Caracterização de Técnicas de Codificação de Instruções Integrados a uma Infraestrutura Unificada de Software

Luiz Henrique Neves Bonifacio

SERVIÇO DE GRADUAÇÃO DA FACOM-UFMS

Data de Depósito:

Assinatura:_____

Luiz Henrique Neves Bonifacio

Orientador: *Prof. Me. Renan Albuquerque Marks*

Trabalho de conclusão de curso apresentado à Faculdade de Computação – FACOM-UFMS como parte dos requisitos necessários à obtenção do título em Bacharel em Engenharia de Computação.

UFMS – Campo Grande
Abril/2016

Agradecimentos

Ouve-se por ai que motivação e agradecimentos não duram para sempre. Bem, tampouco dura o efeito do banho, por isso recomenda-se diariamente. Por isso, gostaria de agradecer a todas as pessoas que de forma direta ou indireta colaboraram com a conclusão deste trabalho.

Meu maior e eterno obrigado à minha mãe, Dilma Aparecida das Neves, por todo o esforço, dedicação, amor e paciência comigo desde as minhas primeiras palavras até as incontáveis dificuldades da vida acadêmica. Esta conquista jamais aconteceria sem o seu apoio e suporte e tudo o que sou hoje devo a você. Não é possível expressar com palavras o gigantesco sentimento de gratidão que tenho por você.

Agradeço também a todos meus amigos, presentes e ausentes, aos de longuíssima data e aos que a faculdade me trouxe, por todas as palavras de motivação, os gestos de carinho e por se fazerem presentes como uma família.

Imprescindível agradecer também a todos os professores que tive durante a graduação, que tanto contribuíram para minha formação profissional. Um agradecimento especial à meu orientador, Professor Renan Albuquerque Marks, pelos 3 anos de orientação, pela inesgotável paciência, dedicação e pela imensurável contribuição para a minha jornada acadêmica.

Resumo

Neste trabalho é descrito um processo de codificação de instruções da arquitetura SPARCv8 para sua extensão arquitetural SPARC16. A abordagem adotada no processo de codificação realiza a conversão do programa que deseja-se codificar, para sua representação intermediária. Uma representação intermediária é um formalismo que descreve o comportamento (fluxo) de um programa de computador, fazendo uso de objetos que representam as partes que compõem o programa.

Neste trabalho, aborda-se primeiramente conceitos teóricos fundamentais para o entendimento do trabalho realizado, incluindo as definições da arquitetura SPARCv8, bem como de sua extensão SPARC16, definições sobre o tipo de representação intermediária utilizada no trabalho e uma descrição completa sobre o uso da biblioteca libFIRM, principal ferramenta de desenvolvimento do trabalho.

A descrição do desenvolvimento do trabalho preocupa-se em relatar de forma objetiva como ocorreram as etapas práticas deste trabalho, tais como a manipulação de arquivos ELF e utilização de rotinas da biblioteca libFIRM. Ao final, são apresentados exemplos que corroboram com a funcionalidade deste trabalho.

Palavras-chave: arquitetura de computadores, SPARCv8, SPARC16.

Abstract

This paper describes an instruction coding process, from SPARCv8 architecture to its architectural extension SPARC16. The approach adopted in the encoding process performs the program conversion that is intended to encode, to their intermediate representation. An intermediate representation is a formalism that describes the behavior (flow) of a computer program, making use of objects which representing parts composing the program.

In this work, it approaches first fundamental theoretical concepts for the understanding of the work performed, including the settings of SPARCv8 architecture, as well as its SPARC16 extension, settings about the kind of intermediate representation used at this work and a full description about the use of the libFIRM library, the main development tool work.

The work development description is concerned to report objectively as occurred practical steps in this work, such as manipulating ELF files and using routines from the libFIRM library. At the end, there are examples that corroborate the functionality of this work.

Keywords: computer architecture, SPARCv8, SPARC16.

Sumário

Lista de Figuras

CAPÍTULO

1

Introdução

As atuais necessidades tecnológicas demandam sistemas computacionais cada vez mais complexos, com mais capacidade de processamento e melhor desempenho. Somado a isto, existe ainda a necessidade da integração destes sistemas em um único chip. Entretanto, como o espaço disponível em um chip é reduzido, existem restrições de energia e memória. Diversas soluções para este problema foram propostas, como por exemplo, soluções que visam diminuir o tamanho do código dos programas que executam nestes sistemas. Neste contexto, existem duas abordagens que visam diminuir o tamanho do código: compressão de instruções e codificação de instruções.

A compressão de instruções reduz o tamanho das instruções não se preocupando com sua semântica, ou seja, não é possível identificar operandos e demais informações de uma instrução após a mesma ser comprimida. Por outro lado, a codificação de instruções reduz o tamanho da instrução, preocupando-se em manter o valor dos campos que a compõe, sendo possível identificar operandos e demais informações de uma instrução após sua codificação. Na literatura, diversos trabalhos estão disponíveis, dentre eles podemos citar a técnica PBIW (*Pattern Based Instruction Word*) [2], que é baseada na fatoração de instruções codificadas e padrões. Um algoritmo mapeia as instruções do programa, extraindo delas padrões que são armazenados em uma memória cache denominada *P-cache*. Nestes padrões estão armazenados sinais de controle, *opcodes* e ponteiros para operandos armazenados nas instruções. Os operandos (registradores e imediatos) estão armazenados nas instruções codificadas, que por sua vez são armazenadas na *I-cache*. Esta codificação foi projetada para ser aplicada em arquiteturas que possuam instruções grandes como, por exemplo, processadores VLIW (*Very Long Instruction Word*).

Neste trabalho será abordada a técnica de codificação de instruções da arquitetura SPARCv8, através da conversão de programas para uma representação intermediária.

CAPÍTULO

2

Contextualização do Projeto

2.1 SPARCv8 e SPARC16

A arquitetura SPARC, acrossemia para ***S**calable **P**rocessor **Arc**hitecture*, é uma arquitetura de processadores que possui um conjunto reduzido de instruções e está definida no padrão IEEE 1754-1994 [7]. Possui registradores de propósito geral, ponto flutuante e de controle e 72 instruções de 32 bits distribuídas em três formatos distintos. O processador SPARC possui uma unidade de inteiros e pode ou não apresentar uma unidade de ponto flutuante, juntamente com um coprocessador.

O conjunto de registradores de propósito geral é composto por 32 registradores, respeitando a seguinte divisão: 8 registradores globais e 24 registradores de janela. Um fato importante sobre a arquitetura SPARCv8 é que a execução das rotinas é feita através de janelas. Tais janelas são definidas como segmentos onde distribui-se as partes de um programa que faz uso do conjunto de registradores. Uma implementação da arquitetura SPARCv8 pode ter de 2 a 32 janelas. Cada janela contém 8 registradores de entrada, 8 registradores locais e 8 registradores de entrada e saída da janela adjacente. A janela corrente é indicada por um ponteiro chamado CWP (*Current Window Pointer*), que mantém o controle sobre as janelas. Cada janela compartilha os registradores de entrada e saída com outras duas janelas adjacentes.

Em determinado momento, um programa pode acessar os 8 registradores globais e uma janela com mais 24 registradores. Se uma rotina que está em execução faz uma chamada a outra rotina, então a janela de registradores é alterada, fazendo com que a rotina que foi chamada tenha acesso aos registradores adjacentes da rotina chamadora. A Figura 2.1 [4] mostra o funcionamento das janelas para a chamada de funções (sentido *SAVE-TRAP*) e retorno de funções (sentido *RESTORE-RETT*).

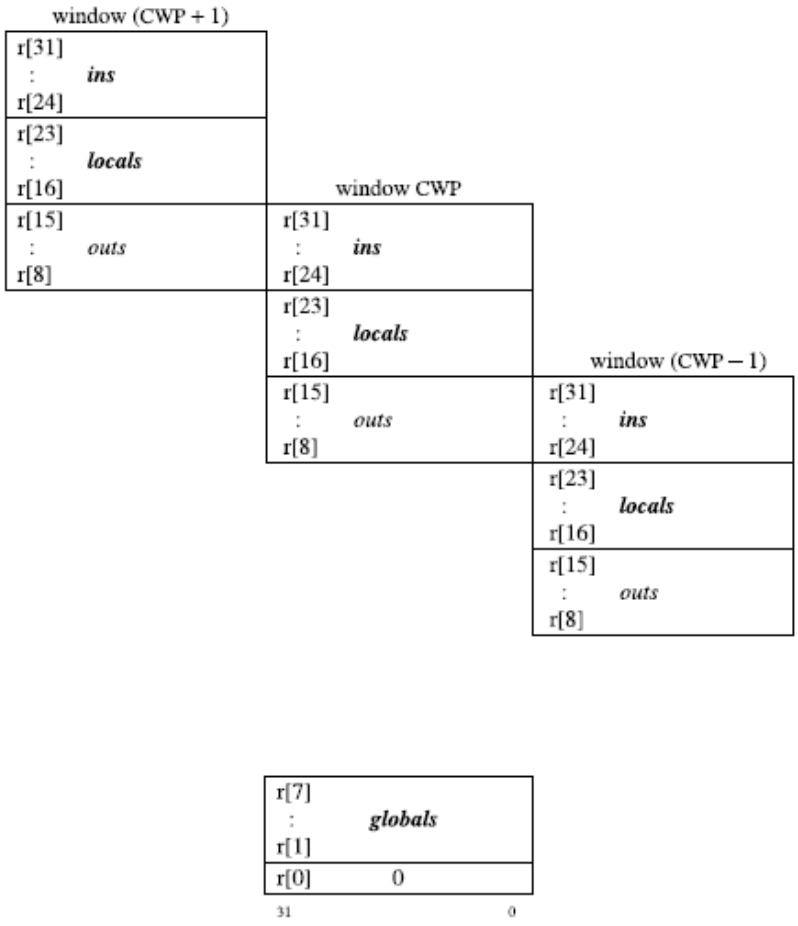

Figura 2.1: Três janelas sobrepostas e os 8 registradores globais

As 72 instruções SPARCv8 são codificadas em 3 formatos de 32 bits, conforme ilustrado na Figura 2.2 [4], divididas nos formatos 1, 2 e 3. A codificação do campo *op* é o que determina o formato de uma instrução. As instruções do formato 1 são instruções de salto incondicional (*call*) e possuem apenas dois campos, um campo *op* e outro para um imediato de 30 bits, cujo valor é somado ao PC (*Program Counter*) quando uma instrução deste tipo é executada. Já as instruções do formato 2 estão subdivididas em dois subformatos: o primeiro subformato corresponde à instrução *sethi*, que é utilizada para carregar um imediato de 22 bits nos bits mais significativos de um registrador, representado pelo campo *rd*. O segundo subformato corresponde as instruções de desvio condicional (*branch*). O campo que representa o *op* secundário (*op2*) é responsável por diferenciar qual é o tipo de instrução. Por último, o formato 3 representa as instruções lógicas, aritméticas, de deslocamento de bits e de acesso à memória, subdivididas em três subformatos.

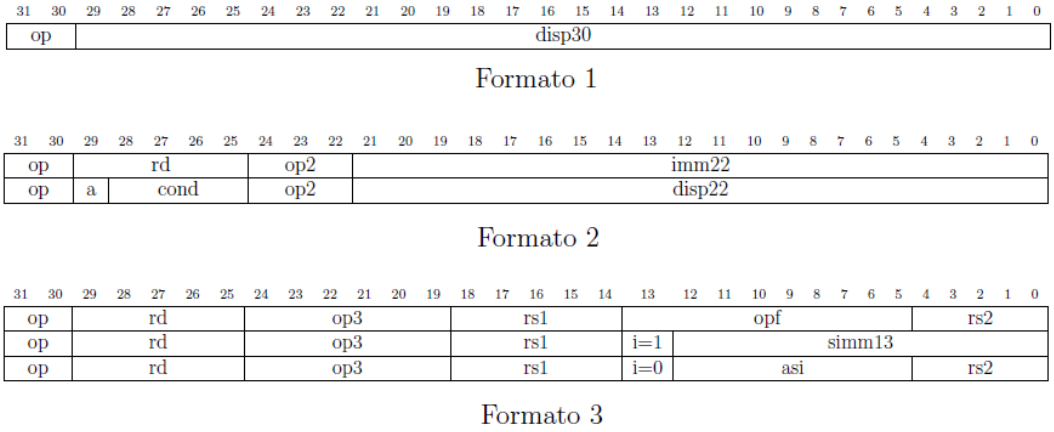

Figura 2.2: Formato das instruções SPARCv8

A extensão arquitetural SPARC16 tem suas instruções codificadas em 16 bits. Devido a isto, ela apresenta um conjunto reduzido de registradores e os valores imediatos representados são menores. A extensão SPARC16 possui um conjunto de 8 registradores visíveis, divididos em 3 registradores de entrada, 3 registradores de saída, 1 registrador local e 1 registrador global. A extensão SPARC16 também utiliza o mecanismo de janelas para acesso ao conjunto de registradores, assim como o SPARCv8.

Para minimizar os problemas causados pelo número reduzido de registradores disponíveis, utiliza-se o mecanismo de *spill*. Este mecanismo permite armazenar variáveis na memória quando não é possível que todas as variáveis utilizadas em um programa sejam armazenadas em registradores. Assim, a variável é transportada do registrador para a memória, liberando o registrador para outro uso. Todas as referências feitas a esta variável agora são feitas ao endereço de memória em que ela está armazenada. Ao ser usada, a variável é restaurada da memória para o registrador, caracterizando o processo de *fill*.

A extensão SPARC16 faz referência implícita a alguns registradores em suas instruções, como é o caso da instrução *save*, que utiliza o registrador *%sp* (*stack pointer*). Além deste, outros 3 registradores também são utilizados de maneira implícita: registrador %g0 (zero), %fp (*frame pointer*) e %ra (*return address*). É possível realizar saltos entre rotinas codificadas em SPARC16 para SPARCv8 e vice-versa. Estes saltos ocorrem através das instruções *call with exchange* (*callx*), que invoca rotinas implementadas em SPARCv8, *call on register with exchange* (*callregx*), utilizada para saltar de código SPARC16 para SPARCv8, *jump on register with exchange* (*jumpregx*) e *branch with exchange* (*bx*), que realiza saltos incondicionais para rotinas implementadas em SPARCv8. Para saltar de rotinas SPARCv8 para SPARC16 existem duas instruções: *branch with exchange* (*bx*) já citada anteriormente e a instrução *jump and link with exchange* (*jmplx*).

O número de formatos de instruções SPARC16 é maior se comparado ao SPARCv8. Entretanto, esta característica permite que instruções que aparecem com mais frequência no código, tenham campos maiores para seus imediatos e possam operar fazendo uso de 3 registradores (2 registradores fonte e 1 destino). Devido ao número de reduzido de bits para codificar as

instruções, nem todas as instruções podem fazer uso de 3 registradores.

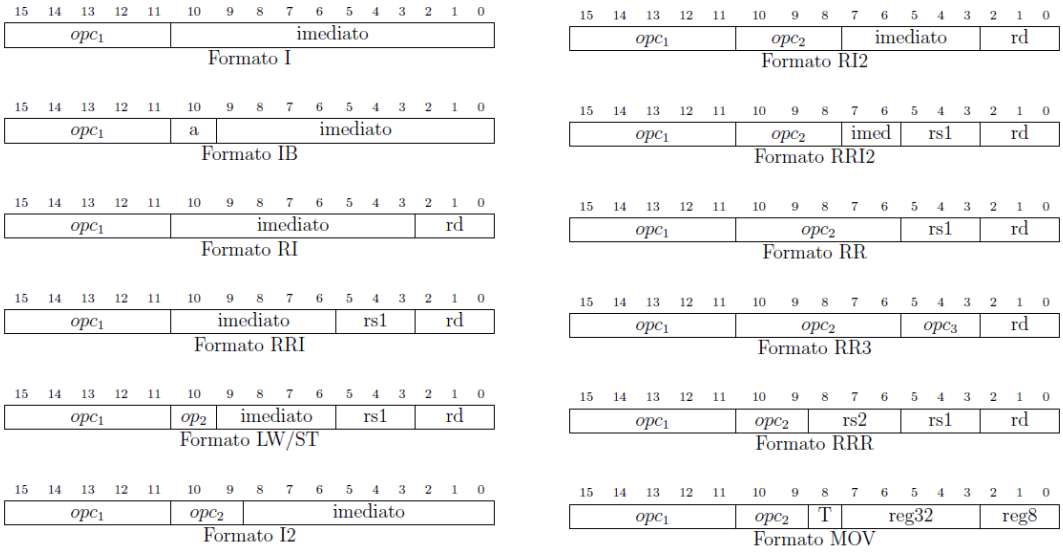

Figura 2.3: Formato das instruções da extensão SPARC16

A Figura 2.3 [4] ilustra os 12 formatos utilizados para codificar as instruções na extensão SPARC16. No formato I estão codificadas as instruções do tipo *call* e *call with exchange*, responsáveis por realizar chamadas a funções e execução de rotinas codificadas em SPARCv8, respectivamente.

No Formato IB estão codificadas as instruções que realizam saltos, *branchs*. No formato RI estão as instruções de comparação (*cmp*) e cópia de dados (*mov*) e no formato RRI estão as instruções aritméticas, que fazem uso de dois registradores e um imediato.

No Formato *LW/ST* estão codificadas as instruções de carregamento de dados, classificadas como menos convencionais (por exemplo, *load unsigned half word*). Vale ressaltar que a extensão SPARC16 utiliza deslocamento de imediatos em suas instruções de *load*/*store* para carregar imediatos do tipo *double word*. O imediato é deslocado 3 bits à esquerda, o que equivale a multiplica-lo por 8. Isso é necessário porque o padrão SPARC exige que *double words* estejam alinhadas em posições da memória que sejam múltiplas de 8.

No formato RRR estão as instruções que utilizam apenas registradores em seus operandos e no formato RR estão codificadas as operações pouco frequentes.

Nos formatos I e I2 estão instruções que utilizam apenas um imediato. No Formato RI2 estão instruções que fazem uso do *stack pointer* ou *frame pointer*.

No formato RRI2 estão instruções similares as do formato RI2, porém acrescidas de um imediato para operações aritméticas.

No formato RR3 estão instruções de apenas um registrador, que saltam para o endereço contido no registrador.

Finalmente, no formato MOV estão codificadas instruções com capacidade de copiar valores dos registradores visíveis da extensão SPARC16 para registradores invisíveis (que não podem ser acessados diretamente) do SPARCv8.

2.2 Static Single Assignment (SSA)

Desenvolvida na década de 80 por pesquisadores da IBM [1], a representação SSA (***S**tatic **S**ingle **A**ssignment*) é uma representação intermediária (RI) do fluxo de dados de um programa. Nela, cada variável (dado) encontrada no escopo do programa é definida uma única vez. Tal característica é o que difere a forma SSA das demais representações intermediárias, que por não utilizarem uma definição única para cada variável, acabam criando uma trilha de definições repetidas para representar o programa. A forma SSA foi criada com o objetivo de tornar mais eficiente a análise de um programa. É importante ressaltar que, apesar de cada variável ser definida apenas uma vez, ela pode ser usada e referenciada múltiplas vezes durante a representação intermediária.

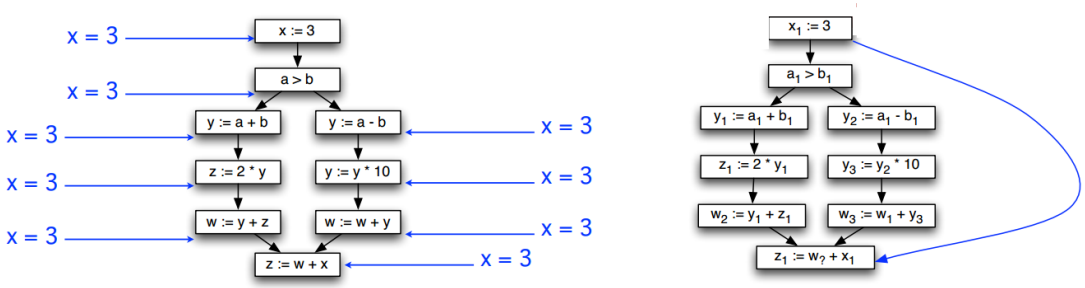

Figura 2.4: Exemplo de duas representações intermediárias do mesmo programa

Na Figura 2.4 observa-se dois exemplos de representações intermediárias. O exemplo da direita ilustra a forma SSA. Nota-se que no exemplo da esquerda, o valor da variável x é propagado em todas as linhas do código, mesmo que ele não seja utilizado, pois as variáveis envolvidas nas operações são outras. Já na forma SSA, o valor de x é propagado através de uma aresta, somente quando este é necessário. Outra característica importante da forma SSA é como esta lida com os desvios condicionais. Para cada desvio, é criado um nó chamado *phi* (ϕ). Neste nó, são alocados os argumentos envolvidos no desvio. Assim, o argumento é selecionado dependendo de como o fluxo de dados atingir este nó, determinando o valor utilizado no destino do desvio. Operacionalmente, esse comportamento é implementado atribuindo-se o mesmo registrador ou endereço de memória para todos os argumentos de um nó ϕ.

A Figura 2.5 ilustra a criação do nó ϕ na forma SSA.

No exemplo, a variável x_2 é comparada com o número 3. O passo seguinte, depende estritamente do resultado de tal comparação, bem como o que será feito com as demais variáveis do programa, existindo dois possíveis resultados. Como não é possível se antecipar ao resultado do programa, é necessário que todas as variáveis envolvidas sejam armazenadas para posterior uso. Entretanto, somente uma delas será utilizada. Para evitar atribuições repetitivas, é criado então o nó ϕ. O resultado da operação será atribuído a variável y_3.

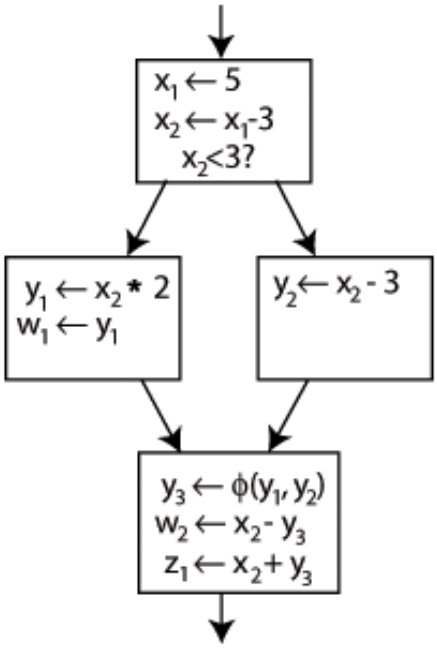

Figura 2.5: Ilustração de um nó ϕ

2.3 libFIRM

O libFIRM [3] é uma biblioteca C que implementa a representação intermediária SSA que é utilizada para transformar um programa de computador em um Grafo de Dependência Explícita (GDE), afim de tornar mais fácil analisá-lo e possibilitar sua transformação. Um programa, quando representado através de um GDE por meio do libFIRM, consiste em um conjunto de entidades, também definidas como os nós que compõem o GDE. Este conjunto inclui funções, variáveis globais, strings e valores inteiros, estruturas de inicialização, classes e tabelas de métodos virtuais [3]. O libFIRM conta com um sistema de modelagem em tipos, aninhado em estruturas, classes hierárquicas e funções. O GDE é um grafo dirigido, ou seja, suas arestas possuem sentido, e marcado, onde cada nó possui um rótulo. Deste modo, cada função de um programa é representado por um grafo, e as operações contidas nesta função são representadas pelos nós do grafo. Arestas conectando estes elementos denotam as dependências entre eles, tais como dependência de dados e controle, uma vez que o libFIRM define seu modelo baseado em dependências ao invés de fluxo de execução. Como os nós do grafo representam as operações, estes podem produzir múltiplos resultados, combinados em uma tupla. Um nó denominado *Proj* é o responsável por extrair o valor desta tupla. Cada nó do grafo é marcado com uma assinatura que representa seu funcionamento e estão ordenados de acordo com as entradas e saídas do programa. As arestas que conectam os nós determinam os tipos de dependências que ocorrem entre as operações, divididas entre dependência de dados, memória e controle de fluxo.

```
int x;
if (7 > 4) {
        x = 9;
} else {
        x = 3;
}
return x;
```

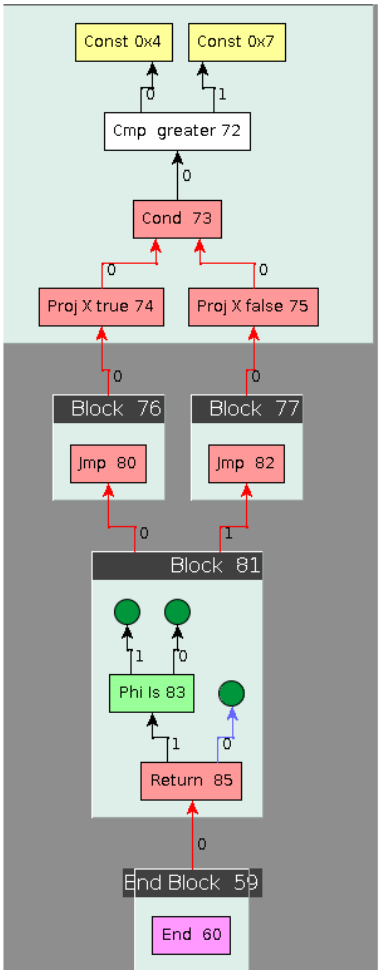

Figura 2.6: Representação intermediária gerada pelo libFIRM

A Figura 2.6 mostra a representação intermediária implementada pelo libFIRM, a partir do trecho de código contido na figura. Nela, nota-se a existência de blocos básicos, que representam os segmentos do código que são executados de maneira contígua, sem interrupções, nós contidos dentro dos blocos, que representam as instruções executadas no código e arestas ligando as instruções e os blocos.

O significado de cada aresta está diretamente relacionado com a sua cor [3]. Arestas de cor preta representam dependência de dados, ou seja, as instruções conectadas diretamente com seus operandos. Arestas de cor vermelha representam o fluxo de controle. Todas as arestas apontam para o sentido oposto do fluxo de execução, apontando para a operação e não para o bloco. Vale ressaltar que estas arestas não representam dependências de controle. Arestas de cor verde representam dependências sem semântica especial. Arestas de cor violeta representam código morto, porém alcançável, tais como um *loop* infinito.

O libFIRM pode receber como entrada programas escritos em linguagem C e Java, definindo assim vários *front-ends*, implementados, por exemplo, pelo *cparser*, que recebe como entrada

programas escritos em C, *Java ByteCode2Firm*, que converte *Java bytecodes* para código de máquina e *BrainFuck*, que converte programas *brainfuck*. Estas ferramentas contém um pré-processador, um *parserC*, o construtor da árvore sintática abstrata, além de realizar a análise semântica do programa de entrada.

O *middle-end*, que consiste no próprio libFIRM, onde é possível realizar as alterações desejadas para que se obtenha o resultado no *back-end*. Já o *back-end* consiste na geração de código de máquina nativo e de grafos de dependência explícita. O *back-end* do libFIRM possui suporte para as arquiteturas SPARC, ARM e x86.O *back-end* possui ainda a característica de realizar otimizações durante a análise e geração de código, tais como: eliminação de código morto, *constant folding* (valores constantes são précalculados), eliminação de código inalcançável, otimizações de *Load/Store* e de fluxo de controle.

Neste trabalho, foi acrescentado ao *front-end* do libFIRM a capacidade de receber como entrada um arquivo ELF contendo as instruções em formato binário de um programa SPARCv8. Acrescentou-se ao *back-end* do libFIRM o suporte a geração de código de máquina referente a extensão SPARC16. Para tal, criou-se uma cópia dos arquivos do libFIRM que descrevem o *back-end* do SPARCv8, com o objetivo de alterar esses arquivos, criando assim uma descrição do *back-end* da extensão SPARC16. Nestes arquivos, alterou-se o conjunto de registradores e a sintaxe de instruções, para que ao final da execução, fosse possível gerar código SPARC16.

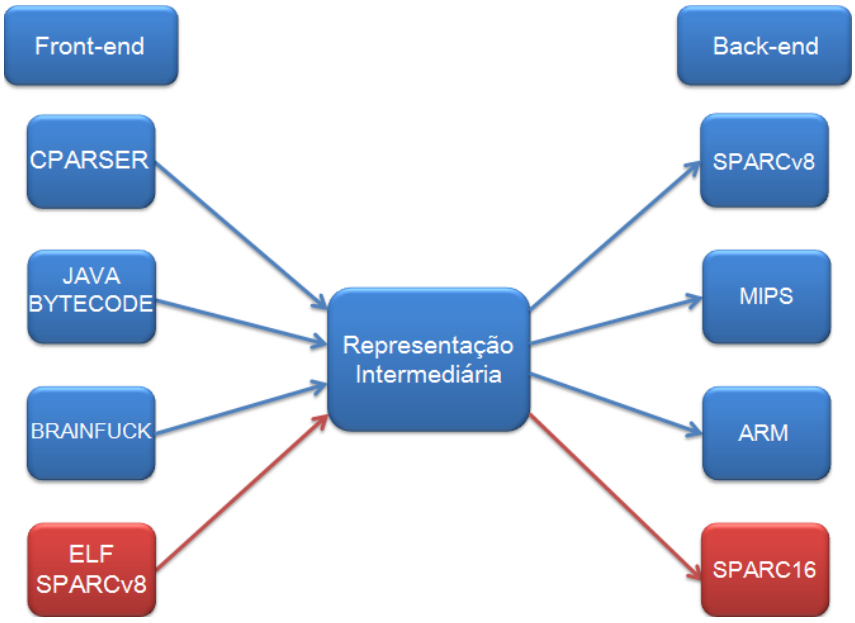

Figura 2.7: Relação entre Representação Intermediária, *Front-end* e *Back-end*

CAPÍTULO

3

Desenvolvimento

3.1 *Executable and Linkable Format (ELF)*

Um arquivo ELF (*Executable and Linkable Format*) é um padrão para arquivos executáveis, arquivos-objeto e bibliotecas. Teve origem originalmente no *System V Unix*, sistema operacional comercial da empresa *AT&T*, no ano de 1980 [6]. Alguns anos depois, um conjunto de empresas, dentre elas *IBM* e *Intel*, passou a controlar este padrão de arquivos. Este padrão de arquivos garante portabilidade, ou seja, é possível link-editar códigos gerados por compiladores de diferentes sistemas operacionais. Um arquivo ELF tem sua estrutura dividida em seções, conforme ilustrado na Figura 3.1 [6].

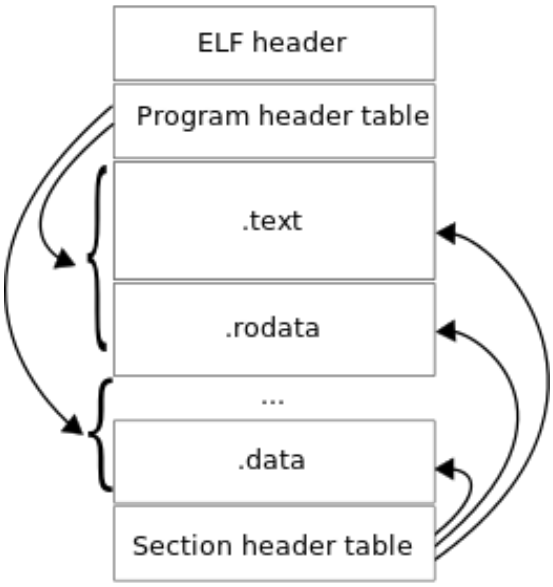

Figura 3.1: Organização interna de um arquivo ELF

As principais seções são: ***ELF header***, que descreve a organização do arquivo; ***Program***

header table, que específica como executar o programa; ***Section header table***, que contém informações sobre todas as seções do arquivo; ***text***, que armazena as instruções do programa em formato binário; ***data***, que armazena as variáveis utilizadas no programa; ***rodata***, que armazena *strings*. Neste trabalho, para a geração de arquivos ELF, utilizou-se a linguagem para descrição de arquiteturas **ArchC** [8], que torna possível simular processadores.

Neste contexto, utilizou-se a ferramenta "*sparc−elf−as*", que recebe como entrada um programa *assembly* SPARCv8 e devolve como saída um arquivo ELF que contém todas as informações do programa *assembly* dado como entrada, em formato binário. A seção de maior interesse é a seção *text*, que contém todas as instruções SPARCv8 do programa. Para a leitura e manipulação do conteúdo dos arquivos ELF, utilizou-se a biblioteca ELFIO [5], que provê uma interface para a leitura de tais arquivos, dispondo de um conjunto de rotinas, capazes de extrair informações como o tamanho do arquivo, número e tamanho das seções e a leitura do conteúdo de cada seção.

A seção de maior interesse, como mencionado anteriormente, é a seção de texto, que armazena as instruções SPARCv8 em formato binário.

Criou-se um programa que realiza a manipulação do arquivo ELF, fazendo uso das rotinas disponíveis na biblioteca ELFIO. Na primeira parte do programa é realizada uma verificação do arquivo ELF que se deseja ler, afim de verificar se o arquivo realmente existe na pasta indicada, bem como se todas suas seções existem e não apresentam erros. Se qualquer uma das verificações falhar, uma exceção é lançada e o programa é finalizado. Caso contrário, o processo de leitura do arquivo ELF tem sequência.

Verifica-se então todas as seções do arquivo, de onde obtêm-se o índice da seção *text*. Com o índice da seção de interesse em mãos, obtém-se o tamanho desta seção. Com estas informações, leu-se o conteúdo da seção *text* e este conteúdo foi armazenado em uma variável do tipo *string*. Posteriormente, manipulou-se o conteúdo da *string*, retirando os valores de suas posições, onde cada posição da *string* correspondia a 1 *byte*. Assim, uma instrução possui seus 32 bits divididos em 4 posições consecutivas. Retirou-se então o conteúdo das posições, em valores múltiplos de 4, resultando em uma instrução completa de 32 bits. As instruções foram armazenadas em uma nova estrutura de dados, lista encadeada, que armazena o índice da instrução, que representa a posição da instrução no código, e seus respectivos 32 bits, que a formam. O pseudocódigo 1 que descreve o processo descrito.

Algoritmo 1: Manipulação de arquivo ELF

Data: ELF file

Result: Estrutura de Dados contendo as instruções

```
1  verifica se arquivo ELF está na pasta;
2  contabiliza seções do arquivo ELF;
3  obtém índice da seção .text;
4  obtém tamanho da seção .text;
5  lê conteúdo da seção .text e armazena em string;
6  t ← tamanho da string ;
7  j ← 0;
8  while j < t do
9  |   lista_ligada ← string[j];
10 |   j ← j + 1;
11 end
```

3.2 Análise Estática do arquivo ELF

Após todas as instruções terem sido retiradas do arquivo ELF e armazenadas em uma lista ligada, iniciou-se então o processo de identificação das instruções. Para tal processo, criou-se um *parser* com a finalidade de processar e identificar cada instrução e todos os campos que a formam. Este *parser* foi construído a partir de informações coletadas do *datasheet* [7] da arquitetura SPARCv8. Neste documento, estão especificados os formatos de todas as instruções, bem como os valores e tamanhos de todos os campos que as compõem. A partir dessas informações, construiu-se o *parser*. Criou-se também uma estrutura de dados do tipo *struct*, para armazenar as instruções, auxiliando o processo de conversão. Esta estrutura possui 3 classes, referentes aos 3 formatos de instruções do SPARCv8. Cada uma das classes possui campos que representam os campos que compõem uma instrução, como representado na Figura 3.2.

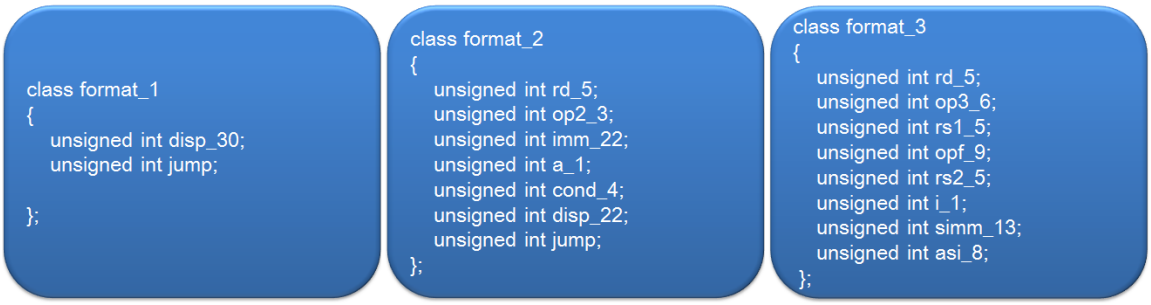

Figura 3.2: Classes que representam os formatos e campos do SPARCv8

Um pré-processamento é necessário para identificar todos os desvios existentes no programa e seus respectivos destinos. A cada desvio ou destino identificado, cria-se um novo bloco básico. Um bloco básico é uma entidade do libFIRM que representa um trecho de código que é executado de maneira contínua, sem interrupções. O pseudocódigo 2 representa a etapa de

pré-processamento.

Algoritmo 2: Préprocessamento de instruções

```
Data: Estrutura de Dados contendo as instruções
Result: GDE contendo todos os blocos básicos
1  t ← tamanho da string ;
2  j ← 0;
3  while j < t do
4      verifica o formato da instrução;
5      verifica o tipo da instrução;
6      if é instrução de desvio then
7          obtém o endereço de destino do desvio;
8          cria novo bloco básico no endereço de desvio;
9          flag[endereço] ← 1;
10         cria um novo bloco básico no endereço seguinte a instrução atual;
11         flag[atual + 1] ← 1;
12     else
13         processa a próxima instrução;
14     end
15     j ← j + 1;
16 end
```

No pré-processamento, verifica-se o formato e tipo de cada instrução, afim de descobrir se a instrução é uma instrução de desvio ou não. Caso seja uma instrução de desvio, é necessário obter seu campo que armazena o endereço de desvio. A partir deste endereço, é verificado se este imediato é um número positivo, referente a um endereço de desvio à frente do endereço da instrução atual, ou um imediato negativo, referente a um desvio para trás do endereço da instrução atual. Para instruções de desvio com condições, ou seja, quando uma comparação é feita para decidir se o desvio é tomado, o mesmo tem dois caminhos possíveis. Neste caso, cria-se dois blocos básicos, referentes aos dois possíveis caminhos gerados pelo desvio. Juntamente com a declaração dos blocos básicos, define-se também uma variável indicadora do tipo *flag*, com a finalidade de sinalizar na estrutura de dados que armazena as instruções, onde os blocos básicos têm início. Após todos os blocos básicos definidos, dá-se início ao processo de conversão para a representação intermediária.

3.3 Representação Intermediária do libFIRM

O processo de conversão para a representação intermediária inicia-se com o processamento da estrutura de dados que armazena as instruções. Através das informações retiradas do arquivo ELF, cria-se também uma entidade que representa o espaço utilizado na pilha (*stack*) pelo programa. Por meio de um *parser*, verifica-se primeiramente o campo *opcode* da instrução, identificando o tipo da instrução. Após identificado o o tipo da instrução, verifica-se então seu

opcode secundário, que indica qual a instrução em questão. Após a instrução corretamente identificada, retirou-se os bits referentes a cada campo da instrução. Nesta etapa, utilizou-se rotinas que foram implementadas para receber como entrada os 32 bits de uma instrução, a posição do primeiro bit do campo e número de bits do campo que deseja-se obter.

Com base nestas informações, essas rotinas extraem dos bits da instrução apenas os bits referentes ao campo desejado. Para cada formato e tipo de instrução, o número de bits que compõem determinado campo, bem como a posição do bit de início de cada campo, é variável. Portanto, as funções que implementam o *parser* são separadas de acordo com cada uma das instruções do SPARCv8. É importante ressaltar que o processamento do *parser* e a conversão para a representação intermediária ocorrem ao mesmo tempo. Isso significa que, uma instrução, ao ser identificada pelo *parser* é em seguida convertida para a sua representação intermediária. O processo de conversão é ilustrado na Figura 3.3.

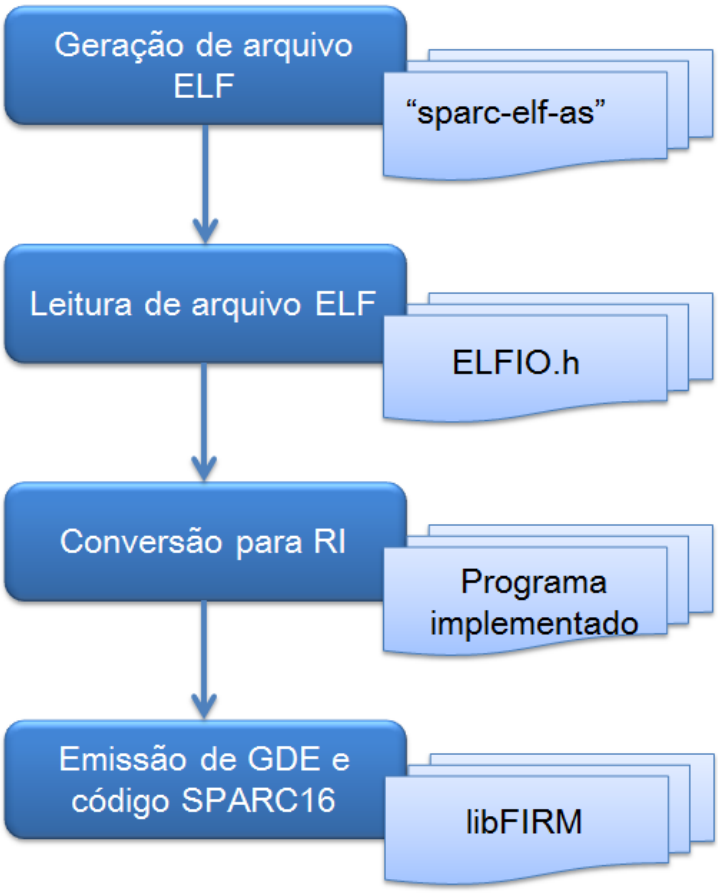

Figura 3.3: Processo de Conversão SPARCv8 para SPARC16

Através das rotinas implementadas pelo libFIRM, é possível criar uma correspondência entre a instrução e sua respectiva representação. Entretanto, é importante ressaltar que o libFIRM não é capaz de representar todas as instruções SPARCv8, ou seja, para algumas instruções SPARCv8 não existem representações diretas no libFIRM, como por exemplo, a instrução *NOR* do SPARCv8. Para resolver este problema, adotou-se a ideia de construir a representação intermediária dessas operações através de operações básicas. No caso do exemplo, construiu-se primeiramente a operação *OR* com os operandos e em seguida, com o resultado desta operação,

criou-se a operação *NOT*, que faz a negação do resultado da operação. Desta forma foi possível representar todas as instruções SPARCv8, fazendo uso de uma ou mais operações da representação intermediária do libFIRM. Ao final do processo de conversão, foram geradas duas saídas: um grafo (GDE) contendo todos os blocos básicos e operações da representação intermediária e um arquivo contendo as instruções SPARC16.

CAPÍTULO

4

Resultados

Nesta seção, apresenta-se os resultados gerados pelo trabalho. Serão apresentados exemplos da conversão entre programas simples escritos em SPARCv8 para a extensão SPARC16.

Para o primeiro exemplo, deu-se como entrada para o programa a função que calcula o fatorial de um número. A seguir, apresenta-se o código SPARCv8 dado como entrada:

```
  .section ".text"
# -- Begin  fatorial_v8
.p2align 4,,15
.globl fatorial_v8
.type fatorial_v8, #function
fatorial_v8:
/*.L0:*/
save %sp, -96, %sp
/* fallthrough to .L1 */
/*.L1:*/
or %g0, 1, %i3
or %g0, 1, %l5
ba .L2
.L3:
smul %i3, %l5, %i3
add %l5, 1, %l5
/* fallthrough to .L2 */
.L2:
cmp %l5, 7
ble .L3
/* fallthrough to .L4 */
```

```
/*.L4:*/
jmp %i7+8
 restore
.size fatorial_v8, .-fatorial_v8
# -- End  fatorial
```

Este código foi dado como entrada para a ferramenta "*sparc−elf−as*", que devolveu como saída um arquivo ELF contendo as instruções em formato binário. Usou-se este arquivo como entrada no programa implementado neste trabalho, obtendo-se como saída o GDE, representado na Figura A.1, que se encontra no Apêndice [A] deste trabalho, e o código SPARC16:

```
  .section ".text"
# -- Begin  fatorial_16
.p2align 4,,15
.globl fatorial_16
.type fatorial_16, #function
fatorial_16:
/*.L0:*/
or %g0, 1, %i0
ba .L2
 or %g0, 1, %l0
.L1:
add %l0, 1, %l0
/* fallthrough to .L2 */
.L2:
cmp %l0, 7
ble .L1
 smul %i0, %l0, %i0
/* fallthrough to .L3 */
/*.L3:*/
jmp %o7+8
 restore
.size fatorial_16, .-fatorial_16
# -- End  fatorial_16
.local data
.comm data,96,1
```

Neste exemplo, pode-se observar que, apesar do uso de poucos registradores, há uma pequena diferença entre os códigos SPARCv8 e SPARC16, como por exemplo o índice dos registradores usados pelas instruções. No código SPARCv8, faz-se uso dos registradores %i3 e

%l5. Entretanto, na extensão SPARC16, o conjunto de registradores é menor, resultando num mapeamento diferente de registradores, como pode-se observar no código SPARC16, que faz uso dos registradores %i0 e %l0. Além disso, pode-se notar que o código SPARC16 faz o uso de *delay slot* com a instrução *or* após a instrução que salta para o rótulo L2. Isso ocorre devido ao fato do libFIRM realizar otimizações internas durante a análise da árvore sintática.

O segundo exemplo apresenta uma função onde uma comparação determina qual fluxo seguir, existindo 3 fluxos possíveis. Este exemplo visa mostrar como o libFIRM manipula os blocos básicos previamente citados. O código dado como entrada foi:

```
.section ".text"
# -- Begin  ex_2_v8
.p2align 4,,15
.globl ex_2_v8
.type ex_2_v8, #function
ex_2_v8:
/*.L0:*/
save %sp, -96, %sp
/* fallthrough to .L1 */
/*.L1:*/
or %g0, 5, %l1
or %g0, 3, %l0
sub %g0, 1, %l2
cmp %l2, 0
be .L2
/* fallthrough to .L3 */
/*.L3:*/
cmp %l2, 1
be .L4
/* fallthrough to .L5 */
/*.L5:*/
jmp %i7+8
 restore %g0, 0, %o0
.L2:
sll %l0, %l1, %i0
jmp %i7+8
 restore
.L4:
sra %l0, %l1, %i0
jmp %i7+8
 restore
.size ex_2_v8, .-ex_2_v8
```

```
# -- End  ex_2_v8
```

Gerou-se então o arquivo ELF a partir deste código e deu-se como entrada para o programa. O GDE obtido na saída é apresentado na Figura B.1 no Apêndice [B] e o código SPARC16 é apresentado a seguir.

```
.section ".text"
# -- Begin  ex_2_16
.p2align 4,,15
.globl ex_2_16
.type ex_2_16, #function
ex_2_16:
/*.L0:*/
or %g0, 3, %l0
or %g0, 5, %g1
sub %g0, 1, %o0
cmp %o0, 0
be .L1
/* fallthrough to .L2 */
/*.L2:*/
cmp %o0, 1
be .L3
  sra %l0, %g1, %i0
/* fallthrough to .L4 */
/*.L4:*/
jmp %o7+8
 restore %g0, 0, %o0
.L1:
sll %l0, %g1, %i0
jmp %o7+8
 restore
.L3:
jmp %o7+8
 restore
.size ex_2_16, .-ex_2_16
# -- End  ex_2_16
.local data
.comm data,96,1
```

Neste exemplo, nota-se novamente o uso do *delay slot*, gerado pelas otimizações internas do libFIRM. Nota-se, ao observar a Figura B.1, que existem 3 possíveis fluxos através das arestas que ligam os blocos básicos.

O terceiro exemplo tem por objetivo mostrar o processo de *spill* de registradores, quando um programa SPARCv8 é codificado para a extensão SPARC16, que possui um conjunto reduzido de registradores. Para este exemplo, criou-se uma função que utiliza muitos registradores, pois ela realiza diversas operações e utiliza todas as variáveis. O código pode ser visto abaixo:

```
.section ".text"
# -- Begin  ex_3_v8
.p2align 4,,15
.globl ex_3_v8
.type ex_3_v8, #function
ex_3_v8:
/*.L0:*/
save %sp, -96, %sp
/* fallthrough to .L1 */
/*.L1:*/
or %g0, 2, %l1
or %g0, 1, %l0
sra %l0, %l1, %g2
and %g2, 2, %g1
and %g1, %g2, %l7
sra %l7, %g1, %l6
add %g2, %g1, %l0
sll %l6, %l0, %l5
and %l5, %l6, %l4
sub %l6, %l4, %l0
or %l0, %g2, %l3
sra %l7, %l3, %l2
and %l3, %g2, %l0
or %l0, %g1, %l1
or %l1, %g2, %l0
add %l0, %l2, %l2
cmp %l2, %g1
bg .L2
 cmp %l0, %l1
/* fallthrough to .L3 */
/*.L3:*/
bg .L4
 and %l5, %l4, %l0
/* fallthrough to .L5 */
/*.L5:*/
cmp %l3, %l1
```

```
bl .L6
 add %l2, %l0, %l0
/* fallthrough to .L7 */
/*.L7:*/
cmp %l5, %l4
be .L8
 sra %l0, %g2, %l0
/* fallthrough to .L9 */
/*.L9:*/
or %l0, %l6, %i0
jmp %i7+8
 restore
.L2:
jmp %i7+8
 restore %l0, %l1, %o0
.L4:
jmp %i7+8
 restore %g2, %l3, %o0
.L6:
or %l0, %l7, %i0
jmp %i7+8
 restore
.L8:
sub %l2, %l4, %l0
sub %l0, %l6, %i0
jmp %i7+8
 restore
.size ex_3_v8, .-ex_3_v8
# -- End  ex_3_v8
```

Após a geração do arquivo ELF e execução do programa, obteve-se

```
     .section ".text"
# -- Begin  ex_3_16
.p2align 4,,15
.globl ex_3_16
.type ex_3_16, #function
ex_3_16:
/*.L0:*/
or %g0, 2, %g1
or %g0, 1, %l0
sra %l0, %g1, %ra
```

```
and %ra, 2, %i2
and %i2, %ra, %i1
sra %i1, %i2, %g1
add %l0, 4294, %l0
st %g1, [%l0]
add %ra, %i2, %l0
sll %g1, %l0, %i0
and %i0, %g1, %o2
sub %g1, %o2, %l0
or %l0, %ra, %o1
sra %i1, %o1, %g1
and %o1, %ra, %l0
or %l0, %i2, %o0
or %o0, %ra, %l0
add %l0, %g1, %g1
cmp %g1, %i2
bg .L7
 cmp %l0, %o0
/* fallthrough to .L5 */
/*.L5:*/
bg .L3
 and %i0, %o2, %l0
/* fallthrough to .L5 */
/*.L5:*/
cmp %o1, %o0
bne .L6
 add %g1, %l0, %l0
/* fallthrough to .L9 */
/*.L9:*/
cmp %i0, %o2
be .L8
 sub %g1, %o2, %g1
/* fallthrough to .L2 */
/*.L2:*/
sra %l0, %ra, %g1
add %l0, 4294, %l0
ld [%l0], %l0
or %g1, %l0, %i0
jmp %i7+8
 restore
```

```
.L7:
jmp %i7+8
 restore %l0, %o0, %o0
.L3:
jmp %i7+8
 restore %ra, %o1, %o0
.L6:
or %l0, %i1, %i0
jmp %i7+8
 restore
.L8:
add %l0, 4294, %l0
ld [%l0], %l0
sub %g1, %l0, %i0
jmp %i7+8
 restore
.size ex_3_16, .-ex_3_16
# -- End  ex_3_16

.local data
.comm data,96,1
```

Analisando-se o código SPARC16 gerado pelo programa, percebe-se que nele existem instruções de *load* e *store* que não existem no código SPARCv8. Isso se deve ao fato de que na extensão SPARC16, para este programa, o número máximo de registradores disponíveis é atingido, obrigando o libFIRM a realizar o *spill*, para que um registrador seja liberado para uso. O primeiro *spill* acontece no primeiro bloco básico (rótulo L0), onde realiza-se o *store* do valor do registrador %g1 no endereço resultante da soma do registrador %l0 com o imediato 4294. Assim, o registrador %g1 agora está livre para uso, podendo recuperar seu valor anterior a qualquer momento, como ocorre no bloco básico com rótulo L2. Nele, o valor do registrador %g1 previamente armazenado é recuperado da memória e salvo novamente no registrador, para posterior uso.

CAPÍTULO

5

Considerações finais

Este trabalho teve como objetivo estabelecer uma nova forma de codificar instruções SPARCv8 para a sua extensão arquitetural SPARC16. Foi proposta uma codificação com um processo intermediário, que consiste em converter o programa que deseja-se codificar para sua representação intermediária. Para este processo fez-se uso da ferramenta libFIRM, uma biblioteca C que constrói a representação intermediária de um programa. Com o programa em sua representação intermediária, é possível realizar otimizações e transformá-lo para um *back-end* desejado, que neste trabalho, é a extensão SPARC16. Este processo facilita a codificação de instruções, não sendo necessário preocupar-se com a correspondência entre os conjuntos de registradores e instruções, por exemplo, tarefas que ficam a cargo da ferramenta que constrói a representação, o libFIRM.

Dentre as limitações e dificuldades do trabalho, pode-se citar a dificuldade em manipular a biblioteca libFIRM, que possui uma documentação escassa e pouco útil. Em alguns momentos, percebeu-se que a ferramenta apresentava um comportamento diferente do esperado, com peculiaridades para manipular algumas estruturas, o que acarretou em maior gasto de tempo para realizar um estudo sobre o libFIRM.

Este trabalho proporcionou a chance de conhecer mais profundamente a área de Compiladores, abordada na implementação da representação intermediária, e também a área de Arquitetura de Computadores, tratando-se da arquitetura SPARCv8 e sua extensão SPARC16. A interdisciplinaridade entre essas duas áreas permitiu o esclarecimento de alguns conceitos e a criação de novos, buscando compreender o funcionamento e integração de sistemas complexos, seja de uma abordagem em alto nível, como a representação intermediária de um programa, ou a nível binário, como a construção de uma instrução de 32 bits. Compreender o funcionamento isolado dessas partes e ter a capacidade de compreender como elas funcionam em conjunto, formando um sistema, é a capacidade desejada para um bom engenheiro.

Entre as sugestões de trabalhos futuros, estão: validação do código SPARC16 gerado, atra-

vés da simulação do processador SPARC, de forma que sejam executados o programa original e o programa emitido e seus resultados sejam comparados; acréscimo de instruções de ponto flutuante para a codificação SPARC16.

Referências Bibliográficas

[1] B. Apern, M N. Wegman, and F. K. Zadeck. Detecting equality of values in programs. . *Conf. Record of the 15th ACM Symposium on the Principles of Programming Languages (POPL)*, Janeiro 1988.

[2] Rafael Batistella, Ricardo Santos, and Rodolfo Azevedo. Pbiw: Uma codificação de instruções alternativa para arquiteturas de alto desempenho. In *WSCAD-SSC Simpósio em Sistemas Computacionais*, pages 151–158, Campo Grande - MS, 2008. Editora UFMS.

[3] Matthias Braun, Sebastian Buchwald, and Andreas Zwinkau. *FIRM: A Graph−Based Intermediate Representation*. Karlsruhe Institute of Technology (KIT), 2011.

[4] Leonardo Luiz Ecco. *SPARC16: uma nova visão de compressão para processadores SPARC*. PhD thesis, UNICAMP−Universidade Estadual de Campinas.

[5] Serge Lamikhov−Center. *ELFIO Tutorial and User Manual*.

[6] J. R. Levine. *Linkers & Loaders*. Morgan Kaufmann Publishers, San Francisco, CA, 2000.

[7] Inc SPARC International˙ *The SPARC Architecture Manual, Version 8*, 1992. Revision SAV080SI9308.

[8] The ArchC Team. *The ArchC Architecture Description Language Reference Manual*. Instituto de Computação−Universidade Estadual de Campinas, CampinasSP, Agosto 2007. http://www.archc.org.

APÊNDICE

A

GDE gerado pelo primeiro exemplo

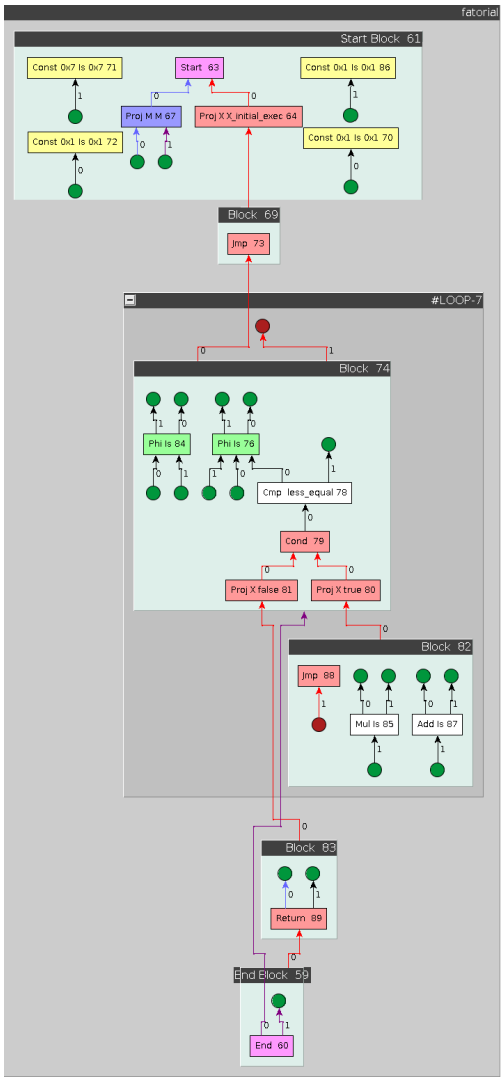

Figura A.1: GDE gerado pelo primeiro exemplo

APÊNDICE

B

GDE gerado pelo segundo exemplo

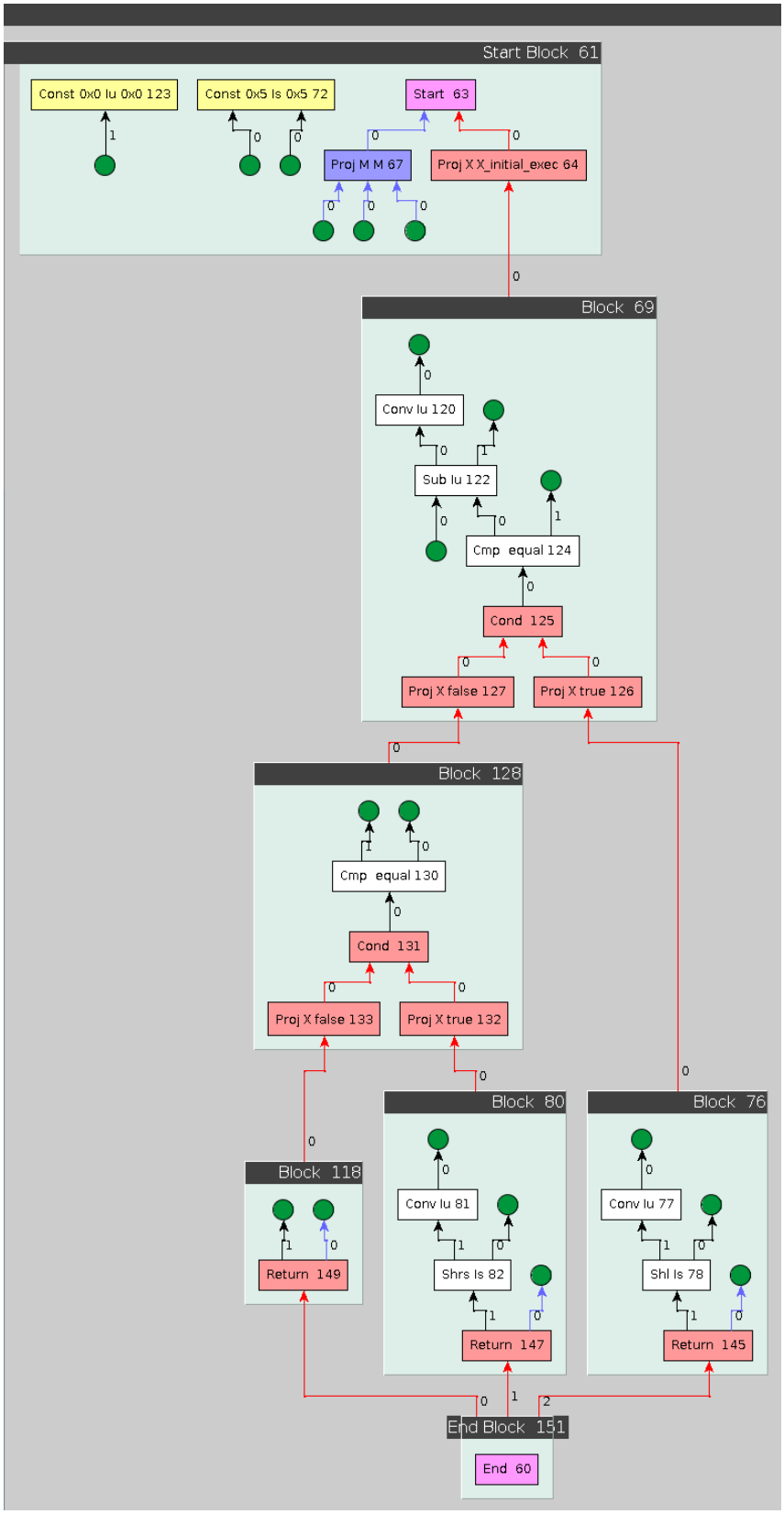

Figura B.1: GDE gerado pelo segundo exemplo