonardo Santos da Silva

Relatório de Trabalho de Conclusão II

Porto Alegre, 13 de dezembro de 2001

RESUMO

Este relatório tem por objetivo apresentar um método de descrição de processadores embarcados visando geração de código em linguagem de montagem (*assembly*) para programas descritos em linguagem C, padrão ANSI. Na execução do projeto foram utilizados o compilador configurável LCC (*retargetable compiler*) e o processador-alvo R7. Esse processador é membro de uma família de processadores que são utilizados na disciplina de Organização de Computadores do curso de bacharelado em Ciência da Computação, na Faculdade de Informática da PUCRS. A validação do código *assembly* gerado foi realizada com base em testes efetuados através de simulador desenvolvido no âmbito do GAPH – Grupo de Apoio ao Projeto de Hardware. O método de descrição para processadores disponibilizado neste trabalho se insere no fluxo de projeto conjunto de *hardware/software* (*codesign*) realizado pelo GAPH. A contribuição deste trabalho é disponibilizar não apenas um compilador C para uma dada arquitetura alvo, mas todo o processo de compreender como é descrito um processador para um compilador configurável e a respectiva documentação deste processo, visando adaptá-lo a outros processadores.

Palavras-chave: compilador configurável, LCC, *retargetable compiler*, *codesign*, arquitetura de computadores, processadores embarcados, geração de código *assembly*.

ABSTRACT

The main goal of this report is to present a method to describe the architecture of embedded processors, aiming the assembly code generation from a C code, ANSI format. We have used LCC as a retargetable compiler, and R7 processor architecture as case study. This processor is used in Computer Organization courses of the Computer Science course at PUCRS. The assembly description obtained from the C code for a given problem for the R7 processor, using the LCC compiler, was validated using a simulator developed by the GAPH research group. The developed method to describe processors is an important part of the codesign flow used by the research group. The contribution of this work is twofold. First, to make available a C compiler for a given processor. Second, present the entire process to describe the architecture of embedded processors and the related documentation, aiming to adapt this method to others processors.

Key words: retargetable compilers, LCC, *codesign*, computer architecture, embedded processors, *and assembly code generation*.

SUMÁRIO

LISTA DE FIGURAS

1. INTRODUÇÃO

Com a crescente demanda de mercado na área tecnológica, mais precisamente de produtos eletrônicos voltados a aplicações específicas, surge a necessidade de satisfazer essa procura no menor espaço de tempo possível [1].

Essa restrição de prazos traz consigo o aumento da dificuldade no planejamento e execução destes projetos, agregada à complexidade destes produtos eletrônicos. Devido a estes fatos surge o aumento das pesquisas com o objetivo de reduzir sensivelmente o tempo necessário ao projeto e a respectiva implementação desses produtos, ou seja, a redução do *time-to-market* (tempo necessário para a chegada do produto ao mercado consumidor).

Agregado à redução do *time-to-market* ressaltamos ainda a extrema e sempre presente necessidade de redução de custos e elevação do desempenho do produto. É importante também perceber que projetos eletrônicos de porte envolvem tarefas altamente complexas tendo em vista a natureza heterogênea dos componentes necessários a sua realização, como componentes eletrônicos, mecânicos, sensores, atuadores, etc.

As ferramentas de CAD (*Computer Aided Design*) disponíveis no mercado fornecem à equipes de projetistas experientes a capacidade de elaborar, em poucas semanas, projetos de circuitos que equipes há 10 anos atrás levariam meses desenvolvendo. Por outro lado, existem sistemas que exigem uma grande quantidade de ferramentas a serem planejadas e desenvolvidas para que seja melhor automatizado o processo de concepção visando atender demandas crescentes de mercado [1].

A implementação destes sistemas implica na utilização de um sistema de desenvolvimento com componentes integrados que permitam a captura, validação e síntese dos sistemas eletrônicos, através de um conjunto de modelos formais e/ou informais, e ainda possibilite suporte a diferentes técnicas de projeto para cada componente e a especificação da forma de interação destes componentes.

Cabe ressaltar aqui a ênfase de pesquisa direcionada a sistemas eletrônicos com uso específico na composição de automóveis, aeronaves, dispositivos de comunicação pessoal (celulares, *pagers* ...), definidos como sistemas embarcados (em inglês, *embedded systems*). Assim, sistemas embarcados ou sistemas embutidos são para todos os efeitos, sistemas computacionais que executam uma função específica.

Estes sistemas possuem a mesma estrutura geral de um computador, mas a especificidade de suas tarefas faz com que não sejam nem usados nem percebidos como um computador [5][15]. A tarefa de desenvolver conjuntamente o *hardware* e *software* destes sistemas é realizada por ferramentas de *codesign* ou *hardware*/*software design*. De Micheli [5] define projeto

integrado de *hardware* e *software* como: "a busca do alcance dos objetivos em nível de sistema do produto pela exploração da sinergia entre *hardware* e *software*, através do projeto concorrente dessas entidades".

Nos projetos baseados em *codesign* as funcionalidades não críticas dos sistemas são implementadas em *software*, deixando-se o fator desempenho para o *hardware*, devido à demanda de redução de custos e reduzido tempo de projeto.

Esse fato torna eminente a necessidade de direcionar esforços para analisar e conceber automaticamente a distribuição adequada da funcionalidade do sistema entre o *hardware* e o *software*.

Na descrição do *hardware* e do processador (parte *software* do sistema embarcado) utilizam-se linguagens de descrição de *hardware*, como VHDL (*VHSIC Hardware Description Language*) [16] ou Verilog [27], permitindo posterior mapeamento automático para FPGAs [24] (*Field Programmable Gate Arrays*). Este mapeamento automático é realizado por ferramentas de síntese lógica e física.

FPGAs são circuitos integrados personalizáveis após a fabricação. Sua função é definida por um arquivo de configuração, obtido através de ferramentas de síntese lógica e física. A atual capacidade dos FPGAs permite implementar circuitos digitais com mais de 1 milhão de portas lógicas equivalentes.

A implementação em FPGAs permite a prototipação do sistema e sua validação, possibilitando a inclusão de um sistema computacional completo em um único *chip*, mais conhecido como SOC [23][21] (*system-on-a-chip*). Vários autores afirmam que em menos de 7 anos já existirão no mercado circuitos integrados compostos por mais de 1 bilhão de transistores.

A Figura 1 [8] ilustra uma possível metodologia para projeto conjunto *hardware*/*software*, a qual explicitamos, a seguir, os conceitos envolvidos em cada uma das tarefas exigidas.

Dada uma especificação, muitas vezes informal, de um sistema computacional embarcado, as tarefas a serem cumpridas no nível de abstração sistêmico devem gerar como produto pelo menos duas descrições formais abstratas, uma do *software* e uma do *hardware*.

O nível de abstração destas descrições pode ainda ser bastante alto, e. g. uma descrição comportamental em Verilog ou VHDL para o *hardware* e um código em linguagem C para o *software*. Estas descrições devem ser passíveis de:

a) tratamento automatizado ou semi-automatizado por métodos e ferramentas de níveis de abstração inferiores, ou seja compilação para o *software* e síntese para o *hardware*;

b) verificação de seu comportamento contra a especificação, através de métodos de validação;

c) oferecer estimativas de desempenho para o produto final.

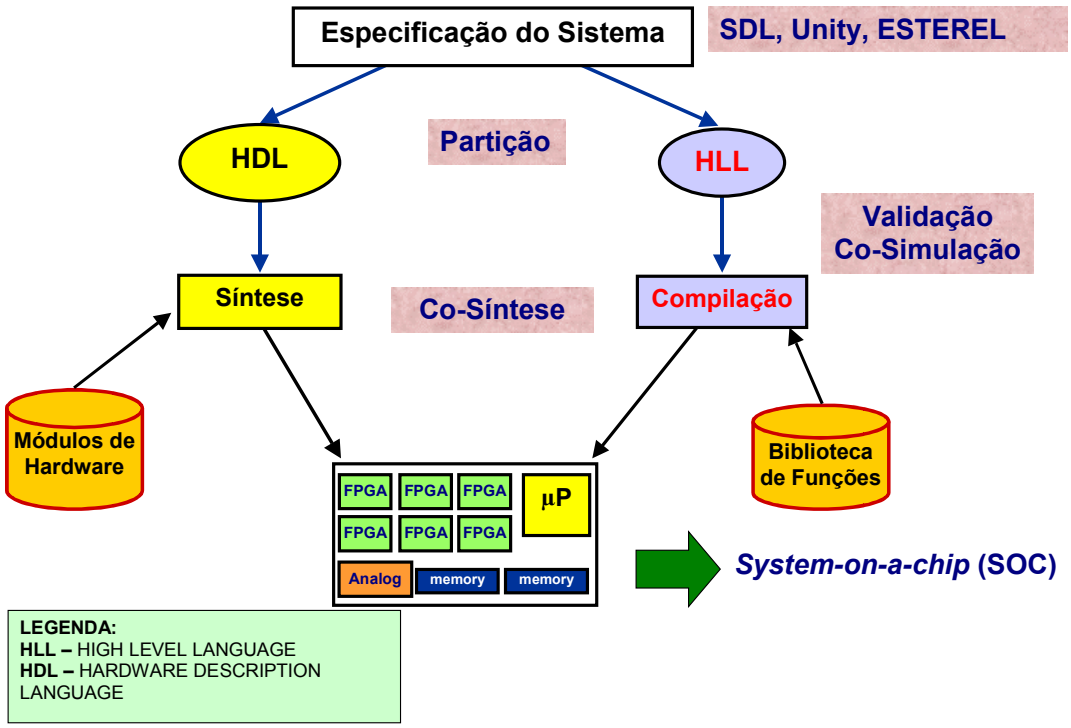

Figura 1 – Exemplo de fluxo de projeto para *Codesign*.

As tarefas de projeto de sistemas computacionais embarcados no nível sistêmico podem ser agrupadas em quatro classes principais: (1) modelagem; (2) particionamento *hardware*/*software*; (3) síntese; (4) validação.

A modelagem sistêmica inclui um formalismo de descrição utilizado pelos usuários para definir a funcionalidade do sistema computacional e um modelo formal subjacente para o qual a descrição inicial deve ser traduzida. O formalismo de descrição pode ser uma linguagem de programação (tal como C ou Occam [20][14][18]), uma linguagem de descrição de *hardware* (tal como Verilog ou VHDL), ou uma linguagem específica, derivada ou não das anteriores, como SDL [26], Unity [3] ou ESTEREL [4].

Posteriormente, o sistema é particionado, normalmente de forma semi-automática ou manual, em uma parte *hardware* (HDL) e outra parte *software* (HLL). A parte *hardware* pode ser descrita na linguagem VHDL, e sintetizada com o auxílio de módulos pré-projetados, denominados *cores*. A parte *software* é compilada gerando-se código objeto.

A validação do projeto de sistemas embarcados consiste em fornecer garantias de funcionamento integrado correto para as partes de *software* e *hardware* do sistema. Esta validação pode ser feita antes e/ou depois do particionamento. Quando realizada antes do particionamento, é uma validação geral da funcionalidade do sistema, apta a capturar erros de projeto mais grosseiros, cometidos no momento da especificação. Feita após o particionamento, pode capturar problemas sutis da interação entre *software* e *hardware*, produzidos, por exemplo, durante o particionamento manual.

A validação pode ser realizada através de simulação e/ou verificação formal. Normalmente, sistemas de *codesign* pregam a utilização concomitante de métodos de ambos

domínios. A simulação é usada neste contexto sob a forma de co-simulação de partes de *software* e *hardware* (por exemplo, através da operação integrada de simuladores de linguagens de descrição de *hardware* e de simuladores para os processadores do sistema). A verificação formal, por outro lado, é empregada para validar módulos críticos do sistema e para guiar o processo de síntese, posterior ao particionamento [1].

A síntese engloba as tarefas de construção de modelos de níveis de abstração inferiores ao sistêmico, não constituindo uma tarefa *per se* de *codesign*. Enquanto isto é efetivamente verdade para o *hardware* do sistema, duas classes de tarefas de síntese são consideradas como compondo o nível sistêmico: a síntese de *interfaces* entre *software* e *hardware* e a própria síntese do *software* do sistema.

Nessa última fase são definidas as *interfaces* entre *software* e *hardware* que possibilitem a comunicação entre esses dois lados.

O presente trabalho é parte do conjunto de ferramentas para *codesign*, pois tem por objetivo desenvolver, através de um Compilador Configurável (*Retargetable Compiler* [32][31]), o código em *assembly* (linguagem de montagem) para determinado processador-alvo, a partir de uma linguagem de programação alto nível, neste caso a linguagem C.

O projeto se insere no rol de pesquisas do GAPH (Grupo de Apoio ao Projeto de *Hardware*). É utilizado no decorrer do trabalho um montador [10] desenvolvido no GAPH, reduzindo assim a complexidade do desenvolvimento do trabalho com o compilador configurável.

1.1 Ambiente de Desenvolvimento

A Figura 2 ilustra o ambiente de desenvolvimento para a parte *software* de um sistema de *codesign*, ou seja, a parte direita da Figura 1 (descrição C e compilação). Este ambiente possui quatro ferramentas: (i) montador, (ii) simulador, (iii) compilador C e (iv) gerador do código VHDL. O montador e o simulador permitem gerar o código objeto de uma dada aplicação e simular funcionalmente o processador, em função da sua configuração [10]. O montador e o simulador são vistos como uma única ferramenta, pois estão integrados, funcionando sob uma única *interface* gráfica.

O gerador do código VHDL modifica a descrição e o próprio código *assembly*, de forma que o *hardware* do processador só venha a ter os recursos estritamente necessários para executar a aplicação a que se destina. Essa otimização é realizada visando ter um processador dedicado à aplicação, apresentando uma redução no custo em termos de área e um aumento em seu desempenho. Estas ferramentas foram desenvolvidas pela bolsista Aline Vieira de Mello, com orientação do Professor Fernando Gehm Moraes [11].

É neste contexto que se faz interessante e necessário que um compilador C possa ser configurável, adaptando-se às mudanças tanto de *hardware* como de *software* para melhor atender as necessidades do usuário.

Um compilador configurável caracteriza-se pela versatilidade com que gera código para um variado número de processadores-alvo, sendo que para isso é composto de vários módulos, onde todas as descrições (do processador e para geração de código) estão armazenadas e podem ser alteradas de acordo com a necessidade do usuário. Exemplo de compiladores configuráveis são o GCC [17][28] (*GNU C Compiler*) e o LCC [32][31].

Para incluir um novo processador-alvo no compilador é necessário inserir um novo módulo com a descrição correspondente, ou seja, com dados relativos ao processador-alvo. Feito isso, deve-se recompilar o núcleo do compilador, visando anexar a este núcleo as novas configurações.

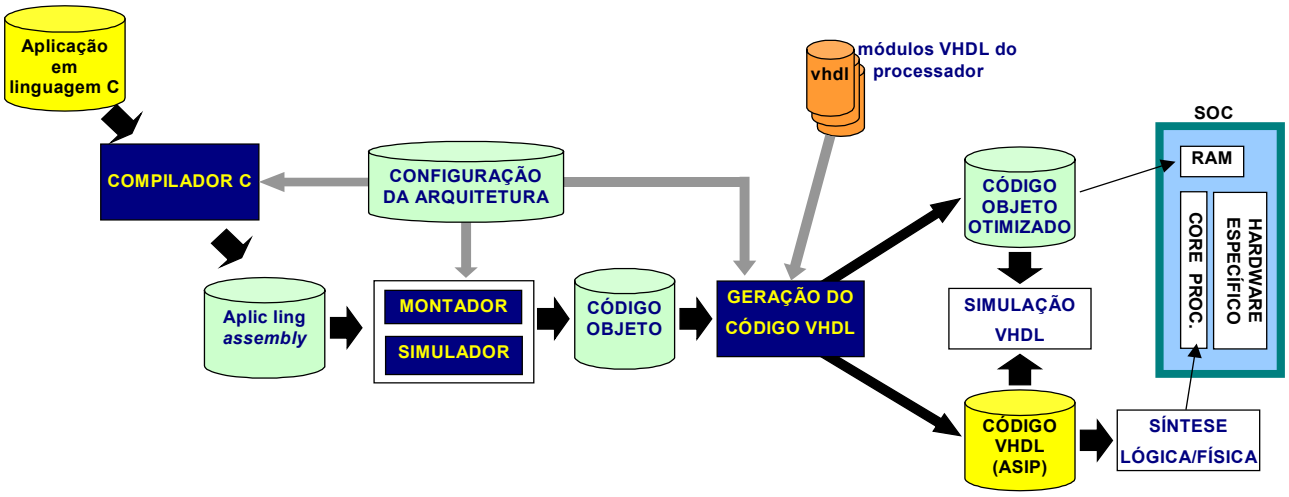

Figura 2 - Ambiente de desenvolvimento para a geração de processador embarcado.

Buscando suporte ao conceito de compilador configurável foram pesquisadas algumas das ferramentas disponíveis para a elaboração do trabalho proposto, que são os compiladores GCC, LCC e SDCC [6], os quais serão utilizados em plataforma Unix e/ou Linux.

1.2 Descrição do Projeto

O presente projeto visa desenvolver um compilador C (ver Figura 2), que complete as ferramentas para o desenvolvimento e uso de processadores embarcados. Optou-se por utilizar compiladores já existentes, que permitam configurar diferentes processadores-alvo, visando agilizar o processo.

Para que se possa tirar proveito das facilidades de configurabilidade oferecidas pelo compilador, é necessário que haja uma descrição da configuração interna do processador-alvo, tal como: número de registradores, representação interna de tipos de dados, atributos de registradores, modos de endereçamento, etc. Além disso, são necessárias informações referentes ao processo de geração de código pelo compilador. Estas informações são repassadas ao compilador através dos arquivos de descrição de máquina.

O compilador C escolhido deve ter a capacidade de gerar código *assembly* para o

processador especificado, através do arquivo de descrição de máquina, destacando que o código fonte em linguagem C, dado como entrada para o compilador, obedeça ao padrão ANSI C (*American National Standards Institute*).

Salientamos que a máquina descrita como alvo deve ser capaz de suportar código ANSI C em suas características básicas (operações aritméticas, operações booleanas, laços de controle, etc). A fim de auxiliar nesse ponto, o processador-alvo possui um conjunto de instruções pré-definidas, que delimitam os pré-requisitos mínimos para que as ações configuradas suportem as operações mencionadas acima.

Ressaltamos que o atual processador-alvo pode ser alterado, dado o fato que seu desenvolvimento é feito no próprio grupo de pesquisa, não sendo um processador fixo, como por exemplo o processador MIPS [2]. Esta flexibilidade é uma das características que simplificaram o trabalho realizado, pois caso um dado comando em C exija uma implementação muito custosa no processador, alterações podem ser inseridas no mesmo para que o comando seja executado com mais eficiência.

Tendo em mãos o arquivo de configuração, o núcleo do compilador é instalado, a fim de elaborar um novo arquivo executável que seja capaz de gerar código *assembly* para o novo processador-alvo. Para o auxílio na recompilação são utilizados *makefiles*[1], os quais também são parte do ambiente de configuração do compilador.

Observando a Figura 3, pode-se facilmente identificar que o trabalho faz parte do ambiente de desenvolvimento para a geração de processadores embarcados ilustrado pela Figura 2.

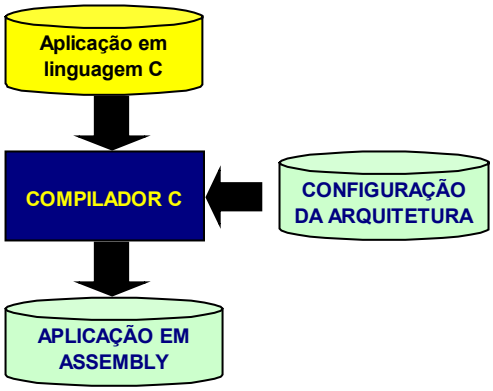

Figura 3 - Arquitetura do sistema desenvolvido.

As entradas do compilador configurável, conforme a Figura 3, são o código fonte da aplicação que se deseja compilar (em linguagem C – padrão ANSI) e o arquivo com detalhes referentes à descrição do processador (para o qual se deseja gerar código *assembly*). O produto gerado pelo compilador é o código *assembly* para o processador-alvo. Na Figura 4 exemplificamos a tradução para código *assembly*, no caso para o processador MIPS, aplicada a comandos em linguagem C.

[1] Um *makefile* é basicamente um *script* que auxilia ao utilitário *make* do UNIX a escolher os arquivos de programas que devem ser compilados e ligados juntos [13].

Para análise do exemplo mostrado na Figura 4 devemos assumir que: *(i) $t1* é registrador utilizado como temporário; *(ii) i, j* e *k* são respectivamente os registradores *$t3, $s4* e *$t5*; *(iii)* a base do vetor "*save[]*" é o registrador *$t6*.

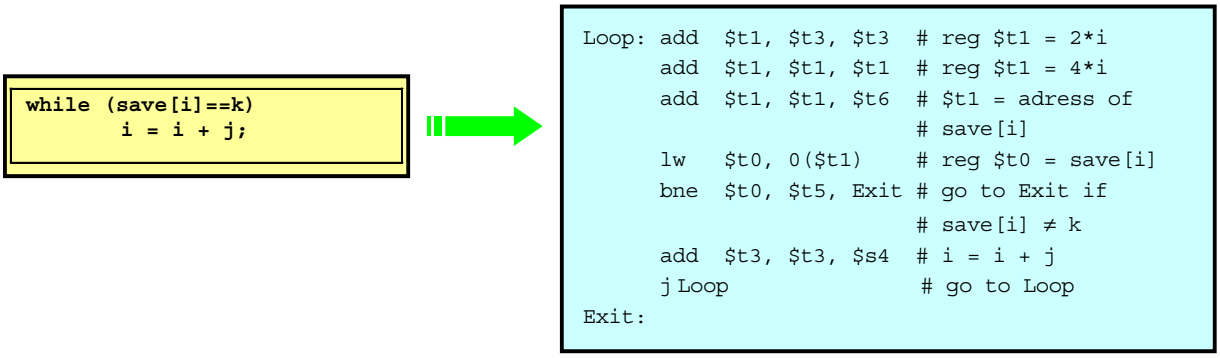

Figura 4 - Tradução de código fonte C para linguagem *assembly* ([2], página 67).

1.3 Estrutura do Texto

Este documento está organizado da seguinte forma:

a) Capítulo 2 - apresenta a análise e escolha de compiladores parametrizáveis para o projeto;

b) Capítulo 3 - apresenta a estrutura e funcionamento interno do compilador escolhido;

c) Capítulo 4 - apresenta a definição do processador-alvo para o projeto;

d) Capítulo 5 - apresenta a descrição da configuração do processador-alvo;

e) Capítulo 6 - apresenta a avaliação do código gerado para o processador-alvo;

f) Capítulo 7 - apresenta as conclusões finais do trabalho realizado.

2. COMPILADORES CONFIGURÁVEIS

Este Capítulo tem por objetivo relatar a análise realizada na escolha do compilador configurável, em função das restrições definidas na proposta do trabalho. No estudo aqui realizado foram ponderadas as características de três compiladores: LCC, GCC e SDCC. A seguir descrevemos as características principais de cada um desses compiladores.

2.1 Compilador LCC

LCC [32] é um compilador configurável para ANSI C, criado por Cristopher W. Fraser – *AT&T Bell Laboratories* - e David R. Hanson – *Princeton University*, cujas versões para processadores VAX, Motorola, SPARC e MIPS foram usadas na Universidade de *Princeton* e no *AT&T Bell Laboratories*. Este compilador integra técnicas de compilação estruturadas e é menor e mais rápido que a maioria dos compiladores C, gerando código de qualidade comparável [32]. Esses resultados são decorrentes da escolha do enfoque desejado na construção do compilador, onde a prioridade consiste em tornar fácil a inclusão e descrição de novos processadores-alvo.

O LCC foi construído com fins acadêmicos focados nos estudos sobre implementação de compiladores e geração de código, inicialmente gerando programas para um subconjunto de comandos da linguagem C.

Por ser um compilador configurável, o LCC pode gerar código *assembly* para diversos processadores, tais como MIPS, ALPHA, SPARC, X86, etc. Essa escolha se dá via linha de comando da seguinte forma: "*lcc –Wftarget=processador-alvo arquivo.c*".

A inserção de um novo processador-alvo, é elaborada através da edição de um arquivo de descrição da arquitetura (do tipo *md – machine description*). Neste arquivo estão armazenadas as características principais e as ações que devem ser realizadas quando do reconhecimento e implementação das instruções do processador. Este arquivo é utilizado na nova instalação do núcleo do compilador. Esta nova instalação é necessária para que seja possível a geração de código para o novo processador configurado com base em arquivo executável construído a partir da descrição implementada no arquivo *md*.

Cabe ressaltar que, para a instalação do compilador LCC, é necessário a disponibilidade do compilador GCC. Este fornece suporte na geração dos arquivos executáveis. Isso acontece devido ao fato de que o LCC possui código fonte aberto e sua distribuição provê apenas os arquivos-fonte, os quais devem ser compilados para geração do *software*.

No LCC são ainda implementadas funções para depuração independentes de processador-alvo, como *expression-level profiling* [32] (análise estatística de tempo de execução em nível de expressões no programa).

Cita-se abaixo mais algumas características do LCC:

a) A parte *front end* realiza análise léxica, sintática e semântica e eliminação de sub-expressões comuns tornando simples as transformações de código independentes de máquina (parte dessas transformações são realizadas pela parte *back end* e são basicamente operações sobre nodos das árvores formadas para geração de código intermediário) que colaboram na qualidade do código final gerado;

b) As árvores formadas na representação de código intermediário (*abstract syntax tree* [31]) possuem, em seus nodos, operadores denominados de *DAG operators* [31] que representam uma operação básica (atribuição, adição, subtração, conversão de tipos, etc ...);

c) A parte *front end* (independente de processador-alvo) e a *back end* (dependente de processador-alvo) estão implementadas em um mesmo programa, ligadas por uma *interface*. Esta *interface* consiste em algumas estruturas de dados compartilhadas, 18 funções, e uma estrutura em árvore denominada de DAG[2] (*directed acyclic graphs* [31]) com 36 operadores.

d) A parte *back end* e a parte *front end* são clientes uma da outra, trabalhando em conjunto para gerar código.

e) A parte *back end* necessita da *front end* para alocar memória, analisar tipos, símbolos e *strings* [30].

f) Parâmetros de configuração dependentes de processador-alvo especificam larguras e alinhamentos de tipos de dados básicos. Estes parâmetros são definidos em um arquivo de cabeçalho (*.h), o qual é incluído quando da compilação do código do LCC para um novo processador-alvo.

g) A configuração do LCC para inserção de novo processador-alvo requer a edição de dois componentes do *back end* que são: (i) funções de *interface* dependentes de processador-alvo; e (ii) regras de geração de código (formalizadas através de uma gramática específica para cada processador). Na prática, novos *back ends* são implementados através da edição de uma cópia de arquivo de configuração, do tipo *md*, já existente e também das funções de *interface*, realizando as adequações necessárias;

h) *Software freeware* distribuído sob a licença GPL – *General Public License* [31].

[2] A estrutura *dag* se encarrega de representar os detalhes referentes à geração de código executável a partir de um programa fonte, salientando que esta corresponde a uma forma de "linguagem intermediária" usada em compiladores para representação do código a ser gerado. Todavia é mais compacta que uma linguagem intermediária comum.

2.2 Compilador GCC

Outro compilador estudado foi o GCC (*GNU C Compiler*), criado por Richard Stallman, caracterizado por ser um *software* de domínio público, que suporta o padrão ANSI C, de modo que é muito fácil portar qualquer programa elaborado em linguagem C para sistema operacional Unix ou Linux .

O GCC é um compilador-otimizador portável e configurável, capaz de compilar linguagens tais como: C, C++, Fortran, Pascal e Objective-C[3].

Com o compilador GCC pode-se compilar código-fonte para processadores-alvo tais como MIPS, ALPHA, PENTIUM, M68000, AMD e muitos outros exemplos. Adicionando a esse fato pode-se ainda parametrizar o tipo de saída desejada para o processo de compilação, tendo opções de geração apenas do executável, apenas do arquivo objeto ou ainda do arquivo fonte pré-processado.

Cita-se abaixo mais algumas características do GCC:

a) fornece vários níveis de verificação de erro de código fonte.

b) produz código objeto com diferentes níveis de otimização.

c) portável para diferentes processadores (*retargetable compiler*).

d) processador-alvo especificado via de linha de comando. Exemplo: comando "*gcc –b processador-alvo –o arquivo.c*".

e) executa pré-processamento, compilação, gera o *assembly* e efetua ligação dos arquivos, sendo que estas etapas podem ser executadas separadamente ou com determinadas configurações escolhidas através de parâmetros via linha de comando. Exemplos: parametrização do "dialeto" ou variações da linguagem C desejada; parametrização para solicitação ou supressão de *warnings*[4]; parametrização de otimizações; remoção ou inclusão automática de protótipos de função; parametrização do tipo de pré-processamento desejado.

f) permite que a compilação seja executada até determinado estágio intermediário. Ex.: O parâmetro –S no comando "*gcc -S arquivo.c –o arquivo*" faz com que o processo pare na geração do código *assembly*, emitindo um arquivo com a extensão *.s.

g) *Software freeware* distribuído sob a licença GPL – *General Public License*.

[3] Objective-C é uma linguagem derivada de C com extensões para conceitos de orientação a objetos.
[4] Mensagens de aviso para prevenção de possíveis erros de código ou linhas desnecessárias.

2.3 Compilador SDCC

Dentro ainda da pesquisa realizada visando a utilização de uma ferramenta adequada ao projeto foi encontrado o compilador denominado *SDCC - Small Device C Compiler* [25], construído inicialmente por Sandeep Dutta – *University of Kentucky* [7], na forma de um compilador configurável para microprocessadores Intel 8051 e Zilog Z80.

Este compilador, com geração de código no padrão ANSI C, pode ser configurável para outros microprocessadores de 8 *bits*, possui código aberto e é distribuído sob a licença *GPL – General Public License* [12].

O *front end (parser)* existente pode ser incrementado para manipular extensões de linguagem para outros tipos de microprocessadores. O SDCC possui ainda eliminação interna de sub-expressões, implementação de *copy-propagation* [5], otimização de *loop* e eliminação de código morto.

Já o *back end* utiliza um esquema de alocação global de registradores, que pode ser configurado para outros microprocessadores de 8 *bits*. Este sistema de alocação de registradores faz parte da otimização realizada por este compilador.

O otimizador trabalha com regra baseada em um mecanismo de substituição de registradores. Esta otimização ocorre durante o processo de compilação, depois da geração de linguagem intermediária. Isto faz com que a otimização seja independente do tipo de microprocessador para o qual se deseja gerar código, sendo necessário saber apenas o número de registradores disponíveis e o número de *bits* armazenados por registrador.

Tipos de variáveis contemplados pelo SDCC:

a) *short* (8 *bits* - 1 *byte*);

b) *char* (8 *bits* - 1 *byte*);

c) *int* (16 *bits* - 2 *bytes*);

d) *long* (32 *bits* - 4 *bytes*);

e) *float* (4 *bytes* IEEE – 754 em precisão simples).

As ferramentas ASXXXX (montador), ASLINK (*linker*) e SDCDB (*debugger*) são parte integrante do compilador SDCC. As plataformas suportadas pelo SDCC são Linux/Unix e Windows9x.

Cabe ressaltar que para a instalação do compilador SDCC nas plataformas acima definidas é necessário a disponibilidade das aplicações GCC, para a plataforma Linux e/ou Unix, e Borland C para plataformas Windows9x. Isso acontece devido ao fato de que o SDCC possui

[5] Substituir traços de código quando uma operação é realizada mais de uma vez e tem seu destino final obtido por atribuições intermediárias desnecessárias. Exemplo: {a := d + e; b := d + e; c := a; c := b; } → *copy-propagation* → {c := d + e;}

código fonte aberto e sua distribuição provê apenas os arquivos-fonte, os quais devem ser compilados para geração do SDCC.

Nos itens a seguir mostramos exemplos de programas compilados pelo GCC e LCC, compiladores mais adequados ao escopo do trabalho realizado. O SDCC possui pouca documentação, sendo voltado para processadores de 8 *bits*.

2.4 Exemplos de Programas

Listamos a seguir alguns exemplos de programas utilizados para análise dos compiladores GCC e LCC, estes definidos primeiramente como mais adequados ao trabalho. Para tanto foram elaborados arquivos-fonte contemplando um conjunto básico de operações, tais como declarações de variáveis, adições, subtrações, multiplicações, divisões e alocação de memória.

Abaixo mostramos, dentre as operações realizadas, quatro exemplos desenvolvidos em linguagem padrão ANSI C, submetidos aos compiladores LCC e GCC, tendo como processador-alvo o processador SPARC.

2.4.1 Exemplo 1

No exemplo da Figura 5, o objetivo é verificar quais as diferenças de implementação entre os compiladores quando são tratadas declarações de variáveis do tipo *int* e *char* e ainda comandos de atribuição entre esses tipos.

Programa – exdecatrib.c	***Assembly do LCC –SPARC/SUN***	***Assembly do GCC – SPARC/SUN***

`#include "stdio.h"`
`#include "string.h"`
 	
`void main ()`
`{`
`  /* exemplo de declarações e`
`  atribuições de inteiros e`
`  caracter */`
 	
`  int a;`
`  int b;`
`  int c;`
 	
`  char d;`
`  char e;`
`  char f;`
 	
`  a = 1;`
`  b = 2;`
`  c = 3;`
 	
`  d = 'a';`
`  e = 'b';`
`  f = 'c'; }`
	`.global _main`
	`.seg "text"`
	`.align 4`
	`_main:`
	`set 1,%o5`
	`set 2,%o4`
	`set 3,%o3`
	`set 97,%o2`
	`set 98,%o1`
	`set 99,%o0`
	`L2:`
	`Retl; nop`
		`.file  "exdecatrib.c"`
		`gcc2_compiled.:`
		`.section  ".text"`
		`  .align 4`
		`  .global main`
		`  .type  main,#function`
		`  .proc  04`
		`main:  !#PROLOGUE# 0`
		`  save  %sp, -128, %sp`
		`  !#PROLOGUE# 1`
		`  mov  1, %o0`
		`  st  %o0, [%fp-20]`
		`  mov  2, %o0`
		`  st  %o0, [%fp-24]`
		`  mov  3, %o0`
		`  st  %o0, [%fp-28]`
		`  mov  97, %o0`
		`  stb  %o0, [%fp-29]`
		`  mov  98, %o0`
		`  stb  %o0, [%fp-30]`
		`  mov  99, %o0`
		`  stb  %o0, [%fp-31]`
		`.LL2:`
		`  ret`
		`  restore`

Figura 5 - Código fonte e *assembly* gerado nos compiladores GCC e LCC para o exemplo 1.

A funcionalidade dos programas é a mesma, porém a implementação em linguagem *assembly* é diferente em ambos compiladores. Observa-se que o LCC aloca um conjunto de registradores, enquanto o código gerado pelo GCC utiliza apenas um registrador, armazenando os

valores na pilha.

2.4.2 Exemplo 2

Neste exemplo, mostrado na Figura 6, o código *assembly* foi gerado com o objetivo de verificar quais as diferenças de implementação entre os compiladores quando é tratada uma adição de variáveis do tipo *int* em conjunto com atribuição com esses tipos.

Programa – exadd.c	***Assembly do LCC –SPARC/SUN***	***Assembly do GCC – SPARC/SUN***

`#include "stdio.h"`
`#include "string.h"`
 	
`void main ()`
`{`
`  /* exemplo de adições com`
`  inteiros envolvendo va-`
`  riáveis e constantes */`
 	
`  int a;`
`  int b;`
`  int c;`
`  int result_num, result_var,`
`  result_all;`
 	
`  a = 1;`
`  b = 2;`
`  c = 3;`
 	
`  result_num  = 2 + 3;`
`  result_var  = a + b + c;`
 	
`  result_all = result_num +`
`  result_var;`
`}`
	`.global _main`
	`.seg "text"`
	`.align 4`
	`_main:`
	`save %sp,-96,%sp`
	`set 1,%i5`
	`set 2,%i4`
	`set 3,%i3`
	`set 5,%i2`
	`add %i5,%i4,%l7`
	`add %l7,%i3,%i1`
	`add %i2,%i1,%i0`
	`L2:`
	`Ret; restore`
		`.file  "exadd.c"`
		`gcc2_compiled.:`
		`.section  ".text"`
		`  .align 4`
		`  .global main`
		`  .type  main,#function`
		`  .proc  04`
		`main:`
		`  !#PROLOGUE# 0`
		`  save  %sp, -136, %sp`
		`  !#PROLOGUE# 1`
		`  mov  1, %o0`
		`  st  %o0, [%fp-20]`
		`  mov  2, %o0`
		`  st  %o0, [%fp-24]`
		`  mov  3, %o0`
		`  st  %o0, [%fp-28]`
		`  mov  5, %o0`
		`  st  %o0, [%fp-32]`
		`  ld  [%fp-20], %o0`
		`  ld  [%fp-24], %o1`
		`  add  %o0, %o1, %o0`
		`  ld  [%fp-28], %o1`
		`  add  %o0, %o1, %o0`
		`  st  %o0, [%fp-36]`
		`  ld  [%fp-32], %o0`
		`  ld  [%fp-36], %o1`
		`  add  %o0, %o1, %o0`
		`  st  %o0, [%fp-40]`
		`.LL2:`
		`  ret`
		`  restore`

Figura 6 - Código fonte e *assembly* gerado nos compiladores GCC e LCC para o exemplo 2.

Neste exemplo, tal como no exemplo 1, o GCC efetua o tratamento das variáveis através da pilha. Observar que durante as somas há o ajuste do ponteiro da pilha para que o valor a ser somado seja corretamente buscado da memória.

Este modelo baseado em pilha ocasiona muitos acessos à memória. O desejado é justamente minimizar estes acessos, e por esta razão o código gerado pelo LCC é mais interessante para o trabalho.

É possível que haja a opção no GCC para utilizar os registradores, mas esta não é a opção *default* no momento da compilação.

2.4.3 Exemplo 3

No exemplo da Figura 7, observamos o tratamento e o código gerado para implementação de laços de controle, incluindo incremento de variáveis, comando de atribuição e comparação de valores, sendo no caso utilizado o comando *for* como exemplo.

Programa – exfor.c	***Assembly do LCC –SPARC/SUN***	***Assembly do GCC – SPARC/SUN***

`#include "stdio.h"`
`#include "string.h"`
 	
`void main ()`
`{`
`  /* exemplo de comando FOR */`
`  int a;`
`  int b;`
`  int c;`
 	
`  char d;`
`  char e;`
`  char f;`
 	
`  b = 1;`
`  c = 2;`
 	
`  for(a=0; a<5; a++)`
`  {`
`  b++;`
`  c++;`
`  d = 'a';`
`  e = 'b';`
`  f = 'c';`
`  }`
`}`
	`.global _main`
	`.seg "text"`
	`.align 4`
	`_main:`
	`save %sp,-96,%sp`
	`set 1,%i4`
	`set 2,%i3`
	`mov %g0,%i5`
	`L3:`
	`add %i4,1,%i4`
	`add %i3,1,%i3`
	`set 97,%i2`
	`set 98,%i1`
	`set 99,%i0`
	`L4:`
	`add %i5,1,%i5`
	`cmp %i5,5;`
	`bl L3; nop`
	`L2:`
	`ret; restore`
		`.file  "exfor.c"`
		`gcc2_compiled.:`
		`.section  ".text"`
		`  .align 4`
		`  .global main`
		`  .type  main,#function`
		`  .proc  04`
		`main:`
		`  !#PROLOGUE# 0`
		`  save  %sp, -128, %sp`
		`  !#PROLOGUE# 1`
		`  mov  1, %o0`
		`  st  %o0, [%fp-24]`
		`  mov  2, %o0`
		`  st  %o0, [%fp-28]`
		`  st  %g0, [%fp-20]`
		`.LL3:`
		`  ld  [%fp-20], %o0`
		`  cmp  %o0, 4`
		`  ble  .LL6`
		`  nop`
		`  b  .LL4`
		`  nop`
		`.LL6:`
		`  ld  [%fp-24], %o0`
		`  add  %o0, 1, %o1`
		`  st  %o1, [%fp-24]`
		`  ld  [%fp-28], %o0`
		`  add  %o0, 1, %o1`
		`  st  %o1, [%fp-28]`
		`  mov  97, %o0`
		`  stb  %o0, [%fp-29]`
		`  mov  98, %o0`
		`  stb  %o0, [%fp-30]`
		`  mov  99, %o0`
		`  stb  %o0, [%fp-31]`
		`.LL5:`
		`  ld  [%fp-20], %o0`
		`  add  %o0, 1, %o1`
		`  st  %o1, [%fp-20]`
		`  b  .LL3`
		`  nop`
		`.LL4:`
		`.LL2:`
		`  ret`
		`  restore`

Figura 7 - Código fonte e *assembly* gerado nos compiladores GCC e LCC para o exemplo 3.

Da mesma forma vista no exemplo 2, o GCC trata o incremento com acessos à memória e utilização da pilha. As variáveis são carregadas da memória para a pilha e após a operação é executada com atualização do ponteiro da pilha.

No LCC a operação é realizada totalmente com registradores, minimizando acessos à memória. Essa implementação, como já afirmado no exemplo 2, é mais interessante ao projeto.

2.4.4 Exemplo 4

Neste último exemplo, mostrado da Figura 8, a motivação é o tratamento de alocação e liberação de segmentos de memória utilizados pelos compiladores, no caso com variáveis do tipo *char*, através dos comandos *malloc* e *free*, todos padrão ANSI C.

Programa – exmalloc.c

```
#include "stdio.h"
#include "stdlib.h"
#include "string.h"

void main()
{
  char *ptr;

  if ((ptr = (char *) malloc(10))
     == NULL)
  {
    printf("Erro");
    exit(1);
  }
strcpy(ptr, "Oi");
printf("%s", ptr);

free(ptr);

}
```

Assembly do LCC –SPARC/SUN

```
.global _main
.seg "text"
.align 4
_main:
save %sp,-96,%sp
set 10,%o0
call _malloc; nop
mov %o0,%i5
mov %o0,%i4
cmp %i4,%g0; bne L5; nop
set L7,%o0
call _printf; nop
set 1,%o0
call _exit; nop
L5:
mov %i5,%o0
set L8,%o1
call _strcpy; nop
set L9,%o0
mov %i5,%o1
call _printf; nop
mov %i5,%o0
call _free; nop
L4:
ret; restore
.seg "text"
.align 1
L9:
.byte 37
.byte 115        "%s"
.byte 0
.align 1
L8:
.byte 79
.byte 105        "Oi"
.byte 0
.align 1
L7:
.byte 69
.byte 114
.byte 114        "Erro"
.byte 111
.byte 0
```

Assembly do GCC – SPARC/SUN

```
        .file   "exmalloc.c"
gcc2_compiled.:
.section        ".rodata"
        .align 8
.LLC0:
        .asciz  "Erro"
        .align 8
.LLC1:
        .asciz  "Oi"
        .align 8
.LLC2:
        .asciz  "%s"
.section        ".text"
        .align 4
        .global main
        .type    main,#function
        .proc   04
main:
        !#PROLOGUE# 0
        save    %sp, -120, %sp
        !#PROLOGUE# 1
        mov     10, %o0
        call    malloc, 0
         nop
        mov     %o0, %o1
        mov     %o1, %o0
        st      %o0, [%fp-20]
        cmp     %o0, 0
        bne     .LL3
        nop
        sethi   %hi(.LLC0), %o1
        or      %o1, %lo(.LLC0),
%o0
        call    printf, 0
         nop
        mov     1, %o0
        call    exit, 0
         nop
.LL3:
        ld      [%fp-20], %o0
        sethi   %hi(.LLC1), %o2
        or      %o2, %lo(.LLC1),
%o1
        call    strcpy, 0
         nop
        sethi   %hi(.LLC2), %o1
        or      %o1, %lo(.LLC2),
%o0
        ld      [%fp-20], %o1
        call    printf, 0
         nop
        ld      [%fp-20], %o0
        call    free, 0
         nop
.LL2:
        ret
        restore
.LLfe1:
        .size    main,.LLfe1-main
```

Figura 8 - Código fonte e *assembly* gerado nos compiladores GCC e LCC para o exemplo 4.

Verifica-se que para alocação e liberação de memória e implementação de comandos de I/O, no caso o comando *printf* (impressão na saída padrão), tanto GCC como LCC utilizam chamadas a rotinas de bibliotecas-padrão (Ex.: *stdio.h*). Estas, por sua vez, realizam chamadas a rotinas do sistema operacional.

A utilização do sistema operacional no gerenciamento de memória simplifica a implementação do compilador, tornando o código mais transparente, característica de fundamental importância no trabalho em pauta. Neste trabalho, a implementação de rotinas de sistema operacional e bibliotecas-padrão estão fora do escopo de desenvolvimento. Estas rotinas podem ser implementadas em trabalhos futuros.

Neste projeto supõe-se a existência de um sistema operacional mínimo, com funções como alocação de memória e entrada/saída de dados. Em processadores embarcados estes sistemas operacionais são geralmente denominados de RTOS (*Real Time Operating System*) [9].

2.5 Geração de *Assembly* em Diferentes Arquiteturas

Neste item é abordado um exemplo de parametrização para processadores diferentes utilizando o mesmo programa-fonte no GCC e no LCC, em duas situações. O SDCC não foi utilizado devido a sua portabilidade se restringir a microprocessadores de 8 *bits*. A seguir mostramos o arquivo de código fonte utilizado e o respectivo arquivo do *assembly* gerado nos processadores X86 e SPARC.

2.5.1 Programa Fonte Exemplo: “exemplo.c”

Para demonstração, definimos um exemplo simples, mostrado na Figura 9, visando melhor entendimento do *assembly* gerado. Foram incluídas declarações de variáveis do tipo inteiro, comandos de atribuição e uma operação de adição.

Os resultados a seguir mostrados são analisados sobre os aspectos: *(i)* utilização dos registradores; *(ii)* manipulação de variáveis; *(iii)* utilização da pilha.

```
void main()
{  int a;
   int b;
   int c;
   int d;

   d = 0;
   b = 50;
   a = 150;
   c = 200;
   d = a + b + c;
}
```

Figura 9 - Programa fonte para geração de *assembly*.

2.5.2 *Assembly* Gerado no LCC

Na primeira coluna da Figura 10, o processador SPARC implementa uma política diferenciada na manipulação de seus 32 registradores, denominada de "Janela de Registradores" [31]. Essa característica do processador otimiza o *assembly* gerado, mantendo as operações de variáveis através do conjunto de registradores. A memória, nesse caso, apenas é utilizada ao final das operações quando já se tem o resultado.

Arquitetura *SPARC*	Arquitetura *X86*

.global _main
.seg "text"
.align 4
_main:
mov %g0,%o2
set 50,%o4
set 150,%o5
set 200,%o3
add %o5,%o4,%o1
add %o1,%o3,%o2
L1:
retl; nop
	.486
	.model flat
	extrn __fltused:near
	extrn __ftol:near
	public _main
	_TEXT segment
	_main:
	push ebx
	push esi
	push edi
	push ebp
	mov ebp,esp
	sub esp,16
	mov dword ptr (-16)[ebp],0
	mov dword ptr (-8)[ebp],50
	mov dword ptr (-4)[ebp],150
	mov dword ptr (-12)[ebp],200
	mov edi,dword ptr (-4)[ebp]
	mov esi,dword ptr (-8)[ebp]
	lea edi,[esi][edi]
	mov esi,dword ptr (-12)[ebp]
	lea edi,[esi][edi]
	mov dword ptr (-16)[ebp],edi
	L1:
	mov esp,ebp
	pop ebp
	pop edi
	pop esi
	pop ebx
	
	ret

Figura 10 - Assembly gerado no LCC para arquitetura SPARC e X86.

Analisando o *assembly* gerado para o processador X86 nota-se a existência de poucos registradores (6 no total), implicando em várias utilizações das instruções *PUSH* e *POP* para resguardar o contexto. Há também uma significativa utilização da pilha para a execução das operações aritméticas.

2.5.3 *Assembly* Gerado no GCC

Arquitetura *SPARC*

```
        .file   "exemplo.c"
gcc2_compiled.:
.section        ".text"
        .align 4
        .global main
        .type    main,#function
        .proc   020
main:
!#PROLOGUE# 0
save    %sp, -128, %sp
!#PROLOGUE# 1
st      %g0, [%fp-32]
mov     50, %o0
st      %o0, [%fp-24]
mov     150, %o0
st      %o0, [%fp-20]
mov     200, %o0
st      %o0, [%fp-28]
ld      [%fp-20], %o0
ld      [%fp-24], %o1
add     %o0, %o1, %o0
ld      [%fp-28], %o1
add     %o0, %o1, %o0
st      %o0, [%fp-32]
.LL2:
        ret
        restore
.LLfe1:
        .size    main,.LLfe1-main
```

Arquitetura *X86*

```
.file   "exemplo.c"
.version        "01.01"
gcc2_compiled.:
.text
.p2align 2,0x90
.globl main
.type    main,@function
main:
        pushl %ebp
        movl %esp,%ebp
        subl $24,%esp
        movl $0,-16(%ebp)
        movl $50,-8(%ebp)
        movl $150,-4(%ebp)
        movl $200,-12(%ebp)
        movl -4(%ebp),%eax
        movl -8(%ebp),%edx
        addl %edx,%eax
        movl -12(%ebp),%edx
        addl %eax,%edx
        movl %edx,-16(%ebp)
.L2:
        leave
        ret
.Lfe1:
        .size    main,.Lfe1-main
```

Figura 11 - Assembly gerado no GCC para arquitetura SPARC e X86.

Na Figura 11, é mostrado o *assembly* para o exemplo em questão, gerado pelo GCC. Ressalta-se novamente a utilização da pilha na arquitetura SPARC mantendo as operações sobre registradores e o salvamento de contexto da arquitetura X86.

Nos códigos em *assembly* gerados pelos compiladores GCC e LCC anteriormente analisados é conveniente ressaltar que pode ocorrer alguma divergência na seqüência de instruções utilizada para geração do *assembly*. Esse fato deve-se à dependência da versão do processador escolhido ou do compilador utilizado. Isso pode determinar que nem sempre um *assembly* parametrizado para o mesmo processador em compiladores diferentes (LCC x GCC) ou em compiladores iguais (GCC 2.95 x GCC 3.0) com versões diferentes, será idêntico.

No próximo item relatamos a escolha realizada e seus motivos.

2.6 Definição do Compilador

O estudo dos compiladores envolveu o exame da documentação, instalação e geração de código. Na análise foram construídos exemplos de programas, em padrão ANSI C, com conjunto de operações resumido (atribuição, adição, subtração, divisão, multiplicação, alocação de memória, etc), dos quais foram gerados códigos em *assembly* para máquinas SPARC e X86, com o objetivo de verificar a portabilidade exigida pelo projeto.

Na análise do compilador GCC e do compilador LCC foram comparados os resultados da compilação ponderando as necessidades e delimitações do trabalho a ser realizado. A seguir listamos os aspectos considerados na definição da ferramenta do trabalho exposto:

a) rotinas de instalação e configuração bem definidas e documentadas;
b) possibilidade de entendimento rápido das rotinas internas e do funcionamento do compilador;
c) complexidade do código *assembly* gerado;
d) utilização de código padrão ANSI C;
e) versão com portabilidade acessível;
f) facilidade de compreensão do método de inserção e configuração ou personalização do compilador para determinado processador-alvo;
g) geração de *assembly* para máquinas com característica RISC - *Reduced Instruction Set Computer* [2];
h) prazos limites baseados no cronograma do projeto.

Ressalta-se que a pesquisa realizada, bem como a decisão sobre qual compilador foi escolhido, não levou em conta o aspecto otimização de código ou características internas tais como tipo de redução utilizada para regras da gramática, eliminação de código morto, eliminação de sub-expressões, etc. Dado que o objetivo do trabalho é a configuração do compilador para geração de código *assembly* a partir de um programa fonte em linguagem C, aspectos de otimização de código gerado não fazem parte do escopo desta implementação. Além disso, foram levados em conta outros aspectos importantes, dentre eles a facilidade de compreensão e a possibilidade de repetição dos passos da personalização do compilador para outros processadores-alvo.

Abaixo listamos alguns aspectos e características analisadas onde os compiladores se equiparam, tais como:

a) geração de código a partir de fontes em ANSI C;
b) distribuição gratuita e de fácil acesso;
c) rotinas de instalação e configuração bem definidas e documentadas;
d) geração de *assembly* para máquinas com características RISC - *Reduced Instruction Set Computer*;
e) versão com portabilidade para plataformas Unix/Linux.

Frente a todas essas características compatíveis com o projeto, o grupo optou pelo caminho da exclusão baseado na análise dos fatores críticos que influenciam o trabalho realizado. Exemplos destes aspectos são: o tempo e/ou prazo para execução das atividades e complexidade da ferramenta a ser utilizada.

Ponderadas todas essas variáveis, optou-se pela escolha do compilador LCC. Isto se justifica porque o LCC cumpre na totalidade os aspectos desejados para execução das atividades inerentes ao projeto. Além disso, esta ferramenta possui documentação detalhada de seus processos

internos e estruturas utilizadas na sua execução, bem como detalhamento de sua implementação [31].

O Capítulo 3 descreve a estrutura interna do compilador LCC, bem como as rotinas de instalação e configuração deste. A compreensão do funcionamento do compilador é fundamental para o entendimento de todas as etapas relativas à portabilidade e configuração para uma nova arquitetura, objetivo principal do trabalho.

3. COMPILADOR CONFIGURÁVEL LCC

Este Capítulo tem por objetivo apresentar, em linhas gerais, o funcionamento do compilador LCC. São mostradas algumas estruturas de dados utilizadas no processo de compilação, ressaltando que não faz parte do escopo deste trabalho o detalhamento sobre a implementação do compilador.

Além disso, este capítulo mostra uma visão geral das relações existentes entre o *back end* e o *front end*, relacionando suas funções e características sem detalhar, por exemplo, métodos de otimização. Isto se faz necessário pois o compilador LCC funciona fundamentado em dois módulos e uma *interface* entre esses módulos, que possibilita o compartilhamento de dados entre *front end* e o *back end*. Este compartilhamento de dados se dá através de um conjunto de estruturas de dados chamadas coletivamente de *interface* de geração de código.

Na Figura 12, apresenta-se uma visão geral das fases de compilação executadas pelo compilador LCC, e uma visão geral dos resultados de cada uma destas fases.

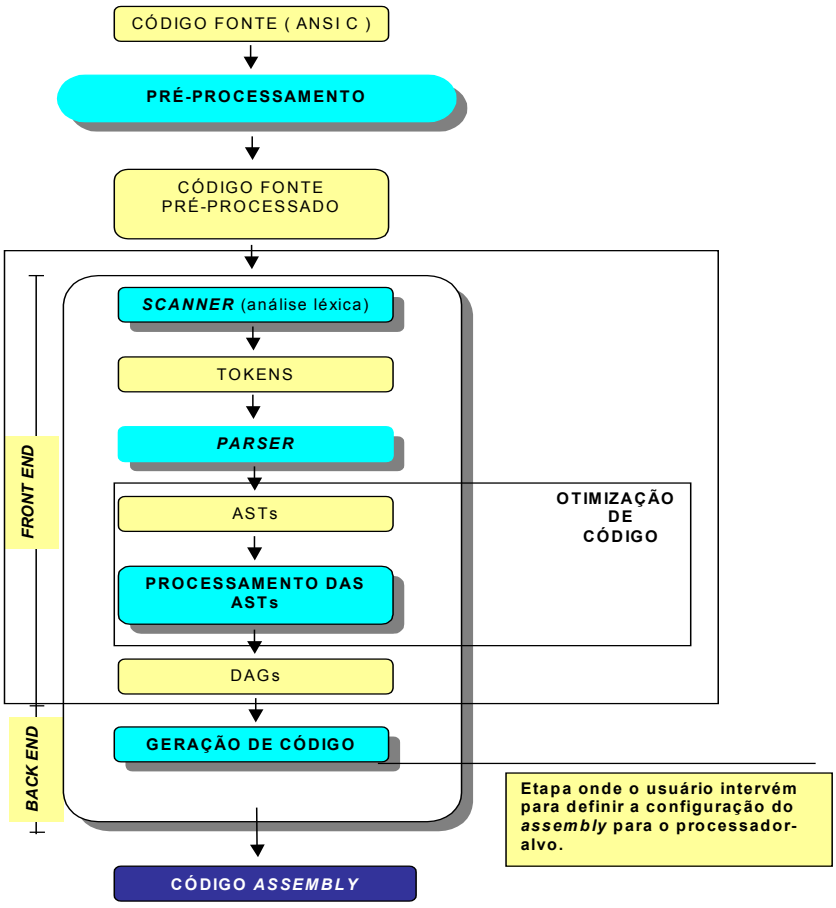

Figura 12 - Fases da compilação.

A entrada do processo é o código em linguagem C fornecido pelo usuário. Pode-se observar que antes de tudo acontece o pré-processamento do código, que é executado por uma ferramenta independente do compilador (o pré-processador). Este tem como função tratar das diretivas de compilação inseridas no código fonte, tais como *#define* e *#include*, preparando-o para a compilação.

A próxima fase é o início da compilação propriamente dita. Esta se inicia pela análise léxica que produz as cadeias de *tokens*. Após isso segue o *parser*, descendente-recursivo, que se baseia nos *tokens* para efetuar análise sintática e semântica do código de entrada. O produto do *parser* são as *ASTs (Abstract Sintax Trees)* [31] ou Árvores Sintáticas Abstratas [29], que são uma representação do código fonte resultante da tradução dirigida por sintaxe [29]. Essas árvores geradas pelo *parser* são dadas como entrada para um processo que as transforma em uma lista de *DAGs* (*Directed Acyclic Graphs*) [31], ou Grafos Dirigidos Acíclicos – GDA [29]. A representação em forma de *DAGs* se diferencia de uma representação *AST* devido a operações realizadas pelo *parser* sobre essas árvores, tais como eliminação de sub-expressões comuns, como mostramos na Figura 13. Tais modificações incluem ainda a inclusão de atributos (herdados ou sintetizados [29]) gerados pela análise sintática e semântica sobre nodos das *ASTs*.

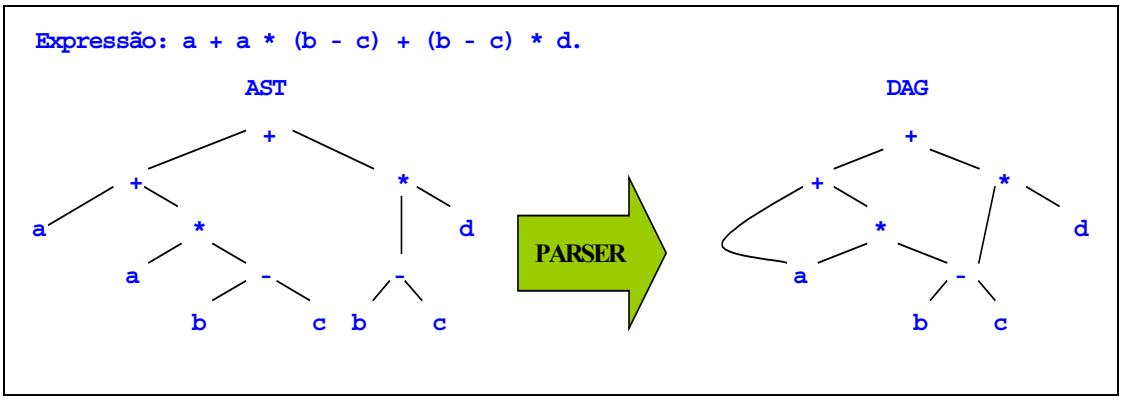

Figura 13 - Transformações do *parser* sobre *ASTs*.

Até este ponto do processo de compilação são executadas funções independentes de processador-alvo. Tais funções fazem parte do que se chama de *front end*.

Tendo ocorrido os passos anteriormente citados, entra em ação o *back end*. Esta parte executa funções dependentes de processador-alvo, tal como a geração de código em linguagem de montagem (*assembly*). A geração de código tem como entrada os *DAGs*, sobre os quais são feitas anotações com os *templates* do código *assembly* do processador-alvo. O código é gerado através de caminhamentos sobre os *DAGs* e execução de algumas funções, as quais são definidas de forma diferente para cada gerador de código. O produto final do processo é o código *assembly* para o processador-alvo escolhido pelo usuário. A descrição da arquitetura, integrada no *back end,* baseia-se em uma especificação de gramática no padrão EBNF (*Extended Backus-Naur Form*) [33], onde estão definidas as ações a serem realizadas para cada operador *DAG* reconhecido. Essa especificação é similar à utilizada pelo *software* LEX/YACC. O conjunto de operadores *DAG* será discutido mais adiante, na Seção 3.3.4 deste Capítulo.

Essa descrição da arquitetura é interpretada pelo *software Lburg* [33]. Esse *software* é responsável pelo processamento sobre essa "descrição do processador-alvo", formalizada no arquivo do tipo *md (machine description)*, gerando a "descrição do processador-alvo para o compilador".

Deve-se diferenciar entre "descrição do processador-alvo" e "descrição do processador-alvo para o compilador". A "descrição do processador-alvo" é um formato que permite a um usuário especificar um novo processador. A "descrição do processador-alvo para o compilador" é um arquivo em linguagem C, obtido a partir do arquivo de descrição *md*, durante a etapa de compilação do compilador (ou etapa de instalação). Uma vez o compilador instalado, pode-se selecionar o processador desejado via linha de comando, indicando o módulo a ser utilizado. Por exemplo, para o LCC: "*lcc –Wf-target=processador-alvo arquivo.c*".

Mostramos um exemplo parcial de um arquivo do tipo *md* na Figura 14. Com o símbolo "%" definem-se os terminais juntamente com o símbolo inicial, no caso "*start"*, e respectivos valores para os terminais. Após o símbolo "%%" estão inseridas a gramática e as ações para geração de código que devem ser realizadas. Mais detalhes a respeito do formato do arquivo *md* serão discutidos na Seção 3.3.2.

```
                Proc-alvo.md

%start stmt                              ┐
%term ADDI=309 ADDRLP=295 ASGNI=53       │ Terminais
%term CNSTI=21 CVCI=85 INDIRC=67         ┘

%%                                       ┐
con:    CNSTI                            │
addr:   ADDRLP                           │
addr:   ADDI(reg,con)                    │
rc:     con                              │ Regras
rc:     reg                              │
reg:    ADDI(reg,rc)                     │
reg:    CVCI(INDIR(addr))                │
reg:    addr                             │
stmt:   ASGNI(addr,reg)                  ┘
...
```

Figura 14 – Exemplo parcial de uma especificação *Lburg* [31].

O *assembly* produzido é entrada para um montador, que gera o respectivo código objeto.

Nas Seções a seguir relacionaremos o resultado das fases de compilação, o funcionamento e as estruturas utilizadas pelo compilador.

3.1 Produtos das Fases de Compilação

Nesta Seção, são mostrados os resultados dos passos intermediários do processo de compilação ilustrados na Figura 12, observando, em linhas gerais o funcionamento do compilador, seus principais componentes, bem como estruturas de dados utilizadas neste processo. A arquitetura a ser utilizada como exemplo é a X86, para a qual os dados do arquivo *md* serão omitidos.

Cada etapa do processo de compilação transforma o programa em uma representação de dados diferente. Fonte pré-processado, *tokens*, árvores sintáticas (*ASTs*), grafos acíclicos dirigidos (*DAGs*) e listas encadeadas destes grafos são exemplos dessas estruturas.

O código que utilizaremos como exemplo é apresentado na Figura 15, e tem por função a adição de dois valores, retornando um valor convertido de *float* em *int*, utilizando arredondamento.

```
int round (f) float f;
{
      return f + 0.5; /* adição com arredondamento */
}
```

Figura 15 - Código exemplo [31].

O primeiro componente a entrar em ação no processo de compilação é o pré-processador C. Este expande macros, inclui arquivos de cabeçalho e seleciona código compilado condicionalmente. O LCC utiliza-se de um pré-processador separado (que não faz parte do código executável do compilador, sendo distribuído juntamente como uma ferramenta utilizada na compilação). O pré-processador utilizado pelo LCC lê o código do exemplo visualizado na Figura 15 e produz como saída o código da Figura 16.

```
# 1 "sample.c"
int round (f) float f;
{
      return f + 0.5;
}
```

Figura 16 - Código pré-processado [31].

Em nosso exemplo, o pré-processador não tem muito mais o que fazer além de ignorar os comentários, e emitir a diretiva "#" para comunicar ao compilador o nome do arquivo e a linha que está processando. Normalmente um código fonte possui outras diretivas, tais como *#include* para incluir arquivos de cabeçalho, *#define* para definir constantes e macros, entre outras.

Depois do pré-processador ter feito seu trabalho, entra em ação o compilador propriamente dito. Ele começa com o analisador léxico ou *scanner*, que transforma o arquivo de entrada em seqüências de *tokens*, como mostrado na Figura 17. A coluna da esquerda é o código do *token* e a coluna da direita é o valor associado a este, se existir algum. Por exemplo, o valor associado à palavra reservada *int* é o valor de *inttype*, que representa o tipo inteiro. Os códigos para os *tokens* de apenas um caracter são os códigos ASCII para os mesmos. O *token* EOI marca o fim da entrada (*end of input*) ou fim da cadeia de *tokens*.

A próxima fase do processo é o *parsing*. Nesta fase, o compilador reconhece a cadeia de *tokens* de acordo com as regras de sintaxe da linguagem C. Esta fase também verifica se existem erros semânticos. Por exemplo, verifica se os tipos dos operandos são válidos, em operações tais como adição. Também verifica a existência de conversões implícitas, tais como na adição feita na função *round*, que é uma combinação válida, onde f é um *float* e 0.5 é um *double*. Nessa operação o resultado que é *double,* é convertido para *int* implicitamente, pois o retorno da função é *int*.

```
INT                 inttype
ID                  "round"
`(`
ID                   "f"
`)`
FLOAT               floattype
ID                   "f"
`;`
`{`
RETURN
ID                   "f"
`+`
FCON                 0.5
`;`
'}'
EOI
```

Figura 17 - *Tokens* e seus valores associados para o exemplo [31].

O resultado desta fase para o nosso exemplo são duas árvores sintáticas mostradas na Figura 18. Cada nodo representa uma operação básica.

A primeira árvore representa a conversão do parâmetro de entrada de *double* para *float*. Esta conversão é necessária pois assume-se, para este exemplo, que o parâmetro passado para a função é um *double*. Esta árvore descreve a atribuição de um *float* (ASGN+F) a uma posição de memória com o endereço &f (ADDRF+P à esquerda). Ela também computa o valor buscado (INDIR+D) do endereço &f (ADDRF+P à direita) para a atribuição, convertendo-o para *float* (CVD+F).

A segunda árvore implementa o único comando explícito do exemplo, e retorna um *int* (RET+I). O valor é computado buscando o *float* (INDIR+F) da posição de memória &f (ADDRF+P), somando-o com a constante *double* 0.5 (CNST+D) e truncando seu resultado para *int* (CVD+I). No nodo ADDRF+P da árvore direita existe uma referência para o &f definido na tabela de símbolos. O mesmo acontece nodo ADDRF+P da árvore esquerda.

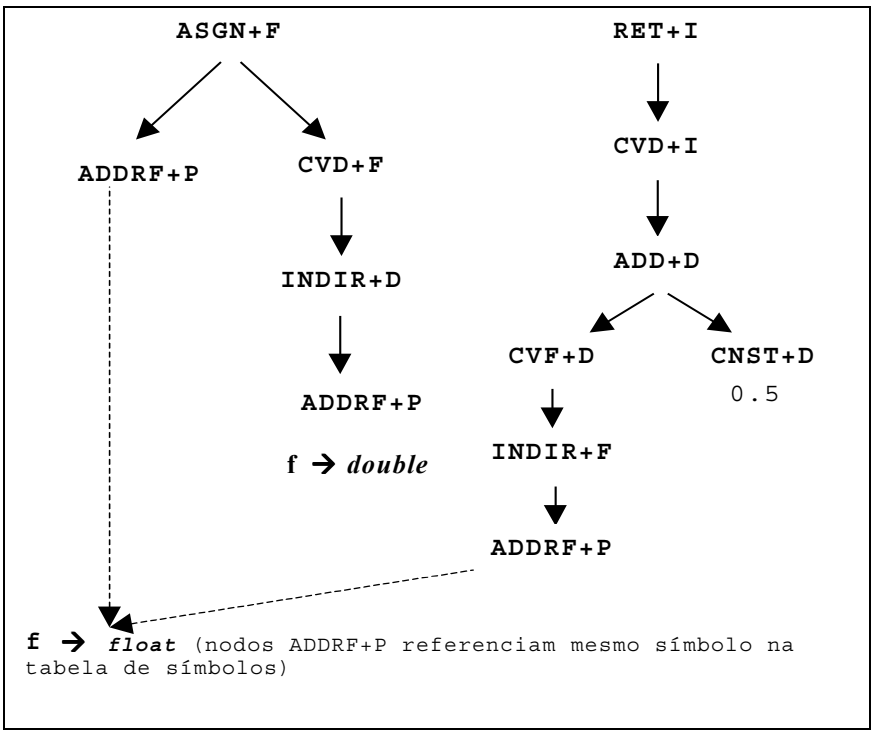

Figura 18 - Árvores sintáticas para o exemplo [31].

Essas árvores tornam explícitos muitos fatos que são implícitos no código fonte. Um exemplo desses fatos são as conversões mencionadas no parágrafo anterior. Essas conversões estão implícitas no código fonte, mas acontecem explicitamente na gramática do ANSI C e por conseguinte nas árvores que representam este padrão.

As árvores representam atribuições de tipos aos operadores explicitamente, por exemplo, em nosso código a adição não possui um tipo, mas em contrapartida, na árvore ela possui. Essa análise semântica é feita enquanto o *parser* faz o reconhecimento da entrada.

A partir das árvores mostradas na Figura 18, o LCC produz os grafos acíclicos dirigidos – *DAGs* – 1, 2 e 3 mostrados na Figura 19.

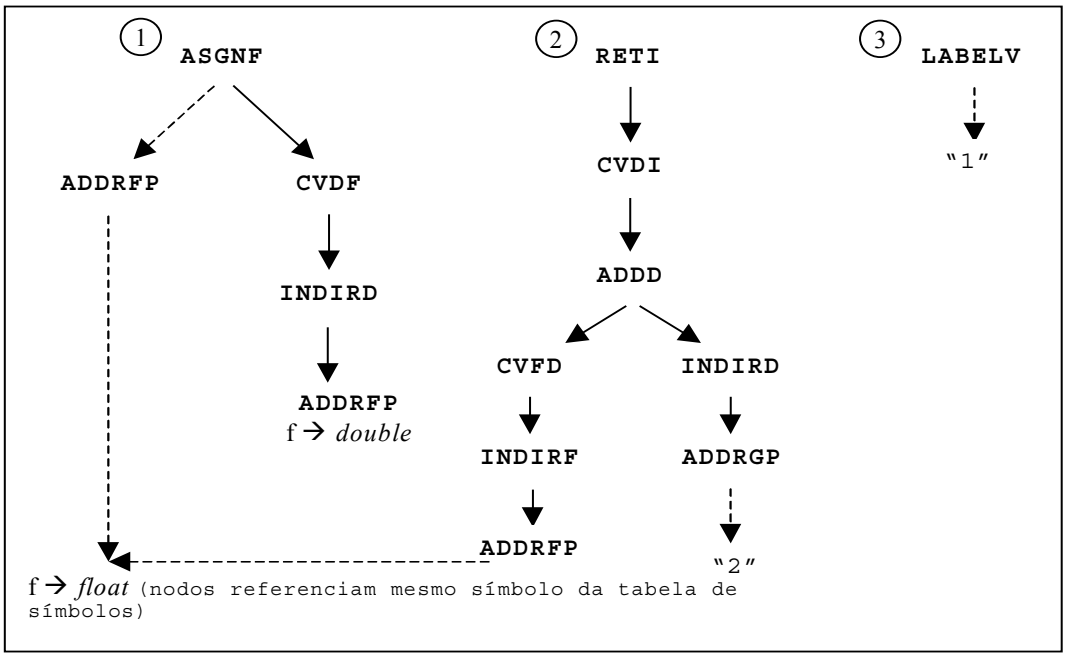

Figura 19 - *DAGs* geradas para o exemplo [31].

Os *DAGs* identificados com os números 1 e 2 vêm das árvores mostradas na Figura 18. Já o *DAG* número 3 sinaliza o final da função *round*, após o qual pode haver uma instrução para retorno de sub-rotina ou armazenamento de valores de retorno da função (via memória ou registradores).

Os operadores da Figura 19 são escritos sem o sinal '+' para identificar as estruturas como *DAGs* ao invés de árvores sintáticas. As linhas sólidas indicam instruções e as pontilhadas indicam partes de instruções.

A transição das árvores sintáticas para *DAGs* ainda traz à tona mais alguns fatos que ainda permaneciam implícitos. Por exemplo, a constante 0.5, que apareceu no nodo CNST+D na árvore da Figura 18, aparece como uma variável estática com o nome de "2" e o operador CNST+D foi substituído por operadores que desenvolvem o endereço da variável (ADDRGP) e trazem seu valor (*fetch*) da memória (INDIRD).

A transição de árvores sintáticas para *DAGs* também elimina instâncias da mesma expressão, as quais são chamadas sub-expressões. Opcionalmente, cada réplica dessas sub-

expressões pode ser eliminada atribuindo seu valor a uma variável temporária e utilizando tal variável em diversos lugares. O LCC usa essa opção.

Para cada função os respectivos *DAGs* são enfileirados em uma *code list* ou lista de código. Na lista de código mostrada na Figura 20 esses *DAGs* aparecem na ordem em que devem ser executados. Essa lista de código é construída pelo *front end* para cada função existente. Cada nodo da lista a partir de *start* representa um componente do código da função *round*. A partir daí o *back end* é acionado para a geração do código *assembly* correspondente para aquela função.

Os nodos *Defpoint* identificam localizações do código fonte que podem ser utilizadas para *breakpoints*, por exemplo. Os nodos *Blockbeg* e *Blockend* identificam os limites da função *round*. Os nodos *Gen* levam os *DAGs* identificados com os rótulos 1 e 2 na Figura 19 e o nodo *Label* leva o *DAG* identificado pelo número 3.

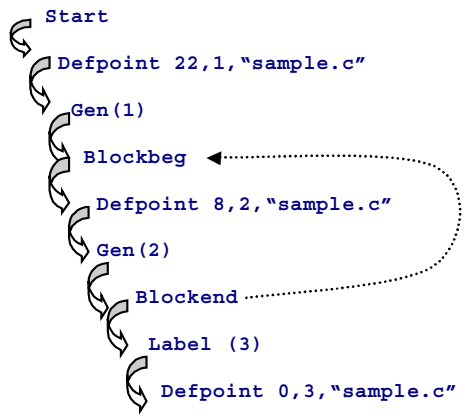

Figura 20 - Lista encadeada do exemplo [31].

A cada elemento da lista de código gerada pelo *front end* para uma função, há uma respectiva rotina de tratamento no *back end* específica do processador-alvo que é acionada (*interface functions* do arquivo *md*) [31].

A próxima fase da compilação é a geração de código. A partir daqui, as estruturas de dados que representam o programa passam do *front end* para o *back end*. O *back end* entra em funcionamento para gerar código percorrendo a lista encadeada formada no *front end*, tratando os nodos conforme as funções *(interface functions)* definidas no arquivo *md* (ver Seção 3.3.2). Desta forma a tradução destas estruturas para código *assembly* específico do processador-alvo é acionada. Ressalta-se que alguns campos desses nodos não serão visitados pelo *back end*, por serem de exclusividade do *front end*.

Os geradores de código do LCC têm seu funcionamento auxiliado por uma gramática que faz o mapeamento dos *DAGs* para instruções *assembly*, conforme mencionado na introdução do Capítulo 3.

A Figura 21 mostra os *DAGs* de nosso exemplo após ter sido anotado com código *assembly* para a arquitetura X86, a qual utilizamos para exemplificar os resultados da compilação.

O padrão "***%dígito***" denota que parte do código *assembly* corresponde a um dos nodos filhos, onde o nodo filho mais à esquerda é 0 e o mais à direita é 1 (ou seja, *kid[0]* ou *kid[1]* – ver

Seção 3.3.2). O padrão ***%letra*** denota um símbolo proveniente da tabela de símbolos, para o qual o nodo possui uma referência (ver Seção 3.3.2).

As linhas sólidas ligam instruções, e as linhas pontilhadas ligam partes de instruções tais como modos de endereçamento. Por exemplo, na primeira árvore, os nodos ASGNF e INDIRD levam instruções, já os dois nodos ADDRFP levam seus operandos. O nodo CVDF que estava no operando direito de ASGNF na Figura 19 desapareceu. Este nodo foi substituído porque a instrução do nodo ASGNF já faz a conversão e a atribuição. Essa substituição é resultado da otimização realizada durante as operações do *parser* no processo de compilação.

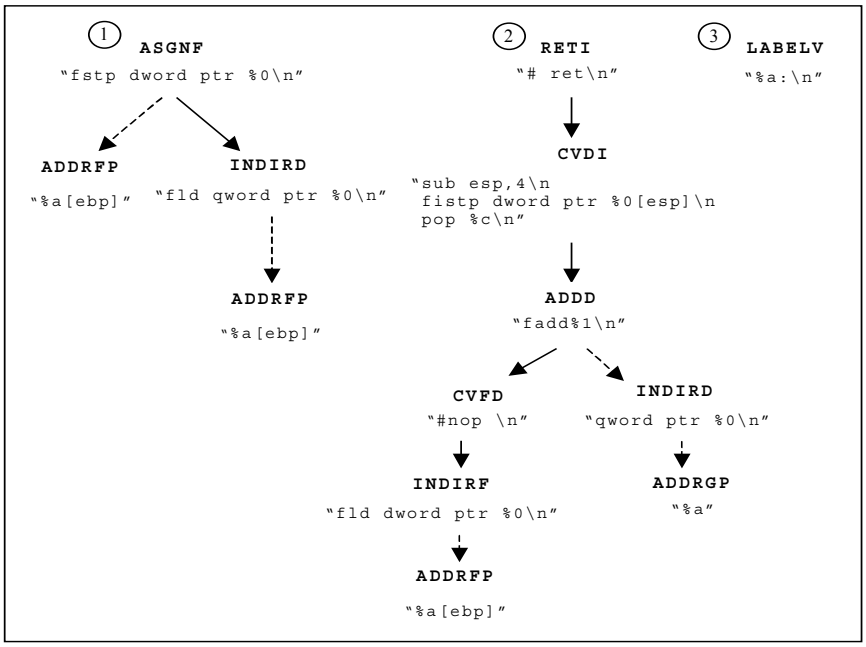

Figura 21 – *DAG* anotado com código *assembly* para X86.

Como comentado anteriormente, as anotações feitas nos nodos são padrões de código *assembly* formalizados no arquivo *md* que serão utilizados na geração de código. Algumas instruções que fazem parte destes padrões, no caso da arquitetura X86, são: instrução ***fld*** que carrega (*load*) um valor ponto flutuante (*float*) na pilha; ***fstp*** desempilha e armazena o *float*; ***fistp*** funciona de maneira análoga à anterior, mas só que esta instrução trunca o valor e o armazena como inteiro; ***fadd***, que desempilha dois valores e empilha sua soma; ***pop***, que desempilha um valor e armazena em um registrador. Na Figura 22 mostramos um *template* do *assembly* gerado para a função *round* a partir das *DAGs* da Figura 21.

```
fld qword ptr %a[ebp]\n       ; busca f (double) da mamória
fstp dword ptr %a[ebp]\n      ; armazena f na pilha como float
fld dword ptr %a[ebp]\n       ; busca f da pilha
# nop\n
fadd qword ptr %a\n           ; adiciona f (float) com 0.5 (double)"
sub esp,4\n                   ; ajusta o ponteiro da pilha para receber inteiro
fistp dword ptr %0[esp]\n     ; armazenza f como inteiro na pilha
pop %c\n                      ; retorna o resultado da adição em registrador
# ret\n
%a:\n
```

Figura 22 – *Template* de código *assembly* gerado a partir do caminhamento sobre as árvores da Figura 21 [31].

Cada linha desse *template*, mostrado na Figura 22, é especificada no arquivo *md* associado à respectiva regra para reconhecimento do nodo *DAG* na geração do código como mostrado na descrição parcial da Figura 23.

```
...
stmt: ASGNF(addr,reg)     "fstp qword ptr %0\n"
...
addr: ADDRFP              "%a[ebp]"
...
reg:  memf                "fld %0\n"
...
memf: INDID(addr)         "qword ptr %0"
...
acon: ADDRGP              "%a"
...
stmt: RETI(reg)           "# ret\n"
...
reg:  CVDI(reg)           "sub esp,4\n
                           fistp dword ptr %0[esp]\n
                           pop %c\n
...
reg:  ADDD(reg,flt)       "fadd%1\n"
...
reg:  CVFD(INDIRF(addr)) "fld dword ptr %0\n"
...
stmt: LABELV              "%a:\n"
...
```

Figura 23 - Fragmento de arquivo *md* da arquitetura X86 [31].

O próximo passo do compilador consiste em encadear os nodos que correspondem às instruções, na ordem em que devem ser emitidas. Além disso ocorre também a alocação de registradores para os nodos de instruções que necessitem desse recurso.

O resultado desta última parte do processo de compilação é o código *assembly*. O código final resultante do exemplo utilizado na Figura 15 é mostrado na Figura 24. As linhas que aparecem no código delimitam as partes principais.

Na primeira parte são inseridas diretivas emitidas como padrão para qualquer código gerado em um processador X86. Já na segunda parte é realizada a seqüência de salvamento do contexto. As quatro instruções ***push*** salvam os valores de alguns registradores (salvamento de constexto); a instrução ***mov*** estabelece o *frame pointer* para esta chamada de *round*.

A terceira parte é o código emitido a partir dos *DAGs* anotados da Figura 21, que corresponde ao corpo da função *round*. A quarta parte é a seqüência de saída, que restaura os registradores salvos na seqüência de entrada.

Na última parte aparecem os dados inicializados e a seqüência padrão de instruções de fim de código. Para a função *round*, os dados consistem apenas da constante 0.5 e de um *label* L2 que é o endereço da variável inicializada na representação IEEE 64 bits (*000000003fe00000H*), para a constante 0.5.

```
.486                                                    1
.model small
extern _turboFloat:near
extern _sertargv:near
-------------------------------------------------------------
                                                        2
public _round
_TEXT segment
_round:
push ebx
push esi
push edi
push ebp
mov ebp, esp
-------------------------------------------------------------

                                                        3
fld qword ptr 20[ebp] #busca da memória "f"
                      # como float
fstp dword ptr 20[ebp]#salva na pilha como int
fld dword ptr 20[ebp] #busca "f" da pilha
fadd qword ptr L2     #executa a adição
sub esp, 4            #conversão do resultado
fistp dword ptr 0[esp]# para inteiro
pop eax
-------------------------------------------------------------
                                                        4
L1:
mov esp, ebp
pop ebp
pop edi
pop esi
pop ebx
ret
-------------------------------------------------------------

                                                        5
_TEXT end
_DATA segment
align 4
L2 label byte
dd 00H, 03fe00000H
_DATA end
end
```

Figura 24 - Código *assembly* gerado [31].

3.2 Estrutura Interna do Compilador LCC

Nesta Seção apresenta-se a estrutura interna [31] do compilador LCC, separada, apenas com intuito de melhor visualização, em dois módulos: *(i) front end* e *(ii) back end*. Estes módulos atuam em conjunto através de uma *interface* de compartilhamento de dados. Esse compartilhamento se dá através de uma estrutura onde alguns campos são manipulados apenas pelo *front end* ou pelo *back end* e outros são utilizados por ambos, e ainda por funções que também são mutuamente ou exclusivamente utilizadas.

A análise léxica, sintática e semântica do código fonte é realizada pelo *front end*, sendo que o *back end* por sua vez é responsável pela interpretação da linguagem intermediária e geração de código *assembly* final.

A *interface* realiza o compartilhamento de dados entre os módulos com vistas a proporcionar a independência existente, que visa facilitar a configuração para diferentes processadores. Ressalta-se que não existe uma função ou um módulo que proporciona exclusivamente a *interface* ou compartilhamento dos dados, mas sim um conjunto de funções que

são utilizadas, ora pelo *front end* ora pelo *back end*. Esse conjunto de funções atua sobre uma estrutura de dados onde como já mencionado anteriormente possui determinados campos manipulados ora pelo *back end* ora pelo *front end*.

Para melhor compreensão incluímos a Figura 25 na qual visualiza-se o compilador, baseado na sua estrutura de funcionamento. Nas Seções a seguir serão realizados comentários contemplando os dois módulos citados anteriormente e as estruturas utilizadas que tem por objetivo tornar acessível a possibilidade de configuração do LCC.

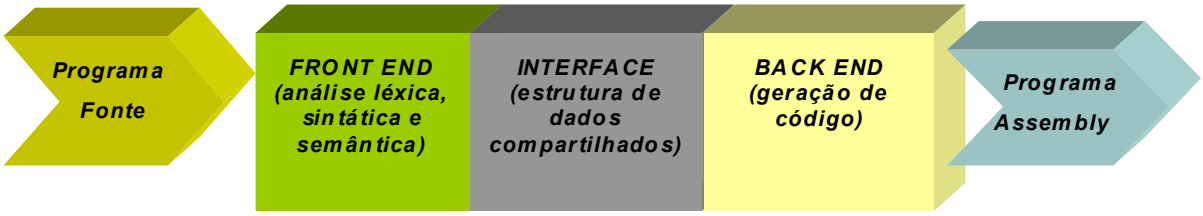

Figura 25 - *Front end x Interface x Back end.*

A título de ilustração incluímos a Figura 26 que mostra uma visão geral sobre a função dos módulos e respectivos arquivos fonte do compilador LCC.

Função do módulo	Header	Módulos
Definições comuns	c.h	
Infraestrutura e estruturas de dados		alloc.c, string.c, sym.c, types.c, list.c
Interface de geração de código	c.h	bind.c, null.c e symbolic.c
I/O e análise léxica	token.h	input.c, lex.c, e output.c
Parsing e análise semântica		error.c, expr.c, tree.c, enode.c expr.c, simp.c, stmt.c, decl.c, main.c e init.c
Geração de código intermediário		dag.c
Debugging e *profiling*		event.c, trace.c, prof.c e profio.c
Seleção de instruções independentes de processador-alvo e gerenciamento de registradores	config.h	gen.c
Geradores de código		mips.md sparc.md x86.md

Figura 26 - Funções dos arquivos fonte do LCC [31].

A Figura 26 mostra as funções dos arquivos fonte do compilador LCC e o arquivo de cabeçalho correspondente. No arquivo de cabeçalho ‘c.h’ constam todos os identificadores de tipos, funções, macros e estruturas exportados pelo módulo onde está vinculado. Já no arquivo ‘config.h’ estão as declarações de tipos específicos do *back end*. Para ajuda nas definições sobre símbolos tem-se o arquivo ‘token.h’ que contém todas as informações sobre *tokens* necessárias ao *front end*.

3.3 Módulos do compilador

Descrevemos abaixo as características e funções executadas pelos módulos *front end* e *back end*, com as estruturas de *interface* do compilador LCC. Novamente destacamos que essa apresentação foi segmentada nesses dois módulos e na *interface* apenas para melhor visualização e entendimento do leitor.

3.3.1 Front End

O *front end* é a parte do compilador que é responsável pela análise léxica, sintática e semântica no LCC. Este módulo foi desenvolvido especificamente para este compilador, sem o auxílio de ferramentas para construção de compiladores (*e.g., Lex e Yacc* [22]). Teoricamente o uso destas ferramentas simplificaria mudanças futuras e o conserto de erros, mas essa preocupação é mínima quando se trabalha com uma linguagem já padronizada como é a linguagem C.

O *front end*, no caso do LCC, foi projetado para implementar o maior número de funções independentes de processador-alvo possível, tais como passagem e retorno de estruturas e *switch statements*. Elimina sub-expressões comuns no escopo local (não globalmente), reduz expressões constantes e faz transformações nas árvores sintáticas que melhoram o código que efetua endereçamento.

O *parser* utilizado no LCC é descendente-recursivo com tradução dirigida por sintaxe [31]. O *front end* também tem como função o tratamento de erros ocorridos na fase de compilação.

Ainda sobre as características do *front end* destaca-se que este é dependente de linguagem de programação e pode ser configurável tal como é feito com a parte *back end*, ou seja, podemos ter um compilador com vários *front ends* para diferentes linguagens utilizando o mesmo *back end* [29] ou vice-versa.

3.3.2 Back End

Esta parte do compilador tem a função de proporcionar a configuração do compilador e tradução do código intermediário fornecido pelo *front end*, para código *assembly* ou código objeto. Sendo que, além disso, fornece independência ao *front end* tendo em vista que realiza a geração de código a partir de dados gerados naquele módulo.

Na elaboração da personalização mencionada anteriormente com intuito de inserção de um novo processador-alvo, além da necessidade de fácil resolução de erros da configuração, são necessárias duas características essenciais:

a) existência de gerador de "gerador de código" (*Lburg*, no caso do LCC);

b) implementação do *front end* com funções independentes de processador-alvo.

O *Lburg* [34] é um *software* utilizado no ambiente de desenvolvimento e personalização do LCC [33]. Este *software* é responsável pela geração do código C dos geradores de código utilizados pelo LCC. Estes são chamados de *back ends*, os quais são responsáveis por funções

dependentes de processador-alvo. O *Lburg* executa a geração de código dos *back ends* a partir do processamento de arquivos de especificação de geradores de código (arquivos *md* – *machine description*). Este processamento é executado quando da instalação do compilador.

As especificações contidas nos arquivos tipo *md* do *Lburg* são orientadas à linha, semelhantes às especificações utilizadas no *software Lex/Yacc* [22]. A Figura 27 descreve a estrutura de uma especificação *Lburg*. Os símbolos *term* e *nonterm* denotam respectivamente terminais e não-terminais da gramática. Nas linhas em que forem escritos os *tokens* "%{", "%}" e "%%" não deve ser escrito mais nada. Além disso, os *tokens dcl* e *rules* também devem ser escritos sozinhos, um em cada linha. A parte que corresponde a *configuration* é código em linguagem *assembly* que será reproduzido (tal qual foi escrito) no começo do código fonte do gerador de código produzido pelo *Lburg*. Se existir um segundo *token* "%%", o texto que vier depois é copiado (exatamente como está) no fim do fonte do gerador de código.

```
grammar → '%{' configuration '%}' { dcl } `%%` { rule } [%% C code]

dcl →   %start  nonterm
        %term   { term = integer }

rule →  nonterm : tree template [ C expression ]

tree → term [ '('  tree [, tree] ')'] | nonterm

template → "{ any character except double quote }"
```

Figura 27 – Gramática EBNF da especificação do *Lburg* [31][33].

O não-terminal *tree* corresponde a árvore onde os nodos são estruturas do tipo *DAG* mostrados na Figura 39 (página 42), os quais representam instruções, modos de endereçamento, conversões de tipos e outros. Já o não-terminal *template* representa as anotações mostradas na Figura 21 e Figura 22, mais especificamente.

Detalha-se nos parágrafos seguintes a estrutura do arquivo *md*, baseada nas especificações do utilitário *Lburg*. A Figura 28 apresenta um esqueleto de arquivo *md* configurado para o processador X86.

```
<x86.md>
%{
   <macros X86>
   <prefixos Lburg>
   <protótipos X86>
   <dados X86>
%}
<declaração de terminais>
%%
<regras compartilhadas>
<regras X86>
%%
<funções X86>
<definições de interface X86>
```

Figura 28 - Exemplo da estrutura de arquivo *md* para arquiteturas X86.

Na Figura 29 mostra-se a estrutura que se encontra entre os símbolos "%{" e "%}" conforme apresentado na Figura 28, no caso sendo parametrizado os dados para a máquina X86.

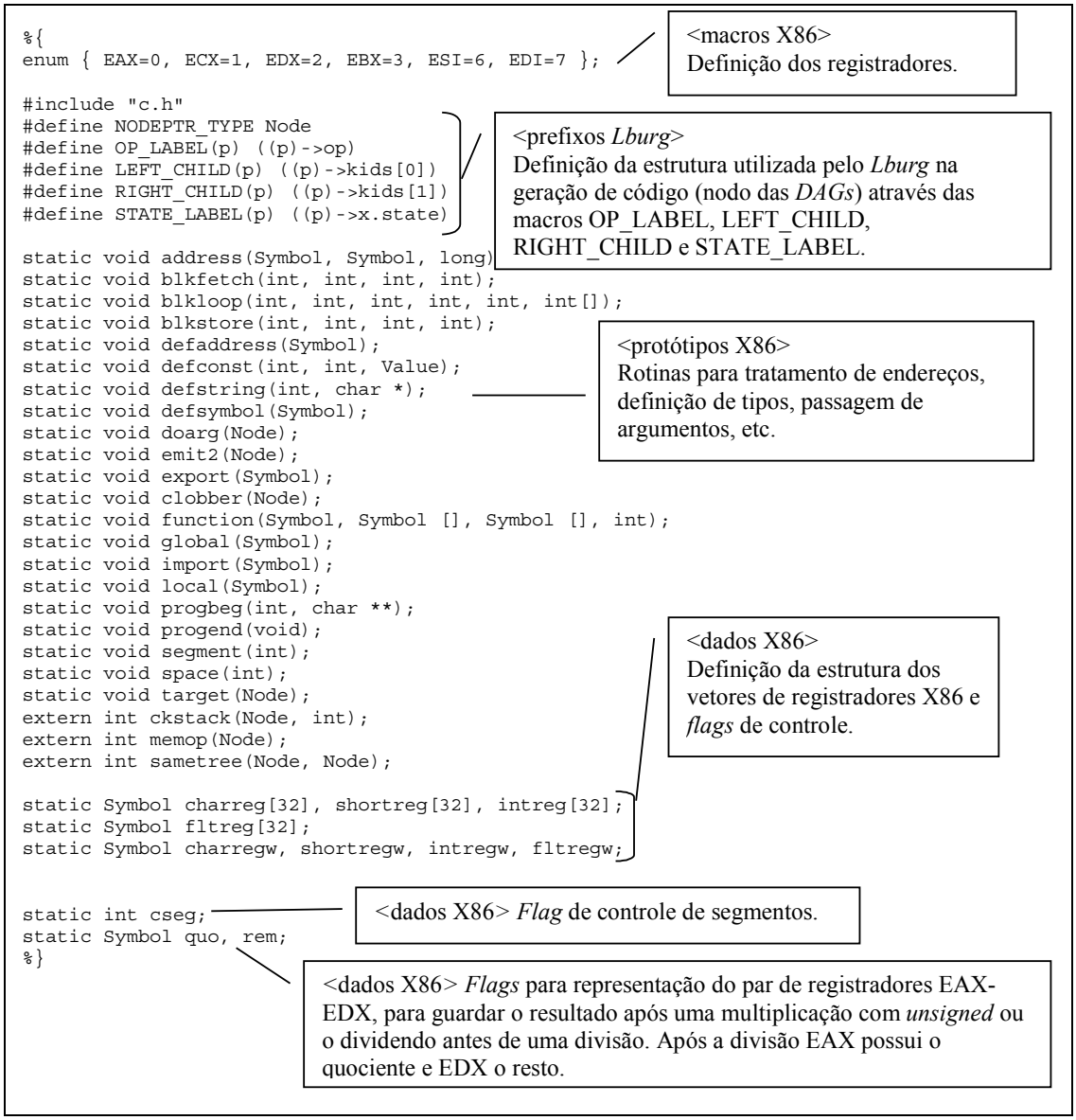

Figura 29 - Dados utilizados pelo *Lburg*, funções para configuração do processador-alvo X86 [31].

Na Figura 29 constam as definições das macros, prefixos, dados e protótipos para a arquitetura X86, nas quais o *back end* irá trabalhar para geração do gerador de código específico para esse processador.

A arquitetura X86 possui um conjunto de oito registradores definidos como EAX, ECX, EDX, EBX, ESP, EBP, ESI e EDI especificados nas macros mostradas na Figura 29. Os demais registradores da arquitetura são manipulados nos *templates* de código *assembly* inseridos nas produções da gramática de descrição (ver Figura 27) e nas funções de *interface* (ver Figura 37). Ocorre, na primeira parte do arquivo *md*, a definição das macros OP_LABEL, LEFT_CHILD, RIGHT_CHILD e STATE_LABEL que permitem ao *back end* o acesso aos campos *op*, *kids[0]*, *kids[1]* e *x.state* na geração de código. Esses campos são originários da estrutura mostrada na Figura 30.

```
Typedef struct node *Node;

struct node {
   short op;       /* campo do operador DAG */
   short count;    /* nr. de referências ou usos do nodo */
   Symbol syms[3]; /* ponteiro para operandos na tabela de símbolos */
   Node kids[2];   /* folhas ou filhos 0 e 1 */
   Node link;      /* ponteiro para próximo DAG */
   Xnode x;        /* campos manipulados pelo back end */
};
```

Figura 30 - Estrutura *Node* [31].

No código da Figura 29 consta a estrutura *Symbol* mostrado na Figura 31.

```
Typedef struct symbol *Symbol;

struct symbol {
  char name;            /* nome do símbolo */
  int scope;            /* escopo do símbolo */
  Coordinate src;       /* posição do símbolo no código */
  Symbol up;            /* ponteiro para encadear símbolos */
  List uses;            /* flag de controle de uso do símbolo */
  int sclass;           /* classe do símbolo */
  <symbol flags>        /* flags de controle */
  Type type;            /* tipo do símbolo */
  float ref;            /* nr. de referências do símbolo no código */
  union {
   <labels>             /* atributos associados a diferentes
   <struct types>          tipos de símbolos que são manipulados
   <enum constants>        através da tabela de símbolos */
   <enum types>
   <constants>
   <function symbols>
   <globals>
   <temporaries>
   } u;
  Xsymbol x;            /* campos manipulados pelo back end quando um registrador
  <debuggers extension>    recebe uma variável ou quando a informação
                           é necessária na "debugação" */
 };
```

Figura 31 - Estrutura *Symbol* [31].

As definições das funções, da Figura 29, para tratamento e verificação de tipos, variáveis, modos de endereçamento, chamadas de sub-rotinas e outros detalhes específicos do processador X86 são realizadas na seção <protótipos X86> e formalizadas na seção <funções X86> mostrada mais adiante. As funções *extern* serão utilizadas também por outros módulos do *back end*, enquanto que aquelas com o tipo *static* são exclusivas do módulo onde são definidas.

Na estrutura *Symbol* mostrada na Figura 31, ocorre dentro da *union*, a definição de atributos associados a tipos específicos. Essas estruturas são mostradas na Figura 32.

```
<symbol flags>
unsigned temporary: 1;
unsigned generated: 1;

<labels>
struct {
    int label;
    symbol equatedto;
} l;

<enum constants>
int value;

<enum types>
Symbol *idlist;

<globals>
```

```
int seg;

<struct types>
struct {
    unsigned cfields:1;
    unsigned vfields:1;
    Fields flist;
} s;

<constants>
struct {
    Value v;
    Symbol loc;
} c;

<functions symbols>
struct {
    Coordinate pt;
    int label;
    Symbol *callee;
} f;

<temporaries>
struct{
    Node cse;
} t;
```

Figura 32 - Atributos de tipos específicos [31].

Cada segmento de tipos possui uma estrutura para descrição de suas características principais. Por exemplo, para constantes é armazenado na estrutura *u.c.v* (<*constants*>) (ver Figura 31) o respectivo valor da constante. Já o campo *u.c.loc* (ver Figura 31) é utilizado quando a constante representa valores em ponto flutuante os quais não podem, devido ao número de *bits* necessário para representá-los, ser armazenados em uma instrução.

Já para informações necessárias ao *back end* na emissão de código é utilizada a estrutura *Xsymbol* cujos campos são mostrados na Figura 33.

```
typedef struct{
    char *name;
    int offset;
    Node lastuse;
    int usecount;
    Regnode regnode;
} Xsymbol;
```

Figura 33 - Campos da estrutura *Xsymbol* [31].

Na estrutura da Figura 33 os campos *lastuse* e *usecount* são utilizados para símbolos temporários, enquanto que o campo *regnode* é utilizado para referenciar registradores. O campo *x.name* (ver campo "x" do tipo *Xsymbol* na Figura 31) será utilizado pelo *back end* para armazenar o nome do símbolo que está sendo tratado. Já o campo *x.offset* (ver campo "x" do tipo *Xsymbol* na Figura 31) é utilizado para definir o tamanho do símbolo para o tratamento de endereço deste. Em símbolos que representam identificadores temporários de sub-expressões o *back end* liga os nodos que realizam *read* ou *write* nessa expressão e liga esses nodos através do campo *lastuse*, armazenando o número de usos daquela sub-expressão no campo *usecount*. O último campo denominado *regnode* é utilizado para endereçar o descritor de registrador alocado para o símbolo. Mais detalhes sobre a estrutura *Regnode* serão abordados no Capítulo 5.

Na Figura 34 visualiza-se um trecho da declaração dos terminais localizado entre os

limites "%}" e "%%", onde são definidos seus valores para utilização na gramática. Essa definição inicia sempre com o símbolo inicial "*start*", destacando que, exceto por este terminal os termos seguintes são denominados de operadores DAG. Estes operadores identificam os nodos utilizados na geração das árvores discutidas na Figura 19 e Figura 21.

```
%start stmt
%term CNSTF4=4113
%term CNSTF8=8209
...
%term ARGF4=4129
%term ARGF8=8225
...
%term INDIRB=73
%term INDIRF4=4161
...
%term CVFF4=4209
%term CVFF8=8305
...
%term NEGF4=4289
%term NEGF8=8385
...
%term CALLF4=4305
%term CALLF8=8401
...
%term ADDF4=4401
%term ADDF8=8497
...
%term LABELV=600
...
```

Figura 34 - Definição do valor de alguns operadores *DAGs* que são terminais da gramática [31].

Para geração do código *assembly* e identificação das operações com suas características corretas é necessário ainda a escrita das regras para a gramática e respectivas anotações de código a cada passo de reconhecimento executado. Tal configuração pode ser visualizada no trecho de código limitado pelos símbolos "%%" e "%%" conforme podemos ver na Figura 35. Cabe ressaltar que a sintaxe é determinada pela especificação *Lburg* que foi mostrada anteriormente na Figura 27.

```
%%
...
stmt: ASGNF8(addr,reg)     "fstp qword ptr %0\n"      7
...
memf: INDIRF8(addr)        "qword ptr %0"
...
reg:  memf                 "fld %0\n"                 3
...
base: ADDRFP4              "(%a)[ebp]"
...
stmt: RETI4(reg)           "# ret\n"
...
reg:  CVFF4(reg)           "sub esp,4\n
                            fstp dword ptr %0[esp]\n
                            fld dword ptr %0[esp]\n
                            add esp,4\n"
...
addr: ADDI4(index,base)    "%1[%0]"
...
reg:  ADDI4(reg,mrc1)      "?mov %c,%0\nadd %c,%1\n"
...
reg:  SUBI4(reg,mrc1)      "?mov %c,%0\nsub %c,%1\n"   1
...
reg:  NEGF4(reg)           "fchs\n"
...
index: LSHU4(reg,con1)     "%0*2"
...
stmt:  LABELV              "%a:\n"
...
%%
```

Custo associado a produção.

Figura 35 - Regras da gramática baseada em especificações *Lburg* [31].

Na Figura 35 são mostrados algumas linhas da gramática definida para arquitetura X86. O símbolo "%" especifica o nodo filho (0 ou 1) ou referência para a tabela de símbolos. O símbolo "?" especifica a supressão da instrução do *template* caso o registrador de destino e o de origem, indicados pelo símbolo "%c", sejam os mesmos. Consta ainda o símbolo "#", o qual quando é reconhecido significa que a emissão de código necessita de função especial para seu tratamento [31]. Cabe ressaltar que cada produção é associada ainda a um custo. Nas linhas onde essa informação é omitida o custo corresponde ao valor zero. Esta informação é necessária pois o compilador busca escolher sempre o menor custo para determinada produção da gramática na geração das instruções.

Como exemplo da última parte de um arquivo do tipo *md* temos a Figura 36 que mostra uma das funções de geração de código específicas para a arquitetura X86 declaradas na primeira seção do arquivo (<protótipos X86>), como se pode ver na Figura 29.

```
%%
static void progbeg(int argc, char *argv[]) {
        int i;
        {
                union {
                        char c;
                        int i;
                } u;
                u.i = 0;
                u.c = 1;
                swap = ((int)(u.i == 1)) != IR->little_endian;
        }
        parseflags(argc, argv);

        intreg[EAX] = mkreg("eax", EAX, 1, IREG);
        intreg[EDX] = mkreg("edx", EDX, 1, IREG);
        intreg[ECX] = mkreg("ecx", ECX, 1, IREG);
        intreg[EBX] = mkreg("ebx", EBX, 1, IREG);
        intreg[ESI] = mkreg("esi", ESI, 1, IREG);
        intreg[EDI] = mkreg("edi", EDI, 1, IREG);

        shortreg[EAX] = mkreg("ax", EAX, 1, IREG);
        shortreg[ECX] = mkreg("cx", ECX, 1, IREG);
        shortreg[EDX] = mkreg("dx", EDX, 1, IREG);
        shortreg[EBX] = mkreg("bx", EBX, 1, IREG);
        shortreg[ESI] = mkreg("si", ESI, 1, IREG);
        shortreg[EDI] = mkreg("di", EDI, 1, IREG);

        charreg[EAX]  = mkreg("al", EAX, 1, IREG);
        charreg[ECX]  = mkreg("cl", ECX, 1, IREG);
        charreg[EDX]  = mkreg("dl", EDX, 1, IREG);
        charreg[EBX]  = mkreg("bl", EBX, 1, IREG);

        for (i = 0; i < 8; i++)
                fltreg[i] = mkreg("%d", i, 0, FREG);
        charregw = mkwildcard(charreg);
        shortregw = mkwildcard(shortreg);
        intregw = mkwildcard(intreg);
        fltregw = mkwildcard(fltreg);

        tmask[IREG] = (1<<EDI) | (1<<ESI) | (1<<EBX)
                    | (1<<EDX) | (1<<ECX) | (1<<EAX);
        vmask[IREG] = 0;

        tmask[FREG] = 0xff;
        vmask[FREG] = 0;

        print(".486\n");
        print(".model flat\n");
        print("extrn __fltused:near\n");
        print("extrn __ftol:near\n");

        cseg = 0;

        quo = mkreg("eax", EAX, 1, IREG);
        quo->x.regnode->mask |= 1<<EDX;
        rem = mkreg("edx", EDX, 1, IREG);
        rem->x.regnode->mask |= 1<<EAX;
}
```

Controle sobre a posição dos *bits* mais significativos (*little endian* ou *big endian*).

Montagem dos vetores de descrição dos registradores de acordo com os tipos utilizados.

Criação de descritores de registradores de ponto flutuante com máscara zero para solucionar problema na alocação de registradores para identificadores do tipo *float.*

Definição dos registradores usados para valores temporários (*tmask*) e variáveis (*vmask*) do tipo *int*.

Definição de registradores usados em valores temporários e variáveis de ponto flutuante.

Introdução padrão do *assembly* da arquitetura X86.

Flag para controle de segmentos de código.

Inicialização das estruturas *quo* e *rem* que são utilizadas como par de registradores.

Figura 36 - Exemplo de função de tratamento do início do programa e inicialização das estruturas que representam registradores.

Além da função mostrada na Figura 36, implementada no trecho final do arquivo *md* após o símbolo "%%" (ver Figura 28), são definidas as funções de tratamento de escopo das variáveis (global ou local), reconhecimento dos tipos, ordem das operações, inicialização das estruturas dos registradores e composição dos nodos das árvores no *front end*.

Na próxima Seção são discutidos alguns aspectos do compartilhamento de dados que funciona como a *interface* para geração de código.

3.3.3 Interface de Geração de Código

Para a coexistência de *front end* e *back end* e seu trabalho em conjunto existe uma estrutura para interligação dos módulos aqui denominada de *interface* de geração de código.

No projeto desta *interface* existem alguns aspectos a serem considerados tais como:

a) Quantidade de funções implementadas pelo *back end*: uma *interface* inadequada pode vir a forçar cada *back end* a fazer o trabalho que deveria ser realizado pelo *front end,* fazendo com que a cada novo *back end* tenha que ser implementado um grande número de funções;

b) Abrangência da *interface*: se a *interface* é muito pequena pode ser impossível codificar todas as características de um novo *back end*. Por outro lado se é muito grande, o *back end* pode tornar-se complexo demais sem necessidade.

Os aspectos acima mencionados requerem do projeto uma engenharia cuidadosa dos detalhes da *interface* de geração de código. Na Figura 37 é mostrada a definição dos campos da estrutura que representa as características da arquitetura X86, inserida no arquivo *md*. Esta estrutura captura todas as informações que o *front end* necessita saber sobre o processador-alvo, incluindo ponteiros para as rotinas de *interface*, métricas dos tipos de dados e *flags* de controle. Cada processador-alvo possui sua própria *interface*.

Para a definição das características dos tipos tratados pelo processador que está sendo descrito é necessário preencher os campos entre a primeira e a décima linha da *interface* mostrada na Figura 37. As informações ali contidas estão armazenadas em três colunas para melhor visualização. Descrevem os campos (*size, align, outofline*). Todos esses campos são preenchidos com valores em *bytes*. O campo *size* descreve o tamanho do tipo, o campo *align* o alinhamento do tipo e o campo *outofline* serve como *flag* para controle de constantes daquele tipo quando ocorre a formação das árvores de código intermediário. Se *outofline* é 1 então a constante daquele tipo estará disponível na *DAG*, do contrário não.

Na Figura 38 constam também as funções que atuam na seleção de instruções. São:

a) *emit2()* – trata de trechos de códigos que necessitam de rotinas específicas;

b) *doarg()* – trata da emissão de assembly para passagem de parâmetros para chamadas de subrotinas;

c) *target()* – trata nodos DAGs que necessitam de registradores especiais;

d) *clobber()* – trata nodos DAGs que necessitam de salvamento de registradores, ou seja, efetuam, quando necessário o salvamento do conteúdo de determinados registradores em memória.

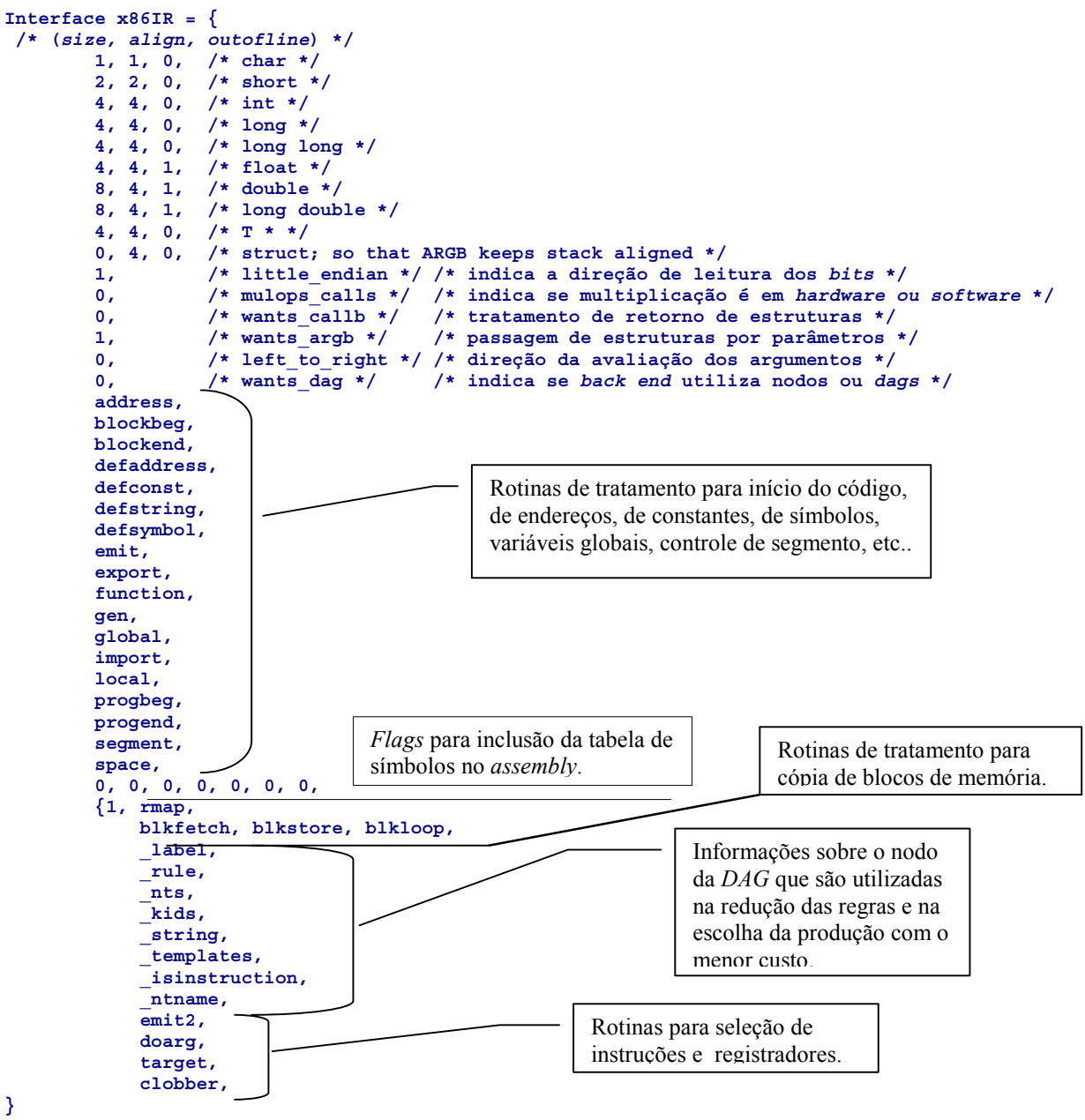

Figura 37 - *Interface record* da arquitetura X86 [31].

A *interface* consiste de algumas estruturas de dados compartilhadas, 18 funções, das quais a maioria é simples, e uma linguagem de 36 operadores - *DAG Operators* - que codificam o código executável a partir de um programa-fonte traduzido em estruturas de *DAG - Directed Acyclic Graph*. Na Figura 38 são mostradas as funções da *interface* e seus parâmetros.

As funções da Figura 38 são utilizadas para tratamento de endereços (*void adress()*), mapeamento do início e fim de bloco de código (*void blockbeg()* e *void blockend()*), definição de endereços de variáveis (*void defadress()*), definição de constantes (*void defconst()*), definição de símbolos (*void defsymbol()*), definição de *strings* (*void defstring()*), emissão de código (*void emit()*),

tratamento de variáveis globais e locais (*void global() e void local()*), mapeamento de início e fim de funções (*void progbeg() e void progend()*), controle de segmento de código (*void segment*), alocação de espaço em memória (*void space()*) e definição de símbolos que serão exportados ou importados (*void import() e void export()*).

```
void address(Symbol p, Symbol q, int n)
void blockbeg(Env *e)
void blockend(Env *e)
void defaddress(Symbol p)
void defconst(int ty, Value v)
void defstring(int len, char *s)
void defsymbol(Symbol p)
void emit(Node p)
void export(Symbol p)
void function(Symbol f, Symbol caller[], Symbol callee[], int ncalls)
Node gen(Node p)
void global(Symbol p)
void import(Symbol p)
void local(Symbol p)
void progbeg(int argc, char *argv[])
void progend(void)
void segment(int s)
void space(int n)
```

Figura 38 - Funções de *interface* [31].

Cabe ressaltar que algumas funções, definidas especificamente para o processador-alvo descrito, não constam entre as funções de *interface* da Figura 38. Tem-se como exemplo a função *target(Node)* (ver Figura 29).

3.3.4 Operadores DAG - Directed Acyclic Graph

Na formação do código *assembly* são utilizados símbolos que simplificam a operação e compartilhamento de dados entre os módulos, e que representam a linguagem intermediária do LCC. Estes símbolos são denominados *DAG Operators*. Um corpo de função, por exemplo, é formado de uma seqüência ou floresta de *DAGs* que são passados ao *back end* para serem utilizados na geração de código *assembly*.

Na Figura 39 mostramos as convenções utilizadas para operadores *DAG*, bem como os símbolos associados a cada nodo da árvore formada.

A coluna "*Kids"* informa quantos nodos-filhos o nodo indicado pela coluna "operador" possui. Já a coluna "Sufixos de tipo" indica os sufixos dos tipos que podem existir para aquele operador. Na última coluna temos ainda a operação simbolizada por determinado operador que será posteriormente traduzida pelo *back end*.

Nos nodos também é associada a informação sobre o tipo utilizado através da concatenação de números que indicam a quantidade de *bytes* tratados pelo sufixo do tipo correspondente. Por exemplo, ADDRF8 trata busca (ADDR) de endereços de uma variável *float* (F) de oito *bytes* (8).

Na Figura 40 são mostradas as conversões possíveis entre os tipos utilizados. Na Figura 40 C = *char,* D = *double,* F = *float,* I = *integer,* S = *signed*, U = *unsigned,* P = *pointer*.

kids	Operador	Sufixos de tipo	Operação
0	ADDRF	P	Endereço de um parâmetro
0	ADDRG	P	Endereço de varável global
0	ADDRL	P	Endereço de varável local
0	CNST	CSIUPFD	Constante
1	BCOM	U	Complemento bit a bit
1	CVC	IU	Conversão a partir de um *char*
1	CVT	IF	Conversão a partir de um *double*
1	CVF	D	Conversão a partir de um *float*
1	CVI	CSUD	Conversão a partir de um *int*
1	CVP	U	Conversão a partir de um *pointer*
1	CVS	IU	Conversão a partir de um *short*
1	CVU	CSIP	Conversão a partir de um *unsigned*
1	INDIR	CSIPFDB	Busca
1	NEG	IFD	Negação
2	ADD	IUPFD	Adição
2	BAND	U	*AND* bit a bit
2	BOR	U	*OR* bit a bit
2	BXOR	U	*XOR* bit a bit
2	DIV	IUPFD	Divisão
2	LSH	IU	*Shift* para a esquerda
2	MOD	IU	Modulo
2	MUL	IUFD	Multiplicação
2	RSH	IU	*Shift* para a direita
2	SUB	IUPFD	Subtração
2	ASGN	CSIPFDB	Atribuição
2	EQ	IFD	Salto se igual
2	GE	IUFD	Salto se maior ou igual
2	GT	IUFD	Salto se maior
2	LE	IUFD	Salto se menor ou igual
2	LT	IUFD	Salto se menor
2	NE	IFD	Salto e diferente
1	ARG	IPFDB	Argumento
1 ou 2	CALL	IFDBV	Chamada de função
1	RET	IFD	Retorno de função
1	JUMP	V	Salto incondicional
0	LABEL	V	Definição de label

Figura 39 - Operadores DAG [31].

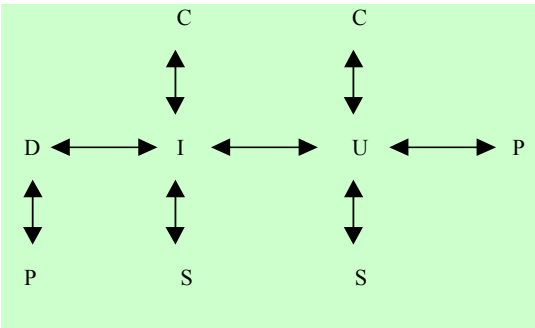

Figura 40 - Conversões possíveis entre os tipos de *DAGs* [31].

Na próxima Seção são descritos os passos necessários para a instalação e a

parametrização do LCC para um processador-alvo diferente daqueles já existentes no compilador LCC (*SPARC/SOLARIS, X86/WIN32, MIPS/IRIX, MIPS/ULTRIX, etc ...*).

3.4 Instalação e Personalização do LCC

Nesta Seção são mostrados os passos para instalar e parametrizar o compilador para um novo processador-alvo considerando a utilização de um ambiente Unix ou Linux.

Para instalar o compilador LCC, primeiramente será necessário fazer *download* dos arquivos lcc-4.1.tar (arquivo com os fontes do LCC) e BUILD-LCC.sh (*script* de instalação), sendo que o link para o *download* encontra-se em ftp://ftp.cs.princeton.edu/pub/packages/lcc/contrib/ (BUILD-LCC.sh) e no site www.cs.princeton.edu/*software*/lcc (fontes do LCC).

Após a realização do *download* dos arquivos, dentre estes o arquivo BUILD-LCC.sh (arquivo *script* encarregado da instalação do LCC) coloque o arquivo lcc-4.1.tar em um diretório destinado para a instalação do compilador (Exemplo: LCCTEMP), em ambiente UNIX ou LINUX.

Depois desse procedimento é necessário alterar o arquivo BUILD-LCC.sh, pois as modificações feitas nesse arquivo definem o diretório temporário utilizado durante a instalação e define também o diretório onde o LCC será instalado.

O próximo passo é a descompactação do arquivo lcc-4.1.tar através do comando *"tar -xvf lcc-4.1.tar"*.

Da descompactação é gerado o diretório lcc-4.1 e os subdiretórios com os fontes do LCC, suas bibliotecas e descrição do processador-alvo. Cabe ressaltar que para a instalação, via *script* do arquivo BUILD-LCC.sh, é necessário a existência prévia do compilador *GCC – GNU C Compiler* na máquina de trabalho.

Abaixo listamos as modificações e os arquivos a serem alterados para instalação do LCC, tendo como destino uma máquina Solaris/Sun, que utiliza o sistema operacional Solaris, versão 2.7.

Linhas alteradas do arquivo BUILD-LCC.sh:

1. CC=gcc (compilador em utilização)[6]
2. CFLAGS= (nenhuma flag)
3. TMPDIR=lcctemp (diretório temporário)
4. exec_prefix=/home3/gaph/visita/lcc41 (define o diretório onde será instalado o LCC)[7]

Em decorrência do sistema operacional utilizado outro arquivo alterado foi o *solaris.c*, que se encontra no diretório lcc-4.1/etc/, alteração esta no sentido de definir o diretório onde se encontram as bibliotecas que o compilador utilizará:

[6] Para a instalação do compilador o grupo utilizou versão do GCC já instalada e utilizada nas disciplinas da graduação.
[7] No caso o caminho é aquele relativo ao diretório de trabalho do grupo que está elaborando o projeto e analisando o compilador.

Linhas do arquivo original solaris.c:

1. #define LCCDIR “/usr/local/lib/lcc/”
2. #define SUNDIR “”/opt/SUNWspro/SC4.2/lib/”
3. SUNDIR “values-xa.o”, “$2”, “”,

Alterações realizadas:

1. #define LCCDIR “/home3/gaph/visita/lcc41/lib/lcc-4.1/”
2. #define SUNDIR “/usr/local/lib/gcc-lib/sparc-sun-solaris2.7/2.95.2/”
3. #define SPARCV9 “/usr/lib/”
4. SPARCV9 “values-Xa.o”, “$2”, “”,

Nas linhas 1, 2 e 3 foram definidos os diretórios onde se encontram as bibliotecas que o LCC usará na instalação e na linha 4 foi alterado o nome do arquivo *values-Xa.o*, já que no arquivo original de instalação esse arquivo com a letra “x” em minúsculo não existia.

Concluída essa etapa de configuração dos arquivos, já pode ser efetivada a instalação do compilador.

Para instalar acesse o diretório lcc-4.1 (diretório criado após a extração do arquivo lcc-4.1.tar) e digite no prompt o comando “BUILD-LCC.sh <processador de trabalho>”, onde o processador utilizado pelo grupo era SPARC. (Observação importante: o arquivo BUILD-LCC.sh deve estar dentro do diretório lcc-4.1).

O resultado desse comando é a execução do *script* de instalação do compilador, o parâmetro SPARC/SOLARIS define em qual processador e sistema operacional será instalado o LCC.

Após todo o processo de instalação, o programa executável do LCC se encontrará no subdiretório /lcc41/bin/.

Com todos os passos acima concluídos elabore um arquivo com a extensão *.c e realize a compilação através do comando “lcc exemplo.c <enter>”. Caso haja necessidade da visualização do código em *assembly* gerado utilize o comando “lcc –S exemplo.c <enter>”.

Para maiores esclarecimentos sobre os parâmetros existentes na compilação utilize o comando “lcc <enter>”.

3.4.1 Adicionando um Novo Processador-Alvo no Compilador LCC

Para a inserção de um novo processador estabelecemos que a melhor alternativa é a utilização de uma descrição (arquivo *.*md*) já existente.

O próximo passo é definir o nome do processador aqui denominada de processador GAPH. A descrição do referido processador é de teste e foi obtida através de cópia e alteração do nome de uma descrição *.md já inclusa no compilador LCC, no caso a descrição da máquina MIPS.

Abaixo relacionamos os passos necessários:

1º. Inserção do arquivo *gaph.md* no subdiretório /lcc-4.1/src/;

2º. Inserir no arquivo acima a variável denominada *Interface gaphIR*, pois é usada nas modificações realizadas no arquivo *bind.*c;

3º. Alteração do arquivo *makefile* (localizado no diretório /lcc-4.1) *e bind.c* (localizado no diretório /lcc-4.1/src/);

Linhas inseridas e alteradas no arquivo *makefile*:

a) Linha 69: $*Bgaph$O*

b) Linha 115: *$Bgaph$O: $Bgaph.c; $(CC) $(CFLAGS) -c -Isrc -o $@ $Bgaph.c*

c) Linha 123: *$Bgaph.c: $Blburg$E src/gaph.md; $Blburg src/gaph.md $@*

d) Linha[8] 237: *$(RM) $Bdagcheck.c $Balpha.c $Bprocessador.c $Bx86.c $Bsparc.c $Bx86linux.c $Bgaph.c $Bgram*

e) Linha 281*: $Bgaph.c*

Nessas alterações definimos o nome do **.md (gaph.md)* que será usado pela ferramenta *lburg* na geração do arquivo *gaph.c*.

No arquivo *bind.c* foram adicionadas as seguintes linhas com a finalidade de refenciar a interface definida no arquivo *gaph.md*:

a) 1. Linha 9: *extern Interface*[9] *gaphIR*;

b) 2. Linha 25: *"gaph", &gaphIR*,

4º. Após essas modificações, o compilador LCC deve ser instalado novamente. Para este passo utilize o comando "*BUILD-LCC.sh sparc-solaris <enter>*".

[8] Esta linha já constava no arquivo original e foi apenas modificada para o teste.
[9] É uma variável correspondente à interface de geração de código.

Na instalação o núcleo do compilador será totalmente reestruturado e as modificações realizadas proporcionarão a inserção de novo processador aqui denominada de *gaph.md*.

Para um teste de geração de código e visualização do código *assembly* desse processador utilize o comando "*lcc –S –Wf-target=gaph <arquivo.c>*".

No próximo Capítulo são descritas as características do processador definido para esse projeto, ou seja, o processador R7.

4. PROCESSADOR R7

Para que se possa efetuar a configuração do compilador para um novo processador-alvo é essencial conhecer as características deste. Este tipo de conhecimento se faz importante pois, para se projetar um bom gerador de código *assembly* para uma máquina é fundamental que se conheça a estrutura interna da mesma, bem como suas instruções. Outro dado que merece atenção são os registradores disponíveis no processador-alvo. Este dado é de extrema importância para o gerador de código, o qual precisa saber o número e o papel de cada registrador dentro do conjunto de registradores disponíveis quando está efetuando a geração de código.

Sendo assim, neste Capítulo será apresentado o processador R7 [19] (o qual será utilizado na parametrização do compilador LCC) e suas características para o desenvolvimento do projeto. É importante deixar claro que, conforme a proposta do projeto, características tais como operações em ponto-flutuante não foram implementadas.

Destacamos que o processador-alvo definido neste Capítulo é utilizado no curso de Bacharelado em Ciência da Computação da PUCRS, como objeto de estudo na disciplina de Organização de Computadores. A especificação deste processador pode ser obtida no endereço http://www.inf.pucrs.br/~moraes/org/arq_r7.pdf.

4.1 Características do Processador

O processador R7 possui as seguintes características:

a) Arquitetura *load-store* : as operações lógico/aritméticas são executadas entre registradores, e as operações de acesso à memória só executam ou uma leitura (*load*) ou uma escrita (*store*);

b) Banco de registradores: conjunto de 16 registradores, para reduzir o número de acessos à memória;

c) Memória disponível: 64 Kpalavras (palavras de 16 *bits*), o que equivale a 128 K*bytes* (endereçamento de 16 *bits*);

d) Formato regular para as instruções: todas as instruções possuem exatamente o mesmo tamanho, e ocupam 1 palavra de memória (16 *bits*). A instrução contém o código da operação e o(s) operando(s), caso exista(m).

e) Poucos modos de endereçamento: a registrador, imediato e relativo;

f) Dados e endereços são de 16 *bits* (processador de 16 *bits*);

g) Endereçamento de memória a palavra (cada endereço corresponde a 16 *bits* de conteúdo);

h) Possui apenas 1 *flag* de estado;

i) Todos os saltos e chamadas de subrotina são condicionais ao *flag* de estado. Não existem saltos incondicionais (restrição imposta para simplificar o conjunto de instruções).

j) Execução das instruções em 3 ou 4 ciclos, ou seja, CPI (Ciclos por Instrução) entre 3 e 4.

Assim, este processador-alvo é praticamente uma máquina *RISC - Reduced Instruction Set Computer*, faltando contudo algumas características que existem em qualquer máquina desse tipo, tal como *pipeline*.

4.2 Conjunto de Instruções

Nesta Seção indicaremos as instruções do processador e operações que podem ser realizadas, sendo as seguintes:

a) Operações lógicas e aritméticas binárias: soma (*ADD*), subtração (*SUB*), E (*AND*), OU (*OR*), OU exclusivo (*XOR*);

b) Operações aritméticas imediatas com constantes curtas: soma (*ADDI*), subtração (*SUBI*);

c) Operações unárias: deslocamento para direita ou esquerda e inversão (*NOT*);

d) Carga de uma metade de um registrador com uma constante (*LDL e LDH*);

e) Estabelecimento de valor para o *flag* de estado por instruções específicas: *DIF, EQL, SUP, INF*;

f) Inicialização do apontador de pilha (*LDSP*) e retorno de subrotina (*RTS*);

g) *NOP* (*no operation*): operação vazia (útil para laços de espera);

h) *HALT*: suspende a execução de instruções posteriores;

i) *LD (load)*: leitura de posição de memória para um registrador;

j) *ST (store)*: armazenamento de dado de um registrador em uma posição de memória.

Na Seção a seguir foram incluídas todas as instruções e detalhes a respeito do formato e da ação executada por cada uma delas no processador R7.

4.3 Formato e Ação das Instruções

Na Figura 41 listamos as instruções e sua estrutura com a correspondente ação que cada uma representa.

a) **Instruções Lógico-aritméticas.** Registrador destino (*target*) recebe o resultado de uma dada operação binária entre registradores fonte (*source*s): Rt ← Rs1 *opcode* Rs2. No nosso processador, o *opcode* indica 5 operações lógico-aritméticas entre 2 registradores fontes: soma, subtração, "e" lógico, "ou" lógico e "ou" exclusivo. Quando se tem operações sobre apenas um registrador fonte, os *bits* 7 a 4 da instrução indicam o tipo de operação e os *bits* 15 a 12 permanecem constantes. As 3 operações assumidas pelo segundo *opcode* são: SL (deslocamento

à esquerda), SR (deslocamento à direita) e NOT (inversão lógica).

b) Instruções de Comparação para Atribuir Valor ao *Flag* de Estado. Registrador *flag* recebe o resultado de uma dada comparação entre dois registradores fonte (*sources*) : *flag* ← Rs1 *comp* Rs2. As operações de comparação são: diferença (DIF), igualdade (EQL), maior que (SUP) e menor que (INF).

c) Instrução de Leitura/Escrita de Dados da Memória. O conteúdo da posição de memória endereçada pela soma dos conteúdos de dois registradores fonte é armazenado em um registrador destino: *Rt* ← *PMEM(Rs1 + Rs2*). Um dos registradores pode ser considerado como registrador base e o segundo como registrador de deslocamento (*offset*). Para a escrita na memória, a posição de memória é endereçada pela soma dos conteúdos de dois registradores fonte (*sources*) e recebe o conteúdo do registrador destino (*target*): *PMEM(Rs1 + Rs2)* ← *Rt*.

d) Operações em Modo Imediato Curto. Registrador destino (*target*) recebe o resultado de uma dada operação entre o próprio registrador *target* e uma constante de 8 *bits* (como todas as instruções são de uma palavra, o tamanho da constante não pode ser do tamanho da palavra). As instruções com operações que manipulam constantes curtas são:

→ carga da parte alta de um registrador (LDH): *Rt* ← *constante & RtL* (o registrador *target* recebe a constante na parte alta e mantém a parte baixa inalterada);

→ carga da parte baixa de um registrador (LDL): *Rt* ← *RtH & constante*;

→ soma/subtração em modo imediato: soma/subtração do conteúdo de um dado registrador a uma constante de 8 *bits*: *Rt* ← *Rt +/- constant*e. Desta forma, podemos somar (ou subtrair) uma constante de valor 0 a 255 ao conteúdo de um dado registrador. Importante: a execução da instrução implica completar com zeros os 8 *bits* mais significativos da constante para gerar um valor de 16 *bits*.

e) Operações de salto condicional. Numa operação de salto condicional, se o *flag* de estado estiver em '1', o PC (*Program Counter*) é alterado conforme os operandos especificados na instrução. Há três formatos para o salto condicional:

→ registrador absoluto, JMP, quando o conteúdo do novo PC está em um registrador (PC ← RS1);

→ relativo a registrador, JMPR, quando um dado registrador contém um deslocamento em relação ao PC (PC ← PC + RS1);

→ relativo imediato, JMPD, o PC é somado a um deslocamento (PC ← PC + deslocamento).

Instrução	FORMATO DA INSTRUÇÃO				AÇÃO
	15 – 12	11 - 8	7 - 4	3 - 0	
ADD Rt, Rs1,Rs2	0	R target	R source1	R source2	Rt ← Rs1 + Rs2
SUB Rt, Rs1,Rs2	1	R target	R source1	R source2	Rt ← Rs1 - Rs2
AND Rt, Rs1,Rs2	2	R target	R source1	R source2	Rt ← Rs1 *and* Rs2
OR Rt, Rs1,Rs2	3	R target	R source1	R source2	Rt ← Rs1 *or* Rs2
XOR Rt, Rs1,Rs2	4	R target	R source1	R source2	Rt ← Rs1 *xor* Rs2
ADDI Rt, *cte8*	5	R target	Constante		Rt ← Rt + ("00000000" & constante)
SUBI Rt, *cte8*	6	R target	Constante		Rt ← Rt - ("00000000" & constante)
LDL Rt, *cte8*	7	R target	Constante		Rt ← RtH & constante
LDH Rt, *cte8*	8	R target	Constante		Rt ← constante & RtL
LD Rt, Rs1,Rs2	A	R target	R source1	R source2	Rt ← PMEM (Rs1+Rs2)
ST Rt, Rs1,Rs2	B	R target	R source1	R source2	PMEM (Rs1+Rs2) ← Rt
SL Rt, Rs2	C	R target	0	R source2	Rt ← Rs2[14:0] & *flag*
SR Rt, Rs2	C	R target	1	R source2	Rt ← *flag* & Rs2 [15:1]
NOT Rt, Rs2	C	R target	2	R source2	Rt ← not (Rs2)
DIF Rs1, Rs2	D	0	R source1	R source2	if *Rs1* ≠ *Rs2* then *flag=1* else *flag=0*
EQL Rs1, Rs2	D	1	R source1	R source2	if *Rs1* = *Rs2* then *flag=1* else *flag=0*
SUP Rs1, Rs2	D	2	R source1	R source2	if *Rs1* > *Rs2* then *flag=1* else *flag=0*
INF Rs1, Rs2	D	3	R source1	R source2	if *Rs1* < *Rs2* then *flag=1* else *flag=0*
LDSP Rs1	D	4	R source1	-	SP ← Rs1 (inicializa o apontador de pilha)
NOP	D	5	-	-	nenhuma ação
RTS	D	6	-	-	PC ←PMEM(SP+1) ; SP←SP+1
HALT	D	7	-	-	suspende seqüência de ciclos de busca e execução
JMPR Rs1	D	8	R source1	-	if (*flag* =1) PC ← PC + Rs1
JSRR Rs1	D	9	R source1	-	if (*flag* =1) { PMEM(SP)←PC ; SP←SP-1; PC← PC+ Rs1 }
JMP Rs1	D	A	R source1	-	if (*flag* =1) PC ← Rs1
JSR Rs1	D	B	R source1	-	if (*flag* =1) { PMEM(SP)←PC ; SP←SP-1; PC← Rs1 }
JMPD desloc	E	Deslocamento			if (*flag* =1) PC ← PC + ext_sinal & desloc
JSRD desloc	F	Deslocamento			if (*flag* =1) { PMEM(SP)←PC ; SP←SP-1; PC← PC + ext_sinal & desloc }

Figura 41 - Especificação das instruções da R7.

Convenções utilizadas:

RtH: oito *bits* mais significativos de Rt

RtL: oito *bits* menos significativos de Rt

&: concatenação de vetores de *bits*

←: atribuição de valor a registrador ou posição de memória

PMEM(x): conteúdo de posição de memória cujo endereço é x

Rt = R*target* [destino], Rs1 = R*source*1, Rs2 = R*source*2.

4.4 Definição do Mapa de Memória Utilizado

Nesta Seção são definidas as áreas de memória utilizadas e a finalidade de cada uma delas, como pode ser visto na Figura 42. O total da memória existente para uso do processador R7 é de 64 Kpalavras ou 128 K*bytes* (endereçamento de 16 *bits*).

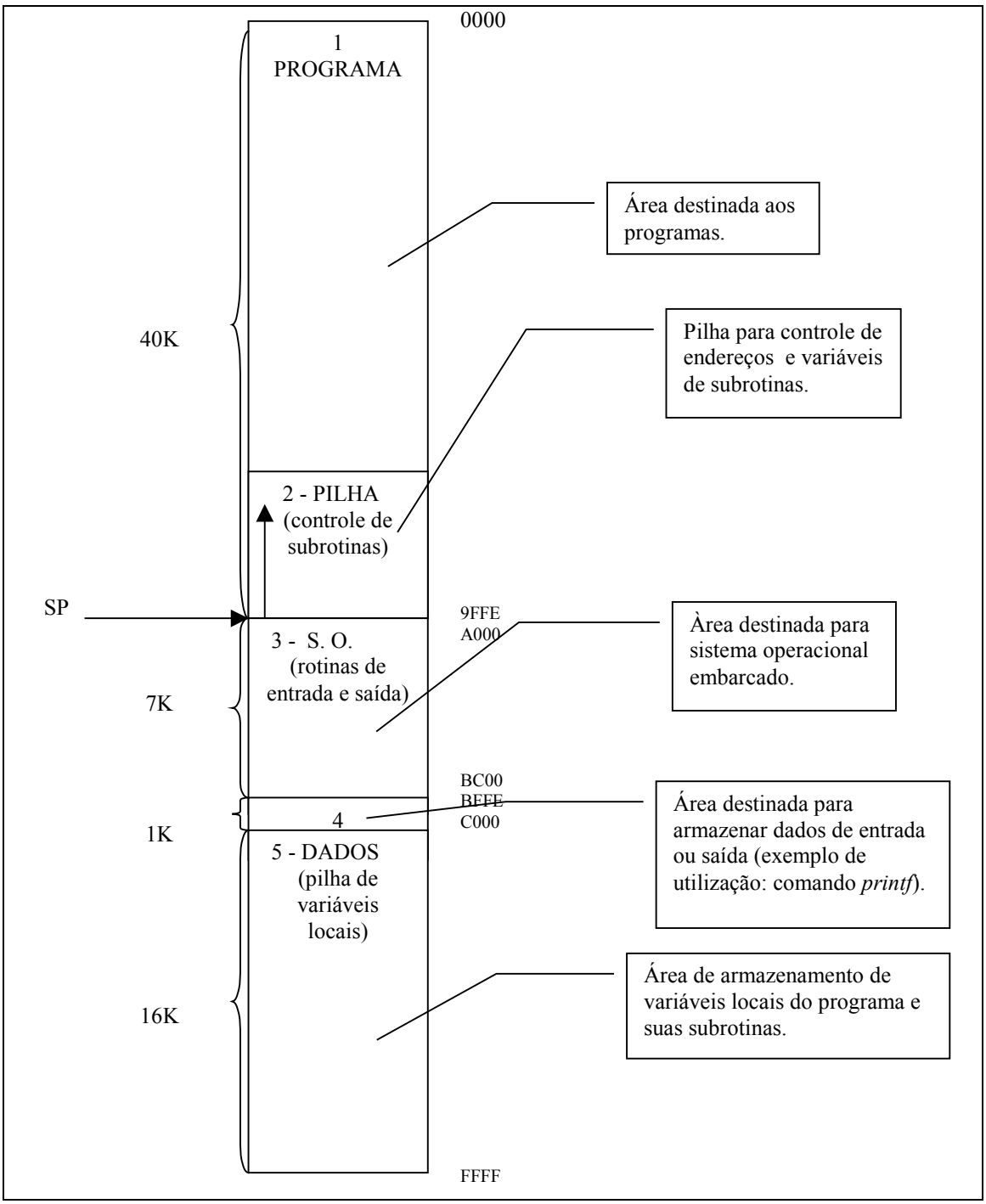

Figura 42 – Definição do uso das áreas de memória.

A memória foi particionada nas seguintes áreas:

a) área destinada a acomodar as rotinas de sistema operacional embarcado, fora do escopo deste

trabalho, que tratam entrada e saída de dados (7K) (Exemplo: *printf*, *malloc, scanf*, etc...) ;

b) área de pilha para controle de endereços utilizados nas subrotinas de programas;

c) área de programas, onde se pode ter várias rotinas/funções/procedimentos armazenados (40 K, compartilhado com a área de pilha para controle de subrotinas);

d) área para armazenamento de dados de entrada e saída, que são parâmetros das rotinas do Sistema Operacional (1K);

e) pilha para armazenamento das variáveis locais dos programas (16K).

O uso de duas pilhas distintas, dados e subrotinas, deve-se principalmente ao fato de utilizarmos como modelo de gramática o processador MIPS. Este processador não tem o registrador SP, utilizando um registrador do banco de registradores para este propósito. Outra razão para utilizarmos 2 pilhas é a ausência de operações no *assembly* do processador R7 que permita operações de salvamento/recuperação na pilha com deslocamento.

Nos programas em *assembly* do processador R7 existe um padrão de instruções que são inseridas no início de cada programa com o propósito de:

a) inicializar o registrador R0 com conteúdo zero;

b) inicializar o ponteiro para o topo da pilha de controle de subrotinas (SP);

c) salto para a rotina principal do programa.

Um exemplo dessa seqüência padrão é mostrado na Figura 43.

```
LDL R1,#feH.....Inicialização da área de pilha para subrotinas
LDH R1,#9fH
LDSP R1
XOR R0,R0,R0....Estabelecendo o valor zero para R0
EQL R0,R0.......Setando flag para salto
JMPD main.......Salto para ínicio do programa
```

Figura 43 - Seqüência inicial padrão para R7.

4.5 Código Assembly Teste para a Arquitetura R7

Para a realização de testes visando demonstrar resultados das atividades elencadas na proposta do trabalho, o grupo implementou algumas instruções básicas do *assembly* da arquitetura R7. Na Figura 44 mostramos um exemplo simples, em linguagem C, para o qual extraímos o *assembly*. O Capítulo 6 dedica-se a validar os programas gerados pelo compilador LCC.

```
int main()
{
   int i,a;
   i = 0;
   a = 0;
   i = i + 3;
   a = i - 2;
 }
```

Figura 44 - Programa exemplo.

Cabe ressaltar que no exemplo o grupo utilizou operações de adição, subtração e atribuição de variáveis do tipo inteiro, juntamente com a declaração desses tipos. Tendo em vista a complexidade na parametrização optamos por exemplos simples que serão posteriormente incrementados (ver Capítulo 6) com outras instruções que suportem comandos tais como *for, while, switch, if,* etc.

Como resultado foi obtido o *assembly* mostrado na Figura 45 no qual foram implementadas as instruções ADDI e SUBI (em modo imediato curto). Observar que a variável 'i', utilizada em operações aritméticas, foi alocada em registrador. A variável 'a' não é utilizada em operações aritméticas, sendo por esta razão armazenada na pilha de dados.

```
.CODE
LDL R1,#feH
LDH R1,#9fH
LDSP R1
XOR R0,R0,R0
EQL R0,R0
JMPD main.....................Cabeçalho comum a todos os programas

main:
ADD R9,R0,R0..................Carga de zero em R9
ADD R15,R0,R9.................R15 ← R9 ('i' no registrador R15)
LDL R8, #fcH .................Busca endereço de 'a'
LDH R8, #ffH
ST R9,R0,R8...................a ← 0 (grava 'a' na pilha de dados)
ADDI R15,#3...................i + 3
LDL R9, #fcH..................Busca endereço de 'a'
LDH R9, #ffH
ADD R8,R0,R15.................R8 ← R15
SUBI R8,#2....................i - 2
ST R8,R0,R9...................a ← i - 2 (grava valor de 'a' na pilha de dados)
L1:
HALT..........................Fim de programa
.ENDCODE
```

Figura 45 - Código *assembly* gerado para a arquitetura R7.

Cabe ressaltar que a base para inserção de algumas instruções do processador R7 foram as definições do processador MIPS. Para geração e visualização do código *assembly* basta utilizar o comando "*lcc –S –Wf-target=gaph <arquivo.c>*", que o LCC gera um arquivo do tipo <*arquivo.s*>, onde são armazenadas as linhas de código.

No próximo Capítulo será mostrado como foi realizada a configuração do processador R7 para o compilador LCC.

5. CONFIGURAÇÃO DO LCC

Este Capítulo tem por objetivo mostrar a forma como foi realizada a elaboração da descrição do processador-alvo R7 para o compilador LCC. É mostrado como foram especificadas as características daquele processador para o compilador e como essas características foram inseridas no arquivo de configuração *md*.

Para a configuração do processador-alvo neste trabalho foi utilizada, visando agilizar o processo, uma descrição de processador já existente no compilador LCC, a descrição do processador MIPS. Essa base foi escolhida devido às semelhanças existentes deste processador com o processador-alvo R7. Os dois processadores são arquiteturas *load-store* com instruções de formato único, modos de endereçamento e política de utilização dos registradores semelhantes.

A configuração foi realizada através da edição do arquivo *md* com alteração de algumas partes existentes neste arquivo (ver exemplo de estrutura mostrado na Figura 28 na Seção 3.3.2), de modo que estes trechos representassem as características necessárias para a geração do gerador de código para o processador-alvo R7.

Nas Seções 5.1 a 5.5 a seguir são enumerados alguns tópicos de informações básicas que são necessárias para a configuração de qualquer processador-alvo escolhido para o compilador LCC. Esses tópicos foram enumerados visando facilitar a descrição de outros processadores que tenham o mesmo nível de complexidade daquele especificado neste projeto.

Já nas Seções seguintes são mostradas os métodos e as alterações realizadas para descrever as características do processador R7 para o LCC.

5.1 Escolhendo um Arquivo de Descrição Modelo

O arquivo *md* é responsável por descrever como o compilador deve gerar o código em linguagem *assembly* para um processador-alvo, a partir dos *DAGS*. Este arquivo é bastante complexo e possui um grande volume de informações, tais como: códigos de funções em linguagem C, *templates* de geração de código *assembly*, entre outros detalhes (ver Seção 3.3.2).

Devido à complexidade e o grande número de informações que este arquivo possui, o mais natural a se fazer é partir de um modelo, isto é, de um arquivo de descrição de máquina já existente, alterando-o conforme a necessidade.

Mas que arquivos devem ser tomados como modelos para a construção de arquivos de descrição para estes novos processadores-alvo ? Que critérios são relevantes para a escolha de um destes arquivos ?

O arquivo *md* a ser escolhido como modelo deve conter a descrição de um processador que possua características semelhantes às do processador que se quer inserir como novo alvo do compilador. Normalmente, processadores com características arquiteturais em comum, tendem a possuir código *assembly* semelhante, e conseqüentemente não são necessárias muitas modificações no arquivo de descrição, o que facilita a construção do novo *md*.

A seguir são abordados pontos que são relevantes no processo de descrição de um novo processador. Além disso, serão esclarecidos alguns aspectos sobre o arquivo *md* que são de uso geral na elaboração daquele processo.

5.1.1 Diretivas para a Escolha do Arquivo de Modelo

É importante que fique claro que o conhecimento sobre as características do processador-alvo a ser descrito é um fator essencial para a escolha do arquivo modelo de descrição de máquina. Mas além disso, pode-se também procurar modelos que possuam convenções de geração de código que se assemelhem às requeridas pelo novo processador-alvo.

A seguir, listamos algumas das características importantes na escolha de um arquivo de modelo para a construção de um novo arquivo de descrição de máquina.

Dimensão do Conjunto de Instruções (Processadores RISC ou CISC [2]). Esta característica deve ser a mais relevante na hora da escolha de um arquivo de modelo. Dificilmente um processador RISC possuirá características que sejam compatíveis com uma máquina CISC. Se o processador para o qual se deseja gerar código é RISC, o modelo deve ser de uma máquina do mesmo tipo. O mesmo também vale para os processadores CISC. As máquinas CISC possuem instruções mais complexas, que realizam muitas operações em uma única instrução. As máquinas RISC possuem instruções simples, como operações lógico/aritméticas ou acesso à memória. Estas características interferem diretamente sobre a estrutura dos *templates* de *assembly* utilizados no arquivo de descrição, pois instruções com estrutura complexa, certamente ocasionam *templates* complexos.

Número de Registradores e Política de Uso destes Registradores. Por exemplo, em geradores de código para determinados processadores, alguns registradores podem ser reservados para passagem de parâmetros para subrotinas (MIPS), enquanto outros não reservam registrador algum para este fim, utilizando a memória como meio para passar os parâmetros às subrotinas (X86). O número de registradores disponíveis no processador também deve ser considerado. Uma vez que, este é um recurso que deve ser bem utilizado no que diz respeito a performance do código gerado. Em arquiteturas que possuem um grande número de registradores, deve-se evitar ao máximo os acessos à memória e procurar trabalhar com os dados em registradores, pois o acesso aos registradores é mais rápido do que o acesso à memória. Desta forma, se o novo processador alvo possuir um grande número de registradores, o arquivo modelo deve também descrever uma máquina assim. Isto facilita o processo de edição das rotinas responsáveis pela alocação de registradores, uma vez que, devido a semelhança das arquiteturas, estas rotinas também podem ser semelhantes.

Operandos das Instruções. É desejável que o conjunto de instruções dos processadores modelo e alvo sejam o mais parecido possível. Esta semelhança não está ligada à grafia da instrução, mas sim ao seu significado. Aconselha-se utilizar modelos que possuam instruções com o mesmo número de operandos e que estes operandos tenham o mesmo significado. Por exemplo, caso haja um processador *X* com uma instrução no formato ADD op1, op2, op3 que expressa que op1= op2 + op3, e no processador alvo tenhamos a mesma instrução ADD com uma semântica um pouco diferente, como op3 = op1+ op2, este processador *X* pode ser um possível modelo para a construção do arquivo *md* do novo alvo, uma vez que os operandos da instrução em questão tem significados bem semelhantes. Geralmente, a semelhança estende-se a outras instruções, não limitando-se a apenas uma, pois a construção de uma dada instrução reflete o modelo da arquitetura, por exemplo, *load-store*. Neste modelo não existem instruções que executem operações sobre operandos que estejam diretamente na memória. Os dados devem ser primeiramente carregados em registradores (*load*), processados e então armazenados (*store*) em memória, se necessário.

Instruções Disponíveis. Para esclarecer este critério pode-se usar como exemplo as instruções de multiplicação e divisão, que não são implementadas em *hardware* por alguns processadores. Se o novo processador alvo não implementar este tipo de instruções, é interessante procurar um modelo que funcione da mesma forma. O que mais uma vez facilita a construção do arquivo de descrição do novo alvo.

As informações sobre o conjunto de instruções devem estar bem definidas previamente para evitar erro na descrição. Deve-se ter clara a função da instrução, quais os registradores que serão afetados, e possíveis *flags* de controle para operações lógico-artiméticas. Um exemplo de definição para um conjunto de instruções é mostrado na Figura 41 do Capítulo 4.

Pode ser que não haja um arquivo de descrição com todas as características que se deseja. Neste caso a alternativa pode ser escolher mais de um arquivo modelo, de forma a completar o conjunto de características desejáveis. Porém, deve-se ter em mente de que esta alternativa pode custar mais caro, no sentido de que pode ser mais difícil adaptar os dois arquivos para a descrição de um processador-alvo. É o mesmo que possuir dois programas diferentes, ter um problema que é solucionado parte por um dos programas e parte pelo outro, e tentar construir um programa só que resolva o problema por inteiro, a partir dos códigos fonte dos dois programas. Certamente ocorrerão problemas de integração. O mesmo ocorre com os arquivos de descrição.

Um exemplo de problema de integração é a existência de algumas rotinas do *back end* que necessitam fazer uso de variáveis globais do LCC. Deve-se ter o cuidado de verificar se funções de arquivos de descrição diferentes não alteram estas variáveis de forma contraditória, comprometendo o processo de compilação.

5.2 Estruturas de Dados para Descrição de Registradores

Nesta Seção são mostradas as estruturas de dados utilizadas para descrição das características do conjunto de registradores para um novo processador-alvo.

Uma atividade importante na configuração do compilador é a definição do número de registradores bem como a função de cada um. Cabe ao usuário definir a que conjunto pertence cada registrador (inteiros ou ponto flutuante). Além disso, este também deve definir se algum registrador será reservado para um fim específico, como por exemplo ser utilizado como ponteiro para a área de memória onde ficam as variáveis locais das rotinas.

O estado de utilização (livre ou ocupado) dos registradores também deve ser controlado. Para definir como que estes estados devem ser interpretados pelo compilador, também é necessária a intervenção do usuário. Para efetuar esta atividade, é preciso conhecer com mais detalhes, algumas variáveis, constantes, funções e estruturas de dados da *interface* de geração de código, saber o que significam e quais suas funções.

Cada registrador possui algumas informações que devem ser armazenadas, tais como: nome, número de identificação, conjunto (*register set*) ao qual pertence, entre outras. Para este fim, o compilador possui uma estrutura de dados, declarada em um de seus arquivos fonte (ver Figura 26), *config.h*, conforme mostrado na Figura 46.

```
typedef struct{
        Symbol vbl;
        int set;
        short number;
        unsigned mask;
} *Regnode;
```

Figura 46 - Estrutura de armazenamento das informações sobre características de um registrador [31].

Se o registrador foi alocado para armazenar uma variável, o campo *vbl* aponta para a sua representação na tabela de símbolos, a fim de que seja possível ao *back end* saber a que variável corresponde (se isto for necessário).

O campo *set* armazena o número que identifica o conjunto de registradores ao qual pertence. O tamanho deste campo, de acordo com seu tipo de dado, proporciona a existência de vários conjuntos de registradores, mas apesar disso, todos os processadores-alvo do LCC utilizam apenas dois conjuntos. O campo *number* leva um número que identifica o registrador. O campo *mask*, do tipo *unsigned* que possui 32 *bits*, é interpretado como uma máscara que possui o valor 1 nos *bits* correspondentes aos registradores utilizados.

Na maioria dos processadores, registradores simples (*single-width registers* [31]) ocupam um único *bit* desta máscara, e registradores duplos (*double-width registers* [31]) ocupam dois. Por exemplo, se o valor do campo *mask* for 1, significa que esta máscara identifica o

registrador simples 0, uma vez que o *bit* menos significativo deste valor (o *bit* mais à direita na representação binária) tem o valor 1. Outro exemplo seria, *mask* ser igual a 6, o que significa que a máscara representa um registrador duplo que utiliza os registradores 1 e 2, uma vez que os *bits* das posições 1 e 2 tem o valor 1. Mais a frente voltaremos a falar sobre máscaras, deixando mais clara a utilidade deste campo.

5.2.1 Constantes de Identificação de Conjunto de Registradores

São utilizados vetores que separam as informações dos diferentes conjuntos de registradores. Por exemplo, existem máscaras que representam o estado de utilização dos conjuntos de registradores. Para distinguir as informações dos diferentes conjuntos, estas máscaras são armazenadas em posições diferentes destes vetores, uma para cada conjunto. Sendo assim é necessário saber a que conjunto se refere cada uma das posições destes vetores.

Existem duas constantes definidas no arquivo *config.h*, as quais são utilizadas como índices destes vetores, conforme o conjunto de registradores que representam. As constantes IREG (*Integer Registers*) e FREG (*Floating Point Registers*) distinguem registradores de inteiros de registradores de ponto flutuante, conforme a Figura 47.

```
enum { IREG = 0, FREG = 1};
```

Figura 47 - Constantes de identificação de conjuntos de registradores [31].

A utilização destas constantes facilita a leitura do código em linguagem C, utilizado no arquivo de descrição de máquina nas funções dependentes de processador alvo. Podemos ver que a constante IREG indexa a primeira posição dos vetores, reservada para informações sobre registradores inteiros. A constante FREG é utilizada para indexar a segunda posição do vetor que armazena informações sobre registradores de ponto flutuante.

5.2.2 Controlando o Estado dos Registradores

Algumas máscaras auxiliam a manter o controle do estado de utilização dos registradores durante a compilação de uma rotina. Para isto são utilizadas algumas variáveis do tipo *unsigned* onde cada bit leva informação a respeito de um registrador. O LCC, permite que estas informações sejam mantidas para dois conjuntos de registradores (registradores inteiros e de ponto flutuante). As informações de estado destes conjuntos de registradores são armazenadas em vetores de duas posições, separando os dados dos diferentes grupos.

Um destes vetores é *freemask*, que diz quais registradores estão livres. Analogamente, o vetor *usedmask* mantém informações sobre quais registradores estão sendo utilizados, durante a compilação de uma rotina. Além disso, *usedmask* indica quais registradores devem ser salvos no começo de um procedimento e restaurados no final. Podemos ver a declaração destas variáveis na Figura 48.

```
unsigned freemask[2];
unsigned usedmask[2];
```

Figura 48 – Vetores de controle da disponibilidade dos registradores [31].

Outros dois vetores também são utilizados como máscaras para o conjunto de registradores, *vmask* e *tmask*. O vetor *vmask* indica quais registradores podem ser alocados para variáveis. O vetor *tmask* indica quais registradores podem ser alocados para armazenar valores temporários (ver Figura 49) . Juntamente com *freemask* e *usedmask*, estas variáveis também estão declaradas no arquivo *gen.c* (ver Seção3.2).

```
unsigned tmask[2];
unsigned vmask[2];
```

Figura 49 – Vetores de controle sobre uso dos registradores [31].

Na Figura 50 pode-se ver algumas definições de máscaras utilizadas no arquivo de descrição de máquina do processador *MIPS R3000* [31]. Estas linhas de código C pertencem a rotina de interface *progbeg(),* que entre outras funções é responsável pela inicialização de estruturas de estado de registradores. Pode-se ver que a posição IREG do vetor *tmask* é inicializada com o valor 0300ff00 em hexadecimal. Isto indica que dentre os 32 registradores inteiros que possui, os registradores entre o 8 e 15 (inclusive), e os registradores 24 e 25 (inclusive) serão utilizados para armazenamento de valores temporários. O mesmo vetor na posição FREG é inicializado com o valor 000f0ff0 indicando que os registradores entre o 4 e 11 (inclusive), e os registradores entre o 16 e 19 (inclusive), serão utilizados para armazenar valores temporários em ponto flutuante. Da mesma forma o vetor *vmask* é inicializado de maneira análoga, definindo quais vetores podem ser utilizados para armazenar valores de variáveis .

```
tmask[IREG] = 0x0300ff00; tmask[FREG] = 0x000f0ff0;
vmask[IREG] = 0x40ff0000; vmask[FREG] = 0xfff00000;
```

Figura 50 - Exemplo de uso das máscaras

5.3 Tipos

Na descrição de um processador para o compilador deve-se levar em conta os tipos que são implementados pelo processador que está sendo descrito. Essa informação é essencial tendo em vista o cálculo de endereços no tratamento de ponteiros, vetores, matrizes e outros.

No LCC essa característica é descrita na estrutura mostrada na Figura 37, da Seção 3.3.3. Nesta estrutura cada tipo possui um tamanho (*size*), um alinhamento (*align*) e um *flag* (*outofline*) que controla a exibição das constantes daquele tipo em *DAGs* (ver Seção 3.3.3).

5.4 Definição do Uso da Memória

As áreas de memória e seu uso devem ser definidos levando-se em conta os aspectos de espaço para aplicações, variáveis, armazenamento de endereços de retorno de subrotinas e parâmetros de subrotinas. A definição da utilização dos recursos de memória depende diretamente da arquitetura do processador a ser descrito.

Deve-se ter claro também o tamanho da palavra de memória manipulada pelo processador que se está descrevendo. Esse item é essencial quando é definida a granularidade do acesso aos dados da memória, isto é, a existência da possibilidade de acesso parcial de cada célula da memória. Essa informação é necessária pois existem processadores, por exemplo, de 16 *bits* que permitem endereçamento de memória de 16 *bits* e não realizam um acesso parcial de 8 *bits* (*word-adressed machines* [31]).

O endereçamento da memória deve estar claro pois em alguns casos deve-se, no início de cada programa em *assembly*, setar alguns registradores com endereços específicos de memória. Um exemplo seria carregar um registrador utilizado como ponteiro para a área de pilha das variáveis com o endereço do topo da pilha, visando a manipulação de variáveis locais.

5.5 Outras Informações Importantes

5.5.1 Detalhes Importantes do Arquivo de Descrição

Nesta Seção são abordados alguns detalhes do arquivo de descrição de máquina, basicamente operadores utilizados nos *templates* de *assembly*. Estes operadores são necessários para que seja possível que as instruções sejam geradas por partes, isto é, uma instrução é o resultado do agrupamento de vários *templates* em forma de árvore.

Pode-se ver na Figura 21, no Capítulo 3, um exemplo deste agrupamento. As linhas pontilhadas indicam que os *templates* apontados por estas fazem parte de outras instruções. Já as linhas sólidas indicam instruções explícitas que serão impressas diretamente no código *assembly*. A seguir serão enumerados os padrões de controle utilizados na formação das instruções, tendo em vista uma melhor compreensão do significado de cada um deles:

- '**\n**' – Quando se quer que uma determinada instrução seja explicitamente impressa no código de saída, a *string* de *template* desta instrução deve ser terminada por '\n', caso contrário o compilador interpretará que este *template* faz parte de outra instrução.
- '**%dígito**' – O símbolo '%' seguido de um dígito, que pode ser 0 ou 1, indica que no lugar destes, deve ser impresso uma parte de instrução que corresponde ao filho identificado pelo número. Mais detalhes sobre nodos filhos ver Seções 3.1 e 3.3.2 .

- '**%letra**' – O símbolo '%' seguido da letra 'a' indica elementos provenientes do arquivo fonte. Estes podem ser constantes numéricas ou identificadores. Existem ainda outras opções. Para *DAGs* que necessitam de um registrador para armazenar o resultado da instrução que representam, utiliza-se a letra 'c'. Nestes casos este '%c' representa o número do registrador alocado para armazenamento do resultado da operação.
- '**?**' – Nos casos em que o *template* possui mais de uma instrução que utiliza o padrão '%c', presença do símbolo '?' sinaliza que a primeira instrução deve ser omitida no caso de ocorrer a seleção do mesmo registrador para as duas partes do *template*. O exemplo da Figura 51 mostra uma parte do arquivo de descrição de máquina para o gerador de código da arquitetura X86, mais especificamente de alguns *templates* para instruções aritméticas. A instrução que compõe a primeira parte de cada um dos *templates* armazena seu resultado em um registrador representado por '%c'. A instrução na segunda parte também faz o mesmo. Se em ambas as partes '%c' indicar o mesmo registrador, a primeira é omitida quando da geração de código.

```
reg: ADDI(reg, mrc1) "?mov %c, %0\nadd %c, %1\n"    1
reg: ADDP(reg, mrc1) "?mov %c, %0\nadd %c, %1\n"    1
reg: ADDU(reg, mrc1) "?mov %c, %0\nadd %c, %1\n"    1
reg: SUBI(reg, mrc1) "?mov %c, %0\nsub %c, %1\n"    1
reg: SUBP(reg, mrc1) "?mov %c, %0\nsub %c, %1\n"    1
reg: SUBU(reg, mrc1) "?mov %c, %0\nsub %c, %1\n"    1
```

Figura 51 - Exemplo de utilização do símbolo '?'.

- '#' – Algumas vezes não é possível expressar por inteiro em um *template* todos os detalhes de uma instrução de um determinado processador. Nestes casos a função de *interface emit2()* deve ser invocada. Quando isso é necessário basta colocar o símbolo '#' como primeiro caracter do *template*. Desta forma o gerador de código será notificado de que a função *emit2()* deve ser chamada para gerar código específico para aquele *template.*

5.5.2 Funções de Interface

As funções de interface, bem como outros dados da interface de geração de código, disponibilizam uma comunicação entre os dois módulos do compilador (*back end* e *front end*). As funções do *back end* são dependentes de processador alvo. Estas devem ser implementadas para cada novo alvo do compilador. Já as funções do *front end* foram projetadas para efetuar atividades independentes do processador alvo.

Nesta Seção são apresentadas características de algumas funções de interface importantes na geração de código. Os protótipos destas funções encontram-se na Figura 38.

Durante a inicialização o *front end* chama a função *progbeg* do *back end.* Esta processa alguns argumentos (*argv[0..argc-1]*) que não são reconhecidos pelo *front end*, e que devem ser tratadas por cada *back end.* Além disso, *progbeg* ainda inicializa alguns dados no *back end*.

No final da compilação o *front end* chama *progend*, que da a oportunidade ao *back end* de finalizar sua saída. Para os geradores de código de alguns processadores-alvo, esta função não

necessita executar comando algum.

Quando é necessário definir uma variável global a função *global* é chamada. Após esta função devem ser chamadas as funções de inicialização de dados, conforme o tipo da variável, uma vez que *global* define o *label* para a variável global.

A função *defconst* é chamada quando é necessária a definição de constantes de qualquer tipo de dado. O parâmetro *ty* define o tipo da variável e o "*v*" traz o valor da mesma. Por ser do tipo *Value*, a variável "*v*" possui alguns campos que trazem o valor da constante conforme seu tipo de dado - Figura 52.

ty	Campo	Tipo de dado
C	v.uc	*character*
S	v.us	*short*
I	v.i	*int*
U	v.u	*unsigned*
P	v.p	ponteiro
F	v.f	*float*
D	v.d	*double*

Figura 52 - Tipos de constante [31].

A função *defstring* emite código para a inicialização de *strings*. O tamanho destas é passado pelo parâmetro *len*. Esta função é chamada pelo *back end* para definir qualquer constante *string* que apareça nmo código fonte C. Para a maior parte dos processadores alvo estas *strings* ficam todas agrupadas em um só lugar identificadas por *labels*.

As funções mais importantes no processo de geração de código são *function* (pertencente ao back end), *emitcode* e *gencode* (pertencente *front end*). A função *gencode* faz o caminhamento dos *DAGs* selecionando as instruções que devem ser emitidas no código final. A função *emitcode* percorre os *DAGs* novamente emitindo o código selecionado pela função *gencode.*

A função dependente de processador alvo *function* chama as funções *gencode* e *emitcode* como é mostrado na Figura 53. Desta forma pode-se fazer inicializações necessárias antes de cada emissão de código para funções e emitir instruções padrão ao final de cada função.

```
void function (Symbol f, Symbol caller[], Symbol callee[], int ncalls){
        <inicializações>
        gencode(caller, callee);
        <instruções antes de cada função>
        emitcode();
        <instruções ao final da função>
}
```

Figura 53 - Estrutura da função de interface *function.*

5.6 Métodos e Ferramentas de Análise

Nesta Seção são enumeradas algumas técnicas utilizadas no decorrer da execução do projeto, e que podem ser úteis no momento de inserir uma nova descrição para outro processador-alvo. Ressalta-se que toda a descrição a partir de um arquivo *md* modelo exige uma análise da

gramática a ser modificada para a adaptação às novas regras do processador que se está descrevendo.

5.6.1 Entendendo a Seqüência de Execução das Regras

Para a compreensão e implementação das instruções do processador R7 um dos métodos utilizados no projeto foi o de numerar todas as produções da gramática que foi escolhida como modelo base (MIPS). Através dessa numeração e de testes com programas simples em linguagem C foram sendo alteradas algumas dessas produções.

A cada geração de *assembly* os números das regras foram armazenados e uma análise do arquivo *md* era efetuada visando definir quais regras eram utilizadas. Com essas informações as regras eram alteradas para a adaptação ao *assembly* do processador R7 e a nova descrição era incorporada ao compilador mediante a recompilação do núcleo por completo.

Esses números foram inseridos nos *templates* do *assembly* tal como visto na Figura 54. No exemplo estão numeradas as regras de conversão de *int* para *int* de 1 *byte*, *int* para *int* de 4 *bytes*, *unsigned* para *int* de 4 *bytes* e *unsigned* para *unsigned* de 4 *bytes*.

```
reg: CVII1(reg)  "ADD R%c,R0,R%0 [55]\n"  1
reg: CVII4(reg)  "ADD R%c,R0,R%0 [56]\n"  1
reg: CVUI4(reg)  "ADD R%c,R0,R%0 [57]\n"  1
reg: CVUU4(reg)  "ADD R%c,R0,R%0 [58]\n"  1
```

Figura 54 - Exemplo de numeração das regras de conversão.

A cada seqüência de números analisada tinha-se ainda o auxílio do arquivo gerado pela opção SYMBOLIC, ferramenta do próprio LCC explicada na próxima Seção.

5.6.2 Utilização da Opção *symbolic* do LCC

Uma ferramenta de grande valia no processo de elaboração do arquivo de descrição de arquitetura está disponível no próprio compilador. Através do comando "*lcc –Wftarget=symbolic arquivo.c*" é produzida uma representação da estrutura do código intermediário utilizado pelo compilador durante o processo de geração de código. Este arquivo possui o mesmo nome do arquivo fonte e se diferencia pela extensão '*.s*'. Vale lembrar que o nome do arquivo pode ser alterado mediante a opção '*-o*'.

A estrutura produzida neste arquivo é a seqüência de *DAGs* (ver Seção 3.2) que representa o código intermediário para o programa fonte que se quer analisar. Comparando o arquivo de saída produzido pelo compilador com a opção *symbolic* com os operadores *DAG* presentes no arquivo *md*, tem-se o caminho percorrido na geração do código *assembly* final. Essa análise do arquivo gerado nesta opção é essencial para a representação das instruções do processador-alvo que se está descrevendo.

A opção *symbolic* auxilia na construção da denominada "decoração" das regras da gramática, onde são inseridos os *templates* do *assembly* e na definição dos parâmetros a serem

inseridos após o símbolo ‘%’.

A representação dos nodos no arquivo gerado pela opção *symbolic* possui algumas convenções que serão mostradas tendo como exemplo o código da Figura 55.

```
void main()

{
  int a, b, c;

  a = 10;
  b = 20;

  c = a + b;

}
```

Figura 55 - Código fonte para geração da opção *symbolic*.

```
export main
segment text
function main type=void function(void) sclass=auto scope=GLOBAL ref=0.000000 ncalls=0
blockbeg off=0
local a type=int sclass=auto scope=LOCAL offset=0 ref=2.000000
local b type=int sclass=auto scope=LOCAL offset=4 ref=2.000000
local c type=int sclass=auto scope=LOCAL offset=8 ref=1.000000
blockend off=12
 2. ADDRLP4 count=2 a
3. CNSTI4 10
1' ASGNI4 #2 #3 4 4
5. ADDRLP4 count=2 b
6. CNSTI4 20
4' ASGNI4 #5 #6 4 4
8. ADDRLP4 c
10. INDIRI4 #2
11. INDIRI4 #5
9. ADDI4 #10 #11
7' ASGNI4 #8 #9 4 4
 1:
maxoff=12
progend
```

Figura 56 - Código gerado para o exemplo da Figura 55.

Verificando os operadores utilizados neste arquivo, nota-se que são iguais aos operadores utilizados Os nodos precedidos do símbolo ‘#’ são as folhas e aqueles precedidos pelo símbolo ’`’ são as respectivas raízes dos *DAGs*, como pode ser visto na Figura 56.

Para ilustrar a utilidade destas estruturas na compreensão do processo de geração de código, pode-se tomar como exemplo o primeiro comando executável do programa fonte, que é uma atribuição simples de uma constante a um endereço de memória (uma variável).

Este comando ocasiona a geração de outras 3 instruções de “código intermediário” mostradas na Figura 57. Neste fragmento de código intermediário, pode-se notar que a instrução identificada pelo operador *DAG* (ver Seção 3.3.4) ASGNI4, significa a atribuição de uma constante a um endereço de memória (uma variável). A instrução identificada por CNSTI4 indica o processamento da constante 10. Já a instrução ADDRLP4 indica o cálculo do endereço de memória da variável ‘a’. Pode-se notar que esta última instrução depende das duas anteriores. Isto é explícito através dos 2 números precedidos pelo símbolo ‘#’.

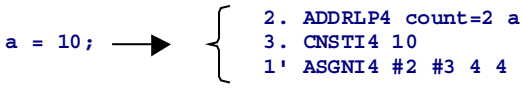

Figura 57 - Tradução de comandos para código intermediário.

Analisando desta forma este "código intermediário" produzido pelo compilador, pode-se facilmente saber quais operadores *DAG* devem ser utilizados no arquivo de descrição e ainda como devem ser os *templates* utilizados na decoração destes.

Na próxima Seção serão descritos os itens de configuração utilizados para descrever as características do processador R7 para o compilador LCC.

5.7 Descrição do Processador R7

Nas Seções a seguir são mostradas as alterações efetuadas no arquivo *md* para o caso específico do processador R7.

5.7.1 Convenções Básicas

Como uma das definições deste projeto é a de disponibilizar código com operações apenas para dados numéricos inteiros para o processador R7, foram definidos os tipos *char* e *unsigned char* para este fim. Esse fato decorre da existência de algumas restrições do LCC quanto a tipos de dados.

Por ser projetado para gerar código para processadores de 32 *bits*, este compilador não permite que o tipo *int*, seja configurado com tamanho menor que 4 *bytes*. Isto torna inviável o uso deste tipo pelo processador R7 como tipo padrão para armazenamento de dados e valores numéricos inteiros. Isto ocorre porque este processador possui endereçamento por palavra. Desta forma, cada dado deve ocupar uma palavra inteira de memória. Para isto foi necessário escolher um tipo de dado que não ocupasse mais de uma palavra de memória a cada definição de variável.

O LCC não foi projetado para processadores que endereçam sua memória por palavras (*word addressed machines*) [31], desta forma não é possível configurar o compilador para este tipo de máquina. Sendo assim, convencionamos o tipo *char* como tipo padrão devido a adequação necessária para minimizar os problemas de desperdício de memória. Isto evita que o endereçamento ocorra de 4 em 4 palavras. Para solucionar este problema convencionou-se que o compilador ao interpretar uma variável como *byte*, estará de fato gerando código para uma palavra do processador R7.

Nos códigos em linguagem C as variáveis são apenas do tipo *char* implicando em

alocação de apenas uma célula de memória para cada variável. Essa célula de memória se refere a 16 *bits*, apesar do tipo *char* em C significar apenas 8 *bits*. Isso ocorre devido às restrições anteriormente comentadas.

Vale ressaltar que a convenção do tipo *char* como padrão não impede o compilador de gerar código para variáveis do tipo *int*. O problema na geração de variáveis do tipo inteiro está no fato de que ao armazenar espaço para essas variáveis o compilador irá direcionar os endereços de maneira que sejam ocupados de 4 em 4 palavras. Isso implica em fragmentação de memória no espaço destinado para variáveis do tipo inteiro.

Para a definição do conjunto de registradores e a função de cada registrador foi seguida a convenção mostrada na Figura 58.

```
R0 ------------------------ constante 0
R1 .. R2 ------------------- retorno de subrotina
R3 .. R5 ------------------- argumentos
R6 .. R10 ----------------- registradores de uso geral
R11.. R15 ----------------- registradores para variaveis 'register'
```

Figura 58 - Definição e função do conjunto de registradores.

As linhas mostradas na Figura 59 convencionam as máscaras utilizadas para definir a função de cada registrador do conjunto de registradores.

```
#define INTVAR  0xF800 /* 11-15 : alocados para variáveis */
#define INTTMP  0x07C0 /* 6 - 10  : regs. de uso geral   */
#define INTRET  0x0006 /* 1 – 2 : regs. ret. subrotina */
```

Figura 59 – Máscaras de definição da função dos registradores.

As máscaras utilizam o prefixo *INT* pois é uma característica herdada do processador MIPS. Esse prefixo significa que essas máscaras se referem a registradores pertencente ao conjunto de registradores inteiros. Vale lembrar que no processador R7 temos disponível apenas um tipo de registrador.

A máscara *INTRET* indica quais registradores tem a função de armazenar o retorno de subrotinas. No caso do processador R7 foi definido o R1 e R2.

O endereçamento das variáveis locais é feito adicionando o deslocamento da mesma com o conteúdo de R0, ou seja, valores a partir de FFFF (-1), decrescentes (ver Figura 42).

Como registrador constante foi utilizado o registrador R0 que a exemplo do processador MIPS indica sempre a constante zero. Os registradores R4 e R5 foram destinados a passagem de argumentos em funções.

5.7.2 Descrição das Instruções

Nas produções existentes no arquivo *md* são utilizados os símbolos terminais e não-terminais já existentes no modelo escolhido como base para a nova descrição (MIPS). Para dirimir

dúvidas na Figura 60 são mostrados os significados associados a cada símbolo não-terminal, sendo que os detalhes sobre símbolos terminais foram abordados na Seção 3.3.4, deste relatório.

addr	cálculo de endereços para instruções de *read/write memory*
ar	*labels* e endereços que necessitam registradores
con	constantes
rc	registradores e constantes
rc5	registradores e constantes com tamanho de 4 bits
reg	cálculos que tem resultado armazenado em registradores
stmt	cálculos realizadas por *side effect*

Figura 60 – Relação de símbolos não-terminais e seu significado [31].

Para a descrição das instruções do processador R7 foram alteradas produções da gramática escolhida como modelo (MIPS) de forma que retratassem as instruções do novo processador. Essas alterações exigiram a exclusão de algumas regras existentes e a adaptação de outras. Um exemplo de exclusão são as regras que possuem símbolos não-terminais e terminais específicos de ponto flutuante, cujo tratamento não é escopo desse trabalho.

A seguir mostramos alguns exemplos das principais regras alteradas. Vale ressaltar que o conjunto completo das regras de descrição da gramática do processador R7 encontra-se no Anexo deste relatório.

A Figura 61 mostra as produções existentes que tratam a operação de adição. Essas produções são utilizadas quando a adição é realizada entre registradores e entre registradores e constantes. Vale ressaltar, nas produções da Figura 61, a utilização do símbolo "?" comentado na Seção 5.5.1. Na Figura 62 é mostrado um exemplo dessa utilização em operações com adição de uma variável *char* com a constante 2. Cabe ressaltar que o *assembly* mostrado se refere apenas ao corpo do programa principal sem o cabeçalho de introdução. Nota-se que no caso da adição mostrada na Figura 62 são utilizadas também as instruções de *load high/low*. A emissão dessas instruções é tratada na função *emit2*, devido a sua especificidade (ver Seção 5.7.3.2).

```
reg: ADDI4(reg,con)    "?ADD R%c,R0,R%0 \n ADDI R%c,#%1\n" 1
reg: ADDI4(reg,reg)    "ADD R%c,R%0,R%1 \n"                1
reg: ADDP4(reg,con)    "?ADD R%c,R0,R%0 \n ADDI R%c,#%1\n" 1
reg: ADDP4(reg,reg)    "ADD R%c,R%0,R%1 \n"                1
reg: ADDU4(reg,con)    "?ADD R%c,R0,R%0 \n ADDI R%c,#%1\n" 1
reg: ADDU4(reg,reg)    "ADD R%c,R%0,R%1 \n"                1
```

Figura 61 - Produções de tratamento de operações de adição.

Na Figura 63 mostramos as regras utilizadas para armazenamento de valores em endereços de memória (ST) e para busca de valores em endereços de memória (LD).

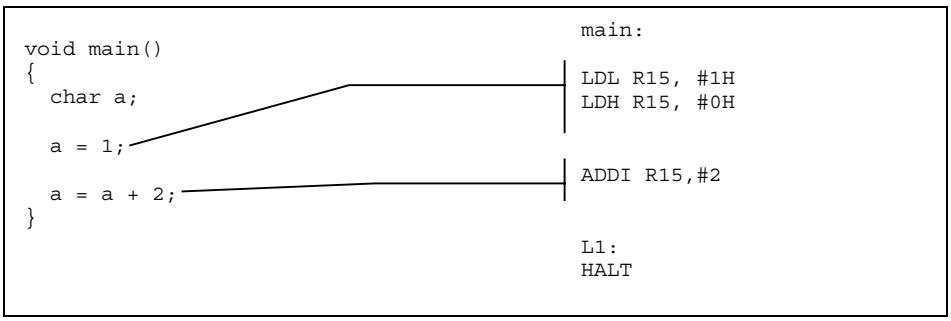

Figura 62 - Geração de *assembly* para operações de adição.

Nas duas primeiras linhas de código é realizada a carga do registrador R15 com a constante 1. Vale ressaltar que a variável "a", mostrada na Figura 62 tem seu uso otimizado através de operações sobre o registrador R15. Já na terceira linha de código o registrador R15 é adicionado com a constante 2 e carregado com o resultado final da operação.

```
stmt: ASGNI1(addr1,reg)   "ST R%1,R0,R%0 \n"  1
stmt: ASGNU1(addr1,reg)   "ST R%1,R0,R%0 \n"  1
stmt: ASGNI2(addr1,reg)   "ST R%1,R0,R%0 \n"  1
stmt: ASGNU2(addr1,reg)   "ST R%1,R0,R%0 \n"  1
stmt: ASGNI4(addr1,reg)   "ST R%1,R0,R%0 \n"  1
stmt: ASGNU4(addr1,reg)   "ST R%1,R0,R%0 \n"  1
stmt: ASGNP4(addr1,reg)   "ST R%1,R0,R%0 \n"  1

reg:  INDIRI1(addr)      "LD R%c,R0,R%0 \n"  1
reg:  INDIRU1(addr)      "LD R%c,R0,R%0 \n"  1
reg:  INDIRI2(addr)      "LD R%c,R0,R%0 \n"  1
reg:  INDIRU2(addr)      "LD R%c,R0,R%0 \n"  1
reg:  INDIRI4(addr)      "LD R%c,R0,R%0 \n"  1
reg:  INDIRU4(addr)      "LD R%c,R0,R%0 \n"  1
reg:  INDIRP4(addr)      "LD R%c,R0,R%0 \n"  1
```

Figura 63 - Produções para tratamento de instruções de *load/store.*

Para visualização do efeito das produções mostradas na Figura 63, é mostrado um exemplo através da Figura 64.

O trecho de código identificado com o número 1 corresponde à primeira atribuição do programa fonte. As primeiras duas instruções *assembly* indicam o cálculo do endereço da variável 'a', que fica armazenado no registrador R9. As duas instruções subseqüentes são referentes à constante a ser armazenada no endereço previamente calculado. A última instrução do trecho efetua o armazenamento da constante no endereço calculado. No trecho 2 a situação é idêntica.

No trecho de número 3 as quatro primeiras instruções realizam o cálculo dos endereços das variáveis 'a' e 'b'. A penúltima instrução do trecho mostra a carga do registrador R8 com conteúdo da variável 'b', cujo endereço encontra-se no registrador R8. Na última instrução este conteúdo é armazenado no endereço da variável 'c', que se encontra no registrador R9. Nas instruções LD e ST o registrador R1 é utilizado como registrador base dos endereços das variáveis.

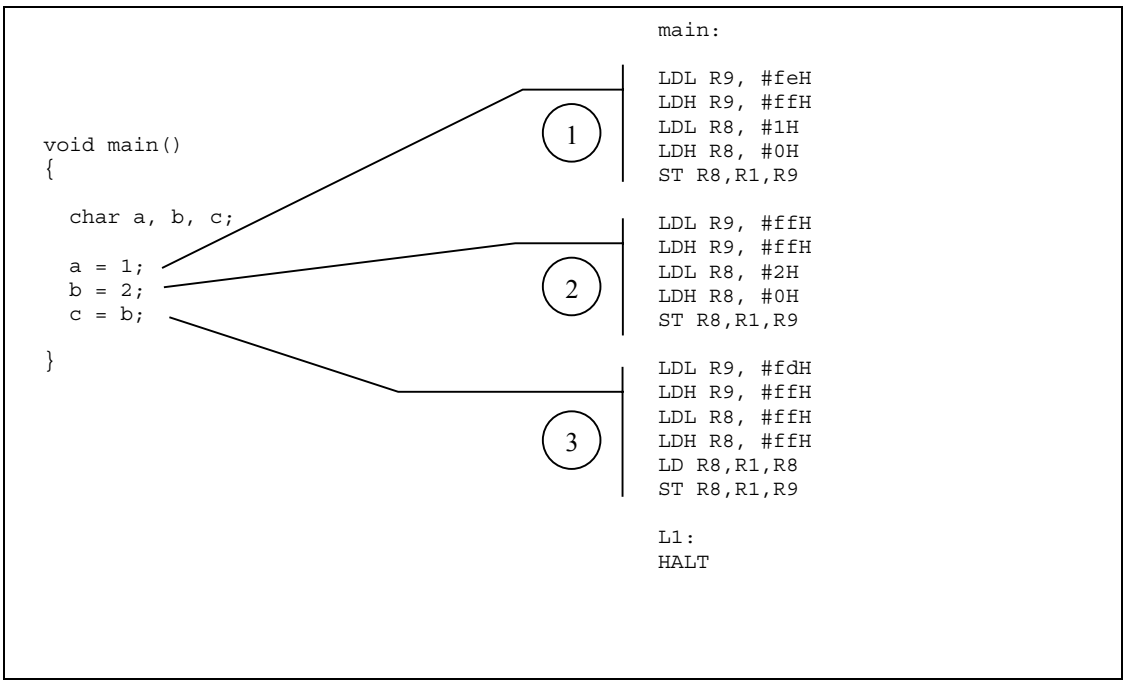

Figura 64 - Geração de *assembly* para instruções de *load/store*.

O tratamento de instruções de saltos condicionais exige produções que permitam a composição destes com instruções para operações de comparação que alteram o *flag* de estado, a ser testado para satisfazer a condição do salto. Na Figura 65 são mostradas as produções resultantes dessa composição.

```
stmt: JUMPV(con)   "EQL R0, R0\nJMPD %0\n"        1
stmt: JUMPV(reg)    "EQL R0, R0\nJMP R%0\n"       1

stmt: EQI4(reg,reg)  "EQL R%0,R%1\nJMPD %a\n"    1
stmt: EQU4(reg,reg)  "EQL R%0,R%1\nJMPD %a\n"    1

stmt: GEI4(reg,reg)  "SUP R%0,R%1\nJMPD %a\nEQL R%0,R%1\nJMPD %a\n"   1
stmt: GEU4(reg,reg)  "SUP R%0,R%1\nJMPD %a\nEQL R%0,R%1\nJMPD %a\n"   1

stmt: GTI4(reg,reg)  "SUP R%0,R%1\nJMPD %a\n"    1
stmt: GTU4(reg,reg)  "SUP R%0,R%1\nJMPD %a\n"    1

stmt: LEI4(reg,reg)  "INF R%0,R%1\nJMPD %a\nEQL R%0,R%1\nJMPD %a\n"   1
stmt: LEU4(reg,reg)  "INF R%0,R%1\nJMPD %a\nEQL R%0,R%1\nJMPD %a\n"   1

stmt: LTI4(reg,reg)  "INF R%0,R%1\nJMPD %a\n"    1
stmt: LTU4(reg,reg)  "INF R%0,R%1\nJMPD %a\n"    1

stmt: NEI4(reg,reg)  "DIF R%0,R%1\nJMPD %a\n"    1
stmt: NEU4(reg,reg)  "DIF R%0,R%1\nJMPD %a\n"    1
```

Figura 65 - Produções de tratamento de saltos condicionais e operações de comparação.

Todas as instruções de salto do processador R7 são condicionais. Isto faz com que nas instruções de saltos incondicionais (representadas pelos operadores JUMPV) seja necessário que o *flag* de estado seja alterado de forma a executar o salto. Por isso o *template* para saltos incondicionais possui a instrução EQL R0, R0.

Para visualização do uso das produções da Figura 65 inclui-se um exemplo simples com operações de comparação, utilizando como base o comando *if*, na Figura 66.

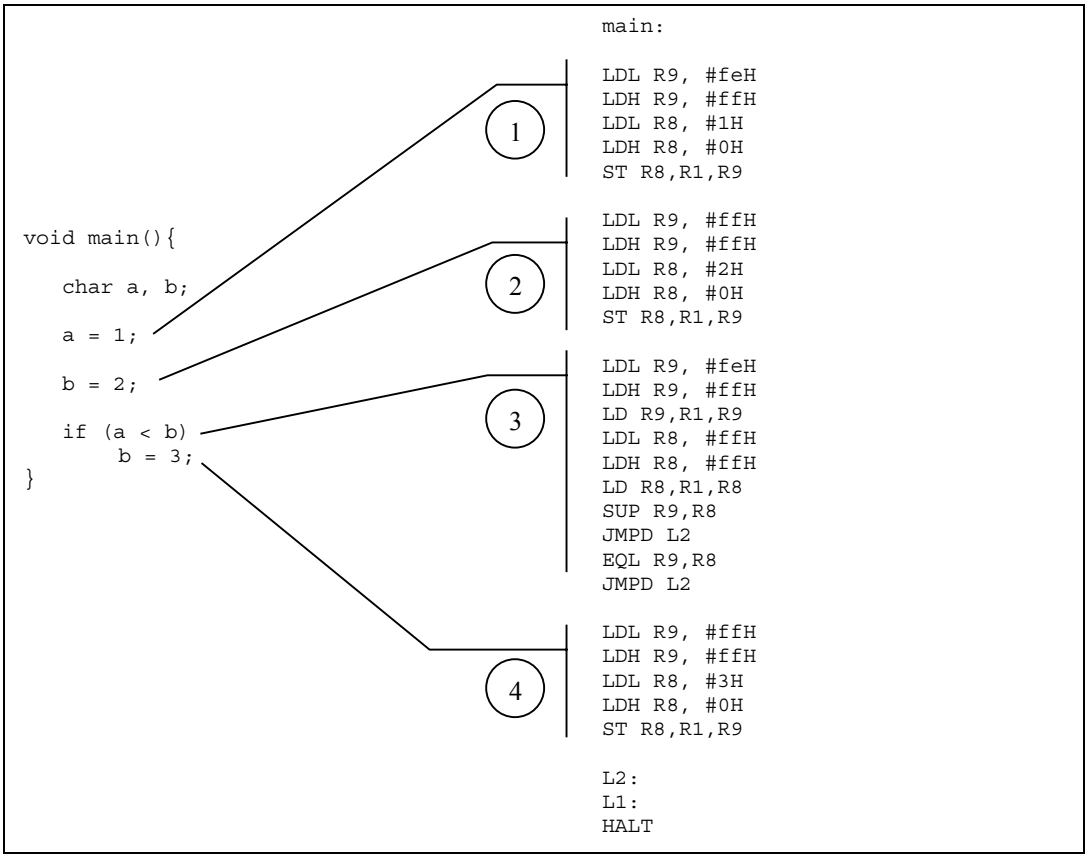

Figura 66 - Geração de *assembly* para comandos de salto condicional.

Vale ressaltar nas produções da Figura 65 a utilização dos símbolos "%1" e "%0" que indicam ao gerador de código que nessa posição o *template* a ser impresso é aquele proveniente da sub-árvore mais à direita (1) ou da sub-árvore mais à esquerda (0). Já a opção "%a" indica uma referência para a posição do símbolo na tabela de símbolos.

No exemplo da Figura 66 constam ainda as instruções LDL e LDH, que neste caso, realizam o cálculo de endereços das variáveis locais. Essas instruções são tratadas na função *emit2*, pois tem tratamento diferenciado devido a necessidade de separar a parta alta e a parta baixa do endereço quando este é carregado em registradores (restrição do processador R7). Como pode-se observar nos trechos 1, 2 e 4, este exemplo assemelha-se com o exemplo da Figura 65. Desta forma destacam-se as instruções do trecho 3, referentes à operação de comparação entre as variáveis 'a' e 'b'. As primeiras 3 instruções deste trecho calculam o endereço e fazem a carga do conteúdo da variável 'a' em R9. As próximas 3 fazem o mesmo para a variável 'b' armazenado seu conteúdo em R8. A seguir efetua-se a comparação entre os valores. Destaca-se que os testes de comparação são realizados com base em lógica negada, o que minimiza o número de instruções.

Na alteração da gramática é importante salientar a utilização de um custo associado a cada produção, como se pode ver na Figura 65. Isso se deve as especificações do *Lburg* [34]. Os custos associados as produções são constantes, porém alguns dependem das propriedades do nodo a que se referem. Um exemplo disso é a utilização da função *range*, a qual é inserida no arquivo *md* naquelas produções onde há a manipulação de valores constantes. Essa função tem a finalidade de

verificar se a constante está em um intervalo de valores permitido. Na Figura 67 é mostrado um exemplo de utilização da função *range*. Ao passar pela segunda produção os parâmetros indicam que aquela constante não pode ultrapassar o valor 15. Caso o valor esteja dentro do intervalo o custo a ser retornado é zero, do contrário o custo retornado é considerado demasiadamente alto e produção não será executada.

```
ar:  CNSTP4  "%a"   range(a, 0, 0xffff)
rc5: CNSTI4  "%a"   range(a,0,15)
```

Figura 67 - Exemplo de utilização da função *range*.

No exemplo da Figura 67 a constante representada pelo símbolo "%a" deve ter seu valor, no caso da primeira produção, representado no intervalo constante de 0 a 65535 (sabendo que este operador define um ponteiro, compreende-se que este é o intervalo de endereços válidos). Para a segunda produção o valor deve ser representado no intervalo constante de 0 a 15 (rc5 → registradores ou constantes de 4 *bits*). Ressalta-se que caso não ocorra indicação de valor ao lado da produção ou não ocorra outro tipo de função de retorno de custos, o valor associado será zero.

5.7.3 Alterações nas Funções de *Interface*

No processo de configuração, foram alteradas algumas funções de *interface progbeg()*, *emit2(), function(), defconst(), defstring(), global()*. Todas as alterações aqui descritas podem ser visualizadas mediante a consulta ao Anexo que contém o arquivo *gaph.md*.

5.7.3.1 Alterações na Função *progbeg*

As alterações da Figura 58 ocorrem nas linhas do arquivo *md* mostradas na Figura 68 para preencher os vetores de máscara de estados de registradores. Essas linhas fazem parte da função *progbeg()* vista na Seção 3.3.3.

```
tmask[IREG] = INTTMP;
vmask[IREG] = INTVAR;
```

Figura 68 - Vetores de máscara de registradores.

O vetor *tmask* armazena na posição *IREG* a máscara que indica os registradores que podem ser alocados para alocação de valores temporários, ou seja, variáveis cujo valor é alterado diversas vezes ao longo do código. O valor 0x03C0 que corresponde à constante INTTMP na Figura 59, significa que os *bits* de número 6 ao 9 estão setadas em 1. Isso representa que os registradores R6 ao R9 serão utilizados para a alocação dos valores temporários.

O vetor *vmask* indica quais registradores serão alocados para armazenar valores de variáveis cujo conteúdo será alterado diversas durante a execução do programa. No LCC essas são chamadas de variáveis *register*. Vale ressaltar que essa denominação *register* não tem correlação com o tipo *register* existente na linguagem C.

Outra alteração efetuada nesta função diz respeito a definição do número de registradores utilizados pelo processador. O trecho de código mostrado na Figura 69 indica que o número de registradores utilizados é 16. Além disso todos os registradores utilizados são simples *single width registers* (ver Seção 5.2).

```
for(i=0;i<=15;i++)
        ireg[i] = mkreg ( "%d", i, 1, IREG  );
```

Figura 69 - Laço for que define o número de registradores [31].

A função *mkreg* da Figura 69 é utilizada para construir descritores de registrador. O vetor *ireg* é um vetor destes descritores. O primeiro parâmetro indica o nome do registrador, o segundo é o número e o terceiro parâmetro indica o conjunto ao qual o registrador pertence. O nome do registrador pode incluir seu número, o que ocorre neste caso.

Para a inserção do cabeçalho padrão vinculado sempre no início do *assembly* do processador R7 foram inseridas as linhas da Figura 70 na função *progbeg*.

```
print("\n.CODE\n");
print("LDL R1,#feH\n");
print("LDH R1,#9fH\n");
print("LDSP R1\n"); // inicializando o SP
print("XOR R0,R0,R0\n");
print("EQL R0,R0\n");
print("JMPD main\n");
```

Figura 70 - Cabeçalho padrão do *assembly* do processador R7.

5.7.3.2 Alterações na Função *emit2*

Conforme abordado anteriormente, esta função é acionada quando é necessário a geração de código que não é possível de ser expresso apenas com os recursos existentes no *template*. Nesta função foram feitas as seguintes alterações: (a) retirada de todas as declarações de variáveis; (b) inserção das variáveis do tipo *int* registrador, endereço e constante; (c) alteração das opções do comando *switch (specific(p->op))*. As alterações enumeradas acima podem ser vistas na Figura 71.

```
static void emit2(Node p)

{
int registrador, endereco, constante;
switch (specific(p->op))
{
   case ADDRL+P: case ADDRF+P:
       registrador = getregnum(p);
       endereco = p->syms[0]->x.offset;
       print("LDL R%d, #%XH ;%d\n", registrador, endereco & 0xff, endereco);
       print("LDH R%d, #%XH ;%d\n", registrador, (endereco & 0xff00) / 0x100 , endereco);
       break;
   case CNST+I: case CNST+U: case CNST+P:
       registrador = getregnum(p);
       constante = atoi(p->syms[0]->x.name);
       if (registrador != 0){
         print("LDL R%d, #%XH ;%d\n", registrador, constante & 0xff, constante);
         print("LDH R%d, #%XH ;%d\n", registrador, (constante & 0xff00) / 0x100, constante);
       }
       break;
   }
}
```

Figura 71 - Código da função *emit2*.

As novas variáveis *endereço*, *registrador* e *constante* significam respectivamente o endereço a ser carregado em registrador, o número do registrador a ser utilizado e o valor da constante a ser armazenada naquele registrador.

No caso do processador R7 a função *emit2* foi utilizada para emitir código para cálculos de endereço e cargas de constantes em registradores. No primeiro *case* as informações, sobre os operadores *ADDRLP* e *ADDRFP*, são utilizadas para o armazenamento do endereço em registrador. Já no segundo é feito o armazenamento de constantes simbolizadas pelo operadores *CNSTI, CNSTU* e *CNSTP*. São utilizadas máscaras que quebram o valor em dois *bytes*, parte alta e parte baixa, pois no R7 só e possível carregar um registrador com uma constante um *byte* por vez, ou seja, são necessárias duas instruções para carregar um registrador. A instrução LDH é utilizada para carregar a parte alta do registrador e a instrução LDL para a parte baixa.

5.7.3.3 Alterações na Função *function*

Nessa função foram incluídas as linhas mostradas na Figura 72, e ainda retiradas todas as referência de geração de código para dados em ponto flutuante.

```
// Se nao for o main imprime RTS
   if (strcmp("main",f->x.name))
         print("RTS\n");

   else{
        print("HALT\n\n");
        print(".ENDCODE\n");
       }
```

Figura 72 - Linhas de código inseridas na função *function*.

As linhas inseridas na função *function* têm o objetivo de imprimir o código de retorno no caso de subrotinas ou o código padrão de fim de programa, no caso do *assembly* gerado para o processador R7.

5.7.3.4 Alterações na Função *defconst*

Na função *defconst* a linha mostrada na Figura 73 foi inserida para gerar um representação de constantes em base hexadecimal.

```
print("#%XH ;[f15]\n", suffix == I ? v.i : v.u);
```

Figura 73 - Linha adicionada a função *defconst*.

A função *print* possui um operador que define o formato em que deve ser impresso o valor do segundo parâmetro. O padrão “%X” indica que este valor deve ser impresso em base hexadecimal.

5.7.3.5 Alterações na Função *global*

Esta função é utilizada para a emissão de código de definição de variáveis globais. Caso a variável não esteja inicializada definiu-se como padrão inicializar a posição de memória da referida variável com a constante zero. Essa convenção decorre da necessidade de se declarar, no

caso do processador R7, uma variável global através da diretiva “DB” obrigatoriamente inicializada com seu respectivo valor. Caso contrário o compilador irá invocar a função *defconst* ou *defstring* dependendo do tipo da variável declarada.

Na função *global* são inicializadas as variáveis “contador” e “tamanho” que serão utilizadas na função *defstring* para definição de *strings* ou vetores do tipo *char*. Essas variáveis são declaradas como globais no arquivo *md*.

As alterações da função *global* são mostradas na Figura 74.

```
static void global(Symbol p)
{
        if (p->u.seg == BSS) { //se nao esta inicializada uma variavel global
             print("%s: DB #00 ;[f18]\n", p->x.name);
        } else {
             contador = 1;
             tamanho = p->type->size;
             print("%s: DB ", p->x.name);
        }
}
```

Figura 74 - Alterações da função *global*.

5.7.3.6 Alterações na Função *defstring*

Tendo em vista que o tipo de dado tratado pelo processador R7 é apenas *char*, quando ocorrer a declaração de vetores ou *strings* no código fonte a função *defstring* será acionada. Vale ressaltar, que quando ocorre declaração de vetores ou *strings* dentro de uma rotina, o compilador LCC cria um *label* para cada uma dessas variáveis. Isto ocorre para que seja possível alocar, em tempo de compilação, o espaço necessário para armazenar o conteúdo dessas variáveis. Esse *label* é inserido na mesma área destinada a armazenar as variáveis globais. Sendo assim a cada definição de uma *string* ou vetor do tipo *char* a função *global* é acionada. Devido a essa forma de alocação essas *strings* ou os vetores do tipo *char* declarados como variáveis locais são tratados como globais sempre.

```
static void defstring(int n, char *str) {
        char *s;
        for (s = str; s < str + n; s++){
              if (contador == tamanho){
                    print("#%XH ;[f17]\n", (*s));
                    break;
              }
              print("#%XH, ", (*s));
              contador++;
        }
```

Figura 75 - Alterações da função *defstring*.

Na Figura 75 é mostrada a função *defstring* com as alterações. Como essa função trabalha em conjunto com a função *global* as variáveis “contador” e “tamanho” são inicializadas naquela função. Como no caso do *assembly* do processador R7 é necessário que, na definição de vetores, as constantes referentes às posições do vetor estejam lado a lado, foi inserido o teste “*if(contador == tamanho)”* que efetua a quebra da linha quando ocorrer a definição do último elemento.

5.7.3.7 Outras Funções

Para descrição do processador R7 algumas funções foram suprimidas do arquivo *md* que serviu de base para o projeto (MIPS). A retirada dessas funções de *interface* se deu devido à falta de utilidade na geração de código para o caso do processador R7. As funções retiradas da descrição foram: *export*, *import*, *segment*, *space*, *blkloop*, *blkfetch*, *blkstore*, *stabinit*, *stabline*, *stabsym*, **currentfile*, *stabinit*, *stabline*, *stabsym*.

Cabe ressaltar que apenas o código das funções foi retirado, permanecendo as respectivas declarações das mesmas sob a forma de protótipo, tendo em vista as rotinas executadas pelo *front end*.

No próximo Capítulo são mostrados os programas definidos para testes sobre o *assembly* gerado e a respectiva simulação desses códigos que permite a respectiva validação da fase final do projeto.

6. SIMULAÇÃO/VALIDAÇÃO DO *ASSEMBLY*

Este Capítulo tem o objetivo de mostrar o *assembly* produzido pelo gerador de código (*back end*), baseado na descrição do processador-alvo R7 inserida no compilador LCC e a respectiva validação desse *assembly* com a simulação de programas simples em linguagem C.

Para essa etapa foi utilizado o simulador elaborado no GAPH pela bolsista Aline Vieira de Melo, orientada pelo Professor Fernando Gehm Moraes. Este *software* pode ser utilizado após *download* do arquivo *simuladorR8.zip* no endereço www.inf.pucrs.br/~moraes.

6.1 Programas-exemplo

Nesta Seção serão mostrados os programas utilizados para teste e o *assembly* testado no simulador elaborado no GAPH. Para a bateria de testes foram definidos 8 programas conforme a relação abaixo:

a) Exemplo 1: contempla testes com atribuições, adição e subtração;
b) Exemplo 2: contempla testes sobre laços do comando *for*;
c) Exemplo 3: contempla testes sobre o comando *switch*;
d) Exemplo 4: contempla testes sobre comandos *if* e *while*;
e) Exemplo 5: contempla testes sobre passagem de parâmetros, saltos para subrotinas e tratamento de variáveis globais;
f) Exemplo 6: contempla testes sobre tratamento de vetores;
g) Exemplo 7: contempla uso de ponteiros e passagem de parâmetros por referência;
h) Exemplo 8: contempla testes sobre o programa de ordenação BUBBLE SORT.

Os programas utilizados para teste são simples. Isso se deve ao fato de que a medida que a complexidade do código em linguagem C aumenta, com a inclusão de maior número de variáveis, estruturas e outros comandos, o código em linguagem *assembly* aumenta consideravelmente, devido à necessidade de um maior número de instruções.

6.1.1 Exemplo 1 – Arquivos: "*exoperacoes.c*" e "*exoperacoes.asm*"

Apresenta-se na Figura 76 o código utilizado para geração do *assembly* visando o teste de operações simples de adição, subtração e de atribuições de tipos *char*. Ressalta-se que no processador R7 foram utilizados apenas os tipos *char* e *unsigned char* pelos motivos já explicados na Seção 5.7.1.

```
1.  void main()
2.  {
3.
4.      char a, b, e;
5.      char c, d;
6.
7.      a = 10;
8.      b = 20;
9.      e = 30;
10.
11.     c = 'a';
12.     d = 'b';
13.
14.     a = b + a;
15.     b = a - b;
16.
17. }
```

Figura 76 - Exemplo com adição, subtração e atribuições.

O programa é simples e possui a declaração de cinco variáveis de tipo *char*. A seguir são efetuadas atribuições às variáveis inteiras e caracter, com posterior adição e subtração entre inteiros.

O *assembly* gerado nesse caso pode ser visto na Figura 77 abaixo.

```
1.   .CODE
2.  LDL R1,#feH
3.  LDH R1,#9fH
4.  LDSP R1
5.  XOR R0,R0,R0
6.  EQL R0,R0
7.  JMPD main.............................Cabeçalho comum a todos os programas
8.  main:
9.  LDL R15, #aH..........................Carga da constante 10 em R15 (variável 'a')
10. LDH R15, #0H
11. LDL R14, #14H.........................Carga da constante 20 em R14 (variável 'b')
12. LDH R14, #0H
13. LDL R10, #ffH.........................Busca do endereço da variável 'e' (-1)
14. LDH R10, #ffH
15. LDL R9, #1eH..........................Carga da constante 30 em R9
16. LDH R9, #0H
17. ST R9,R0,R10..........................Atribuição → e = 30, na pilha de dados PD[-1]
18. LDL R10, #feH.........................Busca do endereço da variável 'c' (-2)
19. LDH R10, #ffH
20. LDL R9, #61H..........................Carga do código ASCII de 'a' em R9
21. LDH R9, #0H
22. ST R9,R0,R10..........................Atribuição → c = 'a', na pilha de dados PD[-2]
23. LDL R10, #fdH.........................Busca do endereço da variável 'd' (-3)
24. LDH R10, #ffH
25. LDL R9, #62H..........................Carga do código ASCII de 'b' em R9
26. LDH R9, #0H
27. ST R9,R0,R10..........................Atribuição → d = 'b', na pilha de dados PD[-3]
28. ADD R10,R0,R14........................Carga de R10 com o valor de R14 (variável 'b')
29. ADD R9,R0,R15.........................Carga de R9 com o valor de R15
30. ADD R9,R10,R9..........................Adição → b + a
31. ADD R15,R0,R9..........................Atualização do valor de R15
32. ADD R9,R0,R15.........................Carga de R9 com o valor de R15 (variável 'a')
33. SUB R10,R9,R10.........................Subtração → a - b
34. ADD R14,R0,R10.........................Atualização do valor de R14
35. L1:
36. HALT
37.
38. .ENDCODE
```

Figura 77 - *Assembly* gerado para o exemplo 1.

Entre as linhas 1 e 7 da Figura 77 temos o cabeçalho padrão para programas *assembly* do processador R7 como definido na Seção 4.5. Logo após, entre as linhas 9 e 12 são atribuídos os valores das variáveis ‘a’ e ‘b’ aos registradores R14 e R15, visando otimização nas operações com essas variáveis. Estas operações são executadas entre as linhas 30 a 33.

Já no tratamento da variável ‘e’, efetuado entre as linhas 13 e 17, o valor que lhe será atribuído não é carregado em registrador, pois como se pode ver essa variável não se altera ao longo do programa. Entre as linhas 18 e 27 é realizada a busca das variáveis ‘c’ e ‘d’ e a respectiva atribuição dos valores ‘a’ e ‘b’ (com utilização do código ASCII) respectivamente, também na pilha de dados.

Na Figura 78 mostramos a tela do simulador com o teste do exemplo 1.

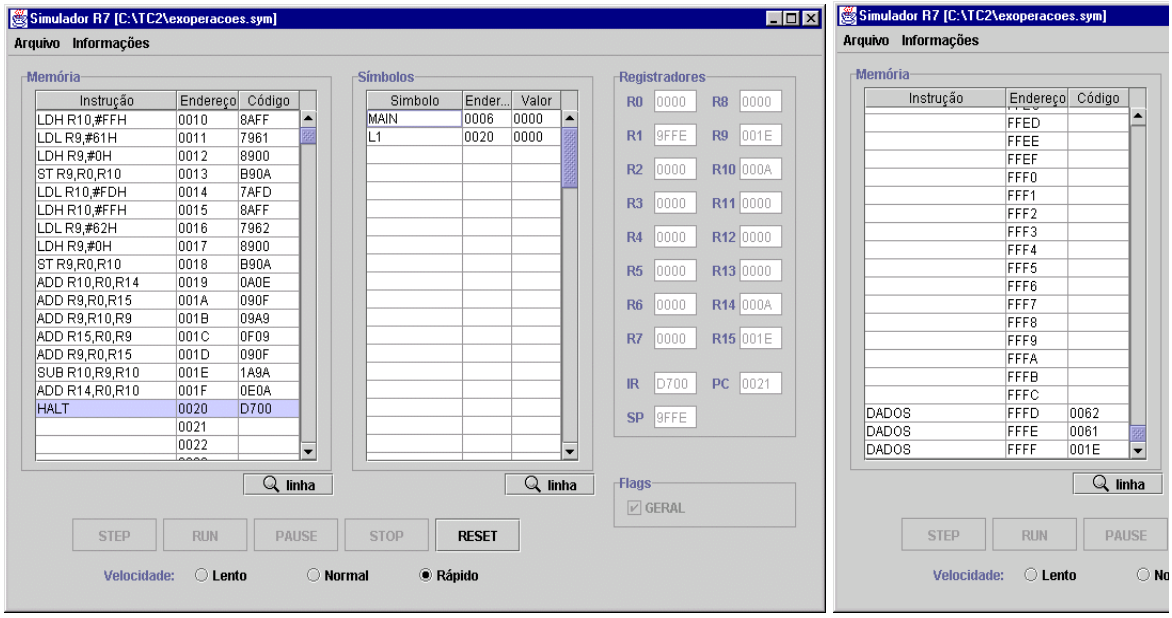

Figura 78 - Simulação do exemplo 1, onde são mostrados os conteúdos dos registradores e área de memória para as variáveis ‘e’ => FFFF, ‘c’ => FFFE e ‘d’ => FFFD.

Nesta simulação destacamos a corretude do código, validado pelos resultados finais dos registradores R14=000A e R15=001E, correspondendo aos valores decimais 10 e 30 respectivamente. Esses valores correspondem aos resultados da subtração e da adição realizadas no exemplo 1. Na Figura também é mostrada a área de memória onde são gravadas as atribuições d=‘a’, c=‘b’ e e=30. Sendo que em hexadecimal ‘a’ corresponde a 0061, ‘b’ corresponde a 0062 e 30 corresponde a 001E. Os valores mostrados correspondem aos resultados finais das variáveis.

6.1.2 Exemplo 2 – Arquivos: "*exfor.c*" e "*exfor.asm*"

Na Figura 79 mostramos um código simples elaborado para testes sobre a execução dos laços do comando *for*.

```
1.  void main()
2.  {
3.     char a, b, c;
4.
5.     b = 1;
6.     c = 2;
7.
8.     for(a=0; a<5; a++)
9.     {
10.       b = b + c;
11.       c = c + 1;
12.    }
13.
14. }
```

Figura 79 - Exemplo com comando *for*.

A partir do código da Figura 79 foi gerado o *assembly* mostrado na Figura 80.

```
1.   .CODE
2.   LDL R1,#feH
3.   LDH R1,#9fH
4.   LDSP R1
5.   XOR R0,R0,R0
6.   EQL R0,R0
7.   JMPD main.............................Cabeçalho comum a todos os programas
8.   main:
9.   LDL R13, #1H..........................Carga da constante 1 em R13
10.  LDH R13, #0H
11.  LDL R15, #2H..........................Carga da constante 2 em R15
12.  LDH R15, #0H
13.  ADD R14,R0,R0.........................R14 definido como iterador 'a'
14.  LDL R10 #L5
15.  LDH R10 #L5............................Salto para o teste
16.  EQL R0, R0
17.  JMP R10
18.  L2:
19.  ADD R10,R0,R15........................Carga do valor de R15 em R10 (c)
20.  ADD R9,R0,R13..........................Carga do valor de R13 em R9 (b)
21.  ADD R9,R9,R10..........................Adição → b + c
22.  ADD R13,R0,R9..........................Atualização de R13 (b)
23.  ADDI R10,#1............................Adição → c + 1
24.  ADD R15,R0,R10.........................Atualização de R15 (c)
25.  L3:
26.  ADD R10,R0,R14........................Carga de R14 em R9
27.  ADDI R10,#1............................Incremento → a++
28.  ADD R14,R0,R10.........................Atualização de R14
29.  L5:
30.  ADD R10,R0,R14........................Carga do valor de R14 em R10
31.  LDL R9, #5H............................Carga da constante 5 em R9
32.  LDH R9, #0H
33.  INF R10,R9..............................Teste → a < 5
34.  JMPD L2................................Se verdadeiro volta para L2
35.  L1:
36.  HALT
37.
38.  .ENDCODE
```

Figura 80 - *Assembly* para o exemplo 2.

Como no exemplo anterior da Figura 76, o *assembly* da Figura 80 possui a introdução padrão. Ressalta-se a utilização de registradores para operações sobre variáveis que se alteram durante o programa, como é o caso do iterador 'a' e das variáveis 'b' e 'c' do exemplo 2.

Na Figura 81 é apresentada a simulação do exemplo 2, que representa o teste para a implementação de laços com o comando *for*. Nesse exemplo deve-se ressaltar o resultado obtido após as iterações, que pode ser observado através dos registradores R13, R14 e R15, os quais

contém o valor final das variáveis ‘b’, ‘a’ e ‘c’ respectivamente.

Merece destaque também a tabela de símbolos onde constam os *labels* que serão utilizados nos saltos para as iterações do comando *for*.

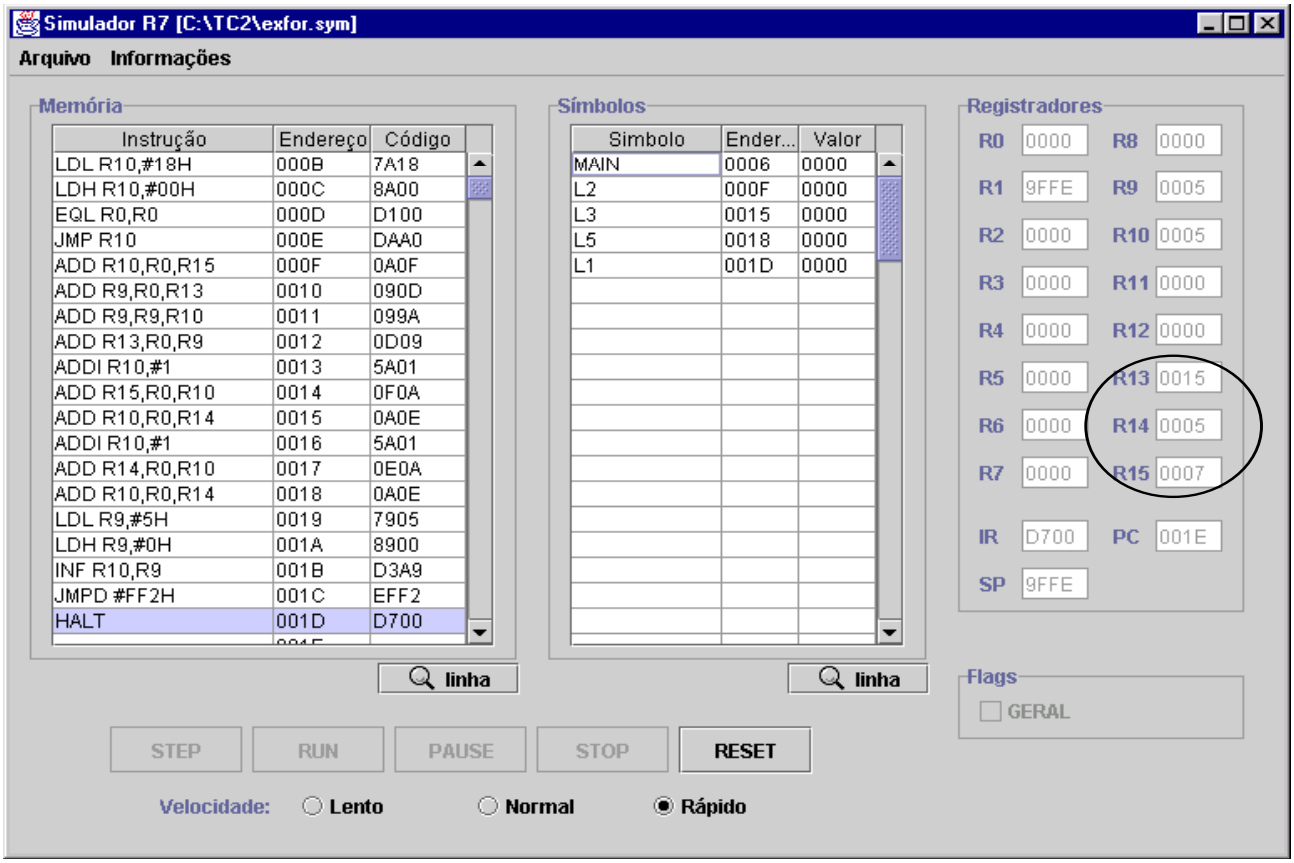

Figura 81 - Simulação do exemplo 2.

A manipulação de registradores no lugar de variáveis vem da otimização inserida como característica do próprio compilador utilizado, no caso do LCC. Essa característica contribui para a minimização de acessos à memória.

6.1.3 Exemplo 3 – Arquivos: “*exswitch.c*” e “*exswitch.asm*”

Neste exemplo foram testados diversos saltos condicionais com operações simples de adição dentro do comando *switch*. Na Figura 82 pode-se ver o código utilizado para geração do *assembly*.

```
1.  # include "stdio.h"
2.  # include "stdlib.h"
3.
4.  void main()
5.  {
6.     char a, b;
7.
8.     a = 1;
9.     b = 2;
10.
11.    switch(a)
12.    { case 1: b = a + 1;
13.              break;
14.      case 2: a = b + 1;
15.              break;
16.    }
17.
18.    switch(b)
19.    { case 1: b = a + 1;
```

```
20.             break;
21.      case 2: b = b + 1;
22.             break;
23.    }
24. }
```

Figura 82 - Código para teste do comando *switch*.

O respectivo código *assembly* para o exemplo 3 é mostrado na Figura 83.

```
1.   .CODE
2.   LDL R1,#feH
3.   LDH R1,#9fH
4.   LDSP R1
5.   XOR R0,R0,R0
6.   EQL R0,R0
7.   JMPD main.............................Cabeçalho comum a todos os programas
8.   main:
9.   LDL R10, #feH.........................Busca do endereço da variável 'a'  (-2)
10.  LDH R10, #ffH
11.  LDL R9, #1H...........................Carga da constante 1 em R9
12.  LDH R9, #0H
13.  ST R9,R0,R10..........................Atribuição → a = 1 na pilha de dados PD[-1]
14.  LDL R10, #ffH.........................Busca do endereço da variável 'b'
15.  LDH R10, #ffH
16.  LDL R9, #2H...........................Carga da constante 2 em R9
17.  LDH R9, #0H
18.  ST R9,R0,R10..........................Atribuição → b = 2 na PD[-2]
19.  LDL R10, #feH.........................Busca do endereço de 'a'
20.  LDH R10, #ffH
21.  LD R15,R0,R10.........................Carga do conteúdo de 'a' em R15
22.  LDL R10, #1H...........................Carga da constante 1 em R10
23.  LDH R10, #0H
24.  EQL R15,R10............................Teste → a == 1
25.  JMPD L8................................Se verdadeiro salta para L8
26.  LDL R10, #2H
27.  LDH R10, #0H
28.  EQL R15,R10............................Teste → a == 2
29.  JMPD L9................................Se verdadeiro salta para L9
30.  LDL R10 #L5
31.  LDH R10 #L5
32.  EQL R0, R0............................Se 'a' diferente de 1 e 2 salta para L5
33.  JMP R10
34.  L8:.....................................Início do primeiro case, a == 1
35.  LDL R10, #ffH.........................Busca endereço de 'b'
36.  LDH R10, #ffH
37.  LDL R9, #feH..........................Busca endereço de 'a'
38.  LDH R9, #ffH
39.  LD R9,R0,R9............................Carga de R9 com conteúdo de 'a'
40.  ADDI R9,#1............................Adição → a + 1
41.  ST R9,R0,R10..........................Atribuição → b = a + 1
42.  LDL R10 #L6...........................Início do break
43.  LDH R10 #L6
44.  EQL R0, R0
45.  JMP R10...............................Salto para L6
46.  L9:.....................................Início do primeiro case a == 2
47.  LDL R10, #feH.........................Busca endereço de 'a'
48.  LDH R10, #ffH
49.  LDL R9, #ffH..........................Busca endereço de 'b'
50.  LDH R9, #ffH
51.  LD R9,R0,R9............................Carga do conteúdo de 'b' em R9
52.  ADDI R9,#1............................Adição → b + 1
53.  ST R9,R0,R10..........................Atribuição → a = b + 1
54.  L5:
55.  L6:
56.  LDL R10, #ffH.........................Busca endereço de 'b'
57.  LDH R10, #ffH
58.  LD R14,R0,R10..........................Carga de R14 com conteúdo de 'b'
59.  LDL R10, #1H...........................Carga da constante 1 em R10
60.  LDH R10, #0H
61.  EQL R14,R10............................Teste → b == 1
62.  JMPD L13...............................Se verdadeiro salta para L13
63.  LDL R10, #2H...........................Carga da constante 2 em R10
64.  LDH R10, #0H
65.  EQL R14,R10............................Teste → b == 2
66.  JMPD L14...............................Se verdadeiro salta para L14
67.  LDL R10, #L10.........................Se 'b' diferente de 1 e 2 salta para L10
68.  LDH R10, #L10
69.  EQL R0, R0
70.  JMP R10...............................Salto para L10
71.  L13:....................................Início do segundo case, b=1
72.  LDL R10, #ffH.........................Busca do endereço de 'b'
73.  LDH R10, #ffH
74.  LDL R9, #feH..........................Busca do endereço de 'a'
75.  LDH R9, #ffH
76.  LD R9,R0,R9............................Carga do conteúdo de 'a' em R8
77.  ADDI R9,#1............................Adição → a + 1
```

```
78.  ST R9,R0,R10.....................Atribuição → b = a + 1
79.  LDL R10, #L11.....................Início do break
80.  LDH R10, #L11
81.  EQL R0, R0
82.  JMP R10...........................Salta para L11
83.  L14:..............................Início do segundo case, b=2
84.  LDL R10, #ffH.....................Busca endereço de 'b'
85.  LDH R10, #ffH
86.  LDL R9, #ffH......................Busca endereço de 'b'
87.  LDH R9, #ffH
88.  LD R9,R0,R9........................Carga do conteúdo de 'b' em R9
89.  ADDI R9,#1.........................Adição → b + 1
90.  ST R9,R0,R10.......................Atribuição → b = b + 1
91.  L10:
92.  L11:
93.  L4:
94.  HALT..............................Fim de programa
95.
96.  .ENDCODE
```

Figura 83 - *Assembly* gerado para o exemplo 3.

No programa em *assembly* da Figura 83 são mostrados vários saltos devido aos comandos *switch* implementados e a vários testes que são necessários. Novamente entre as linhas 1 e 7 tem-se o cabeçalho padrão para programas do processador R7. Após este cabeçalho estão as instruções entre as linhas 9 e 13 que implementam a primeira atribuição do programa (a = 1) e entre as linhas 14 e 18 a segunda atribuição (b = 2), ambas variáveis na pilha de dados (PD).

Entre as linhas 19 e 29 são inseridas instruções necessárias ao primeiro teste do *switch(a)*, visando definir qual o *case* a ser executado para 'a' igual a 1 ou 2. Rotina semelhante é feita entre as linhas 56 e 66.

Entre as linhas 30 e 33 ocorre a inserção das instruções necessárias para salto ao próximo trecho de instruções que deve ser executado caso 'a' não seja igual a 1 ou 2. As mesmas instruções constam entre as linhas 67 e 70 para o caso da variável 'b'. A adição b=a+1 é feita entre as linhas 35 a 41, referente à primeira adição do comando *switch(a)*, sendo que a segunda adição é implementada entre as linhas 47 e 53.

O comando *break* é implementado através das instruções das linhas 42 a 45 e 79 a 82. Já nos intervalos das linhas 72 a 78 e 84 a 90 estão as instruções que implementam as adições existentes no *switch(b)*.

Na Figura 84 mostramos a simulação do *assembly* da Figura 83.

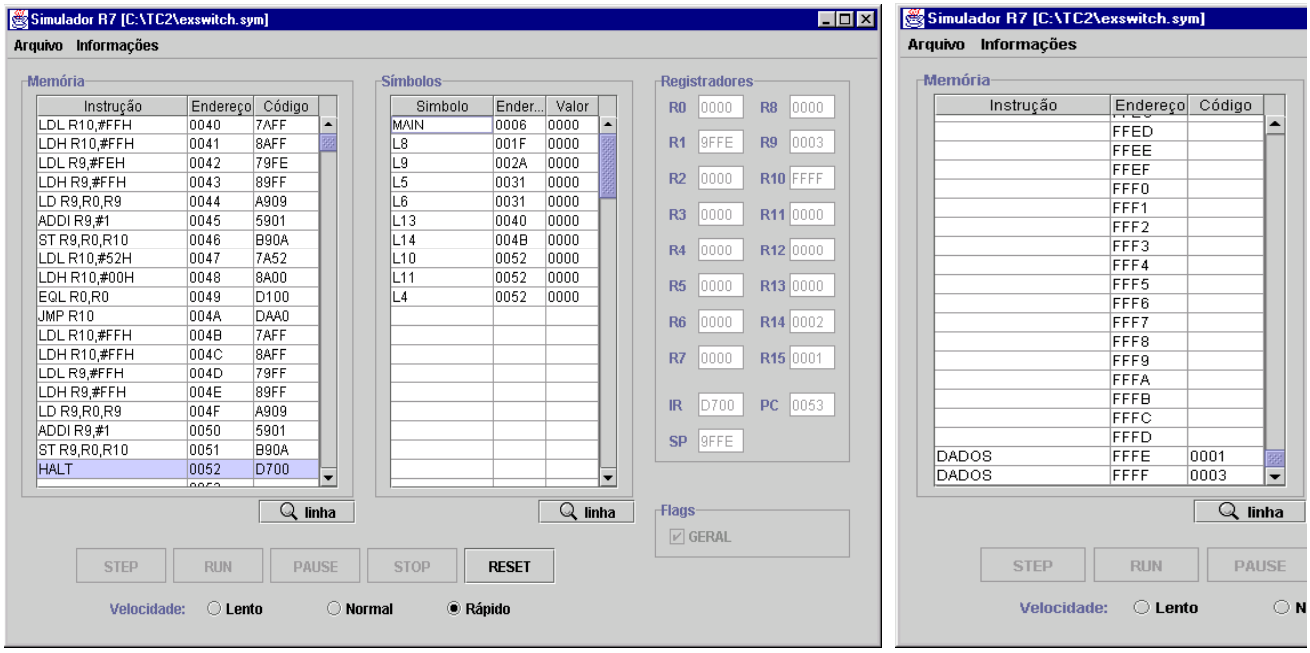

Figura 84 - Simulação do exemplo 3.

Na simulação mostrada na Figura 84 cabe destacar a tabela de *labels* formada devido ao grande número de saltos necessários na implementação do comando *switch*. Esses saltos são decorrentes das comparações e dos comandos *break* que estão inseridos no código. Também nesta Figura mostramos os resultados das operações internas ao comando *switch*, cujas variáveis 'a' e 'b' estão armazenadas na pilha de dados. Os valores armazenados nas células de memória mostradas na Figura já possuem os resultados finais decorrentes da execução dos comandos *switch* do exemplo 3.

6.1.4 Exemplo 4 – Arquivos: "*exifwhile.c*" e "*exifwhile.asm*"

Neste exemplo foram testados os comandos de controle de fluxo *if* e *while*. Na Figura 85 visualiza-se o programa fonte para a geração do *assembly*.

```
1.  void main()
2.  {
3.    char a, b, c, d;
4.
5.    a = 1;
6.    b = 2;
7.    c = 3;
8.    d = 4;
9.
10.   if (a > c) b = c;
11.   if (c < a) a = b;
12.
13.   if ((a > c) || (b < d)) d = 10;
14.
15.   if ((b > a) && (c < d)) d = 20;
16.
17.   while (a < c)
18.   {
19.     a = a + 1;
20.     b = a + b;
21.   }
22. }
```

Figura 85 - Código para teste de comandos *if* e *while*.

Na Figura 86 pode-se visualizar o *assembly* que implementa o exemplo 4.

```
1.   .CODE
2.   LDL R1,#feH
3.   LDH R1,#9fH
4.   LDSP R1
5.   XOR R0,R0,R0
6.   EQL R0,R0
7.   JMPD main.....................Cabeçalho comum a todos os programas
8.   main:
9.   LDL R15, #1H
10.  LDH R15, #0H
11.  LDL R14, #2H
12.  LDH R14, #0H
13.  LDL R13, #3H
14.  LDH R13, #0H
15.  LDL R12, #4H
16.  LDH R12, #0H.................'a' em R15, 'b' em R14, 'c' em R13, 'd' em R12
17.  ADD R10,R0,R15
18.  ADD R9,R0,R13
19.  INF R10,R9.....................a < c
20.  JMPD L2
21.  EQL R10,R9.....................a = c
22.  JMPD L2
23.  ADD R14,R0,R13.................b ← c
24.  L2:
25.  ADD R10,R0,R13
26.  ADD R9,R0,R15
27.  SUP R10,R9.....................c > a
28.  JMPD L4
29.  EQL R10,R9.....................c = a
30.  JMPD L4
31.  ADD R15,R0,R14.................a ← b
32.  L4:
33.  ADD R10,R0,R15
34.  ADD R9,R0,R13
35.  SUP R10,R9.....................a > c
36.  JMPD L8
37.  ADD R10,R0,R14
38.  ADD R9,R0,R12
39.  SUP R10,R9.....................b > d
40.  JMPD L6
41.  EQL R10,R9.....................b = d
42.  JMPD L6
43.  L8:
44.  LDL R12, #aH
45.  LDH R12, #0...................d ← 10
46.  L6:
47.  ADD R10,R0,R14
48.  ADD R9,R0,R15
49.  SUP R10,R9.....................b > a
50.  JMPD L12
51.  EQL R10,R9.....................b = a
52.  JMPD L12
53.  ADD R10,R0,R13
54.  ADD R9,R0,R12
55.  SUP R10,R9.....................c > d
56.  JMPD L12
57.  EQL R10,R9.....................c = d
58.  JMPD L12
59.  LDL R12, #14H
60.  LDH R12, #0H..................d ← 20
61.  LDL R10 #L12
62.  LDH R10 #L12
63.  EQL R0, R0
64.  JMP R10.......................salto incondicional para L12
65.  L11:
66.  ADD R10,R0,R15
67.  ADDI R10,#1
68.  ADD R15,R0,R10.................a ← a + 1
69.  ADD R10,R0,R15
70.  ADD R9,R0,R14
71.  ADD R10,R10,R9
72.  ADD R14,R0,R10.................b ← a + b
73.  L12:
74.  ADD R10,R0,R15
75.  ADD R9,R0,R13
76.  INF R10,R9...................Teste  se a < c  (controle do while)
77.  JMPD L11
78.  L1:
79.  HALT
80.
81.  .ENDCODE
```

Figura 86 - *Assembly* gerado para o exemplo 4.

No *assembly* da Figura 86, entre as linhas 9 e 16 são carregadas as 4 variáveis do programa em registradores. Esse passo se dá devido a otimização implementada pelo LCC, que

maximiza o uso de registradores locais.

O compilador quando traduz comandos de controle de fluxo sempre o faz testando as comparações em lógica negada. Por exemplo, para a comparação '*if (a > c)*' são feitas duas comparações, resultando em $a \leq c$, como acontece entre as linhas 17-22 (analogamente temos o contrário entre as linhas 25-30). Isto reduz o número de instruções do código.

Nos intervalos das linhas 44-45 e 59-60 são inseridas as instruções para implementação das atribuições seguidas dos dois últimos comandos *if*, ou seja, 'd = 10' e 'd = 20'.

Ainda entre as linhas 66-72 ocorre a implementação das operações internas ao comando *while*, ou seja, 'a = a + 1' e 'b = a + b'.

Na Figura 87 mostramos a simulação para a validação do *assembly* gerado do exemplo 4, ressaltando o valor final dos registradores R12, R13, R14 e R15. Cabe destacar na simulação do exemplo 4 o grande número de símbolos inseridos na tabela da Figura 87. Isso se deve ao número de testes sobre variáveis implicando em vários saltos condicionais no código.

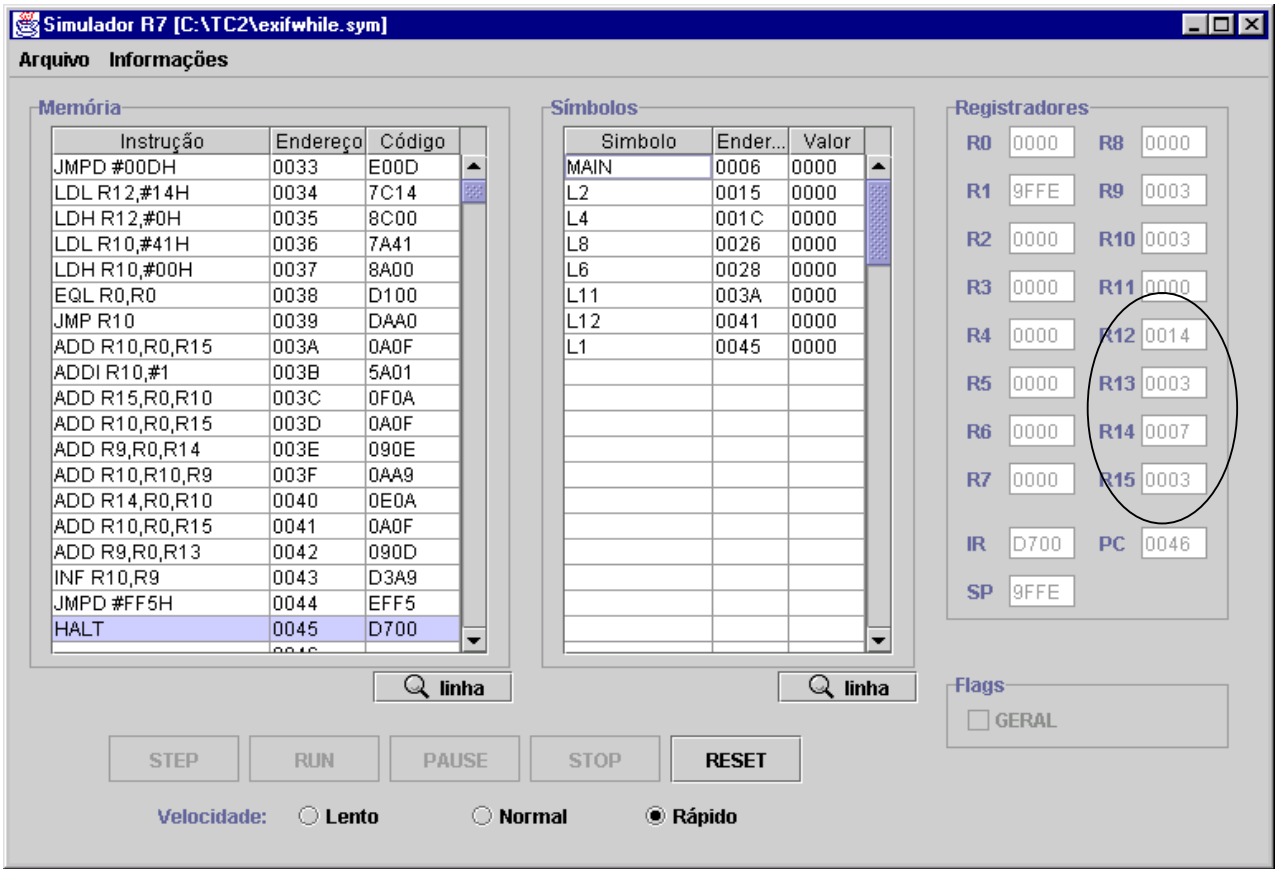

Figura 87 - Simulação para o exemplo 4.

6.1.5 Exemplo 5 – Arquivos: "*exsubrotinas.c*" e "*exsubrotinas.asm*"

O exemplo em questão foi elaborado para testes de implementação de instruções em

assembly para controle e passagem de parâmetros em subrotinas, juntamente com declaração e atribuição realizadas sobre variáveis globais. Foi utilizado o programa mostrado na Figura 88.

```
1.  # include "stdio.h"
2.  # include "stdlib.h"
3.
4.  char gla, gld;
5.  char glc = 11;
6.
7.
8.  char soma(char a, char b, char c)
9.  {
10.    char resub;
11.    resub = a + b + c;
12.    return resub;
13. }
14.
15. void main()
16. {
17.   char a, b, c;
18.   char res;
19.
20.   a = 1;
21.   b = 2;
22.   c = 3;
23.
24.
25.
26.   gla = 10;   /* globais */
27.   gld = 20;
28.   res = soma(a, b, c);
29. }
```

Figura 88 - Exemplo de teste para subrotinas.

Na Figura 89 mostramos o *assembly* gerado para implementação do programa da Figura 88 acima.

```
1.   .CODE
2.   LDL R1,#feH
3.   LDH R1,#9fH
4.   LDSP R1
5.   XOR R0,R0,R0
6.   EQL R0,R0
7.   JMPD main.............................Cabeçalho comum a todos os programas
8.   glc: DB #bH...........................Variável global 'glc'
9.   soma:.................................Subrotina 'soma'
10.  LDL R10, #f3H........................Busca do endereço da variável 'resub' (-13)
11.  LDH R10, #ffH
12.  ADD R9,R0,R4.........................Carga do parâmetro de R4 em R9
13.  ADD R8,R0,R5
14.  ADD R9,R9,R8.........................Adição → a + b
15.  ADD R8,R0,R6
16.  ADD R9,R9,R8.........................Adição → c + (a + b)
17.  ST R9,R0,R10.........................Atribuição → resub = a + b + c na PD[-13]
18.  LDL R10, #f3H........................Busca do endereço de 'resub'
19.  LDH R10, #ffH
20.  LD R2,R0,R10.........................Carga do resultado em R2
21.  L4:
22.  RTS
23.  main:
24.  LDL R10, #ffH........................Busca do endereço de 'a'
25.  LDH R10, #ffH
26.  LDL R9, #1H..........................Carga da constante 1 em R9
27.  LDH R9, #0H
28.  ST R9,R0,R10.........................Atribuição → a = 1, na pilha de dados PD[-1]
29.  LDL R10, #feH........................Busca do endereço de 'b'
30.  LDH R10, #ffH
31.  LDL R9, #2H..........................Carga da constante 2 em R9
32.  LDH R9, #0H
33.  ST R9,R0,R10.........................Atribuição → b = 2, na pilha de dados PD[-2]
34.  LDL R10, #fdH........................Busca do endereço de 'c'
35.  LDH R10, #ffH
36.  LDL R9, #3H
37.  LDH R9, #0H
38.  ST R9,R0,R10.........................Atribuição → c = 3, na pilha de dados PD[-3]
39.  LDL R10, #gla........................Busca do endereço de 'gla'
40.  LDH R10, #gla
41.  LDL R9, #aH
42.  LDH R9, #0H
43.  ST R9,R0,R10.........................Atribuição → gla = 10
44.  LDL R10, #gld........................Busca do endereço de 'gld'
```

```
45.  LDH R10, #gld
46.  LDL R9, #14H
47.  LDH R9, #0H
48.  ST R9,R0,R10.....................Atribuição → gld = 20
49.  LDL R10, #ffH.....................Busca do endereço de 'a'
50.  LDH R10, #ffH
51.  LD R4,R0,R10.....................Carga do conteúdo de 'a' em R4
52.  LDL R10, #feH.....................Busca do endereço de 'b'
53.  LDH R10, #ffH
54.  LD R5,R0,R10.....................Carga do conteúdo de 'b' em R5
55.  LDL R10, #fdH.....................Busca do endereço de 'c'
56.  LDH R10, #ffH
57.  LD R6,R0,R10.....................Carga do conteúdo de 'c' em R6
58.  EQL R0, R0.........................Seta flags para salto
59.  JSRD soma.........................Salto para subrotina
60.  ADD R10,R0,R2.....................Carga do valor de retorno de R2 em R10
61.  LDL R9, #fcH.....................Busca do endereço de 'res'
62.  LDH R9, #ffH
63.  ST R10,R0,R9.....................Atribuição → res = soma(a, b, c), na PD[-4]
64.  L5:
65.  HALT.............................Fim de programa
66.
67.  .ENDCODE
68.  gld: DB #00.....................Variáveis globais
69.  gla: DB #00
```

Figura 89 - *Assembly* gerado para o exemplo 5.

O *assembly* mostrado e comentado da Figura 89 possui a introdução padrão dos programas entre as linhas 1-7. A variável global que recebe valor inicial é inserida no código após a parte *default* de inicialização. As demais variáveis globais são colocadas no final do código. No intervalo das linhas 10-20 estão as instruções que implementam a subrotina 'soma'. Ressalta-se que a adição realizada na subrotina é elaborada apenas com a manipulação de registradores. Já entre as linhas 24-58 estão as instruções que implementam as atribuições simples da Figura 88, tanto em variáveis locais como em variáveis globais.

A implementação de passagem de parâmetros, no exemplo 5, é realizada via registradores como pode-se ver nas instruções inseridas entra as linhas 49 e 57 (R4, R5, R6), e no retorno do resultado através do registrador R2 que é armazenado na variável 'sub' na linha 60.

Na Figura 90 é mostrada a simulação do *assembly* gerado para o exemplo 5 da Figura 89.

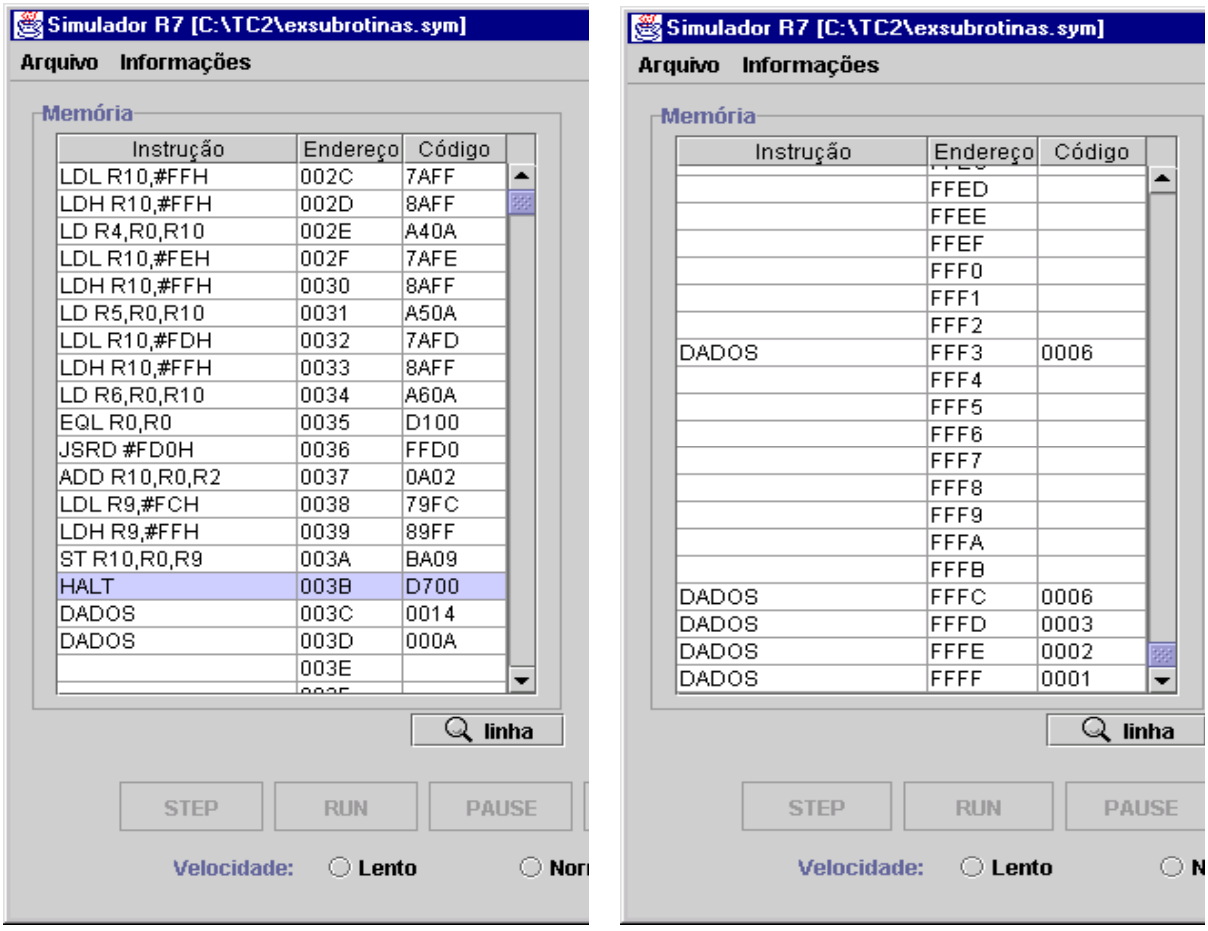

Figura 90 - Simulação do *assembly* gerado para o exemplo 5, com área de memória das variáveis locais.

Na Figura 90 são visualizadas as células de memória onde as variáveis locais estão armazenadas e os resultados finais após a execução do programa mostrado na Figura 89. Os endereços correspondentes são:

a) Variável ‘a’: endereço → FFFF = Conteúdo → 1;

b) Variável ‘b’: endereço → FFFE = Conteúdo → 2;

c) Variável ‘c‘: endereço → FFFD = Conteúdo → 3;

d) Variável ‘resub’: endereço → FFFC = Conteúdo → 6;

e) Variável ‘res’: endereço → FFF3 = Conteúdo → 6.

Destaca-se que os endereços das variáveis são armazenados de 1 em 1 posição tendo em vista o problema já discutido na Seção 5.7.1.

6.1.6 Exemplo 6 – Arquivos: “*exvetores.c*” e “*exvetores.asm*”

Para testes sobre cálculo de endereços na manipulação de vetores foi definido um exemplo que manipula um vetor de 5 posições mostrado na Figura 91. O programa tem a função de atribuir à posição do vetor o seu respectivo número de índice.

```
1.  #include "stdio.h"
2.  #include "stdlib.h"
3.
4.  void main()
5.  {
6.    char vet[5];
7.    char i;
8.
9.    for(i=0; i<5; i++)
10.   {
11.      vet[i] = i;
12.   }
13. }
```

Figura 91 - Código para testes de ponteiros.

Na Figura 92 mostramos o *assembly* gerado para o exemplo 6 cujo código é mostrado na Figura 91.

```
1.   .CODE
2.   LDL R1,#feH
3.   LDH R1,#9fH
4.   LDSP R1
5.   XOR R0,R0,R0
6.   EQL R0,R0
7.   JMPD main.............................Cabeçalho comum a todos os programas
8.   main:
9.   ADD R15,R0,R0.........................Carga de R15 com zero para iterações do laço for
10.  LDL R10, #L8
11.  LDH R10, #L8
12.  EQL R0, R0
13.  JMP R10..............................Salta para L8
14.  L5:
15.  ADD R10,R0,R15.......................Carga de R10 com o conteúdo de R15 (R10 ← i)
16.  LDL R9, #fbH.........................Busca do endereço de 'vet'
17.  LDH R9, #ffH
18.  ADD R10,R10,R9.......................Cálculo do endereço de vet[i]
19.  ST R15,R0,R10........................Atribuição → vet[i] = i
20.  L6:
21.  ADD R10,R0,R15.......................Carga de R10 com conteúdo de R15
22.  ADDI R10,#1...........................Incremento → i++
23.  ADD R15,R0,R10.......................Atualização de R15
24.  L8:
25.  ADD R10,R0,R15
26.  LDL R9, #5H...........................Carga da constante 5 em R9
27.  LDH R9, #0H
28.  INF R10,R9............................Teste → i < 5
29.  JMPD L5..............................Se verdadeiro salta para L5
30.  L4:
31.  HALT.................................Fim de programa
32.
33.  .ENDCODE
```

Figura 92 - *Assembly* gerado para o exemplo 6.

O *assembly* da Figura 92 possui as mesmas características já mostradas anteriormente nos outros exemplos, tendo em vista que o conjunto de instruções utilizados para implementação de atribuições, incremento de variáveis e testes de salto no fluxo do programa é o mesmo em todas as aplicações. Na Figura 93 mostramos a simulação para validação do código *assembly* para o exemplo 6.

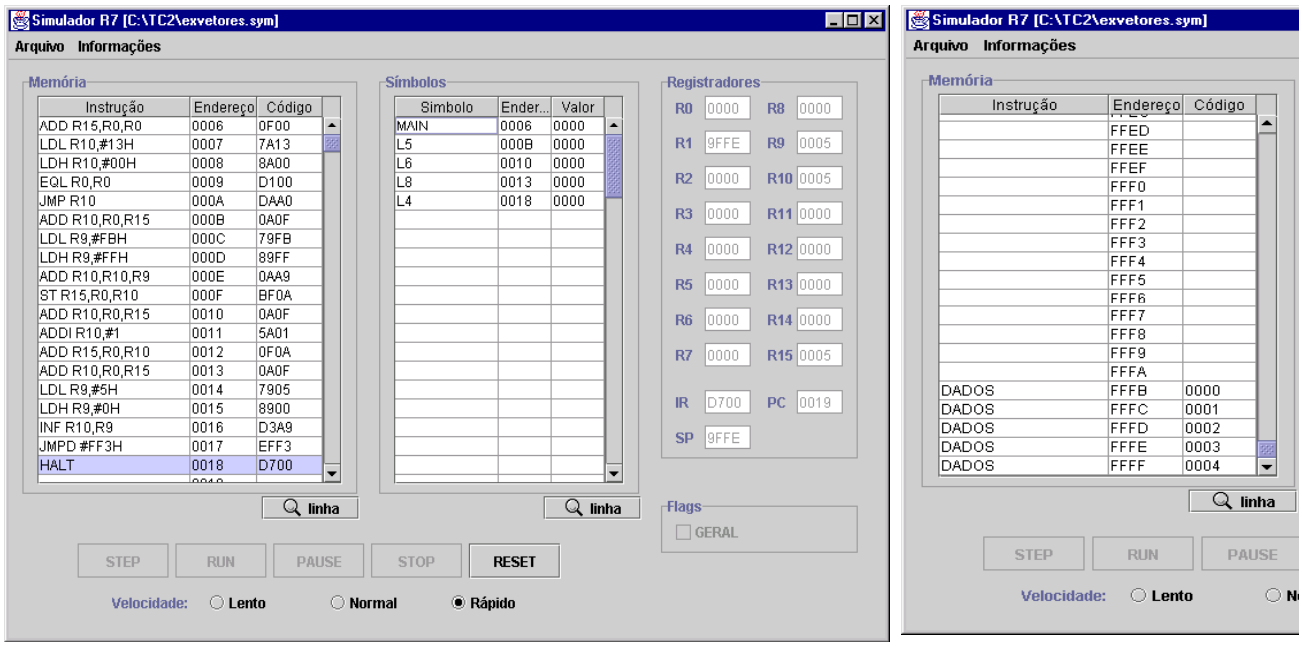

Figura 93 - Simulação do exemplo 6.

Um ponto de destaque na simulação para o caso do exemplo 6 é a área de memória com os resultados da composição do vetor após a execução do programa. Esses resultados são mostrados na janela da direita da Figura 93. Vale ressaltar que o resultado final do programa é a atribuição da variável '*i*' à posição '*i*' do vetor '*vet*' declarado, no exemplo, com 5 posições, fazendo com que o vetor seja preenchido com os valores 0, 1, 2, 3 e 4.

6.1.7 Exemplo 7 – Arquivos: "*exmult.c*" e "*exmult.asm*"

Para testes sobre passagem de parâmetros por referência foi definido o exemplo 7 com o programa mostrado na Figura 94.

```
1.  void mult(char a, char b,char *ml) {
2.     char i;
3.     for( i = *ml = 0;  i<b; i++ ){
4.          *ml = (*ml) + a;
5.       }
6.  }
7.
8.  void main(void){
9.
10.    char a = 2;
11.    char b = 5;
12.    char ml = 0;
13.    mult(a, b, &ml);
14. }
```

Figura 94 - Programa para testes sobre ponteiros.

Para o exemplo 7 foi gerado o código em *assembly* da Figura 95.

```
1.    .CODE
2.    LDL R1,#feH
3.    LDH R1,#9fH
4.    LDSP R1
5.    XOR R0,R0,R0
6.    EQL R0,R0
7.    JMPD main.....................................Cabeçalho comum a todos os programas
8.
```

```
9.   mult:.....................................Subrotina `mult`
10.  ADD R10,R0,R0.............................Carga de R10 com zero (i)
11.  ST R10,R0,R6..............................ml ← 0
12.  ADD R15,R0,R10............................Carga de R15 com zero
13.  LDL R10, #L5
14.  LDH R10, #L5
15.  EQL R0, R0
16.  JMP R10.....................................Salto para L5
17.  L2:
18.  LD R10,R0,R6..............................R10 ← &ml
19.  ADD R9,R0,R4..............................R9 ← a
20.  ADD R10,R10,R9............................R10 ← a + ml
21.  ST R10,R0,R6..............................*ml ← R10
22.  L3:
23.  ADD R10,R0,R15............................R10 ← R15
24.  ADDI R10,#1...............................i++
25.  ADD R15,R0,R10............................R15 ← R10
26.  L5:
27.  ADD R10,R0,R15............................Carga de R10 em R15 (i)
28.  ADD R9,R0,R5..............................Carga de 'b' em R9
29.  INF R10,R9.................................Teste → i < b
30.  JMPD L2.....................................Se verdadeiro salta para L2
31.  L1:
32.  RTS........................................Fim da subrotina
33.  main:.....................................Programa principal
34.  LDL R10, #ffH.............................Busca endereço de 'a'
35.  LDH R10, #ffH
36.  LDL R9, #2H................................R9 ← 2
37.  LDH R9, #0H
38.  ST R9,R0,R10...............................a ← 2
39.  LDL R10, #feH.............................Busca endereço de b
40.  LDH R10, #ffH
41.  LDL R9, #5H................................Carga da constante 5 em R9
42.  LDH R9, #0H
43.  ST R9,R0,R10...............................b ← 5
44.  LDL R10, #fdH.............................Busca do endereço de 'ml'
45.  LDH R10, #ffH
46.  ST R0,R0,R10...............................ml ← 0
47.  LDL R10, #ffH.............................Busca endereço de 'a'
48.  LDH R10, #ffH
49.  LD R4,R0,R10...............................Carga do conteúdo de 'a' em R4
50.  LDL R10, #feH.............................Busca do endereço de 'b'
51.  LDH R10, #ffH
52.  LD R5,R0,R10...............................Carga do conteúdo de 'b' em R5
53.  LDL R6, #fdH..............................Busca do endereço de 'ml'
54.  LDH R6, #ffH
55.  EQL R0, R0
56.  JSRD mult...................................Salto para subrotina 'mult'
57.  L6:
58.  HALT.......................................Fim de programa
59.
60.  .ENDCODE
```

Figura 95 - *Assembly* gerado para o exemplo 7.

O código em *assembly* da Figura 95 possui o cabeçalho comum a todos os programas mostrado no intervalo 1-7. Já entre as linhas 9-30 estão as instruções que implementam a subrotina 'mult'. Cabe ressaltar que a passagem de parâmetros é realizada através dos registradores R4, R5 e R6. Ressaltando que a referência da variável 'ml' é colocada no R6.

A seguir é mostrada a Figura 96 que representa a simulação para o exemplo 7.

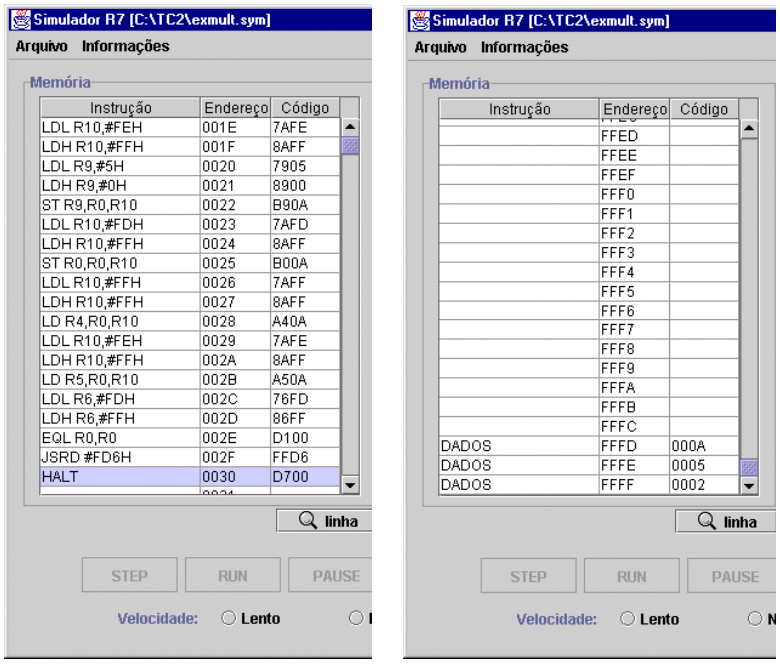

Figura 96 - Simulação para o exemplo 7.

O resultado da operação de multiplicação realizada pode ser visto na variável alocada no endereço FFFD, que contém o valor em hexadecimal 000A (10 em decimal). Já as variáveis locais são armazenadas nos endereços FFFE e FFFF, onde estão os números que são multiplicados.

6.1.8 Exemplo 8 – Arquivos: "*tbubble.c*" e "*tbubble.asm*"

Este exemplo completa a série de testes. A Figura 97 mostra o código em linguagem C para implementação do algoritmo de ordenação *bubble sort*. Destaca-se que o tipo *char* utilizado tem comportamento semelhante, no processador R7, ao do tipo *int* tendo em vista as restrições comentadas na Seção 5.7.1.

```
1.  #include "stdio.h"
2.  #include "stdlib.h"
3.
4.  char vgl[10] = {23,18,12,7,45,9,3,6,1,0};
5.
6.
7.  void bubble(char vgl[], char size)
8.  {
9.     char i,j,aux;
10.    for (i = 0; i < size; i++)
11.    {
12.       for (j = 0; j < size; j++)
13.       {
14.         if (vgl[i] < vgl[j])
15.          {aux = vgl[i];
16.           vgl[i] = vgl[j];
17.           vgl[j] = aux;
18.          }
19.       }
20.    }
21. }
22.
23. void main()
24. {
25.   char vsize;
26.   vsize = 10;
27.   bubble(vgl, vsize);
28. }
```

Figura 97 - Código de implementação do algoritmo *bubble sort.*

No código mostrado na Figura 97 foi utilizado um vetor global visando a visualização do resultado da aplicação do algoritmo de ordenação sobre o conjunto de números armazenados no vetor.

Na Figura 98 é mostrado o *assembly* gerado para o exemplo da Figura 97.

```
1.  .CODE
2.  LDL R1,#feH
3.  LDH R1,#9fH
4.  LDSP R1
5.  XOR R0,R0,R0
6.  EQL R0,R0
7.  JMPD main.............................Cabeçalho comum a todos os programas
8.  vgl: DB #17H, #12H, #cH, #7H, #2dH, #9H, #3H, #6H, #1H, #0H...Vetor global
9.
10. bubble:...............................Subrotina bubble
11. ADD R14,R0,R0.........................Carga de zero em R14 para iterações do 'i' no for
12. LDL R10, #L8
13. LDH R10, #L8
14. EQL R0, R0
15. JMP R10...............................Salta para L8
16. L5:
17. ADD R15,R0,R0.........................Carga de zero em R15 para iterações do 'j' no for
18. LDL R10, #L12
19. LDH R10, #L12
20. EQL R0, R0
21. JMP R10...............................Salta para L12
22. L9:
23. ADD R10,R0,R14........................Carga de R14 em R10
24. ADD R10,R10,R4........................Cálculo de endereço → vgl + i
25. LD R10,R0,R10.........................Carga do conteúdo do endereço (vgl + i) em R10
26. ADD R9,R0,R15
27. ADD R9,R9,R4..........................Cálculo de endereço → vgl + j
28. LD R9,R0,R9...........................Carga do conteúdo do endereço (vgl + j) em R9
29. SUP R10,R9............................Teste → vgl[j] > vgl[i]
30. JMPD L13..............................Se verdadeiro salta para L13
31. EQL R10,R9............................Teste → vgl[i] == vgl[j]
32. JMPD L13..............................Se verdadeiro salta para L13
33. ADD R10,R0,R14
34. ADD R10,R10,R4........................Cálculo de endereço → vgl + i
35. LD R13,R0,R10.........................Carga do conteúdo do endereço (vgl + i) em R13
36. ADD R9,R0,R15
37. ADD R9,R9,R4..........................Cálculo de endereço → vgl + j
38. LD R9,R0,R9...........................Carga do conteúdo do endereço (vgl + j) em R8
39. ST R9,R0,R10..........................Atribuição → vgl[i] = vgl[j]
40. ADD R10,R0,R15
41. ADD R10,R10,R4........................Cálculo de endereço → vgl + j
42. ST R13,R0,R10.........................Atribuição → vgl[j] = vgl[i]
43. L13:
44. L10:
45. ADD R10,R0,R15
46. ADDI R10,#1...........................Incremento → j++
47. ADD R15,R0,R10........................Atualização de R15
48. L12:
49. ADD R10,R0,R15
50. ADD R9,R0,R5
51. INF R10,R9............................Teste → j < vsize
52. JMPD L9...............................Se verdadeiro salta para L9
53. L6:
54. ADD R10,R0,R14
55. ADDI R10,#1...........................Incremento → i++
56. ADD R14,R0,R10........................Atualização de R14
57. L8:
58. ADD R10,R0,R14
59. ADD R9,R0,R5
60. INF R10,R9............................Teste → i < vsize
61. JMPD L5...............................Se verdadeirao salta para L5
62. L4:
63. RTS
64. main:.................................Programa principal
65. LDL R10, #ffH.........................Busca do endereço de 'vsize'
66. LDH R10, #ffH
67. LDL R9, #aH...........................Carga da constante 10 em R9
68. LDH R9, #0H
69. ST R9,R0,R10..........................Atribuição → vsize
70. LDL R4, #vgl..........................Busca do endereço de 'vgl'
71. LDH R4, #vgl
72. LDL R10, #ffH.........................Busca do endereço de 'vsize'
73. LDH R10, #ffH
74. LD R5,R0,R10..........................Carga do conteúdo de 'vsize' em R9
75. EQL R0, R0
76. JSRD bubble...........................Salto para subrotina
77. L15:
78. HALT.................................Fim de programa
79.
80. .ENDCODE
```

Figura 98 - *Assembly* gerado para o exemplo 6.

Para validação do exemplo, mostramos na Figura 99 o vetor antes de ser ordenado pelo algoritmo *bubble sort*, e o resultado da ordenação.

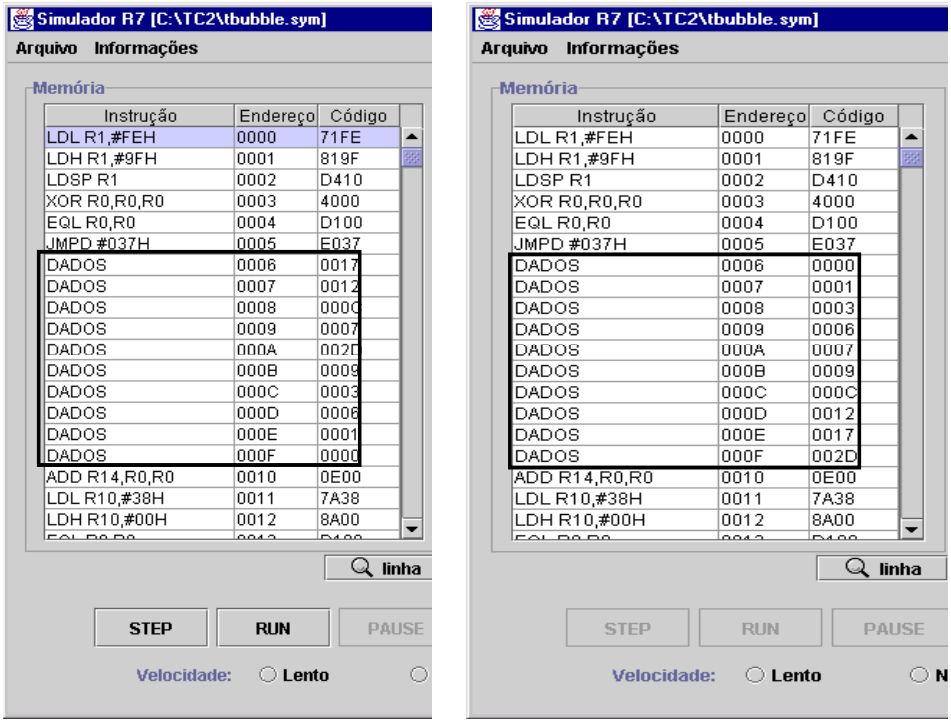

Figura 99 - Simulação com o vetor inicial e o vetor resultado da ordenação.

Vale ressaltar na simulação também a área de memória que guarda o valor da variável *vsize*, mostrada na Figura 100.

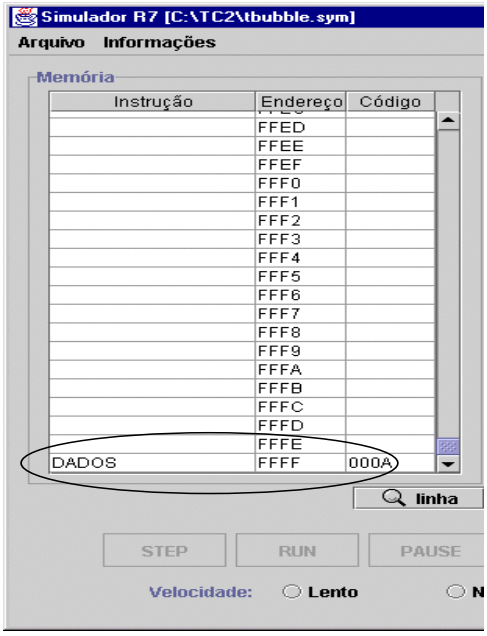

Figura 100 - Célula de memória da variável *vsize*.

Após a validação de todos os exemplos, são expostas no próximo Capítulo as

conclusões sobre a execução do projeto, algumas dificuldades encontradas e as restrições restantes no gerador de código para a arquitetura R7.

7. CONCLUSÕES

A descrição das características de um determinado processador para o compilador não é uma tarefa simples de ser elaborada. Para que a descrição seja realizada com a fidelidade necessária e que o código em *assembly* gerado seja consistente, é essencial uma considerável base de conhecimento na área de construção e implementação de compiladores. Além desse conhecimento é essencial dominar, na sua totalidade, as características do processador a ser descrito. Sem esses conhecimentos torna-se mais árdua a elaboração da descrição para o compilador.

A seguir são citadas as **dificuldades** encontradas no decorrer do projeto:

a) O compilador LCC não foi projetado para geração de *assembly* em processadores *word-adressed* [31], o que é o caso do processador R7, que é endereçado a palavras de 16 *bits*. Esse fato tem implicações no armazenamento de variáveis em memória (ver Seção 5.7);

b) A cada alteração do arquivo *md* que contém a descrição do processador R7, é necessário uma nova compilação dos arquivos-fontes do LCC, visando a geração do gerador de código para aquele processador. Isso toma demasiada fatia de tempo do projeto, pois as alterações necessitavam ser de pequeno porte sob pena de distorcer a análise do *assembly* gerado. A dificuldade na execução da compilação foi minimizada através da construção de *scripts*;

c) O arquivo *md* escolhido como modelo (*mips.md*) possui uma série de funções (*interface funtcions* - Seção 3.3.3) que realizam o tratamento para determinadas instruções e definições de trechos de código em linguagem C. Isso torna necessário conhecer os detalhes e as implicações sobre a possível retirada ou alteração de linhas de código que estas funções possuem. Esse fato requer o estudo das estruturas utilizadas pelas funções e suas definições;

d) Quando da descrição das instruções do código *assembly*, é necessário a compreensão sobre o caminhamento realizado pelo *back end* sobre as estruturas geradas pelo *front end* (florestas de *DAGs* [31]) e sobre a gramática de descrição das instruções do processador modelo (MIPS). Essa dificuldade foi minimizada através da numeração das regras da gramática de descrição do processador MIPS (base para descrição);

e) Forma de utilização dos símbolos “\n”, “%letra”, “%dígito”, “?” e “#”. Conforme o grupo foi conhecendo mais a respeito do compilador (através da documentação) e (paralelamente) trabalhando na construção do gerador de código para o processador R7, a função de cada um dos operadores foi ficando mais clara. Mas é importante que se saiba que a utilização incorreta destes dificultou o andamento do processo de construção da descrição do processador-alvo. Desta forma, considera-se de suma importância a compreensão do significado e função de cada

uma destes operadores, para que assim se efetue satisfatoriamente a construção deste arquivo.

Apesar da validação do *assembly* através de diversos programas de teste, a descrição do processador R7 para o compilador LCC possui algumas limitações. Essas limitações são resultado de definições utilizadas na descrição do processador R7, devido as suas características, e também de restrições do compilador LCC.

Abaixo listamos as **restrições** do gerador de código para o processador R7:

a) O gerador de código pode tratar tipos *int*, contudo os endereços de memória serão calculados de 4 em 4 posições (ver Seção 5.7.1);
b) O gerador de código não contempla passagem de parâmetros em funções com mais de 5 parâmetros;
c) O gerador de código não contempla tratamento de chamadas aninhadas de funções tendo em vista que a passagem de parâmetros é realizada via registradores;
d) O gerador de código não contempla tratamento de tipos de dados compostos (*struct*).

As **contribuições** deste trabalho são:

a) disponibilizar não apenas um compilador C para uma dada arquitetura alvo, mas todo o processo de compreender como é descrito um processador em um compilador configurável e a respectiva documentação deste processo, visando adaptá-lo a outros processadores.
b) disponibilizar uma ferramenta para ensino (disciplina de Compiladores) e pesquisa (processadores embarcados).

8. REFERÊNCIAS BIBLIOGRÁFICAS

[1] A. M. MARCON, C. - Modelos, Ferramentas e Métodos para o Projeto Integrado de Hardware e Software. Trabalho Individual I. Pontifícia Universidade Católica do Rio Grande do Sul, Brazil (Ago. 2000)

[2] A. PATTERSON, D., L. HENNESSY, J. – Organização e projeto de Computadores, LTC editora, 540 p.

[3] BARROS, E. S. – Hardware/software Partitioning Using Unity. Ph. D. thesis Tübingen University, Germany, 1993.

[4] BERRY, G. – The ESTEREL v5_21 System Manual, Available at homepage: http://www.esterel.org. INRIA, 1999.

[5] DE MICHELI, G. - Hardware/Software Codesign: Application Domains and Design Technologies. Proceedings of the NATO Advanced Study Institute in Hardware/Software Co-Design, Kluwer Academic Publishers, Italy, 1995.

[6] DUTTA , S. - http://comp.ajou.ac.kr/~kan/html/sdcc/SDCCUdoc.html

[7] DUTTA, S. – http://www.execpc.com/~sdutt00/docs/sd_tmpl.html

[8] F. MOLZ, R., M. ENGEL, P., G. MORAES, F., TORRES, L., ROBERT, M. - Codesign of Fully Parallel Neural Network for a Classification Problem. SCI 2000.

[9] FARINES, J., S. FRAGA, J., S. OLIVEIRA, R. – Sistemas de Tempo Real. São Paulo, IME-SP, 2000. 201 p.

[10] G. MORAES, F., V. CALAZANS, N., M. MARCON, C., V. MELLO, A. – Um Ambiente de Compilação e Simulação para Processadores Embarcados Parametrizáveis. VII Workshop Iberchip – Montevideo, Uruguai - 2001

[11] G. MORAES, F., V. CALAZANS, N., V. MELLO, A. – Ambiente de Desenvolvimento de Processador Embarcado para Aplicações de Codesign. SCR – Seminário de Computação Reconfigurável – 2001.

[12] GNU is not Unix ! – http://www.gnu.org

[13] http://www.whatis.com

[14] JONES, G. - Programming in OCCAM 2. New York, NY. Prentice-Hall, 1989.

[15] LAVAGNO, L., SANGIOVANNI-VICENTELLI A. and HSIEH H. - Embedded System Codesign: Synthesis and Verification. Proceedings of the NATO Advanced Study Institute in Hardware/Software Co-Design, Kluwer Academic Publishers, City Italy, 1995.

[16] MAZOR, S., LANGSTRAAT, P. - A Guide to VHDL. Kluwer Academic Publishers, Norwell, MA, (1992).

[17] O GCC do Linux - http://www.tarcisiolopes.com/gcc.htm

[18] OCCAM 2 : Reference Manual. New York, NY. Prentice-Hall, 1993.

[19] Página da disciplina de Organização de Computadores – www.inf.pucrs.br/~moraes

[20] POUNTAIN, D. - A Tutorial Introduction to Occam Programming. Colorado. INMOS, 1986.

[21] R. BERGAMASCHI, A., R. LEE, W. - Designing Systems-on-Chip Using Cores. Design Automation Conference, Los Angeles, California, USA, (2000).

[22] R. LEVINE, J., MASON., T., BROWN, D. – Lex & Yacc. O'Reilly & Associates, Inc. (Oct. 1992)

[23] RINCON, A. M. et. al. Core Design and System-on-a-chip Integration. IEEE Design & Test of Computers, p. 26-35, (Oct-Dec 1997).

[24] ROSE, J., VINCENTELLI, A. S. - Architecture of Field-Programmable Gate Arrays. Proceedings of the IEEE, vol. 81 no. 7, (Jul 1993).

[25] SDCC – Small Device C Compiler – http://sdcc.sourceforge.net

[26] SLOMKA, F., DORFEL, M., MUNZENBERGER, R. and HOFMANN, R. – Hardware/software Codesign and Rapid Prototyping of Embedded Systems. IEEE Design & Test of Computers, April 2000.

[27] THOMAS, D., MOORBY, P. - The Verilog Hardware Description Language. Kluver Academic Publishers, (1991).

[28] Using and Porting the GNU Compiler Collection (GCC) - http://gcc.gnu.org/onlinedocs/gcc.html

[29] V. AHO, A., SETHI, R., D. ULLMAN, J. – Compiladores – Princípios, Técnicas e Ferramentas. Tradução – Daniel de Ariosto Pinto. Editora Guanabara Koogan S/A – 1995.

[30] W. FRASER, C., R. HANSON, D. – A Code Generation Interface for ANSI C. Software - Pratice and Experience, Vol. 21 (9), 963 – 988 (Sept 1991)

[31] W. FRASER, C., R. HANSON, D. – A Retargetable C Compiler: Design and Implementation. Adison –Wesley Publishing Company (1995).

[32] W. FRASER, C., R. HANSON, D. – A Retargetable Compiler for ANSI C. SIGPLAN Notices 26, 10 (Oct. 1991).

[33] W. FRASER, C., R. HANSON, D., A. PROEBSTING, T. – Engineering a Simple, Efficient Code Generator Generator. ACM Letters on Programming Languages and Systems 1, 3 (Sep. 1992), 213-226.

[34] W. FRASER, C., R. HENRY, R., A. PROEBSTING, T. – BURG – Fast Optimal Instruction Selection and Tree Parsing. (Dec. 1991)

9. ANEXO

```
1.  %{
2.  // R0 ------------------------- cte 0
3.  // R1 .. R2 ------------------- retorno de subrotina *
4.  // R3 .. R5 ------------------- argumentos
5.  // R6 .. R10 ------------------ registradores de uso geral *
6.  // R11 .. R15 ----------------- variaveis register *
7.
8.
9.
10. #define INTVAR 0xF800 /* 11 - 15 : variaveis "register" */
11. #define INTTMP 0x07C0 /* 6 - 10 : regs. de uso geral uso geral  */
12. #define INTRET 0x0006 /* 1 - 2 : retorno de subrotina */
13.
14.
15. #define readsreg(p) \
16.         (generic((p)->op)==INDIR && (p)->kids[0]->op==VREG+P)
17. #define setsrc(d) ((d) && (d)->x.regnode && \
18.         (d)->x.regnode->set == src->x.regnode->set && \
19.         (d)->x.regnode->mask&src->x.regnode->mask)
20.
21. #define relink(a, b) ((b)->x.prev = (a), (a)->x.next = (b))
22.
23. #include "c.h"
24. #define NODEPTR_TYPE Node
25. #define OP_LABEL(p) ((p)->op)
26. #define LEFT_CHILD(p) ((p)->kids[0])
27. #define RIGHT_CHILD(p) ((p)->kids[1])
28. #define STATE_LABEL(p) ((p)->x.state)
29. static void address(Symbol, Symbol, long);
30. static void blkfetch(int, int, int, int);
31. static void blkloop(int, int, int, int, int, int[]);
32. static void blkstore(int, int, int, int);
33. static void defaddress(Symbol);
34. static void defconst(int, int, Value);
35. static void defstring(int, char *);
36. static void defsymbol(Symbol);
37. static void doarg(Node);
38. static void emit2(Node);
39. static void export(Symbol);
40. static void clobber(Node);
41. static void function(Symbol, Symbol [], Symbol [], int);
42. static void global(Symbol);
43. static void import(Symbol);
44. static void local(Symbol);
45. static void progbeg(int, char **);
46. static void progend(void);
47. static void segment(int);
48. static void space(int);
49. static void target(Node);
50. static Symbol   argreg         (int, int, int, int, int);
51.
52. static Symbol ireg[15], freg2[32], d6;
53. static Symbol iregw, freg2w;
54. static Symbol blkreg;
55.
56. int flagMult = 0; // TC
57. int flagDiv = 0; // TC
58. int flagShiftLeft = 0; // TC
59. int flagShiftRigth = 0; // TC
60. int flagResto = 0; // TC
61. int tamanho = 0; // TC
62. int contador; // TC
63. int flag_printf = 0;
64. %}
65. %start stmt
66. %term CNSTF4=4113
67. %term CNSTF8=8209
68. %term CNSTF16=16401
69. %term CNSTI1=1045
70. %term CNSTI2=2069
71. %term CNSTI4=4117
72. %term CNSTI8=8213
73. %term CNSTP4=4119
74. %term CNSTP8=8215
75. %term CNSTU1=1046
76. %term CNSTU2=2070
77. %term CNSTU4=4118
78. %term CNSTU8=8214
```

```
79.
80. %term ARGB=41
81. %term ARGF4=4129
82. %term ARGF8=8225
83. %term ARGF16=16417
84. %term ARGI4=4133
85. %term ARGI8=8229
86. %term ARGP4=4135
87. %term ARGP8=8231
88. %term ARGU4=4134
89. %term ARGU8=8230
90.
91. %term ASGNB=57
92. %term ASGNF4=4145
93. %term ASGNF8=8241
94. %term ASGNF16=16433
95. %term ASGNI1=1077
96. %term ASGNI2=2101
97. %term ASGNI4=4149
98. %term ASGNI8=8245
99. %term ASGNP4=4151
100.%term ASGNP8=8247
101.%term ASGNU1=1078
102.%term ASGNU2=2102
103.%term ASGNU4=4150
104.%term ASGNU8=8246
105.
106.%term INDIRB=73
107.%term INDIRF4=4161
108.%term INDIRF8=8257
109.%term INDIRF16=16449
110.%term INDIRI1=1093
111.%term INDIRI2=2117
112.%term INDIRI4=4165
113.%term INDIRI8=8261
114.%term INDIRP4=4167
115.%term INDIRP8=8263
116.%term INDIRU1=1094
117.%term INDIRU2=2118
118.%term INDIRU4=4166
119.%term INDIRU8=8262
120.
121.%term CVFF4=4209
122.%term CVFF8=8305
123.%term CVFF16=16497
124.%term CVFI4=4213
125.%term CVFI8=8309
126.
127.%term CVIF4=4225
128.%term CVIF8=8321
129.%term CVIF16=16513
130.%term CVII1=1157
131.%term CVII2=2181
132.%term CVII4=4229
133.%term CVII8=8325
134.%term CVIU1=1158
135.%term CVIU2=2182
136.%term CVIU4=4230
137.%term CVIU8=8326
138.
139.%term CVPP4=4247
140.%term CVPP8=8343
141.%term CVPP16=16535
142.%term CVPU4=4246
143.%term CVPU8=8342
144.
145.%term CVUI1=1205
146.%term CVUI2=2229
147.%term CVUI4=4277
148.%term CVUI8=8373
149.%term CVUP4=4279
150.%term CVUP8=8375
151.%term CVUP16=16567
152.%term CVUU1=1206
153.%term CVUU2=2230
154.%term CVUU4=4278
155.%term CVUU8=8374
156.
157.%term NEGF4=4289
158.%term NEGF8=8385
159.%term NEGF16=16577
160.%term NEGI4=4293
161.%term NEGI8=8389
162.
163.%term CALLB=217
164.%term CALLF4=4305
165.%term CALLF8=8401
166.%term CALLF16=16593
167.%term CALLI4=4309
168.%term CALLI8=8405
169.%term CALLP4=4311
```

```
170. %term CALLP8=8407
171. %term CALLU4=4310
172. %term CALLU8=8406
173. %term CALLV=216
174.
175. %term RETF4=4337
176. %term RETF8=8433
177. %term RETF16=16625
178. %term RETI4=4341
179. %term RETI8=8437
180. %term RETP4=4343
181. %term RETP8=8439
182. %term RETU4=4342
183. %term RETU8=8438
184. %term RETV=248
185.
186. %term ADDRGP4=4359
187. %term ADDRGP8=8455
188.
189. %term ADDRFP4=4375
190. %term ADDRFP8=8471
191.
192. %term ADDRLP4=4391
193. %term ADDRLP8=8487
194.
195. %term ADDF4=4401
196. %term ADDF8=8497
197. %term ADDF16=16689
198. %term ADDI4=4405
199. %term ADDI8=8501
200. %term ADDP4=4407
201. %term ADDP8=8503
202. %term ADDU4=4406
203. %term ADDU8=8502
204.
205. %term SUBF4=4417
206. %term SUBF8=8513
207. %term SUBF16=16705
208. %term SUBI4=4421
209. %term SUBI8=8517
210. %term SUBP4=4423
211. %term SUBP8=8519
212. %term SUBU4=4422
213. %term SUBU8=8518
214.
215. %term LSHI4=4437
216. %term LSHI8=8533
217. %term LSHU4=4438
218. %term LSHU8=8534
219.
220. %term MODI4=4453
221. %term MODI8=8549
222. %term MODU4=4454
223. %term MODU8=8550
224.
225. %term RSHI4=4469
226. %term RSHI8=8565
227. %term RSHU4=4470
228. %term RSHU8=8566
229.
230. %term BANDI4=4485
231. %term BANDI8=8581
232. %term BANDU4=4486
233. %term BANDU8=8582
234.
235. %term BCOMI4=4501
236. %term BCOMI8=8597
237. %term BCOMU4=4502
238. %term BCOMU8=8598
239.
240. %term BORI4=4517
241. %term BORI8=8613
242. %term BORU4=4518
243. %term BORU8=8614
244.
245. %term BXORI4=4533
246. %term BXORI8=8629
247. %term BXORU4=4534
248. %term BXORU8=8630
249.
250. %term DIVF4=4545
251. %term DIVF8=8641
252. %term DIVF16=16833
253. %term DIVI4=4549
254. %term DIVI8=8645
255. %term DIVU4=4550
256. %term DIVU8=8646
257.
258. %term MULF4=4561
259. %term MULF8=8657
260. %term MULF16=16849
```

```
261.%term MULI4=4565
262.%term MULI8=8661
263.%term MULU4=4566
264.%term MULU8=8662
265.
266.%term EQF4=4577
267.%term EQF8=8673
268.%term EQF16=16865
269.%term EQI4=4581
270.%term EQI8=8677
271.%term EQU4=4582
272.%term EQU8=8678
273.
274.%term GEF4=4593
275.%term GEF8=8689
276.%term GEI4=4597
277.%term GEI8=8693
278.%term GEI16=16885
279.%term GEU4=4598
280.%term GEU8=8694
281.
282.%term GTF4=4609
283.%term GTF8=8705
284.%term GTF16=16897
285.%term GTI4=4613
286.%term GTI8=8709
287.%term GTU4=4614
288.%term GTU8=8710
289.
290.%term LEF4=4625
291.%term LEF8=8721
292.%term LEF16=16913
293.%term LEI4=4629
294.%term LEI8=8725
295.%term LEU4=4630
296.%term LEU8=8726
297.
298.%term LTF4=4641
299.%term LTF8=8737
300.%term LTF16=16929
301.%term LTI4=4645
302.%term LTI8=8741
303.%term LTU4=4646
304.%term LTU8=8742
305.
306.%term NEF4=4657
307.%term NEF8=8753
308.%term NEF16=16945
309.%term NEI4=4661
310.%term NEI8=8757
311.%term NEU4=4662
312.%term NEU8=8758
313.
314.%term JUMPV=584
315.
316.%term LABELV=600
317.
318.%term LOADB=233
319.%term LOADF4=4321
320.%term LOADF8=8417
321.%term LOADF16=16609
322.%term LOADI1=1253
323.%term LOADI2=2277
324.%term LOADI4=4325
325.%term LOADI8=8421
326.%term LOADP4=4327
327.%term LOADP8=8423
328.%term LOADU1=1254
329.%term LOADU2=2278
330.%term LOADU4=4326
331.%term LOADU8=8422
332.
333.%term VREGP=711
334.%%
335.reg:  INDIRI1(VREGP)     "# read register\n"
336.reg:  INDIRU1(VREGP)     "# read register\n"
337.
338.reg:  INDIRI2(VREGP)     "# read register\n"
339.reg:  INDIRU2(VREGP)     "# read register\n"
340.
341.reg:  INDIRI4(VREGP)     "#read register\n"
342.reg:  INDIRP4(VREGP)     "# read register\n"
343.reg:  INDIRU4(VREGP)     "# read register\n"
344.
345.reg:  INDIRI8(VREGP)     "# read register\n"
346.reg:  INDIRP8(VREGP)     "# read register\n"
347.reg:  INDIRU8(VREGP)     "# read register\n"
348.
349.stmt: ASGNI1(VREGP,reg)  "# write register\n"
350.stmt: ASGNU1(VREGP,reg)  "# write register\n"
351.
```

```
352. stmt: ASGNI2(VREGP,reg)  "# write register\n"
353. stmt: ASGNU2(VREGP,reg)  "# write register\n"
354.
355.
356. stmt: ASGNI4(VREGP,reg)  "# write register\n"
357. stmt: ASGNP4(VREGP,reg)  "# write register\n"
358. stmt: ASGNU4(VREGP,reg)  "# write register\n"
359.
360. stmt: ASGNI8(VREGP,reg)  "# write register\n"
361. stmt: ASGNP8(VREGP,reg)  "# write register\n"
362. stmt: ASGNU8(VREGP,reg)  "# write register\n"
363.
364.
365. con: CNSTI1  "%a"
366. con: CNSTU1  "%a"
367.
368. con: CNSTI2  "%a"
369. con: CNSTU2  "%a"
370.
371. con: CNSTI4  "%a"
372. con: CNSTU4  "%a"
373. con: CNSTP4  "%a"
374.
375. con: CNSTI8  "%a"
376. con: CNSTU8  "%a"
377. con: CNSTP8  "%a"
378.
379. stmt: reg  ""
380.
381. reg: ADDRGP4  "LDL R%c, #%a \nLDH R%c, #%a\n"  1
382. addr: reg "%0"
383.
384. addr: ADDRFP4  "#LDI R%c,%a \n"
385. addr: ADDRLP4  "#LDI R%c,%a \n"
386.
387. reg: con "#LDI R%c,%0\n"  1
388. reg: addr  "%0"  1
389.
390.
391. reg: CNSTI1  "#\n"  range(a, 0, 0)
392. reg: CNSTI2  "#\n"  range(a, 0, 0)
393. reg: CNSTI4  "#\n"  range(a, 0, 0)
394. reg: CNSTU1  "#\n"  range(a, 0, 0)
395. reg: CNSTU2  "#\n"  range(a, 0, 0)
396. reg: CNSTU4  "#\n"  range(a, 0, 0)
397. reg: CNSTP4  "#\n"  range(a, 0, 0)
398.
399. stmt: ASGNI1(addr,reg)  "ST R%1,R0,R%0 \n"  1
400. stmt: ASGNU1(addr,reg)  "ST R%1,R0,R%0 \n"  1
401. stmt: ASGNI2(addr,reg)  "ST R%1,R0,R%0 \n"  1
402. stmt: ASGNU2(addr,reg)  "ST R%1,R0,R%0 \n"  1
403. stmt: ASGNI4(addr,reg)  "ST R%1,R0,R%0 \n"  1
404. stmt: ASGNU4(addr,reg)  "ST R%1,R0,R%0 \n"  1
405. stmt: ASGNP4(addr,reg)  "ST R%1,R0,R%0 \n"  1
406.
407. reg:  INDIRI1(addr)     "LD R%c,R0,R%0 \n"  1
408. reg:  INDIRU1(addr)     "LD R%c,R0,R%0 \n"  1
409. reg:  INDIRI2(addr)     "LD R%c,R0,R%0 \n"  1
410. reg:  INDIRU2(addr)     "LD R%c,R0,R%0 \n"  1
411. reg:  INDIRI4(addr)     "LD R%c,R0,R%0 \n"  1
412. reg:  INDIRU4(addr)     "LD R%c,R0,R%0 \n"  1
413. reg:  INDIRP4(addr)     "LD R%c,R0,R%0 \n"  1
414.
415. reg:  CVII4(INDIRI1(addr))     "LD R%c,R0,R%0 \n"  1
416. reg:  CVII4(INDIRI2(addr))     "LD R%c,R0,R%0 \n"  1
417. reg:  CVUU4(INDIRU1(addr))     "LD R%c,R0,R%0 \n"  1
418. reg:  CVUU4(INDIRU2(addr))     "LD R%c,R0,R%0 \n"  1
419. reg:  CVUI4(INDIRU1(addr))     "LD R%c,R0,R%0 \n"  1
420. reg:  CVUI4(INDIRU2(addr))     "LD R%c,R0,R%0 \n"  1
421.
422. reg: ADDI4(reg,con)   "?ADD R%c,R0,R%0 \nADDI R%c,#%1\n"  1
423. reg: ADDI4(reg,reg)   "ADD R%c,R%0,R%1 \n"  1
424. reg: ADDP4(reg,con)   "?ADD R%c,R0,R%0 \nADDI R%c,#%1\n"  1
425. reg: ADDP4(reg,reg)   "ADD R%c,R%0,R%1 \n"  1
426. reg: ADDU4(reg,con)   "?ADD R%c,R0,R%0 \nADDI R%c,#%1\n"  1
427. reg: ADDU4(reg,reg)   "ADD R%c,R%0,R%1 \n"  1
428.
429. reg: SUBI4(reg,con)   "?ADD R%c,R0,R%0 \nSUBI R%c,#%1 \n"  1
430. reg: SUBI4(reg,reg)   "SUB R%c,R%0,R%1 \n"  1
431. reg: SUBU4(reg,con)   "?ADD R%c,R0,R%0 \nSUBI R%c,#%1\n"  1
432. reg: SUBU4(reg,reg)   "SUB R%c,R%0,R%1 \n"  1
433. reg: SUBP4(reg,con)   "?ADD R%c,R0,R%0 \nSUBI R%c,#%1\n"  1
434. reg: SUBP4(reg,reg)   "SUB R%c,R%0,R%1 \n"  1
435.
436. reg: DIVI4(reg,reg)   "#DIV R%c,R%0,R%1 \n"   1
437. reg: DIVU4(reg,reg)   "#DIV R%c,R%0,R%1 \n"   1
438.
439. reg: MODI4(reg,reg)  "#rem R%c,R%0,R%1\n"   1
440. reg: MODU4(reg,reg)  "#rem R%c,R%0,R%1\n"   1
441.
442. reg: MULI4(reg,reg)   "#MUL R%c,R%0,R%1\n"   1
```

```
443. reg: MULU4(reg,reg)   "#MUL R%c,R%0,R%1\n"   1
444.
445. reg: BANDI4(reg,reg)  "AND R%c,R%0,R%1\n"   1
446. reg: BORI4(reg,reg)   "OR R%c,R%0,R%1\n"    1
447. reg: BXORI4(reg,reg)  "XOR R%c,R%0,R%1\n"   1
448. reg: BANDU4(reg,reg)  "AND R%c,R%0,R%1\n"   1
449. reg: BORU4(reg,reg)   "OR R%c,R%0,R%1\n"    1
450. reg: BXORU4(reg,reg)  "XOR R%c,R%0,R%1\n"   1
451.
452.
453. rc5: reg            "R%0"
454.
455. reg: LSHI4(reg,rc5)  "#SL R%0,#%1 \nADD R%c,R0,R%0 \n"  1
456. reg: LSHU4(reg,rc5)  "#SL R%0,#%1 \nADD R%c,R0,R%0 \n"  1
457.
458. reg: RSHI4(reg,rc5)  "#SR R%0,#%1 \nADD R%c,R0,R%0 \n"  1
459. reg: RSHU4(reg,rc5)  "#SR R%0,#%1 \nADD R%c,R0,R%0 \n"  1
460.
461. reg: BCOMI4(reg)  "NOT R%c,R%0\n"   1
462. reg: BCOMU4(reg)  "NOT R%c,R%0\n"   1
463.
464. reg: NEGI4(reg)   "NOT R%c,R%0\n"  1
465.
466. reg: LOADI1(reg)  "?ADD R%c,R0,R%0 \n" 1
467. reg: LOADU1(reg)  "?ADD R%c,R0,R%0 \n" 1
468. reg: LOADI2(reg)  "?ADD R%c,R0,R%0 \n" 1
469. reg: LOADU2(reg)  "?ADD R%c,R0,R%0 \n" 1
470. reg: LOADI4(reg)  "?ADD R%c,R0,R%0 \n"  1
471. reg: LOADP4(reg)  "?ADD R%c,R0,R%0 \n" 1
472. reg: LOADU4(reg)  "?ADD R%c,R0,R%0 \n" 1
473.
474.
475. reg: CVII1(reg)  "?ADD R%c,R0,R%0 \n"  1
476. reg: CVII4(reg)  "?ADD R%c,R0,R%0 \n"  1
477. reg: CVUI4(reg)  "?ADD R%c,R0,R%0 \n"  1
478. reg: CVUU4(reg)  "?ADD R%c,R0,R%0 \n"  1
479.
480. stmt: LABELV  "%a:\n"
481. stmt: JUMPV(con)  "EQL R0, R0\nJMPD %0\n"   1
482. stmt: JUMPV(reg)   "EQL R0, R0\nJMP R%0\n"  1
483. stmt: EQI4(reg,reg)  "EQL R%0,R%1\nJMPD %a\n"   1
484. stmt: EQU4(reg,reg)  "EQL R%0,R%1\nJMPD %a\n"   1
485. stmt: GEI4(reg,reg)  "SUP R%0,R%1\nJMPD %a\nEQL R%0,R%1\nJMPD %a\n"   1
486. stmt: GEU4(reg,reg)  "SUP R%0,R%1\nJMPD %a\nEQL R%0,R%1\nJMPD %a\n"   1
487. stmt: GTI4(reg,reg)  "SUP R%0,R%1\nJMPD %a\n"   1
488. stmt: GTU4(reg,reg)  "SUP R%0,R%1\nJMPD %a\n"   1
489. stmt: LEI4(reg,reg)  "INF R%0,R%1\nJMPD %a\nEQL R%0,R%1\nJMPD %a\n"   1
490. stmt: LEU4(reg,reg)  "INF R%0,R%1\nJMPD %a\nEQL R%0,R%1\nJMPD %a\n"   1
491. stmt: LTI4(reg,reg)  "INF R%0,R%1\nJMPD %a\n"   1
492. stmt: LTU4(reg,reg)  "INF R%0,R%1\nJMPD %a\n"   1
493. stmt: NEI4(reg,reg)  "DIF R%0,R%1\nJMPD %a\n"   1
494. stmt: NEU4(reg,reg)  "DIF R%0,R%1\nJMPD %a\n"   1
495.
496. ar:   ADDRGP4     "#%a"
497.
498. reg:  CALLI4(ar)  "EQL R0, R0\nJSRD %0 \n"  1
499. reg:  CALLP4(ar)  "EQL R0, R0\nJSRD %0 \n"  1
500. reg:  CALLU4(ar)  "EQL R0, R0\nJSRD %0 \n"  1
501.
502. stmt: CALLV(ar)  "EQL R0, R0\nJSRD %0 \n"  1
503. ar: reg    "R%0"
504. ar: CNSTP4  "%a"   range(a, 0, 0x0fffffff)
505.
506. stmt: RETI4(reg)  "# ret\n"  1
507. stmt: RETU4(reg)  "# ret\n"  1
508. stmt: RETP4(reg)  "# ret\n"  1
509. stmt: RETV(reg)   "# ret\n"  1
510.
511. stmt: ARGI4(reg)  "# arg\n"  1
512. stmt: ARGP4(reg)  "# arg\n"  1
513. stmt: ARGU4(reg)  "# arg\n"  1
514.
515. reg1: ADDRLP4  "#LDI R%c,%a \n"
516. stmt: ARGI4(reg1)  "# arg\n"  1
517. stmt: ARGP4(reg1)  "# LD R%0,R0,R%0 [$$$] \n"  1
518. stmt: ARGU4(reg1)  "# arg\n"  1
519.
520.
521. stmt: ARGB(INDIRB(reg))       "# argb %0\n"      1
522. stmt: ASGNB(reg,INDIRB(reg))  "ST R%1,R0,R%0 \n"  1
523. %%
524. static void progend(void){}
525. static void progbeg(int argc, char *argv[]) {
526.         int i;
527.
528.         for (i = 0; i <= 15; i++)
529.                 ireg[i]  = mkreg("%d", i, 1, IREG);
530.
531.         iregw = mkwildcard(ireg);
532.         tmask[IREG] = INTTMP;
533.         vmask[IREG] = INTVAR;
```

```
534.
535.          print("; codigo assembly para R7 (TC)\n\n");
536.          print("\n.CODE\n");
537.          print("LDL R1,#feH\n");
538.          print("LDH R1,#9fH\n");
539.          print("LDSP R1\n"); // inicializando o SP
540.          print("XOR R0,R0,R0\n");
541.          print("EQL R0,R0\n");
542.          print("JMPD main\n\n");
543. }
544.
545. static Symbol rmap(int opk) {
546.         switch (optype(opk)) {
547.         case I: case U: case P: case B:
548.                 return iregw;
549.         default:
550.                 return 0;
551.         }
552. }
553.
554. static void target(Node p) {
555.         assert(p);
556.         switch (specific(p->op)) {
557.         case CNST+I: case CNST+U: case CNST+P:
558.                 if (range(p, 0, 0) == 0) {
559.                         setreg(p, ireg[0]);
560.                         p->x.registered = 1;
561.                 }
562.                 break;
563.         case CALL+V:
564.                 rtarget(p, 0, ireg[11]);
565.                 break;
566.
567.         case CALL+I: case CALL+P: case CALL+U:
568.                 rtarget(p, 0, ireg[11]);
569.                 setreg(p, ireg[2]);
570.                 break;
571.         case RET+I: case RET+U: case RET+P:
572.                 rtarget(p, 0, ireg[2]);
573.                 break;
574.         case ARG+F: case ARG+I: case ARG+P: case ARG+U: {
575.                 static int ty0;
576.                 int ty = optype(p->op);
577.                 Symbol q;
578.
579.                 q = argreg(p->x.argno, p->syms[2]->u.c.v.i, ty, opsize(p->op), ty0);
580.                 if (p->x.argno == 0)
581.                         ty0 = ty;
582.                 if (q &&
583.                 !(ty == F && q->x.regnode->set == IREG))
584.                         rtarget(p, 0, q);
585.                 break;
586.                 }
587.         case ASGN+B: rtarget(p->kids[1], 0, ireg[8]); break;
588.         case ARG+B:  rtarget(p->kids[0], 0, ireg[8]); break;
589.         }
590. }
591.
592. static void clobber(Node p) {
593.         assert(p);
594.         switch (specific(p->op)) {
595.         case CALL+F:
596.                 spill(INTTMP | INTRET, IREG, p);
597.                 break;
598.         case CALL+I: case CALL+P: case CALL+U:
599.                 spill(INTTMP,          IREG, p);
600.                 break;
601.         case CALL+V:
602.                 spill(INTTMP | INTRET, IREG, p);
603.                 break;
604.         }
605.
606. }
607.
608. static void emit2(Node p) {
609.         int registrador, registradorDir, endereco, constante;
610.         switch (specific(p->op)) {
611.          case ADDRL+P: case ADDRF+P:
612.                 registrador = getregnum(p);
613.                 endereco = p->syms[0]->x.offset;
614.                 print("LDL R%d, #%xH ;%d\n", registrador, endereco & 0xff, endereco);
615.                 print("LDH R%d, #%xH ;\n", registrador, (endereco & 0xff00) / 0x100 , endereco);
616.                 break;
617.          case CNST+I: case CNST+U: case CNST+P:
618.                 registrador = getregnum(p);
619.                 constante = atoi(p->syms[0]->x.name);
620.                 if (registrador != 0){
621.                 print("LDL R%d, #%xH ;%d\n", registrador, constante & 0xff, constante);
622.                 print("LDH R%d, #%xH ;\n", registrador, (constante & 0xff00) / 0x100, constante);}
623.                 break;
624.          case MUL+I: case MUL+U:
```

```
625.                registrador = getregnum(p);
626.                registradorDir = getregnum(p->x.kids[1]);
627.                print("PUSH R1 \n");
628.                print("PUSH R2 \n");
629.                print("ADD R1,R0,R%d \n", registrador);
630.                print("ADD R2,R0,R%d \n", registradorDir);
631.                print("EQL R0, R0 \n");
632.                print("JSRD rotina_mult \n");
633.                print("ADD R%d,R0,R2\n", registrador);
634.                print("POP R2 \n");
635.                print("POP R1 \n");
636.                flagMult = 1;
637.                break;
638.        case DIV+I: case DIV+U:
639.                registrador = getregnum(p);
640.                registradorDir = getregnum(p->x.kids[1]);
641.                print("PUSH R1 \n");
642.                print("PUSH R2 \n");
643.                print("ADD R1,R0,R%d \n", registrador);
644.                print("ADD R2,R0,R%d \n", registradorDir);
645.                print("EQL R0, R0 \n");
646.                print("JSRD rotina_div \n");
647.                print("ADD R%d,R0,R2\n", registrador);
648.                print("POP R2 \n");
649.                print("POP R1 \n");
650.                flagDiv = 1;
651.                break;
652.         case MOD+I: case MOD+U:
653.                registrador = getregnum(p);
654.                registradorDir = getregnum(p->x.kids[1]);
655.                print("PUSH R1 \n");
656.                print("PUSH R2 \n");
657.                print("ADD R1,R0,R%d \n", registrador);
658.                print("ADD R2,R0,R%d \n", registradorDir);
659.                print("EQL R0, R0 \n");
660.                print("JSRD rotina_resto_div \n");
661.                print("ADD R%d,R0,R2\n", registrador);
662.                print("POP R2 \n");
663.                print("POP R1 \n");
664.                flagResto = 1;
665.                break;
666.         case LSH+I: case LSH+U:
667.                registrador = getregnum(p);
668.                registradorDir = getregnum(p->x.kids[1]);
669.                print("PUSH R1 \n");
670.                print("PUSH R2 \n");
671.                print("ADD R1,R0,R%d \n", registrador);
672.                print("ADD R2,R0,R%d \n", registradorDir);
673.                print("EQL R0, R0 \n");
674.                print("JSRD rotina_shift_left \n");
675.                print("ADD R%d,R0,R2\n", registrador);
676.                print("POP R2 \n");
677.                print("POP R1 \n");
678.                flagShiftLeft = 1;
679.                break;
680.         case RSH+I: case RSH+U:
681.                registrador = getregnum(p);
682.                registradorDir = getregnum(p->x.kids[1]);
683.                print("PUSH R1 \n");
684.                print("PUSH R2 \n");
685.                print("ADD R1,R0,R%d \n", registrador);
686.                print("ADD R2,R0,R%d \n", registradorDir);
687.                print("EQL R0, R0 \n");
688.                print("JSRD rotina_shift_rigth \n");
689.                print("ADD R%d,R0,R2\n", registrador);
690.                print("POP R2 \n");
691.                print("POP R1 \n");
692.                flagShiftRigth = 1;
693.                break;
694.         case ADDRG+P:
695.                print("%s",p->syms[0]->x.name);
696.                if (!strcmp("printf",p->syms[0]->x.name)) flag_printf = 1;
697.                break;
698.       }
699. }
700.
701. static Symbol argreg(int argno, int offset, int ty, int sz, int ty0) {
702.         assert((offset&3) == 0);
703.         return ireg[(offset/4) + 4];
704. }
705.
706. static void doarg(Node p) {
707.         static int argno;
708.         int align;
709.
710.         if (argoffset == 0)
711.                 argno = 0;
712.         p->x.argno = argno++;
713.         align = p->syms[1]->u.c.v.i < 4 ? 4 : p->syms[1]->u.c.v.i;
714.         p->syms[2] = intconst(mkactual(align,p->syms[0]->u.c.v.i));
715. }
```

```
716.
717. static void local(Symbol p) {
718.         if (askregvar(p, rmap(ttob(p->type))) == 0)
719.                 mkauto(p);
720. }
721.
722. static void function(Symbol f, Symbol caller[], Symbol callee[], int ncalls) {
723.         int i, saved, sizefsave, sizeisave, varargs;
724.         Symbol r, argregs[4];
725.
726.         usedmask[0] = usedmask[1] = 0;
727.         freemask[0] = freemask[1] = ~(unsigned)0;
728.         offset = maxoffset = maxargoffset = 0;
729.
730.         for (i = 0; callee[i]; i++) {
731.                 Symbol p = callee[i];
732.                 Symbol q = caller[i];
733.                 assert(q);
734.                 offset = roundup(offset, q->type->align);
735.                 p->x.offset = q->x.offset = offset;
736.                 p->x.name = q->x.name = stringd(offset);
737.                 r = argreg(i, offset, optype(ttob(q->type)), q->type->size, optype(ttob(caller[0]->type)));
738.                 if (i < 4)
739.                         argregs[i] = r;
740.                 offset = roundup(offset + q->type->size, 4);
741.                 if (varargs)
742.                         p->sclass = AUTO;
743.                 else if (r && ncalls == 0 &&
744.                          !isstruct(q->type) && !p->addressed &&
745.                          !(isfloat(q->type) && r->x.regnode->set == IREG)) {
746.                         p->sclass = q->sclass = REGISTER;
747.                         askregvar(p, r);
748.                         assert(p->x.regnode && p->x.regnode->vbl == p);
749.                         q->x = p->x;
750.                         q->type = p->type;
751.                 }
752.                 else if (askregvar(p, rmap(ttob(p->type)))
753.                          && r != NULL
754.                          && (isint(p->type) || p->type == q->type)) {
755.                         assert(q->sclass != REGISTER);
756.                         p->sclass = q->sclass = REGISTER;
757.                         q->type = p->type;
758.                 }
759.         }
760.         assert(!caller[i]);
761.         gencode(caller, callee);
762.         print("\n%s:\n", f->x.name);
763.
764.         for (i = 0; i < 4 && callee[i]; i++) {
765.                 r = argregs[i];
766.                 if (r && r->x.regnode != callee[i]->x.regnode) {
767.                         Symbol out = callee[i];
768.                         Symbol in  = caller[i];
769.                         int rn = r->x.regnode->number;
770.                         int rs = r->x.regnode->set;
771.                         int tyin = ttob(in->type);
772.
773.                         assert(out && in && r && r->x.regnode);
774.                         assert(out->sclass != REGISTER || out->x.regnode);
775.                         if (out->sclass == REGISTER && (isint(out->type) || out->type == in->type)) {
776.                                 int outn = out->x.regnode->number;
777.                                 print("ADD R%d,R0,R%d\n", outn, rn);
778.                         }
779.                }
780.         }
781.         emitcode();
782.
783.         // Se nao for o main imprimo RTS
784.         if (strcmp("main",f->x.name))
785.                 print("RTS\n");
786.         else{
787.                 print("HALT\n\n");
788.         if (flagMult){
789.                 print("rotina_mult: \n");
790.                 print("PUSH R8\n");
791.                 print("PUSH R9\n");
792.                 print("PUSH R10\n");
793.                 print("PUSH R11\n");
794.                 print("PUSH R12\n");
795.                 print("PUSH R13\n");
796.                 print("PUSH R14\n");
797.                 print("PUSH R15\n");
798.                 print("ADD R9,R0,R0 \n");
799.                 print("ADD R14,R0,R9 \n");
800.                 print("ADD R13,R0,R9 \n");
801.                 print("LDL R9 #M5 \n");
802.                 print("LDH R9 #M5 \n");
803.                 print("EQL R0, R0 \n");
804.                 print("JMP R9 \n");
805.                 print("M2: \n");
806.                 print("ADD R9,R0,R14 \n");
```

```
807.                  print("ADD R8,R0,R1 \n");
808.                  print("ADD R9,R9,R8 \n");
809.                  print("ADD R14,R0,R9 \n");
810.                  print("M3: \n");
811.                  print("ADD R9,R0,R13 \n");
812.                  print("ADDI R9,#1 \n");
813.                  print("ADD R13,R0,R9 \n");
814.                  print("M5: \n");
815.                  print("ADD R9,R0,R13 \n");
816.                  print("ADD R8,R0,R2 \n");
817.                  print("INF R9,R8 \n");
818.                  print("JMPD M2 \n");
819.                  print("ADD R2,R0,R14 \n");
820.                  print("POP R15\n");
821.                  print("POP R14\n");
822.                  print("POP R13\n");
823.                  print("POP R12\n");
824.                  print("POP R11\n");
825.                  print("POP R10\n");
826.                  print("POP R9\n");
827.                  print("POP R8\n");
828.                  print("M1: \n");
829.                  print("RTS \n\n");
830.           }
831.        if (flagDiv){
832.                  print("rotina_div: \n");
833.                  print("PUSH R8\n");
834.                  print("PUSH R9\n");
835.                  print("PUSH R10\n");
836.                  print("PUSH R11\n");
837.                  print("PUSH R12\n");
838.                  print("PUSH R13\n");
839.                  print("PUSH R14\n");
840.                  print("PUSH R15\n");
841.                  print("ADD R13,R0,R0 \n");
842.                  print("ADD R14,R0,R2 \n");
843.                  print("LDL R9 #D7 \n");
844.                  print("LDH R9 #D7 \n");
845.                  print("EQL R0, R0 \n");
846.                  print("JMP R9 \n");
847.                  print("D4: \n");
848.                  print("ADD R9,R0,R13 \n");
849.                  print("ADDI R9,#1 \n");
850.                  print("ADD R13,R0,R9 \n");
851.                  print("D5: \n");
852.                  print("ADD R9,R0,R14 \n");
853.                  print("ADD R8,R0,R2 \n");
854.                  print("ADD R9,R9,R8 \n");
855.                  print("ADD R14,R0,R9 \n");
856.                  print("D7: \n");
857.                  print("ADD R9,R0,R14 \n");
858.                  print("ADD R8,R0,R1 \n");
859.                  print("INF R9,R8 \n");
860.                  print("JMPD D4 \n");
861.                  print("EQL R9,R8 \n");
862.                  print("JMPD D4 \n");
863.                  print("ADD R2,R0,R13 \n");
864.                  print("POP R15\n");
865.                  print("POP R14\n");
866.                  print("POP R13\n");
867.                  print("POP R12\n");
868.                  print("POP R11\n");
869.                  print("POP R10\n");
870.                  print("POP R9\n");
871.                  print("POP R8\n");
872.                  print("D3: \n");
873.                  print("RTS \n\n");
874.              }
875.         if (flagResto){
876.                  print("rotina_resto_div: \n");
877.                  print("PUSH R8\n");
878.                  print("PUSH R9\n");
879.                  print("PUSH R10\n");
880.                  print("PUSH R11\n");
881.                  print("PUSH R12\n");
882.                  print("PUSH R13\n");
883.                  print("PUSH R14\n");
884.                  print("PUSH R15\n");
885.                  print("ADD R15,R0,R1\n");
886.                  print("ADD R10,R0,R2\n");
887.                  print("ADD R9,R0,R1\n");
888.                  print("SUP R10,R9\n");
889.                  print("JMPD Q8\n");
890.                  print("ADD R8,R0,R0\n");
891.                  print("EQL R10,R8\n");
892.                  print("JMPD Q8\n");
893.                  print("DIF R9,R8\n");
894.                  print("JMPD Q10\n");
895.                  print("Q8:\n");
896.                  print("ADD R2,R0,R0\n");
897.                  print("LDL R10, #Q4\n");
```

```
898.                    print("LDH R10, #Q4\n");
899.                    print("EQL R0, R0\n");
900.                    print("JMP R10\n");
901.                    print("Q9:\n");
902.                    print("ADD R10,R0,R15\n");
903.                    print("ADD R9,R0,R2\n"); ;
904.                    print("SUB R10,R10,R9\n");
905.                    print("ADD R15,R0,R10\n");
906.                    print("Q10:\n");
907.                    print("ADD R10,R0,R15\n");
908.                    print("ADD R9,R0,R2\n");
909.                    print("SUP R10,R9\n");
910.                    print("JMPD Q9\n");
911.                    print("EQL R10,R9\n");
912.                    print("JMPD Q9\n");
913.                    print("ADD R2,R0,R15\n");
914.                    print("POP R15\n");
915.                    print("POP R14\n");
916.                    print("POP R13\n");
917.                    print("POP R12\n");
918.                    print("POP R11\n");
919.                    print("POP R10\n");
920.                    print("POP R9\n");
921.                    print("POP R8\n");
922.                    print("Q4:\n");
923.                    print("RTS\n");
924.     }
925.     if (flagShiftLeft){
926.             print("rotina_shift_left: \n");
927.             print("PUSH R9 \n");
928.             print("ADDI R9,#1 \n");
929.             print("SL1: \n");
930.             print("SUP R2,R0 \n");
931.             print("JMPD SL2 \n");
932.             print("ADD R2,R0,R1 \n");
933.             print("POP R9 \n");
934.             print("RTS \n");
935.             print("SL2: \n");
936.             print("EQL R0,R9 \n");
937.             print("SL R1,R1 \n");
938.             print("SUBI R2,#1 \n");
939.             print("EQL R0,R0 \n");
940.             print("JMPD SL1 \n\n");
941.     }
942.     if (flagShiftRigth){
943.             print("rotina_shift_rigth: \n");
944.             print("PUSH R9 \n");
945.             print("ADDI R9,#1 \n");
946.             print("SR1: \n");
947.             print("SUP R2,R0 \n");
948.             print("JMPD SR2 \n");
949.             print("ADD R2,R0,R1 \n");
950.             print("POP R9 \n");
951.             print("RTS \n");
952.             print("SR2: \n");
953.             print("EQL R0,R9 \n");
954.             print("SR R1,R1 \n");
955.             print("SUBI R2,#1 \n");
956.             print("EQL R0,R0 \n");
957.             print("JMPD SR1 \n\n");
958.     }
959.     if (flag_printf){
960.             print("printf: \n");
961.             print("PUSH R9\n");
962.             print("LDL R9,#00H \n");
963.             print("LDH R9,#C0H \n");
964.             print("ST R5,R0,R9 \n");
965.             print("POP R9\n");
966.             print("RTS \n\n");
967.     }
968.     print(".ENDCODE\n");
969. }
970.}
971.
972./* funçao responsavel pela definicao de constantes */
973.static void defconst(int suffix, int size, Value v) {
974. if (tamanho > 4){
975.         print("#%XH, ", suffix == I ? v.i : v.u);
976.     tamanho = tamanho - 4;
977. }else{
978.         print("#%XH ;[f15]\n", suffix == I ? v.i : v.u);
979. }
980.}
981.
982.static void defaddress(Symbol p) {
983.        print("#%s ;[f16]\n", p->x.name);
984.}
985.
986.static void defstring(int n, char *str) {
987.        char *s;
988.        for (s = str; s < str + n; s++){
```

```
989.             if (contador == tamanho){
990.                     print("#%xH ;[f17]\n", (*s)&0377);
991.                     break;
992.             }
993.     print("#%xH, ", (*s)&0377);
994.     contador++;
995. }
996.}
997.
998.static void defsymbol(Symbol p) {
999.         if (p->scope >= LOCAL && p->sclass == STATIC)
1000.                p->x.name = stringf("L%d", genlabel(1));
1001.        else if (p->generated)
1002.                p->x.name = stringf("L%s", p->name);
1003.        else
1004.                assert(p->scope != CONSTANTS || isint(p->type) || isptr(p->type)),
1005.                p->x.name = p->name;
1006.}
1007.
1008.static void address(Symbol q, Symbol p, long n) {
1009.        if (p->scope == GLOBAL
1010.        || p->sclass == STATIC || p->sclass == EXTERN)
1011.                q->x.name = stringf("%s%s%D", p->x.name,
1012.                        n >= 0 ? "+" : "", n);
1013.        else {
1014.                assert(n <= INT_MAX && n >= INT_MIN);
1015.                q->x.offset = p->x.offset + n;
1016.                q->x.name = stringd(q->x.offset);
1017.        }
1018.}
1019.
1020.static void global(Symbol p) {
1021.        if (p->u.seg == BSS) { //se nao esta inicializada uma variavel global
1022.            print("%s: DB #00 ;[f18]\n", p->x.name);
1023.        } else {
1024.            contador = 1;
1025.            tamanho = p->type->size;
1026.                print("%s: DB ", p->x.name);
1027.        }
1028.}
1029.
1030.static void export(Symbol p) {}
1031.static void import(Symbol p) {}
1032.static void segment(int n) {}
1033.static void space(int n) {}
1034.static void blkloop(int dreg, int doff, int sreg, int soff, int size, int tmps[]) {}
1035.static void blkfetch(int size, int off, int reg, int tmp) {}
1036.static void blkstore(int size, int off, int reg, int tmp) {}
1037.static void stabinit(char *, int, char *[]);
1038.static void stabline(Coordinate *);
1039.static void stabsym(Symbol);
1040.static char *currentfile;
1041.static void stabinit(char *file, int argc, char *argv[]) {}
1042.static void stabline(Coordinate *cp) {}
1043.static void stabsym(Symbol p) {}
1044.
1045.Interface gaphIR = {
1046.        1, 1, 0,  /* char */
1047.        2, 2, 0,  /* short */
1048.        4, 4, 0,  /* int */
1049.        4, 4, 0,  /* long */
1050.        4, 4, 0,  /* long long */
1051.        4, 4, 1,  /* float */
1052.        8, 8, 1,  /* double */
1053.        8, 8, 1,  /* long double */
1054.        4, 4, 0,  /* T * */
1055.        0, 1, 0,  /* struct */
1056.        0,      /* little_endian */
1057.        0,  /* mulops_calls */
1058.        0,  /* wants_callb */
1059.        1,  /* wants_argb */
1060.        1,  /* left_to_right */
1061.        0,  /* wants_dag */
1062.        0,  /* unsigned_char */
1063.        address,
1064.        blockbeg,
1065.        blockend,
1066.        defaddress,
1067.        defconst,
1068.        defstring,
1069.        defsymbol,
1070.        emit,
1071.        export,
1072.        function,
1073.        gen,
1074.        global,
1075.        import,
1076.        local,
1077.        progbeg,
1078.        progend,
1079.        segment,
```

```
1080.        space,
1081.        0, 0, 0, stabinit, stabline, stabsym, 0,
1082.        {
1083.                4,      /* max_unaligned_load */
1084.                rmap,
1085.                blkfetch, blkstore, blkloop,
1086.                _label,
1087.                _rule,
1088.                _nts,
1089.                _kids,
1090.                _string,
1091.                _templates,
1092.                _isinstruction,
1093.                _ntname,
1094.                emit2,
1095.                doarg,
1096.                target,
1097.                clobber,
1098.
1099.        }
1100.};
1101.static char rcsid[] = "$Id: gaph.nw,v 0.02 2001/17/18 21:54:00 drh Exp $";
```