Geração Automática de Montadores em ArchC

Alexandro Baldassin

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Geração Automática de Montadores em ArchC

Alexandro Baldassin[1]

Abril de 2005

Banca Examinadora:

- Prof. Dr. Paulo Cesar Centoducatte (Orientador)
- Prof. Dr. Olinto José Varela Furtado
  Centro Tecnológico - UFSC
- Prof. Dr. Nelson Castro Machado
  Instituto de Computação - UNICAMP
- Prof. Dr. Rodolfo Jardim de Azevedo (Suplente)
  Instituto de Computação - UNICAMP

[1]**Suporte financeiro pelo CNPq, sob processo 55.2117/2002-1**

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Bibliotecário: Maria Júlia Milani Rodrigues – CRB8a / 2116

Baldassin, Alexandro José

B32g Geração automática de montadores em ArchC / Alexandro José Baldassin -- Campinas, [S.P. :s.n.], 2005.

Orientador : Paulo Cesar Centoducatte

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Arquitetura de computadores. 2. Sistemas de computação. 3. Sistemas embutidos de computação. 4. Assembler (Linguagem de programação de computador) . I. Centoducatte, Paulo Cesar. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: Automatic generation of assemblers using ArchC

Palavras-chave em inglês (Keywords): 1. Computer architecture. 2. Computer systems. 3. Embedded computer systems. 4. Assembly language (Computer program language)

Área de concentração: Arquitetura de Computadores

Titulação: Mestre em Ciência da Computação

Banca examinadora: Prof. Dr. Paulo Cesar Centoducatte (UNICAMP)
Prof. Dr. Olinto José Varella Furtado (UNE-UFSC)
Prof. Dr. Nelson Machado (UNICAMP)
Prof. Dr. Rodolfo Jardim de Azevedo (UNICAMP)

Data da defesa: 20/04/2005

Geração Automática de Montadores em ArchC

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Alexandro Baldassin e aprovada pela Banca Examinadora.

Campinas, 20 de abril de 2005.

Prof. Dr. Paulo Cesar Centoducatte
(Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 20 de abril de 2005, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Olinto José Varela Furtado
CT - UFSC

Prof. Dr. Nelson Castro Machado
IC - UNICAMP

Prof. Dr. Paulo Cesar Centoducatte
IC - UNICAMP

"First they ignore you, then they laugh at you,
then they fight you, then you win."

Mohandas Gandhi

Resumo

Projetistas de sistemas dedicados enfrentam atualmente novos desafios em todas as fases do projeto. A difusão da tecnologia conhecida como SoC (*System on a Chip*) requer novos paradigmas para a especificação, implementação e verificação do projeto. A alta complexidade de tais sistemas e a grande variedade de configurações disponíveis podem tornar a escolha do sistema ideal demorada, prolongando o tempo de projeto e consequentemente seu ingresso no mercado. Em especial, no processo de escolha de um certo processador, o projetista necessita de um conjunto básico de ferramentas que lhe permitam analisar questões como desempenho, potência consumida, ou ainda área de sílicio ocupada. Exemplos de ferramentas importantes nessa fase de avaliação do projeto incluem compiladores, montadores e simuladores de instruções.

Nesse contexto, o uso de uma linguagem para descrição de arquitetura (*Architecture Description Language*, ADL) permite que processadores sejam modelados em níveis altos de abstração, e que um conjunto de ferramentas específicas para o modelo descrito seja gerado automaticamente. ArchC é uma ADL em desenvolvimento no Laboratório de Sistemas de Computação (IC-UNICAMP), e já é capaz de gerar ferramentas de simulação de instruções automaticamente.

Desenvolvemos neste trabalho uma ferramenta para geração automática de montadores a partir de modelos descritos em ArchC, denominada `acasm` [2]. O desenvolvimento de `acasm` nos levou a incorporar novas construções a ArchC para a modelagem da linguagem de montagem e da codificação das instruções. Nossa ferramenta gera um conjunto de arquivos dependentes de arquitetura para o redirecionamento do montador *GNU Assembler* (`gas`).

Usamos `acasm` para gerar montadores a partir de modelos, em ArchC, das arquiteturas MIPS-I e SPARC-V8, e comparamos os arquivos objetos obtidos com os gerados pelo montador `gas` nativo para ambas arquiteturas. Os resultados mostraram que os arquivos gerados pelo nosso montador foram idênticos aos gerados pelo montador nativo para ambas arquiteturas.

[2] Esperamos colocar `acasm` em domínio público por volta de Março ou Abril de 2005.

Abstract

Nowadays, embedded systems designers are facing new challenges at all stages of the design process. The growing of the system-on-chip (SoC) technology is creating new paradigms in the specification, implementation and verification phases of a design. The increasing complexity and the myriad of available configurations make it hard to choose the ideal system, therefore lengthening the design time, as well as time to market. Specially, customization of the processor architecture requires a software toolkit in order to estimate factors such as performance, power dissipation and chip area. Examples of these tools may include compilers, assemblers and instruction level simulators.

In this context, the use of an architecture description language (ADL) allows one to model processors using different levels of abstraction. Based on the model, a software toolkit can be automatically generated. ArchC is an ADL being developed by the Computer Systems Laboratory (IC-UNICAMP) and can automatically generate instruction level simulators at its current stage.

In this work, we have created a tool to automatically generate assemblers from ArchC models, named `acasm` [3]. While developing `acasm` we have introduced new language constructions to ArchC in order to describe the assembly language syntax and the instruction encoding scheme. Our tool retargets the GNU assembler (`gas`) to different architectures by generating a set of architecture depedent files based on ArchC models.

We used `acasm` to generate assemblers to the MIPS-I and SPARC-V8 architectures based on our ArchC models. We then compared the object files created by our assemblers with the ones created by the native gas and no difference between each pair of files was noticed, for both architectures.

[3] We hope to release `acasm` in public domain by March or April of 2005.

Agradecimentos

Agradeço enormemente aos professores Rodolfo e Guido, e em especial ao meu orientador Paulo Centoducatte, pela chance de trabalhar junto ao projeto ArchC e por todo o auxílio durante o desenvolvimento das idéias que resultaram nesta dissertação.

Tenho muito o que agradecer aos amigos de laboratório (LSC), em especial aos colegas do time ArchC, pelas discussões, dicas e apoio. Dentre eles, gostaria de deixar aqui um agradecimento explícito ao hoje Prof. Sandro Rigo, Bartho e Márcio Juliato.

Agradeço também a todos os colegas de disciplinas pela agradável companhia, em especial ao Augusto pela convivência e discussões acerca de assuntos matemáticos e sobre Teoria da Computação.

Por fim, agradeço a minha família por todo apoio durante os dois anos em que me dediquei ao trabalho de mestrado, e a Deus por me dar saúde e força de espírito.

Dedico esta dissertação a todas as pessoas que fazem de seu trabalho seu maior divertimento.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Atualmente mais de 90% dos processadores programáveis produzidos são empregados em sistemas dedicados [17]. Tais sistemas desempenham um papel computacional específico em um sistema maior, cuja principal função geralmente não está vinculada à computação. Encontramos exemplos de sua aplicação em automóveis, tomógrafos, eletrodomésticos, sistemas de segurança, máquinas fotográficas, celulares etc. O alto nível de integração alcançado atualmente tem permitido sintetizar em um único circuito integrado componentes como o núcleo do processador, memória e unidades especializadas, resultando na tecnologia conhecida como *System On a Chip* (SoC). A utilização dessa tecnologia tem proporcionado a implementação de sistemas dedicados menores com alto desempenho e baixo consumo de energia, o que tem aumentado seu uso nas mais diversas aplicações.

Para tornar possível a exploração de arquiteturas para um SoC em um espaço de tempo reduzido, utiliza-se uma especificação em alto nível (*system level design*) e estima-se sua eficiência avaliando-se fatores como desempenho, área do silício a ser ocupada e consumo. Desta forma o projetista pode, em pouco tempo, caracterizar vários modelos e escolher aquele mais apropriado à aplicação. Uma vez definidos a arquitetura do processador e seu sistema de memória, o projetista necessita de um conjunto de ferramentas de software composto de pelo menos um montador (melhor se possuir um compilador) e de um simulador específico para o sistema. No entanto, o curto tempo para se colocar o produto no mercado (*time-to-market*) inviabiliza a confecção manual de tais ferramentas de software durante o tempo de desenvolvimento do sistema. Uma solução ideal é a geração automática dessas ferramentas a partir da especificação do processador e do sistema de memória.

O uso de linguagens de descrição de hardware (*Hardware Description Language*, HDL) para a descrição de sistemas não fornece todas as informações necessárias para a geração automática de ferramentas como um simulador em nível de instruções, um montador e/ou compilador. Uma ADL (*Architecture Description Language*) é uma linguagem que

permite a especificação formal de modelos de arquiteturas, a partir da qual é possível gerar automaticamente um conjunto de ferramentas. As principais ADL's existentes atualmente contam com, no mínimo, ferramentas geradoras de simuladores e montadores.

ArchC [37] é uma ADL que está sendo desenvolvida no Laboratório de Sistemas de Computação (LSC) do IC-UNICAMP, e atualmente já permite a geração automática de simuladores. Existem modelos ArchC para arquiteturas como MIPS-I, SPARC-V8, Intel 8051 e PowerPC.

Nesse contexto, o objetivo dessa dissertação é expandir a linguagem ArchC com a inclusão de recursos que possibilitem a geração automática de montadores, e criar uma ferramenta para a geração automática de montadores baseados nos modelos de arquiteturas escritos em ArchC. Para a expansão da linguagem, estudamos as características das principais ADL's que contam com a geração de montadores, procurando absorver seus principais aspectos positivos em ArchC e criando novas soluções para questões ainda não abordadas. Nossa ferramenta de geração, denominada `acasm`, utiliza a estrutura de redirecionamento do montador da GNU, o `gas` [5]. `acasm` gera, a partir de um modelo descrito em ArchC, os arquivos dependentes de arquitetura no formato exigido pela infra-estrutura do montador `gas`. Esses arquivos podem então ser compilados e instalados utilizando o mesmo processo de compilação de qualquer montador `gas`, resultando em um montador para o modelo ArchC descrito. No atual estágio de desenvolvimento, o montador pode gerar arquivos objetos no formato ELF relocável.

Testamos nossa abordagem através da geração de montadores para as arquiteturas MIPS-I e SPARC-V8. Comparamos os arquivos objetos gerados pelos nossos montadores com os gerados pelos montadores nativos `gas` para ambas arquiteturas, utilizando arquivos do *benchmark* MiBench [11].

Nossas principais contribuições com este trabalho são:

- estudo das características necessárias a uma ADL para geração automática de montadores;
- expansão da linguagem ArchC para o suporte à geração automática de montadores;
- uma metodologia para implementação de montadores redirecionáveis a partir do pacote GNU Binutils.

Esta dissertação está organizada da seguinte forma. O capítulo 2 apresenta as características das principais ADL's no contexto de geração automática de montadores, discute algumas ferramentas redirecionáveis para montadores e justifica a escolha de nossa abordagem. O capítulo 3 apresenta ArchC e seus novos comandos, enquanto o capítulo 4 descreve a implementação da ferramenta `acasm`. A seguir, no capítulo 5, apresentamos

alguns resultados de nossa abordagem. Terminamos discutindo os resultados e apontando aspectos que ainda devem ser trabalhados no capítulo 6.

Capítulo 2

Trabalhos Relacionados

Existem basicamente duas classes de trabalhos a serem mencionadas no contexto dessa dissertação. A primeira classe abrange a expressividade de uma ADL para representar as informações necessárias à geração automática de um montador: sua sintaxe e semântica. A segunda classe envolve a questão de como, a partir de um modelo descrito em tal linguagem, um montador será gerado. Nesse tocante, são de grande importância as ferramentas geradoras de código que sejam redirecionáveis. Estamos principalmente preocupados em destacar aquelas ferramentas as quais possam ser reutilizadas no processo de geração de montadores.

Este capítulo apresenta as características das principais ADL's no contexto de montadores, e ferramentas que possam ser utilizadas no processo de geração de tais montadores. No final destacamos as vantagens e escolha da nossa abordagem sobre as demais.

2.1 nML

nML [10] [7] é uma ADL desenvolvida na *Technical University of Berlin* (TUB) a partir de 1991, voltada à descrição do conjunto de instruções de uma dada arquitetura. A linguagem é baseada em uma gramática de atributos onde cada regra de produção possui duas derivações possíveis: *regras-ou* para indicar alternativas e *regras-e* para indicar compartilhamento de propriedades. Símbolos não-terminais são usados para descrever aspectos da linguagem como modos de endereçamento, operações, classes de armazenamento e representação de dados. A partir das regras de produção são derivadas as *strings* que compõem o conjunto de instruções da arquitetura sendo modelada. Toda a semântica é descrita através de atributos associados às regras de produção. Tais atributos são representados como expressões e podem referenciar outros atributos utilizados em uma *regra-e*.

Como estamos interessados nos recursos da linguagem necessários para a geração de um montador, vamos nos concentrar basicamente em duas regras de produção descritas pelas

palavras-chaves `op` e `mode`. Ambas possuem os atributos `syntax` e `image` pré-definidos, e além disso `op` pode usar o atributo `action`. Uma descrição dessas palavras-chaves é apresentada a seguir:

`op`:
: descreve operações (basicamente instruções).

`mode`:
: permite a descrição de modos de endereçamento.

`syntax`:
: descreve a sintaxe em linguagem de montagem da instrução.

`image`:
: armazena a imagem binária de codificação da instrução.

`action`:
: descreve a semântica da operação quando ela for executada.

Um exemplo mostrando um trecho de um modelo escrito em nML pode ser encontrado na figura 2.1, e é uma versão modificada do apresentado em [16].

```
1 mode MEM(i:index) = M[R[i]]
2 syntax = format("(R%d)", i)
3 image = format("0 %4b", i)
4
5 mode REG(i:index) = R[i]
6 syntax = format("R%d", i)
7 image = format("1 %4b", i)
8
9 mode ADDR = MEM | REG
10
11 op add_op(src:ADDR, dst:ADDR)
12 action = { dst = src + dst; }
13 syntax = format("add %s,%s", src.syntax, dst.syntax}
14 image = format("000001 %b %b", dst.image, src.image)
```

Figura 2.1: Trecho de um modelo escrito em nML

As linhas 1, 5 e 9 criam modos de endereçamento usando `mode`, sendo que 1 e 5 utilizam *regras-e*, enquanto 9 utiliza uma *regra-ou*. Para a definição da sintaxe e codifição é utilizado `format`, que funciona de uma forma similiar à função `printf()` em linguagem

C. Os modos de endereçamento utilizam um tipo de dado chamado `index`, definido como um inteiro de 4 bits, e também elementos de armazenagem previamente definidos: `M` é uma memória, e `R` um banco de registradores.

Uma instrução `add_op` é criada na linha 11 através da palavra-chave `op`. Essa regra é uma *regra-e* que possui dois parâmetros, `src` e `dst`, ambos do tipo `ADDR`. Desse modo, os valores de `src` e `dst` podem ser tratados como os do tipo `MEM` ou `REG`. Além disso, os atributos definidos nos modos de endereçamento podem ser usados para especificar a sintaxe e codificação da instrução em `op`, através do uso do ponto (`'.'`) seguido do nome do atributo. Assim, `src.syntax` e `src.image` herdam, respectivamente, a sintaxe e codificação do tipo pai definidos pelos modos de endereçamento associados.

Vimos portanto que informações necessárias para geração de um montador em nML são especificadas através de duas palavras-chaves: `syntax` e `image`. A existência do comando `format` torna a formatação da sintaxe e codificação mais simples e expressiva através de especificadores de conversão como o `%s` ou `%d`, exatamente como ocorre na função `printf()` em C. A linguagem não dispõe, porém, de um mecanismo para atribuição de múltiplas sintaxes e codificações em uma mesma regra. Considere, por exemplo, que desejemos usar o nome `"R0"` ou `"$zero"` para um mesmo registrador, de valor 0; teríamos de criar duas regras `mode` para cada atribuição (*regras-e*), e uma outra regra `mode` para indicar a alternativa dos nomes (*regra-ou*). A linguagem também não apresenta recursos para descrição de instruções sintéticas[1].

Ferramentas geradas a partir de modelos em nML são em sua grande maioria simuladores e geradores de código. Dentre os geradores de código é mais comum encontrarmos discussões sobre seleção de instruções, técnicas de casamento de padrões e escalonamento de operações, como pode ser visto em [6] e [22]. Infelizmente as questões referentes especificamente à geração de montadores não são discutidas nesses artigos. Um trabalho que mostra um pouco mais de detalhes sobre o montador gerado é devido a um grupo da Cadence Design Systems [16]. O montador gerado por eles executa de forma interpretada e não suporta o uso de diretivas de montagem, rótulos etc., estando portanto bem incompleto.

2.2 Sim-nML

Sim-nML [29] é uma ADL desenvolvida no *Indian Institute of Technology* (IIT) por volta do ano de 1998. Essa linguagem usa como base a nML mas não apresenta algumas limitações presentes nessa última, como o suporte à temporização de operações e estruturas mais complexas de *pipelines*. A principal motivação para a criação da Sim-nML foi o seu

[1] Também conhecidas como pseudo-instruções. São instruções definidas em função de outras pré-existentes na arquitetura.

uso para estimar o desempenho de modelos através de simuladores gerados automaticamente. Não há nenhuma novidade na Sim-nML em relação a diretivas específicas de uso para geradores de montador. Desse modo, `syntax` e `image` continuam sendo usadas da mesma forma, assim como também permanecem seus limites de expressividade já expostos na seção 2.1.

O grupo do IIT responsável pela Sim-nML vêm apresentando muitas ferramentas baseadas nessa linguagem. Eles desenvolveram uma ferramenta chamada de *irg* [30], específica para geração de uma representação intermediária dos modelos escritos em Sim-nML. As ferramentas geradoras trabalham com esse formato intermediário padrão ao invés do modelo original, isolando o modelo das ferramentas. Uma grande vantagem dessa abordagem é que se poderia transformar, em teoria, qualquer modelo escrito em uma outra ADL nesse mesmo formato, o que tornaria possível o uso de todo o conjunto de ferramentas já disponíveis. No entanto, diferenças de expressividade e abrangências de algumas ADL's podem limitar essa transformação, ou mesmo não torná-la possível.

Uma ferramenta geradora de montadores, conhecida como *asmg*, foi descrita em detalhes na dissertação de mestrado de Sarika Kumari [20]. A figura 2.2 apresenta uma visão geral do funcionamento dessa ferramenta e os vários passos necessários à geração do montador e do arquivo objeto alvo. `asmg` utiliza o formato intermediário Sim-nML do modelo descrito, e gera um conjunto de arquivos dependentes de arquitetura que constituem os analisadores léxico e sintático do montador, além de um arquivo com palavras-chaves. Os arquivos que compõem o módulo léxico e sintático seguem os formatos utilizados pelas ferramentas GNU `flex` e `bison`[2]. Existe ainda um outro conjunto de arquivos, independentes de arquitetura, com módulos para manuseio de tabelas de símbolos, um analisador de diretivas e um gerador de arquivo objeto.

O montador gerado trabalha em dois passos: no primeiro, todo o código fonte é examinado e os símbolos que não podem ser resolvidos tem sua resolução adiada; no segundo, referências a símbolos são resolvidas e o código objeto é gerado. O arquivo objeto gerado usa o formato ELF. Para que seja possível resolver problemas de relocações, no momento da montagem do código fonte o usuário deve passar como argumento para o montador um arquivo texto com informações sobre relocações da arquitetura. O montador suporta as diretivas utilizadas pelo montador GNU `gas`, embora algumas sejam interpretadas de forma diferente. Também é possível gerar uma listagem do arquivo montado semelhante ao gerado pelo `gas`.

Podemos enumerar uma série de características que tornam o montador gerado atrativo: suporte a diretivas de montagem, uso de rótulos, geração de arquivo objeto relocável ELF e geração de listagens. No entanto, ao se passar como argumento um arquivo descrevendo relocações da arquitetura direto ao montador, exclui-se do modelo da ADL algumas

[2]Essas e outras ferramentas GNU podem ser encontradas em `http://www.gnu.org/software`

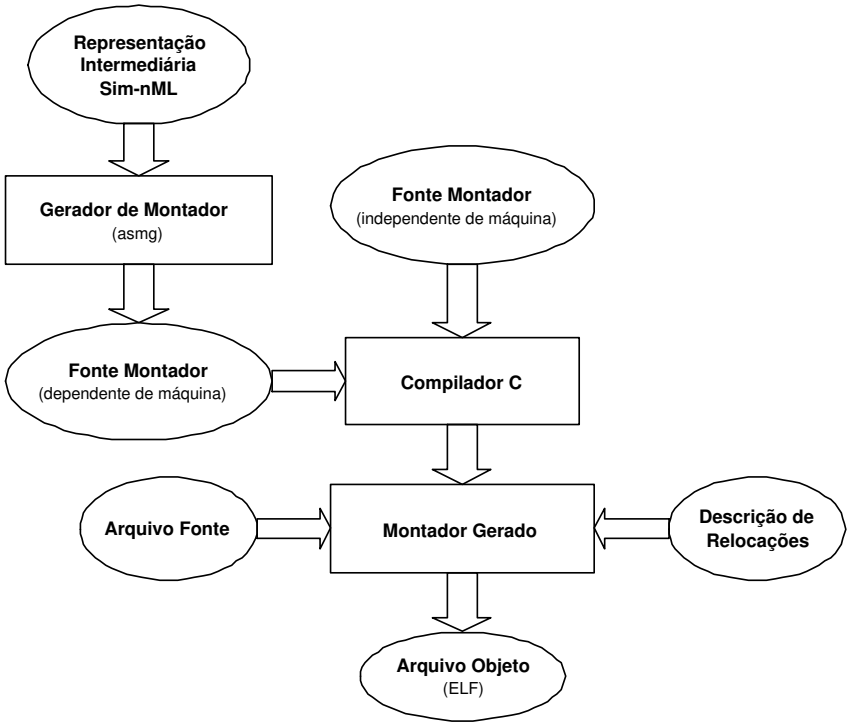

Figura 2.2: Processo de geração de um montador utilizando a ferramenta `asmg` da Sim-nML

informações. Outro fator negativo é que existem diretivas que não são totalmente compatíveis com o `gas`, o que pode causar confusão em usuários habituados ao uso desse último.

Foram gerados montadores para os modelos PowerPC 603, 68HC11, 8085 e Hitachi H/8 usando o `asmg`, e os resultados da montagem de alguns arquivos foram comparados com os do `gas` para as arquiteturas disponíveis. No entanto não há nenhum critério adotado para essa comparação descrito na tese, e não se pode realmente chegar à conclusões sólidas e criteriosas sobre a corretude do montador gerado. A solução adotada pelo IIT praticamente não faz uso de nenhuma outra ferramenta montadora que pudesse ser redirecionável, criando portanto um novo montador com novas sintaxes, diretivas, linha de comando, etc., e sujeito a todos os erros que um programador pode cometer quando da codificação de um software complexo. Apesar do esforço para torná-lo compatível com o `gas`, ficaram visíveis algumas divergências apontadas na própria tese, como aconteceu com as diretivas de montagem.

2.3 ISDL

ISDL [13] [12], de *Instruction Set Description Language*, é uma ADL desenvolvida no *Massachusetts Institute of Technoloy* (MIT) a partir de 1997. Uma das características principais da linguagem é seu recurso para descrição de arquiteturas VLIW (*Very Long Instruction Word*), dificilmente presente em outras ADL's. Também é possível o tratamento de restrições arquiteturais de forma explícita, recurso não presente na nML.

A linguagem é baseada em uma gramática de atributos onde as regras de produção definem padrões usados em operações. Um processador é visto como um conjunto de elementos de estado e de operações para transições desses estados. Uma descrição em ISDL consiste de seis seções:

formato de instrução:
: define a representação binária da instrução, seus campos e subcampos. Esta seção é equivalente à descrição de formatos de instruções em ArchC.

definições globais:
: esta seção é usada para criar definições que serão usadas no restante de uma descrição em ISDL. É composta basicamente de entidades da linguagen chamadas de *tokens* e *non-terminals*. Ambas possibilitam a fatoração tanto de sintaxe, codificação binária de instruções e até mesmo de ações semânticas das operações.

recursos de armazenamento:
: definições de elementos de armazenagem como banco de registradores e memória são declarados nessa seção.

operações:
: esta é a principal seção da linguagem, responsável pela descrição das instruções do processador. Ela pode ser dividida em subseções, cada uma com sua função e sintaxe próprias. De interesse, são definidas aqui a sintaxe exata usada na linguagem de montagem e codificação da instrução. Token e símbolos não terminais definidos previamente são usados nessa seção para a fatoração da descrição.

restrições:
: esse seção descreve as restrições das operações, como por exemplo as combinações de operações que não possam ser executadas pelo processador. Essa seção está bem ligada à característica da linguagem de suporte a arquiteturas VLIW e de informações necessárias à ferramenta geradora de compiladores.

informações adicionais:

o objetivo inicial dessa seção é puramente o de fornecer ao compilador informações para otimização de código. No entanto, a última versão do manual da linguagem não pré-define nenhum comando para essa seção e também não recomenda que seja usado por ferramentas.

Das seções vistas, a de definições globais e de operações são as que provêm maiores informações para ferramentas geradoras de montadores. A figura 2.3 mostra um trecho de um modelo escrito em ISDL que servirá de base para elucidação dessas informações.

```
1 Section Global_Definitions
2
3 Token R[0..31]    REG { [0..31]; };
4 Non_Terminal    ALUSRC:
5 A_D {$$ = 1;} | B_D {$$ = 1;} |
6 X_D {$$ = 2;} | Y_D {$$ = 3;};
7
8
9 Section Instruction_Set
10
11 Field Main:
12 add REG, ALUSRC
13 { Main.OP = 0x1|(REG<<16)|(ALUSRC<<1); }
14 { REG <- REG + ALUSRC; }
```

Figura 2.3: Trecho de um modelo escrito em ISDL

Na figura 2.3, a linha 1 indica o início da seção de definições globais. A linha 3 declara, através da palavra-chave `Token`, o símbolo `REG` para ser usado em seções futuras. `REG` representa as strings formadas por `"R"` seguidas de um número de 0 a 31, ou seja, `"R0"`, `"R1"`, ..., `"R31"`. Os valores entre chaves, declarados após `REG` na linha 3, indicam os valores de retorno desse token. A faixa de valores da direita deve ser igual a da esquerda para que a declaração seja válida. Um único token, no entanto, não consegue descrever uma faixa variada de símbolos e valores. Para isso, ISDL utiliza a diretiva `Non_Terminal`. O não-terminal `ALUSRC`, declarado na linha 4, é composto dos terminais definidos nas linhas 5 e 6. Para cada terminal (`"A_D"`, `"B_D"`, `"X_D"` e `"Y_D"`), um valor de retorno é atribuído em seguida através do símbolo `$$` e do caracter de igual, colocados entre chaves. Símbolos não-terminais ainda permitem a definição da codificação de campos e ações expressas através de uma sintaxe RTL.

A linha 9 inicia a seção de operações. Cada operação começa a ser descrita através da palavra-chave `Field` seguida de um nome. `Field` é usado para designar operações que

podem ser executadas em paralelo. A linha 11 define uma operação `Main` que é composto das linhas 12, 13 e 14. A linha 12 define a sintaxe da instrução `add`, que usa como parâmetros os símbolos definidos `REG` e `ALUSRC`. Seguindo a sintaxe, na linha 13, temos a definição da codificação da instrução. `Main.OP` é o campo da instrução (`OP` é um campo de formato definido na seção de formato de instrução) aonde o valor binário calculado em seguida deve ser armazenado. É possível usar operadores aritméticos e de deslocamento para a descrição da codificação. Seguindo, na linha 14, encontramos uma descrição em RTL da ação da operação. ISDL também permite que sejam associados a essa operação informações de custo e de temporização.

Desde sua concepção, ISDL foi projetada especialmente para geração de ferramentas redirecionáveis, como montadores, desmontadores, geradores de código e simuladores. ISDL foi a primeira a reconhecer a importância da geração de um montador em estados iniciais da linguagem, enquanto outras linguagem estavam inicialmente preocupadas com a geração de simuladores e geradores de código para *backends* de compiladores. O primeiro artigo sobre a linguagem [13] já enfocava a geração automática de um montador, embora não existam informações detalhadas sobre esse processo. Sabe-se que o gerador cria arquivos `lex` e `yacc` que quando compilados são capazes de processar arquivos escritos em linguagem de montagem, gerando um arquivo binário como saída. O montador gerado trabalha em dois passos e permite o uso de rótulos.

Trabalhos posteriores produziram outras ferramentas. Hanono et al. [15] descreveram um gerador de código baseado em descrições ISDL e Hadjiyiannis et al. [14] apresentaram uma ferramenta capaz de gerar simuladores com precisão de ciclo baseados na mesma linguagem.

ISDL possui um mecanismo bastante expressivo para definição de símbolos utilizados pelo montador, através do uso de *tokens* e de não-terminais. A definição da sintaxe das instruções pode então ser feita de maneira simples usando-se esses símbolos pré-definidos. Também existem na linguagem *tokens* pré-definidos para tipos comuns como inteiros, *floats* e nomes (referências simbólicas). Informações sobre a codificação de instruções são feitas de modo explícito através de operadores aritméticos e de deslocamento. Apesar de possuir essas características que permitem a geração de um montador eficiente, uma descrição em ISDL não é trivial devido a complexidade da sintaxe da linguagem; por exemplo, a descrição de operações pode ter até cinco subseções, cada uma com sua sintaxe própria. A definição de não-terminais podem incluir muitas informações, como código RTL expressando ações e efeitos colaterais dessas ações. Essa carga de informação pode gerar dúvidas quanto a forma de fatoração das instruções, suas codificações e descrição das ações. Erros no modelo são bem mais difíceis de serem detectados e corrigidos devido ao fato das informações estarem espalhadas por todas as seções do modelo.

2.4 LISA

LISA [44] [27] [17] é uma ADL desenvolvida na *Aachen University of Technology* (RWTH) por volta do ano de 1996. O propósito inicial da linguagem foi o de tratar, especialmente, a questão do redirecionamento de simuladores compilados. Desde então ela tem evoluido bastante, com a inclusão de novos recursos e uma plataforma de desenvolvimento constituída de um amplo conjunto de ferramentas como simuladores, montadores e linkeditores. A escassez de documentos com detalhes sobre a linguagem em domínio público limita um pouco a discussão sobre sua expressividade. No entanto, o material disponível permite abordar os aspectos referentes às informações necessárias para geração de montadores.

Uma descrição em LISA pode ser dividida basicamente em declarações de recursos e de operações. Recursos descrevem a constituição do hardware, como o banco de registradores, memória e estágios de *pipeline*. Operações descrevem, basicamente, o conjunto de instruções, sua estrutura e comportamento. A linguagem usa para declaração de operações a palavra-chave `OPERATION` seguida de um identificador e uma possível lista de opções. Os atributos de uma operação são definidos em diversas seções pré-definidas pela linguagem, entre elas:

`CODING`:
: contém a imagem binária da instrução.

`SYNTAX`:
: descreve a sintaxe em linguagem de montagem da instrução.

`SEMANTICS`:
: descreve a função de transição da instrução.

`BEHAVIOR`:
: especifica o comportamento da instrução ao ser executada.

`EXPRESSION`:
: define operandos e modos de execução usados no contexto das operações.

`ACTIVATION`:
: possibilita que outras operações sejam ativadas no contexto da operação atual.

`DECLARE`:
: contém declarações de símbolos que podem ser usados em referências a operações ou grupo de operações, na definição de grupos ou em referências entre seções de uma mesma operação.

As seções `CODING` e `SYNTAX` contém as principais informações para uma ferramenta geradora de montadores. A figura 2.4 mostra um trecho de um modelo escrito utilizando LISA. A linha 1 dessa figura cria uma operação chamada `register`. A criação dessa operação serve como referência a registradores quando da definição da operação `add_op` na linha 9. A linha 3 usa uma seção `DECLARE` e uma declaração usando `LABEL` para definir um elemento de ligação interno chamado `index` que será utilizado nas demais seções dessa operação. Isso é necessário porque as seções `CODING`, `SYNTAX` e `EXPRESSION` declaradas nas linhas 4, 5 e 6 respectivamente, precisam fazer referência a um mesmo elemento, no caso um índice usado para acesso ao banco de registradores. A linha 4 cria uma imagem binária para essa operação composta pelos 4 bits menos significativos de `index`, que é ligado ao parâmetro númerico descrito na sintaxe da operação na linha 5. A linha 6 define o operando que é usado quando da descrição comportamental de `register`, onde `R` é um elemento de armazenagem previamente definido (banco de registradores). Não fica claro se `index` possui algum tipo já definido ou se a linha 5 é a responsável pela atribuição de um tipo numérico (possivelmente através de `#U` como um *unsigned int*).

```
1 OPERATION register
2 {
3   DECLARE     { LABEL index; }
4   CODING      { 0bx index:0bx[4] }
5   SYNTAX      { "R" index:#U }
6   EXPRESSION { R[index] }
7 }
8
9 OPERATION add_op
10 {
11   DECLARE  { GROUP dest, src1, src2 = {register}; }
12   CODING   { dest src1 src2 0b01000 0b1000 }
13   SYNTAX   { "ADD" src1 "," src2 "," dest }
14   BEHAVIOR { dest = src1 + src2; }
15 }
```

Figura 2.4: Trecho de um modelo escrito em LISA

A linha 9 da figura 2.4 cria uma operação chamada `add_op`, usada para representar uma instrução que adiciona dois registradores fontes a um destino. A linha 11 usa `DECLARE` e define através de `GROUP` as referências internas `dest`, `src1` e `src2` que herdam os atributos da operação `register` definida anteriormente na linha 1. As demais seções podem usar essas referências simbólicas em suas declarações e automaticamente herdam as seções relativas definidas em `register`. `CODING`, na linha 12, define o formato binário de `add_op`

como sendo a concatenação do formato binário de `dest`, de `src1` e o de `src2`, além dos literais binários `0b01000` e `0b1000`. A linha 13 define a sintaxe em linguagem de montagem como sendo composta pelo mnemônico `"ADD"` seguida da sintaxe de `src1`, do literal `','`, da sintaxe de `src2`, outro literal `','` e da sintaxe de `dest`. A linha 14 define o comportamento da instrução e é descrito como um programa em linguagem C.

Recentemente a linguagem ganhou novas construções para a descrição da sintaxe da linguagem de montagem e para codificação de operandos. Em especial, um bloco `OPERATION` pode ser qualificado com a palavra-chave `ALIAS`, significando que a seção de codificação `CODING` é a mesma de um outro bloco `OPERATION` (já que não é possível dois blocos `OPERATION` possuírem o mesmo valor para `CODING`). Para determinação de operandos que usem endereços simbólicos, a linguagem adotou o uso de um tipo pré-definido chamado `SYMBOL`. O montador trata operandos do tipo `SYMBOL` de forma especial, criando relocações no código.

LISA conta com um amplo conjunto de ferramentas consistindo de geradores de compiladores, montadores, linkeditores, simuladores e de *frontend*s para depuradores. Essas ferramentas atualmente fazem parte de uma plataforma para projetos de processadores conhecida como *LISA Processor Design Platform* (LPDP) [18]. Um modelo escrito em LISA é transformado em um formato intermediário, assim como na Sim-nML, que então pode ser processado pelas ferramentas geradoras. O montador LISA gerado suporta o uso de diretivas de montagem como seções de código e dado, e o uso de rótulos e referências simbólicas. O montador ainda gera como saída um arquivo objeto linkeditável em um formato específico chamado *LISA Object File* (LOF). Arquivos LOF's podem ser lidos pelo linkeditor LISA que gera arquivos no formato COFF.

Toda descrição referente ao conjunto de instruções em LISA é feita através da palavra-chave `OPERATION`. Não existem na linguagem seções específicas para definições de símbolos utilizados pelo montador como existe na ISDL. Na verdade não existem seções específicas para a descrição de um único aspecto de modelagem (exceto a definição de elementos estruturais). Mesmo o comportamento das instruções e informações de temporização ou restrição, são expressos todos através de um único bloco básico: a operação. `SYNTAX` permite a declaração de múltiplas sintaxes para uma mesma codificação, no entanto é necessário usar estruturas condicionais da linguagem para a seleção da sintaxe correta com base em valores de operandos. Não fica claro, pelas nossas referências, como instruções sintéticas compostas podem ser descritas em LISA.

Hoffman et al. [19] apresentou um estudo no qual analisou as características de modelagem em LISA a partir de diversas arquiteturas. Foram apresentados modelos para um micro-controlador ARM7 da Advanced RISC Machines Ltd, um DSP ADSP2101 da Analog Devices, e os DSP's TMS320C54x e TMS320C62x da Texas Instruments; esse último baseado em uma arquitetura VLIW. Em especial, o montador LISA consegue processar

sintaxes algébricas utilizadas na linguagem de montagem do DSP ADSP2101. Em uma sintaxe algébrica, instruções de adição são escritas como $X = Y + Z$ ao invés da forma usual utilizando-se mnemônicos e operandos. Não é mencionado em lugar algum recursos existentes em LISA para descrição de instruções sintéticas. O fato do montador gerar um arquivo objeto com formato próprio possibilita que a questão da relocação de símbolos possa ser tratada de forma mais geral, embora não haja detalhes sobre esse procedimento. A existência de um linkeditor LISA que consegue processar os arquivos gerados pelo montador encerra a cadeia de produção de software. Toda a tecnologia baseada em LISA foi transferida para a Fujitsu Limited em Setembro de 2000.

2.5 SLED

SLED [33], de *Specification Language for Encoding and Decoding*, é uma linguagem formalmente definida no ano de 1997 por Ramsey et al., usada para descrever instruções de máquina de forma simbólica. A linguagem consegue expressar os principais aspectos presentes em um conjunto de instruções (e.g., sintaxe, imagem binária e modos de endereçamento), possibilitando tanto a codificação para binário como a decodifição para uma linguagem de montagem. A linguagem também tem suporte à definição de instruções sintéticas e relocações, e foi desenvolvida para que suas especificações se assemelhem às encontradas em manuais de arquiteturas.

Uma especificação em SLED pode ser dividida basicamente nas seguintes seções:

tokens e campos:
: o objetivo desta seção é definir os formatos das instruções e seus campos. A diretiva `fields` é usada para agrupar campos no que é chamado *tokens* pela linguagem. Uma instrução é vista como uma sequência de um ou mais tokens.

padrões:
: padrões descrevem a representação binária das instruções, restringindo os valores que podem ser associados aos campos. Uma declaração de padrão utiliza a diretiva `patterns` seguida de um nome a ser associado ao padrão.

construtores:
: possibilitam o mapeamento entre a representação abstrata de uma instrução e sua representação binária. A linguagem usa a diretiva `constructors` para a definição de construtores. Existem basicamente dois tipos de contrutores: os tipados e os não-tipados. Construtores tipados são usados na definição de tipos de operandos, enquanto os não-tipados descrevem instruções.

casamento de sentenças:

possibilita que uma imagem binária possa ser decodificada, através da declaração do par `match` e `endmatch`.

A figura 2.5 ilustra um pequeno trecho de um modelo do conjunto de instruções do processador MIPS-I em SLED. A linha 1 declara o token `insn`, de 32 bits, que é formado pelos campos definidos nas linhas 2 e 3. Junto com a definição do nome do campo também é especificado sua posição no token. Podem ser declarados campos que se sobreponham, como o caso do `imm`, linha 2, e `funct`, linha 3. As linha 5, 6 e 7 definem um padrão de bits para `add` e `sub`. `add` representa um padrão com o campo `op` = 0 e o campo `funct` = 2, enquanto `sub` tem `op` = 0 e `funct` = 3. A linha 9 inicia a declaração de um construtor, chamado de `add` na linha 10. Nomes à esquerda da diretiva `is` indicam uma representação em linguagem de montagem da instrução, e possuem um espaço de nome (*namespace*) diferente das declarações em `fields` e `patterns`. O lado esquerdo da linha 10 declara uma instrução `add n, m` em linguagem de montagem, onde `n` e `m` são assumidos como valores inteiros. O lado direito associa um valor binário à instrução definida, composta dos padrãos `add` (definido na linha 6), do campo `rs` com o valor de `n`, e do campo `rd` com o valor de `m`.

```
1 fields of insn (32)
2   op 26:31 rs 21:25 rd 16:20 imm 0:15
3   funct 0:5
4
5 patterns
6   add is op=0 & funct=2
7   sub is op=0 & funct=3
8
9 constructors
10  add n, m is add & rs=n & rd=m
```

Figura 2.5: Trecho de um modelo do conjunto de instruções do processador MIPS-I escrito em SLED

O pacote de ferramentas conhecido como *New Jersey Machine-Code Toolkit* (NJMCT) [32] [34] é uma implementação de SLED, criada pelos mesmos autores da linguagem. Esse conjunto de ferramentas é divido em quatro partes principais:

`translator`:

responsável pela geração de código para a desmontagem do conjunto de instruções definido em um modelo SLED.

`generator`:
gera funções em linguagem C para codificação e relocação de instruções. Para cada bloco `constructors`, `generator` gera uma chamada de função que codifica aquela instrução. Como exemplo, o construtor definido na linha 10 da figura 2.5 gera uma rotina em C com o seguinte protótipo: `void add(unsigned n, unsigned m)`.

`library`:
fornece uma biblioteca de funções escritas em linguagem C de auxílio à `generator`. Essa biblioteca é independente da arquitetura modelada.

`checker`:
checa consistências entre um modelo SLED e montadores existentes.

Aplicativos que manipulam código de máquina (como montadores, desmontadores, geradores de código e depuradores) podem utilzar o pacote NJMCT para a codificação e decodificação de instruções. Existem duas aplicações que usam o NJMCT: um linkeditor redirecionável e otimizador conhecido como `mld` [8]; e um depurador redirecionável chamado `ldb` [35]. Ramsey et al. [31] modelaram as arquiteturas MIPS-I, SPARC-V8, Pentium e Alpha utilizando SLED. Descrições para arquiteturas RISC possuíam entre 100 e 200 linhas, enquanto a arquitetura CISC do Pentium possuía em média 500 linhas. Fernández et al. [9] utilizaram uma abordagem em que é possível testar, através do NJMCT, a corretude da especificação das instruções em SLED. O código gerado pelo pacote pode ter sua consistência testada antes de ser empregado em ferramentas aplicativas.

NJMCT fornece recursos importantes para um montador, como a descrição de instruções sintéticas, relocações e fatoração da sintaxe. No entanto, o pacote facilita somente a tarefa de codificação das instruções. É ainda necessário produzir os outros aspectos de um montador, como leitura e *parsing* de arquivos fontes, análise de expressões algébricas, reconhecimento de símbolos especiais e diretivas de montagem. NJMCT também não oferece formatos padrões para arquivos objetos.

2.6 GNU Binutils

GNU Binutils [28] é um pacote de ferramentas especialmente voltado para manipulação de arquivos objetos. Esse pacote faz parte da cadeia de ferramentas GNU para desenvolvimento de softwares, assim como o *GNU Compiler Collection* (`gcc`) [40] e o *GNU Debugger* (`gdb`) [39]. Fazem parte do pacote GNU Binutils ferramentas como: montador (`gas`), linkeditor (`ld`), gerenciadores de biblioteca (`ar` e `ranlib`), visualizadores de conteúdo de arquivos (`objdump`) etc.

O pacote GNU Binutils é estruturado em torno de duas bibliotecas principais: a *Binary File Descriptor Library* (BFD) [3], e a Opcodes [28]. A BFD é responsável pela abstração para formatos de arquivos objetos, provendo uma interface comum para todas as ferramentas. Já a Opcodes descreve informações do conjunto de instruções da arquitetura, como sua codificação e decodificação.

O GNU *Assembler* (`gas`) é o montador do pacote GNU Binutils. `gas` é usado principalmente como ferramenta de montagem para arquivos gerados pelo compilador `gcc`. O montador permite o redirecionamento de arquiteturas através do uso das bibliotecas BFD e Opcodes, e necessita que um conjunto de rotinas dependente de arquitetura seja escrito para cada novo redirecionamento. A figura 2.6 mostra uma estrutura simplificada do funcionamento do `gas`, onde o símbolo `'*'` indica quais componentes precisam ser reescritos no caso de redirecionamento. O *core* do montador oferece diversas funcionalidades, como: leitura e pré-processamento do arquivo fonte, reconhecimento e tratamento de estruturas comuns (como macros, rótulos, diretivas de montagem, expressões algébricas), interface básica para manuseio de símbolos relocáveis e geração de arquivo objeto. Uma arquitetura alvo deve implementar as rotinas pré-definidas pelo *core* do montador, que é responsável por executá-las em determinados trechos durante a transformação de um programa em linguagem de montagem para o correspondente binário.

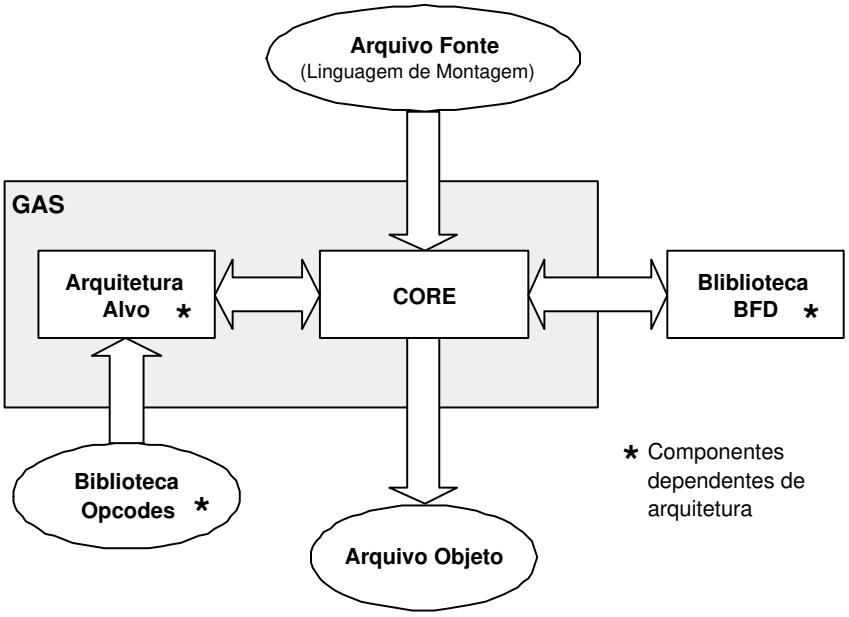

Figura 2.6: Estruturação básica do montador gas

O montador `gas` foi redirecionado para muitas familias de processadores com arquiteturas diversas, como CISC e RISC. Algumas famílias de processadores com versão do montador incluem: AMD 29k, Alpha, ARM, Renesas H8/300, Intel i386, Intel i860, Mo-

torola 680x0, MIPS I-IV, PDP-11, PowerPC, SPARC e TMS320C54x. Formatos de arquivos suportados incluem ELF, a.out e COFF. Stratling [41] mostrou em seu trabalho o redirecionamento do `gas` para acomodar o DSP TMS320C6x da Texas Instruments, que possui uma arquitetura VLIW. Rao [36] também apresentou, em sua dissertação de mestrado, o redirecionamento do `gas` e outras ferramentas do GNU Binutils no contexto de arquiteturas VLIW.

Abbaspour et al. [1] mostraram uma abordagem que redireciona o montador `gas` automaticamente através de uma descrição abstrata do conjunto de instruções de uma arquitetura. Eles trataram a questão da relocação de símbolos através da modelagem de uma ABI do processador. Foram especificados um modelo para um processador SPARC e outro para um Intel 386 usando essa abordagem. A partir desses modelos, suas ferramentas geraram os arquivos dependentes de arquitetura para as bibliotecas BFD, Opcodes e o próprio `gas`. Os autores testaram o montador gerado para o SPARC usando arquivos do *benchmark* SPEC2000 em linguagem de montagem, gerados pelo `gcc`. Os mesmos arquivos também foram processados pelo montador nativo `gas` para o SPARC, sendo que os arquivos gerados pelos dois montadores foram exatamente os mesmos. No entanto, o artigo não descreve com detalhes o modo de funcionamento de suas ferramentas e os aspectos considerados na realização dos testes. Tentamos obter maiores detalhes sobre os testes e resultados através de correspondência eletrônica, mas não obtivemos retorno algum.

O pacote GNU Binutils, e em especial o `gas`, fornece uma estrutura básica que viabiliza seu redirecionamento para diversas arquiteturas. No entanto, a estruturação e organização interna não estão adequadamente documentadas. Os manuais de uso, por exemplo, datam do início dos anos 90 e não há nenhum compromisso com atualizações e revisões frequentes. Os manuais internos estão bastante incompletos e desatualizados, não sendo sequer construídos no ato de compilação e instalação do pacote. Apesar dessa situação, entendemos que o pacote apresenta as características básicas necessárias para o redireciomento de arquiteturas, e justificamos seu uso no projeto ArchC na seção 2.7.

2.7 A Abordagem ArchC

Nesta seção apresentamos a abordagem que escolhemos para descrição das informações relativas a linguagem de montagem e geração de montadores. No restante da dissertação detalharemos os principais aspectos de nossa abordagem aqui discutida. Maiores informações sobre a linguagem ArchC podem ser encontradas na seção 3.1.

Nosso trabalho introduz construções na linguagem ArchC que possibilitam a descrição da sintaxe e símbolos utilizados na linguagem de montagem do conjunto de instruções de uma arquitetura. A descrição de símbolos se dá em uma seção específica e a sintaxe e

codificação das instruções são expressas em conjunto através do uso de formatadores, de uma maneira similar à função `scanf()` da linguagem C. Em ArchC também é possível definir novos formatadores que podem ser usados, por exemplo, na descrição de todo o conjunto de nomes de registadores de uma arquitetura. Além disso a linguagem oferece tipos básicos para operandos comuns, como expressões aritméticas, imediatos e rótulos para endereços. As novas construções permitem que um modelo seja facilmente descrito com base no manual da arquitetura, com suporte para instruções sintéticas.

A partir de um modelo de uma arquitetura descrito em ArchC, criamos a ferramenta `acasm` que é capaz de gerar automaticamente um montador correspondente. `acasm` gera os arquivos necessários para redirecionar o montador `gas` para a nova arquitetura. Justificamos a escolha dessa abordagem pelos seguintes motivos:

- a infra-estrutura do `gas` já fornece um núcleo de rotinas comuns em montadores como: leitura e pré-processamento de arquivos fontes, reconhecimento de diretivas de montagem, macros, rótulos e escrita do arquivo objeto em um formato adequado, como ELF ou COFF. Dessa forma podemos nos concentrar essencialmente na questão da geração de informações relativas ao modelo descrito em ArchC;

- `gas` utiliza uma estrutura compartilhada com todo o pacote Binutils, através das bibliotecas BFD e Opcodes. Dessa forma o redirecionamento de outras ferramentas do pacote é facilitada, já que estamos gerando parte das informações contidas nessas bibliotecas. De especial interesse é o redirecionamento do linkeditor `ld`, que trabalha com os arquivos gerados pelo montador `gas`;

- `gas` é o montador usado pelo `gcc` para a geração de arquivos objeto e tem sido importante para o desenvolvimento de software seguindo a filosofia de código aberto [43]. Existem versões do montador, feitas de forma *ad hoc*, para um grande número de famílias de processadores como: AMD 29k, Alpha, ARM, Intel i386, Motorola 680x0, MIPS I-IV, PDP-11, PowerPC, SPARC e TMS320C54x. A utilização do `gas` como base para a geração de código executável neste trabalho dá uma garantia da confiabilidade e robustez ao sistema, e de sua flexibilidade;

- podemos verificar a expressividade da linguagem ArchC e a corretude da ferramenta `acasm` modelando arquiteturas já redirecionadas no `gas` em ArchC. Os arquivos gerados pelo nosso montador podem então ser comparados aos arquivos gerados pelos montadores nativos `gas`.

2.7.1 Comparação com Outras Abordagens

Aspectos relacionados diretamente a construções das linguagens são difíceis de serem comparados. Encontramos uma dificuldade ainda maior em relação ao nosso trabalho

porque as características específicas sobre questões relacionadas à geração de montadores não são, em geral, explícitas e claras.

A descrição de símbolos para a linguagem de montagem em ArchC é feita de forma muito semelhante a `ISDL`, a partir de uma seção especial de mapeamento entre nomes e valores. `nML`, `Sim-nML` e `LISA` não fazem essa distinção, e os símbolos são de certa forma descritos implicitamente na descrição da sintaxe das instruções.

As outras linguagens dividem a tarefa de descrição da sintaxe da linguagem de montagem e a codificação das instruções em construções diferentes. `nML` e `Sim-nML` utilizam as construções `syntax` e `action`; já `LISA` adota a mesma filosofia, mas com as palavras-chaves `SYNTAX` e `CODING`; `ISDL` o faz através de uma ordem pré-definida de declarações encontradas em uma seção `Instruction_Set`. Em ArchC a sintaxe e codificação dos operandos são escritas conjuntamente através de `set_asm`, embora a codificação de campos fixos ainda seja feita através de `set_decoder` [3].

ArchC ainda possui uma construção específica para a criação de instruções sintéticas através da palavra-chave `pseudo_instr`. Das demais linguagens, somente `LISA` permite a descrição de múltiplas sintaxes para uma mesma instrução, e também o uso de vários blocos `OPERATION` associados a uma mesma codificação. No entanto não está claro se a linguagem consegue descrever instruções sintéticas compostas, ou seja, aquelas definidas em termos de duas ou mais instruções nativas.

A tabela 2.1 mostra algumas características encontradas nos principais montadores gerados a partir das ADL's vistas anteriormente. Em casos onde não foi possível se determinar claramente a presença de determinada característica, utilizamos o ponto de interrogação (`'?'`).

O formato para arquivo objeto do montador gerado atualmente pela nossa abordagem é `ELF`. No entanto novos formatos podem ser especificados com base na biblioteca `BFD`, que já conta com um número grande de formatos disponíveis. Dos demais montadores, somente o gerado pela `Sim-NML` tem suporte para `ELF`, não sendo possível a especificação de outros formatos até o momento. `LISA` utiliza um formato próprio cujas propriedades são desconhecidas.

As características de definição de macros, geração de listagens e diretivas de montagem já estão disponíveis no próprio *core* do montador `gas`, e portanto nossos montadores podem utilizá-las. Nas demais abordagens, `Sim-nML` e `LISA` contam com diretivas de montagem. `LISA` possui ainda recursos para reconhecimento de sintaxe algébrica, enquanto nossa abordagem necessita de um mnemônico e, opcionalmente, uma lista de operandos. Essa abordagem, no entanto, pode ser alterada para versões futuras de nossa ferramenta.

[3] Na verdade toda a descrição da codificação pode ser feita exclusivamente com a construção `set_asm`. No entanto, o uso de `set_decoder` permite a descrição de modelos específicos para a geração de simuladores, sem a necessidade da descrição das sintaxes das instruções imposta por `set_asm`.

Característica	Montadores				
	nML	Sim-nML	ISDL	LISA	ArchC
Formato de arquivo objeto gerado	[1]	ELF	BIN	LOF	ELF[2]
Diretivas de montagem		✓		✓	✓
Definição de macros				?	✓
Geração de listagens		✓		?	✓
Reconhecimento de sintaxe algébrica				✓	
Baseado em ferramentas redirecionáveis					✓
Disponível em domínio público	✓[3]	✓	✓[3]		✓

[1] o montador é interpretado
[2] pode ser facilmente redirecionado para outros formatos
[3] existe manual da linguagem, mas não código fonte

Tabela 2.1: Comparação de características entre as ADL's e montadores gerados

Somente ArchC adota uma abordagem baseada em outra ferramenta para geração de seus montadores. As demais linguagens constroem todos os módulos do montador, geralmente utilizando as ferramentas `flex` e `bison` para a geração do *lexer* e *parser*.

ArchC disponibiliza tanto o *parser* para a linguagem como as ferramentas geradoras ao público. Algumas linguagens como `nML` e `ISDL`, mais antigas, possuem manuais e publicações sobre a linguagem, mas nenhum código fonte disponível. Toda a tecnologia de `LISA` atualmente pertence à empresa Fujitsu Limited. Somente `Sim-nML` adota a mesma abordagem usada por ArchC para divulgação de seu trabalho.

Capítulo 3

Construções ArchC para a Geração Automática de Montadores

Neste capítulo apresentamos as novas construções fornecidas por ArchC que permitem a geração automática de montadores. Começamos revendo alguns aspectos da linguagem e, a seguir, descrevemos os novos comandos utilizados para declaração de símbolos, descrição da sintaxe da linguagem de montagem e da codificação das instruções, e a criação de instruções sintéticas.

3.1 A Linguagem ArchC

Uma descrição de arquitetura em ArchC pode ser dividida basicamente em duas partes: descrições dos recursos (`AC_ARCH`) e do conjunto de instruções (`AC_ISA`). Uma descrição dos recursos fornece informações sobre a estrutura da arquitetura tais como: quantidade de memória endereçável, hierarquia de memória, estrutura de *pipeline*, número de registradores etc. Uma descrição do ISA contém declarações de formatos, nomes e informações usadas na decodificação das instruções. Para cada instrução declarada deve-se ainda especificar seu comportamento descrito em linguagem C/C++, feita em um arquivo separado. Desse modo, um modelo para uma arquitetura em ArchC é basicamente dividido em três arquivos: o de recursos estruturais, o de declarações do ISA e o do comportamento das instruções. A figura 3.1 ilustra essa idéia em termos de blocos de descrição.

As informações que descrevem aspectos sobre a linguagem de montagem e codificação estão praticamente todas contidas na descrição do ISA. A única exceção é a declaração do *endianess* da arquitetura, declarado junto à descrição dos recursos. Portanto nos concentraremos principalmente nos aspectos envolvidos na descrição do ISA. As outras informações são utilizadas principalmente para a geração de simuladores, e podem ser encontradas com maiores detalhes em [37].

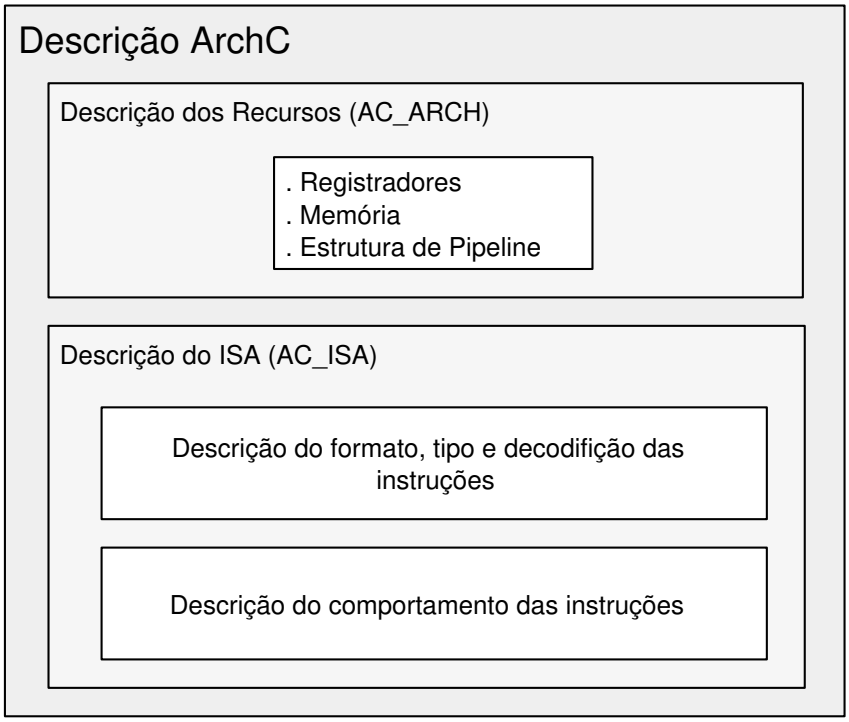

Figura 3.1: Estrutura de uma descrição em ArchC

3.1.1 Construções Existentes

O primeiro passo ao se descrever um ISA é declarar os formatos possíveis das instruções. Essa declaração é feita de maneira simples e intuitiva através do uso da palavra-chave `ac_format`. Um nome deve ser dado ao formato, e a seguir deve ser descrita uma sequência de nomes e tamanhos dos campos que o compõe. A figura 3.2 mostra o formato tipo R para o MIPS-I como descrito no livro de Patterson et al. [25]. Esse formato pode ser facilmente descrito em ArchC como mostra a linha 1 da figura 3.3. Nesse exemplo, o formato é declarado com o nome `tipoR` e é descrito por uma *string* com os nomes dos campos e seus respectivos tamanhos. O caracter `'%'` indica o início de um campo, e o identificador que o segue é seu nome. O valor após o caracter `':'` indica o tamanho do campo em bits.

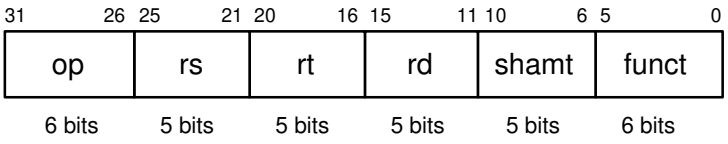

Figura 3.2: Formato de instrução tipo R do MIPS-I

Uma vez declarados os formatos, passa-se à declaração das instruções. Toda instrução em ArchC deve estar associada a um formato previamente definido. A palavra-chave `ac_instr` é usada para a declaração de instruções e usa como argumento um nome de formato já definido, através de uma sintaxe similar a de *templates* em C++. A linha 2 da figura 3.3 mostra a declaração das instruções `add` e `addu` do MIPS-I, associadas ao formato `tipoR` previamente definido.

```
1 ac_format tipoR = "%op:6 %rs:5 %rt:5 %rd:5 %shamt:5 %funct:6";
2 ac_instr<tipoR> add, addu;
3
4 add.set_decoder(op=0x00, funct=0x20);
5 addu.set_decoder(op=0x00, funct=0x21);
```

Figura 3.3: Trechos de um modelo MIPS-I escrito em ArchC

Toda instrução declarada possue propriedades que podem ser configuradas através de métodos. O principal método de nosso interesse é `set_decoder`, que especifica a sequência usada para a decodificação de uma instrução. Essa sequência deve atribuir valores aos campos do formato associado à instrução. As linhas 4 e 5 da figura 3.3 atribuem uma sequência de decodifição para as instruções `add` e `addu`, respectivamente. Assim, uma instrução será decodificada como `add` se, e somente se, os campos `op` e `funct` contiverem respectivamente os valores `0x00` e `0x20`.

Até a versão 1.2, ArchC contava também com o método `set_asm` para descrição da sintaxe em linguagem de montagem de uma instrução. Apesar desse método ter sido introduzido desde as primeiras versões da linguagem, seu uso estava restrito a algumas informações de desmontagem usadas pela ferramenta de simulação. É apropriado dizer ainda que as primeiras versões estavam basicamente preocupadas com a geração de simuladores, e portanto `set_asm` não continha informações suficientes e necessárias à geração de montadores. Estendemos a sintaxe de `set_asm` para capturar as informações necessárias para a geração automática de montadores conforme é descrito na seção 3.2.2.

Finalizamos esta seção apresentando um sumário das declarações já existentes em ArchC que afetam o montador gerado:

`set_endian`:
: configura o *endianess* da arquitetura para *big* ou *little*.

`ac_format`:
: declara um formato de instrução através da descrição de seus campos e respectivos tamanhos.

`ac_instr`:
declara uma instrução que deve ser associada a um formato previamente definido.

`set_decoder`:
método associado a cada instrução, responsável pela descrição de sua sequência única de decodificação.

3.2 Suporte para a Geração de Montadores

A tarefa básica de um montador é a de transformar instruções escritas em uma linguagem de montagem em suas respectivas representações binárias. Uma linguagem de montagem representa simbolicamente as instruções de máquina através de mnemônicos e operandos[1]. Geralmente uma instrução em linguagem de montagem equivale a uma única instrução de máquina. Instruções sintéticas não fazem parte do conjunto de instruções de uma arquitetura, sendo definidas em função de instruções nativas. É comum uma instrução sintética ser definida como sendo duas ou mais instruções nativas.

Para que um montador possa transformar uma instrução em linguagem de montagem em sua respectiva imagem binária são necessárias informações sobre:

sintaxe da instrução:
identifica o mnemônico e os tipos de operandos válidos, como nomes de registradores ou valores imediatos.

codificação da instrução:
associa a cada campo do formato da instrução um valor binário de acordo com seu mnemônico e operandos. Caso nomes de registradores sejam usados, é necessário saber o mapeamento entre cada nome e seu respectivo valor.

A função dos novos comandos é justamente capturar essas informações no modelo da arquitetura. Assumimos, em ArchC, que toda instrução em linguagem de montagem é descrita como um mnemônico e, opcionalmente, uma lista de operandos. Operandos podem ser representados simbolicamente como acontece, por exemplo, em casos de nomes de registradores. Instruções sintéticas podem ser definidas em função de instruções nativas previamente declaradas.

As próximas seções descrevem em detalhes os novos comandos que permitem a geração automática de montadores a partir de modelos escritos em ArchC. A sintaxe formal de cada nova construção pode ser encontrada no apêndice A.

[1]Linguagens algébricas podem não possuir mnemônicos explícitos.

3.2.1 Declarando Símbolos

Um montador geralmente permite o uso de certos símbolos específicos da arquitetura na descrição da sintaxe de instruções. O melhor exemplo do uso de símbolos é na descrição dos nomes de registradores usados como operandos nas instruções.

ArchC permite a declaração de símbolos específicos a uma dada arquitetura através de uma construção `ac_asm_map`. Um bloco `ac_asm_map` deve receber um nome de referência e conter uma lista mapeando cada símbolo a um valor. O nome do bloco pode ser usado na descrição da sintaxe das instruções como descrito na seção 3.2.2. Desse modo, caso um símbolo declarado através de `ac_asm_map` seja encontrado ao se fazer o *parsing* das instruções em linguagem de montagem, o montador sabe o valor que deve ser usado na sua codificação.

```
1  ac_asm_map reg {
2    "$"[0..31] = [0..31];
3    "$zero" = 0;
4    "$at"   = 1;
5    "$v"[0..1] = [2..3];
6    "$a"[0..3] = [4..7];
7    "$t"[0..7] = [8..15];
8    "$t"[8..9] = [24..25];
9    "$s"[0..7] = [16..23];
10   "k"[0..1]  = [26..27];
11   "$gp" = 28;
12   "$sp" = 29;
13   "$s8", "$fp" = 30;
14   "$ra" = 31;
15 }
```

Figura 3.4: Uma construção `ac_asm_map` descrevendo os registradores do MIPS-I

A figura 3.4 mostra um exemplo de uma construção `ac_asm_map` para nomear os registradores do MIPS-I utilizando a nomenclatura encontrada no livro de Patterson et al. [25]. A linha 1 atribui o nome `reg` como referência ao conjunto de símbolos mapeados nas linhas de 2 à 14. Esse nome pode ser usado ao se declarar operandos registradores na sintaxe das instruções MIPS-I mais tarde. As linhas de 2 à 14 compõem o corpo da construção. Um símbolo deve ser descrito entre dupla aspas; dessa forma, seu espaço de nomes é diferente do espaço de nomes ArchC. Seguindo o nome do símbolo, após o sinal de igual (`'='`), seu valor deve ser especificado como um inteiro positivo. Vários símbolos podem ser associados a um mesmo valor, como acontece na linha 13, mas um mesmo símbolo não pode estar associado a dois ou mais valores. A sintaxe permite ainda que

faixas de valores sejam usadas como ocorre normalmente em nomes de registradores. Por exemplo, na linha 5, os símbolos `"$v0"` e `"$v1"` são criados com os respectivos valores `2` e `3`. As faixas de valores especificadas em ambos os lados do sinal igual (`'='`) devem ter o mesmo número de elementos para que a declaração seja válida.

3.2.2 Descrevendo a Sintaxe e Codificação

Informações sobre a sintaxe e codificação das instruções são descritas conjuntamente em ArchC através de `set_asm`. Conforme já descrevemos, `set_asm` é um método que toda instrução declarada a partir de `ac_instr` possui. As informações codificadas por `set_asm` são geralmente dinâmicas, como valores de registradores e referências simbólicas encontradas na linguagem de montagem. Campos especifícos e característicos de uma determinada instrução, como o `opcode`, devem ser especificados através do comando `set_decoder`, usado primeiramente na construção de decodificadores para os simuladores gerados.

A sintaxe e codificação são descritas de uma maneira bastante intuitiva através de `set_asm`. Vamos apresentar seu formato geral de maneira informal (para a definição formal, consulte o apêndice A) através de uma sintaxe usada em linguagem C:

```
set_asm(char *sintaxe, ...)
```

Fizemos essa definição de propósito, devido ao comando lembrar a função `scanf()` da linguagem C em muitos aspectos. A *string* `sintaxe` deve descrever a sintaxe da instrução separada em mnemônico e, opcionalmente, operandos. O mnemônico é separado dos operandos pela ocorrência do primeiro caracter de espaço encontrado em `sintaxe`. Como `scanf()`, `set_asm` aceita especificadores de conversão (doravante chamado de formatadores) que permitem tratar certos aspectos da *string* `sintaxe` de forma não-literal.

Uma especificação de conversão é iniciada pelo caracter `'%'` seguido do nome do formatador. Formatadores podem ser declarados através da palavra-chave `ac_asm_map`, conforme descrito na seção 3.2.1, ou podem ser selecionados de uma lista pré-definida em ArchC. Um formatador informa ao montador gerado o tipo de informação que ele deve reconhecer como válida naquele trecho da sintaxe, enquanto os outros caracteres da *string* `sintaxe` devem ser reconhecidos literalmente. Cada formatador usado em `sintaxe` exige que um dos campos definidos no formato da instrução seja especificado na lista de argumentos (`'...'`). Esses argumentos informam ao gerador de montadores como os valores associados aos formatadores são codificados na imagem binária gerada.

Em `scanf()`, formatadores informam o tipo de informação esperada como entrada (por exemplo, `%s` informa que a entrada deve ser considerada como uma *string*), e o argumento associado informa o endereço de memória aonde a informação deve ser armazenada. Em ArchC, formatadores permitem que montadores reconheçam um determinado tipo de informação em trechos da sintaxe, e o argumento associado informa o campo da instrução

no qual o valor do formatador deve ser armazenado. Veremos mais adiante que formatadores ainda permitem informar como seus valores devem ser codificados através do uso de modificadores.

A figura 3.5 mostra diferentes declarações `set_asm` para um modelo MIPS-I que usaremos como base para algumas exemplificações. A linha 1 mostra a definição da sintaxe para a instrução `add`, definida na linha 2 da figura 3.3. Essa declaração usa como mnemônico a *string* `add`, seguida de operandos descritos através do formatador `reg` definido na figura 3.4. Desse modo, o montador pode reconhecer os nomes de registradores definidos em `reg` como parte da sintaxe e utilizar seus valores durante a codificação. Cada operando registrador declarado refere-se a um campo do formato da instrução conforme definido: o primeiro registrador é o campo `rd`, o segundo é o campo `rs` e o terceiro `rt`. Esses campos foram definidos na declaração do formato da instrução `add` conforme mostra a linha 1 da figura 3.3. Após o caracter `'%'` é opcional colocar o formatador entre colchetes, como acontece no primeiro `reg` da linha 1. A linha 2 complementa a declaração da codificação através de `set_decoder`, para a parte fixa da instrução.

```
1 add.set_asm("add %[reg], %reg, %reg", rd, rs, rt);
2 add.set_decoder(op=0x00, func=0x20);
3
4 addi.set_asm("addi %reg, %reg, %imm", rt, rs, imm);
5 addi.set_decoder(op=0x08);
6
7 sw.set_asm("sw %reg, %exp(%reg)", rt, imm, rs);
8 sw.set_decoder(op=0x2B);
```

Figura 3.5: Trecho de um modelo MIPS-I mostrando o uso de `set_asm`

Formatadores Pré-definidos

ArchC fornece alguns formatadores pré-definidos para os tipos mais comuns de operandos encontrados em instruções. O tipo pré-definido base é chamado de `exp` e pode representar operandos descritos como expressões aritméticas e simbólicas conforme descrito no manual do montador `gas` [5]. A partir de `exp`, são definidas duas especializações: `imm` para operandos inteiros imediatos, e `addr` para operandos que utilizem referências simbólicas. Note que pelo fato de `imm` e `addr` serem especializações de `exp`, todo operando `imm` ou `addr` é também do tipo `exp`.

A linha 4 da figura 3.5 utiliza o formatador `imm` como terceiro operando da instrução `addi` do MIPS-I. Já a linha 7 mostra o uso do formatador `exp` como especificador de *offset* relativo a um registrador base para a instrução `sw`.

Modificadores

Um formatador tem sempre um campo interno de 32-bit associado [2], representando o valor literal de uma certa instância daquele formatador. Quando esse valor é armazenado em um dado campo de uma instrução, seu tamanho é limitado pelo tamanho do campo. Por exemplo, considere que a seguinte sintaxe seja definida, com o campo `imm16` de 16-bit:

```
rst.set_asm("rst %imm", imm16);
```

Uma instância válida para esse formatador é -1, que será representado como um inteiro em 32-bit de valor `0xFFFFFFFF`. Ao ser efetuada a codificação desse valor, o mesmo será truncado para o tamanho limite do campo, no caso 16-bit, e será codificado como `0xFFFF`. Esse comportamento é o correto e esperado para esse exemplo, mas podem existir ocasiões onde o valor do formatador deve ser codificado de maneira diferente.

A figura 3.6 apresenta um caso em que a maneira padrão de codificação falha. Nesse exemplo, desejamos criar a sintaxe e codificação para a instrução `ba` do SPARC-V8. Essa instrução realiza um salto relativo ao PC, usando endereçamento alinhado à palavra. Seguindo a figura 3.6, em (a) está a especificação da instrução em ArchC, enquanto em (b) descrevemos um hipotético trecho de um programa em linguagem de montagem que a utiliza. Pela maneira como foi especificada, a codificação do operando relativo ao endereço do salto está incorreta, conforme apresentado em (c). O resultado esperado seria a codificação do valor -1, já que o salto está sendo feito para a instrução anterior. Os motivos que causaram o erro na codificação dessa instrução são dois:

1. o valor corrente do PC não foi subtraído do endereço simbólico.
2. a codificação do valor do endereço resultante não usou endereçamento alinhado à palavra, e sim ao byte.

Em ArchC é possível alterar a forma padrão de codificação dos valores através do uso de modificadores, especificados juntos aos formatadores pré-definidos. Um modificador é identificado por uma letra maiúscula que ao ser anexada ao nome de um formatador, modifica a forma como seu valor é codificado. Todo modificador executa uma ou várias operações intermediárias sobre o valor associado ao formatador antes do mesmo ser codificado no respectivo campo da instrução. Os principais modificadores existentes atualmente em ArchC são:

`L[ n ][ s|u ]` - bits menos significativos (*Low bits*)
: Este modificador seleciona os n bits menos significativos do valor associado ao formatador. Se s for especificado, o n-ésimo bit é considerado como bit de sinal e é

[2] Estamos sempre considerando o bit menos significado como o mais à direita.

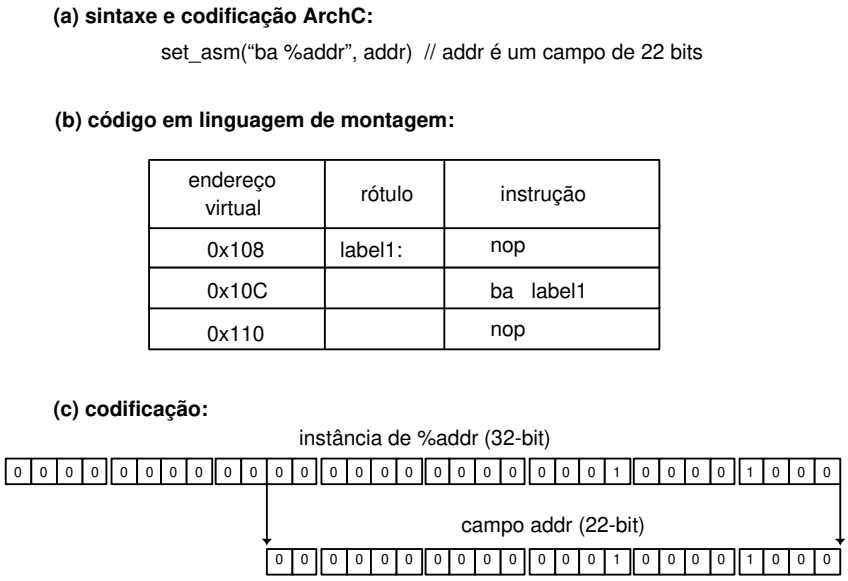

Figura 3.6: Codificação errada para um operando relativo ao PC

extendido antes que o valor final seja codificado (*u* é o padrão, e desconsidera o bit de sinal). O modificador `L` isolado corresponde a forma padrão de codificação, e constitui uma redundância (`%imm == %immL`).

`H[ n ][ s|u ]` - bits mais significativos (*High bits*)
Este modificador seleciona os n bits mais significativos do valor associado ao formatador através de deslocamento à direita de $32-n$ bits. Caso n não seja especificado, o valor usado é o do tamanho do campo associado ao formatador. Se s for especificado, o n-ésimo bit é considerado como bit de sinal e é extendido antes que o valor final seja codificado (*u* é o padrão, e desconsidera o bit de sinal).

`R[ n ]` - relativo ao PC (*PC Relative*)
Este modificador subtrai o valor corrente do PC do valor associado ao formatador. É possível se especificar um adendo positivo n (em bytes) a ser somado ao PC, já que algumas arquiteturas realizam saltos relativo ao `PC+<deslocamento>`, como por exemplo `PC+4` no MIPS-I. Para deslocamentos negativos, é necessario usar o prefixo *b* antes do adendo, como por exemplo `%addrRb4` para `PC-4`.

`A[ n ][ u|s ]` - endereçamento alinhado à n-bytes (*n-byte Aligned addressing*)
Este modificador indica que o valor associado ao formatador usa um endereçamento alinhado à n bytes. Se n não for especificado, o padrão é o tamanho da palavra para a arquitetura sendo descrita (para arquiteturas 32-bit esse valor é 4). O argumento n deve ser uma potência de 2, já que o resultado da aplicação desse modificador

é: $valor formatador / \log_2 n$. Na prática, o resultado da aplicação do logarítmo é truncado para baixo (por exemplo, para $n = 3$ o resultado é 1) e usado para efetuar um deslocamento à direita sobre o valor do formatador. Por padrão, o modificador efetua a extensão de sinal após o deslocamento ser aplicado, mas esse comportamento pode ser evitado ao especificar-se u como sufixo. O modificador `A1` é redundante (`%addr == %addrA1`), já que na forma padrão todo o endereçamento é alinhado ao *byte*.

Somente os formatadores pré-definidos suportam o uso de modificadores. É possível especificar mais de um modificador para um mesmo formatador, porém `H` e `L` são exclusivos. Caso mais de um modificador seja especificado para um mesmo formatador, a ordem em que aparecem não é relevante. Nesse caso a ordem de aplicação dos modificadores é sempre, em ordem de prioridade: `R`, `H/L` e `A`.

As figuras 3.7 e 3.8 apresentam exemplos reais do uso de modificadores em uma instrução SPARC-V8 e MIPS-I, respectivamente. São apresentados a definição `set_asm` para cada instrução, um exemplo onde podem ser encontradas em um programa em linguagem de montagem, e as operações realizadas para a codificação dos campos que necessitaram modificadores. A figura 3.7 mostra a correta codificação da instrução `ba`, em oposição ao que foi apresentado na figura 3.6. Já a figura 3.8 mostra o uso do modificador `H` para a instrução `lui`, selecionando os bits mais significativos do operando imediato.

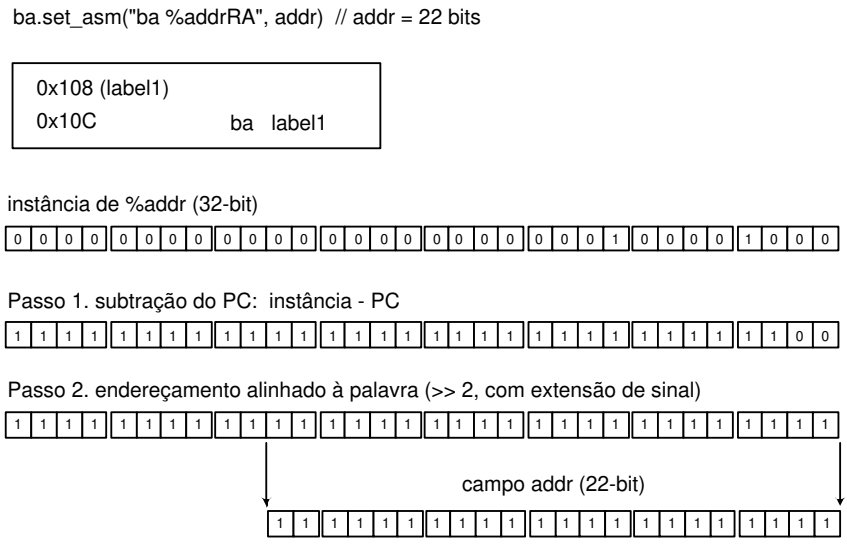

Figura 3.7: Sintaxe `set_asm` correta para a codificação do campo `disp22` da instrução `ba` presente no SPARC-V8

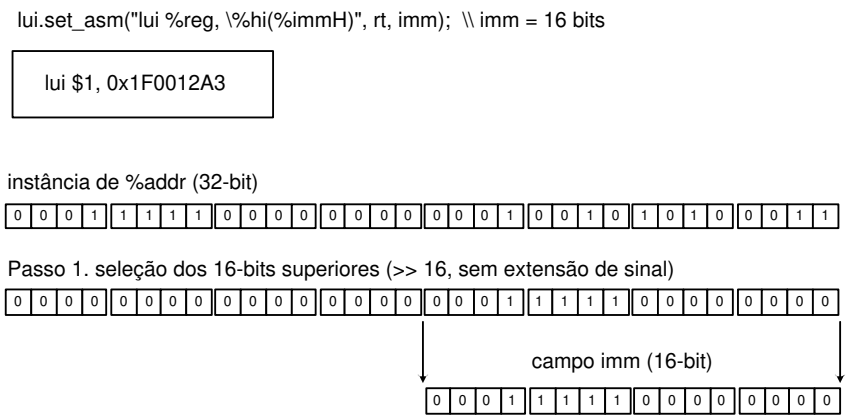

Figura 3.8: Sintaxe `set_asm` correta para a codificação do campo `imm` da instrução `lui` presente no MIPS-I

Sobrecarga da Sintaxe

É possível definir mais de uma declaração `set_asm` para uma mesma instrução, bastando que suas sintaxes sejam diferentes. Diversas declarações tornam possível a sobrecarga de instruções e a definição de instruções sintéticas simples (ver seção 3.2.3). A sobrecarga da sintaxe é útil quando há a necessidade de se expressar operandos de formas diferentes para uma mesma instrução.

A figura 3.9 mostra dois exemplos de sobrecarga de sintaxe para duas instruções presentes no MIPS-I. As linhas 1 e 2 dessa figura descrevem duas sintaxes e codificações diferentes para a instrução `lui`. Já as linhas 5 e 6 descrevem duas sintaxes comumente encontradas para a instrução `jalr`. A versão da linha 6 suprime a necessidade do segundo operando, que fica implicitamente codificado como o registrador `$ra`.

```
1 lui.set_asm("lui %reg, %exp", rt, imm);
2 lui.set_asm("lui %reg, \%hi(%expH)", rt, imm);
3 lui.set_decoder(op=0x0F, rs=0x00);
4
5 jalr.set_asm("jalr %reg, %reg", rd, rs);
6 jalr.set_asm("jalr %reg", rs, rd="$ra");
7 jalr.set_decoder(op=0x00, func= 0x09);
```

Figura 3.9: Trechos de um modelo MIPS-I mostrando o uso de `set_asm` para sobrecarga de sintaxe

Formatadores e Mnemônicos

Embora o principal uso de formatadores seja na definição da sintaxe de operandos, também é possível que sejam usados junto ao mnemônico. O uso de formatadores para especificação de mnemônicos se justifica quando afixos podem ser adicionados à base do mnemônico, condicionados a alguma codificação. Por exemplo, na arquitetura SPARC-V8 algumas instruções de saltos possuem um bit *annul* que pode anular ou não a instrução seguinte. Baseado no valor desse bit, os mnemônicos base para as instruções de saltos podem conter um sufixo `",a"`, significando que o bit *annul* deve ser codificado com o valor 1.

A figura 3.10 mostra um trecho de um modelo ArchC de uma arquitetura SPARC-V8 na qual são declaradas algumas instruções que utilizam o bit *annul*. A linha 1 mostra o formato utilizado pelas instruções declaradas na linha 3. O segundo campo desse formato (`an`) é o bit *annul* a que nos referíamos. A linha 5 declara o formatador `annul` que é usado nas especificações das sintaxes para os mnemônicos das instruções, definidas nas linhas 10, 11 e 12. É possível nesse caso declarar um símbolo nulo, como é feito na linha 6, indicando a não necessidade de um afixo no mnemônico. Note, no entanto, que em outros casos o uso do símbolo nulo pode não fazer sentido, como acontece na definição dos nomes de registradores.

```
1 ac_format Type_F2B = "%op:2 %an:1 %cond:4 %op2:3 %disp22:22:s";
2
3 ac_instr<Type_F2B> ba, bn, bne;
4
5 ac_asm_map annul {
6   "" = 0;
7   ",a" = 1;
8 }
9
10 ba.set_asm("ba%[annul] %expRA", an, disp22);
11 bn.set_asm("bn%[annul] %expRA", an, disp22);
12 bne.set_asm("bne%[anul] %expRA", an, disp22);
```

Figura 3.10: Trechos de um modelo SPARC-V8 mostrando o uso de formatadores em mnemônicos

A sintaxe declarada para a instrução `bne`, definida na linha 12 da figura 3.10, permite que o montador gerado para esse modelo reconheça os seguintes mnemônicos:

```
bne    label   -> codifica o bit annul como 0
bne,a  label   -> codifica o bit annul como 1
```

Na versão atual, ArchC só permite o uso de 1 formatador junto à definição da sintaxe de mnemônicos.

3.2.3 Criando Instruções Sintéticas

Instruções sintéticas, ou pseudo-instruções, são instruções definidas em função de outras pré-existentes por serem convenientes na programação em linguagem de montagem. Instruções sintéticas sempre geram instruções e, portanto, não devem ser confundidas com *pseudo-ops*, que provêm informações para o montador mas usualmente não geram instruções.

Para fins de definição em ArchC, podemos dividir as instruções sintéticas em duas classes: as simples, que permitem um mapeamento de instruções um-a-um e não repetem o uso de registradores como operandos; e as complexas, que geralmente são constituídas de duas ou mais instruções nativas e/ou possuem repetição de operandos na sua definição.

Instruções sintéticas simples podem ser declaradas através da própria palavra-chave `set_asm`. Geralmente alguns de seus operandos possuem valores pré-definidos de forma implícita, expressos como argumentos. A tabela 3.1 mostra algumas instruções sintéticas definidas para o SPARC-V8, presentes no manual da arquitetura [38]. As 3 primeiras instruções sintéticas desta tabela são simples, e podem ser declaradas através de `set_asm` em ArchC como nas linhas 1, 2 e 3 da figura 3.11. Operandos implícitos pré-definidos são declarados como argumentos através de uma atribuição. Atribuições podem incluir valores inteiros imediatos ou símbolos previamente definidos através de `ac_asm_map`. A linha 1 da figura 3.11, por exemplo, define a pseudo instrução `ret` que na verdade é a instrução `jmpl` com seus 3 operandos pré-definidos. Esses operandos são definidos na seção de argumentos da construção `set_asm`, através da atribuição de valores aos campos do formato da instrução.

Instruções Sintéticas		Instruções SPARC-V8	
`ret`		`jmpl`	`%i7+8, %g0`
`mov`	`reg-s, reg-d`	`or`	`%g0, reg-s, reg-d`
`not`	`reg-s, reg-d`	`xnor`	`reg-s, %g0, reg-d`
`not`	`reg`	`xnor`	`reg, %g0, reg`
`set`	`value, reg`	`sethi` `or`	`%hi(value), reg` `reg, %lo(value), reg`

Tabela 3.1: Mapeamento entre instruções sintéticas e instruções SPARC-V8

Em geral, ArchC fornece a construção `pseudo_instr` para a descrição de instruções sintéticas. Nessa construção devem ser especificadas a sintaxe da pseudo instrução, através de uma *string* de forma similar a `set_asm`, e uma lista com as instruções a serem expandidas para a pseudo instrução. Essas instruções necessariamente já devem estar definidas através de uma construção `set_asm`. Os operandos das instruções a serem expandidas podem utilizar operandos definidos na sintaxe de uma pseudo instrução através do caracter `'%'` seguido de um número natural i, com base zero, indicando a posição do operando na pseudo instrução. Desta forma, `%0` indica o primeiro operando, `%1` o segundo, e assim sucessivamente.

```
1 jmpl_imm.set_asm("ret", rs1="%i7", simm13=8, rd="%g0");
2 or_reg.set_asm("mov %reg, %reg", rs1="%g0", rs2, rd);
3 xnor_reg.set_asm("not %reg, %reg", rs1, rs2="%g0", rd);
4
5 pseudo_instr("not %reg") {
6   "xnor %0, \%g0, %0";
7 }
8
9 pseudo_instr("set %imm, %reg") {
10  "sethi \%hi(%0), %1";
11  "or %1, %0, %1";
12 }
```

Figura 3.11: Instruções sintéticas definidas em ArchC para um modelo SPARC-V8

As linhas 5 e 9 da figura 3.11 declaram as duas últimas pseudo instruções definidas na tabela 3.1 para um modelo SPARC-V8. A instrução `not` que utiliza somente um registrador como operando não pode ser descrita através de uma construção `set_asm`, pois repete o uso de um mesmo registrador em dois campos distintos na instrução `xnor` nativa. Ela então foi declarada através de uma construção `pseudo_instr` definida na linha 5. A linha 6 define a única instrução que compõe essa pseudo, e faz referência ao primeiro e único formatador definido na sintaxe através de `%0`. Quando o montador reconhece a sintaxe para uma instrução `not` com um operando, ele subtitui o símbolo utilizado nesse operando em todas as referências especificadas na sintaxe da instrução `xnor`. Nesse caso serão substituídos os primeiro e terceiro operandos, enquanto o segundo é literal. Note ainda que a sintaxe para a instrução `xnor` deve ter sido previamente definida através de uma construção `set_asm` com o mnemônico `"xnor"` e três operandos do tipo `reg` separados por vírgulas.

Ainda na figura 3.11, a linha 9 mostra a declaração de uma pseudo instrução `set` composta de duas instruções nativas, definidas nas linhas 10 e 11. Nesse caso, ambas

instruções utilizam referências aos formatadores definidos na sintaxe da pseudo, através de `%0` para `imm` e `%1` para `reg`. Como exemplo, se o montador encontrar uma instrução `"set 10, %g2"` em uma listagem fonte em linguagem de montagem, ele vai gerar as seguintes instruções nativas:

```
sethi %hi(10), %g2    //  primeiro operando substituído por 10,
or    %g2, 10, %g2    //  e segundo operando substituído por %g2
```

Capítulo 4

`acasm`: O Gerador Automático de Montadores ArchC

Neste capítulo apresentamos uma ferramenta ArchC para geração automática de montadores, denominada `acasm`. Mostramos a infra-estrutura que ArchC proporciona para a geração de ferramentas, as alterações na sintaxe da linguagem para acomodar as novas construções, e a geração dos arquivos dependentes de arquitetura que compõem um novo montador baseada no pacote GNU Binutils.

4.1 A Geração de Ferramentas em ArchC

ArchC possui um único módulo, chamado de `acpp` (*ArchC pre-processor*), responsável pela leitura dos modelos de arquitetura e construção de uma representação intermediária. Essa representação, por motivos práticos, atualmente é mantida em memória e portanto deve ser reconstruída toda vez que uma ferramenta geradora é executada, através de chamadas de funções fornecidas por `acpp`.

A figura 4.1 mostra a estrutura utilizada na geração de ferramentas em ArchC. Os arquivos que compõem a descrição de um modelo para uma dada arquitetura são processados por `acpp`, e uma representação intermediária é gerada em memória. Cada ferramenta geradora tem, desse modo, acesso às informações de que necessita para a construção de sua respectiva ferramenta.

O desenvolvimento da ferramenta `acasm` envolveu dois aspectos:

- alteração do pré-processador – inclusão da sintaxe e semântica para as novas construções da linguagem;
- geração do montador – geração dos arquivos, baseada no pacote GNU Binutils, que compõem o montador.

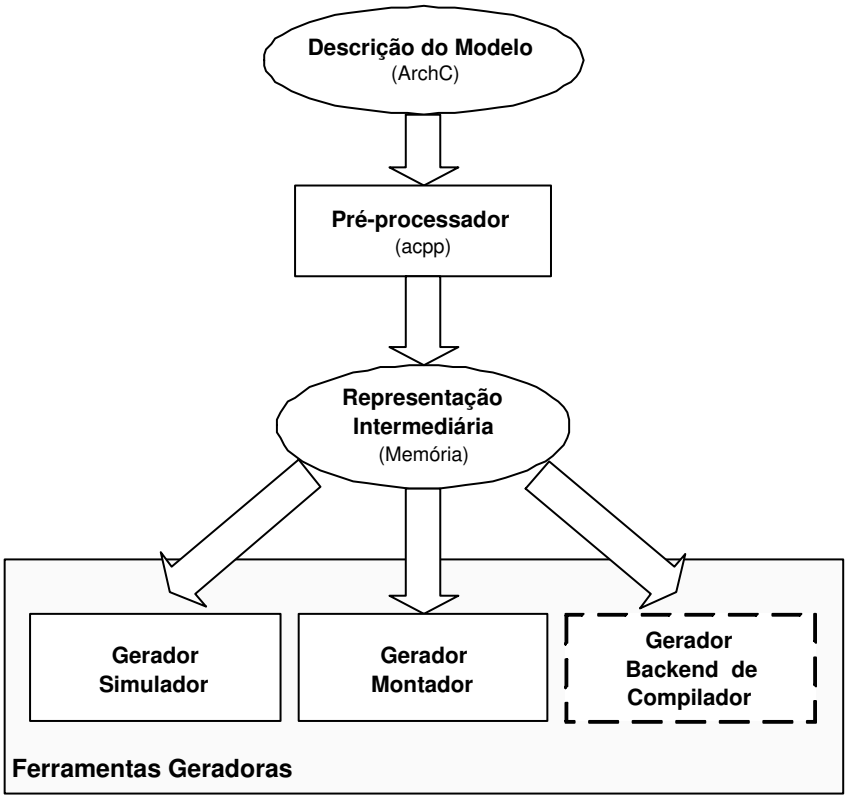

Figura 4.1: Estrutura para geração de ferramentas em ArchC

Descrevemos com maiores detalhes esses dois aspectos no restante deste capítulo.

4.2 O Pré-processador ArchC

`acpp` é composto pelos analisadores léxico, sintático e semântico da linguagem ArchC, gerados através das ferramentas GNU `flex` [26] e `bison` [4]. A vantagem em se utilizar essas ferramentas está na sua simplicidade em se definir os elementos estruturais, como expressões regulares e regras de produção de GLC.

Um programa em `flex`, o gerador de analisadores léxicos, é descrito através das chamadas *regras*, que são pares de expressões regulares e código em linguagem C. Já um programa em `bison`, o gerador de analisadores sintáticos, suporta a descrição de gramáticas livres de contexto LALR através da sintaxe `bison`, a qual provê construções para símbolos da gramática e de suas regras de produção em uma forma parecida com a BNF. Ações semânticas podem ser associadas às regras através de trechos de código em C.

Para a construção da ferramenta geradora de montadores notamos a necessidade da inclusão de novas construções em ArchC, sem as quais seria impossível a geração de

montadores. Essas construções foram motivadas em parte pelo estudo que fizemos sobre as outras ADL's, e em parte pela necessidade de nossa infra-estrutura de montadores, a qual usa o pacote GNU Binutils.

Adicionamos as novas construções ao arquivo `bison` através da descrição de novas regras de produção, conforme definidas no apêndice A. Essa descrição ocorreu quase que de forma direta como pode ser visto na figura 4.2, que mostra a definição da sintaxe para a construção `pseudo_instr` em `bison`. Repare na semelhança entre a descrição em EBNF encontrada no apêndice A e sua representação através de uma gramática `bison` na figura 4.2. A única diferença é que precisamos escrever estruturas de repetição de forma explícita em `bison`, enquanto na sua definição formal usamos estruturas especiais da EBNF.

```
pseudodec: PSEUDO_INSTR LPAREN STR RPAREN
           LBRACE pseudobodylist RBRACE ;

pseudobodylist: pseudobodylist pseudobody
              | pseudobody
              ;

pseudobody: STR SEMICOLON ;
```

Figura 4.2: Definição da sintaxe para `pseudo_instr` em `bison`

Cada ação semântica está associada a uma regra de produção introduzida que valida os nomes de símbolos e faixas de valores utilizados, armazenando as informações em uma estrutura de dados interna. Essas informações ficam armazenadas na memória e são utilizadas pela ferramenta geradora na produção dos arquivos que vão compor o montador.

4.3 A Geração do Montador

A ferramenta `acasm` é essencialmente o módulo de geração de arquivos fontes que compõem os montadores. Esse módulo utiliza as informações capturadas e armazenadas pelo pré-processador para gerar uma série de arquivos dependentes de arquitetura em conformidade com a infra-estrutura fornecida pelo pacote GNU Binutils. Os arquivos gerados são inseridos na árvore de diretórios de uma distribuição Binutils onde podem ser compilados como qualquer outro montador `gas` do pacote. A figura 4.3 mostra o fluxo de produção do montador.

Detalhamos no restante desta seção os aspectos envolvidos na geração dos arquivos dependentes de arquitetura pela ferramenta `acasm`. Informações sobre o uso da ferramenta,

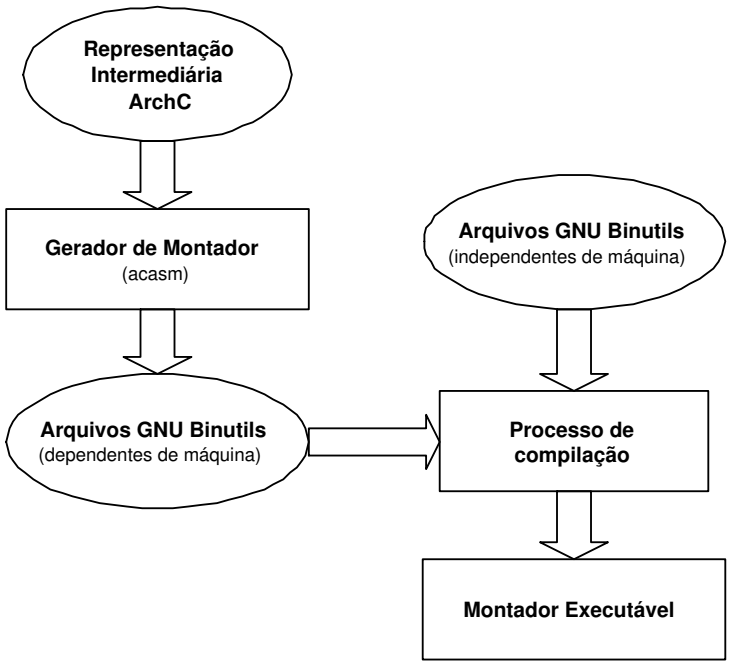

Figura 4.3: Processo de geração automática de um montador executável em ArchC

instalação e construção do montador executável podem ser encontradas no apêndice B.

4.3.1 Restrições

Antes de detalharmos o processo envolvido na geração do montador, é necessário explicitar algumas restrições que essa versão inicial da ferramenta possui. Nosso principal objetivo foi criar uma estrutura base para geração de montadores a qual possa ser escalável de acordo com a produção de novos modelos e a evolução natural da linguagem ArchC. Não tentamos resolver todos os problemas e englobar todas as arquiteturas de uma só vez, pois isso certamente não possibilitaria a conclusão do trabalho em tempo hábil, dado a complicação envolvida em absorver todos os recursos exigidos por todas as arquiteturas em um único passo. Portanto adotamos uma estratégia incremental: garantimos uma estrutura base forte e sólida, a qual pode ser incrementada de acordo com as necessidades experimentais observadas e a evolução da própria linguagem ArchC.

A primeira versão da ferramenta possui as seguintes restrições:

tamanho de palavra máximo:
: o tamanho máximo da palavra para uma dada arquitetura deve ser 32-bit. Para arquiteturas de 8-bit, 16-bit ou 24-bit, `acasm` deve gerar montadores corretos.

instruções de tamanho fixo:
: a arquitetura deve possuir todas instruções de mesmo tamanho;

formato de arquivo objeto:
: um montador não gera um arquivo executável final, mas sim um arquivo com informações relocáveis que pode ser ligado a outros arquivos através de um linkeditor. Como ainda não possuímos um linkeditor, o formato gerado pela ferramenta suprime as informações sobre relocações, ou seja, todas as referências encontradas na descrição de um programa em linguagem de montagem são tratadas e resolvidas localmente (sem referências a símbolos externos). No entanto, isso já é suficiente para que o formato de arquivo objeto gerado pelo montador possa ser executado pelo simulador gerado pela ferramenta `acsim`. O formato de arquivo objeto gerado é o ELF.

4.3.2 Os Arquivos do Pacote GNU Binutils

Introduzimos na seção 2.6 os componentes básicos do pacote GNU Binutils, com destaque para a ferramenta montadora `gas`. Agora iremos mostrar aspectos mais íntimos de funcionamento e redirecionamento desse montador e das bibliotecas de suporte do pacote GNU Binutils. As informações aqui presentes foram extraídas em parte dos manuais internos do `gas` e da biblioteca BFD (os quais podem ser encontrados junto à distribuição do pacote), e em parte de experiências diretas com o pacote.

A árvore de diretórios do pacote, na versão 2.15, é apresentada na figura 4.4. Mostramos apenas aqueles diretórios relativos aos módulos utilizados em nosso trabalho. Descrevemos em seguida o funcionamento e quais arquivos devem ser criados e/ou alterados com base nessa árvore. A seção 4.3.3 descreve como esses arquivos são gerados por `acasm`.

```
- binutils (raiz)
  + bfd          <- biblioteca BFD
  - gas          <- arquivos do core do montador
    . config     <- arquivos dependentes de arquitetura do montador
  - include      <- arquivos gerais de inclusão do pacote
    . opcode     <- arquivos cabeçalhos da biblioteca Opcodes
  + opcodes      <- biblioteca Opcodes
```

Figura 4.4: Parte da árvore de diretórios do pacote GNU Binutils

Biblioteca Opcodes

A biblioteca Opcodes é responsável pela descrição do conjunto de instruções de uma arquitetura e de informações para codificação e decodificação dessas instruções. Essa biblioteca é escrita em C e também é usada por outras ferramentas como o `objdump` e o `gdb`.

Não existe uma estrutura de dados canônica para representação das informações, fazendo com que cada arquitetura adote um modo particular de representação. O módulo dependente de arquitetura do `gas` utiliza essas informações basicamente para reconhecimento da sintaxe e codificação das instruções. A decodificação é feita escrevendo-se um conjunto de chamadas de funções para cada arquitetura.

Para uma dada arquitetura `[arq]`, os arquivos descritos são normalmente nomeados e localizados da seguinte maneira:

`binutils/opcodes/[arq]-opc.c`:
: descreve o conjunto de instruções da arquitetura. Não existe uma forma canônica de representação e algumas arquiteturas, como a i386, sequer descrevem esse arquivo (a versão para a i386 descreve o conjunto de instruções diretamente em `binutils/include/opcode/[arq].h`).

`binutils/opcodes/[arq]-dis.c`:
: contém um conjunto de chamadas de funções que devem ser escritas para a decodificação das instruções da arquitetura. O montador `gas` não faz uso das funções definidas nesse arquivo, que é utilizado principalmente pelas ferramentas `objdump` e `gdb`.

`binutils/include/opcode/[arq].h`:
: contém as definições gerais e macros usadas pelos arquivos da biblioteca Opcodes. Esse arquivo deve ser incluido nos programas que fazem uso das informações arquiteturais dessa biblioteca, como o `gas`.

A biblioteca é compilada e gera o arquivo `libopcodes.a` que deve ser ligado com os programas que utilizam essa biblioteca. A abordagem que escolhemos para geração de nossos arquivos adota a estrutura de arquivos aqui apresentada, com exceção do arquivo `[arq]-dis.c` que não é necessário para o montador.

Biblioteca `BFD`

A biblioteca BFD provê uma interface para leitura e escrita de arquivos objetos de uma forma genérica, ou seja, independente de um formato específico. Aplicações podem utilizar a biblioteca somente para leitura de arquivos objetos (como as ferramentas `gdb` e

`objdump`), somente para escrita (como o `gas`) ou para leitura e escrita (como `objcopy` e o linkeditor `ld`).

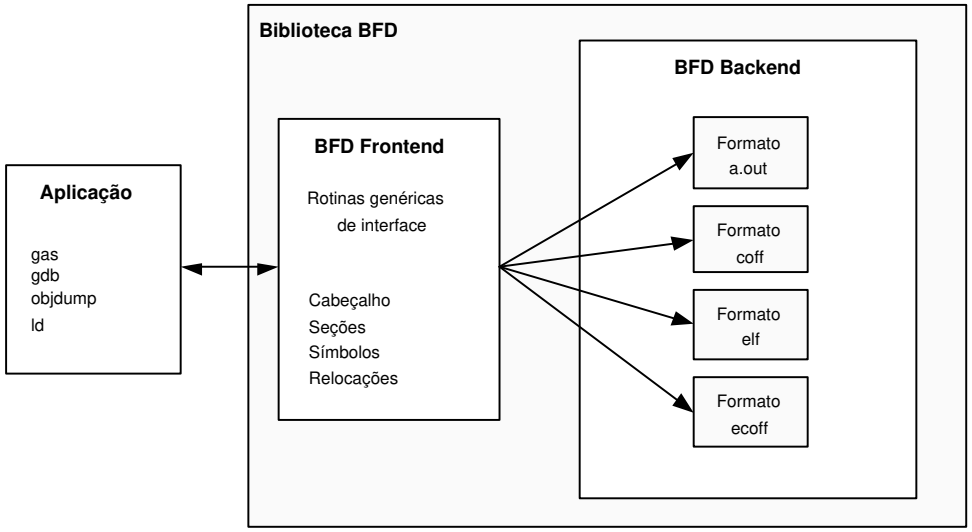

Figura 4.5: Estrutura de interface da biblioteca BFD

A figura 4.5 mostra a estrutura da biblioteca BFD, que pode ser dividida conceitualmente em duas grandes partes:

frontend:
: é a interface da biblioteca com as aplicações. O *frontend* possui estruturas de dados canônicas para representar um arquivo objeto, decide qual formato do *backend* deve ser usado e o momento de executar suas rotinas.

backend:
: implementa formatos específicos de arquivos objetos. Um *backend* para um determinado tipo de arquivo objeto deve fornecer rotinas para transformar suas estruturas de dados em estruturas canônicas utilizadas pelo *frontend*.

Um arquivo objeto é representando pela BFD como uma estrutura interna do tipo `bfd`, a qual funciona como seu *frontend*. Cada arquivo objeto é considerado como sendo composto de:

cabeçalho:
: informações sobre a estruturação do arquivo.

seções:
: a imagem binária de um programa em um arquivo objeto pode estar dividida em

seções, como seções de dados e de código. BFD utiliza uma estrutura de dados chamada `asection` para abstrair seções.

símbolos:
: geralmente cada arquivo objeto possui uma tabela de símbolos associada. Símbolos incluem rótulos de saltos e definições utilizadas em um programa em linguagem de montagem. BFD representa cada símbolo através de uma estrutura chamada `asymbol`, que deve estar associada a uma estrutura `bfd`.

relocações:
: são referências simbólicas que ainda devem ser resolvidas. BFD representa uma relocação através de uma estrutura chamada `arelent`. Relocações são associadas a uma seção `asection` na BFD.

Apesar da BFD proporcionar uma interface genérica para arquivos objetos, a maioria dos formatos possuem informações que não podem ser representadas nela. Nesse caso, para que se evite a perda de dados, é comum a definição de *ganchos* (*hooks*) pelo *backend*. As aplicações podem então utilizar diretamente esses ganchos para acesso às informações específicas do formato. Portanto ganchos não fazem parte da definição do *fronted* da BFD, sendo definidos especificamente para determinados formatos de arquivos objetos.

Geralmente a criação de um *backend* para um novo formato envolve a descrição de dois conjuntos de arquivos: arquivos genéricos de definição do formato, e arquivos específicos da arquitetura. Os formatos ELF e COFF, por exemplo, utilizam essa abordagem. Em geral, para um novo formato `[fmt]` e uma arquitetura alvo `[arq]`, os seguintes arquivos e diretórios devem ser criados:

`binutils/bfd/[fmt].c`:
: embora estejamos aqui representando o novo formato como um único arquivo, isso raramente ocorrerá na prática. A organização dos arquivos é altamente dependente do formato a ser descrito e de como é feito sua integração com a BFD, não havendo portanto um modo padrão imposto. Esse(s) arquivo(s) deve(m) implementar as rotinas de *backend* da BFD e pode(m) também implementar ganchos caso informações não possam ser representadas através da interface genérica.

`binutils/bfd/[fmt]-[arq].c`:
: geralmente é um arquivo só e descreve informações do formato específicas a uma dada arquitetura, como os tipos de relocações e como resolvê-las.

`binutils/include/[fmt]/`:
: esse diretório contém arquivos com declarações de estruturas de dados, definições

e macros utilizados pelos arquivos genéricos do novo formato, ou pelo arquivo específico de arquitetura. Nesse último caso, o arquivo normalmente recebe o nome `[arq].h`.

`binutils/bfd/cpu-[arq].c`:
deve conter informações sobre a família de processadores, expressas através de uma estrutura do tipo `bfd_arch_info_type` e de rotinas para sua manipulação. Essa estrutura contém informações como o tamanho da palavra da arquitetura, número de bits que compõem um byte [1], nome da família e do processador etc.

A introdução de um novo formato exige um considerado esforço em termos de programação e depuração, visto que a biblioteca é muito pouco documentada e não proporciona uma forma padrão para sua especificação. Entretanto, a especificação de uma nova arquitetura para um formato já existente é melhor estruturada e requer somente a descrição de 3 dos arquivos vistos:

```
binutils/bfd/[fmt]-[arq].c
binutils/include/[fmt]/[arq].h
binutils/bfd/cpu-[arq].c
```

A biblioteca BFD é compilada e gera o arquivo `libbfd.a` que deve ser ligado com os programas que a utilizam. Nossa abordagem, para essa primeira versão da ferramenta, gera os 3 arquivos dependentes de arquitetura para o formato ELF, já implementado na BFD e reconhecido pelo simulador gerado por ArchC.

Montador `gas`

`gas` é a ferramenta montadora do pacote GNU Binutils. A versão original deste montador tinha suporte apenas para o formato a.out com 3 seções: `.text`, `.data` e `.bss`. Posteriormente adotou-se o uso da biblioteca BFD para a geração de arquivos objetos, sendo essa a versão mais comum do montador atualmente. Todas as informações contidas nesta seção são relativas a essa versão do montador.

A figura 2.6, apresentada no capítulo 2, mostra a estrutura do montador `gas` e sua interface com as bibliotecas Opcodes e BFD. Funcionalidades gerais, como leitura e *parsing* de arquivos fontes, são fornecidas no *core* do montador. O *core* também define um conjunto de rotinas ganchos que devem ser implementadas pela parte dependente de arquitetura. A biblioteca Opcodes é utilizada pelas rotinas dependentes de arquitetura do

[1] Isso pode soar um pouco estranho para pessoas que sempre associaram 8 bits a um byte. No entanto, o tamanho do byte é definido como sendo o número de bits necessários para representar um caracter em um processador específico, e o grupo de bits daquele tamanho é chamado de byte [21]. O termo usado para designação de um grupo de 8 bits de forma absoluta é **octeto**.

montador para extrair informações da sintaxe e codificação das instruções sendo montadas. Já a biblioteca BFD é utilizada pelo *core* do montador para gerar o arquivo objeto final.

As principais estruturas de dados utilizadas pelo montador são:

`symbolS`:
: representa toda espécie de símbolos encontrados nos arquivos fontes. O montador possui uma estrutura própria e mais flexível para representação de símbolos, que são transformados em símbolos BFD na geração do arquivo objeto final.

`expressionS`:
: todo tipo de expressão encontrada em arquivos fontes é representada por essa estrutura. Um expressão simples tem a forma geral `foo OP bar + AD`, onde `foo` e `bar` são símbolos, `OP` é um operador e `+ AD` representa um possível valor inteiro usado como adendo. Expressões complexas podem ser criadas através da combinação de símbolos representando expressões simples (*expression symbols*) em uma estrutura `expressionS`.

`fixS`:
: estrutura utilizada para representar *fixups*. Um *fixup* representa qualquer coisa que não possa ser resolvida no primeiro passo de montagem. Ao final da montagem, um *fixup* é resolvido ou então se torna uma entrada de relocação no arquivo objeto. Por exemplo, referências simbólicas são representadas como *fixups*, pois seus valores só podem ser determinados ao final da montagem. Caso o símbolo usado na referência não tenha sido definido, o *fixup* não pode ser resolvido e torna-se uma entrada de relocação no arquivo objeto gerado, indicando uma referência externa.

`fragS`:
: representa fragmentos da imagem binária final a ser produzida. Após cada instrução ser codificada, ela é armazenada em uma estrutura desse tipo.

O montador `gas` executa a leitura dos arquivos fontes e constrói o arquivo objeto em um único passo (ou seja, passa somente uma vez pelo arquivo fonte de entrada). Descrevemos em seguida uma versão simplificada do fluxo das operações executadas pelo mesmo. Uma versão mais detalhada com os nomes das rotinas e estruturas utilizadas pode ser encontrada no apêndice C.

1. o montador é inicializado. Esse processo envolve a chamada de várias rotinas para inicialização dos módulos que compõem o montador, como o módulo de leitura de arquivos, BFD e processamento da linha de comando.

2. cada arquivo fonte especificado na linha de comando é aberto para leitura e tem seu conteúdo lido.

3. para cada linha de um arquivo fonte, as seguintes ações podem ser executadas:

   rótulos – ao encontrar um rótulo, o montador cria um símbolo interno para representá-lo. Rótulos geralmente são identificados pela presença do caracter `':'` após seus nomes.

   pseudo-ops – ao encontrar um possível pseudo-op, o montador executa uma busca pelo seu nome em uma tabela *hash* e dispara a rotina associada ao mesmo. Pseudo-ops são identificados pela presença do caracter `'.'` antes de seus nomes. `gas` fornece um conjunto de pseudo-ops padrões para operações comumente encontradas, como divisão de um arquivo fonte em seções (nesse caso, o nome do pseudo-op é `section`). Além disso, permite que a rotina padrão executada por cada pseudo-op possa ser sobreposta por código específico de arquitetura, e também que formatos de arquivos objetos possam definir pseudo-ops específicos.

   instruções – caso não seja encontrado um rótulo ou uma pseudo-op, o montador chama uma função gancho dependente de arquitetura nomeada `md_assemble`. A *string* encontrada no arquivo fonte com a sintaxe da instrução é passada como parâmetro para essa função. O código dependente de arquitetura é responsável pela validação da sintaxe e codificação da instrução, através do uso da biblioteca Opcodes. Códigos específicos de arquitetura ainda podem criar *fixups* para referências simbólicas e devem salvar o valor codificado das instruções em uma estrutura do tipo `fragS`.

4. após todas as linhas dos arquivos fontes serem analisadas, o montador inicia a construção do arquivo objeto final através da biblioteca BFD. Operações nessa fase incluem: atribuição de endereços para os fragmentos (relaxamento), resolução dos símbolos (atribuição de valores), resolução dos *fixups*, criação da tabela de símbolos, criação de entradas de relocação e escrita do conteúdo no arquivo.

5. fecha os arquivos abertos para leitura e encerra os módulos abertos.

O redirecionamento do montador para uma nova arquitetura envolve a criação de um arquivo implementando rotinas ganchos chamadas pelo *core* em determinadas circunstâncias. Para um novo formato de arquivo objeto, ainda é necessário descrever um arquivo com informações relativas ao montador, como pseudo-ops específicas do formato. Os nomes e localizações dos arquivos para uma arquitetura `[arq]` e um formato `[fmt]`, são os seguintes:

`binutils/gas/config/tc-[arq].c|h`:
: arquivos com as implementações das rotinas ganchos chamadas pelo *core* do montador. Essas rotinas normalmente possuem os prefixos `md` (*machine dependent*) ou `tc` (*target cpu*), onde algumas são obrigatórias e outras opcionais. A principal delas é a `md_assemble` que recebe uma *string* com a sintaxe da instrução e deve gerar o código binário correspondente. Uma lista detalhada das principais rotinas e variáveis dependentes de arquitetura pode ser encontrada no apêndice C.

`binutils/gas/config/obj-[fmt].c|h`:
: implementam rotinas do montador específicas para um determinado formato de arquivo objeto. Normalmente são descritas rotinas para tratar de pseudo-ops suportadas pelo formato. Essas rotinas também são chamadas pelo *core* do montador ou então pela parte dependente de arquitetura (raramente), e possuem o prefixo `obj`. O apêndice C mostra uma lista com o nome e uma breve descrição de algumas delas.

Os arquivos dependentes de arquitetura são compilados junto com o *core* do montador e ligados às bibliotecas BFD e Opcodes (e a algumas outras de suporte), gerando o montador executável. Nossa abordagem não precisou criar o arquivo referente ao formato de arquivo, pois o pacote já conta com a versão para ELF.

4.3.3 Arquivos Gerados

A ferramenta `acasm` gera os arquivos dependentes de arquitetura para o pacote GNU Binutils, que quando compilados geram um montador executável para tal arquitetura. O processo pode ser automatizado através da utilização de um *shell script* distribuído junto com a ferramenta. O apêndice B mostra como utilizar a ferramenta e os *scripts* fornecidos.

Apresentamos na seção 4.3.2 a forma como são organizadas as principais bibliotecas, e os arquivos que devem ser gerados quando do redirecionamento do montador. Dado um modelo de uma arquitetura `[arq]` escrito em ArchC, os arquivos gerados pela ferramenta são os apresentados na figura 4.6.

Arquivos da Biblioteca BFD

Os arquivos gerados para a biblioteca BFD contém o mínimo de informação necessária para a geração correta de arquivos objetos ELF relocáveis. Todo o *backend* para esse formato já está implementado na biblioteca BFD, restando portanto a codificação de seções específicas à arquitetura. Como não estamos gerando entradas de relocação no arquivo objeto, o arquivo `elf32-[arq].c` contém o mínimo de informação necessária

```
- binutils
  - bfd
    . cpu-[arq].c
    . elf32-[arq].c
  - opcodes
    . [arq]-opc.c
  - gas
    - config
      . tc-[arq].h
      . tc-funcs.c
      . tc-templ.c  (comum para todas as arquiteturas)
  - include
    - elf
      . [arq].h
    - opcode
      . [arq].h
```

Figura 4.6: Arquivos gerados pela ferramenta `acasm` para uma arquitetura `[arq]`

para que a biblioteca seja compilada. Esse arquivo deve ser espandido em futuras versões da ferramenta, principalmente quando forem gerados linkeditores, já que esses precisam reconhecer e resolver as relocações.

O arquivo `cpu-[arq].c` fornece informações sobre a família de processadores de uma dada arquitetura, não estando diretamente ligado a formatos de arquivos objetos. A geração desse arquivo também é muito simples, bastando preencher uma estrutura de dados do tipo `bfd_arch_info_type` com informações sobre os membros da família, como: tamanho da palavra, número de bits em um byte, nome do processador, etc.

Arquivos da Biblioteca Opcodes

Os arquivos gerados para essa biblioteca são o `[arq]-opc.c` e seu respectivo arquivo cabeçalho `[arq].h`, no diretório `binutils/include/opcode`. Como já discutido, não existe um formato padrão para se descrever as informações dessa biblioteca para a codificação (a decodificação conta com um conjunto pré-definido de rotinas que devem ser escritas), o que nos levou ao desenvolvimento de estruturas de dados que pudessem abranger o maior número possível de arquiteturas.

A abordagem utilizada pela ferramenta `acasm` na geração desses arquivos foi fortemente influenciada pelas versões existentes para as arquiteturas MIPS, SPARC e ARM, e consiste na criação de 3 tabelas:

`Opcodes`:
: cada entrada nessa tabela contém informação específica de uma instrução cuja sintaxe e codificação foi declarada a partir da construção `set_asm` ou `pseudo_instr` em ArchC.

`Símbolos`:
: essa tabela contém o conjunto de todos os mapeamentos descritos a partir da construção `ac_asm_map`.

`Pseudo instruções`:
: uma lista de instruções que devem ser codificadas para uma dada pseudo-instrução.

A tabela de opcodes é descrita pela estrutura `acasm_opcode`, definida como:

```
typedef struct {
  const char *mnemonic;
  const char *args;
  unsigned long image;
  unsigned long format_id;
  unsigned long pseudo_idx;
} acasm_opcode;
```

A seguir descrevemos cada membro dessa estrutura:

`mnemonic`:
: Mnemônico da instrução. O mnemônico é separado da lista de operandos através do primeiro espaço em branco na descrição em `set_asm`. Formatadores utilizados com mnemônicos são expandidos pelo pré-processador antes de serem inseridos na tabela.

`args`:
: Argumentos da instrução. Argumentos incluem formatadores representando operandos e caracteres literais que devem ser encontrados nas sintaxes das instruções em linguagem de montagem. Formatadores possuem uma sintaxe especial nessa *string*:

  `'%'[fmt]':'[fid]':'` – onde `[fmt]` é um formatador definido pelo usuário ou pré-definido; e `[fid]` é um índice para o campo da instrução a que esse operador se refere.

  Essa informação é construída diretamente das informações obtidas em uma construção `set_asm`.

`image`:
: Imagem binária base da instrução. Essa informação é construída com base nas informações definidas para o decoder através de `set_decoder`, representando a parte fixa do código binário. Pseudo instruções possuem um valor padrão 0x00 para este campo.

`format_id`:
: Identificador de formato de instrução. Cada formato de instrução definido a partir de `ac_format` tem um identificador associado pelo pré-processador. Este identificador é utilizado posteriormente na codificação dos campos das instruções pelo montador.

`pseudo_idx`:
: Índice para a lista de pseudo instruções. Este campo aponta para um elemento da lista de pseudo instruções com as instruções nativas relativas à pseudo.

A figura 4.7 mostra parte de uma tabela gerada para um modelo do MIPS-I usando a estrutura `acasm_opcode`.

```
{"add",   "%reg:3:,%reg:1:,%reg:2:",  0x00000020,   0,  0},
{"addi",  "%reg:2:,%reg:1:,%imm:3:",  0x20000000,   1,  0},
{"addiu", "%reg:2:,%reg:1:,%imm:3:",  0x24000000,   1,  0},
{"and",   "%reg:2:,%reg:1:,%imm:3:",  0x30000000,   1,  0},
{"j",     "%expA:1:",                 0x08000000,   2,  0},
{"subu",  "%reg:0:,%reg:0:,%imm:0:",  0x00000000,   99, 10},
{"sw",    "%reg:2:,%exp:3:(%reg:1:)", 0xAC000000,   1,  0}
```

Figura 4.7: Parte de uma tabela de opcodes para o MIPS-I

A tabela de símbolos representa tuplas (símbolo, formatador, valor) gerados a partir da construção `ac_asm_map`. Faixa de valores e atribuições múltiplas são transformadas pelo pré-processador em uma única tupla de saída. Já a tabela de pseudo instruções na verdade é uma lista com as *strings* descrevendo as instruções que compõem uma certa pseudo. Uma sequência de instruções é finalizada pela presença de uma entrada `NULL` na lista. As figuras 4.8 e 4.9 mostram, respectivamente, parte das tabelas de símbolos e pseudo instruções geradas para um modelo MIPS-I.

Arquivos Dependentes de Arquitetura do Montador `gas`

Os arquivos organizados neste módulo possuem a maior quantidade de código de todo o pacote e implementam o montador propriamente dito. As informações necessárias para

```
{"$zero", "reg", 0},
{"$at",   "reg", 1},
{"$v0",   "reg", 2},
{"$v1",   "reg", 3},
{"$a0",   "reg", 4},
{"$a1",   "reg", 5},
{"$t0",   "reg", 8},
{"$t1",   "reg", 9}
```

Figura 4.8: Parte de uma tabela de símbolos para o MIPS-I

```
NULL,
"lui %0,\\%hi(%1)",
"ori %0,%0,%1",
NULL,
"lui $at,\\%hi(%1)",
"lw %0,%1($at)",
NULL,
"addiu %0,%1,-%2",
NULL
```

Figura 4.9: Parte de uma tabela de pseudo instruções para o MIPS-I

o reconhecimento da sintaxe e codificação são extraídas diretamente das estruturas já definidas através da biblioteca Opcodes.

Este módulo é dividido basicamente em dois arquivos: `tc-templ.c` e `tc-funcs.c`. O primeiro implementa as rotinas ganchos dependentes de arquitetura de forma genérica, enquanto o segundo implementa rotinas específicas como funções para reconhecimento de campos e formatos de instruções da arquitetura descrita. Ambos arquivos são combinados pelo *script* de instalação em um único arquivo nomeado `tc-[arq].c`. Aspectos que possam depender diretamente de informações do modelo da arquitetura, como tamanho da palavra e *endian*, são definidos como macros no arquivo cabeçalho `tc-[arq].h`.

As principais rotinas ganchos implementadas em `tc-templ.c` são:

`md_begin`:
: chamada uma única vez no início da execução do montador para inicialização de estruturas internas específicas de arquitetura. Por exemplo, a tabela de opcodes e de símbolos são inseridas em uma tabela *hash* nesse estágio de execução.

`md_assemble`:
: chamada para cada instrução encontrada no arquivo fonte. Deve fazer o reconhecimento da instrução e sua codificação.

`md_apply_fix3`:
: chamada no processo de geração do arquivo objeto para cada *fixup* emitido durante a montagem do código, e depois de todos os símbolos estarem resolvidos. Esta rotina reajusta todo trecho da imagem montada, com os valores corretos dos símbolos referenciados, para a qual um *fixup* foi emitido.

Devido a rotina `md_assemble` representar a maior parte do processamento do montador e sua principal tarefa, vamos apresentar em seguida seu algoritmo de forma informal e descritiva, como implementado em `tc-templ.c`:

```
void md_assemble(char *str)
/* str -> string com a instrução encontrada no código fonte */
```

1. **Busca da instrução na tabela de opcodes**
   A busca é feita isolando-se o mnemônico em `str`, através do primeiro espaço em branco encontrado, e usando-o como chave na tabela *hash* de opcodes. Caso não seja encontrado, uma mensagem de erro é emitida e o montador passa para a próxima instrução.

2. **Validação da sintaxe e codificação dos operandos**
   Após o mnemônico ser extraído de `str`, o restante da *string* é considerado como argumento da instrução, ou, seus operandos. Essa *string* é comparada com a string do campo `args` da estrutura associada ao mnemônico encontrado. Um caracter `'%'` indica que um formatador deve ser reconhecido para aquele trecho da *string*, caso contrário ela deve ser tratada literalmente. Primeiro são reconhecidos os formatadores pré-definidos: `exp`, `addr` e `imm`; e aplicadas funções específicas para resoluções de expressões aritméticas e/ou de referências simbólicas. Caso um tipo pré-definido não seja encontrado, é feita uma busca na tabela de símbolos em Opcodes e seu valor é retornado.

   Para operandos que não façam referências simbólicas, modificadores são aplicados e uma rotina de codificação baseada em campos e formatos de instruções é executada, codificando o valor do operando na imagem binária base da instrução. Operandos que possuam referências simbólicas forçam uma emissão de *fixup* para o fragmento de código atual em 3b. Caso a sintaxe para algum operando seja inválida, o processo retorna ao passo 1, tentando encontrar um outro mnemônico com uma sintaxe de operandos diferente. Se nenhum outro mnemônico for encontrado, uma mensagem de erro de sintaxe é emitida e o montador passa para a próxima instrução.

3. Esse passo pode envolver duas ações diferentes e exclusivas, dependendo se a instrução é uma pseudo instrução ou não.

   (a) **Processamento de pseudo-instruções**
   O campo `pseudo_idx` associado à instrução atual aponta para uma lista de instruções nativas a serem executadas para a respectiva pseudo. Após uma instrução ser buscada nessa lista através do campo chave `pseudo_idx`, seus valores posicionais (%0, %1, ...) são substituídos por valores literais de operandos da instrução corrente, já validados. Em seguida, uma nova chamada a `md_assemble` é feita para cada instrução da lista.

   (b) **Emissão da imagem binária**
   Um novo espaço no fragmento de código atual é alocado e a imagem binária da instrução corrente é salva, de acordo com o *endian* da arquitetura. Caso essa imagem não esteja completamente resolvida, ou seja, ainda dependa de referências simbólicas, um *fixup* é criado relativo ao trecho de código atual. Esse *fixup* será resolvido no fase final de montagem, quando o *core* do montador chama a rotina gancho `md_apply_fix3` para todos os *fixups* ainda não resolvidos.

Para ilustrar as operações realizadas por `md_assemble`, suponha que seu argumento `str` contenha a seguinte *string* representando a instrução a ser processada:

"add $t0, $s1, $sp"

O primeiro passo envolve o reconhecimento e separação do mnemônico de `str`, e sua utilização como chave de busca na tabela *hash* de instruções a partir da biblioteca Opcodes. A figura 4.10 ilustra essa fase do processamento.

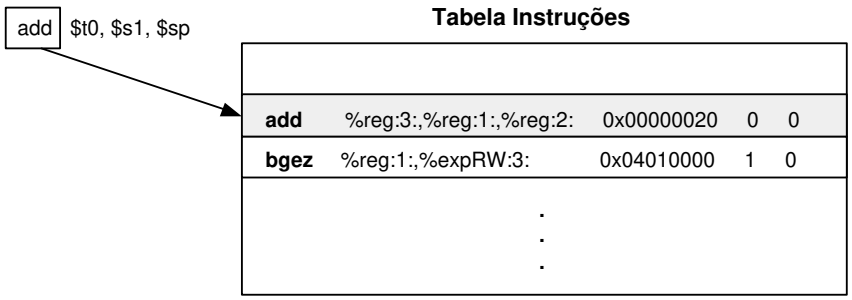

Figura 4.10: Exemplo de busca de uma instrução pelo mnemônico

Neste momento o algoritmo tem acesso às informações de sintaxe dos operandos através do campo `args` retornado pela tabela. A figura 4.11 mostra a separação da *string* composta pelos operandos em *tokens*, a partir da *string* guia `args`. As setas pontilhadas

indicam caracteres que devem ser casados literalmente. No caso de operandos definidos pelo usuário, um *token* é formado percorrendo-se a *string* de operandos até um caracter literal ser encontrado. Nesse exemplo, a *string* `"$t0, $s1, $sp"` é percorrida até que o primeiro caracter literal em `args`, no caso a vírgula, seja encontrado nessa *string*. Dessa forma, o primeiro token reconhecido é `$t0`.

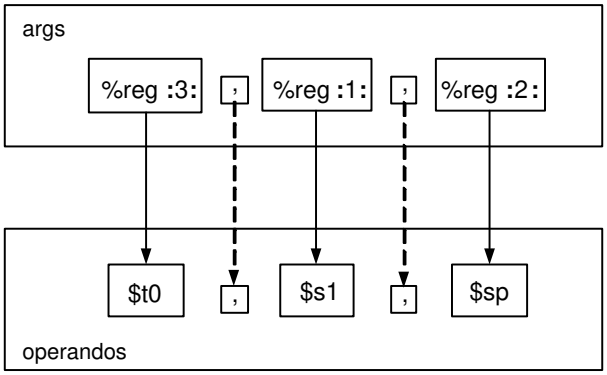

Figura 4.11: Exemplo ilustrando o reconhecimento de operandos

Após um *token* ser reconhecido, seu valor é recuperado através da tabela de símbolos a partir da biblioteca Opcodes. A tabela 4.1 mostra os passos envolvidos na codificação de cada operando. A segunda coluna dessa tabela contém exatamente o valor retornado para cada *token* reconhecido. A seguir, a instrução é codificada utilizando-se o campo (terceira coluna) associado a cada operando em `args`, o formato da instrução (também extraído de `args`) e a imagem binária base associada ao mnemônico (no caso, 0x00000020).

Operando	Valor	C	Ajuste (formato 0)	Imagem
`$t0`	01000 (8)	2	000000 00000 00000 01000 00000 000000	0x0000**4**020
`$s1`	10001 (17)	1	000000 10001 00000 00000 00000 000000	0x0**22**04020
`$sp`	11101 (29)	3	000000 00000 11101 00000 00000 000000	0x02**3D**4020

Tabela 4.1: Codificação dos operandos da instrução `add $t0, $s1, $sp`

A quarta coluna da tabela 4.1 mostra o ajuste efetuado sobre o valor de cada operando. Esse ajuste é feito através do formato da instrução (comum para todos os operandos de uma mesma instrução) e do campo específico de cada operando. Nesse exemplo, a instrução é do tipo 0 (tipo R) e o campo de cada operando é representado na terceira coluna.

A última coluna da tabela 4.1 mostra a imagem binária produzida até aquele ponto da codificação da instrução. A imagem binária base para a instrução `add` é `0x00000020`. Toda vez que um operando é codificado, a imagem binária base é alterada para acomodar

o valor codificado do operando, como pode ser observado na última coluna de cada linha na tabela 4.1. Assim, após a codificação do último operando, o valor binário final da instrução é obtido. Para esse exemplo, o valor é o representado na última coluna da última linha da tabela 4.1, `0x023D4020`.

O restante do processamento aloca um espaço de 32-bit no fragmento de código atual, através de chamadas de rotinas específicas do `gas`, e salva o valor binário obtido nesse espaço. Nesse ponto, `md_assemble` é encerrada, e o *core* do montador passa a processar a linha seguinte do código fonte.

Capítulo 5

Resultados Experimentais

Este capítulo apresenta alguns resultados que obtivemos com montadores gerados a partir de `acasm` para modelos de processadores MIPS-I e SPARC-V8. Descrevemos o critério que adotamos para validação dos arquivos objetos gerados e o processo envolvido nessa validação.

5.1 Metodologia

Para validar a nossa ferramenta de geração de montadores, garantimos que o código objeto gerado está correto, ou seja, todas as instruções descritas em um programa fonte em linguagem de montagem estão codificadas corretamente no arquivo objeto.

Existem basicamente dois métodos que consideramos ao efetuar a verificação do código gerado:

1. comparação binária dos arquivos objetos gerados
   Neste método, o código gerado por um montador deve ser comparado, byte a byte, ao código gerado pelo montador de referência, para um mesmo arquivo fonte. As desvantagens desse método é que ele presume a existência de um montador correto para a arquitetura (o que nem sempre é possível) e que as ações executadas por ambos montadores na geração do código sejam idênticas quando do processamento de instruções sintéticas e diretivas de montagem.

2. execução do código e verificação da saída
   O fato de um mesmo arquivo fonte gerar dois arquivos objetos diferentes não implica necessariamente que um deles está incorreto. Neste método, o arquivo gerado deve ser executado e o resultado da execução deve estar correto para determinadas entradas. Simuladores podem ser usados para a execução do código gerado. A desvantagem desse método é que nem sempre todo o código gerado pode ser executado.

Portanto é possível que determinados trechos sejam gerados incorretamente, mas como não são executados (para determinadas entradas), não é possível a detecção dos eventuais erros. Outra desvantagem é que determinados programas podem levar bastante tempo até o término de sua execução, e ainda assim estarem executando sempre o mesmo trecho de código, o que é irrelevante para nossos propósitos.

Resolvemos adotar uma abordagem híbrida para nossos testes. Para programas pequenos (menores que 1Kb), executamos tanto sua simulação em simuladores gerados pelo próprio ArchC, como a comparação com arquivos gerados pelo montador `gas` nativo para as arquiteturas. Para programas maiores (da ordem de Kb), executamos apenas a comparação entre os arquivos objetos gerados. A justificativa para essa escolha deve-se ao fato de que programas grandes geralmente são construídos em módulos e exigem a linkedição de arquivos e de bibliotecas de sistema, o que ainda não é possível na nossa abordagem.

5.2 Escolha do Conjunto Experimental

Inicialmente testamos os montadores gerados com trechos pequenos de programas escritos manualmente em linguagem de montagem para as respectivas arquiteturas. Esses programas incluem rotinas típicas como cálculo de fatorial, fibonacci, soma de vetores, testes aritméticos e de saltos [1].

Para testes envolvendo arquivos mais complexos (maior quantidade de código, uso de pseudo instruções e diretivas de montagem) a dificuldade é um pouco maior, devido ao fato de programas grandes não serem diretamente escritos em linguagem de montagem. Outra exigência requerida pelo nosso montador é de que todo o código fonte de um determinado programa gere um único arquivo objeto, já que não temos uma ferramenta de linkedição. Como não estamos preocupados em executar esses arquivos, a corretude do programa e a ordem das instruções não importa, desde que o arquivo objeto gerado pelo montador criado através de `acasm` seja idêntico ao gerado pelo montador `gas` nativo.

Resolvemos utilizar arquivos do *benchmark* MiBench, a fim de que terceiros possam replicar e verificar os resultados que obtivemos. Como não temos a opção de compilar módulos completos para cada programa do *benchmark*, escolhemos alguns arquivos isolados com base em seu tamanho (quantidade de código em linguagem C) e por serem representativos de uma determinada área do *benchmark*. Para transformar cada programa da linguagem C para programas em linguagem de montagem do MIPS-I e SPARC-V8, utilizamos o compilador `gcc` de ambas arquiteturas. A próxima seção descreve detalhes sobre o processo envolvido na geração dos resultados.

[1] Esses programas estão disponíveis na página do projeto ArchC: `www.archc.org`

5.3 Geração dos Resultados

O fluxo de geração dos arquivos de entrada e o processo de comparação dos arquivos objetos gerados pelos montadores é apresentado na figura 5.1.

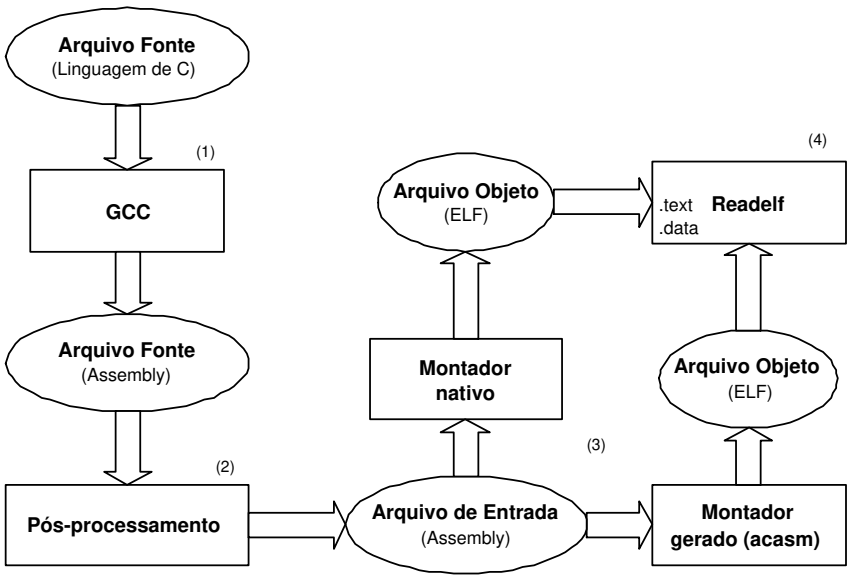

Figura 5.1: Processo de geração e comparação de arquivos em nossos testes

O processo (1) da figura 5.1 faz a tradução de um código escrito em linguagem C para o de linguagem de montagem de um determinado processador. Utilizamos o `gcc` construído para os processadores MIPS-I e SPARC-V8. O processo (2) faz uma filtragem no arquivo gerado pelo `gcc` visando eliminar diretivas de montagem específicas de uma arquitetura, ou sua substituição por uma mais geral. O processo (3) corresponde a execução de ambos montadores e geração dos arquivos objetos correspondentes, que são então comparados em (4). Utilizamos uma outra ferramenta presente no pacote GNU Binutils para verificação das áreas de código (.text) e dado (.data) dos arquivos gerados, chamada `readelf`. A vantagem em se utilizar `readelf` é que ela é independente da biblioteca BFD, ao contrário do `objdump`. Isso permite a verificação dos dados de forma isenta de qualquer erro presente na BFD que possa causar a maquiagem de algum erro.

5.3.1 Resultados

Os arquivos pequenos do nosso conjunto de testes não precisaram passar pelo processo (1), como mostrado na figura 5.1, porque já foram escritos originalmente em linguagem de montagem para os respectivos processadores. Para o MIPS-I foi necessário inserir diretivas

de montagem no arquivo fonte no processo (2), instruindo o montador nativo `gas` a não fazer otimizações como preenchimento de *delay slots* e troca de ordem de instruções.

Executamos também, com sucesso, a simulação desses arquivos no simulador gerado para cada arquitetura através de `acsim`, outra ferramenta ArchC. As seções de código e dados gerados pelos respectivos pares de montadores também foram idênticas. As tabelas 5.1 e 5.2 mostram informações sobre os tamanhos dos arquivos objetos gerados, em bytes, para os processadores MIPS-I e SPARC-V8, respectivamente. Incluímos também os tamanhos para as seções de código e dados dos arquivos.

Arquivo	Tamanho (obj)	.text	.data
add2numbers1.s	474	12	0
add2numbers2.s	520	24	12
storeword.s	569	28	36
swapwords.s	520	24	8
branchjump.s	516	32	0
overflow.s	512	24	4
averagebytes.s	572	60	8
printloop.s	503	20	0
sumofsquares1.s	624	64	33
sumofsquares2.s	664	104	33
proccalls1.s	536	52	0
proccalls2.s	593	68	20
addfirst100.s	633	64	15
factorial.s	854	120	116
factorialrec.s	598	96	0
findmax.s	683	72	36
strlen.s	776	120	36
switch.s	1067	172	63
posneg.s	751	96	40
bubblesort.s	774	100	80
sum2vectors.s	893	80	256

Tabela 5.1: Resultados em termos de tamanho de código objeto gerado para o conjunto interno de testes MIPS-I

Para os arquivos do MiBench, o compilador `gcc` para cada arquitetura foi usado na geração dos arquivos fontes. Utilizamos algumas opções como não gerar otimizações, instruções de ponto flutuante e endereçamentos relativos a `gp` (*global pointer*) para aliviar a sobrecarga da sintaxe das instruções geradas.

A etapa de pós-processamento contou com a eliminação de diretivas de montagem dependentes de arquitetura, diretivas de depuração e de uma instrução sintética condicional

Arquivo	Tamanho (obj)	.text	.data
bitprint.s	604	56	4
div.s	613	64	12
factorial.s	860	140	48
fibonacci.s	1234	228	312
for.s	528	36	0
funccall.s	683	116	0
ifelse.s	586	52	0
loopcounter.s	657	108	0
print.s	548	40	0
repeatst.s	662	72	12
string.s	867	116	76
sumAB.s	1037	148	215
sumarray.s	680	52	84
sumsub.s	647	76	16

Tabela 5.2: Resultados em termos de tamanho de código objeto gerado para o conjunto interno de testes SPARC-V8

MIPS-I, que não pôde ser especificada com a sintaxe `pseudo_instr` atual (a única exceção - a pseudo `li` pode ser expressa como uma ou duas instruções nativas, dependendo do valor imediato ser maior que $2^{16} - 1$). Algumas outras diretivas precisaram ser substituídas, como a `.asciiz` do MIPS-I [2].

As seções de código e dados gerados pelos nossos montadores foram exatamente iguais às geradas pelos montadores `gas` nativos, para todos os arquivos do nosso conjunto de testes MiBench. As tabelas 5.3 e 5.4 mostram informações sobre o tamanho dos arquivos objetos gerados, em bytes, para os processadores MIPS-I e SPARC-V8, respectivamente.

[2]Os arquivos contendo as diretivas utilizadas no pós-processamento podem ser obtidos na página do projeto ArchC: `www.archc.org`

Arquivo		Tamanho (obj)	.text	.data
Automotive	basicmath_large.s	4931	3560	568
	bitcnts.s	2509	960	448
	cubic.s	2694	1856	64
	isqrt.s	826	268	0
	qsort_large.s	2435	1452	120
	qsort_small.s	1488	680	64
	susan.s	67874	64276	1752
Consumer	cjpeg.s	10273	5368	3140
	djpeg.s	10918	6092	3272
	ieeefloat.s	5757	4676	72
	jutils.s	1654	592	320
	lame.s	26648	22544	1032
	tif_codec.s	1626	184	452
	tif_compress.s	4244	2392	484
	tif_getimage.s	64838	59840	1816
Network	dijkstra_large.s	3412	2120	204
	patricia.s	4922	4192	0
Office	bmhasrch.s	2121	1164	256
	bmhsrch.s	1666	992	0
	pbmsrch_large.s	17366	1440	15128
	pbmsrch_small.s	4782	1440	2544
Security	aes.s	47297	25416	21108
	bfspeed.s	2853	1348	488
	bftest.s	9082	5580	2224
	crypto.s	44396	34268	5076
	pgp.s	35834	23176	7288
	random.s	7696	5668	336
	sha.s	3369	2572	32
Telecomm	adpcm.s	2892	1844	420
	crc_32.s	2739	940	1044
	fftmisc.s	1875	996	88
	fourierf.s	3596	2500	136

Tabela 5.3: Resultados em termos de tamanho de código objeto gerado para arquivos do MiBench– MIPS-I

Arquivo		Tamanho (obj)	.text	.data
Automotive	basicmath_large.s	4809	3360	600
	bitcnts.s	2451	832	480
	cubic.s	2946	2068	72
	isqrt.s	814	220	0
	qsort_large.s	2629	1588	136
	qsort_small.s	1566	696	80
	susan.s	62473	58968	1576
Consumer	cjpeg.s	10781	5640	3336
	djpeg.s	11522	6452	3480
	ieeefloat.s	6365	5192	120
	jutils.s	1569	428	320
	lame.s	27895	23572	1112
	prg2lout.s	471827	32176	428448
Network	dijkstra_large.s	3358	1968	232
	patricia.s	4441	3672	0
Office	bmhasrch.s	2088	1088	256
	bmhsrch.s	1632	920	0
	pbmsrch_large.s	17958	1580	15536
	pbmsrch_small.s	4902	1312	2744
Security	aes.s	43971	22048	21112
	bfspeed.s	3411	1832	520
	bftest.s	9847	6212	2320
	crypto.s	47227	36756	5296
	pgp.s	40040	27524	7112
	random.s	7693	5628	336
	sha.s	2923	2092	32
Telecomm	adpcm.s	2696	1612	420
	crc_32.s	2628	784	1048
	fftmisc.s	1769	844	96
	fourierf.s	3499	2344	152

Tabela 5.4: Resultados em termos de tamanho de código objeto gerado para arquivos do MiBench– SPARC-V8

Capítulo 6

Conclusões e Trabalhos Futuros

Apresentamos nesta dissertação o desenvolvimento da ferramenta `acasm` para a geração automática de montadores a partir de modelos de arquiteturas escritos em `ArchC`. `Acasm` gera os arquivos dependentes de arquitetura para redirecionamento do montador `gas`, distribuído junto ao pacote `GNU Binutils`.

Criamos novas construções em `ArchC` que permitem uma descrição mais expressiva da sintaxe da linguagem de montagem e da codificação de operandos. Tais construções surgiram em parte de observações feitas com base em outras `ADL`s e em parte da necessidade imposta pela infra-estrutura do `gas`, embora a dependência entre linguagem e ferramenta seja mínima.

Geramos montadores para modelos de arquiteturas MIPSI e SPARCV8, e, para cada arquitetura, comparamos os arquivos objetos gerados pelos nossos montadores com os gerados pelos montadores `gas` nativos usando arquivos do *benchmark* Mibench. Os resultados apresentaram uma equivalência entre os arquivos objetos, mostrando a expressividade das novas construções `ArchC` e a corretude do montador executável gerado.

As contribuições importantes desta dissertação incluem a documentação das estruturas e organização do montador `gas`, a expansão da linguagem ArchC através da criação de novas construções sintáticas, e uma metodologia para geração automática de montadores com base no pacote `GNU Binutils`.

A maior dificuldade encontrada durante o tempo de desenvolvimento da ferramenta, cerca de 1 ano, foi em relação a documentação e funcionamento das ferramentas `GNU Binutils`, em especial a biblioteca `BFD` e o *core* do montador. Na versão 2.15 do pacote, existem cerca de 700 arquivos e 1.000.000 de linhas de código em linguagem C relacionadas somente ao montador `gas` e às bibliotecas `BFD` e `Opcodes`. Esperamos que as informações sobre o pacote presentes nessa dissertação, principalmente as contidas no apêndice C, sejam úteis em futuros projetos envolvendo nossa ferramenta.

Até o presente momento submetemos um artigo para o SBLP 2005 (*9th Brazilian*

Symposium on Programming Languages), com o nome *Geração Automática de Montadores para Modelos de Arquiteturas Escritos em ArchC* [2], o qual foi aceito. Espera-se que a ferramenta `acasm` esteja em domínio público por volta de Junho/Julho de 2005.

6.1 Trabalhos Futuros

Destacamos em seguida alguns trabalhos que pensamos serem relevantes para um futuro próximo.

Suporte a arquiteturas heterogêneas
: Arquiteturas com instruções de tamanho variável certamente necessitam de suporte adicional na ferramenta `acasm`. Arquiteturas com tamanho de palavra que não sejam múltiplas de 8 (como PICs) também necessitam de algum novo suporte por parte da ferramenta. Aspectos envolvendo outras arquiteturas como a super escalar e `VLIW` necessitam de um estudo mais detalhado, já que tais arquiteturas ainda não possuem suporte na própria linguagem `ArchC`.

Extensões das construções `ArchC` existentes
: Algumas construções `ArchC` podem ser expandidas, visando tornar a descrição mais genérica e expressiva. Por exemplo, a construção `pseudo_instr` poderia suportar a descrição de instruções sintéticas condicionais. Outra idéia interessante seria permitir que usuários construam seus próprios modificadores para operandos. Isso possibilitaria criar operações genéricas de codificação, sendo que todos os modificadores atuais poderiam ser redefinidos explicitamente. Por exemplo, poderíamos definir o modificador atual 'H' da seguinte maneira:

```
ac_define_mod H {
  name = HIGH;
  encoding = value >> field_size;
}
```

: Este tipo de trabalho exigiria a alteração do *parser* da linguagem e dos arquivos gerados por `acasm` referentes à codificação de instruções.

Formato de arquivos objetos
: A ferramenta já tem toda a infra-estrutura para permitir que diversos formatos para arquivos objetos possam ser gerados. Este trabalho deve se concentrar basicamente na geração dos arquivos fontes para a biblioteca `BFD` relativa ao novo formato. Esperamos, para breve, contar com pelo menos o formato `COFF`.

Desmontador

Atualmente `ArchC` fornece todas as informações necessárias para a geração de desmontadores. Algumas opções incluem o redirecionamento da ferramenta `objdump` ou mesmo a construção de uma ferramenta própria com base no gerador de decodificadores `ArchC`, usado pelos simuladores gerados. A vantagem da última abordagem é que o módulo para desmontagem de código binário poderia ser inserido também nos simuladores, e portanto seria possível a visualização das instruções sendo executadas em linguagem de montagem, servindo até como auxílio na detecção de erros.

Redirecionamento do linkeditor `ld`

Este trabalho exige primeiramente a geração de informações sobre relocações por `acasm`, e em seguida o redirecionamento dos arquivos específicos do linkeditor `ld`. Atualmente `acasm` possui um método interno para resolução de relocações baseado em modificadores `ArchC`. É necessário no entanto utilizar a estrutura fornecida pela biblioteca `BFD` para geração dos tipos de relocação, como uso de estruturas de dados e rotinas já definidas. Como o linkeditor `ld` também utiliza a `BFD`, acreditamos que com a geração de relocações genéricas por `acasm`, a tarefa de redirecionamento do linkeditor estará praticamente resolvida. Um cuidado especial a ser tomado é não gerar relocações dependentes de formato de arquivo objeto.

Backend de compilador

Este trabalho pode contribuir na simplificação da construção de *backends* redirecionáveis de compiladores, podendo este gerar código na linguagem de montagem da arquitetura. A ferramenta `acasm` pode então ser utilizada para transformá-lo em código objeto binário.

Apêndice A

Gramática ArchC

Este apêndice descreve formalmente a sintaxe das construções da linguagem ArchC. Para isso usamos a EBNF (uma forma estendida da BNF) e expressões regulares para definições de não-terminais mais custosos, como inteiros ou identificadores. Como na verdade não existe uma única definição para a EBNF, em seguida definimos algumas notações para as sintaxes mais utilizadas.

símbolos terminais:
: são descritos entre aspas simples.

símbolos não-terminais:
: são descritos sem nenhuma alteração na fonte de letra.

separação de símbolos:
: é descrita através de um espaço em branco ou começo de uma nova linha.

agrupamentos:
: símbolos são agrupados ao serem colocados entre os caracteres `'(' e ')'`.

alternativas:
: alternativas podem ser separadas pelo caracter `'|'`.

opcionais:
: as descrições colocadas entre os símbolos `'[' e ']'` são opcionais.

repetições:
: a estrela de Kleene (`'*'`) indica a ocorrência de zero ou mais símbolos; o caractere `'+'` indica a ocorrência de um ou mais símbolos.

Definições Comuns

```
id      := [a-zA-Z][a-zA-Z0-9_]+
int     := [0-9]+
hex     := [0-9A-Fa-f]
str     := "[^"]*"

op_faixa := '[' int '..' int ']'
```

AC_ISA

```
isadec       := 'AC_ISA' '(' id ')' '{' isadecbody '}' ';'
isadecbody   := (formatdec)+ (instrdec)+ (asmmapdec)* ctordec
formatdec    := 'ac_format' id '=' '"' (fmtstr)+ '"' ';'
fmtstr       := '%' id ':' int [':'('s'|'S')]
instrdec     := 'ac_instr' '<' id '>' id (',' id)* ';'
asmmapdec    := 'ac_asm_map' id '{' (lista_map)* ';' '}'
lista_map    := str (',' str)* '=' int          |
                str op_faixa '=' op_faixa       |
                op_faixa str '=' op_faixa       |
                str op_faixa str '=' op_faixa
ctordec      := 'ISA_CTOR' '(' id ')' '{' (ctordecbody)+ '}' ';'
ctordecbody  := asmdec | decoderdec | cyclesdec | rangedec | pseudodec
asmdec       := id '.' 'set_asm' '(' str (args)* ')' ';'
args         := ','(id | id '=' int | id '=' str)
decoderdec   := id '.' 'set_decoder' '(' id '=' hex
                                     (',' id '=' hex)* ')' ';'
cyclesdec    := id '.' 'set_cycles' '(' int ')' ';'
rangedec     := id '.' 'cycle_range' '(' int [, int] ')' ';'
pseu_def     := 'pseudo_instr' '(' str ')' '{' (str ';')+ '}'
```

AC_ARCH

```
archdesc     := 'AC_ARCH' '(' id ')' '{' archdecbody '}' ';'
archdecbody  := ( (storagedec | pipedec)* | [worddec] | [fetchdec] |
                  (formatdec)* (regfmtdec)* )* archctordec
storagedec   := cachedec | regbankdec | memdec | regdec
cachedec     := ('ac_cache' | 'ac_icache' | 'ac_dcache') cacheobjdec
cacheobjdec  := ( id ':' int  unit ';' ) | ( id '(' str ',' int ','
                  int ',' str ',' str [, str] ')' ';' )
regbankdec   := 'ac_regbank' id ':' int ';'
memdec       := 'ac_mem' id ':' int unit ';'
unit         := ['K'|'k'|'M'|'m'|'G'|'g']
regdec       := 'ac_reg' id ';'
pipedec      := 'ac_pipe' id '=' '{' id (',' id)* '}' ';'
worddec      := 'ac_wordsize' int ';'
fetchdec     := 'ac_fetchsize' int ';'
regfmtdec    := 'ac_reg' '<' id '>' id ';'
archctordec :=  'ARCH_CTOR' '(' id ')' '{' archctordecbody '}' ';'
archctordecbody := isafile endian (bind)*
isafile      := 'ac_isa' '(' str ')' ';'
endian       := 'set_endian' '(' str ')' ';'
bind         := id '.' 'binds_to' '(' id ')' ';'
```

Apêndice B

Utilizando `acasm`

Este apêndice fornece informações sobre o processo de geração de um montador executável a partir de um modelo em ArchC para uma dada arquitetura. São apresentados os programas e linhas de comando que devem ser utilizados, e um exemplo completo é mostrado ao final.

Para que a geração do montador executável tenha sucesso, admitimos que o usuário esteja usando um sistema operacional compatível com o `GNU/Linux` e possua os seguintes recursos:

- fonte `GNU Binutils 2.15` (outras versões não foram testadas)
- `GNU Autoconf 2.57` [23] e `GNU Automake 1.7.2` [24]

Note que esses recursos são necessários exclusivamente para a geração do montador. Outras ferramentas ArchC podem requerer outros pacotes, como o `flex` e `bison` para o `acpp`. Consulte a documentação distribuída junto ao pacote ArchC para maiores detalhes.

B.1 Gerando o Montador

A geração do montador basicamente envolve 3 passos, a partir do código fonte ArchC: compilação do pacote ArchC, geração dos programas dependentes de arquitetura para o montador, e construção do montador a partir da árvore `Binutils`. Veremos em detalhes os passos necessários em cada etapa.

B.1.1 Compilando o Pacote ArchC

Uma vez de posse do pacote ArchC [1], descompacte-o em um diretório qualquer e entre nesse diretório. No diretório raiz da distribuição, digite o comando `make` e observe o pacote

[1] O pacote ArchC pode ser obtido pelo endereço eletrônico: `www.archc.org`

ser compilado. Atualmente todas as ferramentas são construídas, portanto o gerador de simuladores também será construído junto com o gerador de montadores.

Caso seja a primeira vez que esteja construindo o pacote, um *script* é executado perguntando sobre a localização de algumas ferramentas, como o compilador e a biblioteca `SystemC` (utilizada na geração de simuladores). Não há nada muito específico sobre o montador aqui. Caso o pacote já esteja instalado e seja necessário a reconstrução das ferramentas, utilize o comando `make clean` antes de um novo `make`.

A árvore de diretórios que o pacote ArchC usa pode ser visualizada na figura B.1. De especial interesse aqui é o diretório `bin`, onde serão construídas as ferramentas geradoras executáveis. É comum colocar esse diretório no caminho (*path*) padrão do *shell*, o que possibilita que as ferramentas possam ser invocadas de qualquer outro diretório.

```
- archc (raiz)
- bin        <- ferramentas executáveis
- config     <- arquivo de configuração
+ examples   <- exemplos distribuídos junto com o pacote
+ include    <- arquivos de inclusão utilizados por ferramentas
                geradas
- lib        <- bibliotecas utilizadas pelas ferramentas geradoras
                e/ou geradas
+ models     <- possível diretório de armazenamento dos arquivos
                de modelos ArchC. O usuário é livre, porém, para
                usar qualquer outro.
- scripts    <- scripts específicos
+ src        <- conjunto de arquivos fontes ArchC
```

Figura B.1: Árvore de diretórios do pacote ArchC

Uma vez o pacote ArchC compilado corretamente podemos passar para o próximo passo, que envolve a geração dos arquivos fontes para o montador.

B.1.2 Gerando os Arquivos Fontes do Montador

A ferramenta geradora dos arquivos fontes para o montador é a `acasm`, e se encontra no diretório `bin` do pacote ArchC após sua compilação. `acasm` gera os arquivos conforme descritos na seção 4.3.3; no entanto, ainda é necessária a realização de outras tarefas como construção de um diretório local, concatenação de alguns arquivos e substituições de nomes em alguns arquivos fontes.

Um *script* de geração chamado `asmgen.sh` é fornecido para automatizar o processo de geração do montador. Esse *script*, devido a sua importância, também está localizado no

diretório `bin` da distribuição ArchC. A função do *script* é criar os diretórios necessários, executar `acasm` e copiar os arquivos fontes gerados para a árvore original de distribuição Binutils, onde podem ser compilados.

A sintaxe de linha de comando para `asmgen.sh` pode ser visualizada através da opção `--help`, e para a atual versão é como mostrada na figura B.2.

```
Usage: asmgen.sh [options] <archc source file> <architecture name>

Create gas dependent source files and copy them to the binutils tree

Options:
  -h, --help            print this help
  -v, --version         print version number

Report bugs and patches to ArchC Team.
```

Figura B.2: Sintaxe e opções de linha de comando de `asmgen.sh`

<`archc source file`> é o arquivo principal do modelo em ArchC, geralmente nomeado com a extensão `.ac`. <`architecture name`> é um nome para arquitetura que será usado na geração dos arquivos fontes. Esse nome é usado como campo `cpu` na tupla de configuração `cpu-fabricante-sistema_operacional`, portanto é importante não utilizar nomes já existentes para evitar erros na geração do montador.

`Asmgen.sh` necessita que duas variáveis de ambiente sejam definidas: `ARCHC_PATH` e `BINUTILS_PATH`. A primeira variável também é necessária para o gerador de simuladores e deve apontar para o diretório raiz de instalação do ArchC; já a segunda deve apontar para a raiz de onde foi descompactado o pacote GNU Binutils.

As principais tarefas realizadas por `asmgen.sh` incluem:

1. Criação de um diretório local com uma sub-árvore do Binutils.
2. Execução de `acasm` para geração dos arquivos no diretório recém-criado.
3. Criação do arquivo `tc-[arq].c` a partir dos arquivos `tc-templ.c` e `tc-funcs.c`.
4. *Patching* de alguns arquivos de configuração na árvore Binutils conforme apontando por `BINUTILS_PATH`. Esse arquivos incluem principalmente os utilizados pelas ferramentas `Autoconf` e `Automake`: `configure.in` e `Makefile.am`.
5. Cópia dos arquivos gerados pela ferramenta `acasm` para a árvore Binutils.

Após `asmgen.sh` ter sido executado corretamente, os arquivos fontes para o montador já estarão corretamente distribuídos pela árvore Binutils e podem ser compilados utilizando-se o mesmo procedimento usado para qualquer outro montador do pacote.

B.1.3 Gerando o Montador Executável

Este último passo deve ser familiar para aqueles já acostumados com o uso das ferramentas GNU `Autoconf` e `Automake`. Não entraremos em todos os detalhes de uso e de opções fornecidas por essas ferramentas, já que muitas não são necessárias na geração do montador. Para maiores detalhes é recomendável a leitura de [42].

De modo geral, a seguinte sequência de passos deve ser realizada para a geração do montador executável (geralmente a partir de um diretório diferente de `BINUTILS_PATH`):

1. Execução do *script* `configure` encontrado no diretório raiz do pacote Binutils. É necessário informar a arquitetura alvo para o montador através da opção `--target=[arq]-elf`, onde [arq] é o nome dado para a arquitetura na execução do *script* `asmgen.sh`. Também é comum o uso da opção `--prefix` para informar o diretório aonde serão copiados os arquivos gerados.

2. Execução do comando `make all-gas`. Note que a execução somente do comando `make` iria provocar a construção de todo o pacote GNU Binutils, o que provavelmente ocasionaria um erro já que outras ferramentas desse pacote não foram redirecionadas.

3. Execução do comando `make install-gas` para copiar os arquivos compilados para o diretório de instalação fornecido através da opção `--prefix` em `configure`.

Após a finalização desses passos, o montador executável deve ter sido gerado e pode ser encontrado sob o diretório fornecido junto com o comando `configure`.

B.2 Um Exemplo Detalhado

Mostramos nessa seção um exemplo detalhado da construção de um montador para um modelo MIPS-I utilizando a ferramenta geradora de montadores ArchC.

Assumimos que o *shell* seja compatível com o *Bourne Again shell* (`bash`) e que os seguintes diretórios estejam criados:

`/archc`
: local aonde está instalada a distribuição ArchC.

`/models/mips1`
: local aonde estão localizados os arquivos com o modelo em ArchC para a arquitetura MIPS-I.

`/install`
: local aonde será instalado o montador executável gerado.

`/build`
: diretório temporário usado para construção do montador. Pode ser apagado após a geração.

`/binutils-2.15`
: local aonde está instalada a distribuição GNU Binutils. Note que serão aplicadas modificações nos arquivos dessa árvore; caso seja de importância a sua preservação, recomendamos que uma cópia do diretório seja criada somente para a instalação.

Algumas variáveis de ambiente necessitam ser definidas, enquanto outras facilitam a tarefa de execução de alguns comandos. Antes de iniciar com a instalação, sugerimos que as seguintes variáveis sejam definidas:

```
export ARCHC_PATH=/archc              (obrigatório)
export BINUTILS_PATH=/binutils-2.15   (obrigatório)
export PATH=$ARCHC_PATH/bin:$PATH
export TARGET_NAME=mips1
```

As duas primeiras variáveis, `ARCHC_PATH` e `BINUTILS_PATH`, precisam obrigatoriamente estar definidas antes da execução de `asmgen.sh`. É interessante colocar no caminho padrão (`PATH`) o diretório das ferramentas executáveis de ArchC, o que é feito no terceiro comando apresentado. Já `TARGET_NAME` é o nome da cpu na qual o montador gerado será nomeado. Esse nome é usado na nomeação dos arquivos gerados para o montador e nos arquivos de configuração do GNU Binutils, portanto é importante não colocar um nome já existente.

Importante!
: Sistemas baseados no `GNU/Linux` utilizam um esquema para nomear configurações composta da tripla: `cpu-fabricante-sistema_operacional`. O campo `cpu` dessa tripla é exatamente o nome que é usado por `acasm` para nomeação dos arquivos gerados e alteração dos arquivos de configuração do Binutils. Caso um nome já existente seja usado (e que não tenha sido criado por usuários do `acasm`) a instalação muito provavelmente não terá êxito. É importante portanto saber que uma dada `cpu` ainda não exista. Uma dica para saber se um dado nome de configuração já existe, é executar o comando `config.sub` presente no diretório raiz do Binutils:

```
$BINUTILS_PATH/config.sub $TARGET_NAME-elf
```

Caso a saída do comando reporte que o nome `TARGET_NAME` para o formato `elf` não exista, ele pode ser usado tranquilamente no processo de instalação.

O primeiro passo é compilar o pacote com a distribuição ArchC. Caso ainda não o tenha feito, entre no diretório raiz da distribuição e execute o comando `make`.

Uma vez compilado o pacote ArchC, o *script* `asmgen.sh` pode ser executado para geração dos arquivos fontes do montador. Para isso, entre no diretório do modelo e execute o *script*, conforme mostrado a seguir:

```
cd /models/mips1
asmgen.sh mips1.ac $TARGET_NAME
```

Nesse exemplo, `mips1.ac` é o arquivo principal do modelo ArchC e usamos `mips1` como nome de arquitetura através da variável `TARGET_NAME`. O *script* `asmgen.sh` executa uma série de operações que envolve a chamada de `acasm` para geração dos arquivos e o *patching* na árvore fonte Binutils. Após a execução desse comando, será criado um subdiretório chamado `binutils` no diretório `/model/mips1` com os arquivos gerados. Esse subdiretório pode ser apagado caso se deseje.

A partir desse ponto, a instalação se torna idêntica a qualquer uma outra instalação de um montador do pacote GNU Binutils. Esse processo envolve o uso do *script* de configuração `configure` localizado no diretório raiz do Binutils. A seguinte sequência de comandos deve instalar o montador no diretório `/install`:

```
cd /build
$BINUTILS_PATH/configure --prefix=/install --target=$TARGET_NAME-elf
make all-gas
make install-gas
```

Importante!

Sempre use os comandos `make all-gas` e `make install-gas`. A utilização de `make all` ou `make install` pode apresentar erros de compilação e causar o cancelamento da instalação, pelo fato de que nem todas as ferramentas presentes no pacote GNU Binutils terem sido redirecionadas para a nova arquitetura.

Para confirmar o sucesso da instalação, tente executar o montador com a opção `--archc`.

Apêndice C

Organização do Montador gas

Este apêndice apresenta aspectos internos do montador `gas`, como os arquivos que formam o *core*, o fluxo de execução principal, e as rotinas e variáveis que devem ser definidas para a arquitetura alvo. As informações obtidas e disponibilizadas aqui foram extraídas de experiências com o próprio pacote GNU Binutils, e visam servir de referência para aqueles que tenham de alterar o pacote ou estejam iniciando seu trabalho com o mesmo. É importante frisar ainda que essas informações são referentes à versão 2.15 do pacote GNU Binutils, e se refere a versão do montador que utiliza a biblioteca BFD.

C.1 Os Arquivos

Nesta seção apresentamos os nomes e uma breve descrição dos principais arquivos que compõe o *core* do montador. Esses arquivos são encontrados no diretório raiz do `gas`.

`as.c|h`
: Este é o arquivo principal do montador, e o local aonde a rotina inicial (`main`) se encontra. As principais rotinas presentes nesse módulo são as que tratam da inicialização do montador (como reconhecimento da linha de comando) e aquelas que controlam o seu fluxo principal de execução, interfaceando principalmente com os módulos `read.c` e `write.c`.

`read.c|h`
: Este módulo trata da leitura de arquivos fontes e do reconhecimento de elementos presentes nesses arquivos, como pseudo-ops, rótulos e instruções. A leitura física dos arquivos é feita através de uma interface com o módulo de pré-processamento, constituído pelos arquivos `app.c` e `input-scrub.c`. A rotina `read_a_source_file` é responsável pelo controle do fluxo principal de leitura de cada linha de um ar-

quivo fonte, reconhecendo pseudo-ops, rótulos e executando chamadas das rotinas dependentes de arquitetura, como a `md_assemble`.

`write.c|h`
: Este módulo é responsável pelas operações de criação do arquivo objeto final, tais como a construção da tabela de símbolos e entradas de relocação, através da biblioteca BFD. A estrutura e operações sobre *fixups* também são definidas nesse arquivo.

`expr.c|h`
: Este módulo define a estrutura utilizada para expressões aritméticas (`expressionS`) e rotinas associadas.

`frags.c|h`
: Este módulo define a estrutura utilizada para armazenar trechos de código, conhecida como frag (`fragS`). O montador trabalha sempre com um ponteiro para o frag atual, através da variável externa `frag_now`, definida nesse módulo. Sempre que uma nova instrução deve ser emitida, uma certa quantidade de bytes deve ser requisitada para o frag através da rotina `frag_more`.

`subsegs.c|h`
: Este módulo define uma estrutura de dados chamada de `frchainS` usada para representar segmentos de código e/ou dados. Essa estrutura é uma lista ligada de frags (estrutura `fragS` definida em `frags.h`) e pode conter também uma lista de *fixups* emitidos para o respectivo segmento. O montador trabalha com a variável global `frchain_now` para representar a estrutura relativa ao segmento atual de montagem.

`symbols.c|h`
: Este módulo define a estrutura `symbolS` utilizada pelo montador para representar símbolos (na verdade a definição da estrutura se dá no arquivo `struc-symbol.h`, para isolar o tipo de dado de suas operações). Essa estrutura contém, entre outras informações, um ponteiro para o símbolo `BFD`, o valor do símbolo (representado como sendo do tipo `expressionS`) e o frag ao qual o símbolo se refere.

A tabela C.1 apresenta as principais rotinas definidas nos módulos que acabamos de descrever.

Modulo	Rotina

as	`main()`
	`perform_an_assembly_pass()`
	`parse_args()`

read	`read_begin()` `read_a_source_file()`

write	`fix_new()`
	`fix_new_exp()`
	`write_object_file()`
	`chain_frchains_together() - (interna)`
	`relax_segment()`
	`subsegs_finish()`
	`adjust_reloc_syms() - (interna)`
	`fixup_segment() - (interna)`
	`set_symtab() - (interna)`
	`write_relocs() - (interna)`
	`write_contents() - (interna)`

expr	`expr_begin()`
	`expr()`
	`operand() - (interna)`
	`get_single_number()`

frags	`frag_init()`
	`frag_new()`
	`frag_more()`

subsegs	`subsegs_begin()`
	`subseg_new()`
	`subseg_set()`
	`subseg_get()`

symbols	`symbol_begin()`
	`symbol_find()`
	`symbol_create()`
	`colon()`
	`S_SET_VALUE()`
	`S_GET_VALUE()`

Tabela C.1: Principais rotinas e módulos presentes na estrutura `gas`

C.2 Fluxo de Execução

O fluxo de execução apresentado nesse seção visa servir de referência rápida, e foi desenvolvido e utilizado durante a implementação de nossa abordagem para geração automática de montadores. Devido a sua complexidade não é possível apresentar todos os detalhes envolvidos na execução do montador, por isso mostramos apenas os pontos principais.

Começamos pelo fluxo inicial através da rotina `main`; funções que julgamos importantes são indicadas através de `[+]` e expandidas mais tarde.

main() [as.c]

```
1. Inicialização:
  1.1  bfd_init()
  1.2  symbol_begin()
  1.3  frag_init()
  1.4  subsegs_begin()
  1.5  parse_args()
  1.6  read_begin()
  1.7  input_scrub_begin()
  1.8  expr_begin()
  1.9  macro_init()
  1.10 output_file_create()
  1.11 tc_init_after_args() - (se definido)
2. Execução:
  2.1 perform_an_assembly_pass() - [+]
3. Finalização:
  3.1 md_end() - (se definido)
  3.2 subsegs_finish()
  3.3 write_object_file() - [+]
  3.4 output_file_close()
```

perform_an_assembly_pass() [as.c]

```
1. Criação de seções:
  1.1 subseg_new(TEXT_SECTION_NAME, 0)
  1.2 subseg_new(DATA_SECTION_NAME, 0)
  1.3 subseg_new(BSS_SECTION_NAME, 0)
  1.4 subseg_new(GAS_REG_SECTION, 0)
  1.5 subseg_new(GAS_EXPR_SECTION, 0)
2. obj_begin() -- (se definido)
3. Para cada arquivo fonte especificado faça
  3.1 read_a_source_file() - [+]
```

read_a_source_file()[1] **[read.c]**

```
1. input_scrub_new_file()
2. enquanto não for fim de arquivo faça
  2.1 se nova linha
    2.1.1 md_start_line_hook() - (se definido)
  2.2 se primeiro caracter for início de símbolo
    2.2.1 get_symbol_end()
    2.2.2 se ultimo caracter for ':' (ou seja, um label)
      2.2.2.1 colon()
      2.2.2.2 tc_check_label() - (se definido)
    2.2.3 senão
      2.2.3.1 se primeiro caracter for '.' (ou seja, pseudo-op)
        2.2.3.1.1 hash_find(po_hash, ...)
        2.2.3.1.2 pop->poc_handler()
      2.2.3.2 senão  (macro ou instrução)
        2.2.3.2.1 se existir macro com esse nome, executa-a
        2.2.3.2.2 senão executa md_assemble()
  2.3 senão
    2.3.1 se símbolo local
      2.3.2 tratamento de símbolos locais
      2.3.3 continua (volta para laço enquanto)
    2.3.2 tc_unrecognized_line() - (se definido)
3. md_cleanup() -- (se definido)
4. input_scrub_close()
```

write_object_file() [write.c]

```
1.  remove seções específicas do gas (reg e expr)
2.  renumber_sections()
3.  chain_frchains_together()
4.  relax_segment()
5.  size_seg()
6.  md_post_relax_hook() - (se definido)
7.  para cada símbolo faça
  7.1 resolve_symbol_value()
8.  tc_frob_file_before_adjust() - (se definido)
9.  obj_frob_file_before_adjust() - (se definido)
10. adjust_reloc_syms()
11. tc_frob_file_before_fix() - (se definido)
```

[1] A versão apresentada aqui é bem simplificada. A original utiliza entrada bufferizada, o que complicaria muito a descrição do fluxo.

```
12. obj_frob_file_before_fix() - (se definido)
13. fixup_segment()
14. criação da tabela de símbolos
15. tc_adjust_symtab() - (se definido)
16. obj_adjust_symtab() - (se definido)
17. set_symtab()
18. tc_frob_file() - (se definido)
19. obj_frob_file() - (se definido)
20. write_relocs()
21. tc_frob_file_after_relocs() - (se definido)
22. obj_frob_file_after_relocs() - (se definido)
23. write_contents()
```

C.3 Variáveis e Rotinas Dependentes de Arquitetura

Nesta seção apresentamos algumas variáveis, definições e rotinas dependentes de arquitetura. Um redirecionamento para uma nova arquitetura necessita que algumas sejam necessariamente definidas, enquanto outras são opcionais. Existem também algumas definições que não são especificamente dependentes de arquitetura, mas que precisam ser conhecidas para um bom redirecionamento. Para cada elemento, apresentamos os módulos a que pertencem, os valores padrões (se aplicável) e uma breve descrição.

C.3.1 Definições

Definições utilizam a diretiva `#define` do pré-processador C. Algumas precisam de algum valor associado, enquanto outras só necessitam ser definidas (geralmente são usadas para indicar trechos de código que devem ser compilados, através do par `#ifdef`/`#endif`).

EMIT_SECTION_SYMBOLS (obrigatório)
Módulos: subsegs, write, symbols
Valor padrão: 1
Se 1, indica que símbolos de seção serão gravados no arquivo objeto final.

LABELS_WITHOUT_COLONS (obrigatório)
Módulos: app, read
Valor padrão: 0
Se 1, indica que o caracter de dois pontos (`':'`) não faz parte do nome usado em rótulos. O padrão é que os dois pontos sejam usados para reconhecimento de rótulos.

LOCAL_LABELS_DOLLAR (obrigatório)
Módulos: expr, read, symbols
Valor padrão: 0
Se 1, indica que o caracter `'$'` é usado para indicar rótulos locais.

LOCAL_LABELS_FB (obrigatório)
Módulos: expr, read, symbols
Valor padrão: 0
Se 1, o montador reconhece o esquema `forwarding/backwarding` para rótulos locais.

NO_PSEUDO_DOT (obrigatório)
Módulos: cond, read
Valor padrão: 0
Se 0, indica que o caracter `'.'` é usado para indicar pseudo-ops.

NOP_OPCODE (obrigatório)
Módulo: frags
Valor padrão: 0
O valor definido é usado para emissão de instruções `NOP`, como byte.

OBJ_SYMFIELD_TYPE
Módulo: symbol
Valor padrão: –
Código dependente de formato de arquivo objeto pode usar essa definição para inserir um tipo de dado específico para seu uso na estrutura `symbolS`.

QUOTES_IN_INSN
Módulo: read
Valor padrão: –
Se definido, instruções podem conter aspas (¨).

RELOC_EXPANSION_POSSIBLE
Módulo: write
Valor padrão: –
Se definido, o valor indica um valor de expansão usado na geração de relocações em tc_gen_reloc().

STRIP_UNDERSCORE
Módulo: symbol
Valor padrão: –
Se definido, o caracter de *underscore* (_) é retirado automaticamente de nomes de símbolos.

TARGET_ARCH (obrigatório)
Módulo: output-file
Valor padrão: deve ser definido pela arquitetura alvo
Deve ser igualado ao tipo definido no tipo enumerado `bfd_architecture` para a arquitetura alvo.

TARGET_FORMAT (obrigatório)
Módulo: output-file
Valor padrão: deve ser definido pela arquitetura alvo
Deve ser igualado à *string* de formato alvo (por exemplo, "elf32-archc").

TARGET_MACH
Módulo: output-file
Valor padrão: se não definido, 0
Número de identificação de membros de família de processadores `TARGET_ARCH`.

TARGET_BYTES_BIG_ENDIAN (obrigatório)
Módulo: read
Valor padrão: configurado através de `configure.in`
Se 0 a arquitetura é considerada como *little endian*; 1 para *big endian*.

TC_CASE_SENSITIVE
Módulo: read
Valor padrão: –
Se definido, o montador diferencia letras maiúsculas de minúsculas em instruções e pseudo-ops.

tc_comment_chars
Módulo: app, read
Valor padrão: se não definido, é igualado a `comment_chars`
Caracteres utilizados para comentário em arquivos fontes.

TC_FINALIZE_SYMS_BEFORE_SIZE_SEG (obrigatório)
Módulo: write
Valor padrão: 1
Se 1, indica que símbolos são resolvidos antes da chamada a size_seg().

TC_FIX_TYPE
Módulo: write
Valor padrão: –
Código dependente de arquitetura alvo pode usar essa definição para inserir um tipo de dado específico para seu uso na estrutura `fixS`.

TC_FRAG_TYPE
Módulo: frags
Valor padrão: –
Código dependente de arquitetura alvo pode usar essa definição para inserir um tipo de dado específico para seu uso na estrutura `fragS`.

TC_GENERIC_RELAX_TABLE
Módulo: write
Valor padrão: –
Se definido, um método alternativo a `md_relax_frag()` é utilizado. Esse método usa `relax_frag()` e só é aplicado em frags do tipo `rs_machine_dependent`.

TC_SEGMENT_INFO_TYPE
Módulo: subsegs
Valor padrão: –
Código dependente de arquitetura alvo pode usar essa definição para inserir um tipo de dado específico para seu uso na estrutura `segment_info_struct`.

TC_SYMFIELD_TYPE
Módulo: symbol
Valor padrão: –
Código dependente de arquitetura alvo pode usar essa definição para inserir um tipo de dado específico para seu uso na estrutura `symbolS`.

WORKING_DOT_WORD
Módulos: as, read, symbols, write
Valor padrão: –
Se definido, o montador não realizará o processamento de *quebra* em diretivas `.word`. Esse processamento adicional só é necessário se `.word` não emite um valor grande suficiente para conter qualquer diferença entre dois endereços.

C.3.2 Variáveis

Variáveis devem ser todas definidas, caso contrário erros na compilação serão gerados.

const char comment_chars[]
: Módulos: app, read
Valor padrão: deve ser especificado pela arquitetura alvo
Caracteres utilizados para comentário em arquivos fontes. Se `tc_comment_chars` for definido, tem privilégio sobre essa variável.

const char EXP_CHARS[]
: Módulo: expr
Valor padrão: deve ser especificado pela arquitetura alvo
Caracteres usados como base de exponenciação para números.

const char FLT_CHARS[]
: Módulo: expr
Valor padrão: deve ser especificado pela arquitetura alvo
Caracteres válidos para números expressos em ponto flutuante.

const char line_comment_chars[]
: Módulos: app, read
Valor padrão: deve ser especificado pela arquitetura alvo
Caracteres utilizados para comentários de linha toda em código fonte.

const char line_separator_chars[]
: Módulos: app, read
Valor padrão: deve ser especificado pela arquitetura alvo
Caracteres utilizados para indicar a separação de linhas no código fonte.

struct option md_longopts[]
: Módulo: as
Valor padrão: deve ser especificado pela arquitetura alvo
Opções longas de linha de comando específicas da arquitetura alvo.

const pseudo_typeS md_pseudo_table[]
: Módulo: read
Valor padrão: deve ser especificado pela arquitetura alvo
Permite especificar pseudo-ops específicos para a arquitetura alvo.

const char *md_shortopts
Módulo: as
Valor padrão: deve ser especificado pela arquitetura alvo
Opções curtas de linha de comando específicas da arquitetura alvo.

const pseudo_typeS obj_pseudo_table[]
Módulo: read
Valor padrão: deve ser especificado pela formato de arquivo objeto alvo
Permite especificar pseudo-ops específicos para um determinado formato de arquivo objeto.

int symbols_case_sensitive
Módulo: symbol
Valor padrão: 1
Se 1, o montador diferencia entre letras maiúsculas e minúsculas em nomes de símbolos.

C.3.3 Rotinas

Existe um grande número de rotinas ganchos, porém nem todas necessitam ser implementadas. Não apresentamos descrições para as rotinas específicas de formato de arquivo objeto (aquelas com prefixo `obj`), mas muitas são equivalentes à sua versão dependente de arquitetura (prefixos `md` ou `tc`).

void md_after_parse_args()
Módulo: as
Possibilita que código dependente de arquitetura possa realizar algum processamento logo após os argumentos de linha de comando serem reconhecidos.

void md_after_pass_hook()
Módulo: read
Possibilita que código dependente de arquitetura possa realizar algum processamento após um passo de montagem do arquivo fonte.

void md_apply_fix3(fixS *, valueT *, segT) (obrigatório)
Módulo: write
Chamado para cada *fixup* emitido durante a montagem. O código escrito nessa rotina deve resolver o *fixup* quando possível e/ou preparar uma entrada de relocação.

void md_assemble(char *str) (obrigatório)

Módulo: read

Principal rotina a ser implementada. `str` aponta para a *string* contendo a instrução a ser reconhecida e codificada.

char *md_atof(int, char *, int *) (obrigatório)

Módulo: read

Rotina para conversão de uma *string* em um tipo ponto flutuante. Em caso de sucesso `NULL` deve ser retornado, ou então uma mensagem de erro.

void md_begin() (obrigatório)

Módulo: as

Rotina chamada antes de `read_a_source_file()`, para iniciar estruturas e estados internos do código dependente de arquitetura.

void md_cleanup()

Módulo: read

Pode ser chamada, caso definida, logo após um arquivo fonte inteiro ter sido lido (mas ainda não processada a sua saída).

void md_convert_frag(bfd *, segT, fragS *) (obrigatório)

Módulo: write

Chamado para todo frag do tipo `rs_machine_dependent`. Algumas arquiteturas podem emitir frags variantes que devem ser resolvidos antes da emissão do código objeto. É função dessa rotina resolver aspectos relacionados a resolução de instruções para frags dependentes de máquina.

void md_end()

Módulo: as

Chamado logo após `perform_an_assembly_pass()` e antes de `write_object_file()`. Código dependente de arquitetura deve realizar tarefas opostas a `md_begin()`, como desalocação de estruturas internas que não serão mais usadas.

int md_estimate_size_before_relax(fragS *, segT) (obrigatório)

Módulo: write

Chamado na rotina de relaxamento de segmentos para frags dependentes de máquina (`rs_machine_dependent`). A rotina deve retornar uma estimativa para o tamanho final desse frag.

void md_number_to_chars(char *, valueT, int) (obrigatório)

Módulo: read, write

Usado para emitir palavras em fragmentos, independente do *endianess* da arquitetura.

void md_operand(expressionS * expressionP) (obrigatório)

Módulo: expr

Essa rotina é chamada para todas as expressões que o *core* do montador não conseguiu resolver.

int md_parse_option(int c, char *arg) (obrigatório)

Módulo: as

Chamada para reconhecer e tratar opções de linha de comando específicas da arquitetura alvo.

int md_parse_name(char *, expressionS *, char)

Módulo: expr

Essa rotina pode ser chamada por `operand()` para que o código dependente de arquitetura possa reconhecer certos nomes que possam ser úteis em certos contextos.

long md_pcrel_from(fixS * fixP) (obrigatório)

Módulo: write

Essa rotina é chamada para *fixups* relativos ao `PC`, e deve retonar uma valor que será somado ao valor do *fixup*.

void md_pop_insert(pseudo_typeS *)

Módulo: read

Usado para adicionar pseudo-ops específicos de arquitetura alvo.

void md_post_relax_hook()

Módulo: write

Pode ser chamado logo após os segmentos serem relaxados.

int md_relax_frag(asection *, fragS, long)

Módulo: write

Pode ser chamado por `relax_segment()` para frags do tipo `rs_machine_dependent`, para que o código específico de arquitetura possa relaxar o fragmento. Caso contrário, `gas` fornece um método padrão para relaxamento, baseado em tabela. Esse modo é processado somente quando `TC_GENERIC_RELAX_TABLE` estiver definido.

valueT md_section_align(segT seg, valueT size) (obrigatório)

Módulo: write

Essa rotina é chamada em `size_seg()` e permite que o código dependente de arquitetura altere o tamanho de segmentos (devido a alinhamentos).

void md_show_usage(FILE *) (obrigatório)

Módulo: as

Essa rotina é chamada para apresentação das opções de linha de comando dependentes de arquitetura.

void md_start_line_hook()

Módulo: read

Pode ser chamada no ínicio de cada linha do código fonte caso o código dependente de arquitetura necessite de algum processamento especial nessas circunstâncias.

symbolS *md_undefined_symbol(char *name)

Módulos: read, symbol

Sempre que um símbolo não definido é encontrado, o *core* do `gas` chama essa função antes de criar um novo símbolo e armazená-lo na seção `undefined`.

int RESOLVE_SYMBOL_REDEFINITION(symbolS *)

Módulo: symbol

`gas` chama essa rotina sempre que um símbolo já definido for re-encontrado no arquivo fonte.

void SET_SECTION_RELOCS(asection *, arelent **, unsigned)

Módulo: write

Última rotina a ser chamada por `write_relocs()`, logo após as relocações terem sido inseridas na seção `BFD`.

void tc_adjust_symtab()

Módulo: write

Chamada antes da escrita da tabela de símbolos, para que o código dependente de arquitetura possa efetuar algumas correções necessárias.

char *tc_canonicalize_symbol_name(char *)

Módulo: symbol

Permite que código dependente de arquitetura adote um esquema de nomeação canônico para símbolos.

void tc_check_label(symbolS *)

Módulo: read

Permite que código dependente de arquitetura tome conhecimento do rótulo encontrado em uma determinada linha do código fonte. `Gas` chama essa rotina sempre que um rótulo é encontrado.

int tc_fix_adjustable(fixS *)

Módulo: write

Essa rotina é chamada em `adjust_reloc_syms()`. Caso retorne 0, o *fixup* atual não é reajustado em relação ao símbolo do segmento.

void tc_frob_file()

Módulo: write

Essa rotina pode ser utilizada pela arquitetura alvo para inspecionar aspectos referentes ao arquivo sendo criado, depois de a tabela de símbolos ter sido criada e antes de se escrever as entradas de relocação.

void tc_frob_file_after_relocs()

Módulo: write

Essa rotina pode ser utilizada pela arquitetura alvo para inspecionar aspectos referentes ao arquivo sendo criado, depois de se escrever as entradas de relocação e antes da escrita do conteúdo do arquivo objeto.

void tc_frob_file_before_adjust()

Módulo: write

Essa rotina pode ser utilizada pela arquitetura alvo para inspecionar aspectos referentes ao arquivo sendo criado, depois da resolução dos símbolos e antes de se reajustar os símbolos relocáveis.

void tc_frob_file_before_fix()

Módulo: write

Essa rotina pode ser utilizada pela arquitetura alvo para inspecionar aspectos referentes ao arquivo sendo criado, depois de se reajustar os símbolos relocáveis e antes de se processar os *fixups* para cada segmento.

void tc_frob_label(symbolS *)
Módulo: write
Essa rotina pode ser utilizada para inspecionar todo novo rótulo que é criado.

void tc_frob_section()
Módulo: write
Essa rotina pode ser utilizada pela arquitetura alvo para inspecionar aspectos referentes ao arquivo sendo criado, após os segmentos terem sido redimensionados (via `size_seg()`).

void tc_frob_symbol(symbolS *, int)
Módulo: write
Essa rotina pode ser utilizada pela arquitetura alvo para inspecionar aspectos referentes ao arquivo sendo criado no momento da resolução dos símbolos.

arelent *tc_gen_reloc(asection *, fixS *) (obrigatório)
Módulo: write
Essa rotina é chamada para cada *fixup* não resolvido para gerar entradas de relocação `BFD`.

void tc_init_after_args()
Módulo: as
Essa rotina é chamada logo após a linha de comando ter sido processada e antes da leitura dos arquivos fontes.

void tc_symbol_new_hook(symbolS *)
Módulo: symbols
Essa rotina pode ser utilizada para inspecionar todo novo símbolo que é criado (em `symbol_create()`).

int tc_unrecognized_line(char)
Módulo: read
Se definida, essa rotina é chamada para toda linha a qual o montador não conseguiu reconhecer como válida.

Referências Bibliográficas

[1] M. Abbaspour and J. Zhu. Retargetable binary utilities. In *Proceedings of the 39th Conference on Design Automation*, pages 331–336. ACM Press, June 2002.

[2] A. Baldassin and P. Centoducatte. Geração automática de montadores para modelos de arquiteturas escritos em ArchC. In *Proceedings of the 9th Brazilian Symposium on Programming Languages*, pages 36–49, May 2005.

[3] S. Chamberlain. *libbfd, the binary file descriptor library*. Free Software Foundation, Inc., April 1991. version 3.0.

[4] C. Donnelly and R. Stallman. *Bison, the YACC-compatible parser generator*. Free Software Foundation, Inc., 1.35 edition, February 2002.

[5] D. Elsner et al. *Using as, the GNU assembler*. Free Software Foundation, Inc., March 1993. version 2.15.

[6] A. Fauth and A. Knoll. Automated generation of DSP program development tools using a machine description formalism. In *Proceedings of the IEEE ICASSP-93*, pages 457–460, April 1993.

[7] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nML. In *Proceedings of the EDTC: The European Design and Test Conference*, pages 503–507. IEEE Computer Society, March 1995.

[8] M. Fernández. Simple and effective link-time optimization of Modula-3 programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 103–115. ACM Press, June 1995.

[9] M. Fernández and N. Ramsey. Automatic checking of instruction specifications. In *Proceedings of the 19th International Conference on Software Engineering*, pages 326–336. ACM Press, May 1997.

[10] M. Freericks. The nML machine description formalism. Technical Report Draft, version 1.5, Technical University of Berlin, July 1993.

[11] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the 4th Annual IEEEWorkshop on Workload Characterization*, pages 3–14, December 2001.

[12] G. Hadjiyiannis. *ISDL: instruction set description language - version 1.0*. MIT Laboratory for Computer Science, November 1998.

[13] G. Hadjiyiannis, S. Hanono, and S. Devadas. ISDL: an instruction set description language for retargetability. In *Proceedings of the 34th Annual Conference on Design Automation*, pages 299–302. ACM Press, June 1997.

[14] G. Hadjiyiannis, P. Russo, and S. Devadas. A methodology for accurate performance evaluation in architecture exploration. In *Proceedings of the 36th ACM/IEEE Conference on Design Automation*, pages 927–932. ACM Press, June 1999.

[15] S. Hanono and S. Devadas. Instruction selection, resource allocation, and scheduling in the AVIV retargetable code generator. In *Proceedings of the 35th Annual Conference on Design Automation*, pages 510–515. ACM Press, June 1998.

[16] M. R. Hartoog, J. A. Rowson, P. D. Reddy, S. Desai, D. D. Dunlop, E. A. Harcourt, and N. Khullar. Generation of software tools from processor descriptions for hardware/software codesign. In *Proceedings of the 34th Annual Conference on Design Automation*, pages 303–306. ACM Press, June 1997.

[17] A. Hoffmann, H. Meyr, and R. Leupers. *Architecture exploration for embedded processors with LISA*. Kluwer Academic Publishers, 2002.

[18] A. Hoffmann, A. Nohl, G. Braun, O. Schliebusch, T. Kogel, , and H. Meyr. 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*, pages 1338–1354, November 2001.

[19] A. Hoffmann, A. Nohl, S. Pees, G. Braun, and H. Meyr. Generating production quality software development tools using a machine description language. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 674–678. IEEE Press, March 2001.

[20] S. Kumari. Generation of assemblers using high level processor models. Master's thesis, Department of CSE, IIT, Kanpur, February 2000.

[21] Y. Langsam, M. Augenstein, and A. Tenenbaum. *Data Structures Using C and C++*. Prentice Hall, second edition, 1996.

[22] D. Lanneer, J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G.Goossens. CHESS: Retargetable code generation for embedded DSP processors. In *Code generation for embedded processors*, chapter 5, pages 85–102. Kluwer Academic Publishers, 1995.

[23] D. Mackenzie, B. Elliston, and A. Demaille. *Autoconf - Creating Automatic Configuration Scripts for version 2.57*. Free Software Foundation, Inc., December 2002.

[24] D. Mackenzie and T. Tromey. *GNU Automake for version 1.7.2*. Free Software Foundation, Inc., December 2002.

[25] D. Patterson and J. Hennessy. *Organização e projeto de computadores - A interface Hardware/Software*. Livros Técnicos e Científicos Editora S.A., segunda edição edition, 2000.

[26] V. Paxson. *Flex, a fast scanner generator*. Free Software Foundation, Inc., 2.5 edition, March 1995.

[27] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr. LISA – machine description language for cycle-accurate models of programmable DSP architectures. In *Proceedings of the 36th ACM/IEEE Conference on Design Automation*, pages 933–938. ACM Press, June 1999.

[28] R. H. Pesch and J. M. Osier. *The GNU binary utilities*. Free Software Foundation, Inc., May 1993. version 2.15.

[29] V. Rajesh. A generic approach to performance modeling and its application to simulator generator. Master's thesis, Department of CSE, IIT, Kanpur, August 1998.

[30] A. Rajiv. Retargetable profiling tools and their application in cache simulation and code instrumentation. Master's thesis, Department of CSE, IIT, Kanpur, December 1999.

[31] N. Ramsey and M. Fernández. New Jersey machine-code toolkit architecture specifications. Technical Report TR-470-94, Department of Computer Science, Princeton University, October 1994.

[32] N. Ramsey and M. Fernández. New Jersey machine-code toolkit reference manual. Technical Report TR-471-94, Department of Computer Science, Princeton University, October 1994.

[33] N. Ramsey and M. Fernández. Specifying representations of machine instructions. In *Proceedings of the ACM Transactions on Programming Language and Systems*, pages 492–524. ACM Press, May 1997.

[34] N. Ramsey and M. F. Fernández. The New Jersey machine-code toolkit. In *Proceedings of the 1995 USENIX Technical Conference*, pages 289–302, January 1995.

[35] N. Ramsey and D. R. Hanson. A retargetable debugger. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 22–31. ACM Press, June 1992.

[36] M. N. Rao. SrijanSoft: a retargetable software synthesis framework for heterogeneous multiprocessors. Master's thesis, Department of CSE, IIT, Delhi, May 2004.

[37] S. Rigo. *ArchC: Uma linguagem de descrição de arquiteturas*. PhD thesis, Instituto de Computação, UNICAMP, Campinas, Julho 2004.

[38] SPARC International, Inc. *The SPARC architecture manual - Version 8*, 1992. Revision SAV080SI9308.

[39] R. Stallman. *Debugging with GDB, the GNU source-level debugger*. Free Software Foundation, Inc., ninth edition, February 2004.

[40] R. Stallman. *Using the GNU compiler collection*. Free Software Foundation, Inc., May 2004. For GCC version 3.4.3.

[41] A. Stratling. Extending the GNU assembler for Texas Instruments TMS320C6x – DSP. Seminar Paper, Chemnitz University of Technology, February 2004.

[42] G. Vaughan, B. Elliston, T. Tromey, and I. L. Taylor. *Autoconf, Automake, Libtool*. Free Software Foundation, Inc., 1.4.2 edition, May 2004.

[43] A. Wolfe. Toolkit:GNU tools, still relevant? *Queue*, 1(9):14–17, December/January 2003-2004.

[44] V. Zivojnovic, S. Pees, and H. Meyr. LISA – machine description language and generic machine model for HW/SW co-design. In *Proceedings of the IEEE Workshop on VLSI Signal Processing*, pages 127–136, October 1996.