View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Repositório Institucional da UFSC

UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Daniel Carlos Casarotto

Utilitários Binários Redirecionáveis: da Linkedição rumo à Tradução Binária

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação.

Luiz Cláudio Villar dos Santos

Florianópolis, março de 2007

Utilitários Binários Redirecionáveis: da Linkedição rumo à Tradução Binária

Daniel Carlos Casarotto

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação, na área de concentração de Sistemas de Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Rogério Cid Bastos

Banca Examinadora

Luiz Cláudio Villar dos Santos

Luís Fernando Friedrich

Olinto José Varela Furtado

Rodolfo Jardim de Azevedo

"A teoria é assassinada mais cedo
ou mais tarde pela experiência."
Albert Einstein

Dedicado à minha família, que sempre esteve comigo em
todos os momentos de minha vida.

Agradecimentos

Ao professor Luiz Cláudio V. dos Santos, que me orientou de maneira exemplar.

Ao colega Alexandro Baldassin pelo seu suporte técnico e confiança dada ao trabalho.

Aos colegas de mestrado Max R. de Oliveira Schultz e José Otávio Carlomagno Filho, que trabalharam em tópicos correlatos, por sempre estarem dispostos a ajudar e trocar idéias.

Sumário

Sumário vi

Lista de Figuras ix

Lista de Tabelas xi

Lista de Algoritmos xii

Lista de Acrônimos xiii

Resumo xiv

Abstract xv

1 Introdução 1

1.1 Projeto orientado a plataforma 2

1.2 A necessidade de técnicas redirecionáveis 3

1.3 A contribuição desta dissertação 4

2 Trabalhos correlatos 5

2.1 Geração automática de linkeditores 5

2.2 Tradução binária 7

3 Fundamentos, modelagem e infra-estrutura 12

3.1 Fundamentos da linkedição 12

3.1.1 Informações de relocação 13

3.2 Descrição de CPUs 15

3.2.1 Modelagem básica de CPUs em ArchC 15

3.3 A infra-estrutura do pacote *GNU Binutils* 20

3.3.1 Biblioteca Opcodes 22
3.3.2 Biblioteca BFD 23

4 Geração de linkeditores **25**
4.1 A abordagem adotada 25
4.2 A geração automática de linkeditores 25
4.2.1 Geração da biblioteca Opcodes 26
4.2.2 Geração da biblioteca BFD 27
4.2.3 Modificações no gerador de montadores 29
4.2.4 Compilação do linkeditor 31
4.2.5 Os arquivos gerados 32
4.2.6 Extensões ao trabalho proposto 32

5 Validação do gerador de linkeditores **35**
5.1 Configuração experimental 35
5.1.1 Caracterização do aparato experimental 35
5.1.2 Validação da correção do linkeditor gerado 37
5.1.3 Fluxo de validação 37
5.2 Análise dos resultados experimentais 39

6 O papel dos utilitários na tradução binária **44**
6.1 Motivação 44
6.2 Proposta de estrutura de um tradutor binário 45
6.3 Formulação do problema de mapeamento de instruções 47
6.3.1 A representação intermediária independente de máquina 48
6.3.2 A captura de semântica das instruções nos modelos de CPU 49
6.3.3 A decomposição de instruções 51
6.3.4 A seleção de instruções 52
6.4 Resultados experimentais preliminares 55

7 Conclusão **58**
7.1 Apreciação do trabalho 58
7.2 Contribuições técnico-científicas e produtos de trabalho 59
7.3 Trabalhos em andamento 59

7.4 Tópicos para investigação futura 59

A Estrutura e mecanismo de relocação na biblioteca BFD 61

Referências Bibliográficas 65

Lista de Figuras

1.1 Fluxo de exploração de soluções 3

2.1 Exemplo de uso da construção `BEHAVIOR` na ADL LISA 9
2.2 Exemplo de uso da construção `SEMANTICS` na ADL LISA 10
2.3 Fluxo do processo de tradução do *framework UQBT* [1] 11

3.1 Exemplo do processo de linkedição 14
3.2 Descrição ArchC dos formatos de instrução do MIPS 16
3.3 Descrição ArchC do mapeamento entre instruções e formatos 17
3.4 Descrição ArchC do mapeamento entre nomes e números de registradores do MIPS 17
3.5 Descrição ArchC de linguagem de montagem e de decodificação 18
3.6 Descrição ArchC de uma pseudo-instrução do MIPS 18
3.7 Trecho da descrição da arquitetura MIPS em ArchC 21
3.8 Estrutura do linkeditor do pacote *GNU Binutils* 22
3.9 Estrutura da biblioteca BFD do pacote *GNU Binutils* 23

4.1 Fluxo de geração dos utilitários binários 26
4.2 Exemplo de um *script* de linkedição que descreve o *layout* do arquivo de saída . . . 31
4.3 Arquivos gerados pela ferramenta 32
4.4 Descrição das relocações na extensão do trabalho 33
4.5 Descrição das relocações das instruções em seu formato original 34
4.6 Descrição das relocações das instruções em seu novo formato 34

5.1 Comparação entre o número de formatos descritos em cada modelo com o tamanho do arquivo `tc-[arch].c` 37
5.2 Comparação entre o número de instruções em linguagem de montagem descritas com o tamanho do arquivo `[arch]-opc.c` 38

5.3 Fluxo de validação da ferramenta `aclink` 39
5.4 Trecho do arquivo que define os símbolos das bibliotecas externas 39

6.1 Fluxo de tradução binária 46
6.2 Exemplo de representação intermediária para a arquitetura MIPS 48
6.3 Fluxo de operação do mapeador 49
6.4 Exemplo de descrição semântica de algumas instruções da arquitetura MIPS 50
6.5 Exemplo de RIs primitivas para algumas instruções do MIPS 51
6.6 Exemplo de decomposição 51
6.7 Exemplo de descrição semântica de algumas instruções da arquitetura SPARC 53
6.8 Grafo resultante da tradução 53
6.9 Descrição do mapeamento de operações primitivas 54
6.10 Exemplo de tradução parcial para a arquitetura MIPS 55
6.11 Exemplo de tradução final para a arquitetura MIPS 55

A.1 Formato de instrução do tipo I da arquitetura MIPS 62

Lista de Tabelas

5.1 Tamanho dos arquivos gerados pelas ferramentas `acasm` e `aclink` (número de linhas) . . . 36
5.2 Complexidade dos modelos utilizados . . . 37
5.3 Caracterização dos programas do *benchmark* MiBench para o MIPS . . . 41
5.4 Caracterização dos programas do *benchmark* MiBench para o SPARC . . . 42
5.5 Caracterização dos programas do *benchmark* MiBench para o POWERPC . . . 43

6.1 Resultados da tradução binária MIPS-SPARC . . . 56
6.2 Resultados da tradução binária SPARC-MIPS . . . 56

Lista de Algoritmos

4.1 Geração da tabela de relocações 28

4.2 Criação de uma entrada na tabela de relocações 30

6.1 Algoritmo de seleção de instruções 52

Lista de Acrônimos

ADL *Architecture Description Language* (Linguagem de Descrição de Arquiteturas)

ASIP *Application Specific Instruction-Set Processor*

CI Circuito Integrado

HDL *Hardware Description Language*

IP *Intellectual Property*

ISA *Instruction Set Architecture*

LOF *LISA Object File*

RI Representação Intermediária

RTL *Register Transfer Level*

SoC *System on Chip*

TLM *Transaction Level Modeling*

Resumo

A grande oferta de *hardware*, aliada à crescente demanda por aplicações de sistemas embarcados, deu origem a sistemas integrados em uma pastilha de silício (SoCs). A exploração do espaço de projeto de SoCs pode requerer a geração de código para diversas CPUs alternativas. Entretanto, não seria eficiente desenvolver um *kit* de ferramentas de geração de código inteiramente novo para cada nova CPU explorada. Tampouco pode um tal *kit* ser sempre reutilizado a partir de um pacote de utilitários padrão, especialmente se a CPU é dedicada (ASIP). Portanto, precisa-se de ferramentas com redirecionamento automático.

Essa dissertação propõe uma técnica de geração automática de linkeditores a partir da descrição formal de uma CPU por meio de uma linguagem de descrição de arquiteturas (ADL). A idéia-chave para tal geração é a generalização da noção de modificadores para descrever relocações em um nível mais alto de abstração. A implementação da técnica baseia-se na ADL ArchC e no conhecido pacote *GNU Binutils*. Para o redirecionamento, reusa-se as bibliotecas independentes de arquitetura e gera-se automaticamente as bibliotecas dependentes da CPU-alvo. A corretude e a robustez da técnica foram comprovadas para três CPUs (MIPS, SPARC e POWERPC) rodando programas do *benchmark* MiBench. Para a validação experimental, foram comparados com sucesso os arquivos executáveis produzidos pela ferramenta gerada com aqueles produzidos pela ferramenta congênere já disponível no pacote *GNU Binutils*.

Além disso, propõe-se uma técnica de tradução binária baseada em redirecionamento automatico. Mostra-se como utilitários binários (desenvolvidos nesta e em outras dissertações) podem ser encadeados para dar origem a um tradutor binário estático. Resultados experimentais preliminares indicam a viabilidade da técnica.

Abstract

The huge offer of hardware combined with the increasing demand from embedded systems gave rise to Systems-on-Chip (SoCs). SoC design space exploration requires code generation for several CPU core alternatives. However, an embedded-software code-generation toolkit cannot be developed from scratch for every target CPU under exploration. Nor can it always be reused from standard packages, especially when the CPU core is an ASIP. That's why automatically retargetable tools are required.

This dissertation proposes a technique for linkeditor automatic generation from a formal description of the target CPU core, written with an architecture description language (ADL). The key idea for such generation is the use of the notion of modifier to describe relocations at a higher level of abstraction. The implementation of the technique relies on the ArchC ADL and on the well-known *GNU Binutils* package. To make it retargetable, the target-independent libraries are reused and the architecture-dependent ones are automatically generated. The technique's correctness and robustness were verified for three target CPUs (MIPS, SPARC and POWERPC) running programs from the benchmark MiBench. For experimental validation, we have successfully compared the executable files produced by the generated tool with those produced by a similar tool available in the *GNU Binutils* package.

Besides, a binary translation technique is proposed which is based on automatic retargeting. We show how retargetable binary utilities can be chained so as to give rise to a static binary translator. Preliminary experimental results indicate the techniques's viability.

Capítulo 1

Introdução

Sistemas embarcados estão cada vez mais presentes em nosso dia-a-dia, seja em produtos eletrônicos de uso doméstico como televisores, aparelhos de DVD, celulares, tocadores de MP3 e câmeras digitais, seja em aplicações nas áreas de telecomunicações e automação industrial, bem como na indústria aeroespacial e automotiva. Mais de 90% dos processadores fabricados são utilizados em sistemas embarcados [2]. Essencialmente, um sistema embarcado é um sistema eletrônico cujos componentes de hardware e de software cooperam para executar uma determinada função. Sistemas embarcados podem ser construídos com componentes na forma de circuitos integrados (CIs).

Em 1965, Gordon Moore fez uma previsão que acabaria sendo usada pela *Semiconductor Industry Association* para quantificar a taxa de crescimento da integração dos circuitos integrados. A Lei de Moore diz que, a cada dezoito meses, dobra o número de transistores que podem ser colocados em um CI. Essa possibilidade de se colocar cada vez mais transistores em uma menor área torna os circuitos integrados cada vez mais velozes. Esta imensa oferta de hardware, combinada com a demanda crescente de sistemas embarcados, deu origem a sistemas dedicados de hardware e software em um único circuito integrado, os assim chamados *Systems-on-Chip* (SoCs) [3]. SoCs são compostos por blocos reusáveis de hardware, construídos para executar determinada tarefa. Tais blocos são denominados *cores* ou IPs (*Intellectual Property*). Um IP pode ser um componente de hardware com comportamento fixo como por exemplo memória. Pode ser também um dispositivo de hardware programável como um processador (CPU).

1.1 Projeto orientado a plataforma

Com os SoCs ficando cada vez mais complexos e com a pressão do *time-to-market*[1] é necessário reduzir ao máximo o tempo de desenvolvimento e produção de um produto. Por isso, uma prática comum é a de agrupar vários IPs em uma *plataforma*, que pode ser reutilizada para realizar um conjunto de aplicações específicas distintas no âmbito de um mesmo domínio de aplicações (multimídia, *wireless*, controle, etc). Assim, ao se lançar novos produtos, componentes desnecessários são removidos e apenas os novos componentes precisam ser realmente desenvolvidos.

SoCs podem ser construídos com CPUs de propósitos gerais ou CPUs dedicadas (ASIPs). Um ASIP é uma CPU cujo conjunto de instruções foi otimizado para uma aplicação específica.

Os componentes de um SoC podem ser combinados de diversas maneiras, a fim de otimizar o desempenho, o consumo de energia e a área ocupada em silício. A *exploração do espaço de projeto* busca encontrar os componentes que resultam na combinação ideal destes fatores para determinada aplicação.

A modelagem de componentes de hardware é feita com o auxílio de linguagens que admitem diferentes estilos e níveis de descrição. Uma *linguagem de descrição de hardware* (HDL: *Hardware Description Language*) é predominantemente usada no nível de descrição denominado *Register Transfer Level* (RTL), que descreve a estrutura do circuito em termos do fluxo de sinais ou dados entre registradores. Exemplos de HDLs são VHDL [4] e Verilog [5].

Em um nível superior de abstração estão as *linguagens de descrição de sistema*, como SystemC [6] e SystemVerilog [7]. Sua principal característica é a separação entre comunicação e a computação dentro de cada componente do sistema. Um estilo bastante promissor no nível de sistema é denominado *Transaction Level Modeling* (TLM) [8].

No entanto, nenhuma das linguagens citadas é adequada à geração automática de ferramentas de software para CPUs (como simuladores, montadores ou linkeditores). Para esse fim, costuma-se utilizar uma *linguagem de descrição de arquiteturas* (ADL: *Architeture Description Language*). Uma ADL permite a descrição de CPUs com informações adequadas à geração automática de ferramentas de software, tais como registradores, formatos de instruções, códigos operacionais, linguagem de montagem, etc. Várias ADLs são reportadas na literatura, tais como nML [9], LISA [10], ISDL [11] e ArchC [12].

[1]Tempo gasto desde a concepção de um produto até ser colocado à venda.

1.2 A necessidade de técnicas redirecionáveis

A exploração do espaço de projeto de SoCs pode requerer a geração de código para diversas CPUs alternativas. Entretanto, não seria eficiente desenvolver um *kit* de ferramentas de geração de código inteiramente novo para cada nova CPU explorada. Tampouco pode um tal *kit* ser reutilizado a partir de um pacote de utilitários padrão, especialmente se a CPU é dedicada (ASIP). Portanto, precisa-se de ferramentas com redirecionamento automático. A Figura 1.1 ilustra o processo de exploração de soluções alternativas com o auxílio de ferramentas redirecionáveis. Dada a descrição de uma CPU, suas ferramentas são geradas automaticamente. Um código aplicativo escrito em linguagem de alto nível pode desta forma ser compilado, montado, linkeditado e simulado para verificar se os requisitos especificados são satisfeitos. Em caso contrário, faz-se uma modificação no conjunto de instruções da CPU ou escolhe-se uma nova CPU a ser explorada. O processo é repetido até que os requisitos sejam satisfeitos.

Na Figura 1.1 as ferramentas montador, linkeditor, depurador e desmontador constituem *utilitários binários*, e são objetos desta e de outras dissertações de mestrado correlatas [13] [14].

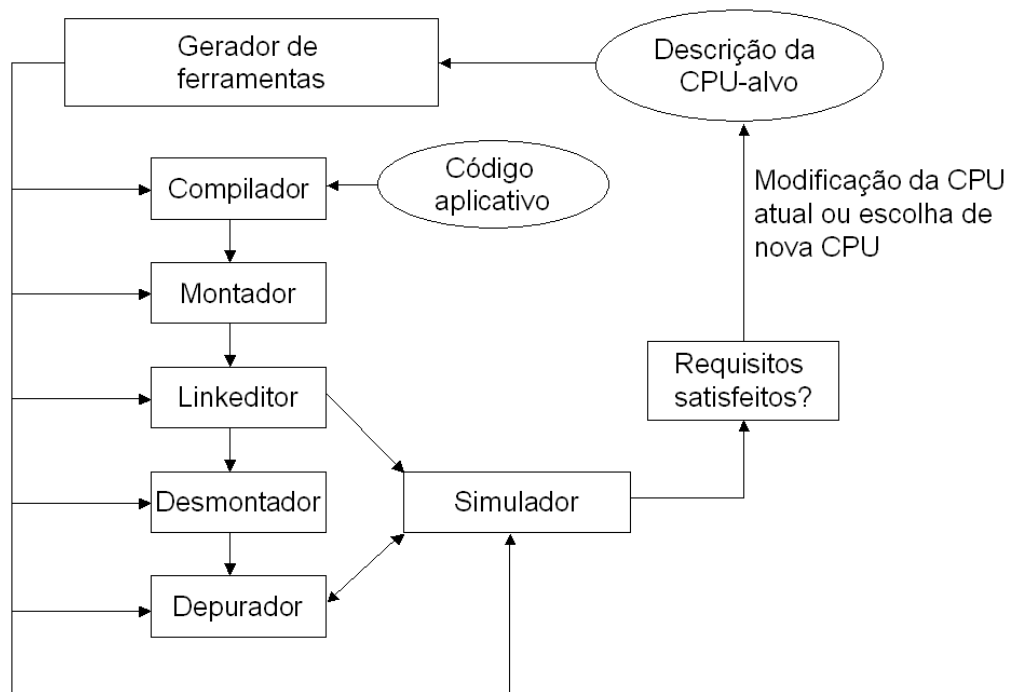

Figura 1.1: Fluxo de exploração de soluções

1.3 A contribuição desta dissertação

O escopo desta dissertação é a geração automaticamente redirecionável de utilitários binários, os quais são então utilizados como parte da cadeia de ferramentas para a geração de código durante a exploração de soluções alternativas.

O foco principal desta dissertação é a geração automática de linkeditores, onde as contribuições mais relevantes são:

- Generalização da noção de modificadores (Seção 3.2.1.1) para fins de relocação.
- Mapeamento automático de modificadores para geração automática da tabela de relocações (Seção 4.2.2).

Como foco secundário, propõe-se uma técnica de tradução binária, a qual também é baseada em utilitários binários automaticamente redirecionáveis.

Esta dissertação está organizada da seguinte forma: o Capítulo 2 apresenta os principais trabalhos correlatos em geração automática de utilitários binários; o Capítulo 3 apresenta os conceitos fundamentais associados aos tipos de utilitários que serão gerados automaticamente e a infra-estrutura necessária para essa geração; o Capítulo 4 discorre sobre a geração automática de linkeditores; o Capítulo 5 mostra a validação experimental do gerador de linkeditores; o Capítulo 6 aborda o uso de utilitários binários para fins de tradução binária; ao final, o Capítulo 7 apresenta as conclusões e aponta tópicos para investigação futura.

O trabalho de pesquisa reportado nesta dissertação foi desenvolvido sob fomento parcial do Programa Nacional de Cooperação Acadêmica (PROCAD) da CAPES, no âmbito do Projeto nº 0326054, intitulado "Automação de Projeto de Sistemas Dedicados Usando uma Linguagem de Descrição de Arquiteturas", que norteia e formaliza as atividades de cooperação científica entre o Programa de Pós-Graduação em Ciência da Computação (PPGCC) da UFSC e o Instituto de Computação da UNICAMP. O trabalho foi executado no Laboratório de Automação do Projeto de Sistemas (LAPS) da UFSC (`http://www.laps.inf.ufsc.br/`).

Capítulo 2

Trabalhos correlatos

2.1 Geração automática de linkeditores

Linkeditores são ferramentas de desenvolvimento de software quase tão antigas quanto os próprios computadores; suas primeiras versões datam da década de 40 [15]. Assim, as técnicas para sua construção são bem conhecidas. O desenvolvimento de linkeditores envolve o conhecimento e manipulação de utilitários binários e é bastante dependente da arquitetura-alvo. Por um lado, como o mercado de *desktops* e servidores é dominado por umas poucas arquiteturas, embora o desenvolvimento de linkeditores seja trabalhoso (pois seu redirecionamento para novas arquiteturas é feito manualmente), o esforço de seu desenvolvimento é amortizado pelo seu reuso intensivo. Isto se traduz na disponibilidade de linkeditores para várias CPUs de propósitos gerais [16]. Por outro lado, no mercado de computação embarcada e, em especial, em aplicações que demandam SoCs, há uma variada gama de processadores dedicados. Neste caso, a cada nova arquitetura explorada, não se justificaria o esforço de desenvolvimento de um novo linkeditor em face das restrições de *time-to-market*. Assim, no contexto de projeto de SoCs orientados a plataforma [17], tem sido apontada a necessidade de se gerar automaticamente, para cada CPU-alvo explorada, uma série de utilitários binários, inclusive o linkeditor.

Dentre as ADLs disponíveis na literatura, há pouca informação em relação à geração de utilitários binários, principalmente a geração automática de linkeditores. CHESS [18] gera montadores e linkeditores, porém não são dados detalhes sobre o processo de redirecionamento. ISDL [11] e nML [19] mencionam a geração automática de montadores e desmontadores, porém não elaboram sua descrição. Sabe-se apenas que se gera uma gramática (a partir de arquivos Lex e Yacc) cuja compilação resulta em um montador de duas fases capaz de processar rótulos

(*labels*). Nem mesmo resultados experimentais são mostrados.

O conjunto de ferramentas denominado *New Jersey Machine-Code Toolkit* [20] [21] usa informações que podem ser descritas na linguagem SLED [22] para gerar um linkeditor, além de outras ferramentas binárias. O linkeditor gerado (mld) gera arquivos executáveis no formato a.out [23]. Resultados experimentais são mostrados para as arquiteturas MIPS e SPARC. O suporte a relocações consiste em primeiramente declarar que um operando é relocável, como segue:

```
relocatable addr
```

Após essa declaração, é possível especificar como a relocação deve ser feita através da construção `constructors`, que conecta as representações simbólica e numérica de uma instrução. Por exemplo:

```
constructors
branch^a addr { addr = L + 4 * disp22! } is L:
```

O rótulo `L` refere-se à posição da instrução e o ponto de exclamação representa uma operação de extensão de sinal.

A ADL Sim-nML [24] suporta a geração automática de montadores e desmontadores. Gera-se uma gramática (a partir de arquivos Lex e Yacc) cuja compilação resulta em um montador. Os arquivos relocáveis gerados pelo montador adotam o formato ELF [25]. A validação é feita comparando-se os arquivos relocáveis produzidos pela ferramenta com os obtidos com montadores GNU convencionais. A geração de linkeditores não é sequer mencionada.

O conjunto de ferramentas LISA [26] [27] é capaz de gerar montadores e linkeditores. O montador é composto de uma parte dita dinâmica, gerada a partir do modelo LISA, e uma parte dita estática, independente de arquitetura. O montador gera arquivos-objeto relocáveis em um formato próprio denominado LOF (LISA Object File). O linkeditor é controlado por um arquivo de comandos que descreve detalhadamente a organização de memória da arquitetura-alvo. O linkeditor lê os arquivos relocáveis no formato LOF e gera como saída arquivos executáveis no formato COFF [28]. Resultados experimentais são mostrados para as arquiteturas ARM7 uC, Analog Devices ADSP2 101, Texas Instruments C62x e C54x.

Em [29], propõe-se um modelo formal abstrato para a geração de ferramentas redirecionáveis, utilizando como substrato o pacote *GNU Binutils* [30]. Dentre as ferramentas descritas, estão um montador e um linkeditor. O modelo formal, similar ao obtido a partir de uma

ADL, descreve as instruções (ISA) e as informações de relocação. Para viabilizar a geração do linkeditor, o modelo associa uma ação de relocação a cada campo do formato de instrução que a requeira. Cada ação de relocação possui os seguintes componentes:

- **Id:** Um número que identifica a relocação;
- **ExpCode:** Indica como é calculado o valor da relocação;
- **RightShift:** Indica o valor do deslocamento à direita que será aplicado ao resultado da relocação;
- **BitSize:** Indica o número de bits menos significativos que serão extraídos do valor final da relocação;
- **BitPos:** Sua função não é explicada em [29];
- **Complain:** Contém o tipo de verificação de *overflow* [1] a ser implementada pelo linkeditor;

Como será visto na Seção 3.3, essas informações são praticamente as mesmas utilizadas internamente por uma biblioteca (BFD) do pacote *GNU Binutils*. Essa descrição explícita das informações de relocação é pouco amigável para o usuário, pois é como se estivesse alterando diretamente a biblioteca do pacote *GNU Binutils*.

A validação da técnica de geração utilizou o *benchmark* SPECInt2000 [31], e restringiu-se à arquitetura SPARC. Embora também mencionado no artigo, nenhum resultado experimental é apresentado para o processador Intel 386. Assim, embora o modelo tenha sido proposto para ser genérico, sua validação restringiu-se a uma única CPU-alvo. Portanto, não há evidência experimental para garantir que a técnica adotada é suficientemente genérica para cobrir uma ampla gama de arquiteturas.

2.2 Tradução binária

Embora alguns dos trabalhos mencionados a seguir não abordam especificamente a tradução binária, neles são discutidos conceitos importantes para sua viabilização.

O primeiro passo para a viabilização da tradução binária é a possibilidade de descrever semanticamente as instruções da arquitetura através de uma ADL. Embora algumas dessas descrições tenham como objetivo primordial a geração automática de compiladores, elas podem

[1] São três tipos: IGNORE, BIT, SIGN e UNSIGNED; a função de cada tipo não é explicada em [29].

também ser usadas para a geração automática de tradutores binários. Na ADL MIMOLA [32], a modelagem da semântica[2] de uma instrução é similar à de uma descrição de hardware em VHDL. Na ADL EXPRESSION [33], a descrição semântica de uma instrução é feita através de um formato intermediário entre a linguagem de alto nível e a CPU-alvo chamado de "Máquina Genérica" [34], que é uma arquitetura RISC com ISA similar à do MIPS. Assim, para cada instrução descrita na ADL, descreve-se um código em linguagem de montagem (da máquina genérica), cujo comportamento é equivalente ao da instrução sob descrição. O compilador gerado primeiramente traduz o código descrito em linguagem de alto nível para código descrito na linguagem da máquina genérica e, depois, faz o mapeamento para o ISA da arquitetura-alvo. Na ADL LISA [27], a semântica de uma instrução é descrita através de um código C/C++ contido dentro de uma seção chamada `BEHAVIOR`. Uma abordagem similar é adotada pela ADL ArchC [12]. Embora adequada para a geração de simuladores, essa abordagem dificulta a extração de informações para a geração de tradutores e compiladores. Em [35] propõe-se uma extensão à linguagem LISA para especificar a semântica das instruções e dar suporte à geração automática de compiladores. A proposta consiste essencialmente na adição da palavra chave `SEMANTICS` em que a semântica da instrução é descrita através de micro-operações. Essa abordagem é bastante parecida com a adotada na ADL EXPRESSION. Desta forma a seção `BEHAVIOR` seria usada para gerar o simulador (pois possui informações de mais baixo nível para possibilitar a geração de simuladores com precisão de ciclos), enquanto a seção `SEMANTICS` seria usada para gerar o compilador. A Figura 2.1 mostra a descrição do comportamento de duas instruções modeladas com um *pipeline*. O trecho mostrado não modela características como *register bypassing*, por exemplo, o que tornaria a descrição ainda mais complexa e mais difícil a extração de sua informação semântica.

A Figura 2.2 mostra a descrição da semântica das mesmas instruções cujo comportamento é mostrado na Figura 2.1.

Existem basicamente dois tipos de tradução binária: a dinâmica, feita em tempo de execução e a estática, feita previamente à execução, dando origem a um novo código executável na arquitetura-alvo. O foco dessa dissertação está na tradução binária estática.

Em [1] e [36] é apresentado o *UQBT (University of Queensland Binary Translator)*, um *framework* utilizado na tradução binária estática. Esse *framework* provê a separação entre características dependentes e independentes de arquitetura. Com isso, durante a criação de

[2]Adotaremos o termo semântica para designar o comportamento de uma instrução considerando-a como uma função, ou seja, não estamos interessados em como o comportamento será implementado (como *pipeline* e *register bypassing*, por exemplo).

```
1.  OPERATION arithm IN pipe.ID{
2.    DECLARE{
3.      GROUP opcode = { ADD || SUB };
4.      GROUP Rs1, Rs2, Rd = { reg_32 };
5.    }
6.
7.    BEHAVIOR{
8.      PIPELINE_REGISTER(pipe, ID/EX).src1 = GPR[Rs1];
9.      PIPELINE_REGISTER(pipe, ID/EX).src2 = GPR[Rs2];
10.   }
11. }
12.
13. OPERATION ADD IN pipe.EX{
14.   BEHAVIOR{
15.     int op1 = PIPELINE_REGISTER(pipe, ID/EX).src1;
16.     int op2 = PIPELINE_REGISTER(pipe, ID/EX).src2;
17.     PIPELINE_REGISTER(pipe, EX/WB).dst = op1+op2;
18.   }
19. }
```

Figura 2.1: Exemplo de uso da construção `BEHAVIOR` na ADL LISA

um tradutor binário, o programador pode manter o foco apenas nas características dependentes de arquitetura, economizando assim tempo no desenvolvimento de um tradutor. Várias linguagens são utilizadas na descrição da arquitetura, dentre as quais:

- **SLED** (*Specification Language for Encoding and Decoding*): usada na descrição da sintaxe das instruções [37];

- **SSL** (*Semantic Specification Language*): usada na descrição da semântica das instruções [38];

- **CTL** (*Control Transfer Language*): usada para identificar as instruções que realizam transferência de controle (como *jumps*, *calls*, etc);

A Figura 2.3, extraída de [1], mostra o fluxo do processo de tradução do *framework*. O arquivo binário original é decodificado e transformado em instruções em linguagem de máquina. Depois, essas instruções são transformadas dando origem a uma representação denominada *RTL (Register Transfer Lists)*, que representa os efeitos das instruções nos registradores. Essa

```
1.  OPERATION arithm IN pipe.ID{
2.    DECLARE{
3.      GROUP opcode = { ADD || SUB };
4.      GROUP Rs1, Rs2, Rd = { reg_32 };
5.    }
6.    SEMANTICS{ opcode|_C|(Rs1, Rs2)->Rd; }
7.  }
8.
9.  OPERATION ADD IN pipe.EX{
10.    SEMANTICS{ _ADD; }
11.  }
12.
13.  OPERATION SUB IN pipe.EX{
14.    SEMANTICS{ _SUB; }
15.  }
```

Figura 2.2: Exemplo de uso da construção `SEMANTICS` na ADL LISA

representação é dependente de arquitetura e é usada como representação intermediária em muitos compiladores. Ela é então traduzida para uma representação denominada *HRTL (Higher Register Transfer Language)*, que é independente de arquitetura. O código descrito em `HRTL` é transformado em código em linguagem C, compilado e ligado, resultando no arquivo binário traduzido.

São mostrados resultados experimentais para traduções entre SPARC e Pentium para três programas diferentes, cujos tamanhos variam de 61 a 204 linhas de *assembly*. O desenvolvimento desse *framework* consumiu três anos e mais de 6 anos-homem, o que indica a grande complexidade do problema.

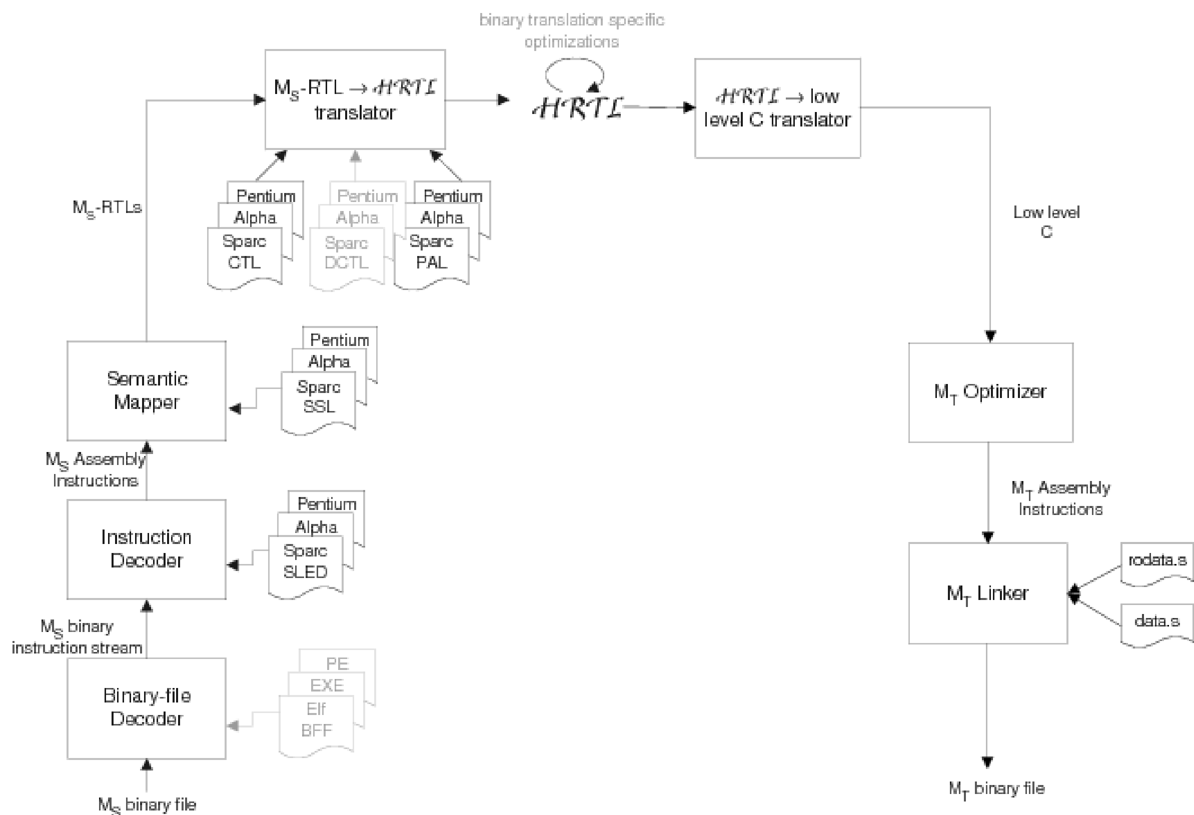

Figura 2.3: Fluxo do processo de tradução do *framework UQBT* [1]

Capítulo 3

Fundamentos, modelagem e infra-estrutura

Esta seção aborda os conceitos fundamentais que amparam a geração automática de utilitários binários, bem como a infra-estrutura necessária para sua implementação. Como o foco principal desta dissertação é a geração automática de linkeditores, seus fundamentos são preliminarmente discutidos.

3.1 Fundamentos da linkedição

O papel de um linkeditor é o de viabilizar a manipulação modular de programas, em vez de se ter que tratar um grande bloco monolítico. Isso acarreta uma enorme economia de recursos computacionais. Especialmente se o usuário estiver usando rotinas que dificilmente mudam. Caso o programa fosse tratado como um grande bloco monolítico, ao se alterar alguma parte do programa, todo ele deveria ser recompilado, inclusive as funções que não sofreram alteração. Para resolver isso, cada procedimento ou grupo de procedimentos é montado de maneira independente. Assim, ao alterar um procedimento, apenas o procedimento modificado precisa ser remontado.

Os arquivos que foram montados de maneira independente e possuem informação para linkedição recebem o nome de *arquivos-objeto*. O arquivo composto da união de um ou mais arquivos-objeto, e que está pronto para execução é chamado de *arquivo executável*.

Dado um conjunto de arquivos em linguagem de máquina, o papel do linkeditor é o de ligá-los em um único arquivo executável [39].

Cada módulo é compilado e montado como se seu código iniciasse no endereço 0. Ademais, como os módulos não são estanques, um módulo pode conter referências externas para outros módulos. Essencialmente, a função de um linkeditor é a de criar um único bloco de

código, realizando duas tarefas principais:

- **Ligação**: consiste na resolução de referências externas, o que requer a conversão de endereços simbólicos em endereços numéricos.
- **Relocação**: consiste no reposicionamento dos módulos em espaços de endereçamento contíguos, o que requer a edição de endereços absolutos utilizados dentro de um módulo, ajustando-os ao seu novo espaço de endereçamento.

A Figura 3.1 ilustra um exemplo de linkedição. Os módulos M1, M2 e M3 representam três arquivos-objeto e E1 representa o arquivo executável (embora seus conteúdos estejam esquematicamente representados em linguagem de montagem para melhor visualização do processo). Como cada módulo é posicionado imediatamente após o outro, o endereço inicial de um módulo é o endereço final do módulo anterior mais 1. Esse endereço é conhecido como *endereço-base*. Note que há três instruções referenciando endereços absolutos (que correspondem a endereços simbólicos já resolvidos pelo montador no escopo de um módulo). Tais instruções precisam ter seus endereços editados para fins de relocação. O endereço relocado é obtido pela soma do valor original com o valor do *endereço-base* do módulo onde reside. Por exemplo, no módulo M1 a instrução `j 04` torna-se `j 07`, pois o endereço-base de M1 em E1 é 3. Note que no módulo M2 há uma referência externa (`j label`) a uma instrução especificada no módulo M3, cujo endereço simbólico precisa ser convertido em numérico. Assim, a instrução `j label` do módulo M2 torna-se `j 9` no arquivo executável, uma vez que a função marcada com label no módulo M3 foi relocada para o endereço 09 no arquivo E1.

3.1.1 Informações de relocação

Para que essas modificações sejam possíveis, é necessário prover informações de relocação, descrevendo **quais** campos da instrução devem ser editados, e **como** devem ser modificados. Por exemplo, considere a instrução `j` do MIPS. O campo a ser editado corresponde aos 26 bits menos significativos da instrução. O novo valor desse campo é obtido da seguinte forma: dado um endereço-alvo de 32 bits, deve-se descartar os 4 bits mais significativos e os 2 bits menos significativos, pois tais bits são concatenados pela CPU, em tempo de execução, para formar o endereço efetivo de memória [39].

As informações de relocação são inseridas nos arquivos-objeto, que são então chamados de *arquivos-objeto relocáveis*. Um arquivo-objeto relocável é subdividido em seções,

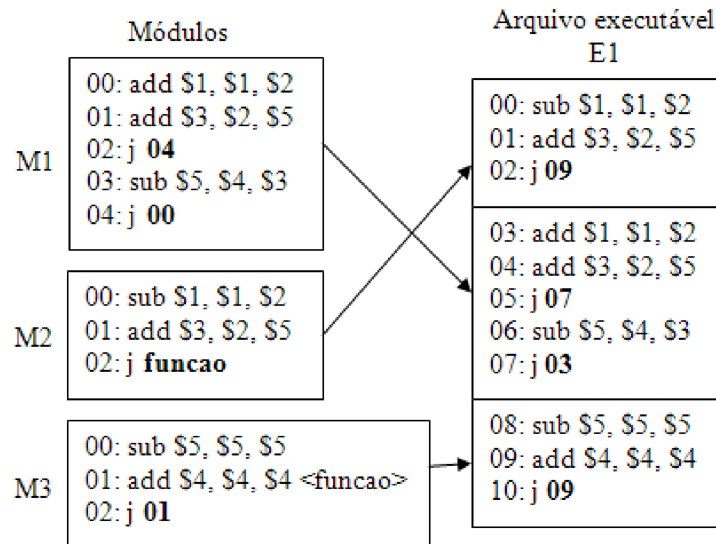

Figura 3.1: Exemplo do processo de linkedição

sendo que cada seção deve ser ligada com sua congênere em outro arquivo objeto. Um arquivo-objeto relocável consiste basicamente de três seções:

- **Seção de código (.text)**: contém o código executável do programa; é uma seção apenas de leitura.
- **Seção de dados (.data)**: contém dados inicializados estaticamente.
- **Seção de dados não inicializados (.bss)**: tem como função reservar espaço de memória para variáveis globais não inicializadas; esse espaço de memória será inicializado com valor 0, quando o programa começa a executar; entradas nessa seção podem ser criadas com a pseudo-operação `.comm`.

Vamos ilustrar o conteúdo das duas últimas seções através de exemplos. O código "`int temp[] = {10, 20, 30};`" em linguagem C gera o seguinte código em linguagem de montagem, que corresponde a três entradas na seção de dados inicializados estaticamente:

```
.data
  .long 10
  .long 20
  .long 30
```

Por outro lado, a variável global "`int temp[256]`" em linguagem C gera o seguinte código em linguagem de montagem, na seção de dados não inicializados:

```
.comm temp, 1024
```

Note que, como um inteiro possui 4 bytes e foi declarado um array com 256 posições, a quantidade de memória a ser reservada deve ser 256 X 4, que é igual a 1024 bytes.

No casos dos arquivos-objeto relocáveis, as seguintes seções também são relevantes no contexto deste trabalho:

- **Code relocation section (.rel.text):** contém informações de relocação referente à seção `.text`.
- **Data relocation section (.rel.data):** contém informações de relocação referente à seção `.data`. As relocações sobre dados [1] são mais simples que as relocações sobre código, pois retornam o endereço absoluto do rótulo, sem modificações.
- **Symbol table (.symtab):** contém os símbolos importados e exportados pelo arquivo; pode estar presente também em arquivos executáveis, pois é utilizada pelo depurador.

Dependendo do formato adotado para o arquivo-objeto, podem existir variantes dessas seções.

3.2 Descrição de CPUs

Como visto no Capítulo 1, o ponto de partida para a geração automática de ferramentas de desenvolvimento de software para CPUs embarcadas costuma ser uma ADL. Dentre as ADLs reportadas na literatura, pode-se citar nML [9], LISA [10], ISDL [11] e ArchC [12].

3.2.1 Modelagem básica de CPUs em ArchC

Nesta dissertação, adota-se a ADL ArchC [12] [40] por dois motivos principais:

- A disponibilidade de ferramentas de domínio público (ao contrário da maior parte das ADLs);
- A viabilidade de geração automática de modelos de CPUs em SystemC [6] compatíveis com o estilo TLM [8];

As principais características de ArchC são:

[1] Da mesma forma que as instruções, os dados tem seu endereço efetivo definido após a linkedição. Tomemos um exemplo: supondo que certo rótulo indique que um dado esteja na posição 0x100 e, após a linkedição, a seção de dados inicie na posição 0x1000, caso o endereço seja de 32 bits, ele deverá ser relocado para o endereço 0x00001100.

- Capacidade de descrição do conjunto de instruções;
- Duas modalidades de modelos: puramente funcional e com precisão de ciclos;
- Suporte à descrição de instruções multi-ciclo;
- Suporte à descrição de instruções executando em pipeline;
- Suporte à descrição de hierarquia de memória;
- Suporte à descrição de linguagem de montagem.

Ademais, no pacote ArchC, estão disponibilizadas as seguintes ferramentas de geração automática:

- Simuladores interpretados (`acsim`);
- Simuladores compilados (`accsim`);
- Montadores (`acasm`).

Vale ressaltar que o gerador de montadores acasm utiliza o pacote *GNU Binutils* como substrato, o qual será descrito na Seção 3.3.

3.2.1.1 Construções da linguagem

A seguir descrevem-se as construções da linguagem ArchC mais relevantes para este este trabalho:

set_endian: Indica a ordem de numeração dos bytes na palavra de dados (*big_endian* ou *little_endian*).

ac_format: Descreve os campos do formato de instrução com o nome e tamanho em bits, conforme mostra a Figura 3.2.

```
1.   ac_format Type_R  = "%op:6 %rs:5 %rt:5 %rd:5 %shamt:5 %func:6";
2.   ac_format Type_I  = "%op:6 %rs:5 %rt:5 %imm:16:s";
3.   ac_format Type_J  = "%op:6 %addr:26";
```

Figura 3.2: Descrição ArchC dos formatos de instrução do MIPS

ac_instr: Faz a associação entre um formato e as instruções pertencentes a este formato, conforme ilustra a Figura 3.3.

```
1.   ac_instr<Type_I> lw, addi, beq;
2.   ac_instr<Type_R> add, slt, instr_and, nop, mult, div, jr;
3.   ac_instr<Type_J> j, jal;
```

Figura 3.3: Descrição ArchC do mapeamento entre instruções e formatos

ac_asm_map: Faz o mapeamento entre os nomes simbólicos de registradores e seus valores, conforme ilustra a Figura 3.4.

```
1.   ac_asm_map reg {
2.     "$"[0..31] = [0..31];
3.     "$zero" = 0;
4.     "$at" = 1;
5.     "$kt"[0..1] = [26..27];
6.     "$gp" = 28;
7.     "$sp" = 29;
8.     "$fp" = 30;
9.     "$ra" = 31;
10.  }
```

Figura 3.4: Descrição ArchC do mapeamento entre nomes e números de registradores do MIPS

set_asm: Descreve a sintaxe de uma instrução em linguagem de montagem. É a construção mais importante para a geração automática de montadores e pode ser reaproveitada também para a geração automática de linkeditores, como veremos no Capítulo 4.

Um exemplo de seu uso pode ser visto a seguir:

```
add.set_asm("add %reg, %reg, %reg", rd, rs, rt);
```

A sintaxe dessa construção é similar à função `printf`/`scanf` da linguagem C. Os caracteres compreendidos entre o início da string até o primeiro espaço formam o mnemônico da instrução. Depois, opcionalmente, vem a lista de operandos. Após cada caractere “%” deve vir um formatador que pode ser definido através da construção `ac_asm_map` ou pode ser um formatador pré-definido da linguagem ArchC. O formatador indica que tipo de informação o montador deve reconhecer no trecho do código em linguagem de montagem. Para cada formatador, deve ser

associado um campo do formato da instrução (após a vírgula). O valor do formatador então será codificado nesse campo. Os formatadores pré-definidos em ArchC são:

- **Expressão aritmética (`exp`)**: por exemplo, `lw $1, %lo(label+4)($2)`.
- **Valores imediatos (`imm`)**: por exemplo, `lw $1, 4($2)`.
- **Valores simbólicos (`addr`)**: por exemplo, `lw $1, %lo(label)($2)`.

Os formatadores `imm` e `addr` são especializações de `exp`, ou seja, o formatador `exp` reconhece também tudo aquilo que é reconhecido por `addr` e `imm`. As linhas 1, 2, 3 e 6 da Figura 3.5 ilustram o uso dessa construção.

set_decoder: Descreve os campos de decodificação da instrução, ou seja, os campos que possuem valores fixos e que caracterizam a instrução. As linhas 4 e 7 da Figura 3.5 mostram exemplos desta construção.

```
1.   lw.set_asm("lw %reg, \%lo(%exp)(%reg)", rt, imm, rs);
2.   lw.set_asm("lw %reg, (%reg)", rt, rs, imm=0);
3.   lw.set_asm("lw %reg, %imm(%reg)", rt, imm, rs);
4.   lw.set_decoder(op=0x23);
5.
6.   add.set_asm("add %reg, %reg, %reg", rd, rs, rt);
7.   add.set_decoder(op=0x00, func=0x20);
```

Figura 3.5: Descrição ArchC de linguagem de montagem e de decodificação

pseudo_instr: Esta construção serve para descrever instruções sintéticas. Descreve-se sua sintaxe entre parênteses, usando-se o mesmo formato da construção `set_asm`, exceto pelo mapeamento de operandos, que não é feito para campos do formato de instrução, mas sim para campos de outra(s) instruções. Para designar o primeiro operando da instrução sintética usa-se `%0`, para o segundo usa-se `%1`, e assim por diante, conforme o exemplo da Figura 3.6.

```
1.   pseudo_instr("bge %reg, %reg, %exp") {
2.     "slt $at, %0, %1";
3.     "beq $at, $zero, %2";
4.   }
```

Figura 3.6: Descrição ArchC de uma pseudo-instrução do MIPS

Em muitos casos é necessário alterar o valor a ser codificado em um campo de uma instrução. Para isso, são usadas construções denominadas de *modificadores*.

Para justificar sua necessidade e ilustrar o seu uso, vamos analisar um exemplo. Seja a instrução `beq` do MIPS:

```
beq $1, $2, label
```

Suponha que `label` refira-se ao endereço 0xF0. Se o valor 0xF0 fosse codificado no campo correspondente, a codificação do valor absoluto estaria errada, pois essa instrução realiza um desvio relativo ao PC. Além disso, a arquitetura MIPS realiza um desvio relativo ao valor de PC+4. Por isso, primeiro deveríamos subtrair o endereço associado a `label` do endereço em que a instrução se encontra adicionado de 4. Suponha que o endereço da instrução é 0xE0; então, 0xF0 - (0xE0 + 4) = 0xC. O valor 0xC ainda não é o valor a ser codificado, pois precisa ser alinhado ao tamanho da palavra da arquitetura MIPS (4 bytes), ou seja, seus dois últimos bits são eliminados como segue: 0xC >> 2 = 0x3. Portanto 0x3 é o valor a ser codificado no campo correspondente a `label`.

Para resolver problemas desta natureza, são definidos modificadores, que atuam sobre o valor associado aos formatadores pré-definidos da linguagem.

Supondo que `n` seja um número inteiro positivo, `s` indique que há extensão de sinal, `u` indique que não há extensão de sinal, `c` indique o uso de *carry* e `b` indique que `n` é um valor negativo, são os seguintes os formatadores definidos em ArchC:

- `L[n][s|u]`: Seleciona os `n` bits menos significativos. Se `n` não for especificado adota-se o tamanho do campo como *default*. Por padrão, não há extensão de sinal. Esse é o modificador padrão, ou seja `%imm` é igual a `%immL`. Por exemplo, ao aplicarmos o modificador `L16` ao valor 0x1234**5678**, o resultado será 0x5678.

- `H[n][c][s|u]`: Seleciona os `n` bits mais significativos. Se `n` não for especificado adota-se o tamanho do campo como *default*. Por padrão, não há extensão de sinal. Quando `c` é especificado, o primeiro bit descartado é aplicado como *carry*. Por exemplo, ao aplicarmos o modificador `H16` ao valor 0x**1234**5678, o resultado será 0x1234.

- `R[n][b]`: Subtrai do valor associado ao formatador o valor do PC. Se um valor `n` for especificado esse valor será adicionado ao resultado. Por exemplo, se o valor de PC é 100 e a instrução está no endereço 80, ao aplicarmos o modificador `R`, o valor final será 20.

- `A[n][u|s]`: Considera que o valor associado ao formatador usa um endereçamento alinhado em `n` bytes, ou seja, o valor é deslocado para a direita de log2(`n`). Por padrão, a extensão de sinal é adotada. Caso `n` não seja especificado, adota-se como *default* o número de bytes do tamanho da palavra de arquitetura. Por exemplo, ao aplicarmos o modificador `A4` ao valor binário **110011**00, o resultado será 110011.

Pode-se especificar mais de um modificador para determinado formatador, porém os modificadores `R` e `H` são mutuamente exclusivos. Esse modificadores podem ser declarados em qualquer ordem, porém sua ordem de aplicação é sempre a mesma: `R`, `H/L` e `A`.

Tentou-se reutilizar ao máximo as construções originais da ADL ArchC para fins de relocação. Entretanto, foi necessário introduzir uma variante ao modificador `R` original, como explicado a seguir.

O modificador original `R` é utilizado em desvios relativos ao PC, os quais não requerem relocação. Entretanto, se a instrução de desvio possui uma referência externa ao módulo onde reside, o montador deve gerar informação de relocação, exclusivamente para viabilizar sua futura linkedição. Para esse fim, foi concebido o modificador `r`, que embora também codifique o desvio como relativo ao PC (como o modificador original `R`), indica que a referência é externa para fins de relocação.

A Figura 3.7 mostra um trecho da descrição do processador MIPS em que todas as construções anteriormente descritas são usadas.

3.3 A infra-estrutura do pacote *GNU Binutils*

O *pacote GNU Binutils* [30] [16] é um conjunto de ferramentas para a manipulação de arquivos binários, desenvolvido para ser usado em conjunto com o pacote denominado *GNU Compiler Collection* (gcc) [41]. Dentre as ferramentas que compõem o pacote estão um montador (gas), um linkeditor (ld), gerenciadores de bibliotecas (ranlib e ar), visualizadores de arquivos binários (nm, objdump, readelf, size), manipuladores de arquivos binários (strip, objcopy, addr2line), etc. Para facilitar a redirecionabilidade das ferramentas, a maior parte do pacote é independente de arquitetura. Portanto, para portar o pacote para uma nova arquitetura, basta adicionar um novo módulo específico para essa arquitetura. Muitas arquiteturas já são suportadas pelo pacote, incluindo máquinas RISC (MIPS, SPARC, POWERPC, ARM, etc) e CISC (i386, VAX, Motorola 68000, etc). Essa modularidade motiva a escolha desse pacote como infra-estrutura básica

```
1.  AC_ISA(mips1){
2.    ac_format Type_R  = "%op:6 %rs:5 %rt:5 %rd:5 %shamt:5 %func:6";
2.    ac_format Type_I  = "%op:6 %rs:5 %rt:5 %imm:16:s";
3.    ac_format Type_J  = "%op:6 %addr:26";
4.
5.    ac_instr<Type_I> lw, beq;
6.    ac_instr<Type_R> instr_and;
7.    ac_instr<Type_J> j;
89.
99.   ac_asm_map reg {
10.      "$"[0..31] = [0..31];
11.      "$zero" = 0;
12.   }
13.
14.   ISA_CTOR(mips1){
15.     lw.set_asm("lw %reg, \%lo(%exp)(%reg)", rt, imm, rs);
16.     lw.set_asm("lw %reg, (%reg)", rt, rs, imm=0);
17.     lw.set_asm("lw %reg, %imm(%reg)", rt, imm, rs);
18.     lw.set_decoder(op=0x23);
19.
20.     instr_and.set_asm("and %reg, %reg, %reg", rd, rs, rt);
21.     instr_and.set_decoder(op=0x00, func=0x24);
22.
23.     j.set_asm("j %expA", addr);
24.     j.set_decoder(op=0x02);
25.
26.     beq.set_asm("beq %reg, %reg, %expR4A", rs, rt, imm);
37.     beq.set_asm("b %expR4A", imm, rs=0, rt=0);
28.     beq.set_asm("beqz %reg, %expR4A", rs, imm, rt=0);
29.     beq.set_decoder(op=0x04);
30.
31.     pseudo_instr("bge %reg, %reg, %exp") {
32.       "slt $at, %0, %1";
33.       "beq $at, $zero, %2";
34.     }
35.   };
36. };
```

Figura 3.7: Trecho da descrição da arquitetura MIPS em ArchC

para a geração automática de ferramentas. Além disso, as ferramentas já disponíveis para essas arquiteturas podem servir como referência para a validação das ferramentas binárias geradas. O pacote é dividido em duas bibliotecas que são compartilhadas pelas ferramentas, a biblioteca `BFD` e a biblioteca `Opcodes`, além de módulos específicos de cada uma das ferramenta.

A Figura 3.8, adaptada de [13] mostra a estrutura do linkeditor do pacote *GNU Binutils*. Ele é composto de um módulo principal, denominado `core`, que recebe os arquivos-objeto relocáveis como entrada. O `core`, que é independente de arquitetura, se comunica com a biblioteca BFD e com um módulo dependente de arquitetura para executar a linkedição, e assim criar um arquivo executável.

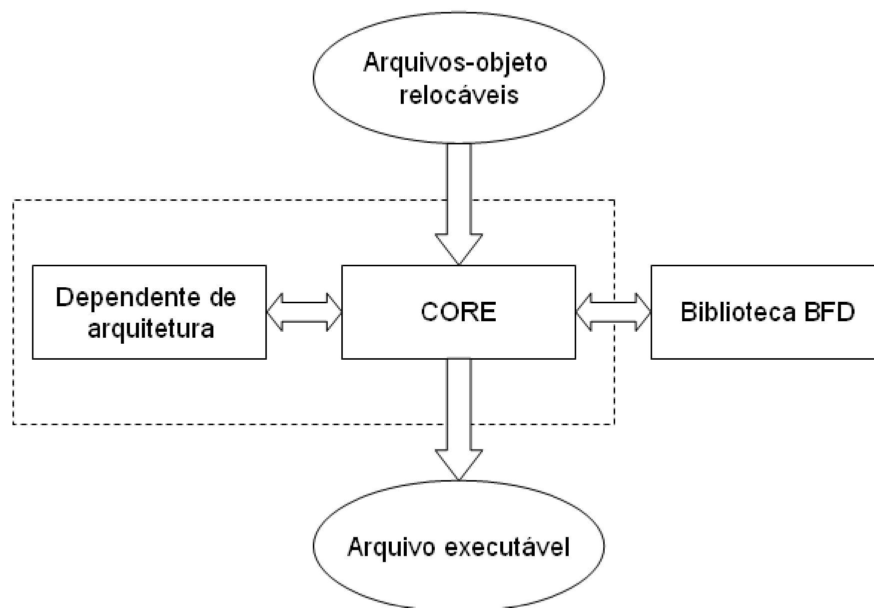

Figura 3.8: Estrutura do linkeditor do pacote *GNU Binutils*

3.3.1 Biblioteca Opcodes

A biblioteca `Opcodes` descreve o conjunto de instruções de determinada arquitetura, em termos de código operacional, formatos de instruções, operandos, etc. Não há uma padronização da forma de implementação dessa biblioteca para uma determinada arquitetura. Porém, geralmente, a implementação é feita através de uma tabela que descreve as várias instruções.

3.3.2 Biblioteca BFD

A biblioteca BFD tem como objetivo prover meios para operar em objetos binários. Vários tipos de formatos binários são suportados (como, por exemplo, a.out, coff, elf e ecoff). Para viabilizar o porte para vários tipos de formatos de arquivos e várias arquiteturas, ela é dividida em basicamente três partes:

1. **Front-end:** Genérico, independente de arquitetura e independente de formato de arquivo; possui funções para a manipulação dos arquivos e se ampara em um formato intermediário.

2. **Back-end do formato de arquivo:** Específico para cada formato binário; ao se adicionar um novo formato à biblioteca, este módulo deve ser estendido.

3. **Back-end do formato de arquitetura:** Específico para cada arquitetura; possui basicamente uma tabela com a descrição dos tipos de relocações da arquitetura, além de algumas funções auxiliares; ao se adicionar uma nova arquitetura à biblioteca, este módulo deve ser estendido.

A estrutura da biblioteca BFD, acima descrita, é mostrada na figura 3.9, adaptada de [13], que também ilustra sua relação com os utilitários binários do pacote *GNU Binutils*.

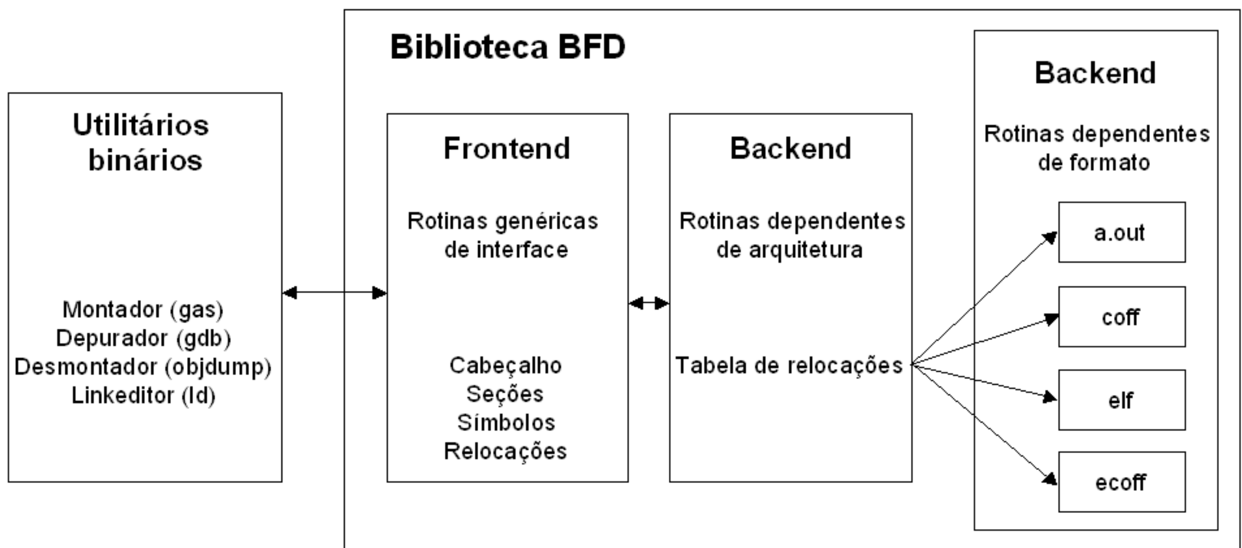

Figura 3.9: Estrutura da biblioteca BFD do pacote *GNU Binutils*

As relocações são descritas em uma *tabela de relocações*, onde cada entrada corresponde a um tipo de relocação da arquitetura. Após ter sido realizado o cálculo de relocação, como descrito na Seção 3.1, o valor final sofre algumas modificações antes de ser inserido na

instrução. A forma como são feitas tais modificações, bem como outras informações sobre o tipo de relocação são descritos nessa tabela. Cada entrada na tabela `relocation_table` pode ser definida como uma tupla:

```
relocation_table_entry = (type, rightshift, size, bitsize,
                          pc_relative, bitpos, complain_on_overflow,
                          special_function, name, partial_inplace,
                          src_mask, dst_mask, pcrel_offset)
```

A descrição detalhada de cada um dos elementos dessa tupla é elaborada no Apêndice A.

Capítulo 4

Geração de linkeditores

Este capítulo descreve sucintamente a abordagem adotada para a geração de utilitários binários a partir da descrição ADL de uma arquitetura. Em seguida, descreve-se detalhadamente a técnica proposta para a geração automática de linkeditores, foco principal desta dissertação.

4.1 A abordagem adotada

A partir de uma descrição da arquitetura na ADL ArchC, através do gerador de utilitários binários, são gerados os arquivos dependentes de máquina do pacote *GNU Binutils*, que, ao serem compilados junto com os arquivos independentes de arquitetura, geram os utilitários binários. O fluxo de geração dos utilitários binários é mostrado na Figura 4.1.

Esta abordagem foi utilizada para a geração automática de montadores [13], linkeditores [42], depuradores e desmontadores [14] no âmbito de diferentes dissertações de mestrado. A contribuição específica desta dissertação é descrita na próxima seção.

4.2 A geração automática de linkeditores

A geração automática de linkeditores implica em modificações nas bibliotecas Opcodes e BFD e um umas poucas alterações no gerador de montadores.

Os principais algoritmos responsáveis pela geração automática são descritos na Seção 4.2.1 e 4.2.2, onde reside a maior parte da contribuição desta proposta. As Seções 4.2.3 e 4.2.4 abordam aspectos puramente técnicos, mas são apresentados para que se tenha uma visão completa da infra-estrutura de implementação.

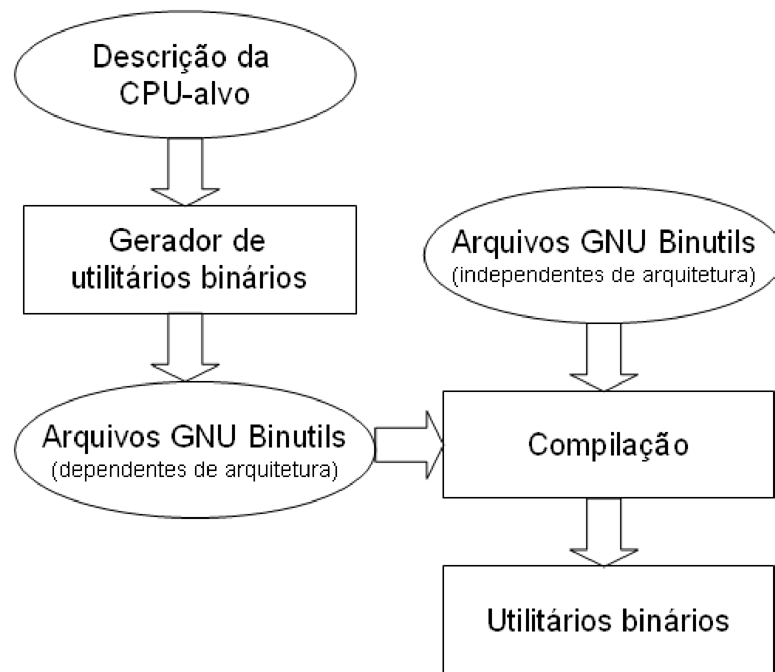

Figura 4.1: Fluxo de geração dos utilitários binários

4.2.1 Geração da biblioteca Opcodes

A geração da biblioteca Opcodes foi desenvolvida originalmente em [13] para a geração automática de montadores. Como essa biblioteca tem relevância para o linkeditor, sua geração será descrita a seguir, mas resumidamente.

O núcleo desta biblioteca é a *tabela de instruções* que descreve cada instrução definida na ADL. Cada entrada na tabela `instruction-table` é uma tupla definida como:

```
instruction-table-entry = (mnemonic, opinfo-list, image, format-id),
```

onde cada elemento é descrito a seguir:

- **mnemonic:** corresponde ao mnemonico da instrução;
- **opinfo-list:** lista de campos da instrução;
- **image:** codificação parcial da instrução;
- **format-id:** formato da instrução.

Cada elemento da lista `opinfo-list` é uma tupla:

```
opinfo-list-entry = (operand-type, field-id),
```

onde cada elemento é descrito a seguir:

- **operand-type:** tipo de operando. (Por exemplo `reg`, `imm`, `exp`);
- **field-id:** campo do formato da instrução.

Para possibilitar a geração automática de linkeditores, este trabalho propõe a adição do campo `reloc-id` na tupla `opinfo-list-entry`. A função desse campo será descrita na Seção 4.2.2

4.2.2 Geração da biblioteca BFD

Originalmente, a geração automática da biblioteca BFD para fins de montagem [13] praticamente não existia, restringindo-se à geração de algumas constantes necessárias para sua compilação correta, já que essa biblioteca tinha pouca relevância para o montador, pois este não gerava informações de relocação. Note que, como foi descrito na Seção 3.3.2, a biblioteca BFD possui módulos dependentes da arquitetura e do formato de arquivo binário. Como o foco dessa dissertação é a geração automática a partir de ADL, a implementação restringiu-se à geração do módulo dependente da arquitetura. Por simplicidade e sem perda de generalidade, adotou-se o formato de arquivo ELF.

Essencialmente, a biblioteca BFD trata de informações de relocação. Seu principal elemento é a *tabela de relocações*, onde são descritos os tipos de relocações. Procurou-se reusar ao máximo o suporte já existente na ADL para descrever as informações de relocação. Por exemplo, a ADL ArchC possui a palavra reservada `set_asm` (descrita na Seção 3.2.1.1) em que são associados modificadores a formatadores da sintaxe da instrução, os quais são associados aos campos da instrução. Originalmente, o gerador de montadores `acasm` utilizava esses modificadores para descrever informações para a resolução de rótulos. A técnica aqui proposta ampara-se na constatação de que os modificadores também podem ser usados para montar a tabela de relocações.

Dada a descrição ADL de uma CPU-alvo, o Algoritmo 4.1 descreve a geração automática da tabela de relocações. O laço externo (delimitado pelas linhas 2 e 18) visita cada uma das instruções na descrição ADL. O laço interno (delimitado pelas linhas 3 e 17) percorre cada um dos campos de uma dada instrução que especificam operandos. Se o campo de instrução visitado for do tipo `addr` ou `exp` (linha 4), e não possuir o modificador `R` (linha 6), ele pode requerer relocação. Para isso, um novo tipo de relocação `relocEntry` é criado para ser possivelmente inserido na tabela (linha 7). A função `NewRelocationEntry` será posteriormente descrita (Al-

goritmo 4.2). A função `InfoExtraction` (linha 5), cujo algoritmo é omitido por simplicidade, serve meramente para extrair da descrição ADL as informações necessárias para se criar uma nova entrada na tabela de relocações, a partir da instrução e do tipo do campo sendo visitado. Entretanto, essa entrada só é efetivamente inserida na tabela se nela não existir uma entrada equivalente (linhas 8 a 13). O elemento `reloc-id` do campo da instrução sendo visitada recebe um ponteiro para a entrada recém-criada na tabela (linha 14). Ao final, a função `CreateDataRelocations` (linha 19) cria os tipos de relocações que atuam sobre a seção de dados (ver Seção 3.1.1). Esses tipos de relocações são independentes de arquitetura (são seis as entradas criadas por esta função e elas atuam sobre dados de 8, 16 e 32 bits).

Algoritmo 4.1 Geração da tabela de relocações

1: *id* ← 0;
2: **for each** *instruction* **do**
3:    **for each** *operand* **do**
4:       **if** *operand-type* **is** ‘addr’ or ‘exp’ **then**
5:          *(value_hl, field_size, align, value_a, word_size, position, carry, ext_pc_rel, pc_rel)* ← InfoExtraction(*instruction*, *operand-type*);
6:          **if** *pc_rel* **is** FALSE **then**
7:             *relocEntry* ← NewRelocationEntry(*value_hl, field_size, align, value_a, word_size, position, carry, ext_pc_rel, id*);
8:             **if** there exists an equivalent entry *equivEntry* in *relocation-table* **then**
9:                *relocEntry* ← *equivEntry*;
10:            **else**
11:               insert *relocEntry* in *relocation-table*;
12:               *id* ← *id* + 1;
13:            **end if**
14:            let *reloc-id* of *operand* be a reference to *relocEntry*;
15:         **end if**
16:      **end if**
17:   **end for**
18: **end for**
19: CreateDataRelocations();

A extração de dados da descrição da ADL é feita pela função `InfoExtraction` da seguinte forma:

- **value_hl**: Valor associado ao modificador `H` ou `L`.
- **field_size**: Tamanho do campo que está sendo tratado (extraído da construção `ac_format`);
- **align**: Indica a presença do modificador `A`;
- **carry**: Indica a presença do modificador `C`;
- **pc_rel**: Indica a presença do modificador `R`;
- **ext_pc_rel**: Indica a presença do modificador `r`;
- **value_a**: Valor associado ao modificador `A`;
- **word_size**: Tamanho da palavra de dados da arquitetura (extraído da construção `ac_wordsize`);
- **position**: Posição do campo que está sendo tratado (extraído da construção `ac_format`).

A função `NewRelocationEntry`, descrita no Algoritmo 4.2 cria os campos da tupla `relocation_table_entry` descrita na Seção 3.3.2, que representa uma entrada na tabela de relocações. A função `GenerateName` (linha 30) cria um nome único que expressa os dados contidos na entrada. A função `CreateMask` (linha 33) gera uma máscara que reflete o campo associado ao modificador; por exemplo, o campo `imm` do formato `Type_I` o qual é mostrado na Figura 3.2, gera a máscara 0x0000FFFF. A função `GenericFunction` é a função genérica da Biblioteca BFD para resolução de relocações e a função `CarryFunction` é uma função desenvolvida no âmbito deste trabalho para tratar o efeito do modificador `C` de forma genérica.

4.2.3 Modificações no gerador de montadores

O gerador de montadores `acasm` utiliza esses modificadores para descrever informações para a resolução de *labels*. Entretanto, como informações de relocação precisam também ser inseridas no arquivo objeto para torná-lo relocável, o `acasm` foi estendido para gerar montadores capazes de inserir tais informações como insumos para o linkeditor. Como visto na Seção 4.2.1, cada operando pode possuir uma relocação associada. Desta forma, ao montar uma instrução, o `acasm` verifica na tabela de instruções se os campos da instrução possuem um tipo de relocação associado. Em caso positivo, são geradas as informações de relocação para aquela instrução no arquivo objeto relocável.

Algoritmo 4.2 Criação de uma entrada na tabela de relocações

```
1: procedure NewRelocationEntry(value_hl, field_size,align, value_a, word_size,position,carry,ext_pc_rel, id)
2: new_value_hl ← value_hl
3: if value_hl != 0 then
4:     new_value_hl ← field_size
5: end if
6: new_value_al ← 0
7: if align is true then
8:     if value_a = 0 then
9:         new_value_al ← log2(word_size/8)
10:    else
11:        new_value_al ← log2(value_a)
12:    end if
13: end if
14: rightshift ← new_value_al
15: if high is true then
16:     new_value_al ← word_size - new_value_hl
17: end if
18: bitpos ← word_size - position - field_size
19: reloc_size ← log2(word_size/8)
20: bit_size ← new_value_hl;
21: bitpos ← word_size - position - field_size;
22: type ← id;
23: pc_relative ← ext_pc_rel;
24: complain_on_overflow ← false;
25: if carry is true then
26:     special_function ← CarryFunction;
27: else
28:     special_function ← GenericFunction;
29: end if
30: name ← GenerateName(type, rightshift, size, bitsize, pc_relative, bitpos, special_function);
31: partial_inplace ← false;
32: src_mask ← 0;
33: dst_mask ← CreateMask(field_size, position);
34: pcrel_offset ← true;
35: return (type, rightshift, size, bitsize, pc_relative, bitpos, complain_on_overflow, special_function, name,
    partial_inplace, src_mask, dst_mask, pcrel_offset);
```

4.2.4 Compilação do linkeditor

Como grande parte do mecanismo de relocação reside na biblioteca BFD (Seção 4.2.2), o processo de redirecionamento do linkeditor resume-se à descrição do modelo de memória através de um *script* principal e de *scripts* auxiliares para automatizar a compilação do pacote.

Cada arquitetura-alvo deve ter seu modelo de memória descrito através de um *script* de linkedição. Escrito em uma linguagem específica [30], o *script* descreve como as seções dos arquivos de entrada são mapeadas para o arquivo de saída. Em outras palavras, o *script* descreve o *layout* de memória do arquivo de saída. A Figura 4.2 mostra um exemplo de *script* de linkedição. Note que é criada uma seção `.text` com mesmo conteúdo das seções `.text` dos arquivos de entrada, o qual inicia na posição 0x10000. O mesmo acontece com a seção `.data`, que tem seu início na posição 0x20000 e a seção `.bss`, que inicia em 0x30000.

```
01. SECTIONS
02. {
03.    . = 0x10000;
04.    .text : { *(.text) }
05.    . = 0x20000;
06.    .data : { *(.data) }
07.    . = 0x30000;
08.    .bss : { *(.bss) }
09. }
```

Figura 4.2: Exemplo de um *script* de linkedição que descreve o *layout* do arquivo de saída

A ferramenta `aclink` gera automaticamente um arquivo de configuração (`emulparams/[arch]elf.sh`) contendo o símbolo inicial[1] para o linkeditor e o endereço inicial do código (seção `.text`). O arquivo de configuração possui os seguintes valores, que são comuns para todas as arquiteturas:

- **Símbolo inicial:** _start
- **Início da seção .text:** 0x0

Esses valores satisfazem as restrições de memória dos simuladores ArchC, com isso garantindo que os arquivos gerados pelo linkeditor rodarão nos simuladores gerados a partir do modelo ArchC da arquitetura, durante a exploração do espaço de projeto. Note, no entanto, que

[1] Rótulo que o linkeditor irá considerar como o início do programa.

esses valores-padrão não limitam o uso do linkeditor gerado em outros contextos, pois essas mesmas informações podem ser passadas ao linkeditor, em tempo de linkedição, através de parâmetros na linha de comando ou através de um *script* de linkedição similar ao mostrado na Figura 4.2.

4.2.5 Os arquivos gerados

A Figura 4.3 mostra a árvore com os arquivos gerados automaticamente pela ferramenta, onde `[arch]` representa o nome da arquitetura. Além desses, alguns arquivos da árvore do pacote *GNU Binutils*, que servem para a compilação dos utilitários binários (como `Makefile.in` e `configure`), são também modificados automaticamente.

```
- binutils
 - bfd
  . cpu-[arch].c
  . elf32-[arch].c
 - opcodes
  . [arch]-opc.c
 - gas
  - config
   . tc-[arch].h
   . tc-[arch].c
 - ld
  - emulparams
   . [arch]elf.sh
 - include
  - elf
   . [arch].h
  - opcode
   . [arch].h
```

Figura 4.3: Arquivos gerados pela ferramenta

4.2.6 Extensões ao trabalho proposto

Portar o mecanismo de relocação da biblioteca BFD é um dos aspectos mais difíceis ao tentar redirecionar o pacote *GNU Binutils* para uma nova arquitetura. A própria documentação da biblioteca BFD admite essa dificuldade [43]: “Clearly the current BFD relocation support is in bad shape”. Seu mecanismo é baseado em uma tabela de relocações e uma função de relocação

genérica (como foi mostrado na Seção 3.3.2). Isso impõe severas restrições ao mecanismo de relocação adotado pela biblioteca BFD. Por exemplo, instruções cujo comprimento não é múltiplo de 8 não podem ser especificadas. Essa abordagem baseada em tabela e em uma função genérica também limita a complexidade do cálculo que pode ser feito para a relocação. Para superar essas limitações, o que geralmente se faz é criar código adicional (dependente de arquitetura) para tratar casos especiais em que a biblioteca não tem suporte. Para resolver essa limitação e possibilitar o geração automática de linkeditores para um número maior de arquiteturas, foi proposta em [44] e [45] uma extensão a esse trabalho. A proposta, similar à descrita em [46] [47], é a de que em vez de se utilizar da estrutura de tabela da biblioteca BFD, uma função seja associada a cada tipo de relocação. Desta forma os modificadores se tornariam funções de relocação, as quais são pequenas funções em código C que especificam como fazer a relocação. Por exemplo, na arquitetura MIPS teríamos três tipos de relocação, conforme a Figura 4.4. Como a descrição da função é feita em código C, é possível descrever qualquer tipo de função que seja necessária, evitando restringir os tipos de arquiteturas suportadas.

Assim, o mecanismo proposto nesta dissertação abriu caminho para a generalização descrita em [44] e [45].

```
01. ac_modifier_encode(carry)
02. {
03.  reloc->field[imm] = (reloc->input + 0x00008000) >> 16;
04. }
05.
06. ac_modifier_encode(align)
07. {
08.   reloc->field[addr] = reloc->input >> 2;
09. }
10.
11. ac_modifier_encode(pcrel)
12. {
13.   reloc->field[imm] = (reloc->input - (reloc->address + 4)) >> 2;
14. }
```

Figura 4.4: Descrição das relocações na extensão do trabalho

Essa generalização será ilustrada através de exemplos. Na Figura 4.4, são mostradas as funções de três tipos de relocações da arquitetura MIPS (`carry`, `align` e `pcrel`). A construção `reloc->field[field]`, contém o campo da instrução que receberá o valor

calculado; a construção `reloc->input`, o valor ainda não relocado (endereço do rótulo); e a construção `reloc->address`, o endereço da instrução sendo relocada.

As Figuras 4.5 e 4.6 ilustram dois cenários, antes e depois da extensão mencionda, para as instruções `lui` e `j`.

```
01. lui.set_asm("lui %reg, %exp", rt, imm);
02. lui.set_asm("lui %reg, \%hi(%immHc)", rt, imm);
03. lui.set_asm("lui %reg, \%hi(%expHc)", rt, imm);
04. lui.set_decoder(op=0x0F, rs=0x00);
05.
06. j.set_asm("j %expA", addr);
07. j.set_decoder(op=0x02);
```

Figura 4.5: Descrição das relocações das instruções em seu formato original

```
01. lui.set_asm("lui %reg, %exp", rt, imm);
02. lui.set_asm("lui %reg, \%hi(%imm(carry))", rt, imm);
03. lui.set_asm("lui %reg, \%hi(%exp(carry))", rt, imm);
04. lui.set_decoder(op=0x0F, rs=0x00);
05.
06. j.set_asm("j %exp(align)", addr);
07. j.set_decoder(op=0x02);
```

Figura 4.6: Descrição das relocações das instruções em seu novo formato

Capítulo 5

Validação do gerador de linkeditores

Este capítulo descreve a validação de um dos utilitários binários desenvolvidos, o da ferramenta geradora de linkeditores, denominada `aclink`. Embora a ferramenta `acasm` seja utilizada para suportar tal validação, sua validação - bem como a de outros utilitários binários - foi feita no âmbito de dissertações de mestrado correlatas [13] [14].

5.1 Configuração experimental

O pacote *GNU Binutils* [30] disponibiliza montadores e linkeditores para diversas CPUs-alvo. Tais montadores e linkeditores, doravante denominados *originais*, serão utilizados para produzir código executável a ser usado como referência durante a validação. Os montadores e linkeditores gerados pelas ferramentas `acasm` e `aclink`, respectivamente, serão doravante denominados *gerados*. Para a validação foi utilizado o *benchmark* Mibench [48], que é representativo de aplicações típicas do mercado de sistemas embarcados.

Para fins de validação, adotou-se as CPUs MIPS, SPARC e POWERPC, uma vez que os respectivos modelos são os mais robustos disponíveis em [49]. O procedimento deverá ser repetido para mais CPUs no futuro (veja Seção 7.4).

5.1.1 Caracterização do aparato experimental

Esta seção mostra a correlação entre a complexidade dos modelos de CPUs e os arquivos gerados pela ferramentas `acasm` e `aclink`.

A Tabela 5.1 mostra os principais arquivos do pacote *GNU Binutils* modificados pela geração automática de dois utilitários binários, o montador e o linkeditor, e as bibliotecas

por eles utilizadas, `Opcodes` e `BFD`. O tamanho de cada um dos arquivos é mostrado para cada CPU-alvo, expresso em número de linhas. Nas tabelas, `[arch]` designa uma das CPUs-alvo.

Tabela 5.1: Tamanho dos arquivos gerados pelas ferramentas `acasm` e `aclink` (número de linhas)

Arquivo		MIPS	SPARC	POWERPC
Montador	`tc-[arch].h`	45	45	45
	`tc-[arch].c`	1733	1948	4611
Linkeditor	`[arch]elf.sh`	6	6	6
Opcodes	`[arch]-opc.c`	221	444	303
	`[arch].h`	27	27	27
BFD	`cpu-[arch].c`	19	19	19
	`elf32-[arch].c`	240	256	309
	`[arch].h`	28	29	33

Note que os arquivos `tc-[arch].h`, `opcodes/[arch].h`, `[arch]elf.sh` e `bfd/cpu-[arch].c` possuem o mesmo número de linhas geradas para as diversas arquiteturas, pois se diferenciam apenas nos valores dos parâmetros contidos pelos arquivos.

Apesar de o arquivo referente ao linkeditor `[arch]elf.sh` possuir apenas 6 linhas, deve-se relembrar que o mecanismo de resolução das relocações pelo linkeditor está embutido na biblioteca BFD.

A Tabela 5.2 mostra, para cada modelo utilizado, o número de formatos, o número de instruções na ISA, o número de instruções descritas em linguagem de montagem[1], o número de pseudo-instruções e o número de tipos de relocações gerados automaticamente para o modelo.

Como pode ser visto na Figura 5.1, o tamanho do arquivo `tc-[arch].c` cresce na mesma proporção do número de formatos da arquitetura sob descrição, pois contém as informações de como codificar as instruções.

Conforme mostra a Figura 5.2, o arquivo `[arch]-opc.c` cresce na mesma proporção do número de instruções *assembly* descritas no modelo, pois esse arquivo contém a tabela com as instruções da arquitetura.

[1] A mesma instrução pode ter mais de uma forma em linguagem de montagem.

Tabela 5.2: Complexidade dos modelos utilizados

CPU	Nº formatos (ISA)	Nº instruções (ISA)	Nº instruções (*assembly*)	Nº pseudo-instruções	Nº entradas na tabela de relocações
MIPS	3	59	101	14	10
SPARC	6	119	250	20	9
POWERPC	49	181	213	13	14

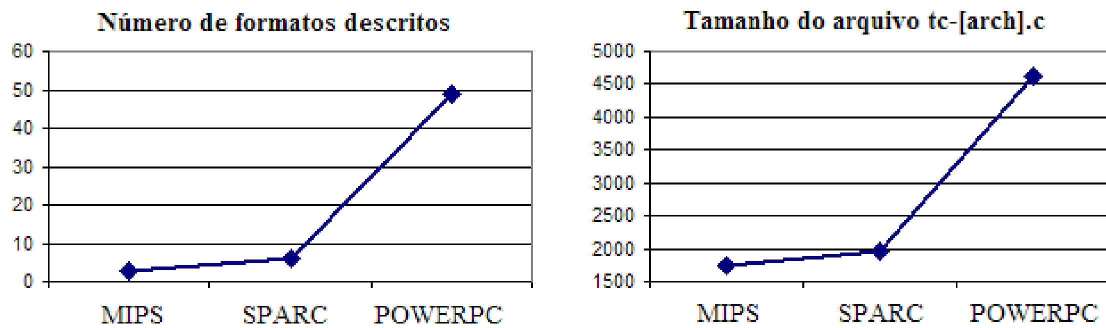

Figura 5.1: Comparação entre o número de formatos descritos em cada modelo com o tamanho do arquivo `tc-[arch].c`

Como esperado, os arquivos `elf32-[arch].c` e `bfd/archc.h` crescem com o número de relocações da arquitetura sob descrição, pois esses arquivos contêm a tabela com os tipos de relocações.

5.1.2 Validação da correção do linkeditor gerado

Dada uma CPU-alvo, o montador e o linkeditor originais são utilizados para gerar um código executável de referência, a ser comparado com o código executável produzido pelo montador e linkeditor gerados automaticamente. Como o montador gerado já foi validado anteriormente [13], a igualdade dos códigos executáveis é uma evidência experimental da correção do linkeditor gerado. Para verificar a redirecionalidade da ferramenta geradora, esse procedimento deve ser repetido para várias CPUs.

5.1.3 Fluxo de validação

A Figura 5.3 ilustra o fluxo de validação da ferramenta `aclink`. Dado um programa composto de n arquivos-fonte, o compilador cruzado GNU `gcc` produz n arquivos com

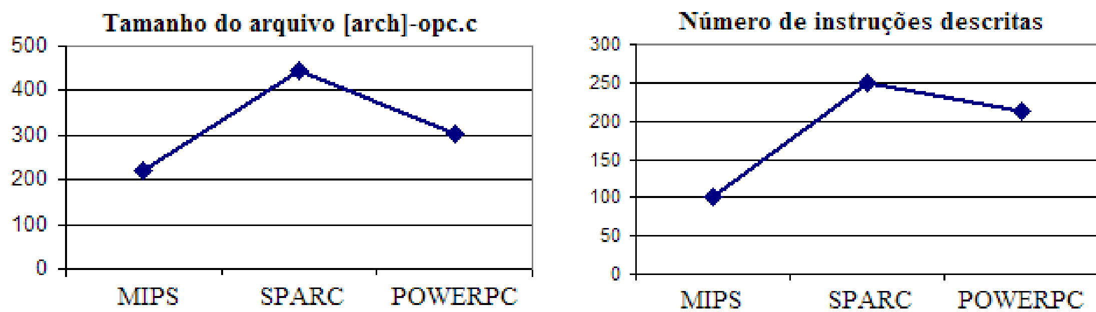

Figura 5.2: Comparação entre o número de instruções em linguagem de montagem descritas com o tamanho do arquivo `[arch]-opc.c`

código em linguagem de máquina. Para compatibilidade com o montador, esses n arquivos são então passados por um filtro, similar ao descrito em [13], que elimina diretivas de montagem dependentes de arquitetura, ou as substitui por diretivas genéricas suportadas pelo `acasm`. Em seguida, o fluxo bifurca em dois ramos distintos, um que utiliza as ferramentas originais, outro que utiliza as ferramentas geradas. Em cada ramo, os arquivos *assembly* são montados, gerando arquivos relocáveis e, depois, linkeditados, dando assim origem a dois arquivos executáveis, um que servirá de referência e outro sob validação. Ao final, verifica-se a igualdade dos arquivos executáveis produzidos em ambos os ramos do fluxo. A comparação dos arquivos executáveis foi implementada da seguinte forma. Para cada arquivo executável, o conteúdo de cada seção (`.text, .data e .bss`) foi extraído para um arquivo texto (em formato hexadecimal) através do utilitário `readelf` (que é independente de arquitetura). Em seguida, cada arquivo texto correspondente a uma seção foi comparado com seu congênere, através do utilitário `diff`.

Como os módulos podem ser linkeditados em diferentes ordens, diferentes códigos executáveis equivalentes podem ser gerados para um mesmo código-fonte. Ora, para garantir que a comparação seja feita de forma simples, é preciso garantir que ambos os ramos do fluxo de validação usem a mesma ordem, de forma que o teste de equivalência possa ser implementado através de uma mera comparação de igualdade. Para isso, em ambos os ramos definiram-se endereços comuns de início de cada seção (`.text, .data e .bss`), para que os símbolos fiquem na mesma posição em ambos os ramos. Devido à incompatibilidade do linkeditor gerado com as bibliotecas que foram utilizadas pelo linkeditor original (por exemplo, a biblioteca `newlib`), e para evitar a necessidade de nova compilação e linkedição dessas bibliotecas com o montador e linkeditor gerados, os símbolos (contidos no *assembly*) que correspondem a essas bibliotecas externas foram adicionados em um arquivo separado, podendo desta forma serem re-

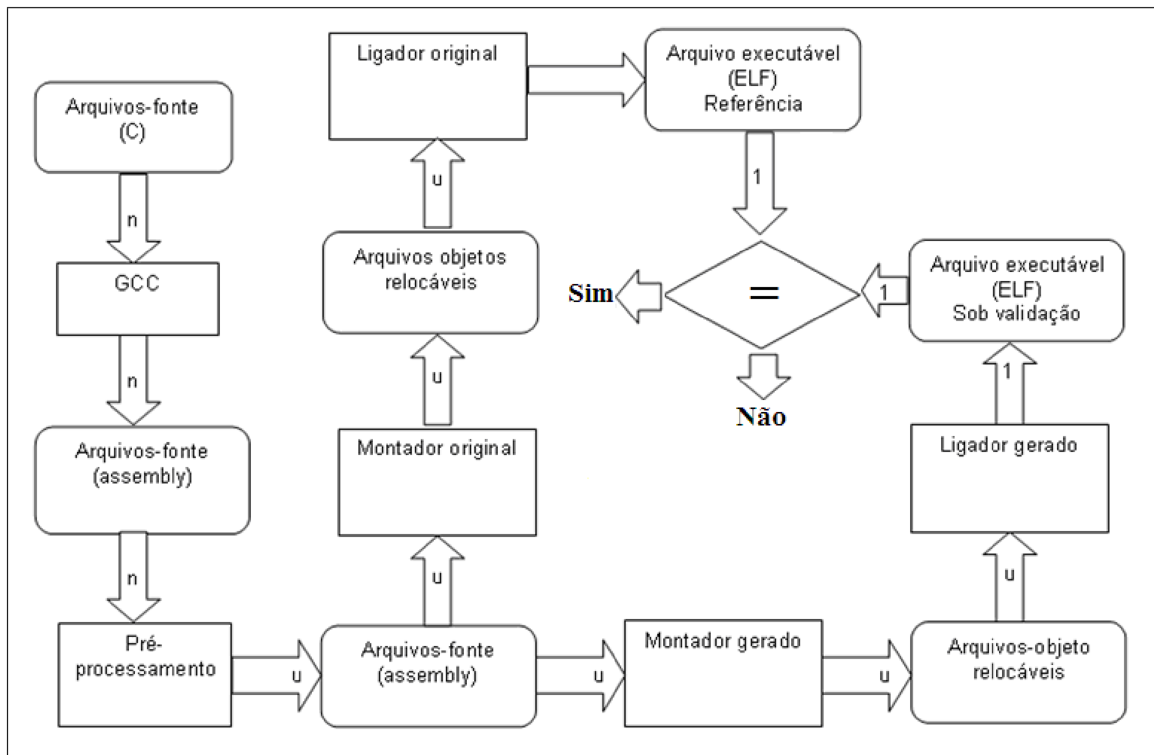

Figura 5.3: Fluxo de validação da ferramenta `aclink`

solvidos pelo linkeditor. Um trecho deste arquivo pode ser visto na Figura 5.4. Como o objetivo é permitir uma mera comparação de arquivos gerados, este procedimento não altera a validade do experimento.

```
01.   .globl memcpy
02.   memcpy: nop
03.
04.   .globl puts
05.   puts: nop
06.
07.   .globl calloc
08.   calloc: nop
```

Figura 5.4: Trecho do arquivo que define os símbolos das bibliotecas externas

5.2 Análise dos resultados experimentais

O fluxo de validação ilustrado na Figura 5.3 foi repetido para cada um dos programas do *benchmark* adotado e para cada uma das CPUs-alvo escolhidas, dando origem aos re-

sultados mostrados nas Tabelas 5.3, 5.4 e 5.5. Nas tabelas, a primeira coluna identifica o programa do *benchmark* e a segunda mostra o número de arquivos-objeto a serem linkeditados. A terceira coluna apresenta o número de relocações que resultaram necessárias durante a linkedição. As três últimas colunas apresentam o tamanho de cada uma das seções do código executável, expresso em *bytes*.

Para todos os programas das Tabelas 5.3, 5.4 e 5.5, o código executável produzido pelas ferramentas geradas coincidiu com o código de referência.

Observe que o número de arquivos linkeditados varia entre 1 e 60, resultando em arquivos executáveis que chegam à ordem de 400KB. Note também que há casos em que mais de 8000 relocações foram necessárias. O fato de se ter obtido resultados corretos para tamanha diversidade de programas reais, com uma grande variação na faixa de relocações, número de arquivos e tamanho de código é uma evidência experimental da robustez dos linkeditores gerados pela ferramenta `aclink`.

Tabela 5.3: Caracterização dos programas do *benchmark* MiBench para o MIPS

Benchmark	# arquivos	# relocações	tamanho (bytes) `.text`	`.data`	`.bss`
basicmath	4	442	6244	664	0
bitcount	9	119	4900	960	0
qsort	1	53	1804	120	0
susan	1	620	64628	1752	0
dijkstra	1	169	2472	204	40840
blowfish	7	97	19116	4416	0
sha	2	40	3232	68	0
crc32	1	22	1288	1044	0
adpcm	2	50	2500	460	2516
fft	3	212	5872	392	0
djpeg	60	3815	285500	11184	16
cjpeg	60	3707	284776	11056	16
typeset	1	8426	29668	426884	7685
ispell	1	45	1140	376	0
stringsearch	4	2956	5604	15384	2072

Tabela 5.4: Caracterização dos programas do *benchmark* MiBench para o SPARC

Benchmark	# arquivos	# relocações	tamanho (bytes) `.text`	`.data`	`.bss`
basicmath	4	353	6324	704	0
bitcount	9	106	4092	1000	0
qsort	1	44	2064	136	0
susan	1	837	59444	1576	0
dijkstra	1	161	2444	232	40840
blowfish	7	76	16484	4352	0
sha	2	25	2844	72	0
crc32	1	17	1256	1048	0
adpcm	2	39	2388	472	2516
fft	3	189	5552	440	0
djpeg	60	2135	240684	11504	16
cjpeg	60	2022	239872	11360	16
typeset	1	8217	32652	428448	7688
ispell	1	39	1172	384	0
stringsearch	4	2920	5596	15792	4140

Tabela 5.5: Caracterização dos programas do *benchmark* MiBench para o POWERPC

Benchmark	# arquivos	# relocações	tamanho (bytes) `.text`	`.data`	`.bss`
basicmath	4	199	5936	248	0
bitcount	9	100	4164	936	0
qsort	1	40	1704	108	0
susan	1	421	52688	1648	0
dijkstra	1	162	2192	204	40840
blowfish	7	82	15744	4412	0
sha	2	27	2752	68	0
crc32	1	19	1276	1044	0
adpcm	2	41	2172	460	2516
fft	3	163	5304	356	0
djpeg	60	1843	228976	11096	16
cjpeg	60	1727	228276	10968	16
typeset	1	8227	25360	426876	7684
ispell	1	41	1064	376	0
stringsearch	4	2922	4912	15384	2048

Capítulo 6

O papel dos utilitários na tradução binária

Este capítulo propõe uma técnica para tradução binária, a qual é baseada na geração automática de ferramentas a partir de modelos de CPUs descritos através de uma ADL. A motivação, a proposta e o estudo experimental de viabilidade dessa técnica foram realizados cooperativamente pelo autor e dois outros mestrandos cujas dissertações abordam tópicos também relevantes para a tradução binária [14][50]. Por essa razão, o texto das Seções 6.1, 6.2 e 6.4 é deliberadamente comum às três dissertações de mestrado. Entretanto, como a implementação do protótipo contou com contribuições distintas e complementares, a Seção 6.3 descreve a contribuição específica do autor para esse trabalho conjunto.

6.1 Motivação

Como discutido no Capítulo 1, durante a exploração do espaço de projeto de um SoC, pode-se ter que avaliar o impacto de várias CPUs alternativas até que os requisitos sejam satisfeitos. Isso requer a rápida geração de código executável para cada uma das CPUs exploradas, os quais podem ser obtidos de três maneiras distintas:

- Alternativa 1 - Disponibilidade de compiladores convencionais portados para cada uma das CPUs candidatas;
- Alternativa 2 - Disponibilidade de um compilador redirecionável;
- Alternativa 3 - Disponibilidade de um compilador convencional para uma das CPUs e de um tradutor binário capaz de gerar código executável para as demais CPUs.

A Alternativa 1 restringe o espaço de soluções exploradas ao uso de CPUs tradicionais, cujo uso intensivo justificou o desenvolvimento de compiladores próprios.

A Alternativa 2 é a mais genérica, mas requer o uso de compiladores redirecionáveis, os quais são invariavelmente proprietários (como é o caso do compilador LisaTek [51]) e cujas licenças são caras já que sua oferta no mercado é bastante pequena.

A Alternativa 3 tem a vantagem de requerer apenas um compilador convencional portado para uma única arquitetura, cuja licença pode ser pública (como a do gcc), desde que se disponha de um tradutor binário para redirecionar o código para as demais arquiteturas a serem exploradas.

Em princípio, um tradutor binário poderia ser obtido através do encadeamento de geradores de utilitários binários (Seção 6.2). Ora, a ADL ArchC provê geradores de utilitários binários sob licença GPL. Assim, o desenvolvimento de um tal tradutor resultaria numa solução de baixo custo para a exploração de CPUs. É de se esperar que o esforço de desenvolvimento de um tradutor binário estático - restrito às necessidades de sistemas embarcados - seja inferior ao requerido para se desenvolver um compilador redirecionável.

A principal dificuldade é a de garantir que a qualidade do código traduzido não seja muito inferior à obtida através de um compilador. Supondo que o compilador tenha realizado otimizações independentes de arquitetura antes de gerar o código executável sob tradução, é de se esperar que o código traduzido tenha qualidade similar, desde que o tradutor suporte otimizações dependentes de arquitetura utilizados em compiladores-otimizadores contemporâneos, tais como seleção de instruções, escalonamento de código e alocação de registradores.

A partir da revisão realizada na literatura, não é de conhecimento do autor que exista algum trabalho de pesquisa similar utilizando geração automática de ferramentas a partir de ADL para fins de tradução binária no contexto de exploração do espaço de projeto. Este fato, aliado à infra-estrutura disponibilizada pelo pacote ArchC, motivou a investigação da viabilidade dessa alternativa, como reportado nas próximas seções.

6.2 Proposta de estrutura de um tradutor binário

O tradutor binário proposto neste capítulo realiza a tradução estaticamente. Uma possível estrutura de um tradutor binário estático é ilustrada na Figura 6.1.

O código executável a ser traduzido deve inicialmente ser transformado em código *assembly* através de um desmontador. Em seguida, um *parser* do código *assembly* utiliza

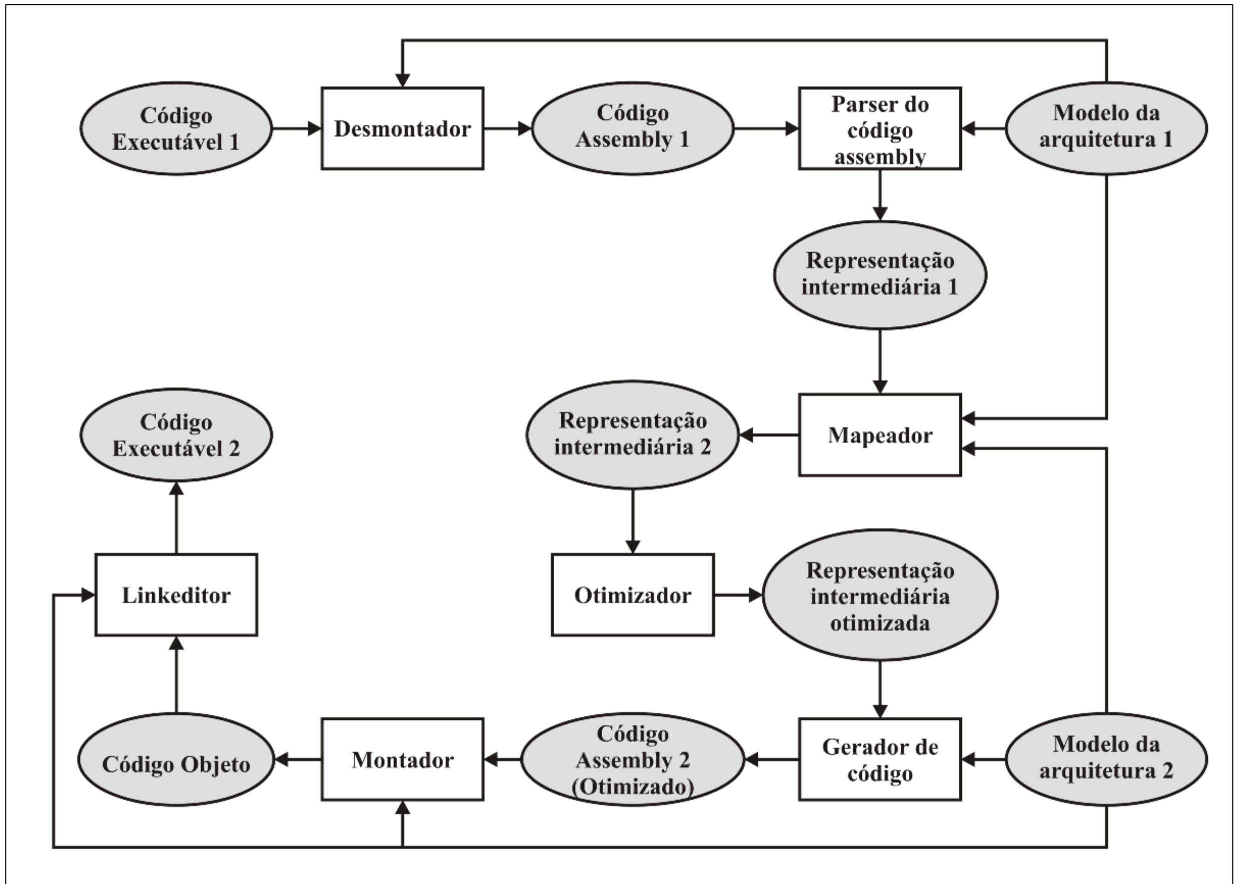

Figura 6.1: Fluxo de tradução binária

informações específicas da arquitetura para que seja gerada uma representação intermediária do código. Em geral este tipo de representação intermediária deve ser uma representação em mais alto nível das instruções do código *assembly* e deve ser independente da arquitetura alvo.

A partir desta representação e de informações como sintaxe e semântica das instruções da arquitetura para a qual o código foi originalmente gerado, o mapeador deve operar fazendo com que ela seja traduzida em outra similar buscando instruções da arquitetura alvo que tenham semântica equivalente. É gerada então uma nova representação que corresponde a instruções da arquitetura-alvo que sejam equivalentes às instruções da arquitetura original.

A representação resultante pode ser o ponto de entrada para um otimizador que opera antes da geração do código. A partir desta representação traduzida e otimizada, usando como insumos as informações específicas da arquitetura-alvo, a ferramenta deve gerar o código *assembly* equivalente ao original, o qual pode ser então montado e linkeditado para dar origem ao código executável traduzido.

As ferramentas usadas nos passos intermediários (desmontador, montador e linkeditor) são geradas automaticamente a partir dos modelos das arquiteturas fonte e alvo descritos

através de uma ADL.

6.3 Formulação do problema de mapeamento de instruções

Todos os módulos que compõe o tradutor binário (representados por retângulos na Figura 6.1) foram desenvolvidos como resultado de trabalhos correlatos [50] [14] [13] [42], exceto o módulo mapeador.

Por isso, esta seção propõe uma técnica capaz de transformar a representação intermediária do código associada à arquitetura-fonte em uma representação intermediária associada à arquitetura-alvo.

Para harmonização com o otimizador, adotou-se uma representação intermediária compatível com a definida em [50]. Embora aquela representação seja capaz de modelar também propriedades de temporização (tais como latência, atrasos e *deadlines*), apenas a relação de precedência entre instruções por ela modelada é relevante para fins de mapeamento. Entretanto, a representação original em [50], embora concebida para fins de escalonamento e análise de restrições temporais, não é independente de arquitetura. A Seção 6.3.1 propõe uma generalização da representação original para torná-la independente de arquitetura, mantendo a compatibilidade com o otimizador. A chave para a generalização é a decomposição de instruções em termos de operações primitivas que são suportadas na vasta maioria das máquinas contemporâneas.

Para viabilizar essa decomposição, que é explicada na Seção 6.3.3, a semântica de cada instrução em termos de operações primitivas deve ser descrita no modelo de cada arquitetura. A Seção 6.3.2 propõe uma solução para se estender o modelo de uma CPU com essa informação semântica.

Ao final, a Seção 6.3.3 propõe um algoritmo de seleção de instruções para a arquitetura-alvo a partir da representação independente de arquitetura e ilustra seu funcionamento através de um exemplo.

A definição formal da representação intermediária (RI) original pode ser encontrada em [50]. Trata-se de um *grafo de precedência* que captura as dependências de dados do código, conforme será ilustrado através do seguinte exemplo. Seja o seguinte código gerado para a arquitetura MIPS:

```
lw $20, 8($6)
addi $21, $20, 10
sw $21, 16($6)
```

O grafo de precedência correspondente é mostrado na Figura 6.2. Trata-se de um grafo orientado polar. O vértice `SOURCE` é um dos pólos e representa a disponibilidade dos valores iniciais a serem consumidos pelas instruções. A relação de precedência entre produtores e consumidores dos dados é representada através de arestas. Por exemplo, como a instrução `addi` consome o resultado da instrução `lw` existe uma aresta ligando os vértices que representam essas instruções. O vértice `SINK` é o outro pólo e representa a disponibilidade dos valores produzidos ao final da execução do código.

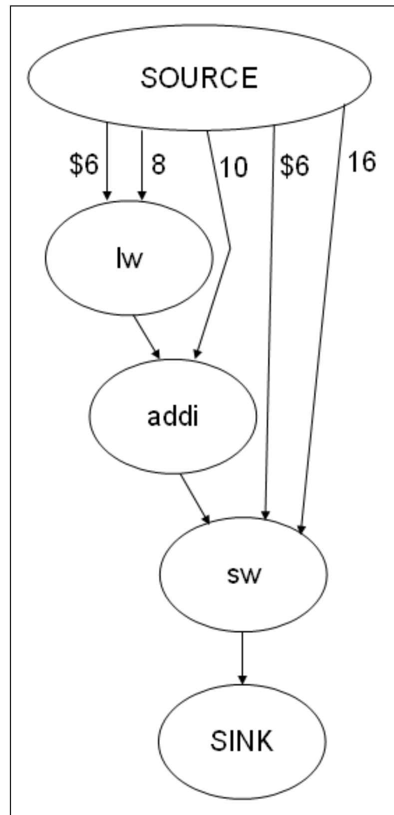

Figura 6.2: Exemplo de representação intermediária para a arquitetura MIPS

6.3.1 A representação intermediária independente de máquina

Conforme mencionado anteriormente, para possibilitar a tradução, generalizou-se a representação intermediária original decompondo a semântica de cada instrução em termos de operações primitivas, dando origem a uma representação independente de arquitetura. Desta forma, o mapeador consiste em dois passos, conforme mostrado na Figura 6.3. A RI que representa o código da arquitetura de origem é primeiramente traduzida para uma RI independente de máquina que, em seguida é traduzida para um grafo representando o código da arquitetura-alvo.

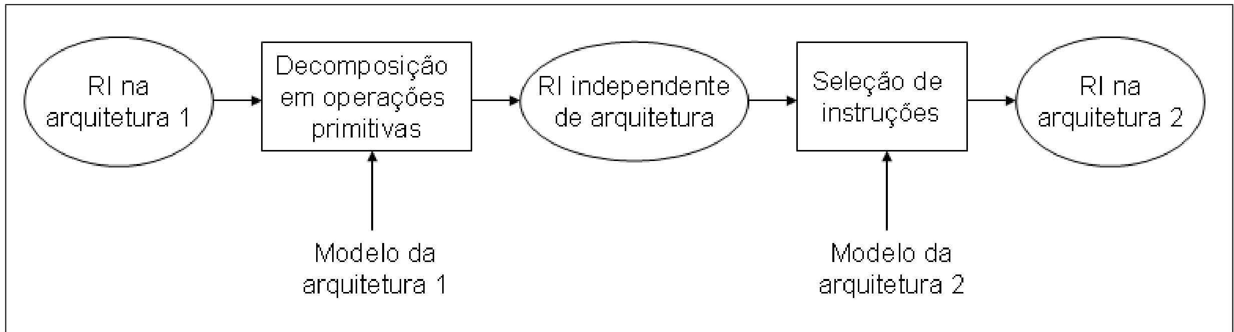

Figura 6.3: Fluxo de operação do mapeador

Para tornar a RI independente de arquitetura, é necessário definir um conjunto de operações primitivas suficientemente genéricas para representar virtualmente qualquer conjunto de instruções de interesse prático.

Para este estudo inicial de viabilidade, propõem-se as seguintes operações primitivas:

- **Operações aritméticas:** soma (ADD), subtração (SUB), multiplicação (MUL), divisão (DIV), deslocamento à esquerda (SHIFTL), deslocamento à direita (SHIFTR);
- **Operações lógicas:** E (AND), OU (OR), OU exclusivo (XOR), negação (NOT);
- **Operações relacionais:** maior que (GREATER), menor que (LESS), igual a (EQUAL).
- **Operações de acesso à memória:** leitura de palavra (LDW), de meia palavra (LDH), de meia palavra sinalizada (LDHU), de byte (LDB), de byte sinalizado (LDBU), escrita de palavra (STW), de meia palavra (STH), de byte (STB);
- **Operações de desvio:** incondicional (JUMP), condicional (BRANCH), chamada de rotina (CALL);
- **Outra:** carga de constante em registrador (LIMM).

Para distinguir operações primitivas de instruções muitas vezes homônimas, letras maiúsculas serão doravante adotadas para designar operações primitivas, enquanto letras minúsculas representarão instruções.

6.3.2 A captura de semântica das instruções nos modelos de CPU

A ADL ArchC possui uma construção denominada `set_property` que pode ser usada para representar novas propriedades para os elementos da linguagem, sejam eles instru-

ções, campos, etc. Isso permite que novas informações sejam adicionadas ao modelo sem que sua gramática seja modificada.

Com base nesse recurso da ADL, criou-se uma nova propriedade, denominada `semantic`, para permitir a extensão dos modelos de CPU, capturando a semântica das instruções em termos das operações primitivas definidas na Seção 6.3.1

O uso dessa propriedade será ilustrado através de um exemplo da arquitetura MIPS:

```
01.  lw.set_asm("lw %reg, %imm(%reg)", rt, imm, rs);
02.  lw.set_property(semantic, (temp, ADD, rs, imm) );
03.  lw.set_property(semantic, (rt, LDW, temp) );
```

A linha 2 indica que, o valor no registrador codificado no campo `rs` é somado (`ADD`) com o valor da constante codificada no campo `imm`. A linha 3, indica que o valor de `temp` é o endereço de memória do valor que será carregado (`LDW`) no registrador-alvo codificado no campo `rt`. Mais exemplos são mostrados na Figura 6.4.

```
01.  lw.set_asm("lw %reg, %imm(%reg)", rt, imm, rs);
02.  lw.set_property(semantic, (temp, ADD, rs, imm) );
03.  lw.set_property(semantic, (rt, LDW, temp) );
04.
05.  addi.set_asm("addi %reg, %reg, %exp", rt, rs, imm);
06.  addi.set_property(semantic, (rt, ADD, rs, imm) );
07.
08.  add.set_asm("add %reg, %reg, %reg", rd, rs, rt);
09.  add.set_property(semantic, (rd, ADD, rs, rt) );
10.
11.  sw.set_asm("sw %reg, %imm(%reg)", rt, imm, rs);
12.  sw.set_property(semantic, (temp, ADD, imm, rs) );
13.  sw.set_property(semantic, (rt, STW, temp) );
```

Figura 6.4: Exemplo de descrição semântica de algumas instruções da arquitetura MIPS

Feita esta extensão nos modelos de CPU, é possível gerar uma RI independente de arquitetura para cada instrução. A Figura 6.5 mostra as RIs resultantes para as instruções `lw`, `addi` e `add` descritas na Figura 6.4.

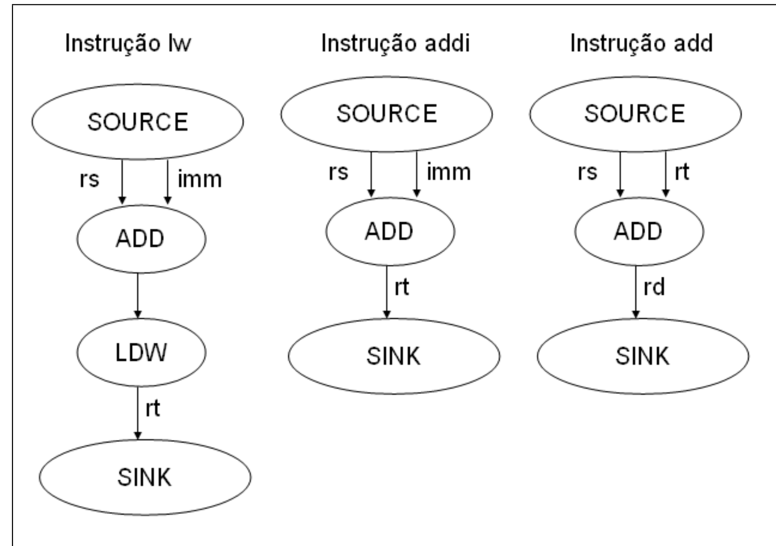

Figura 6.5: Exemplo de RIs primitivas para algumas instruções do MIPS

6.3.3 A decomposição de instruções

Dada uma RI primitiva para cada instrução, a decomposição consiste em substituir cada instrução por sua RI primitiva, dando origem a uma RI independente de arquitetura associada ao código a ser traduzido.

A Figura 6.6 ilustra a decomposição de instruções em operações primitivas para o código representado na Figura 6.2.

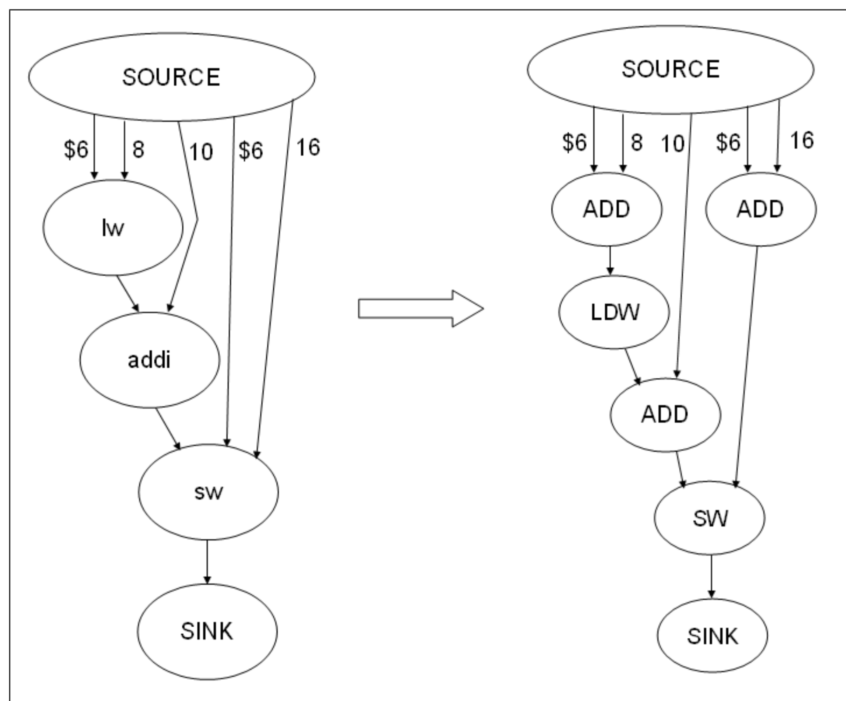

Figura 6.6: Exemplo de decomposição

6.3.4 A seleção de instruções

A partir de uma RI independente de arquitetura, a tradução consiste em encontrar um subgrafo da RI cujas operações primitivas coincidem com as de uma instrução da arquitetura-alvo, como formalizado a seguir.

Seja `primitive` uma função que retorna um grafo `P(V',E')` que serve de RI primitiva de uma dada instrução. Seja `G(V,E)` a RI independente de arquitetura associada ao código sob tradução. O Algoritmo 6.1 descreve a seleção de instruções da arquitetura-alvo. O laço delimitado pelas linhas 2 a 8 percorre todas as instruções descritas no modelo da arquitetura-alvo. A RI primitiva de cada instrução (`P`) é procurada dentro do grafo que representa o código a ser traduzido (`G`) (linha 4). Caso seja encontrado, o sub-grafo é substituído por um vértice que corresponde à instrução da arquitetura-alvo (linha 6).

Algoritmo 6.1 Algoritmo de seleção de instruções

1: let *G(V,E)* be the representation of the code to be translated;
2: **for each** *instr* **of** target-CPU **do**
3:     *P(V',E')* ← primitive(*instr*);
4:     **if** *P* is a subgraph of *G* induced by *V'* **then**
5:         let *v* be a vertex representing *instr*;
6:         replace *P* by *v* in *G* and update connectivity;
7:     **end if**
8: **end for**

Para essa substituição, a ordem de visita das instruções da arquitetura-alvo pode gerar diferentes decomposições. Por isso algum critério heurístico deve ser adotado para orientar a otimização. Por exemplo, podemos querer substituir primeiro as instruções maiores (com mais operações) ou as com menor latência, etc.

Quando aplicamos o Algoritmo 6.1 ao exemplo da Figura 6.6 tendo a arquitetura SPARC como alvo e supondo que desejamos substituir as maiores instruções primeiro (instruções cuja RI possui o maior número de vértices), obtemos o grafo da Figura 6.8, mapeado para a arquitetura-alvo. Como a arquitetura SPARC possui instruções muito similares às da arquitetura MIPS, conforme mostrado na Figura 6.7, o processo de tradução resultante é trivial. O código em linguagem de montagem resultante a partir do grafo da Figura 6.8 é o seguinte:

```
ld [%r6+%lo(8)],%r20
add %r20,%lo(10),%r21
st %r21,[%r6+%lo(16)]
```

```
01.  ld_reg.set_asm("ld [%reg + %reg], %reg", rs1, rs2, rd);
02.  ld_reg.set_property(semantic,(temp, ADD, rs1, rs2));
03.  ld_reg.set_property(semantic,(rd, LDW, temp));
04.
05.  add_imm.set_asm("add %reg, \%lo(%expL10), %reg", rs1, simm13, rd);
06.  add_imm.set_property(semantic, (rd, ADD, simm13, rs1) );
07.
08.  st_imm.set_asm("st %reg, [%reg + \%lo(%expL10)]", rd, rs1, simm13);
09.  st_imm.set_property(semantic,(temp, ADD, rs1, simm13));
10.  st_imm.set_property(semantic,(rd, STW, temp));
```

Figura 6.7: Exemplo de descrição semântica de algumas instruções da arquitetura SPARC

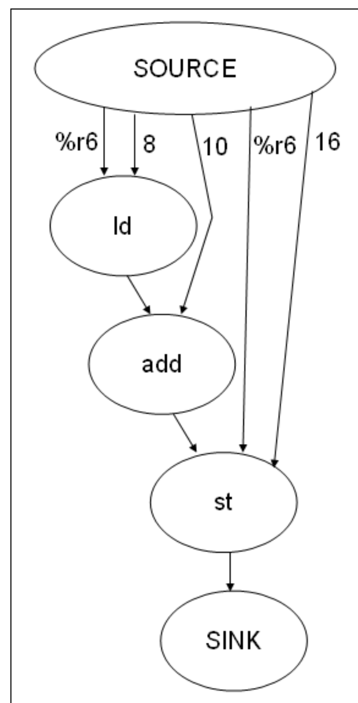

Figura 6.8: Grafo resultante da tradução

No entanto, de forma geral, não é sempre possível realizar substituições que cubram todos os vértices da RI. Para garantir que a tradução se complete, é necessário fornecer algumas diretivas de mapeamento que descrevem sugestões de como mapear operações primitivas em instruções. Isso dá origem à noção de *instrução primitiva*, que mostra como degenerar uma

instrução mais complexa da arquitetura-alvo para implementar uma operação primitiva. A Figura 6.9 mostra a descrição de algumas instruções primitivas para a arquitetura MIPS. O símbolo `$0` corresponde ao resultado da operação e os símbolos `$1`, `$2` correspondem às entradas. Por exemplo, na linha 2 temos que a operação primitiva `LDW` corresponde a um `lw` do MIPS com o valor do campo imediato nulo.

```
01.  mips1.set_property(primitive_instruction, (SUB, "sub $0, $1, $2"));
02.  mips1.set_property(primitive_instruction, (LDW, "lw $0, 0($1)"));
03.  mips1.set_property(primitive_instruction, (NOT, "nor $0, $0, $1"));
04.  mips1.set_property(primitive_instruction, (LIMM, "li $0, 1"));
05.  mips1.set_property(primitive_instruction, (STW, "sw $0, 0($1)"));
```

Figura 6.9: Descrição do mapeamento de operações primitivas

Vamos ilustrar essa situação através de um exemplo. Tomemos o seguinte código em linguagem de montagem da arquitetura SPARC:

```
01.  ld [%r1 + %r2], %r3
02.  add %r3, %r4, %r5
03.  st %r5, [%r1 + 4]
```

A RI para a arquitetura SPARC e a RI independente de arquitetura podem ser ambas vistas na Figura 6.10.

Ao tentar a substituição proposta no Algoritmo 6.1, como o MIPS não possui um `lw` que receba a soma de dois registradores, o subgrafo destacado na figura não será substituído. Porém, como existe uma instrução no MIPS que realiza a soma de dois registradores, a operação primitiva ADD será substituída pela instrução `add`. Assim o grafo resultante ficaria ainda com um vértice por ser substituído, como mostra a Figura 6.11a. Para completar a tradução, a instrução correspondente à operação LDW do vértice remanescente é substituída por uma instrução `lw` degenerada, conforme ilustra a Figura 6.11b.

Note que dessa vez o grafo resultante possui um vértice a mais que o grafo inicial, ou seja, a tradução para a arquitetura-alvo acabou gerando mais instruções que as do código inicial. O código traduzido pode ser visto a seguir:

```
add $8,$2,$1
lw $9,0($8)
add $10,$9,$4
sw $1,%lo(4)($10)
```

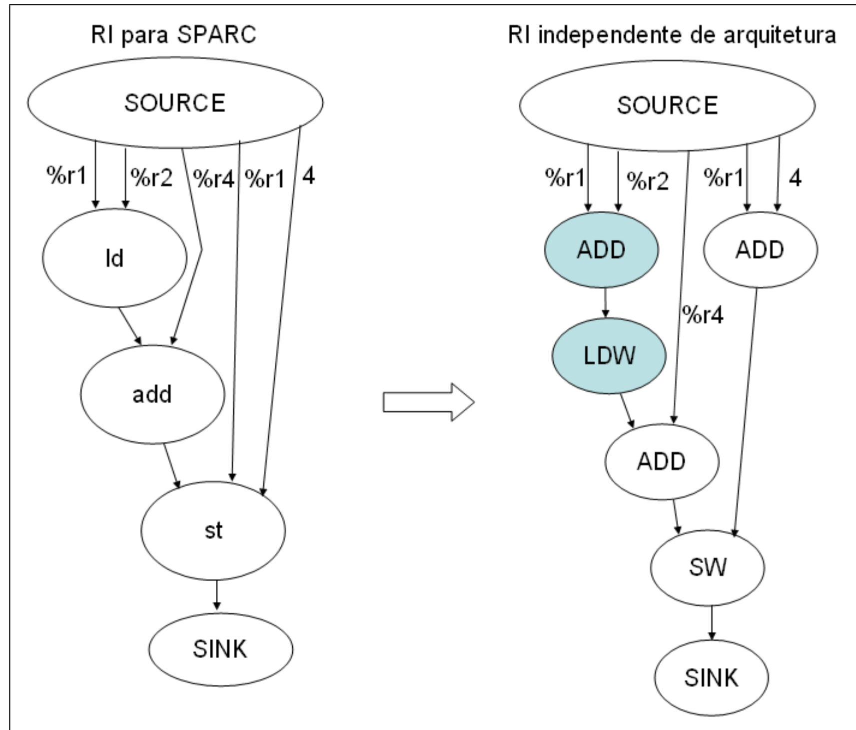

Figura 6.10: Exemplo de tradução parcial para a arquitetura MIPS

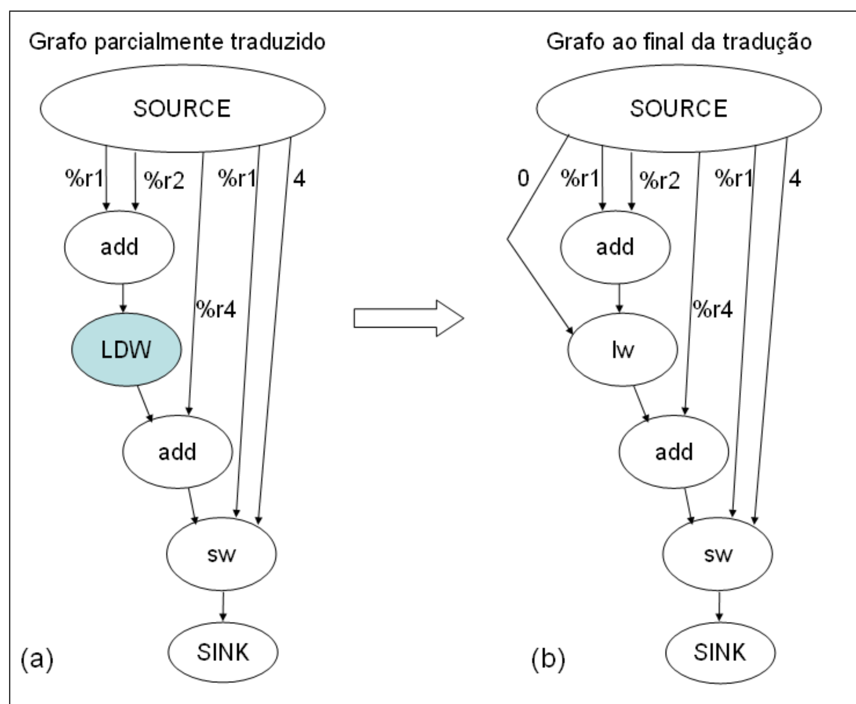

Figura 6.11: Exemplo de tradução final para a arquitetura MIPS

6.4 Resultados experimentais preliminares

Até o momento, o protótipo do tradutor binário proposto restringe-se à tradução binária de blocos básicos [52] e foi validado para as CPUs MIPS e SPARC para o conjunto de *ben-*

chmarks Dalton [53]. Contudo, o protótipo está sendo estendido para suportar código condicional.

As Tabelas 6.1 e 6.2 mostram os resultados preliminares da tradução do MIPS para o SPARC e do SPARC para o MIPS, respectivamente, listando o número de instruções na arquitetura original e o número de instruções geradas no processo de tradução. Estes números diferem nas duas tabelas em função de algumas instruções da arquitetura de origem não possuírem uma instrução equivalente única na arquitetura destino, sendo então necessária a geração de duas ou mais instruções no processo de tradução.

Tabela 6.1: Resultados da tradução binária MIPS-SPARC

Programa	Número de instruções MIPS	Número de instruções SPARC
cast	21	18
fib	67	85
gcd	31	27
int2bin	16	20
negcnt	12	14
xram	20	29

Tabela 6.2: Resultados da tradução binária SPARC-MIPS

Programa	Número de instruções SPARC	Número de instruções MIPS
cast	25	28
fib	68	78
gcd	29	32
int2bin	15	19
negcnt	08	12
xram	33	40

Note que o número de instruções obtidas via tradução para o MIPS na Tabela 6.2 (terceira coluna) é maior ou igual ao número de instruções geradas pelo compilador para o MIPS na Tabela 6.1 (segunda coluna). Note, entretanto, que o número de instruções obtidas via tradução para o SPARC na Tabela 6.1 (terceira coluna) é menor do que o número de instruções geradas pelo compilador para o SPARC em três casos (`cast`, `gcd` e `xram`), conforme a Tabela 6.2 (segunda coluna). Esta aparente anomalia indica que o compilador MIPS foi mais eficaz na

seleção de instruções do que o compilador SPARC. O tradutor binário simplesmente refletiu a melhor qualidade do código de entrada.

A validação mais extensiva da técnica de tradução envolvendo código condicional, outras CPUs e um número maior de programas do *benchmark* será objeto de trabalho futuro (veja Seção 7.4).

Capítulo 7

Conclusão

7.1 Apreciação do trabalho

Esta dissertação apresentou uma técnica para a geração automática de linkeditores a partir da descrição da CPU-alvo usando uma ADL. A implementação da técnica baseia-se na ADL ArchC e no bem conhecido pacote *GNU Binutils*, onde as bibliotecas independentes da CPU são mantidas e as dela dependentes são geradas automaticamente pela ferramenta `aclink`. Para tanto, foram realizadas extensões na ADL e modificações no `acasm`, o gerador de montadores original do pacote ArchC. A corretude e a robustez da técnica proposta foram comprovadas ao se comparar os arquivos linkeditados pela ferramenta gerada com os da ferramenta original para as arquiteturas MIPS, SPARC e POWERPC, utilizando o *benchmark* MiBench [48].

Como em [29], a implementação do gerador de linkeditores se baseia no pacote *GNU Binutils*. Porém, ao contrário de [29], o uso de modificadores faz com que o usuário não precise se preocupar com detalhes de baixo nível (descrição da tabela de relocações). Além disso, este trabalho abriu caminho para que, com as extensões descritas na Seção 4.2.6 fosse possível gerar linkeditores para arquiteturas que não possuíam ainda porte no pacote *GNU Binutils*, (como o i8051).

A concepção e a implementação do gerador automático de linkeditores, combinadas com as de outros geradores de utilitários binários - montador [13] e desmontador [14] - e de um otimizador de código em nível de linguagem de montagem [50] viabilizaram uma investigação preliminar sobre tradução binária.

7.2 Contribuições técnico-científicas e produtos de trabalho

As principais contribuições técnico-científicas desta dissertação são:

- A idéia de se usar a noção de modificadores (Seção 3.2.1.1) para descrever relocações em um nível mais alto de abstração e, portanto mais amigável para a concepção de novos modelos de CPUs.
- O mapeamento automático de modificadores para entradas da tabela de relocações (Algoritmos 4.1 e 4.2).

Os principais produtos deste trabalho de pesquisa e implementação são:

- Dois trabalhos publicados em anais [42] [54] sobre a ferramenta `aclink`.
- Um trabalho publicado em anais (IEEE ISVLSI 2007), descrevendo a ferramenta `acbingen` [45].
- Um artigo submetido para publicação em periódico (ACM TODAES), descrevendo a ferramenta `acbingen` [44].
- Protótipo da ferramenta `aclink`, já integrada à ferramenta `acbingen`.

7.3 Trabalhos em andamento

A técnica de geração de linkeditores desenvolvida no âmbito desta dissertação, combinada às técnicas de geração de montadores, desmontadores e depuradores desenvolvidos em outras dissertações correlatas [13] [14], deu origem ao gerador integrado de utilitários binários denominado `acbingen`. A harmonização dos componentes do gerador integrado motivou uma abordagem a partir de um modelo abstrato, independente de ADL utilizado para descrever a CPU-alvo. Os primeiros resultados do trabalho cooperativo na concepção do gerador integrado de utilitários binários foram objetos de dois artigos submetidos [45] [44].

7.4 Tópicos para investigação futura

Seria desejável que se validasse a técnica de geração de linkeditores para um conjunto mais amplo de arquiteturas. Para isso, entretanto, mais modelos de processadores preci-

sam ser desenvolvidos e validados. Exemplos de arquiteturas cujos modelos estão em avançado estágio de validação são os processadores Altera NIOS 2 e Motorola ColdFire.

O gerador de linkeditores aqui proposto pode ser estendido para suportar também ligações dinâmicas.

O trabalho preliminar aqui iniciado em tradução binária, pode ser estendido para suportar a tradução de código condicional, através da visita (em ordem topológica) dos vértices de um *grafo de fluxo de controle*, como o adotado em [50]. Como cada vértice desse grafo é um bloco básico e como já foi feita uma validação preliminar do tradutor para tais blocos (Seção 6.4), isso requeria uma simples extensão do protótipo original. A maior dificuldade seria a validação mais extensiva contemplando diversos pares de CPUs e um número maior de programas de *benchmarks*. Um outro trabalho que merece atenção é a investigação de diferentes heurísticas que determinam a ordem de visita dos vértices do grafo de precedência usado como RI independente de arquitetura. Com isso se poderia avaliar seu impacto na otimização da seleção de instruções e, consequentemente, na qualidade do código gerado.

Apêndice A

Estrutura e mecanismo de relocação na biblioteca BFD

Cada entrada na tabela que descreve os tipos de relocações de uma arquitetura possui os seguintes campos:

1. **type:** é um inteiro que corresponde ao número de identificação da relocação.
2. **rightshift:** O valor a ser deslocado à direita sobre o cálculo final da relocação.
3. **size:** o tamanho do item a ser relocado. Pode ser 8 bits, 16 bits ou 32 bits. Esses valores são representados internamente por 0, 1 e 2 respectivamente.
4. **bitsize:** o número de bits a ser relocado.
5. **pc_relative:** indica se a relocação é relativa ao PC. Em caso afirmativo, o endereço da instrução que está sendo relocada será subtraído do valor da relocação.
6. **bitpos:** O bit de posição do valor da relocação no destino. Indica de quantos bits o valor relocado será deslocado para a esquerda.
7. **complain_on_overflow:** Especifica qual o tipo de verificação de *overflow*.
8. **special_function:** Indica a função que irá tratar a relocação. Na maioria dos casos usa-se uma função genérica[1] previamente definida na biblioteca, mas qualquer função pode ser usada para poder tratar casos especiais.

[1] Um exemplo de como essa função funciona será dado a seguir.

9. **name:** Especifica o nome do tipo de relocação, e serve apenas para a visualização das relocações presentes em um arquivo binário (através de ferramentas como `objdump`).

10. **partial_inplace:** Indica se o adendo[2] da relocação é guardado na própria instrução, dentro do campo que deverá ser relocado ou se deverá ser guardado junto com a própria relocação.

11. **src_mask:** Máscara que indica a parte da instrução (ou dado) a ser usada para extrair o adendo, (no caso do adendo ser guardado junto com a relocação, ou seja, `partial_inplace` como sendo falso, esse valor pode ser 0)

12. **dst_mask:** Máscara que indica qual parte da instrução (ou dado) é substituída com valor relocado.

13. **pcrel_offset:** Esta informação é dependente do formato de arquivo e não tem relevância para o escopo deste trabalho.

Para exemplificar tomamos como exemplo o tipo de relocação do processador PowerPC `R_PPC_ADDR24`, esse tipo de relocação é gerada para a instrução `ba`. O formato dessa instrução é o seguinte:

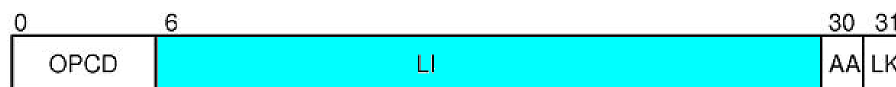

Figura A.1: Formato de instrução do tipo I da arquitetura MIPS

Neste caso, o valor relocado deve ser colocado no campo LI, de forma que seu valor esteja alinhado ao tamanho da palavra da PowerPC (32 bits). Sabendo disso pode-se inferir os seguintes valores para os dados descritos anteriormente para a entrada na tabela de relocações:

1. **type:** Um inteiro qualquer que não conflite com outro tipo de relocação.

2. **rightshift:** 2 (pois o número deve ser alinhado em 32 bits, ou seja os 2 últimos bits sempre serão eliminados).

3. **size:** 2 (pois o tamanho da instrução é de 32 bits).

4. **bitsize:** 24 (pois os bits que serão relocados pertencem ao campo li que possui 24 bits).

[2] É o endereço, dentro da seção, do símbolo a ser resolvido. Esse valor pode ser guardado junto à tabela de relocações ou no próprio campo a ser relocado. Veja `src_mask`.

5. **pc_relative:** false (não é relativo ao pc).

6. **bitpos:** 2 (note que estamos trabalhando em 32 bits (vide atributo size) e os dois últimos bits correspondentes aos campos aa e lk não fazem parte dos bits que queremos modificar, então teremos que deslocar o valor para à esquerda em 2 bits).

7. **complain_on_overflow:** Não verificar (não vamos nos preocupar com verificação de overflow no momento).

8. **special_function:** Genérica.

9. **name:** "R_PPC_ADDR24" (esse nome serve apenas para identificar o tipo de relocação em outros utilitários do pacote *GNU Binutils* como o `objdump`).

10. **partial_inplace:** false (não queremos que o addendo seja guardado junto com a instrução).

11. **src_mask:** 0x0 (como partial_inplace = false, então essa máscara deve ser 0, pois nenhum bit será selecionado para a extração do adendo).

12. **dst_mask:** 0x3fffffc (essa máscara reflete o campo LI, que é o campo em que o valor final será colocado).

13. **pcrel_offset:** true (como esse valor depende do formato de arquivo usaremos o valor verdadeiro como padrão).

A partir desses dados, a função genérica de resolução de relocação do linkeditor, pode montar a instrução já relocada. Por exemplo, vamos supor a seguinte situação: `ba label`, onde label corresponde ao endereço 0x8F4 do início da mesma seção onde se encontra a instrução e a seção se inicia no endereço 0x1000. A instrução ainda não relocada está codificada da seguinte forma:

0100 10 **00 0000 0000 0000 0000 0000 00** 11

Os bits em negrito correspondem aos bits que deverão ser modificados quando a relocação for aplicada.

1. Primeiro, vamos calcular o valor final do endereço conforme descrito na seção 3.1. Como partial_inplace é falso, não há a necessidade de extrair o addendo da instrução usando a máscara src_mask. O addendo (que no caso é 0x8F4) não está guardado na própria instrução, mas sim junto com a informação de relocação.

```
0x1000 + 0x8F4 = 0x18F4
```

que corresponde em binário:

```
0000 0000 0000 0000 0001 1000 1111 0100
```

2. Aplicamos então o rightshift, neste caso com o valor 2:

```
0000 0000 0000 0000 0000 0110 0011 1101
```

3. Agora vamos fazer o deslocamento à esquerda para posicionar o valor no local correto (bitpos) com o valor 2.

```
0000 0000 0000 0000 0001 1000 1111 0100
```

4. Basta agora aplicar a mascará de destino (dst_mask) de forma a encaixar esse valor na instrução da seguinte forma:

```
0000 0000 0000 0000 0001 1000 1111 0100
AND
0000 0011 1111 1111 1111 1111 1111 1100
=
0000 0000 0000 0000 0001 1000 1111 0100
OR
0100 1000 0000 0000 0000 0000 0000 0011
=
0100 1000 0000 0000 0001 1000 1111 0111
```

Desta forma, a instrução resulta relocada.

Referências Bibliográficas

[1] CIFUENTES, C. et al. *Preliminary Experiences with the Use of the UQBT Binary Translation Framework*. 1999. Disponível em: <citeseer.ist.psu.edu/cifuentes99preliminary.html>.

[2] SCHLIEBUSCH, O. et al. Architecture implementation using the machine description language LISA. In: *Proceedings of the 15th International Conference on VLSI Design*. [S.l.]: IEEE Computer Society, 2002. p. 239–244.

[3] BERGAMASCHI, R. A.; COHN, J. The A to Z of SoCs. In: *Proceedings of the IEEE/ACM international conference on Computer-aided design*. [S.l.]: ACM Press, 2002. p. 790–798. ISBN 0-7803-7607-2.

[4] SHAHDAD, M. An overview of vhdl language and technology. In: *DAC '86: Proceedings of the 23rd ACM/IEEE conference on Design automation*. Piscataway, NJ, USA: IEEE Press, 1986. p. 320–326. ISBN 0-8186-0702-5.

[5] THOMAS, D. E.; MOORBY, P. R. *The Verilog hardware description language (4th ed.)*. Norwell, MA, USA: Kluwer Academic Publishers, 1998. ISBN 0-7923-8166-1.

[6] HTTP://WWW.SYSTEMC.ORG. *SystemC Website*.

[7] HTTP://WWW.SYSTEMVERILOG.ORG. *System Verilog Website*.

[8] GHENASSIA, F. *Transaction-level modeling with SystemC - TLM concepts and applications for embedded systems*. [S.l.]: Springer, 2005.

[9] FAUTH, A.; Van Praet, J.; FREERICKS, M. Describing instruction set processors using nML. In: *Proceedings of the EDTC: The European Design and Test Conference*. [S.l.]: IEEE Computer Society, 1995. p. 503–507.

[10] PEES, S. et al. LISA – machine description language for cycle-accurate models of programmable DSP architectures. In: *Proceedings of the 36th ACM/IEEE Conference on Design Automation*. [S.l.]: ACM Press, 1999. p. 933–938.

[11] HADJIYIANNIS, G.; HANONO, S.; DEVADAS, S. ISDL: an instruction set description language for retargetability. In: *Proceedings of the 34th Annual Conference on Design Automation*. [S.l.]: ACM Press, 1997. p. 299–302.

[12] RIGO, S. *ArchC: Uma linguagem de descrição de arquiteturas*. Tese (Doutorado) — Instituto de Computação, UNICAMP, Campinas, Julho 2004.

[13] BALDASSIN, A. *Geração automática de montadores em ArchC*. Dissertação (Mestrado) — Instituto de Computação, UNICAMP, Campinas, Abril 2005.

[14] SCHULTZ, M. R. de O. *Geração automática de ferramentas de inspeção de código para processadores especificados em ADL*. Dissertação (Mestrado) — Departamento de Informática e Estatística, (UFSC), Florianópolis, Fevereiro 2007.

[15] LEVINE, J. R. *Linkers and Loaders*. [S.l.]: Morgan Kufmann Publishers, 2000.

[16] HTTP://WWW.GNU.ORG/SOFTWARE/BINUTILS. *The GNU Binutils Website*. 2005.

[17] SANGIOVANNI-VINCENTELLI, A.; MARTIN, G. Platform-based design and software design methodology for embedded systems. *IEEE Design and Test of Computers*, v. 18, n. 6, p. 23–33, November/December 2001.

[18] LANNEER, D. et al. CHESS: Retargetable code generation for embedded DSP processors. In: *Code generation for embedded processors*. [S.l.]: Kluwer Academic Publishers, 1995. cap. 5, p. 85–102.

[19] HARTOOG, M. R. et al. Generation of software tools from processor descriptions for hardware/software codesign. In: *Proceedings of the 34th Annual Conference on Design Automation*. [S.l.]: ACM Press, 1997. p. 303–306.

[20] RAMSEY, N.; FERNÁNDEZ, M. *New Jersey machine-code toolkit architecture specifications*. [S.l.], October 1994.

[21] RAMSEY, N.; FERNÁNDEZ, M. *New Jersey machine-code toolkit reference manual*. [S.l.], October 1994.

[22] RAMSEY, N.; FERNANDEZ, M. F. Specifying representations of machine instructions. *ACM Transactions on Programming Languages and Systems*, v. 19, n. 3, p. 492–524, May 1997. ISSN 0164-0925. ACM Portal.

[23] HTTP://EN.WIKIPEDIA.ORG/WIKI/A.OUT_(FILE_FORMAT). *a.out file format*.

[24] MOONA, R. Processor models for retargetable tools. In: *Proceedings of the 11th IEEE International Workshop on Rapid System Prototyping*. [S.l.: s.n.], 2000. p. 34–39.

[25] TAYLOR, I. L. *Executable and Linking Format ELF*. [S.l.], May 1995.

[26] HOFFMANN, A. et al. A survey on modeling issues using the machine description language LISA. In: *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*. [S.l.: s.n.], 2001.

[27] HOFFMANN, A. et al. A novel methodology for the design of application specific instruction set processors (ASIP) using a machine description language. In: *Proceedings of the IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. [S.l.: s.n.], 2001. p. 1338–1354.

[28] COFF - The Common Object File Format. In: AT&T UN/X System V Support Tools Guide. [S.l.: s.n.]. cap. 8.

[29] ABBASPOUR, M.; ZHU, J. Retargetable binary utilities. In: *Proceedings of the 39th Conference on Design Automation*. [S.l.]: ACM Press, 2002. p. 331–336.

[30] PESCH, R. H.; OSIER, J. M. *The GNU binary utilities*. [S.l.], May 1993. Version 2.15.

[31] HENNING, J. L. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer*, v. 33, n. 7, p. 28–35, July 2000.

[32] BASHFORD, S. et al. *The MIMOLA Language - Version*. 1994. Disponível em: <citeseer.ist.psu.edu/bashford94mimola.html>.

[33] HALAMBI, A.; GRUN, P. *EXPRESSION: A language for architecture exploration through compiler/simulator retargetability*. 1999. Disponível em: <citeseer.ist.psu.edu/halambi99expression.html>.

[34] BISWAS, P. *EXPRESSION User Manual (version 1.0)*. 2003.

[35] CENG, J. et al. Modeling instruction semantics in adl processor descriptions for c compiler retargeting. *J. VLSI Signal Process. Syst.*, Kluwer Academic Publishers, Hingham, MA, USA, v. 43, n. 2-3, p. 235–246, 2006. ISSN 0922-5773.

[36] CIFUENTES, C.; EMMERIK, M. V. Uqbt: Adaptable binary translation at low cost. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 33, n. 3, p. 60–66, 2000. ISSN 0018-9162.

[37] RAMSEY, N.; FERNANDEZ, M. F. Specifying representations of machine instructions. *ACM Trans. Program. Lang. Syst.*, ACM Press, New York, NY, USA, v. 19, n. 3, p. 492–524, 1997. ISSN 0164-0925.

[38] CIFUENTES, C.; SENDALL, S. Specifying the semantics of machine instructions. In: *6th International Workshop on Program Comprehension - IWPC'98, Ischia, Italy, June 24-26 1998*. IEEE Computer Society, 1998. p. 126–133. Disponível em: <citeseer.ist.psu.edu/cifuentes98specifying.html>.

[39] PATTERSON, D.; HENNESSY, J. *Organizacao e projeto de computadores - A interface Hardware/Software*. Segunda edicao. [S.l.]: Livros Tecnicos e Cientificos Editora S.A., 2000.

[40] RIGO, S. et al. ArchC: A SystemC-based architecture description language. In: *Proceedings of the 16th International Symposium on Computer Architecture and High Performance Computing*. [S.l.: s.n.], 2004. p. 66–73.

[41] HTTP://GCC.GNU.ORG. *The GNU Compiler Collection Website*. 2003.

[42] CASAROTTO, D. C.; SANTOS, L. C. V. dos. Automatic link editor generation for embedded cpu cores. In: *The 4rd International IEEE-NEWCAS Conference Proceedings*. Gatineau, Canada: [s.n.], 2006.

[43] TAYLOR, I. L. *BFD Internals*. [S.l.]. Generated from Binutils source package.

[44] BALDASSIN, A. et al. Automatic retargeting of binary utilities for embedded code generation. *Submetido ao ACM Transactions on Design Automation of Electronic Systems (TODAES)*.

[45] BALDASSIN, A. et al. Automatic retargeting of binary utilities for embedded code generation. In: *IEEE Computer Society Annual Symposium on VLSI*. Porto Alegre, Brazil: [s.n.], 2007.

[46] CASAROTTO, D. C.; FILHO, J. O. C. *Trabalho de conclusão de curso: Geração Automática de Montadores a Partir de ArchC: Um Estudo de Caso com o PowerPC 405*. Dezembro 2004.

[47] TAGLIETTI, L. et al. Automatically retargetable pre-processor and assembler generation for asips. In: *The 3rd International IEEE-NEWCAS Conference Proceedings*. [S.l.: s.n.], 2005. p. 215–218.

[48] GUTHAUS, M. R. et al. MiBench: A free, commercially representative embedded benchmark suite. In: *Proceedings of the 4th Annual IEEE Workshop on Workload Characterization*. [S.l.: s.n.], 2001. p. 3–14.

[49] HTTP://WWW.ARCHC.ORG. *The ArchC Website*. 2005.

[50] FILHO, J. O. C. *Escalonamento Redirecionável de Código sob Restrições de Tempo Real*. Dissertação (Mestrado) — Departamento de Informática e Estatística, (UFSC), Florianópolis, Fevereiro 2007.

[51] COWARE. *LisaTek*. 22 jan. 2007 2006. Http://www.coware.com.

[52] AHO, A. V.; SETHI, R.; ULLMAN, J. D. *Compiladores - Princípios, Técnicas e Ferramentas*. [S.l.]: Livros Técnicos e Científicos Editora S.A., 1995.

[53] UCR. *The UCR Dalton Project*. 31 jan. 2007 2007. Http://www.cs.ucr.edu/d̃alton/.

[54] CASAROTTO, D. C.; SANTOS, L. C. V. dos. Automatic link-editor generation for embedded cpu cores. In: *Proceedings XII Workshop IBERCHIP*. San José, Costa Rica: [s.n.], 2006.