Universidade Federal do Rio Grande do Norte
Centro de Ciências Exatas e da Terra
Departamento de Informática e Matemática Aplicada
Bacharelado em Ciência da Computação

Uma plataforma de desenvolvimento de software baseada no LLVM para sistemas embarcados com processador RISCO

Giuliano de Souza Vilela Cid

Natal-RN
Dezembro de 2010

Giuliano de Souza Vilela Cid

Uma plataforma de desenvolvimento de software baseada no LLVM para sistemas embarcados com processador RISCO

Monografia de Graduação apresentada ao Departamento de Informática e Matemática Aplicada do Centro de Ciências Exatas e da Terra da Universidade Federal do Rio Grande do Norte como requisito parcial para a obtenção do grau de bacharel em Ciência da Computação.

Orientador(a)

Prof. Dr. Edgard de Faria Corrêa

Universidade Federal do Rio Grande do Norte – UFRN
Departamento de Informática e Matemática Aplicada – DIMAp

Natal-RN

Dezembro de 2010

Monografia de Graduação sob o título *Uma plataforma de desenvolvimento de software baseada no LLVM para sistemas embarcados com processador RISCO* apresentada por Giuliano de Souza Vilela Cid e aceita pelo Departamento de Informática e Matemática Aplicada do Centro de Ciências Exatas e da Terra da Universidade Federal do Rio Grande do Norte, sendo aprovada por todos os membros da banca examinadora abaixo especificada:

Prof. Dr. Edgard de Faria Corrêa
Orientador(a)
Departamento de Informática e Matemática Aplicada
Centro de Ciências Exatas e da Terra
Universidade Federal do Rio Grande do Norte

Prof. Dr. Márcio Eduardo Kreutz
Departamento de Informática e Matemática Aplicada
Centro de Ciências Exatas e da Terra
Universidade Federal do Rio Grande do Norte

Prof. Dr. Marcelo Ferreira Siqueira
Departamento de Informática e Matemática Aplicada
Centro de Ciências Exatas e da Terra
Universidade Federal do Rio Grande do Norte

Prof. Dr. Martin Alejandro Musicante
Departamento de Informática e Matemática Aplicada
Centro de Ciências Exatas e da Terra
Universidade Federal do Rio Grande do Norte

Natal-RN, 10 de Dezembro de 2010

Agradecimentos

Agradeço primeiramente a minha família, a qual sem o suporte dado ao longo do caminho nunca estaria aqui. Agradeço a meu pai e minha mãe por terem me dado todas as oportunidades que procurei.

Dedico este trabalho também aos meus amigos e colegas da universidade, os quais tornaram estes 4 anos de graduação uma experiência inesquecível. E também à namorada, pela paciência de santa nestes últimos meses.

Agradeço aos professores do DIMAp com quem tive a oportunidade de estudar por me darem a base para conseguir realizar este trabalho. Ao Prof. Edgard Corrêa e o Prof. Márcio Kreutz, cujas ideias e insights guiaram este trabalho e ao Prof. Marcelo Siqueira pela inspiração desta atração pelo estudo de compiladores e pela ajuda extra que sempre proveu em outros assuntos.

Computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty. A programmer who subconsciously views himself as an artist will enjoy what he does and will do it better.

Donald Ervin Knuth. *Computer Programming as an Art*, 1974.

Uma plataforma de desenvolvimento de software baseada no LLVM para sistemas embarcados com processador RISCO

Autor: Giuliano de Souza Vilela Cid
Orientador(a): Prof. Dr. Edgard de Faria Corrêa

Resumo

Este trabalho descreve o projeto e a implementação de um sistema de compilação e análise de código para sistemas embarcados contendo o processador RISCO, utilizando o projeto LLVM. Sistemas embutidos em um componente maior constituem a maioria dos sistemas computacionais em uso atualmente, sendo empregados nos mais diversos equipamentos. Devido a sua natureza, um sistema embarcado apresenta restrições de eficiência, consumo e tamanho, entre outras, que introduzem desafios únicos ao seu projeto. Nesse contexto, o processador RISCO, da família RISC e semelhante a arquitetura MIPS, nasceu como uma tentativa de desenvolver um processador simples, eficiente e que possa ser uma alternativa real às opções comerciais na sua faixa de preço. A plataforma de desenvolvimento descrita aqui habilita o desenvolvimento, a simulação e a análise de software em C e C++ para a plataforma RISCO com um conjunto de ferramentas de código aberto. Este trabalho discute as decisões de projeto envolvidas e os resultados obtidos com as ferramentas.

Palavras-chave: processador RISCO, LLVM, compilador, montador, simulador, análise estática, sistemas embarcados

A LLVM based development environment for embedded systems software targeting the RISCO processor

Author: Giuliano de Souza Vilela Cid
Advisor: Prof. Dr. Edgard de Faria Corrêa

Abstract

This work describes the design and implementation of a compilation and code analysis toolchain for embedded systems software targeting the RISCO processor, using the LLVM project. Small systems embedded in a larger device are by far the most common kind of computational system in use today, deployed in various types of equipments. Because of their nature, an embedded system presents interesting size, efficiency and energy consumption restrictions, among others, that impose unique challenges on a project. In that cenario, the RISCO processor, a RISC architecture similar to MIPS, was created as a simple, efficient, processor that could prove to be a practical alternative to the available commercial options in its price range. The toolchain we developed permit the development, simulation and analysis of software in C and C++ for the RISCO platform, with open source tools. This work discusses the design decisions involved in the development of the compilation and analysis system, and the results obtained testing the tools.

Keywords: RISCO processor, LLVM, compiler, assembler, simulator, static analysis, embedded systems

Lista de figuras

Lista de tabelas

Lista de abreviaturas e siglas

VLIW – Very Large Instruction Word

RISC – Reduced Instruction Set Computer

LLVM – Low Level Virtual Machine

SE – Sistema Embarcado

IP – Intellectual Property

UCP – Unidade Central de Processamento

ASIC – Application-specific integrated circuit

FPGA – Field-programmable gate array

E/S – Entrada / Saída

CMOS – Complementary metal–oxide semiconductor

PSW – Processor Status Word

PC – Program Counter

SP – Stack Pointer

RI – Representação Intermediária

SSA – Static Single Assignment

SIMD – Single Instruction, Multiple Data

DAG – Directed Acyclic Graph

DDT – Desenvolvimento Dirigido a Testes

WCET – Worst Case Execution Time

GCF – Grafo de Controle de Fluxo

Lista de símbolos

ϕ – função Phi, utilizada em códigos na forma SSA

$u \gg v$ – u domina v (em um grafo de controle de fluxo)

Lista de algoritmos

Sumário

1 Introdução

No século XX, o mundo presenciou imensos avanços na ciência e no modo como entendemos as máquinas, avanços que mudaram o então presente e marcaram o futuro como sendo uma era de extremo desenvolvimento tecnológico. Em torno de 1902, já existiam máquinas que trabalhavam com cartões perfurados, as precursoras do que em 1940 começou a ser chamado de computador (PIGUET, 1995). Desde então, vimos que a eletrônica vem evoluindo com uma constância incrível, modificando a natureza dessas máquinas.

Hoje em dia, sistemas computacionais fazem parte do cotidiano de virtualmente todas as pessoas. Direta ou indiretamente, interagimos ou dependemos de dispositivos digitais, controlados por microprocessadores. Dependemos dos seus componentes de hardware e de software para realizar tarefas essenciais. Como exemplo disto temos sistemas de controle de tráfego, gerenciamento de rede elétrica, computadores, celulares, etc. O estudo do projeto e desenvolvimento de sistemas computacionais é um assunto muito abrangente, envolvendo diversas disciplinas da Ciência da Computação. Envolve áreas de pesquisa bem estabelecidas, cujos resultados são empregados quase imediatamente no mercado mundial. As grandes empresas multinacionais, com seus interesses comerciais, forçam a pesquisa em direção ao estudo de soluções para problemas reais da atualidade. Os resultados da academia tem o potencial de otimizarem algum aspecto da produção de um produto comercial, trazendo maiores lucros para a empresa. A maioria desses sistemas recebe a denominação de *sistema embarcado*. Trata-se de um sistema computacional embutido em um dispositivo eletrônico, apresentando restrições severas de eficiência, tamanho e consumo, de modo a ser uma opção viável comercialmente. Um dos componentes que é projetado sob essas restrições é o *processador*.

O processador é o componente central de qualquer sistema computacional, agregando os componentes físicos responsáveis pela execução da lógica do sistema, propriamente dita, em forma de software. Os atributos desses sistemas, tais como tamanho, eficiência e consumo de energia, precisam ser cuidadosamente considerados no projeto de um sistema embarcado, analisando-se a interação deles com os requisitos originais do sistema. Em

geral, processadores para sistemas embarcados não podem ser tão complexos como os processadores de uso geral comumente utilizados em computadores de mesa, além de ter que apresentar um custo competitivo.

Neste âmbito de processadores para sistemas embarcados, existem diversas opções de microprocessadores largamente utilizados no mercado. Devido a multitude de aplicações deste tipo de sistema, e ao contrário do que é observado no mundo dos computadores pessoais, que é dominado pelos processadores x86, diversos modelos de arquitetura básica são utilizados, com destaque a arquiteturas RISC, de propósito mais geral, e as arquiteturas SIMD e VLIW, com um nicho em aplicações de processamento digital de sinais e/ou processamento paralelo de dados com alto desempenho.

Os processadores baseados em arquiteturas RISC são uma presença dominante. Exemplos deles são: o SparcV8 (SPARC, 1993), o MIPS32 (HENNESSY et al., 1981) e a família de processadores ARM (RYZHYK, 2006). Uma alternativa nacional, nesta categoria, foi criada por Junqueira e Suzim (1993), denominada *RISCO*. Trata-se de um processador RISC simples e eficiente, capaz de competir com outras opções comerciais na sua faixa de preço devido ao seu custo-benefício. Apesar da sua maturidade, o processador RISCO ainda apresentava uma deficiência comum aos projetos que nascem no mundo acadêmico: a falta de ferramentas completas e robustas, com qualidade para competir no mercado, que dêem suporte ao desenvolvimento de software em linguagens de alto nível. A existência desse suporte apresentaria inúmeras vantagens: facilitaria o aprendizado da arquitetura para os estudantes, facilitaria o desenvolvimento de software para este processador e ainda permitiria que os mais diversos softwares para sistemas embarcados, por exemplo um sistema operacional de tempo real, fossem modificados e portados para uma nova arquitetura que utilize o RISCO.

Este trabalho visa apresentar os detalhes da concepção, do projeto, do desenvolvimento e dos testes de um conjunto de ferramentas que possibilitam a compilação de software nas linguagens C (KERNIGHAN; RITCHIE, 1988) e C++ (STROUSTRUP, 1986) tendo como alvo o processador RISCO, e com o objetivo de gerar código otimizado com relação às restrições comuns para sistemas embarcados. Além disso, é apresentada uma ferramenta de análise estática específica ao código de máquina RISCO, com funcionalidades para verificação de custos de instruções em termos de blocos básicos e análise de tempo de execução para o pior caso. Discutimos as opções que foram consideradas para o rumo do desenvolvimento das ferramentas, as dificuldades encontradas para a sua conclusão e os resultados obtidos.

1.1 Organização do documento

O restante deste trabalho está organizado da forma que segue. O capítulo 2 trata de definir e apresentar o conceito de sistema embarcado, com diversos exemplos e aspectos que são relativos ao trabalho, e também uma visão geral do papel do software nesses sistemas. Em seguida, no capítulo 3, apresentamos uma descrição do projeto do processador RISCO e uma discussão breve da motivação por trás do desenvolvimento deste trabalho voltado especificamente para este processador, e da necessidade destas ferramentas.

O capítulo 4 é o primeiro que apresenta ferramentas desenvolvidas no trabalho. Elas constituem a plataforma base para a compilação e execução de software escrito na linguagem de montagem do RISCO.

O capítulo 5 apresenta o projeto LLVM do ponto de vista dos desenvolvedores de tecnologia para compiladores. É abordado o seu surgimento recente, seu prestígio e importância atuais, as vantagens e desvantagens da sua utilização no contexto deste trabalho, o detalhamento dos seus módulos e como eles interagem. A linguagem intermediária utilizada no LLVM, que será o foco principal do desenvolvimento do módulo RISCO, é tratada no capítulo 6.

O capítulo 7 apresenta uma discussão em torno do código gerado a partir do módulo RISCO-LLVM, tal como manipular certas características do código através de opções do compilador. Além disso, é detalhada a ferramenta de análise estática RISCO-CFG. Finalmente, no capítulo 8, algumas conclusões sobre o trabalho são delineadas, e são apontadas possíveis direções para trabalhos futuros.

2 Sistemas Embarcados

No início do uso do computador como um dispositivo produzido em escala moderada, em torno de 1940 e 1950, as máquinas eram dedicadas a uma única tarefa, mas não como as conhecemos hoje (PIGUET, 1995). Os computadores eram muito grandes, e muito caros, para serem utilizados no dia a dia. Desde essa época, a indústria eletrônica avança e inova constantemente, diminuindo o tamanho dos dispositivos, mantendo preços de acordo e adicionando sempre novas funcionalidades, impulsionando o mercado e a pesquisa em novas direções.

No mundo atual, interagimos com dispositivos eletrônicos em todo lugar e a todo momento: em casa, no escritório, na rua, no carro, em lojas. Comumente, mesmo sem o conhecimento do homem, o uso de sistemas automatizados abrange todos os aspectos do cotidiano moderno. A cada momento, é comum dependermos de um ou mais dispositivos para executar uma certa tarefa. A partir do momento que o uso de tais aparelhos se torna necessário ou tem um papel importante para, por exemplo, a segurança de um ser humano, é preciso um planejamento detalhado e cuidadoso das suas características, que precisam ser bem conhecidas e especificadas. Tais necessidades fazem com que parte do estudo de sistemas computacionais seja voltada a esta classe em particular.

Hoje em dia um sistema computacional por si só não apresenta utilidade, porém ele pode ser inserido em um contexto que o faça útil para o homem, facilitando a realização de alguma tarefa em particular ou habilitando-o a participar de novos modos de interação com outros indivíduos. Existe uma disciplina da Ciência da Computação que estuda este fenômeno, denominada de *computação pervasiva* ou *computação ubíqua* (ARAUJO, 2003). Os pesquisadores da área acreditam em um futuro *pós-desktop*, onde o ser humano se depara com dezenas de dispositivos computacionais a todo momento, todos trabalhando em sincronia, unidos por alguma forma de rede auto adaptativa, com boas garantias de eficiência e qualidade de serviço. Neste modelo, o computador tradicional como o conhecemos hoje irá perder lugar para que a sua funcionalidade seja distribuída entre vários componentes. Entretanto, não é preciso ser um futurista para observar que o modo como

utilizamos um dispositivo computacional mudou drasticamente já nas últimas décadas. De mainframes corporativos, capazes de ocupar diversos andares de edifícios, passamos a utilizar equipamentos cada vez mais diminutos, com poder e eficiência que ultrapassam os mais antigos por ordens de magnitude. Alguns exemplos disto são os desktops atuais, notebooks, netbooks e tablets.

Ao pensar em "sistema computacional", a grande parte das pessoas irá associar o termo ao computador propriamente dito, seja ele um desktop ou não. O fato é que esta concepção, herdada historicamente pelo modo como tais sistemas foram introduzidos na sociedade, é errada. A grande maioria dos sistemas computacionais em uso hoje em dia não são computadores. *Sistema Embarcado* (SE) é a denominação dada para um sistema computacional projetado desde a sua concepção para a realização de um pequeno conjunto de tarefas simples e bem definidas, para as quais foi programado, possivelmente com diversas restrições de tempo real (VAHID; GIVARGIS, 2001). Em geral, trata-se de um equipamento que é *embutido* em um dispositivo eletrônico maior, que o complementa em um sistema de grande porte, servindo para controlar os mais diversos dispositivos com os quais interagimos.

Existem diversos exemplos de sistemas embarcados projetados para uso doméstico. Celulares, *smartphones*, consoles de vídeo game, tocadores de MP3, câmeras de vídeo e/ou fotográficas, impressoras, televisões e ar condicionados são exemplos de aparelhos projetados para o conforto e entretenimento dos seus usuários finais. Outros são designados a tarefas mais críticas, tais como os sistemas de bordo em carros e aviões. Estes envolvem, entre outros, controle de freios, mistura de combustível, regulação do motor e controle da transmissão. Um carro popular comum produzido recentemente apresenta dezenas de dispositivos internos que contém um elemento computacional.

Os sistemas embarcados apresentam, em geral, algumas características em comum, embora haja diversas exceções à regra. É interessante catalogar estas características de modo a entender melhor as necessidades destes sistemas. Abaixo, apresentamos uma caracterização de sistemas embarcados baseada na que foi apresentada em (VAHID; GIVARGIS, 2001):

- **Função singular**: Uma única funcionalidade, ou um propósito único e bem definido. Em geral, isso significa que o dispositivo executa um único programa indefinidamente, com um pequeno conjunto de funcionalidades de baixa complexidade. Embora as tarefas que o SE precise realizar sejam simples, ele, em geral, é composto por um processador capaz de executar software arbitrariamente complexo, caso haja

necessidade. Como veremos a frente, devido a outros motivos, isto nem sempre é desejável.

- **Restrições não-funcionais**: Conjunto de restrições aplicadas a esses dispositivos devido ao seu contexto de uso, tais como pequeno volume, baixo consumo elétrico, alta responsividade e baixa latência. Algumas destas restrições são cruciais para a funcionalidade do dispositivo, enquanto outras agregam valor do ponto de vista comercial.

- **Otimização**: Por ter como propósito poucas tarefas, o sistema embarcado pode ser otimizado especialmente para a execução delas. Em geral, isto resulta em um projeto menor e mais conciso, especialmente criado para que o dispositivo execute essas tarefas da forma mais eficiente possível.

- **Reatividade**: O dispositivo apresenta sensores e uma interface com o ambiente bem definida, reagindo continuamente a mudanças em variáveis externas. Exemplos de sistemas embarcados em que a reatividade é uma característica dominante são sistemas de controle industrial.

- **Tempo real**: É comum tais dispositivos estarem embarcados em ambientes onde os requisitos temporais das suas tarefas sejam críticos. Isto é, uma tarefa tem o seu sucesso diretamente relacionado com o cumprimento ou não de certos limites de tempo. Cada sistema tem requisitos distintos, onde estes limites de tempo (*deadlines*) são mais rígidos ou não.

- **Interface básica**: Os primeiros sistemas embarcados eram mais simples e, em geral, não apresentavam uma interface de interação com o usuário. Com o avanço das tecnologias utilizadas e da eficiência dos sistemas, os sistemas foram incorporando esta funcionalidade. Um exemplo deste processo natural é a recente utilização de telas sensíveis ao toque como alternativas aos projetos mais tradicionais que utilizavam teclados clássicos. O uso de tais telas requer hardware mais complexo, e mais custoso, além de adaptações do software que o controla.

A indústria de sistemas embarcados, extremamente diversa, tem um horizonte amplo de possibilidades no presente e no futuro. Todo dia novos produtos que incluem sistemas embarcados são projetados e, em um futuro próximo, começam a ser produzidos. O preço do hardware utilizado nesses dispositivos diminui com o tempo, com a evolução natural das tecnologias utilizadas, possibilitando que novas aplicações, antes impossíveis, sejam

consideradas. Sistemas embarcados são utilizados em casa, no trabalho, em automóveis, lugares públicos e hospitais, com as mais diversas funções. O estudo sistemático desses sistemas, assim como das técnicas de otimização que lhe podem ser aplicadas, faz-se obviamente necessário.

2.1 Restrições

O projeto de sistemas embarcados está fortemente ligado às tendências atuais da indústria. Devido ao fato de estarem predominantemente embarcados em outros dispositivos eletrônicos, estes sistemas estão sujeitos as restrições impostas pelo projeto de tais dispositivos. Estas restrições podem ser estritamente técnicas ou não.

Tais restrições podem ser observadas mais praticamente tomando como exemplo um aparelho celular inteligente (*smartphone*) de última geração, contendo uma interface de iteração com o usuário baseada em uma tela sensível ao toque (*touchscreen*). Este sistema pode ser identificado como um sistema embarcado utilizando a classificação que foi apresentada acima:

- *Função singular*: Como observado, alguns sistemas não apresentam esta característica e o celular smartphone moderno é um deles. Este aparelho é programado para realizar diversas funções desde a saída da fábrica, sendo ainda extensível através da instalação de programas. Ainda mais, é comum que estes celulares executem sistemas operacionais de propósito geral. Um kernel Linux modificado para atender as restrições não-funcionais do aparelho, por exemplo.

- *Restrições não-funcionais*: O aparelho precisa ser pequeno, com boas propriedades ergonômicas, assim como consumir pouca energia, de modo a conservar a sua bateria e aumentar o conforto do usuário. Otimização: Apesar de executar programas arbitrários, um smartphone é equipado com software e hardware otimizado para as suas tarefas principais.

- *Reatividade*: Além de, obviamente, observar as iterações com a interface com o usuário, o aparelho está constantemente em contato com uma antena celular próxima, e possivelmente com pontos de acesso para redes Wi-Fi. Eventos que acontecem nesses pontos de interface com o ambiente externo determinam o funcionamento do aparelho.

- *Tempo-real*: O celular é sujeito a restrições temporais de intensidade variável de acordo com o estado atual. Por exemplo, é preciso apresentar uma boa responsividade quando o usuário interage com o menu ou está executando algum programa interativo (ex: um jogo). Porém, é mais importante atender os requisitos temporais associados a codificação e decodificação do áudio em uma ligação telefônica.

Estes aspectos introduzem restrições técnicas ao projeto do sistema embarcado que acompanha um celular smartphone comum. Pode-se citar, por exemplo, o tamanho do sistema, que deve ser uma das principais preocupações em tempo de projeto. Isto implica a preferência por componentes de pequena área. Além disso, tais componentes precisam ter uma eficiência suficiente para que a execução do software do smartphone seja satisfatória. Isto é, tenha uma boa performance na iteração com o usuário. Outro aspecto para o conforto geral do usuário final do produto é uma boa autonomia da sua bateria, uma característica que é diretamente influenciada pelo hardware utilizado e, de certa maneira, pelo software também (WILLIAMS; CURTIS, 2008).

2.1.1 Influência do mercado

Além das restrições técnicas que são impostas aos sistemas embarcados, existem também as restrições de mercado. Os produtos em que os sistemas embarcados são incluídos devem ser projetados de modo a atender satisfatoriamente os requisitos do usuário final. Nas empresas de ponta, que trabalham com o estado da arte em suas respectivas áreas e estão sempre liberando produtos inovadores no mercado, as limitações das tecnologias sendo utilizadas em um dado momento não influenciam decisivamente o projeto final. Pelo contrário, as necessidades do usuário final ditam o rumo da inovação tecnológica, e tais empresas apresentam a necessidade de investir em novas tecnologias.

Estas restrições são capazes de mudar o rumo de um projeto, afetando algumas decisões mesmo que não haja motivo técnico. São restrições baseadas em valores de mercado, como lucro, tempo de lançamento do produto no mercado, tempo de projeto, custo de projeto e custo unitário, entre outros. Vahid e Givargis (2001) mostram que é preciso considerar e otimizar diversas métricas de projeto de acordo com decisões tanto administrativas quanto financeiras. Algumas destas métricas são:

- **Custo de unidade**: custo da produção de uma unidade do produto, contando materiais e mão de obra, excluindo custos de projeto. É o custo do processo de fabricação em si.

- **Custo de engenharia**: custo expendido durante todo o projeto do sistema embarcado, não influenciando o valor do custo de unidade. Inclui, por exemplo, o capital investido em pesquisa e desenvolvimento do primeiro protótipo.
- **Tempo para o protótipo**: a quantidade de tempo necessária para que um primeiro protótipo funcional do sistema embarcado seja produzido.
- **Tempo para o mercado**: a quantidade de tempo necessária para que a empresa esteja pronta para a produção do sistema em larga escala, para o mercado.
- **Sucesso do produto**: é uma medida um tanto subjetiva, que pode ser descrita de maneiras diferentes. Por exemplo, como o lucro da empresa com este produto, ou o nível de aceitação dele pelo público-alvo.

As relações entre estas 5 métricas são bastante subjetivas, e variam de projeto para projeto. Entretanto algumas correspondências são válidas para a maioria dos casos. Um maior investimento em pesquisa acarreta um maior tempo para o protótipo e custo de engenharia, porém tem o potencial de diminuir o custo de unidade e aumentar o sucesso do produto.

A otimização destas métricas em termos de minimização ou maximização afeta, entre outras coisas, a escolha dos componentes de hardware de um sistema embarcado, assim como a reutilização de código no seu software. O projeto de um sistema embarcado tradicional apresenta um imenso espaço de decisões. Isto é, inúmeras escolhas são apresentadas aos projetistas, que devem avaliar suas vantagens e desvantagens, os compromissos (trade-offs) envolvidos, e decidirem o caminho a ser escolhido, levando em conta os objetivos da gerência com relação às métricas apresentadas acima.

Uma tendência clara que pode ser observada, exemplificada em (FILIPPI et al., 1998), são os esforços empregados na compra e reuso de componentes em forma de propriedade intelectual (IP, do inglês *intellectual property*) de terceiros. O uso de componentes pré-projetados, corretos e testados diminui significantemente o tempo para o protótipo e o tempo para o projeto. De acordo com o seu preço, o uso de um componente projetado por terceiros tem o potencial de otimizar também o custo de engenharia e de unidade. Além disso, a facilidade de incorporação de IP's em um projeto pode ajudar a flexibilizá-lo, agilizando as mudanças e adaptações do projeto com relação às mudanças constantes de requisitos no mundo real. Como veremos no capítulo 3, o microprocessador de um sistema embarcado é um forte candidato a ser adquirido via IP.

2.2 Software para Sistemas Embarcados

Para exercitar os diversos tipos de tarefas mencionados anteriormente, os sistemas embarcados precisam de uma unidade de processamento central (UCP), semelhante a de um computador pessoal tradicional. As diferenças estão relacionadas com as restrições explicitadas na seção 2.1, fazendo com que haja um nicho específico de processadores para sistemas embarcados.

Em geral, existem algumas direções distintas para o projeto de uma UCP para um sistema embarcado: utilizar um microprocessador "de prateleira", um microcontrolador, contendo alguns periféricos em sua placa, um processador customizado ASIC (do inglês *application-specific integrated circuit*), ou até um sistema proprietário desenvolvido utilizando a tecnologia FPGA (do inglês *field-programmable gate array*). Cada opção tem suas vantagens e desvantagens e vários compromissos (*tradeoffs*) estão envolvidos, assuntos que fogem do escopo deste trabalho.

O software presente em sistemas embarcados, que será executado por essas UCP's, apresenta diferenças importantes em relação ao software comum desenvolvido para dispositivos mais potentes e com menos imposições técnicas. É interessante notar que as restrições técnicas dos sistemas embarcados afetam tanto o software produzido como os processos e ferramentas utilizados no seu desenvolvimento.

É possível delinear um esquema básico do processo seguido pelos programadores deste tipo de software. Primeiramente, é criada uma versão inicial do programa (protótipo do firmware) do SE, onde a funcionalidade básica está presente, porém o programa ainda não está totalmente otimizado para as tarefas em questão e algumas funções que necessitam de uma interface de baixo nível com o hardware da plataforma do sistema embarcado ainda não estão implementadas. Em iterações bem definidas, a equipe desenvolve otimizações sob o código original, possivelmente reescrevendo algumas partes em linguagem de montagem (*assembly*) para melhorar a performance da aplicação ou introduzir alguma funcionalidade que necessite de E/S avançado em baixo nível.

Este tipo de cenário onde o programador precisa conhecer detalhes de baixo nível sobre o hardware do SE é muito comum neste âmbito, embora isto não seja necessariamente desejável. Ao desenvolver software escrito em linguagem de montagem, o programador precisa atentar a diversas questões que em outras situações são totalmente transparentes:

- *Conjunto de instruções do processador*: estas instruções funcionam como a "lingua-

gem de programação" utilizada e definem toda a semântica do código em linguagem de montagem. O programador precisa conhecê-las em detalhe, de modo a poder usufruir da funcionalidade completa que o processador lhe oferece.

- *Organização da memória*: qual o tipo da memória, o seu tamanho e como é dividida. Isto é, detalhes sobre a hierarquia de memória presente na arquitetura (HENNESSY; PATTERSON, 2003). Exemplos: detalhes dos segmentos e da técnica de paginação utilizada, quantidade de níveis de cache e seus tamanhos, para a arquitetura em questão.

- *Registradores*: quais os registradores o processador disponibiliza e quais as suas funções e restrições.

- *E/S*: quais são as convenções utilizadas para realizar entrada e saída, tais como memória mapeada e acesso direto. Quais são os passos necessários para que um programa se comunique com um dispositivo externo, por exemplo.

Tanto para o protótipo como para as suas versões posteriores, é vantajoso utilizar uma linguagem de programação de médio nível de abstração, como por exemplo C, sempre que possível. Tal linguagem permite que o código seja escrito em uma linguagem com boas abstrações de software, porém com flexibilidade suficiente para fazer uma interface direta com o hardware caso seja necessário, utilizando-se de trechos de linguagem de montagem em meio ao código. A equipe também tem a opção de utilizar um sistema operacional completo, capaz de simplificar algumas destas tarefas.

Em geral, é um consenso que as linguagens de programação estruturadas são um avanço importante com relação a linguagens de assembly puro, com vantagens significativas no contexto dos sistemas embarcados. O uso destas linguagens é capaz de cortar vários custos do projeto como um todo, seja em tempo de desenvolvimento ou até em tempo de manutenção do software. A tendência é que, com a evolução do hardware utilizado nos sistemas embarcados, e a diminuição dos seus custos, o software desenvolvido seja mais alto nível, de modo a obter uma maior competitividade nos tempos de lançamento do produto no mercado. Tradicionalmente, a linguagem C (seguida de perto por C++), junto as linguagens de montagem das diferentes plataformas, dominam o nicho dos SE's, porém há esforços para viabilizar a utilização de linguagens de mais alto nível, como Java (HIGUERA-TOLEDANO; ISSARNY, 2000; CLAUSEN et al., 2000). Esta tendência é um ponto importante na argumentação da relevância do trabalho apresentado aqui.

2.2.1 Processos de compilação e evolução do software

Para melhor entender os detalhes de como acontece a construção, compilação e execução de software para sistemas embarcados, iremos apresentar uma visão geral dos processos e ferramentas envolvidos.

Um possível ambiente de desenvolvimento de software para SE's, que possibilite a construção de software que esteja de acordo com os requisitos explicitados anteriormente, requer uma composição de ferramentas que trabalham em conjunto para prover esse suporte. Para identificar as ferramentas necessárias, analisaremos dois fluxogramas tradicionais de desenvolvimento para um sistema embarcado, em um projeto que utiliza um misto de código em linguagem de alto nível (neste caso, C) e em linguagem de montagem. O primeiro diz respeito ao processo de compilação de código, enquanto o segundo exemplifica o processo de execução, análise, depuração e otimização do programa.

A figura 1 mostra o modelo tradicional de compilação de software, uma extensão do fluxograma proposto em (VAHID; GIVARGIS, 2001). Convencionamos neste exemplo em particular que o projeto apresentado é constituído por 2 códigos-fonte C, `A.c` e `B.c`, e 1 arquivo de código fonte assembly, `C.asm`. O arquivo `B.c` pode ser uma biblioteca de código reutilizável com certa funcionalidade requisitada pelo programa principal, contido em `A.c`. A unidade `C.asm` pode conter implementações em linguagem de montagem do conjunto de funções que necessitam de acesso direto ao hardware.

A primeira consideração a ser feita é que este processo utiliza a técnica de *compilação cruzada*. O desenvolvedor cria o código fonte e o compila em uma máquina host, possivelmente um computador pessoal comum. O arquivo binário final gerado, entretanto, contêm instruções correspondentes ao processador alvo do sistema embarcado. Esta é uma abordagem bastante comum. Nestes casos, diz-se que o compilador é um *compilador cruzado* (do inglês *cross-compiler*). Esta classificação é somente didática, pois um compilador cruzado não apresenta diferenças técnicas com relação a um compilador "normal".

Seguindo a numeração da figura 1, o processo é apresentado a seguir. O programador cria, com um editor de texto comum, os arquivos de código fonte (1). Um compilador C é executado (2), tendo como entrada os códigos fonte em C. Ele é responsável pela tradução do código nesta linguagem para uma representação mais próxima do código binário interpretado pela máquina, a linguagem de montagem.

Em seguida, o *montador* é executado (3), tendo como entrada os arquivos de código fonte em linguagem de montagem originais e os gerados pelo compilador. Ele produz um

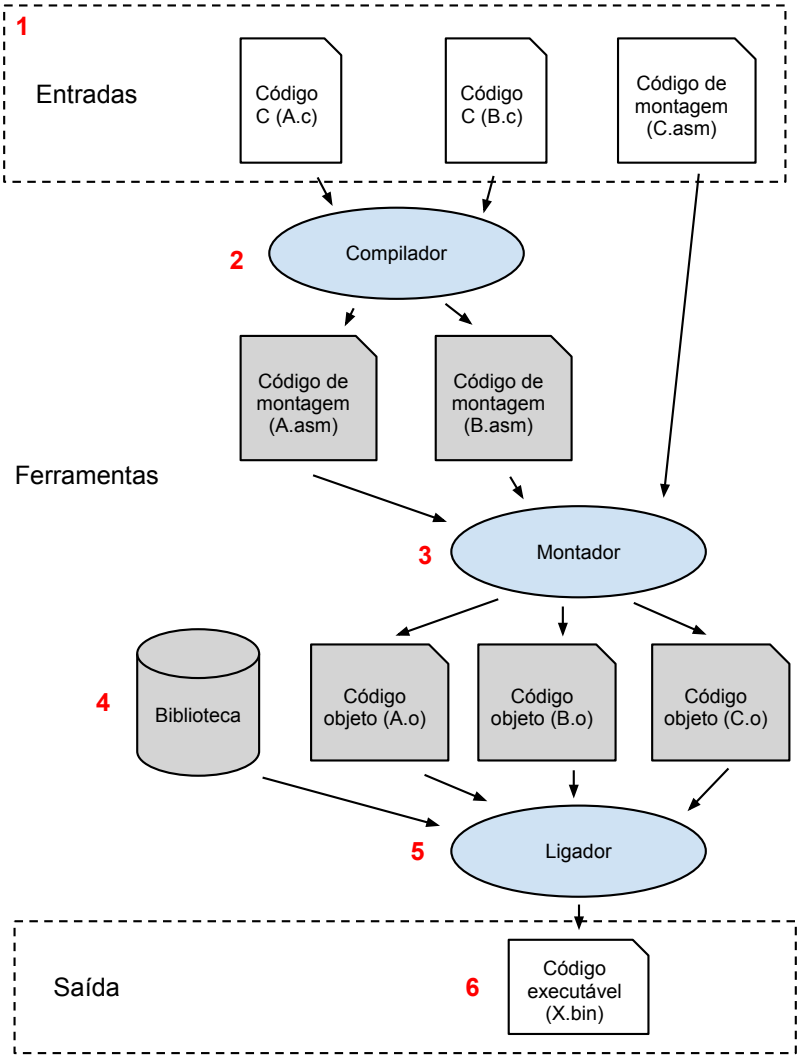


Figura 1: Fluxo tradicional de compilação de software misto C e assembly

arquivo em formato de código binário relocável. Neste momento (4), há um conjunto de arquivos independentes com código binário, cada um constituindo uma *unidade de compilação* diferente. Além dos arquivos originais que o programador usou como entrada, ainda são utilizados os códigos binários da biblioteca padrão da linguagem, além de outras externas que possam ser utilizadas. Logo após, o *ligador* é executado (5), tendo como entrada todos os arquivos de código binário. Neste instante, as referências a símbolos externos presentes em cada unidade de código são resolvidas, e um único arquivo binário é criado, contendo o executável final. Finalmente, o arquivo executável final está pronto (6), e pode ser utilizado para execução do programa.

A cada protótipo do sofware que é finalizado, a equipe entra em um ciclo de teste, análise, depuração e otimização de código, como apresentado na figura 2, baseada nos conceitos em (HENNESSY; PATTERSON, 2003).

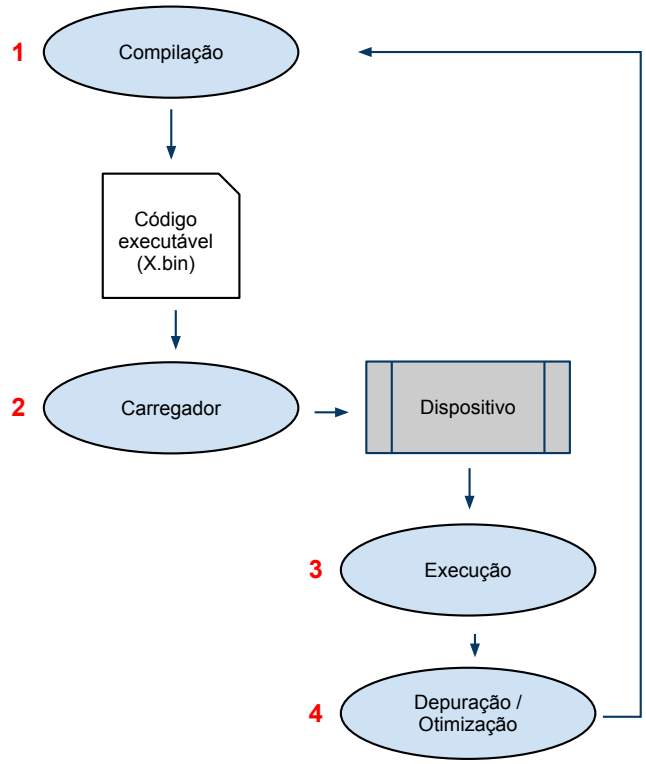


Figura 2: Fluxo tradicional de execução, depuração e otimização de software

O fluxo de atividades é detalhado a seguir. Primeiramente, a versão atual do código fonte é compilada (1), através do processo apresentado anteriormente, e o código binário executável é gerado novamente. Um programa denominado *carregador* é executado (2), tendo como entrada o arquivo executável. Este programa é responsável por carregar a imagem estática do programa na memória do dispositivo, preparando-o para sua execução.

Note que a definição apresentada para o carregador é intencionalmente vaga. Este poderia ser, por exemplo, um módulo de um sistema operacional de propósito geral, ou até um programa que transfere o código binário através de uma porta serial para a memória secundária de um dispositivo eletrônico, como um micro-controlador.

Em seguida, o programa é executado em um dispositivo (3). Note que o dispositivo não é necessariamente o sistema embarcado em si. Pode ser um protótipo contendo somente alguns componentes necessários para a iteração atual, ou um software simulador da arquitetura do SE. Finalmente, os dados de saída do software (4), de acordo com as suas respectivas entradas, são utilizados para realizar correções no código fonte (depuração) ou modificações com o intuito de melhorar a eficiência do programa (otimização). Em ambos, caso alguma modificação seja necessária, o processo é reiniciado e o código fonte é recompilado.

As ferramentas apresentadas nesta seção são de caráter básico para o desenvolvimento de software para SE's. Mostramos no capítulo 3 as ferramentas que de fato foram implementadas neste trabalho, e como elas se inserem nos modelos acima.

2.2.2 Arquiteturas Comuns

Basicamente, o software que é executado em um sistema embarcado pode ser dividido em duas classes (NOERGAARD, 2005): *software do sistema básico* e *software de aplicação*. Software do sistema básico é todo código que tem como fim dar suporte as aplicações que o utilizam. Nesta categoria estão os drivers de dispositivo, os sistemas operacionais e os *middlewares* de suporte básico. O software de aplicação compõe a camada superior da pilha de software de um SE, ele define toda a funcionalidade de alto nível do sistema, utilizando os serviços das camadas inferiores. É o software que faz a interface com os usuários do sistema. Ao projetar um sistema embarcado deve-se atentar a definição detalhada de uma arquitetura de software que equilibre as vantagens e desvantagens de cada tipo de software. Por exemplo, software do sistema básico é comumente mais complexo e passível de erros do que o software de aplicação, porém tem acesso direto ao hardware da plataforma, podendo realizar operações com maior controle. A partir desta divisão, surgiram diversos padrões de arquiteturas de software para SE's (NOERGAARD, 2005). Alguns significativos são:

- **Loop simples**: Todo o software é software básico. O código é simplesmente um loop, chamando rotinas diversas, cada uma gerenciando uma parte do hardware.
- **Sistema baseado em interrupções**: Diferentes tarefas são acionadas a partir de interrupções recebidas pelo loop principal, cada uma realizando uma ação diferente.
- **Multiprocessamento cooperativo**: O programador define diversas tarefas, cada uma com seu próprio ambiente de execução. Elas cooperam entre si para dividir os recursos de hardware.
- **Multiprocessamento com preempção**: Geralmente conta com um sistema operacional completo como base. O código da aplicação usa o modelo de processos distintos, utilizando os serviços do software básico.

As ferramentas de desenvolvimento mencionadas em 2.2.1 devem ser completas o bastante para suportar o desenvolvimento de software pertencente a qualquer uma destas arquiteturas.

3 RISCO

Como foi observado no capítulo 2, o uso de componentes terceirizados é uma fase crucial do projeto de sistemas embarcados. A escolha dos componentes com as características certas influencia diversos aspectos do projeto. Tais componentes necessitam ser projetados tendo em vista todos os requisitos inerentes a esses sistemas. Em geral, têm no mínimo o seu tamanho, custo/benefício e consumo de energia otimizados. Consequentemente, um campo natural de pesquisa na academia é a especificação, projeto e construção de componentes de hardware ou software otimizados para sistemas embarcados.

Os trabalhos de Vahid e Gajski (2010) e Douglass (1997) mostram esforços em direção à padronização de técnicas de modelagem de sistemas embarcados. Existem também muitas opções de sistemas operacionais destinados a sistemas embarcados com requisitos de tempo real, tais como o QNX (QNX..., 1982) e o RT-Linux (AYERS; YODAIKEN, 1997). Além de componentes de software, o projeto de componentes de hardware para sistemas embarcados também é bastante explorado: Furber, Edwards e Garside (2000), por exemplo, apresentam um microprocessador projetado para SE's, baseado no MIPS. Como visto na seção 2.1.1, o uso de componentes terceirizados, pré-fabricados ou não, traz vantagens significativas ao projeto de um SE, tais como a diminuição do seu custo ou do tempo de protótipo ou de produção. Isto reforça a importância de se considerar atentamente a escolha de um processador para um SE. Em geral, existem 3 opções:

- É possível projetar o sistema por si próprio, com uma equipe de desenvolvimento interna. Isto é, desenvolver um processador dedicado à tarefa em questão, projetado do início ao fim. Isto resulta em um projeto otimizado, um processador de alta performance para a sua função designada. Entretanto, aumenta muito o tempo de projeto e de protótipo. Como vantagem, o seu custo de produção por unidade pode ser mais competitivo com relação às outras opções, caso um número suficiente de unidades sejam produzidas.
- Existem empresas que vendem o projeto de um processador sob a forma de IP:

em geral, uma especificação completa do processador em uma linguagem formal de descrição de hardware (ex: VHDL). Esta escolha aumenta o custo de projeto, porem não aumenta o seu tempo. Além disto, esta é uma opção flexível pois permite que a especificação do processador seja modificada para atender a certos requisitos específicos da aplicação de um SE.

- Por fim, a equipe pode optar por utilizar um processador de propósito geral "de prateleira". Isto é, um processador pré-fabricado. Esta opção não impacta o custo de projeto, porém em geral aumenta o custo total de uma unidade do produto. Além disto, o projeto terá que ser adaptado de acordo com a interface do processador escolhido.

Neste contexto, vê-se que uma especificação aberta de um microprocessador simples e eficiente para sistemas embarcados é uma boa opção. Pode-se utilizar a sua especificação em um projeto particular, obedecendo a restrições de licenciamento, obtendo as melhorias de tempo e flexibilidade mencionadas acima. Há a opção de utilizá-lo do modo como ele é disponibilizado, semelhantemente a utilização de um processador sob a forma de IP proprietária, ou pode-se modificar a especificação original para produzir um processador customizado, específico a uma determinada aplicação (ASIC). Na literatura existem diversos exemplos de desenvolvimento de especificações abertas de hardware (WEISS, 2008), incentivando a iniciativa comum sob a legenda de *Open Hardware* (OHF..., 2010).

O processador **RISCO** (JUNQUEIRA; SUZIM, 1993) foi criado pelo grupo de microeletrônica da UFRGS com o objetivo de adquirir um projeto próprio de processador, assim como uma aprofundação de conhecimentos sob o assunto. Entretanto, além disso, ele também se encaixa perfeitamente como um processador de SE, como foi detalhado em 3.1.

Nas seções a seguir são apresentados argumentos em relação à validade da utilização do RISCO no âmbito de sistemas embarcados e sistemas de tempo real, salientando as vantagens e desvantagens das decisões de projeto que foram tomadas no seu desenvolvimento.

3.1 Detalhes do projeto

O RISCO é um projeto antigo (1993) de um micro-processador RISC. Seu projeto teve como objetivo a simplicidade de implementação e sintetização, facilidade de simu-

lação e baixo consumo de energia. Tais características o fazem um bom candidato para implantação nos mais diversos sistemas embarcados.

Os processadores que seguem a filosofia RISC apresentam um conjunto de instruções menor, mais simples e altamente otimizado, em vez de um grande conjunto de instruções complexas para as mais diversas situações como nos CISC, podendo acarretar em um aumento de performance significativo. Algumas famílias de processadores RISC bem conhecidas no mercado atual são a Alpha AXP, ARM, MIPS, PowerPC e SPARC (DANDAMUDI, 2005).

Em geral, como discutido em (HENNESSY; PATTERSON, 2003; NOERGAARD, 2005), os processadores RISC necessitam de mais instruções do que máquinas CISC para executar uma mesma tarefa, isto é, apresentam uma menor densidade de código. Entretanto, os processadores RISC em si tem uma implementação mais simples e previsível, contendo somente instruções básicas, que cobrem 90% das instruções utilizadas por programas reais (JUNQUEIRA; SUZIM, 1993). O uso de poucas instruções, cada qual com uma funcionalidade não complexa, faz com que a execução de uma única instrução seja bastante eficiente. Isto é, o ciclo do relógio do processador pode ser reduzido. Além disso, a regularidade do conjunto de instruções de um processador RISC, aliado aos conceitos de ortogonalidade empregados, fazem com que seja possível a implementação de estágios de pipelines mais bem definidos e divididos, com um melhor comportamento quando comparados a um processador CISC.

Outras características de um processador RISC são: formato uniforme de instruções, resultando em um circuito de baixa complexidade para a sua decodificação; uso de registradores de propósito geral idênticos, podendo ser utilizados sem restrições em todas as instruções; somente modos de endereçamento simples, com operações mais complexas necessitando serem realizadas com mais de uma instrução; baixo consumo de energia devido a uma menor complexidade do circuito interno.

Devido a estas características, os processadores RISC são a preferência, hoje em dia, no projeto de sistemas embarcados. Em especial, o ótimo custo/benefício dos seus processadores, aliado a eficiência no consumo de energia são bastante atrativos para as empresas deste ramo. Os processadores RISC dominam quase que completamente o mercado de celulares e tablets. Devido a grande quantidade de tais dispositivos, a família RISC é largamente mais utilizada do que os processadores CISC (NOERGAARD, 2005) no conjunto de todos os dispositivos computacionais existentes, embora tenha uma baixa penetração no mercado de computadores pessoais, dominado pela família de processadores x86 (SHANLEY, 2010).

Atualmente, o RISCO é distribuído sob duas formas. É possível obter uma especificação do processador como uma modelagem em alto nível, utilizando a ferramenta SystemC (ARNOUT, 2000), ou como uma especificação sintetizável de hardware na linguagem VHDL (ASHENDEN, 2000). Ambas as modelagens já foram testadas em outros trabalhos. Algumas das características do RISCO são:

- Dados e instruções são palavras de 32 bits, e um byte da memória principal pode ser endereçado diretamente por uma palavra. Consequentemente, suporta a utilização de até 2^{32} bytes de memória, o equivalente a 4GB, quando programado em modo real. Em modo protegido, esta quantidade seria menor.
- Comunica-se com a memória através de um barramento multiplexado de 32 bits, utilizado tanto para dados como para endereços.
- Apresenta um pipeline de instruções tradicional RISC com 3 estágios: (i) decodificação e busca de operandos, (ii) realização da operação e (iii) escrita dos resultados.

O seu projeto simples corresponde a uma arquitetura RISC bastante eficiente e de baixo consumo. Isto o torna uma possível opção comercial viável para uso em sistemas embarcados. Os detalhes da arquitetura interna do RISCO, relativos a sua especificação oficial definida em (JUNQUEIRA; SUZIM, 1993) fogem o escopo deste trabalho. Entretanto, certos detalhes deste tipo são mencionados ao longo do texto, caso sejam diretamente relevantes para a definição da semântica das instruções do RISCO.

3.2 Conjunto de instruções

Seguindo as tradições de projetos RISC, o conjunto de instruções do RISCO é composto por instruções de até 3 endereços, sendo um destino e dois operandos. Os operandos são sempre constantes incluídas na palavra ou identificadores para um dos registradores da UCP. Apresenta 3 formatos de instruções diferentes, porém todos com o mesmo tamanho. As instruções são classificadas em: instruções lógicas e aritméticas, saltos, acesso a memória (no esquema *load* e *store*), e chamada de sub-rotinas. O projeto das instruções enforça os princípios já conhecidos de ortogonalidade e simetria das operações de processadores que seguem a filosofia RISC. Isto é, as instruções podem utilizar todos os tipos de dados e todos os modos de endereçamento.

Uma instrução RISCO é uma palavra de 32bits acessada na memória principal. Todos os 3 formatos de instrução seguem o modelo apresentado na figura 3. Os 8 bits que vão de

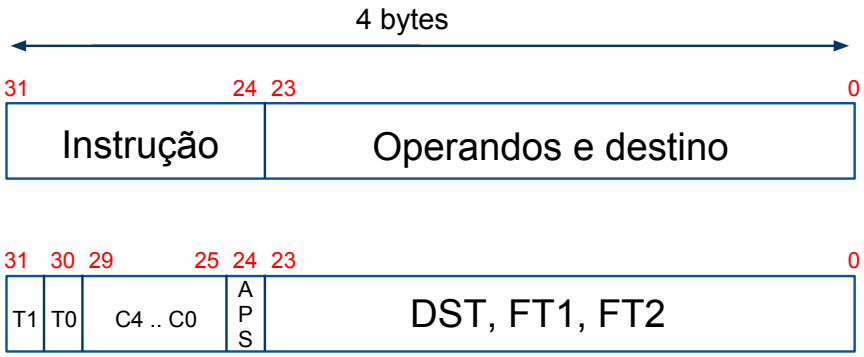


Figura 3: Formato base das instruções RISCO

24 a 31 da palavra identificam unicamente a instrução que será executada, junto com o *modificador de status*. Os bits restantes, da posição 0 a 23, indicam quais são os operandos e o destino da instrução, na tradição dos códigos de 3 endereços. A partir do campo dos operandos, o decodificador da UCP determina 3 valores, DST, FT1 e FT2, a serem utilizados pela instrução. DST sempre é um identificador de registrador, enquanto FT1 e FT2 podem ser registradores ou constantes, como veremos mais adiante. A instrução, portanto, pode ser vista como uma função de 3 parâmetros: `instrucao(DST, FT1, FT2)`.

Os campos T1 e T0 indicam o tipo da instrução, que pode ser: aritmético-lógica (00), salto (01), acesso a memória (10) ou chamada de sub-rotina (11). Os campos C4 a C0 indicam a instrução específica dentro do grupo T1-T0. O campo APS é o bit indicador da modificação de status. Caso seja 1, após a realização da operação indicada pela instrução, a UCP atualiza o seu *registrador de status*, PSW, baseando-se nos valores das operações.

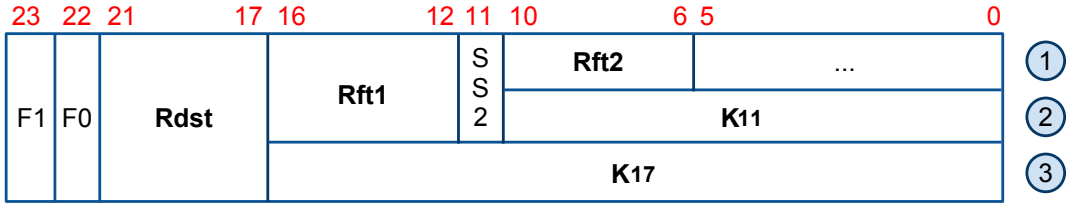


Figura 4: Os 3 formatos de identificação dos operandos RISCO

Os bits 0 a 23, que determinam os operandos e o destino da instrução, tem 3 formatos diferentes, conforme a figura 4. Nesta figura, os campos Rdst, Rft1 e Rft2, de 5 bits cada, indicam um dos registradores dentro dos 32 possíveis. K_{17} e K_{11} são constantes numéricas com sinal, de 17 e 11 bits, representadas em complemento de 2. A partir destes 3 formatos de campos na instrução, existem 5 interpretações diferentes para os valores DST, FT1 e FT2 mencionados acima, de acordo com a tabela 1.

Note que os 5 formatos são identificados a partir dos valores nos campos F1, F0 e

	DST	**FT1**	**FT2**	F1	F0	SS2
1	Rdst	Rft1	Rft2	0	0	0
2	Rdst	Rft1	K_{11}	0	0	1
3	Rdst	R0	K_{17}	0	1	x
4	Rdst	Rdst	$K_{17}[15:0]$ & $K_{17}[16]$ * 16	1	0	x
5	Rdst	Rdst	K_{17}	1	1	x

Tabela 1: Interpretação dos operandos RISCO

SS2. Em todos os 5, o parâmetro DST sempre é o registrador indicado por Rdst. Os casos interessantes são: no formato 3, FT1 é um parâmetro implícito sempre apontando para o registrador R0; nos formatos 4 e 5, FT1 é sempre Rdst; no formato 4, FT2 é a concatenação dos bits 0 a 15 de K_{17} com o seu bit mais significativo replicado 16 vezes. Variações do formato 4 são comuns em arquiteturas RISC, como o MIPS (HENNESSY; PATTERSON, 2003), onde são utilizadas para carregar uma sequência de bits nos bits mais significativos de um registrador, utilizando somente uma instrução.

Em geral, os 32 registradores do RISCO são idênticos. Os casos especiais são:

- **R00**: Sempre guarda o valor 0, e operações de escrita não tem efeito[1] quando ele é o destino.
- **R01**: Guarda a palavra de estado do processador, PSW. Contêm bits indicando propriedades do resultado da última operação que teve o bit APS setado, sendo eles: N (negativo), O (overflow), Z (zero) e C (carry). Não pode ser escrito por software.
- **R31**: Guarda o valor do contador de programa (PC). Pode ser escrito explicitamente por uma instrução, ou implicitamente em uma instrução de salto.

O apêndice A contém uma descrição do conjunto completo de instruções do processador RISCO. Aqui foram incluídos somente alguns comentários.

As decisões de projeto do RISCO com relação ao seu pipeline interno introduziram certas dificuldades para a programação do processador. Todas as instruções de salto e chamada de subrotina apresentam um atraso de 1 instrução. Isto é, a instrução logo após o salto sempre é executada, mesmo quando o salto for tomado. O programador deve

[1] O único efeito possível é a atualização do registrador PSW, caso o bit APS da instrução seja 1. Isto é comumente utilizado para realizar uma comparação rápida entre dois valores.

atentar e inserir uma instrução adequada neste espaço, ou então uma instrução que não tenha efeitos.

O conjunto de operações aritmético-lógicas é bastante extenso e completo. Todas as operações mais comuns estão presentes, com duas exceções significativas: as operações de multiplicação e divisão. Esta decisão foi tomada para simplificar o projeto da unidade lógico-aritmética do RISCO, com a desvantagem de impor uma penalidade na performance dos programas compilados para esta arquitetura, como será visto no capítulo 6. As operações cujos mnemônicos terminam em "c" utilizam o bit C (carry) do registrador PSW nas operações, possibilitando a utilização de técnicas interessantes para a aritmética com inteiros de mais de 32 bits. Além dessas instruções, existem diversas variações de deslocamento e rotações que são úteis na manipulação de inteiros com menos de 32 bits.

As instruções de acesso a memória utilizam o modelo tradicional de *load* e *store* do RISC. O RISCO também incluiu variantes destas duas instruções para os casos comuns em que há um incremento ou decremento de uma variável utilizada no endereçamento da memória. Incrementos antes e depois do endereçamento são representados pelos sufixos "`pri`" e "`poi`", enquanto decrementos após o endereçamento são identificador por "`pod`". São situações comuns em códigos que manipulam arranjos.

As instruções de salto são implementadas como somas condicionais. Isto é, são instruções `add` que só são executadas caso certo bit do registrador PSW esteja ligado ou não. O programador precisa ter o cuidado de especificar que o operando DST seja o registrador R31, e que a soma FT1 + FT2 resulte no endereço desejado.

A instrução de chamada de sub-rotina do RISCO (`sr`), e suas variantes condicionais, determinam o endereço destino como nas instruções de salto, porém também guardam o valor atual de R31 na posição especificada pelo registrador DST (que neste caso não equivale ao destino). O programador precisa atentar para que DST-1 seja um espaço de memória válido e esteja intacto quando a subrotina retornar. Note que não há uma instrução explícita para o retorno da sub-rotina. Entretanto, dependendo da escolha da convenção de chamada, ela pode ser derivada a partir de `ldpoi`. Isto será discutido em detalhe no capítulo 6.

3.3 Plataforma de desenvolvimento

Como foi visto na seção 2.2.1, os processos tradicionais de desenvolvimento de software para sistemas embarcados necessitam de um conjunto mínimo de ferramentas específicas para a plataforma em questão.

Além disto, não é suficiente contar apenas com ferramentas que suportem o desenvolvimento de software em linguagem de montagem. Por exemplo, apesar de se tratar de uma arquitetura RISC, o conjunto de instruções do RISCO é, em certos pontos, muito simplista para a programação manual do dispositivo. Isto é, algumas decisões ligadas à filosofia RISC dificultaram a produção direta de código assembly para este processador:

- Não há instruções para multiplicação e divisão. O programador deve codificar uma função equivalente ou utilizar uma já existente.
- Uso excessivo de ortogonalidade nas instruções faz com que algumas combinações de endereçamento sejam válidas do ponto de vista sintático, porém inválidas semanticamente. É possível fazer uso abusivo dos modos de endereçamento para produzir uma instrução que não faça sentido ou que não seja o desejado.
- Carregar uma constante de 32 bits em um registrador utiliza duas instruções, ao invés de uma instrução de tamanho estendido, pois o RISCO usa um esquema de tamanho fixo.
- As instruções de salto que utilizam endereçamento direto estão limitadas a um salto cujo destino seja representado por uma constante de 17 bits. Isto é, em programas com mais de 128 KB já não é mais possível realizar um salto direto para uma determinada posição. Saltos maiores com endereçamento direto só são possíveis quando utiliza-se mais de uma instrução.

Em 2.2 discutimos como já é bem estabelecido o fato de que, até no contexto de sistemas embarcados, a programação utilizando linguagens de alto nível apresenta inúmeras vantagens com relação à programação direta em linguagem de máquina. As decisões minimalistas presentes no projeto do RISCO, explicadas acima, simplificaram o projeto do processador (e também a sua eficiência e consumo de energia), porém aumentaram a dificuldade de programação na linguagem de máquina do RISCO. Além desta barreira de aprendizado, a falta de suporte a linguagens estruturadas inibe o uso sério do processador como uma opção prática para a construção de um sistema embarcado.

Para qualquer uso real do processador RISCO, assim como para facilitar a continuidade das pesquisas acadêmicas que o envolvam (CARRO; SUZIM, 1996), é clara a necessidade de um conjunto de ferramentas que facilitem a programação deste processador, suportando uma linguagem que apresente suporte às abstrações de software estruturado.

Até o momento da conclusão deste trabalho, o desenvolvimento de software para este processador se dá somente na academia. A tese de mestrado de Junqueira e Suzim (1993), contendo a primeira descrição do RISCO, apresenta um anexo com uma descrição funcional do processador em um dialeto de C (HDC). Desde essa primeira especificação, foram desenvolvidas versões do RISCO em SystemC e VHDL. Com relação ao suporte a programação para o processador, havia somente um montador de linguagem de montagem para execução do programa junto ao modelo SystemC.

3.3.1 Ferramentas

O objetivo principal deste trabalho foi a construção de um conjunto de ferramentas para o desenvolvimento de software para o processador RISCO em uma linguagem de alto nível. Além de suplantar e/ou estender as ferramentas atuais existentes para o RISCO, espera-se que este ambiente seja suficientemente maduro e robusto para que o processador ganhe uma visibilidade maior como uma opção viável para o desenvolvimento de sistemas embarcados. Todo o software resultante está disponível sob a forma de código livre, com a licença LGPL. Ele pode ser obtido em (VILELA, 2010). O conjunto de ferramentas é divido em camadas, de acordo com quais etapas dos fluxogramas nas figuras 1 e 2 cada uma delas representa.

- *Plataforma base*: São as ferramentas que habilitam o desenvolvimento e a execução de software escrito na linguagem de montagem do RISCO. São o **risco-as** e o **risco-sim**.
- *Compilador*: é o módulo RISCO para o projeto LLVM, sob o nome de **risco-llvm**. Com ele, é possível aproveitar as ferramentas de compilação disponibilizadas pelo LLVM, incluindo o compilador de C, C++ e Object-C.
- *Análise de código*: São os componentes que permitem analisar os grafos de fluxo de execução de um código RISCO. São disponibilizados como a ferramenta **risco-cfg**.

Os próximos capítulos discutem o projeto e o desenvolvimento desses módulos.

4 Plataforma base de desenvolvimento

O primeiro passo na construção de uma plataforma de desenvolvimento de software completa para uma dada arquitetura nova é o desenvolvimento de uma camada básica que possibilite a construção de software em linguagem de montagem. A programação neste nível, comumente denominado de "baixo nível", apresenta vantagens e desvantagens, considerações que já foram discutidas no capítulo 2. Como vimos, um programa escrito manualmente em baixo nível tem a liberdade de poder aplicar otimizações de eficiência ou diminuir o tamanho do executável usando truques que são difíceis de serem empregados por um compilador, porém estão ao alcance de um profissional com experiência na área e um conhecimento profundo da arquitetura.

Mesmo que não seja o caso de desenvolver tal software em baixo nível, todo o desenvolvimento de software em alto nível, como o presente em sistemas embarcados em geral, tem um passo de compilação onde o código é traduzido da linguagem original e o resultado da tradução passa a utilizar as instruções que a arquitetura base provê, preservando a semântica do programa original. A plataforma base de desenvolvimento é o pilar responsável pela execução de todo o software para o RISCO.

Entretanto, a geração do código binário executável não é o objetivo final da plataforma base. O desenvolvedor necessita, de algum modo, executar este programa e observar o seu comportamento, a fim de realizar modificações em geral no código original, corrigindo-o. Ainda mais, é necessário que ele possa executar o código a partir de uma máquina que não utiliza o processador RISCO, obtendo os mesmos efeitos que seriam obtidos caso ela o utilizasse.

As ferramentas `risco-as` e `risco-sim` compõem a plataforma base de desenvolvimento de software em linguagem de montagem para o processador RISCO, atendendo às necessidades delineadas acima. Nas seções a seguir tem-se uma exposição destas ferramentas, as suas motivações, decisões de projeto e funcionalidades.

4.1 Montador

O montador é o software responsável pela tradução de um código fonte escrito em linguagem de montagem, utilizando instruções do processador alvo, para um arquivo binário contendo instruções, codificadas de acordo com as regras do alvo, e dados, que serão utilizados pelas instruções. De acordo com a definição dada, ele é de fato um compilador. Entretanto, não é comum utilizar essa denominação pois existem algumas diferenças entre as características de um montador e de um compilador tradicional:

- Em geral, a linguagem fonte e a linguagem destino de um montador são quase idênticas, diferindo somente em detalhes, como será visto adiante.
- A linguagem fonte não apresenta diversos tipos de dados ou construções, nem meios de composição de tipos. Ela é suficientemente simples para ser uma representação fiel do conjunto de instruções do processador.
- Como a linguagem de montagem é bastante simples, sem uma grande quantidade de construções sintáticas, como uma linguagem de alto nível, o processo de tradução é mais mecânico. Não são necessárias diversas análises no código, ou passes de otimização.

A interação entre o compilador e o montador aborda diversos aspectos interessantes, fazendo com que o projetista deva balancear o conjunto de todas as funcionalidades requisitadas entre os dois. Como será visto na seção 4.1.1, a linguagem de montagem está em um nível de abstração um pouco acima do conjunto de instruções do compilador. As facilidades que ela apresenta simplifica consideravelmente a implementação do compilador. Porém há um limite, pois quanto mais funcionalidade depender do montador, mais sua complexidade irá aumentar. No outro extremo, existe a opção de se livrar do montador como um todo e fazer com que o próprio compilador gere o código binário executável (LOPES, 2009a). Isto elimina o passo intermediário de geração do arquivo de texto que seria passado para o montador, aumentando a eficiência do processo como um todo. A maior desvantagem seria o aumento de complexidade no compilador, que teria de lidar com todas as tarefas do assembler.

A ferramenta **risco-as**, o montador construído para a plataforma RISCO, apresenta algumas decisões de projeto que diferem da norma apresentada acima e do fluxograma da figura 1. A diferença é que a funcionalidade de ligador foi incluída nesse programa. Desse modo, ele recebe como entrada um conjunto de arquivos em linguagem de montagem e

gera o arquivo executável final, sem precisar realizar o passo de ligação com um programa externo. A desvantagem mais significativa desta decisão é que ele não consegue unir um programa sendo escrito em linguagem de montagem com uma biblioteca pré-compilada, disponível em código binário. Na prática, isto não prejudica o processo de desenvolvimento como um todo, pois ainda é possível utilizar a biblioteca caso o seu código fonte esteja disponível. Entretanto, perde-se o modelo de compilação separada, e o tempo médio de compilação aumenta.

O papel principal do montador, em termos de funcionalidade, é facilitar a geração de código do compilador tomando conta das tarefas mais mecânicas deste processo (SALOMON, 1992). As tarefas mais importantes são a codificação de instruções e o cálculo dos endereços de salto.

A codificação de instruções, no contexto do código RISCO, é a funcionalidade que traduz uma descrição textual de uma instrução RISCO (§3.2) em uma palavra de 32 bits que seja reconhecida pelo processador. Já o cálculo dos endereços de salto é uma análise que determina todas as constantes utilizadas em instruções de salto, sejam elas um `jmp` ou `sr`. Isto se dá através do uso de rótulos na linguagem de montagem.

Algoritmo 1 Exemplo do uso de rótulos na linguagem de montagem

```
    ... # conjunto A
label1:
  add $r1, $r2, $r3
  sub $r4, $r0, $r4
  jmp label2
  ... # conjunto B
label2:
  add! $r0, $r1, $r2
  jmpeq label1
```

Considere o exemplo 1, na sintaxe do `risco-as`. Os rótulos `label1` e `label2` são constantes inteiras que representam a posição das instruções que os seguem de imediato. Desta maneira, eles podem ser usados como operandos em instruções, como no exemplo acima. Entretanto, o valor destas duas constantes não é aparente. Ele depende do número de instruções nos conjuntos A e B (omitidos por clareza). Qualquer reorganização do código nestes conjuntos, ou nas instruções mostradas acima, iria mudar estes valores. Este cálculo é uma tarefa tediosa, porém perfeitamente dentro das possibilidades de análise de um montador.

A seguir será analisada em detalhes a linguagem de montagem utilizada pelo montador do RISCO.

4.1.1 Linguagem de montagem do RISCO

A especificação original do RISCO é acompanhada por um trabalho (LIMA, 1993) que descreve uma linguagem de montagem para este processador, porém não foi possível conseguir uma cópia deste software. Desta maneira, foi decidido que uma nova linguagem de montagem seria definida para o RISCO, implementada pelo `risco-as`.

A linguagem foi modelada tendo como base a gramática do montador/simulador Spim (LARUS, 2004). Não foi utilizada a linguagem original do primeiro montador do RISCO pois esta apresenta uma sintaxe que difere bastante das linguagens em uso atualmente, com relação às diretivas e a definição de macros. Ao mesmo tempo, foi aproveitada esta oportunidade para incluir certas funcionalidades que são assumidas pelo módulo de geração de código do LLVM.

Para facilitar a sua implementação, as funcionalidades de definição de constantes internas e macros foi delegada ao pré-processador C (KERNIGHAN; RITCHIE, 1988), que é executado antes que o arquivo fonte seja entregue ao `risco-as`. Desta maneira, é possível utilizar a sintaxe já conhecida por programadores C para definir macros textuais em um arquivo da linguagem de montagem.

O restante do montador foi dividido em 4 módulos simples: analisador léxico, analisador sintático, analisador semântico e o gerador de código. Os analisadores léxico e sintático foram desenvolvidos utilizando-se as ferramentas livres **Flex** e **Bison** (LEVINE, 2009).

O Flex é um gerador de analisadores léxicos baseados em autômatos. Ele lê uma descrição dos lexemas da linguagem em forma de expressões regulares e gera o analisador léxico. Já o Bison é um gerador de analisadores sintáticos. Ele recebe como entrada uma gramática LALR(1) anotada com ações, trechos de código em C que são executados quando uma regra da gramática é reconhecida, e gera um programa C capaz de realizar a análise sintática a partir dos tokens gerados pelo produto do Flex.

Elementos léxicos

O analisador léxico reconhece os seguintes elementos:

- Comentários em uma única linha, iniciados pelo caractere "#". Os símbolos de pontuação utilizados são: `'.'`, `':'`, `','`, `'+'`, `'['`, `']'`, `';'`, `'!'` e `'@'`.
- Identificadores de diretiva, tais como: `.word`, `.ascii`, `.resw`, `.text` e `.global`. Também reconhece identificadores de instruções, utilizando a nomenclatura do apêndice A. Exemplos: `add`, `xor` e `sll`.
- Identificadores de registradores, da forma: `[%|$][rR][0-9]+`. Também reconhece apelidos de registradores utilizados pelo compilador, tais como: `$pc`, `$psw` e `$sp`.
- Identificadores alfanuméricos que não iniciem com um dígito, strings literais cercadas por aspas duplas e constantes numéricas em decimal e hexadecimal.
- Todos os espaços em branco entre os lexemas são ignorados.

Gramática e Semântica da linguagem

A gramática da linguagem[1] de montagem será apresentada utilizando a notação BNF (BACKUS, 1960) estendida. Os símbolos entre `''<''` e `''>''` são não terminais. Todos os outros são representações de tokens. Os símbolos entre aspas duplas indicam o próprio valor do token. O símbolo `''#''` indica a produção vazia.

```
<programa>    ::= <declaracoes> <secoes>
<declaracoes> ::= <declaracoes> ".start"  t_identifier |
                  <declaracoes> ".global" t_identifier |
                  <declaracoes> ".extern" t_identifier | #
<secoes>      ::= <secoes> <dados> | <secoes> <codigo> | #
```

Um programa é definido como uma sequência de *declarações* e uma sequência de *seções*. Uma declaração serve para indicar 3 casos: se um símbolo é visível externamente em outra unidade de compilação (`.global`), se ele pertence a outra unidade (`.extern`) ou se ele será o ponto de início do programa (`.start`). Após as declarações, seguem as especificações de seções. O binário final irá conter somente duas seções, uma de dados e outra de código executável. Entretanto, o texto poderá conter fragmentos delas separadamente.

```
<dados> ::= ".data" <diretivas>
```

[1]Para simplificar o texto, apresentamos uma versão reduzida da gramática original escrita na sintaxe do Bison. Entretanto, não há mudanças significativas com relação a original

```
<diretivas> ::= <diretivas> ".ascii" t_string_literal |
                <diretivas> ".word" t_integer |
                <diretivas> ".byte" t_integer |
                <diretivas> ".resw" t_integer |
                <diretivas> ".resb" t_integer | #
```

A seção de dados é utilizada para o armazenamento estático de variáveis utilizadas pelo programa. É, por exemplo, a região de armazenamento utilizada para as variáveis globais em um programa C. As diretivas `.ascii`, `.word` e `.byte` reservam o espaço de cada tipo e já iniciam o seu valor com o argumento dado. As diretivas `.resw` e `.resb` só reservam a quantidade de variáveis indicada, inicialmente com valor 0.

```
<codigo> ::= ".text" <instrucoes>
<instrucoes> ::= <instrucoes> <rotulos> <instrucao> [";"] | #
<instrucao> ::= <aritmetica_logica> ["!"] <operandos_normais> |
                <salto>             ["!"] <operandos_salto>   |
                <chamada_rotina>    ["!"] <operandos_salto>   |
                <leitura_memoria>   ["!"] <operandos_leitura> |
                <escrita_memoria>   ["!"] <operandos_escrita>
<rotulos> ::= <rotulos> t_identifier ":" | #
```

A seção de código consiste de uma lista de instruções. Cada uma pode ser precedida por n rótulos e, caso o seu mnemônico seja seguido pelo caractere **‘‘!’’**, o seu bit APS (3.2) será 1. Os elementos não terminais correspondentes aos tipos de instrução equivalem aos tokens dos opcodes de cada classe. Dependendo do tipo da instrução, há uma sintaxe diferenciada para a especificação dos seus operandos.

```
<operandos_normais> ::= t_register "," t_register "," t_register
                      | t_register "," t_register "," <constante>
                      | t_register "," <constante_estendida>
<operandos_salto> ::= t_register | <constante> | <operandos_normais>
<operandos_leitura> ::=
    t_register "," "[" <constante> ["+" t_register] "]"
  | t_register "," "[" t_register  ["+" (t_register | <constante>)] "]"
<operandos_escrita> ::=
    "[" <constante> ["+" t_register] "]" "," t_register
  | "[" t_register  ["+" (t_register | <constante>)] "]" "," t_register
```

A regra para operandos normais equivale aos 3 formatos do campo de operandos, discutido anteriormente. A distinção entre as 5 interpretações se dá analisando quais registradores foram escolhidos. Para as operações de salto, é possível especificar somente o destino do salto, e o montador se encarrega de utilizar o registrador 31 e 0 quando necessários. Para operações de leitura e escrita na memória, a sintaxe tradicional com colchetes é utilizada.

```
<constante> ::= t_integer | t_identifier
<constante_estendida> ::= ["@"] <constante>
```

As constantes podem ser inteiros ou rótulos. Em ambos os casos, caso o caractere ‘‘@’’ preceda a constante, o formato da instrução será o 4 (a constante é carregada nos bits mais significativos do operando).

O programa 2 é um exemplo completo de código na linguagem de montagem do RISCO. Ele define uma string na área de dados (terminada com um byte nulo) do código, e quando for executado conta o tamanho da string e o imprime na saída padrão. O código faz uso da biblioteca padrão do RISCO e de funcionalidades do `risco-sim` que serão mencionadas posteriormente.

Observando o código deste exemplo é possível destacar algumas das funcionalidades do montador que já foram mencionadas. O código passa pelo pré-processador C antes de ser lido pelo `risco-as`. Desta maneira, foi possível definir os dois macros de instruções e incluir um arquivo externo. Além disso, o uso dos rótulos facilitou a codificação do exemplo. Não é preciso se preocupar com os endereços reais das instruções.

Para calcular os valores dos rótulos, o montador realiza um processo de 2 passos sobre o código fonte. Primeiramente, uma tabela de símbolos é criada, mapeando símbolos para valores inteiros, inicialmente vazia. A medida que o programa percorre o código fonte pela primeira vez, ele mantêm um contador indicando quantos bytes já foram observados, em termos de quantidade de instruções. Dessa maneira, ao encontrar uma definição de rótulo ele pode marcar na tabela qual o seu valor, que é igual a posição da próxima instrução no texto. Este processo deve levar em conta as diretivas de dados, introduzidas na próxima seção. Ao percorrer o código fonte pela segunda vez, ao encontrar uma instrução que utilize um rótulo como operando, o rótulo é substituído pelo seu valor na tabela.

O programador também não se preocupa com os 5 formatos de instruções de operandos discutidos no capítulo 3. Tomando como exemplo a instrução de leitura da memória, `ldpoi`. Foi possível escrevê-la em uma notação familiar, sem atentar para a definição

Algoritmo 2 Exemplo de programa na linguagem de montagem RISCO

```
#include "risco/stdlib.h"
#define cmp(a, b) add! $zero, a, b
#define nop       add $zero, $zero, $zero

.global main

.data
str: .ascii "Hello World!" .byte 0

.text
main:
        add $t1, $zero, $zero    # index
        add $a0, $zero, $zero    # counter
loop:
        ldpoi $t2, [str + $t1]
        cmp ($t2, 0)
        jmpeq out
        nop
        add $a0, $a0, 1
        jmp loop
        nop
out:
        sr print_int
        nop
        add $a0, $zero, $zero
        sr exit
        nop
```

de DST, FT1 e FT2. Dependendo dos tipos e da ordem dos argumentos definidos para uma instrução, o montador deduz qual será a instrução de máquina equivalente. Estas funcionalidades também facilitaram a implementação do compilador.

4.1.2 Funcionalidade do ligador

No exemplo 2 é possível observar que o código fez uso de subrotinas que não foram definidas neste arquivo: `print_int` e `exit`. O código fonte destas duas rotinas, em linguagem de montagem, está em outro arquivo. O segundo arquivo contêm diretivas `.global` para esses dois símbolos, de modo que eles sejam visíveis externamente. Além disso, o arquivo `risco/stdlib.h`, incluído pelo código do exemplo, contêm diretivas `.extern` para os símbolos.

O software denominado de *ligador* é comumente responsável por agregar todas as unidades de compilação de um programa, identificar símbolos externos, símbolos locais e dependências entre as unidades para então gerar um único arquivo binário com todo o código, como no modelo da figura 1. Uma das decisões de projeto para o `risco-as` foi integrar a funcionalidade de ligador nesta ferramenta. Esta direção fez com que a complexidade do montador aumentasse, porém é um risco aceitável quando comparado aos esforços necessários para desenvolver e manter o ligador.

Foi estudada a opção de utilizar um formato de código binário relocável existente, como o ELF para sistemas Linux. Algumas das vantagens seriam (LU, 1995): não precisaria definir outro formato; pode-se utilizar o conjunto de ferramentas já disponíveis que trabalham com o ELF, inclusive o ligador `ld`; permite manipulação de alta performance; é um formato flexível e extensível. Optou-se por seguir com a implementação do ligador embutido no montador após serem analisados os requisitos da plataforma de desenvolvimento para o RISCO e ser percebido que portabilidade e alta performance ainda não são prioridades. O que era necessário era agilidade na implementação desta peça básica, a fim de que o estudo e desenvolvimento das ferramentas para linguagens de alto nível pudesse ser iniciado.

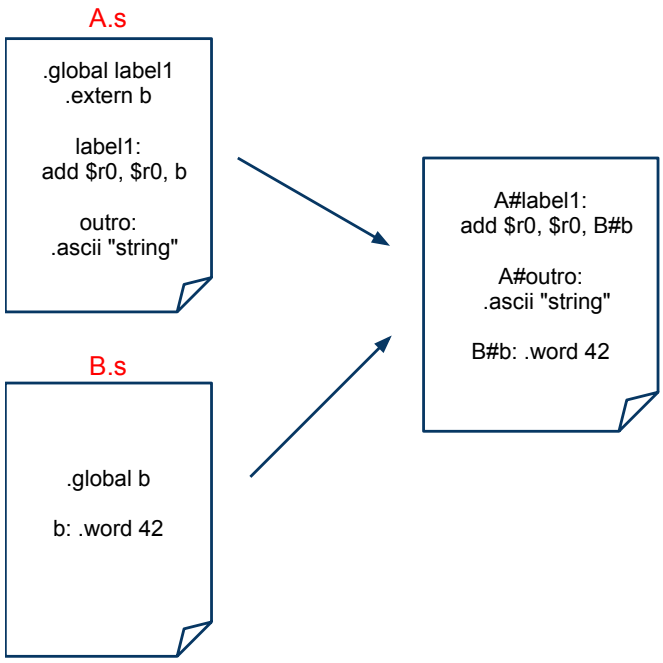


Figura 5: Processo de união de unidades de compilação no `risco-as`

A resolução de símbolos é feita através de um esquema simples, baseado no modelo de cálculo dos valores dos rótulos apresentado na sessão 4.1.1. O código que resulta da ligação das diferentes unidades de compilação pode ser visto como a união do código de

todas, fazendo a reescrita dos seus rótulos, como na figura 5. Com o arquivo resultante desta união, pode-se utilizar o esquema de resolução de rótulos discutido anteriormente, que trabalha em um único arquivo. Na prática, este é um esquema ineficiente pois pode gerar um arquivo de tamanho excessivo. Por este motivo, os rótulos globais são mantidos em uma tabela de símbolos globais, separadamente das tabelas locais de cada arquivo. A união das unidades de compilação nunca é feita explicitamente.

4.2 Simulador

O montador `risco-as` gera um arquivo binário puro contendo a codificação exata das instruções descritas no programa em linguagem de montagem. As instruções contidas neste arquivo não correspondem ao conjunto de operações definidas pelo computador *host*, onde ocorre o desenvolvimento do software. O desenvolvedor necessita de um dispositivo que execute fielmente as instruções contidas no executável, e que possa informá-lo do comportamento apresentado pelo programa. O dispositivo que representa o sistema embarcado contendo o processador RISCO pode assumir basicamente 3 formas distintas:

- *Dispositivo real*: o código é executado em um equipamento físico real, um SE baseado em processador RISCO. Neste caso, o carregador é um software capaz de se comunicar com o dispositivo externo e carregar o código em sua memória. Como exemplo disto, pode-se citar um FPGA programado para simular um processador RISCO.

- *Especificação executável*: uma técnica comum para prototipagem rápida de sistemas é utilizar uma linguagem de especificação de hardware que seja executável, possibilitando que um software especializado realize uma simulação de alta fidelidade. Neste caso, o carregador é um módulo particular deste software, responsável por incluir o código executável na representação interna da memória do circuito. Um exemplo desta forma são ferramentas de simulação VHDL.

- *Simulador artificial*: neste caso, o dispositivo é representado por um software, escrito em uma linguagem de programação de alto nível, responsável por ler um arquivo binário executável de entrada e executar suas instruções uma a uma, de modo que os efeitos colaterais possam ser observados na máquina hospedeira (*host*).

O trabalho inicial em cima do RISCO (JUNQUEIRA; SUZIM, 1993) envolveu a sua sintetização em um circuito integrado, logo seria lógico considerar a primeira possibi-

lidade de execução do código. Porém, esta alternativa é pouco prática pois o processo de carregamento do código na memória do dispositivo não é trivial, e isto desacelera o desenvolvimento de software para a plataforma consideravelmente.

A utilização de uma especificação executável é uma opção válida, tendo em vista que haviam modelos do RISCO em SystemC e VHDL disponíveis. Porém estes modelos não tem como objetivo suportar o desenvolvimento de software para o processador, e sim validar a especificação do processador em si.

Desta forma, desenvolvemos um simulador artificial que implementa a arquitetura do RISCO com fidelidade. Existem diversas vantagens para essa abordagem. Em geral, simuladores deste tipo apresentam performance superior aos outros. Isto decorre do fato de que esse simulador tem o privilégio de ser executado na máquina de desenvolvimento do código, que comumente tem uma performance muito superior a do sistema embarcado. Neste caso, o carregador é simplesmente um módulo interno do simulador que lê o arquivo binário proveniente do montador.

Além disso, como visto no capítulo 2, o programador deve contar com um ambiente que suporte simulações rápidas da execução de um programa, e mesmo assim possa obter resultados iguais ou comparáveis aos resultados obtidos quando o código é executado em um dispositivo real.

Com relação a utilização de simuladores artificiais para plataformas RISC, cabe mencionar como exemplo a linha de processadores MIPS, que é amplamente utilizada para o ensino de arquitetura e organização de computadores (BRANOVIC; GIORGI; MARTINELLI, 2004), assim como em sistemas embarcados (FURBER; EDWARDS; GARSIDE, 2000). Existem diversos simuladores para este processador, tais como o apresentado em (VOLLMAR; SANDERSON, 2006) e o Spim (LARUS, 2004).

O simulador em si foi desenvolvido como um interpretador simples que considera o conjunto de instruções do RISCO como *bytecodes*, e age como uma máquina virtual. É um padrão de projeto bastante conhecido, tendo como exemplo destacado de uso a máquina virtual da plataforma Java (LINDHOLM; YELLIN, 1999). Uma simplificação do loop de interpretação está presente no exemplo 3

Uma instrução é lida como um inteiro sem sinal. Para realizar a sua interpretação e execução, a função `decode` decodifica a palavra e separa os seus campos relevantes. A rotina `run` apresenta o corpo do loop em si, onde cada instrução é lida da memória, decodificada e executada. Note que o simulador inicia o contador de programa (PC) e o

Algoritmo 3 Loop de interpretação do `risco-sim`, simplificado

```
word reg[NUM_REGISTERS]
byte memory[MEMORY_SIZE];

struct Decoded {
  unsigned opcode;
  bool aps;
  int operand_format;
  ...
};

decode(unsigned instruction, Decoded& decoded) {
  decoded.opcode = instruction >> 25;
  decoded.aps = instruction & (1 << 24);
  int f1_f0 = (instruction >> 22) & 0x03;
  ...
}

void run() {
  reg[PC_REG] = start_position;
  reg[SP_REG] = MEMORY_SIZE-1;

  while (running) {
    unsigned instr = memory_read(reg[PC_REG]);
    reg[PC_REG] += 4;
    Decoded decoded = decode(instr);
    switch (decoded.opcode) {
      case OP_ADD:
        reg[decoded.dst] = decoded.ft1 + decoded.ft2;
        break;
      ...
    }
    if (decoded.aps) update_psw();
  }
}
```

ponteiro da pilha (SP) automaticamente, entre outras coisas. Iremos abordar o registrador SP mais adiante.

4.2.1 Formato do executável

Para que as ferramentas `risco-as` e `risco-sim` consigam trabalhar juntas, o arquivo binário gerado pelo montador precisa seguir um certo formato pré-combinado entre os

dois softwares. Este formato é denominado formato de arquivo executável, ou formato de interface binária entre aplicações (LEVINE, 1999). Como o ambiente de desenvolvimento para o RISCO não suporta o uso de compilação separada ou bibliotecas dinâmicas, tal formato pode ser mais simples, contendo somente duas seções: dados e instruções. O formato binário é detalhado na figura 6

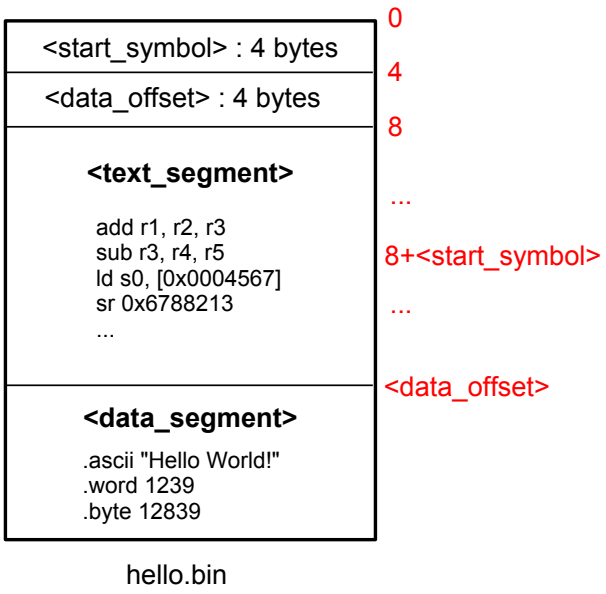


Figura 6: Formato de arquivo executável na plataforma RISCO

O arquivo inicia com dois números inteiros de 4 bytes cada. Eles são codificados com os bytes menos significativos no início do arquivo (codificação *little-endian*). O primeiro aponta para uma posição dentro do segmento de texto onde fica o símbolo de início de programa, definido no código em linguagem de montagem com a diretiva `.start`. O segundo número é o deslocamento necessário para partir do início do arquivo até o início da seção de dados. Este valor não é utilizado atualmente no simulador.

A vantagem mais significativa em utilizar um formato de executável simples como o apresentado acima é que o carregador do programa não precisa realizar nenhum pré-processamento no arquivo. Após ler os dois números do início, ele pode mapear o resto do arquivo na memória e começar a executar as instruções. Não é preciso mover as seções de texto e de dados das suas posições originais.

4.2.2 Chamadas de sistema

Durante a execução de um código RISCO, o simulador é capaz de exibir informações de depuração, tais como os valores de todos os registradores, ou o valor de alguma posição

de memória. Além disso, é possível interromper a execução do programa e continuar de certo ponto. Entretanto, este tipo de interação com o código, onde é necessário examinar os valores dos registradores manualmente, requer mais esforço do programador para entender o comportamento apresentado pelo programa. Utilizando somente as instruções descritas no apêndice A, o código não tem os meios para interagir com a plataforma onde executa. As únicas operações que podem ser feitas são manipulações com os valores da memória e dos registradores.

Devido a estes problemas, introduzimos a utilização de *chamadas de sistema* no `risco-sim`. Desta maneira, um código em linguagem de montagem pode requisitar uma certa funcionalidade da plataforma base, que poderá executar operações de interação com o usuário já que o simulador é um programa normal no sistema operacional host.

Para realizar uma chamada de sistema, o programa deve colocar o código da chamada no registrador `$A0` (R03), e os respectivos parâmetros, quando necessários, nos registradores `$A1` (R04) a `$A4` (R07). Depois da preparação destes registradores, é necessário invocar a plataforma base utilizando a instrução `syscall`. Após esta instrução, o registrador `$V0` (R02) irá conter o seu valor de retorno, caso ele exista. Esta instrução não pertence ao conjunto original do RISCO, ela só é reconhecida pelo `risco-sim`. Para o seu opcode, utilizamos um dos códigos não utilizados das instruções do tipo aritmético-lógico (§3.2).

Nome	**Código**	**Argumentos**	**Retorno**	**Equivalente em C**
`print_int`	1	`int i`		`printf("%d", i)`
`print_uint`	2	`unsigned int i`		`printf("%u", i)`
`print_char`	3	`char c`		`printf("%c", c)`
`print_string`	4	`char* s`		`puts(s)`
`read_int`	5		`int i`	`scanf("%d", &i)`
`read_char`	6		`char c`	`scanf("%c", &c)`
`read_string`	7	`char* s; int n`		`fgets(s,n,STDIN)`
`fopen`	8	`char* f, m`	`int fd`	`fopen(f,m)`
`fwrite`	9	`char* b; int n, fd`	`int k`	`fwrite(b,1,n,fd)`
`fread`	10	`char* b; int n, fd`	`int k`	`fread(b,1,n,fd)`
`fclose`	11	`int fd`	`int st`	`fclose(fd)`
`malloc`	12	`int n`	`void* ptr`	`malloc(n)`
`free`	13	`void* ptr`		`free(ptr)`
`exit`	99	`int code`		`exit(code)`

Tabela 2: Chamadas de sistema do `risco-sim`

Na tabela 2 apresentamos o conjunto de chamadas de sistema implementado, junto com os seus identificadores. A semântica da operação é dada por um trecho de código C cujo efeito é igual ao da chamada. Incluímos chamadas para leitura e impressão de

valores simples na entrada e saída padrão, respectivamente. Como veremos no capítulo 6, estas chamadas, junto com as chamadas `malloc` e `free`, foram essenciais para facilitar a verificação do compilador RISCO. As rotinas de manipulação de arquivos também são úteis neste contexto.

Em um código C em ambiente Unix, as chamadas de sistema tem um funcionamento semelhante ao apresentado acima. Elas seguem a especificação POSIX (KERRISK, 2010), com algumas extensões. Na prática, os códigos no espaço de usuário (fora do espaço do kernel) não utilizam estas chamadas diretamente. A biblioteca padrão C inclui funções que encapsulam o ato de invocar a chamada de sistema com uma função normal, que pode ser invocada utilizando a convenção de chamada do sistema em questão. Por exemplo, o código da função `printf`, parte da biblioteca padrão, é o responsável por realizar a chamada de sistema com uma instrução parecida com a `syscall`.

Adotamos este esquema de funções *proxy* nas chamadas de sistema do RISCO. Como visto no exemplo 2, o código fez uma chamada de subrotina normal para `print_int`. Esta função é responsável por carregar o código correto da chamada de sistema (neste caso, 1) e executar a instrução `syscall`. Existe uma função proxy para cada chamada de sistema apresentada na tabela 2, com os nomes da coluna "Nome".

Este capítulo conclui a apresentação das ferramentas que compõem a plataforma base de desenvolvimento para o processador RISCO. Nos próximos capítulos será analisado o suporte a linguagens de alto nível. Inicialmente é feita uma exposição do projeto LLVM e da sua significância para este trabalho. Em seguida, as ferramentas `risco-llvm` e `risco-cfg` são discutidas.

5 A infraestrutura de compiladores LLVM

O projeto LLVM, inicialmente denominado de *Low Level Virtual Machine*, consiste de uma coleção modular de componentes reutilizáveis para implementação de compiladores e de ferramentas para análise e otimização de código executável (LLVM..., 2002). Ele iniciou-se como um projeto de pesquisa na University of Illinois e hoje em dia conta com um número bastante significativo de contribuintes internacionais e de sub-projetos criados em topo destes componentes. É um destaque dentre os exemplos de casos de sucesso em termos de pesquisa acadêmica na área de compiladores, e uma referência entre as ferramentas de compilação tanto de código aberto como proprietárias.

Lattner e Adve (2004a) destacam a principal motivação para a construção deste projeto como sendo um esforço coletivo para gerenciar a complexidade inerente na construção de compiladores no estado-da-arte atual. As aplicações modernas necessitam de tecnologias de compilação que excedam o modelo tradicional do compilador estático, que realiza análises e otimizações sob o código fonte original e simplesmente gera o executável final. Um *sistema de compilação* moderno deve apresentar as seguintes características (LATTNER; ADVE, 2004a):

- O gerador de código deve ser capaz de realizar análises complexas no código fonte de modo a entender as dependências entre os seus componentes e poder realizar otimizações além das barreiras das unidades de compilação.
- É preciso um conjunto de ferramentas que permitam a otimização de um programa ao longo do seu ciclo de vida, e não somente em tempo de compilação.
- O sistema deve ser capaz de realizar otimizações dinâmicas em tempo de execução, utilizando informações obtidas com o comportamento observado do software em uma de suas execuções.
- O sistema deve prover ferramentas para análise estática do código, capazes de extrair

informações interessantes sobre o programa. Por exemplo: correção automática de bugs, detecção de vazamentos de memória, etc.

Os componentes (módulos) do LLVM participam de todos os passos de análise de código ao longo do processo de desenvolvimento, provendo ferramentas de código aberto que podem ser utilizadas e estendidas livremente. Os componentes do LLVM não são voltados somente para a construção de compiladores estáticos. O projeto dá suporte a construção de (LATTNER; ADVE, 2004b): interpretadores com geração de código em tempo de execução (também conhecidos como compiladores *just in time* (JIT)), ferramentas de monitoramento de dados em tempo de execução (*profiling*), máquinas virtuais, ferramentas de análise estática, etc. Cada um dos projetos que utilizam os módulos de baixo nível do LLVM são independentes da arquitetura alvo, e suportam diversas plataformas diferentes. Algumas delas são: x86, x86-64, SparcV8, PowerPC, ARM, MSP430 e MIPS.

Para os usuários finais das ferramentas derivadas deste projeto, o LLVM é mais conhecido pelo seu compilador oficial de C e C++[1], o Clang (LATTNER, 2008). A ferramenta Clang utiliza extensivamente a infraestrutura de geração de código do LLVM para criar um compilador completo, com qualidade de produção. Além de estar sobre a bandeira do LLVM, há uma relação simbiótica entre os dois projetos. A construção de um compilador com o porte do Clang fez com que diversos avanços ocorressem nas bibliotecas do LLVM. Ao mesmo tempo, o Clang atua como o produto "oficial" do projeto, promovendo o seu uso.

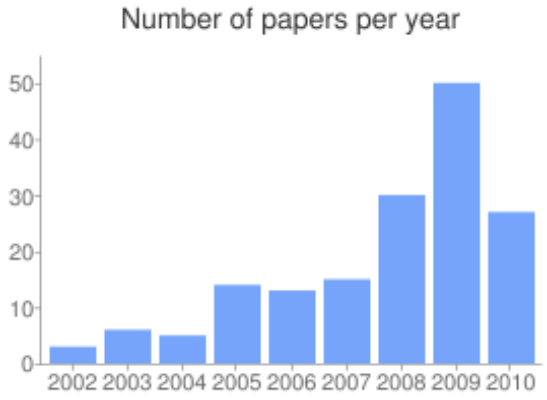


Figura 7: Quantidade anual de trabalhos publicados que utilizam ou desenvolvem em topo do projeto LLVM. *Retirado da página (LLVM..., 2002)*

Atualmente, projetistas de compiladores e pesquisadores na academia utilizam o LLVM como uma plataforma robusta para implementar novas ideias e técnicas da área.

[1] A ferramenta Clang também dá suporte à geração de código para as linguagens Object-C e Object-C++

A boa qualidade do projeto, aliado a grande base de contribuidores e usuários e a sua documentação extensiva ajudam a reduzir o tempo necessário para o desenvolvimento das ferramentas. Além disso, não é prático para um pesquisador construir um compilador completo, ou tentar modificar um compilador existente que não tenha uma boa documentação, para que se possa desenvolver um trabalho. Com o LLVM, é possível utilizar somente os módulos necessários, bem documentados, diminuindo o esforço. Os dados na figura 7 comprovam estes argumentos para a popularidade do projeto.

No início deste trabalho foram analisadas outras alternativas, com o objetivo de poder criar um compilador de C e C++ para a plataforma RISCO. Foi considerada a utilização da coleção de compiladores de código aberto GCC (GCC..., 1987), portando o seu módulo de geração de código para o RISCO. O GCC é o sistema de compilação oficial em virtualmente todos os sistemas operacionais baseados no GNU/Linux. Apesar de ser conhecido tradicionalmente pelos seus compiladores C e C++, a coleção é bastante vasta e o seu código visa a portabilidade para diversas arquiteturas diferentes[2]. A principal desvantagem na utilização do GCC para a implementação de um novo alvo de geração de código é que o seu código é bastante maduro e otimizado, de difícil legibilidade para desenvolvedores externos ao projeto. Em outras palavras, a curva de aprendizado necessário para realizar uma contribuição a este projeto, como é comum aos projetos GNU, é bastante significativa. Por estes motivos optou-se por utilizar o projeto LLVM para este trabalho.

5.1 Arquitetura

O processo de compilação de código com as ferramentas do projeto LLVM é representado pela figura 8, adaptada de (MACHADO, 2008).

O software é desenvolvido em uma linguagem de alto nível e depois é compilado, com um compilador estático tradicional como o Clang, para uma linguagem de representação intermediária (RI), a *LLVM-IR*. Esta representação do código é universal entre as ferramentas do LLVM. Em outro passo, um otimizador em tempo de ligação pode unir o código desenvolvido com bibliotecas na LLVM-IR ou em código nativo, descobrindo as dependências entre as unidades para obter mais informações que guiem as otimizações. O executável gerado, junto com a sua RI equivalente, é produzido.

[2]De fato, ele foi portado para mais arquiteturas diferentes do que qualquer outro compilador em uso (GCC..., 1987)

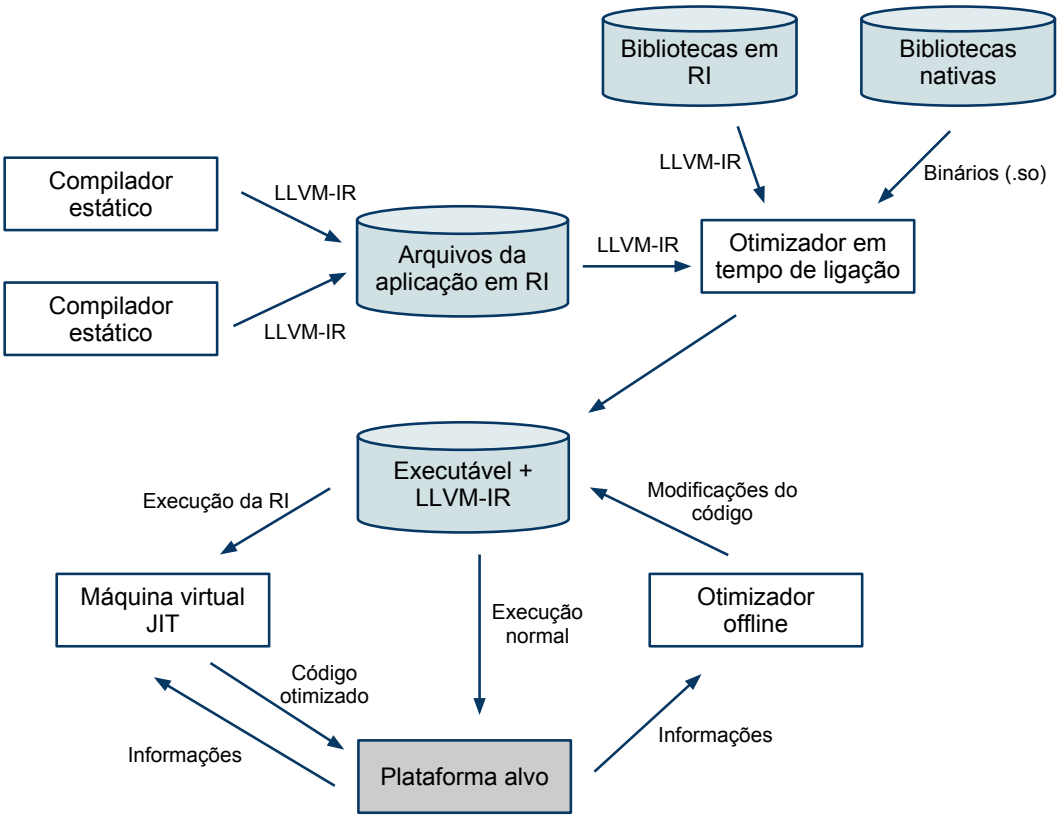


Figura 8: Arquitetura das ferramentas do projeto LLVM

Com este código, existem duas opções de otimização em tempo de execução (LATTNER; ADVE, 2004b). É possível interpretar a sua RI em uma máquina virtual com compilação em tempo de execução. Ela executa o código interpretado, recebe informações sobre o seu comportamento, e tenta compilar trechos da IR para código nativo da plataforma. A outra opção é executar o código binário normalmente, e utilizar as informações de uma execução em particular para guiar um otimizador em tempo de compilação, capaz de melhorar a eficiência do código original. O foco deste trabalho é nos módulos de compilação estática do LLVM. Não é dado suporte às otimizações em tempo de execução mencionadas.

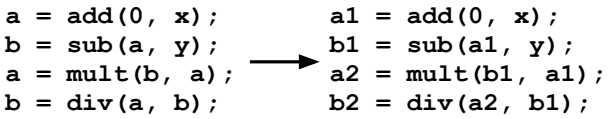
5.2 Representação Intermediária

Como visto na seção anterior, todas as ferramentas do projeto LLVM trabalham com uma representação do código original em uma linguagem intermediária denominada de *LLVM-IR*. Em particular, o módulo de geração de código do LLVM recebe como entrada um programa nesta representação. O objetivo da LLVM-IR é (ADVE et al., 2003): ser um alvo fácil de geração de código; ser independente da linguagem fonte ou da linguagem alvo; suportar diversos tipos de análises e transformações eficientemente. Nos componentes

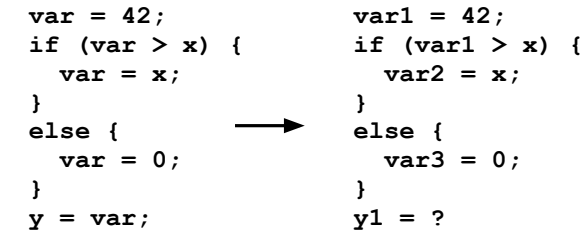
do LLVM, o código na RI aparece em 3 modos diferentes: representação em memória, representação binária para armazenamento externo e uma representação textual para manipulação manual.

A LLVM-IR é uma representação com estilo de linguagem de montagem, não contêm construções complexas de fluxo, somente saltos. Ela é próxima de uma linguagem de montagem, porém suficientemente "alto nível" para que o conjunto de análises e otimizações tradicionais presentes na literatura possam ser implementados sem dificuldade (LATTNER, 2002). A linguagem utiliza instruções de 3 endereços, semelhante a uma arquitetura RISC. Entretanto, contêm instruções de mais alto nível, para que possa implementar funcionalidades mais específicas.

Ao contrário de variáveis normais, esta linguagem utiliza um conjunto infinito de registradores virtuais que só podem ser declarados e definidos uma única vez. Cada instrução que realiza uma operação gera um outro valor e nunca modifica os seus operandos. Esta representação, denominada de *Single Static Assignment* (SSA), é bastante comum (AHO et al., 2007; APPEL; GINSBURG, 1998). Aho et al. (2007) define a SSA como sendo uma forma de representação intermediária onde todas as atribuições são para nomes diferentes. O uso do SSA facilita o desenvolvimento de análises do fluxo de dados do programa.



(a) Exemplo 1



(b) Exemplo 2

Figura 9: Exemplo de tradução de código de 3 endereços para a forma SSA.

Considere os exemplos da figura 9. No primeiro caso, a conversão entre as duas representações é direta. No segundo exemplo, não é possível determinar qual será o valor origem de `y` pois `var` pode ser definida em dois *blocos básicos*[3] diferentes. As representa-

[3]Bloco básico é um trecho de código que tem um único ponto de entrada e um único ponto de saída. Isto é, somente a primeira instrução do bloco pode ser o destino de um salto, e somente a sua última instrução pode ser um salto. Podem haver chamadas de subrotinas dentro do bloco.

ções SSA lidam com este caso utilizando uma função especial, representada por ϕ. Para o exemplo 2, a última linha seria: `y1 = `ϕ`(var2, var3)`. Esta é uma instrução fictícia, que retorna um valor dentre todos os possíveis valores de uma variável, dependendo do fluxo tomado no programa (APPEL; GINSBURG, 1998). Ela é eliminada no passo de geração de código.

A LLVM-IR é uma linguagem tipada, com construções de tipos compostos e operadores de conversão (*cast*) entre tipos. Suporta inteiros, boleanos, números em ponto flutuante, arranjos, vetores (de tamanho fixo), estruturas compostas e ponteiros (LATTNER, 2002). Apresentamos a seguir um resumo da estrutura dos programas nesta linguagem, como definido em (LLVM..., 2002). É necessário se familiarizar com a LLVM-IR para entender o funcionamento do módulo `risco-llvm`.

Um programa LLVM é definido como um *Módulo* e equivale a uma unidade de compilação. Cada módulo é composto por: funções, variáveis globais e declarações de símbolos. Os módulos podem ser combinados com o ligador do LLVM (`llvm-ld`), que realiza otimizações em tempo de ligação. Uma função é uma lista de blocos básicos, cada um iniciado com um rótulo. Um bloco básico é uma lista de instruções, seguindo as restrições apresentadas anteriormente. Todos os valores na linguagem são tipados. Os tipos podem ser classificados em 4 classes: inteiros (um tipo para cada quantidade de bits no inteiro), ponto flutuante, primitivos (rótulos, `void`) e derivados (arranjos, funções, ponteiros, etc.). As instruções podem ser classificadas em:

- *Instruções de término*: São as instruções que devem aparecer no final de um bloco básico, manipulando o fluxo de controle no programa. Fazem parte destas: `ret` (retorno de subrotina), `br` (salto direto, opcionalmente condicional), `indirectbr` (salto com endereçamento indireto) e `invoke` (chamada de rotina).

- *Operações binárias*: realizam as operações aritméticas e lógicas do programa. Cada instrução tem dois operandos do mesmo tipo e produz um único valor. Exemplos deste tipo são: `add`, `sub`, `mul`, `udiv` e `shl` (deslocamento para esquerda).

- *Operações vetoriais*: trabalham com operandos que representam um vetor de valores, úteis para arquiteturas SIMD. Existem operações para extrair e modificar elementos do vetor, por exemplo.

- *Operações de conversão*: convertem um valor de um tipo para outro tipo. Inclui operações de truncamento (`trunc..to`) e extensão com ou sem sinal (`zext..to` e `sext..to`).

```
int f(int* n) {
  return *n + 1;
}

int main(void) {
  int x;
  x = 4;
  return f(&x);
}
```

(a) C

```
int %f(int* %n) {
 entry:
  %tmp.1 = load int* %X
  %tmp.2 = add int %tmp.1, 1
  ret int %tmp.2
}
int %main() {
 entry:
  %x = alloca int
  store int 4, int* %x
  %tmp.3 = call int %f(int* %x)
  ret int %tmp.3
}
```

(b) IR

Figura 10: Exemplo de tradução de um programa C para a LLVM-IR

- *Instruções de acesso a memória*: as posições de memória fogem da representação SSA utilizada para os valores primitivos e são manipuladas com instruções `load` e `store`, que recebem ponteiros como argumentos. Existe uma instrução especial para alocar um valor na pilha (`alloca`).
- *Instrução* ***phi***: tem o significado da função ϕ discutida anteriormente.

A figura 10 mostra um exemplo de tradução de um código C para a representação intermediária do LLVM. Note a utilização da forma SSA e das instruções de 3 endereços. A LLVM-IR tem as construções baixo nível necessárias para representar as operações com detalhes e, ao mesmo tempo, mantêm informações de tipos de ponteiros, necessárias para as otimizações mais agressivas (ADVE et al., 2003).

5.3 Framework para geração de código

O módulo de geração de código do LLVM consiste de um conjunto de componentes reutilizáveis responsáveis pela tradução de código na representação intermediária para o código de máquina da arquitetura alvo em forma de linguagem de montagem ou diretamente para o código binário executável. A segunda forma está relacionada com a implementação das otimizações em tempo de execução do LLVM, discutidas em 5.1, e está fora do escopo do trabalho. Será analisada somente a geração de código em linguagem de montagem.

O framework em si é escrito em C++ e, como o compilador Clang, é implementado como uma série de *passes* de análise e transformação, responsáveis por gradualmente produzir o código de máquina final. A geração de código ocorre após todos os passes de otimização na linguagem intermediária terem sido executados. Todos os seus componentes são facilmente substituíveis, incluindo o alocador de registradores (existem 4 opções disponíveis), o seletor de instruções e o escalonador de instruções. Existem 4 tipos de componentes no gerador que são relevantes para o RISCO (LOPES, 2009c):

- Um conjunto de interfaces abstratas que especificam propriedades da arquitetura alvo e são implementadas por cada módulo dos alvos suportados pelo LLVM. Para uniformizar as notações utilizadas no texto, todo o código específico a um alvo será referido como sendo um dos "backends" do LLVM.
- Classes que modelam o código de máquina em si, independentemente da arquitetura alvo. Por exemplo: instruções, funções e registradores.
- Algoritmos independentes do alvo, implementados como passes nas representações intermediárias do gerador de código. Aqui pertencem os algoritmos de alocação de registradores, por exemplo.
- Os diversos backends pertencentes ao projeto LLVM. São os aglomerados de classes e descrições que dão suporte a uma determinada arquitetura alvo, distribuídos como bibliotecas dinâmicas (LEVINE, 1999). O `risco-llvm` é um dos backends do LLVM.

O processo de geração de código em si é dividido em 7 passos (LLVM..., 2002), mencionados abaixo. Além destes, um backend em particular pode adicionar mais passos no processo, para algum requisito específico. Esta técnica é utilizada pelo backend RISCO.

1. **Seleção de instruções**: Descobre como representar as instruções da LLVM-IR utilizando as instruções da arquitetura alvo. Transforma o código original em um grafo de instruções da máquina.
2. **Escalonamento de instruções**: Tendo como entrada o grafo de instruções de máquina, esta etapa determina uma ordenação das instruções que atenda as restrições da plataforma alvo. O código é emitido na forma SSA (infinitos registradores).
3. **Otimizações na forma SSA**: É uma etapa opcional com otimizações específicas ao alvo, comumente denominadas de otimizações *peephole* (AHO et al., 2007) pois trabalham em pequenos trechos sequenciais do código.

4. **Alocação de registradores**: É o passo que elimina as referências para registradores virtuais, escolhe os registradores físicos correspondentes e emite código para realizar leitura e escrita a memória, quando necessário.

5. **Emissão do prólogo e do epílogo**: Após a alocação de registradores, o gerador já sabe qual vai ser o tamanho do quadro da função na pilha e pode emitir o código de prólogo e epílogo da função.

6. **Otimizações finais**: É o último passo de otimização e trabalha com uma ordenação final das instruções. Serve para eliminar as últimas redundâncias que não foram encontradas nos passes de otimização anteriores.

7. **Emissão do código**: É o passo que de fato emite o código, escrevendo o arquivo de texto final utilizando a sintaxe do montador alvo.

Nas seções a seguir, são discutidos em mais detalhes os componentes pertencentes ao gerador de código que são relevantes para o desenvolvimento de um novo backend.

5.3.1 Descrição do alvo

As descrições de todas as características da arquitetura alvo são concentradas em um conjunto de classes que implementam interfaces abstratas em C++, definidas no framework LLVM. Os métodos virtuais destas interfaces permitem que a parte genérica do gerador de código consiga trabalhar com os detalhes de cada plataforma alvo. A implementação de certas interfaces para cada backend LLVM é uma tarefa mecânica, envolvendo a definição de muitos métodos que retornam informações simples ou repetitivas (LOPES, 2009b). Para introduzir diversas funcionalidades que facilitam este tipo de tarefa, o projeto LLVM utiliza uma linguagem de descrição de dados denominada *TableGen* (LLVM..., 2002).

Uma especificação no formato TableGen consiste de um arquivo contendo descrições textuais de dois tipos de entidades básicas: registros e classes. Um registro é um identificador associado a uma lista de atributos e seus respectivos valores. Uma classe funciona como um template para a construção de registros, agregando informações que se repetem entre diversos registros. Os valores podem ser de tipos primitivos, como inteiros, boleanos e strings, como também de tipos compostos, como grafos completos. Por exemplo: pode-se definir uma classe `Instrução`, que tem como parâmetros um inteiro, correspondente ao seu `código` e uma string com o seu `mnemônico`. Após essa definição, podemos especificar

um registro para cada tipo de instrução, instanciando a classe `Instrução` com os dados de cada uma. Existem outras facilidades na ferramenta TableGen que também auxiliam a reduzir o tamanho das definições das classes de descrição dos alvos.

As interfaces que devem ser implementadas pelos backends do LLVM permitem a obtenção de informações diversas sobre o alvo, tais como (LOPES, 2009c): layout de memória dos executáveis, alinhamento dos tipos de dados básicos, tamanhos dos ponteiros; informações sobre os registradores da máquina, seus identificadores, tamanhos, restrições e sub-registradores; informações sobre as instruções da plataforma, seus mnemônicos, número e tipo dos operandos, modos de endereçamento, padrões para seleção de instruções, etc.

5.3.2 Seleção de Instruções

Como já foi mencionado, a etapa de seleção de instruções é responsável por transformar as instruções da representação intermediária em instruções da plataforma alvo. A tradução não é realizada a cada instrução, buscando uma equivalente na plataforma alvo, pois este tipo de seleção gera código bastante ineficiente (APPEL; GINSBURG, 1998). O motivo é que nem todas as instruções da LLVM-IR sempre têm uma contrapartida na plataforma alvo. Mais ainda, em alguns casos, existem diversas maneiras diferentes de traduzir uma instrução da RI. A escolha a ser tomada pode depender do contexto em que as instruções serão inseridas. Como foi visto no capítulo 3, um dos papéis de um backend LLVM é disponibilizar um conjunto de padrões de seleção de instruções da RI para instruções da máquina. O seletor de instruções pode utilizar as informações deste conjunto para guiar as suas decisões.

O seletor de instruções trabalha com uma representação do código na forma de um *grafo acíclico direcionado*, denominado de `SelectionDAG` (do inglês *Directed Acyclic Graph*). O SelectionDAG é uma abstração do código na representação intermediária. Os nós do grafo são pertencentes a duas classes: nós de *operação* e nós de *ordenação*. Os primeiros representam uma operação do código, onde os seus operandos são arestas direcionadas para os nós que definem os operandos. Note que esta definição segue diretamente do código, pela sua representação SSA na LLVM-IR. Os nós de ordenação são nós fictícios inseridos na etapa de construção do grafo que servem para determinar uma ordenação parcial para os nós do grafo que respeite a ordem original dos acessos a memória com as instruções `load` e `store`, que não obedecem a forma SSA.

Aho et al. (2007) demonstra a utilização de uma representação em forma de DAG

para cada expressão de um programa (de modo a identificar sub-expressões em comum) diferentemente da abordagem utilizada pelo LLVM, onde um bloco básico inteiro pode ser considerado como um DAG. O processo de seleção de instruções abordado em (AHO et al., 2007) e (APPEL; GINSBURG, 1998) é baseado em reescrita de árvores, ao contrário de grafos. Ambos abordam um algoritmo guloso, denominado *maximal munch*, e um algoritmo baseado em programação dinâmica que consegue determinar a seleção ótima de instruções, baseando-se nos custos de cada padrão.

Algoritmo 4 MaximalMunch

Entrada: **Node**, o nó raiz da árvore; **Patterns**, a lista de padrões da arquitetura alvo
Saída: **Node** é modificado para utilizar as instruções selecionadas.

```
se Node não é uma instrução ou expressão nativa então
  P ← {pat ∈ Patterns | casa(pat, Node)}
  pat ← o padrão em P com maior número de nós, ignorando empates
  reescrever(node, pat)
fim se
para todo next ∈ filhos(Node) faça
  MaximalMunch(next)
fim para
```

Em geral, o problema de encontrar a seleção de instruções ótima para grafos acíclicos direcionados é NP-completo (KOES; GOLDSTEIN, 2008), ao contrário do que foi visto para árvores. O algoritmo utilizado pelo LLVM é uma variação do algoritmo guloso maximal munch, adaptado para trabalhar com o DAG (LATTNER, 2002). O funcionamento básico do algoritmo (para árvores) é descrito no exemplo 4, adaptado de (APPEL; GINSBURG, 1998). Koes e Goldstein (2008) apresenta um trabalho de desenvolvimento de um novo seletor de instruções para o LLVM, capaz de encontrar soluções "quase ótimas". Não é do nosso conhecimento se este seletor já está em uso na versão oficial do LLVM. O processo de seleção do LLVM, além do algoritmo propriamente dito, apresenta as seguintes etapas (LLVM..., 2002):

1. *Construção do DAG*: a primeira versão do grafo de seleção é construída diretamente do código em LLVM-IR. Este grafo é denominado "ilegal" pois utiliza instruções e tipos de dados que não são suportados pela plataforma alvo.

2. *Otimizações genéricas (A)*: Um passe que realiza simplificações no grafo é executado.

3. *Legalização*: O passe de legalização é responsável por transformar o SelectionDAG e eliminar o uso de instruções e tipos de dados não pertencentes no alvo. O grafo é reescrito pelo backend.

4. *Otimizações genéricas (B)*: O passe de otimização genérica é executado novamente para eliminar as redundâncias que possam vir a ser introduzidas pelo processo de legalização do grafo.

5. *Seleção*: Finalmente, o algoritmo de seleção de instruções é executado, transformando o SelectionDAG em um grafo em que todos os nós de operação utilizam instruções da arquitetura alvo. Esta seleção é baseada nos padrões disponibilizados pelo backend.

6. *Escalonamento*: Este passe decide uma ordenação completa para as instruções do grafo de seleção legalizado.

Após este processo, o código já utiliza instruções da plataforma alvo, porém ainda utiliza registradores virtuais e apresenta redundâncias que serão eliminadas em passes de otimização posteriores, como já foi descrito.

5.3.3 Emissão de código

Após a alocação de registradores e os últimos passes de otimização no código de máquina, o gerador de código do LLVM irá emitir um arquivo texto contendo o código final em linguagem de máquina. Para poder produzir este arquivo, o gerador consulta uma classe do backend responsável por informar as características sintáticas da linguagem de montagem. O componente emissor assume um conjunto mínimo de diretivas e construções para: declarar e definir dados, instruções, macros, rótulos, etc. A linguagem descrita em 4.1.1 atende a estes requisitos.

Nos próximos capítulos será discutido o trabalho que foi desenvolvido utilizando o projeto LLVM e o seu framework de geração de código, exposto nesta seção.

6 Módulo RISCO para o LLVM

O backend RISCO para o projeto LLVM é o módulo que permite o desenvolvimento de software em uma linguagem de alto nível tendo como alvo o processador RISCO. Como argumentado no capítulo 2, o uso de uma linguagem de programação estruturada é imprescindível para o desenvolvimento moderno de sistemas embarcados. Além disso, a existência deste suporte permite que o RISCO se torne uma opção viável para o estudo de arquitetura de computadores e de compiladores em geral. O sistema de instruções (3.2) desta plataforma, seguindo a filosofia RISC, é simples, conciso e completo. O usuário final, utilizando o framework LLVM e o módulo RISCO, pode observar todos os passos envolvidos na tradução do código original em linguagem de alto nível para a linguagem de montagem do RISCO.

Os motivos da escolha do LLVM para esta tarefa foram apresentados no capítulo 5. Em teoria, como o backend RISCO habilita a tradução de código na linguagem intermediária LLVM-IR para a sua linguagem de montagem, todas as ferramentas descritas na arquitetura do LLVM da figura 8 podem funcionar em uma plataforma RISCO. Na prática, o suporte a estas ferramentas requer mais do que a implementação deste passo de tradução, pois é preciso portar as ferramentas em si para a arquitetura desejada. Neste trabalho procuramos criar somente o que foi necessário para a compilação das linguagens de alto nível com um compilador estático, que não inclui otimizações em tempo de execução (LATTNER; ADVE, 2004a). O foco principal do backend RISCO é o suporte a compilação de programas nas linguagens C e C++. Essas linguagens foram escolhidas pois, além de serem as linguagens que dominam o desenvolvimento de sistemas embarcados (NOERGAARD, 2005), são as que tem o suporte mais estável no projeto LLVM (LLVM..., 2002). O compilador Clang é capaz de gerar código LLVM-IR a partir de C e C++, e é utilizado na ferramenta `risco-c`, que será descrita posteriormente.

A apresentação do projeto e desenvolvimento do backend RISCO se dá em 3 partes: (i) decisões de projeto, (ii) especificação do backend e (iii) a etapa de verificação do código gerado.

6.1 Decisões de projeto

A implementação do módulo RISCO para o LLVM envolveu algumas decisões de projeto necessárias para o entendimento da arquitetura da solução, assim como para a utilização correta da ferramenta. Antes de implementar os padrões de tradução que serão abordados na seção 6.2 é preciso decidir, entre outras coisas, como será o processo de desenvolvimento, como o módulo será utilizado pelo usuário final e quais instruções serão suportadas.

O módulo foi desenvolvido tendo como base os outros backends do LLVM cuja plataforma alvo seja um processador da família RISC. Em particular, o código de suporte ao MIPS e ao ARM, descritos por Lopes (2009b), foram de importância significativa, pois estes dois conjuntos de instruções apresentam várias semelhanças com relação às instruções RISCO. O backend RISCO é invocado pelo usuário final utilizando um *driver* do compilador. Hyde (2010) define um driver de compilador como sendo o programa que se encarrega de coordenar a execução das diferentes ferramentas necessárias para a compilação de um código em linguagem de alto nível até o binário executável final. Geralmente estas ferramentas são o pré-processador C, o compilador, o montador e o ligador. O driver ordena a execução de cada ferramenta e direciona a sua saída para a próxima. Um exemplo de driver de compilador C em ambiente Unix é o programa `gcc`, do projeto GCC (GCC..., 1987).

Foi desenvolvido um driver de compilador próprio para o RISCO, denominado `risco-c`. Trata-se de um script na linguagem Python (ROSSUM, 1995) que executa os seguintes programas:

- `clang`: Responsável por emitir um módulo LLVM-IR para cada unidade de compilação especificada na entrada do `risco-c`.
- `opt`: É a ferramenta do LLVM responsável por executar os passes de análise e otimização na representação intermediária do código. Ela é executada para cada módulo LLVM-IR, modificando o seu conteúdo. Segundo (LOPES, 2009c), em 2009 o LLVM contava com 32 passes de análise e 63 passes de otimização. Os argumentos passados para o `opt` permitem especificar quais desses passes serão executados sob o código, levando em conta os objetivos finais. Por exemplo, o conjunto de passes executados pode ser diferente para uma compilação que tenta minimizar o tamanho final do código, com relação a outra que tenta maximizar a sua performance final. Alguns passes de otimização específicos ao backend RISCO também são executados nesta

etapa.

- `llc`: É o compilador da linguagem de representação intermediária, incluído com o LLVM. É responsável por ler o código em RI e gerar o código final em linguagem de montagem. O `risco-c` passa um argumento para o `llc` informando que o backend RISCO deverá ser utilizado.

- `risco-peep-opt`: A geração de código utilizando o backend RISCO ainda deixa algumas oportunidades de otimização *peephole* no código final, como será explicado na seção 6.2. Este script foi desenvolvido para realizar pequenas modificações no código final em linguagem de montagem, antes da execução do montador.

- `risco-as`: Finalmente, o driver executa o montador/ligador discutido no capítulo 4. Ele recebe como entrada os arquivos em linguagem de montagem de todas as unidades de compilação e gera o binário executável final, que pode ser passado ao simulador `risco-sim` posteriormente.

Tanto o backend RISCO como os scripts que o auxiliam foram desenvolvidos com uma variação do processo de desenvolvimento dirigido a testes, como descrito em (BECK, 2002). Ele é apresentado em mais detalhes na seção 6.3. O `risco-c` facilita o processo de verificação do compilador fazendo com que só seja necessário um comando para a execução do processo de compilação.

Nas próximas seções serão discutidas as decisões de projeto relacionadas ao código que será gerado pelo backend RISCO.

6.1.1 Suporte às funcionalidades da LLVM-IR

A linguagem de representação intermediária do projeto LLVM, como descrita em 5.2, contêm diversas funcionalidades que não são diretamente relevantes para a plataforma RISCO ou que não podem ser mapeadas para ela. Dessa maneira, o módulo RISCO dá suporte a um subconjunto dessas funcionalidades, diminuindo a complexidade da sua implementação.

Com relação aos atributos de um módulo LLVM-IR, o backend RISCO atualmente ignora os diferentes tipos de ligação de símbolos que não sejam *private* , *global* ou *extern*. Entre esses, estão: *linkonce*, *weak*, *appending* e *weak_ odr*. Estes atributos são funcionalidades avançadas dos ligadores, servindo para a implementação eficiente de algumas construções de linguagens de programação avançadas, como os templates do C++ (LEVINE,

1999). Além destes, os atributos que contêm as informações de depuração do programa, em forma de metadados da LLVM-IR, também são ignorados pois a plataforma RISCO ainda não conta com o suporte de um depurador.

A maioria das instruções da RI é suportada, com as seguintes exceções:

- As instruções com ponto flutuante não podem ser facilmente sintetizadas para a plataforma RISCO pois não há suporte a este tipo de dados no processador. Portanto, a sua utilização causa um erro em tempo de compilação.
- A LLVM-IR introduziu o conceito de *funções intrínsecas* (ADVE et al., 2003) para representar instruções complexas da arquitetura base, tais como manipulação avançada de bits, funções trigonométricas, operações atômicas na memória principal e suporte a listas variáveis de argumentos nas subrotinas. O módulo RISCO não dá suporte a estas instruções.
- As instruções de multiplicação, divisão e resto são suportadas indiretamente pelo backend RISCO, utilizando implementações em software. Foi criado um passe de transformação de código para o LLVM, denominado `MulDivTransform`, que identifica estas operações no código intermediário e as transforma em chamadas para as suas respectivas funções. As funções tem complexidade de pior caso $O(1)$ com relação aos valores dos parâmetros e complexidade $O(n^2)$ com relação ao número de bits dos parâmetros. Em geral, elas são eficientes o bastante para uso geral, porém ineficientes para aplicações numéricas.

6.1.2 Interface binária

Antes de começar a implementação de um compilador para uma nova arquitetura é preciso decidir como será a "organização" do programa em tempo de execução, e como ele utiliza cada um dos registradores da arquitetura. A organização do programa diz respeito a como ele utiliza a região de memória denominada de *pilha* para manter os *registros de ativação* (AHO et al., 2007) das rotinas sendo executadas. Já o uso dos registradores indica qual o protocolo de utilização dos registradores quando diversas funções executam sob a mesma área de memória. Estas duas convenções são comumente denominadas de *convenção de chamada* (HENNESSY; PATTERSON, 2003) da plataforma. O processador em si é independente da convenção de chamada utilizada, porém a definição dela se faz necessária para a geração automática de código que possa ser reutilizado em outra compilação. Por exemplo, o código de uma biblioteca necessita utilizar a mesma convenção

do código que utiliza a biblioteca. Caso contrário, haverá uma incompatibilidade entre os binários e o código gerado não será correto.

A convenção de chamada descrita nesta seção é específica para sistemas de 32 bits, e foi nomeada *RISCO32*. Na prática, as arquiteturas RISC em geral utilizam convenções de chamada semelhantes (LEVINE, 1999). A principal exceção é a arquitetura SparcV8, cujo conceito de janela de registradores (SPARC, 1993) influencia a convenção. A RISCO32 é baseada na convenção utilizada pela arquitetura MIPS (LOPES, 2009b), com adaptações para lidar com diferenças nos registradores e na instrução de chamada de subrotinas.

Uma convenção de chamada dita regras para as instruções que serão geradas em 4 pontos do código. Assumindo que uma função pai `F` quer chamar a subrotina `G`, a convenção atuaria em (HENNESSY; PATTERSON, 2003): (1) antes de `F` executar a instrução de chamada da subrotina; (2) no início da execução de `G`; (3) antes que `G` execute uma instrução de retorno; (4) assim que a função `F` reinicia a sua execução. A convenção de chamada divide o conjunto de registradores em 2 classes:

- Salvo pela rotina pai (do inglês *caller-saved*): são registradores de características temporária, geralmente sobrescritos pela rotina que será chamada. Caso a rotina pai queira preservar o valor dos registradores para utilizá-los após a chamada, ela deve salvá-los na pilha e recuperá-los após a chamada.
- Salvo pela rotina chamada (do inglês *callee-saved*): são registradores de característica persistente, cujos valores devem se manter ao longo das chamadas de subrotinas. Caso uma rotina queira escrever em um registrador deste tipo, ela deve primeiramente salvar o seu valor antigo na pilha, e recuperá-lo antes de retornar para a rotina pai.

Os registradores da plataforma RISCO foram classificados nestas duas classes, conforme a tabela 3. A tabela também informa quais são os identificadores utilizados para referenciar os registradores na linguagem de montagem.

A convenção de chamada é detalhada a seguir. Em cada passo, o número entre colchetes indica qual dos 4 pontos do código mencionados anteriormente está sendo referenciado no momento. Para um melhor entendimento da convenção é necessário conhecimento sobre o conceito de registro de ativação de uma rotina (AHO et al., 2007).

- [1] Na subrotina pai, cada registrador caller-saved cujo valor será necessário após a chamada da subrotina filha deve ser salvo na pilha, em qualquer ordem.

Nome no risco-as	Código	Classe	Comentário
`$zero`	R00	(N/A)	Sempre guarda 0
`$psw`	R01	caller-saved	Palavra de status
`$v0`	R02	caller-saved	Valor de retorno de uma subrotina
`$a0 - $a4`	R03 - R07	caller-saved	Argumentos da rotina
`$t0 - $t9`	R08 - R17	caller-saved	Propósito geral
`$s0 - $s7`	R18 - R25	callee-saved	Propósito geral
(N/A)	R26 - R28	(N/A)	Reservados
`$fp`	R29	callee-saved	Ponteiro do registro
`$pc`	R31	(N/A)	Contador do programa

Tabela 3: Convenções de uso para os registradores RISCO

- [1] Os 5 primeiros argumentos da subrotina são postos nos registradores `$a0` a `$a4`. Os argumentos que não cabem em um registrador e os demais que não foram para os registradores são postos na pilha, na ordem em que foram declarados.

- [2] Ao iniciar a sua execução, a subrotina primeiramente reserva a memória na pilha com o tamanho do seu registro de ativação. A pilha cresce dos valores mais altos para os mais baixos, e o espaço é alocado decrementando o registrador `$sp`. Ela também salva o antigo valor do registrador `$fp` e guarda nele a posição da pilha onde se inicia o registro de ativação da subrotina atual.

- [2] A subrotina chamada salva na pilha os registradores callee-saved que terá que usar posteriormente. Além disso, ela utiliza o espaço do seu registro na pilha para guardar as suas variáveis locais, caso seja necessário.

- [3] Após a execução da subrotina chamada, ela salva o valor que será retornado a subrotina pai no registrador `$v0`. Caso o valor não caiba em um registrador, ele é retornado na pilha.

- [3] Antes de executar a instrução de retorno, a subrotina chamada também recupera os registradores callee-saved que foram salvos anteriormente.

- [4] A subrotina pai recupera o valor dos registradores caller-saved que foram salvos na pilha.

A figura 11 representa o estado da pilha no momento da execução de uma subrotina chamada por uma rotina pai, como nos exemplos dados acima. Note que, realizando um acesso a pilha com endereçamento indireto tendo como base o registrador `$fp`, é possível

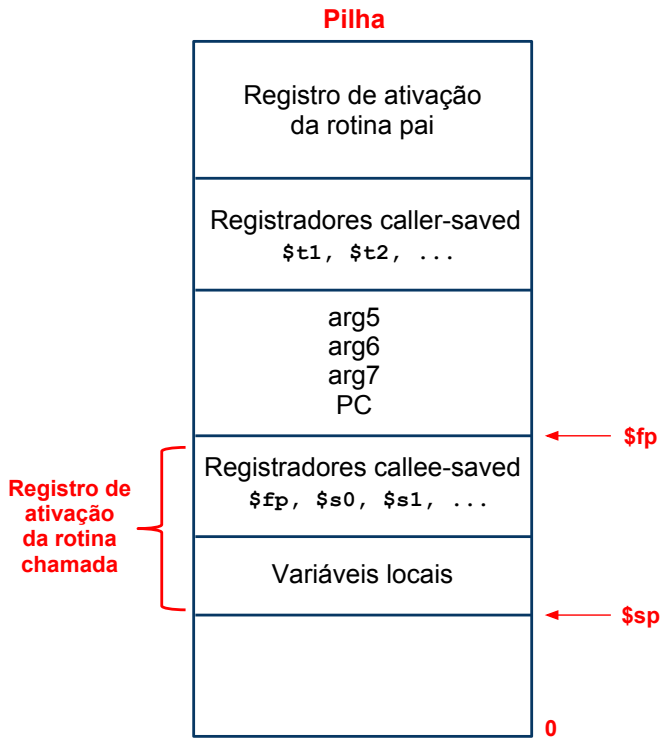


Figura 11: Registro de ativação de uma rotina na convenção RISCO32

acessar os argumentos e as variáveis locais com deslocamentos fixos, facilitando a geração de código pelo compilador.

6.2 Especificação do backend

Com a definição dos detalhes da interface binária RISCO32 pronta, foi possível a implementação do módulo RISCO para o LLVM. Como visto em 5.3, um *backend* LLVM é uma biblioteca dinâmica contendo classes que implementam interfaces abstratas, utilizadas pela parte do gerador de código que é independente da arquitetura alvo (LOPES, 2009c). O gerador utiliza estas classes para descobrir as características específicas da arquitetura, utilizando-as durante os algoritmos de geração de código. Dessa maneira, tais algoritmos são genéricos o bastante para serem reutilizados em todas as arquiteturas alvo do LLVM (LATTNER; ADVE, 2004a). Detalhes sobre os componentes do módulo são descritos nas seções 6.2.1 a 6.2.5.

6.2.1 TargetMachine

O componente `RISCOTargetMachine` implementa a interface `LLVMTargetMachine` e é o ponto de principal de comunicação entre o framework de geração de código e o módulo RISCO. Além de conter métodos de acesso para todos os outros componentes do módulo, ele tem as seguintes funcionalidades:

- O `TargetMachine` informa ao gerador de código como os dados são representados na arquitetura. Isto inclui tamanho da palavra, alinhamento dos bytes na memória e tamanho de um ponteiro. Para o RISCO, todos os dados são representados em 4 bytes, organizados na memória com a codificação *little-endian*.
- Este componente pode informar ao framework LLVM que existem diferentes versões do processador, cada qual com algumas funcionalidades presentes ou omitidas. Para o RISCO, só foi informado um modelo do processador, o que foi descrito no capítulo 3.
- Finalmente, é neste componente que o módulo RISCO é registrado no framework LLVM. Após o passo de registração, o módulo se torna visível nas ferramentas de geração de código (como o `llc`).

6.2.2 RegisterInfo e CallingConv

Estes componentes são responsáveis pela descrição do conjunto de registradores da plataforma alvo, junto com as suas propriedades, e as convenções de chamada que são suportadas. Juntas, elas informam ao gerador de código as principais características da interface binária RISCO32.

Os registradores podem ser divididos em classes de acordo com o tipo de dados que eles guardam. No RISCO, todos são da mesma classe (guardam sempre uma palavra inteira). Também é possível especificar o conceito de sub-registradores, existentes, por exemplo, nos processadores X86 (SHANLEY, 2010), e informações de depuração associadas a registradores. Estas funcionalidades não foram necessárias para a especificação dos registradores RISCO. Devido a alta repetição de informações na especificação de cada registrador, o código para a classe `RISCORegisterInfo` é gerado automaticamente a partir de uma especificação *TablenGen*, como foi visto em 5.3.1. O exemplo 5 mostra um trecho simplificado da especificação TableGen `RISCORegisterInfo`.

Algoritmo 5 Trecho simplificado da especificação dos registradores RISCO

```
class RiscoReg<bits<5> num, string n> : Register<n> {
  field bits<5> Num;
  let Num = num;
}
let Namespace = "Risco" in {
  ...
  def A0   : RiscoReg< 4, "$a0">;
  def A1   : RiscoReg< 5, "$a1">;
  def A2   : RiscoReg< 6, "$a2">;
  def A3   : RiscoReg< 7, "$a3">;
  ...
}
```

A especificação da convenção de chamada inicialmente era feita utilizando código C++ escrito a mão, descrevendo as regras da convenção (LLVM..., 2002). Recentemente, foi implementado o suporte à geração automática do módulo C++ a partir de uma especificação TableGen, assim como o `RegisterInfo`. A figura 6 mostra um trecho da especificação para o RISCO32. A classe `CallingConv` consiste de uma lista de ações que são analisadas em ordem para cada argumento ou valor de retorno de uma subrotina. O registro `RetCC_RISCO32` descreve a convenção para os valores de retorno, e o `CC_RISCO32` descreve a convenção para a chamada de subrotina com argumentos.

Algoritmo 6 Trecho da especificação da convenção de chamada RISCO32

```
def RetCC_RISCO32 : CallingConv<[
  CCIfType<[i32], CCAssignToReg<[V0]>>,
  CCAssignToStack<4, 4>
]>;

def CC_RISCO32 : CallingConv<[
  CCIfType<[i8, i16], CCPromoteToType<i32>>,
  CCIfType<[i32], CCAssignToReg<[A0, A1, A2, A3, A4]>>,
  CCAssignToStack<4, 4>
]>;
```

6.2.3 TargetLowering

Este é o componente inicial na etapa de seleção de instruções. De acordo com o processo apresentado na seção 5.3.2, a seleção de instruções se dá sobre um grafo direcionado acíclico denominado *SelectionDAG*. A etapa de seleção em si transforma o SelectionDAG

em outro grafo "legalizado", que utiliza somente operações e tipos de dados presentes na arquitetura alvo. Entretanto, antes da execução do algoritmo de seleção, o SelectionDAG original (criado a partir do código em LLVM-IR) tem alguns nós substituídos por nós específicos do backend. A introdução destes nós simplifica a especificação dos padrões de reescrita de instruções, que serão discutidos na seção 6.2.4.

Apenas parte dos nós específicos utilizados são obrigatórios, sendo estes: `JmpCall`, representando uma chamada de subrotina, e `Ret`, representando o retorno. Os nós opcionais escolhidos foram `Hi` e `Lo`. São nós de operação, representando a obtenção dos 16 bits mais ou menos significativos, respectivamente, de um número. A introdução destes nós facilitou o desenvolvimento dos padrões de reescrita para as instruções com formato 4 (seção 3.2).

6.2.4 InstrInfo

O componente `RISCOInstrInfo` contêm a especificação de todas as instruções que o LLVM pode utilizar na emissão de código para a plataforma RISCO, junto com algumas propriedades e os seus padrões de seleção.

É interessante notar que não existe uma bijeção entre o conjunto de instruções RISCO que foram especificadas no backend LLVM e o conjunto de instruções reais da plataforma, apresentadas no apêndice A. Este fato está relacionado com uma diferença na organização do conjunto de instruções do RISCO com o das outras arquiteturas RISC comuns, como o MIPS (HENNESSY et al., 1981). Na codificação de uma instrução RISCO há uma separação física e semântica entre a definição da instrução e a definição dos seus operandos (JUNQUEIRA; SUZIM, 1993). Esta separação faz com que seja possível utilizar qualquer combinação de instrução $\times$ operandos, dentre os 5 formatos de operandos existentes. Pode-se dizer que as instruções RISCO apresentam um tipo de *polimorfismo* básico pois são independentes dos tipos dos seus operandos. Como discutido na seção 3.2, algumas destas combinações geram instruções que não apresentam aplicação prática.

O impacto desta característica na implementação do módulo RISCO é que grande parte das instruções reais deve ser mapeada para no mínimo duas instruções no backend LLVM: uma versão com dois operandos em registradores e outra com um operando em registrador e outro operando imediato (constante). Por exemplo, a instrução `add` aparece no componente `RISCOInstrInfo` como `ADDrr` e `ADDri`. Estas duas formas cobrem as instruções com formato 1 e 2. Os formatos 3 e 5 são identificados pelo montador `risco-as`, que é capaz de analisar os operandos da instrução e determinar a codificação mais adequada para cada caso. As instruções de formato 4 são um caso especial, descrito mais

adiante nesta seção.

A especificação de uma instrução em um backend LLVM consiste de: nome da instrução; um template, em forma de string, para a geração de código em linguagem de montagem, especificando como a instrução e os seus operandos devem ser impressos no arquivo final; especificação do nó que representa a instrução no SelectionDAG incluindo, por exemplo, o número de arestas de saída para operandos e de entrada para os seus resultados; e por fim o padrão de reescrita, que especifica qual parte do grafo SelectionDAG original pode ser substituído por esse nó. Todo o código C++ equivalente a definição das instruções é gerado automaticamente a partir de uma especificação TableGen contendo estas informações.

Algoritmo 7 Especificação da instrução `ADDrr` no backend RISCO

```
let isCommutable = 1 in
class ArithR<bits<6> op, bits<6> func, string instr_asm, SDNode OpNode>:
  FR< op, func, (outs CPURegs:$dst), (ins CPURegs:$b, CPURegs:$c),
      instr_asm,
      [(set CPURegs:$dst, (OpNode CPURegs:$b, CPURegs:$c))]>;

def ADDrr : ArithR<0x00, 0x00, "add $dst, $b, $c", add>;
```

Primeiramente, são definidas classes TableGen para cada tipo de instrução e tipo de operandos. Por exemplo: uma classe para instruções aritmético-lógicas com dois operandos em registradores. Em seguida, cada instrução é definida por um registro que instancia a sua respectiva classe, preenchendo os valores necessários. O padrão de reescrita é definido em dois passos. Parte dele é comum a todas as instruções de uma certa classe e pode ser definido na própria classe. A parte que é particular a cada instrução é passada como parâmetro de instanciação da classe. Observe o exemplo 7 para a definição da instrução `ADDrr`. O padrão de reescrita é `(set CPURegs:$dst, (OpNode CPURegs:$b, CPURegs:$c))`, onde `OpNode` é instanciado com a operação `add` na instanciação do registro. Desta maneira, quando o seletor de instruções encontra este padrão no grafo SelectionDAG (uma adição entre dois registradores, com o resultado sendo escrito em outro registrador) ele poderá substituí-lo por uma instrução `ADDrr`. A string ‘‘`add $dst, $b, $c`’’ é o template utilizado para imprimir a instrução no código em linguagem de montagem.

A maior parte das instruções definidas tem correspondentes diretos na LLVM-IR, como o caso da instrução de adição. Entretanto, outras tem que ser modeladas como uma sequência de instruções RISCO. O exemplo 8 mostra 3 instruções com esta característica.

- `SLT`: É o padrão de reescrita para o nó `setlt` do SelectionDAG. A operação significa:

Algoritmo 8 Exemplos de instruções compostas na especificação `RISCOInstrInfo`

```
def SLT     : SetCC_R<0x00, 0x2a,
  "cmp ($b, $c)\n\t"
  "and $dst, $$psw, LT_MASK\n\t"
  "srl $dst, $dst, LT_POS",
  setlt>;

def LB      : LoadM<0x20,
  "ld $dst, $addr\n\t"
  "sll $dst, $dst, 24\n\t"
  "sra $dst, $dst, 24",
  sextloadi8>;

def BEQ     : CBranch<0x04,
  "cmp ($a, $b)\n\t"
  "jmpeq $offset",
  seteq>;
```

`$dst` recebe 1 caso `$b` $<$ `$c`, caso contrário recebe 0. Com instruções RISCO, é preciso comparar os dois números e realizar duas operações aritméticas para conseguir o mesmo resultado, como mostrado no exemplo.

- `LB`: É o padrão relativo ao nó `sextloadi8` do SelectionDAG, e trata-se de carregar um 1 byte da memória para um registrador, preservando sinal. Como no RISCO o acesso a memória só é possível para recuperar uma palavra inteira, é preciso carregar uma palavra e calcular o byte requisitado manualmente.

- `BEQ`: Especifica a operação de salto com a condição de igualdade entre dois operandos. No RISCO, são necessárias duas instruções: uma para a comparação e outra para o salto.

Algoritmo 9 Padrão de reescrita para o suporte a constantes de 32 bits

```
def : Pat<(i32 imm:$imm),
          (ORi (LUi (HI16 imm:$imm)), (LO16 imm:$imm))>;
```

A última seção da especificação contêm padrões de reescrita mais genéricos, que não correspondem a uma só instrução. Por exemplo, para carregar uma constante de 32 bits em um registrador com o RISCO são necessárias duas instruções: uma para carregar a parte mais significativa (instrução de formato 4) e outra para somar o resultado com a parte menos significativa. O padrão que especifica esta técnica é descrito no exemplo 9.

6.2.5 MCAsmInfo

O módulo `RISCOMCAsmInfo` é responsável por informar os elementos sintáticos da linguagem de montagem RISCO para o gerador de código. Por exemplo, as diretivas do montador apresentadas na subseção 4.1.1. Na impressão de cada instrução, o template contido na sua especificação TableGen tem os seus parâmetros preenchidos com os operandos da instrução antes de ser impresso no arquivo final. Este é o último componente do módulo RISCO. Os exemplos abaixo mostram o resultado de um processo de tradução para a linguagem de montagem do RISCO. O código 10 contêm o código original em C e o código 11 apresenta o código de montagem equivalente.

Algoritmo 10 Exemplo de compilação de código para o RISCO: código em C

```
#include "risco/cstdlib.h"

int main(void) {
  int a = read_int();
  int b = a + 1;
  print_int(b);
  return 0;
}
```

Algoritmo 11 Exemplo de compilação de código para o RISCO: código na linguagem de montagem do RISCO

```
#include "risco/ext.h"
#include "risco/stdlib.h"
        .global main
        .text
main:
        add $sp, $sp, -24
        st [$sp + 20], $ra
        sr read_int
        nop
        add $a0, $v0, 1
        sr print_int
        nop
        add $v0, $zero, 0
        ld $ra, [$sp + 20]
        nop
        add $sp, $sp, 24
        ret
        nop
```

É possível observar no código os aspectos da convenção RISCO32 discutidos anteriormente, assim como o uso de funções da biblioteca padrão que encapsulam as chamadas de sistema específicas do simulador `risco-sim` (subseção 4.2.2).

6.3 Verificação do compilador

O processo de desenvolvimento de um compilador requer técnicas e cuidados específicos devido a complexidade inerente a esta classe de softwares. Aho et al. (2007) e Appel e Ginsburg (1998) comentam que os algoritmos complexos utilizados e as relações internas entre os seus módulos fazem com que as tarefas de construção e manutenção de um compilador sejam difíceis. Como o código de entrada, em linguagem de alto nível, é transformado sucessivamente de acordo com os diferentes passes de otimização em um compilador moderno, um erro no código gerado não é facilmente rastreado para um erro no código do tradutor. Por esses motivos, o projetista de um compilador deve adaptar técnicas modernas e eficientes de testes para garantir um bom nível de confiança no produto final. Trabalhos recentes na área de testes de compiladores (BAZZICHI; SPADAFORA, 2006; YOSHIKAWA; SHIMURA; OZAWA, 2003) mostram uma tendência clara com relação à utilização de diferentes técnicas para geração automática de casos de teste.

Para a construção do módulo RISCO para o LLVM não foram utilizados testes automatizados pois estes são guiados para implementações mais maduras, onde testam funcionalidades avançadas do compilador. Nesse contexto, casos de teste mais simples, criados manualmente, seriam mais efetivos para garantir o funcionamento básico do compilador. Com relação à metodologia, tanto o backend RISCO como os scripts que o auxiliam foram desenvolvidos utilizando uma variação do processo de desenvolvimento dirigido a testes (DDT), como descrito em (BECK, 2002). Na prática, não houve a atenção necessária para o rigor dos testes de unidades que é comum no desenvolvimento baseado em DDT. O foco do processo esteve na execução de testes de regressão a cada modificação do módulo, com o seguinte esquema:

1. Um conjunto de programas de teste é criado. Inicialmente, um programa em C contendo o mínimo possível de código para ser compilado com sucesso é adicionado no conjunto.

2. O processo de compilação é executado para todos os programas do conjunto de testes.

3. Caso uma das compilações resulte em um erro interno do compilador, o erro é consertado e o processo é reiniciado do passo 2.

4. Todos os programas de teste são executados. Cada programa tem um arquivo de texto contendo uma entrada pré-definida e um outro arquivo contendo a saída esperada[1].

5. Caso um dos programas apresente um saída diferente da esperada, o compilador gera código incorreto para o exemplo. O erro é consertado e o processo é reiniciado do passo 2.

6. Caso todos os programas passem nos testes, existem duas opções: É possível realizar modificações para adicionar complexidade em um dos testes existentes ou então um novo programa de teste é criado e adicionado no conjunto.

7. O processo é repetido indefinidamente, voltando ao passo 2.

O processo utilizado ajudou a encontrar defeitos no código do compilador que resultavam na geração de código incorreto. Com o passar do tempo houve uma estabilização e todos os testes que foram adicionados a partir deste momento compilavam com sucesso, com algumas exceções. Foi criado um repositório alternativo com alguns casos de teste que ainda não são compilados corretamente para que os defeitos sejam consertados no futuro. O uso do simulador `risco-sim` foi imprescindível no processo de testes do compilador pois ele permite agilidade e flexibilidade na execução dos programas compilados com o módulo RISCO. Além disso, as chamadas de sistema definidas pelo simulador permitiram a criação de programas com características mais práticas do que programas de teste mais simples, sem interação com a plataforma hospedeira (*host*).

A utilização da biblioteca padrão pode ser observada no exemplo 12. Para facilitar a escrita dos programas teste, foi desenvolvida uma pequena interface para as chamadas de sistema do simulador que imita a biblioteca padrão do C++ com relação a utilização dos objetos `cin` e `cout` (STROUSTRUP, 1986). Além disso, algumas funções e estruturas de dados básicas foram incluídas no arquivo `risco/stl.hpp`. O conjunto de testes utilizado consiste de 43 programas em C e C++, dos quais 35 são soluções para problemas utilizados em competições de programação, cujos enunciados são disponibilizados publicamente pela Universidad de Valladolid na página (REVILLA; LIU, 2008). Os arquivos com as soluções

[1] A saída esperada para uma certa entrada de um programa poderia ser determinada manualmente. Entretanto, para obter mais confiança no processo, a saída é criada a partir da execução de uma outra versão do mesmo programa, compilada com o GCC.

Algoritmo 12 Programa teste do compilador `risco-c` em C++

```
#include "risco/iostream.hpp"
#include "risco/stl.hpp"
using namespace std;

char keyboard [47] = {
  char(96),'1','2','3','4','5','6','7','8','9','0','-','=',
  'Q','W','E','R','T','Y','U','I','O','P','[',']','\\',
  'A','S','D','F','G','H','J','K','L',';',char(39),
  'Z','X','C','V','B','N','M',',','.','/'
}, line[500];

int main(void) {
  while (cin >> line && strcmp(line, "#")) {
    int line_len = strlen(line);
    for (int i = 0; i < line_len; i++) {
      for (int j = 0; j < 47; j++) {
        if (line[i] == keyboard[j]) {
          line[i] = keyboard[j-1];
          break;
        }
      }
    }
    cout << line << endl;
  }
  return 0;
}
```

já existiam previamente e só foram adaptados para utilizar a biblioteca padrão provida pelo compilador RISCO.

Este capítulo encerra a apresentação do módulo RISCO para o LLVM. O capítulo 7 discute as análises realizadas sob o código gerado por este backend.

7 Análise de código RISCO

A atividade de desenvolvimento de software moderno para sistemas embarcados apresenta requisitos únicos a este nicho, assim como a necessidade de um ecossistema de ferramentas especializadas como, por exemplo (NOERGAARD, 2005): montadores, compiladores, simuladores, depuradores, *profilers*, analisadores estáticos e otimizadores em tempo de execução. No capítulo 5 apresentou-se os argumentos de que a maior motivação para a criação do projeto LLVM foi a necessidade de um framework de componentes reutilizáveis que suportassem o desenvolvimento de todos os tipos de ferramentas necessárias para um sistema de compilação com estas características (LATTNER; ADVE, 2004a), tendo como artefato de unificação a sua linguagem de representação intermediária, LLVM-IR. No encerramento deste trabalho, serão apresentadas as análises e ferramentas desenvolvidas para o estudo e otimização do código RISCO gerado a partir de programas escritos em linguagens de alto nível. O trabalho descrito no capítulo 6, possibilitou o uso destas linguagens para a plataforma RISCO, porém trata-se apenas do primeiro passo para o suporte completo a um conjunto de ferramentas com as características mencionadas acima. É preciso um processo de melhoramento contínuo do módulo RISCO para o LLVM, não apenas a verificação de corretude do código gerado para os casos de teste descritos na seção 6.3.

As análises realizadas sob o código RISCO consistiram de dois caminhos: estudo do tamanho do código gerado e análise estática do grafo de fluxo de controle. Ambos são detalhados neste capítulo.

7.1 Densidade do código

Lefurgy et al. (1997) trabalha com a definição de *densidade do código* como sendo uma medida indicativa, em uma dada arquitetura, do número de instruções necessárias para a implementação de uma determinada funcionalidade: quanto menor o tamanho do código, maior a sua densidade. Foi considerada a realização de um outro estudo com relação a efi-

ciência do código gerado quando comparado a outras alternativas RISC. Entretanto, como o código gerado pelo módulo RISCO executa sob o simulador `risco-sim`, a qualidade da implementação do simulador com relação aos outros simuladores para outros processadores influenciaria diretamente os resultados, e este não era o objetivo da comparação. A densidade de código é um atributo particularmente importante no desenvolvimento de software para sistemas embarcados que apresentam restrições de memória severas. Nestes casos, quaisquer ganhos na densidade do código são traduzidos em benefícios reais para a aplicação. O objetivo da análise da densidade do código RISCO gerado foi entender a eficiência do módulo `risco-llvm` com relação a utilização dos padrões de reescrita na compilação da linguagem intermediária, assim como as características de eficiência de espaço que são intrínsecas ao conjunto de instruções do processador.

Arquitetura - Otimização	**Mínimo**	**Máximo**	**Média**	**Desvio padrão**
SPARC - O1	40	1452	558.96	340.40
MIPS - O1	52	1580	561.40	341.48
SPARC - O2	40	1452	566.12	342.00
SPARC - O3	40	1452	566.96	342.60
MIPS - O2	52	1580	569.20	344.12
MIPS - O3	52	1580	569.88	344.68
RISCO - O2	52	2128	735.08	452.88
RISCO - O1	52	2120	748.84	467.52
RISCO - O3	52	2120	766.60	474.84

Tabela 4: Resultados da compilação do conjunto de testes para o RISCO, MIPS e Sparc, utilizando os 3 níveis de otimização (-O1, -O2, -O3): tamanho dos segmentos de instruções dos executáveis, em bytes

Os testes foram realizados comparando-se o número de instruções emitidas para todos os programas do conjunto de testes do `risco-c` (43 programas) quando compilados para as plataformas RISCO, MIPS (HENNESSY et al., 1981) e SparcV8 (SPARC, 1993). Note que, para as 3 arquiteturas, restringimos a geração de código para somente utilizar as instruções de tamanho fixo, todas com 4 bytes. O número de instruções não é calculado a partir do tamanho do arquivo binário executável gerado pelo compilador, pois isto iria incluir as informações de codificação do formato binário de cada plataforma, que não são relevantes. O número é calculado por uma análise a partir do seu código de montagem, com um script Python (ROSSUM, 1995). Uma característica dos compiladores modernos é que o tamanho do código gerado varia com relação ao nível das otimizações aplicadas sob o código. Desta maneira, para cada plataforma alvo considerada, o conjunto de testes é compilado 3 vezes, uma vez para cada nível de otimização disponível no LLVM. Após este

passo, continuamos a análise utilizando, para cada plataforma, a rodada de compilação que gerou a menor média para o número de instruções emitidas.

A tabela 4 explicita a influência do nível de otimização no tamanho do código. A especificação desta opção tem a forma "-On". Em geral, n é uma medida indicadora da agressividade das otimizações utilizadas pelo compilador (LLVM..., 2002), diretamente relacionada com a eficiência do código gerado e também com o seu tamanho. A única exceção é a plataforma RISCO, onde a opção "-O2" gerou código de menor tamanho médio com relação à opção "-O1". Para priorizar os menores tamanhos de código, iremos comparar as alternativas "RISCO - O2", "MIPS - O1" e "SPARC - O1".

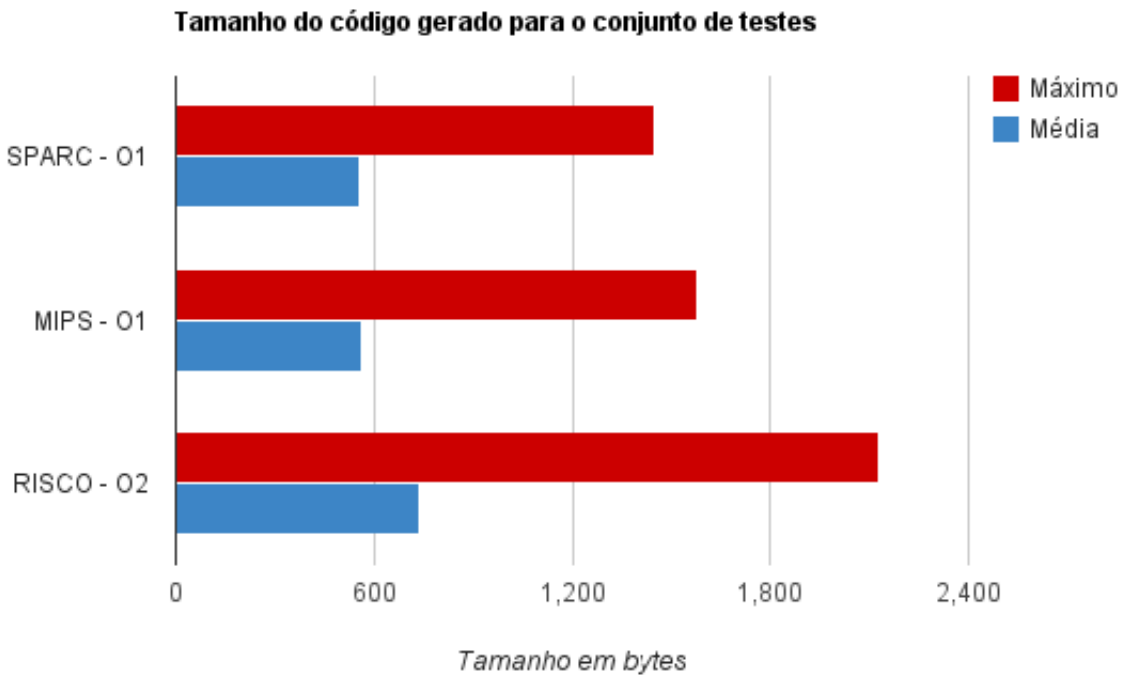


Figura 12: Número médio e máximo de instruções emitidas para o conjunto de programas nas 3 arquiteturas.

O gráfico da figura 12 mostra a comparação direta entre as 3 alternativas. É possível notar que tanto para o valor médio quanto para o máximo, a ordem da maior densidade de código até a menor é: SparcV8, MIPS e RISCO. Algumas observações pertinentes a esse resultado são:

- O fato de que ambos os backends MIPS e Sparc geram código de maior densidade que o RISCO não foi uma surpresa. São implementações mais maduras e otimizadas (LOPES, 2009c), enquanto o RISCO ainda implementa somente as funcionalidades

mais básicas de geração de código.

- A arquitetura Sparc contêm um conjunto de instruções similar ao MIPS e o RISCO. Entretanto, dispõe de um número maior de registradores (SPARC, 1993): ao total são 160 de propósito geral, divididos entre as diversas classes como os registradores normais nas janelas de registradores e os registradores específicos para valores em ponto flutuante. Isto pode ter influenciado a pequena vantagem observada com relação ao MIPS.

- O RISCO implementa menos padrões de seleção do que as outras duas arquiteturas. Isto gera um menor espaço de soluções na etapa de seleção de instruções (subseção 5.3.2).

- A falta de instruções de multiplicação e divisão faz com que uma chamada de subrotina seja feita para cada operação, o que inclui todo o processo de salvar e restaurar registradores de acordo com a convenção de chamada RISCO32 (subseção 6.1.2). Além disso, as operações de salto condicionado necessitam de 2 instruções, enquanto nas outras duas arquiteturas existem instruções específicas para comparação e salto simultâneos.

A principal conclusão desta análise é que, mesmo com as desvantagens apontadas acima, o seletor de instruções do gerador de código LLVM consegue utilizar o conjunto reduzido de padrões de seleção do módulo `risco-llvm` para gerar código diferente nos 3 níveis de otimização (vide os valores das médias na tabela 4). Isto mostra o potencial de melhora que existe caso o componente `RISCOInstrInfo` seja melhorado continuamente, como nos backends MIPS e Sparc.

7.2 Tempo de Execução no Pior Caso

Wilhelm et al. (2008) definem a análise do tempo de execução de uma dada tarefa no pior caso (WCET, do inglês *worst case execution time*) como sendo uma técnica para determinação do maior tempo possível de execução de um programa em uma plataforma específica. Esta medida pode ser feita, por exemplo, como o número de ciclos da UCP gastos. Tal análise é de grande importância para os sistemas de tempo real rígido, onde a corretude do programa depende da satisfação de todos os seus requisitos temporais (como o cumprimento dos prazos das tarefas, *deadlines*) (VAHID; GIVARGIS, 2001). Uma ferramenta capaz de estimar o pior tempo de execução de uma tarefa consegue determinar

estaticamente se um dado algoritmo terá sucesso neste ambiente ou não. Um exemplo de uma ferramenta comercial de sucesso que realiza este tipo de análise é a coleção "*aiT WCET Analyzers*" (HECKMANN et al., 2004), utilizada no desenvolvimento de sistemas aviônicos.

O problema da determinação do WCET para uma dada subrotina de um programa é indecidível (WILHELM et al., 2008) (por uma redução trivial ao problema da parada). Logo, todas as ferramentas existentes para este tipo de análise trabalham com uma classe restrita de programas que sejam passíveis de análise. As restrições são específicas a cada ferramenta e dependem da técnica que elas utilizam. Este tipo de análise é difícil pois um dado trecho de código pode apresentar inúmeros fluxos de execução distintos, cada um com o seu respectivo tempo de execução. Machado (2008) afirma que a ferramenta precisa utilizar heurísticas próprias para analisar menos caminhos distintos, de modo a terminar a análise em tempo hábil.

É importante notar que todas as ferramentas existentes não conseguem determinar o WCET exato mesmo para a classe restrita de programas que é aceita (WILHELM et al., 2008). Elas determinam um *limite superior* para este valor. Tal limite é útil para as aplicações em sistemas de tempo real, mencionadas acima. Além disso, o WCET calculado, em geral, só é válido sob a hipótese de que o programa é o único executando na plataforma hospedeira (host), em um sistema sem multitarefa.

7.2.1 Grafo de Controle de Fluxo

Neste trabalho, desenvolvemos uma ferramenta para a determinação do WCET de cada função pertencente a uma subclasse de programas em C e C++, quando estes são executados na plataforma RISCO, utilizando uma análise do grafo de controle de fluxo correspondente a cada rotina (AHO et al., 2007). Um grafo de controle de fluxo (GCF) $G(V, E)$, para uma determinada função F, é um grafo com as seguintes propriedades:

- Há um nó em V para cada bloco básico BB em F.
- Uma aresta $(\mathrm{b1}, \mathrm{b2})$ pertence a E se e somente se o bloco b2 é um dos possíveis destinos do bloco b1 em F.

É possível notar que, como a definição assume que F é dividida em blocos básicos, o programa deve estar escrito, preferencialmente, em uma linguagem próxima à linguagem de montagem, utilizando instruções de salto para manipulação de fluxo. A figura 13 mostra

o GCF equivalente a um trecho de uma sub-rotina na linguagem de montagem do RISCO. É possível notar também que o grafo é extraído diretamente da definição dos rótulos e das instruções terminadoras dos blocos básicos. Outra característica relevante é que os ciclos deste grafo indicam os *laços* de um programa (AHO et al., 2007).

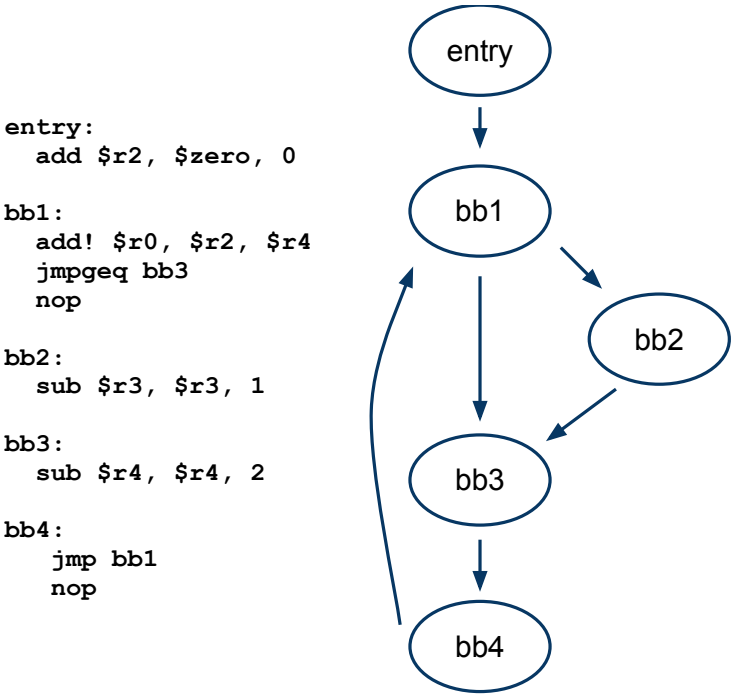


Figura 13: Exemplo de grafo de controle de fluxo

Para definir a classe de programas que é aceita pela ferramenta, precisamos introduzir os seguintes conceitos (LENGAUER; TARJAN, 1979):

Nó inicial Nó que representa o bloco básico de entrada da função. Pode ser introduzido artificialmente caso não exista de início. É o único com grau 0 de entrada.

Relação de dominância Em um grafo G, um nó u domina um nó v ($u \gg v$) se e somente se todos os caminhos do nó inicial até v passam por u.

Aresta de retorno Uma aresta $(u, v) \in G$ tal que $v \gg u$.

Árvore de dominância É a árvore Dom(G) onde: o conjunto de vértices é o mesmo de G; a raiz é o nó inicial de G; dado um nó $u \in$ Dom(G) e um nó v descendente de u, têm-se que $u \gg v$. A árvore de dominância correspondente ao GCF da figura 13 está na figura 14.

São conceitos essenciais para as análises de fluxo de dados nos compiladores modernos (LLVM..., 2002). Para este trabalho, são conceitos necessários para a definição de *laço*

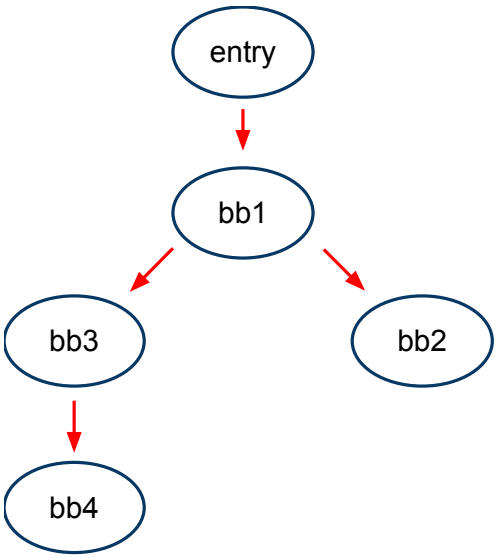


Figura 14: Exemplo de árvore de dominância

natural. Um laço natural apresenta as seguintes propriedades[1] (MUCHNICK, 1997): o seu nó inicial domina todos os nós do laço (1); tem uma única aresta de retorno nos seus nós internos, cujo destino é o nó inicial do laço e cujo nó fonte é denominado de nó final do laço (2); contêm o menor conjunto de nós que inclui o inicial e o final e não tem predecessores fora do conjunto (3). Segundo a definição, o laço apresentado no GCF da figura 13 é um laço natural.

A árvore de dominância de um GCF, neste contexto, serve para identificar facilmente as arestas de retorno e, consequentemente, os laços naturais do código. E o conceito de laço natural foi introduzido para restringir o conjunto de programas que a ferramenta de cálculo do WCET para programas RISCO admite: só são aceitos aqueles em que todos os laços são naturais. Na prática, isto restringe os programas em C e C++ que utilizam formas menos tradicionais de laços, tais como aqueles que fazem uso do comando `continue`.

7.2.2 RISCO-CFG

A ferramenta denominada `risco-cfg` faz uso de funcionalidades da plataforma LLVM para implementar a análise de WCET para programas executados na plataforma RISCO. Uma outra implementação semelhante (que também utiliza o LLVM) é descrita em (MACHADO, 2008). A ferramenta descrita aqui difere tanto na técnica utilizada para cálculo do WCET, assim como no fato de ser específica ao processador RISCO. O objetivo deste

[1] A definição apresentada por Muchnick (1997) é mais abrangente, incluindo laços com mais de uma aresta de retorno. Na ferramenta, estes laços não são suportados.

componente é determinar, para um dado módulo LLVM-IR, uma fórmula que descreva o número máximo de ciclos gastos na execução de uma função quando fornecida a sua entrada de pior caso. A fórmula não é fechada, e pode conter incógnitas correspondentes aos valores de entrada da função.

O cálculo do WCET em si é realizado combinando informações de dois grafos de controle de fluxo: um para o código em representação intermediária e outro para o código em linguagem de montagem, como será detalhado adiante usando como exemplo o programa 13.

Algoritmo 13 Rotina exemplo para cálculo do WCET

```
void test_tepc(int a, int b, int c) {
 for (int i = 5; i < a+10; ++i)
   cout << '*';

 for (int i = 0; i < b-10; ++i)
   cout << '*';

 for (int i = c; i < a+b; ++i)
   cout << '*';
}
```

A primeira etapa da execução do `risco-cfg` cria o GFC baseado no código em linguagem de montagem da função `test_tepc`, denominado *GFC-AS*. Cada bloco básico do GFC-AS contêm instruções nativas do RISCO. Internamente, a ferramenta calcula para cada bloco básico o número de instruções de cada tipo (aritmético/lógico, acesso a memória, salto ou chamada de subrotina) e armazena esta tabela na memória. O segundo passo é a geração do *GFC-IR*, o grafo de controle de fluxo equivalente ao código em LLVM-IR. O GFC-IR equivalente ao código do exemplo 13 está na figura 15. Com este grafo, é possível associar as informações dos tipos de instruções obtidas no grafo GFC-AS com os blocos básicos do GFC-IR. Além disso, o `risco-cfg` utiliza a análise `LoopPass` (LLVM..., 2002) do LLVM para identificar os laços naturais do programa e o número de iterações de cada um[2].

A terceira etapa utiliza as informações obtidas com os dois grafos anteriores para formar um conjunto de possíveis expressões para o WCET da função. O procedimento utilizado é: para todo possível caminho de execução da função, calcular o tempo estimado (TE) deste caso e adicionar no conjunto.

[2]O número de iterações nem sempre pode ser calculado facilmente e, quando retornado, pode depender dos valores dos parâmetros da função

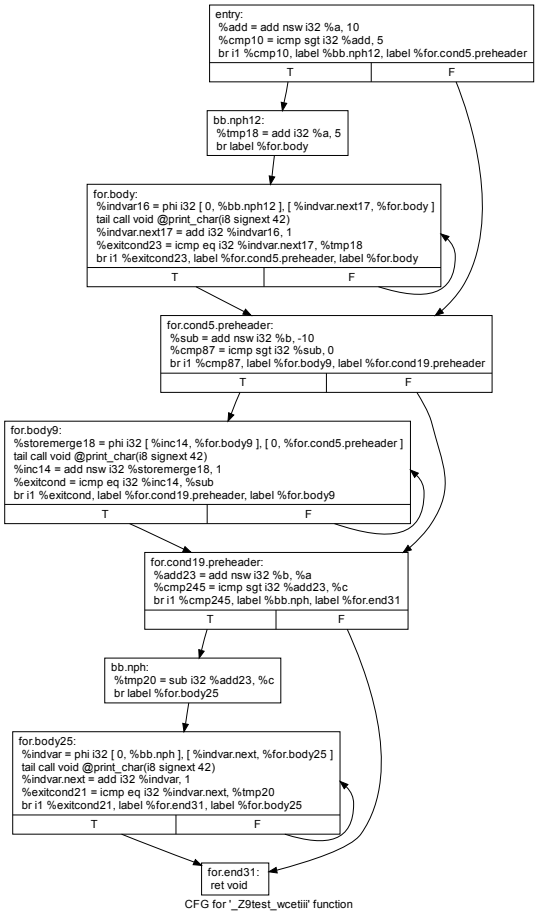


Figura 15: GFC-IR para o exemplo 13

O algoritmo utilizado é descrito no exemplo 14. A idéia do algoritmo é percorrer todos os caminhos do GFC-IR utilizando uma busca em profundidade e mantendo o tempo de execução atual do percurso (na variável `acumulador_te`) e dos predecessores do nó atual (na tabela `te_memo`). Ao encontrar o nó final de um laço, o algoritmo determina o custo total de execução do laço e não retorna para o seu nó de início, continuando com o percurso a partir de outros sucessores do nó final.

Este algoritmo apresenta complexidade exponencial no número de blocos básicos, pois trata-se de uma busca exaustiva. Para o cálculo das fórmulas, foi utilizada uma biblioteca de matemática simbólica denominada *SymPy* (SYMPY..., 2010). O uso desta biblioteca automatizou a tarefa de manipulação dos polinômios com incógnitas no cálculo do TECP propriamente dito. Aplicando a ferramenta ao exemplo 13, pode ser visto que, para os diferentes valores de `a`, `b` e `c`, existem diferentes fluxos de execução que passam por mais ou menos blocos básicos. No pior caso, a execução passa pelos três laços e a fórmula de WCET obtida foi:

Algoritmo 14 Algoritmo para cálculo do WCET

```
te_memo = { };      // mapeamento nó -> fórmula
te_set = { };       // conjunto das fórmulas já encontradas

função calcula_tecp(node, acumulador_te):
  te_memo[node] = acumulador_te
  acumulador_te += custo(node)

  se node contêm a instrução "ret":
    te_set.insert(acumulador_te)
    te_memo.remove(node)
    retorne

  se node for o nó final de um laço natural:
    inicio_loop = nó inicial do laço
    custo_interno_loop = acumulador_te - te_memo[inicio_loop]
    custo_loop = custo_interno_loop * #iterações do laço

    acumulador_te = te_memo[inicio_loop] + custo_loop

    para todo vizinho N de node que não seja inicio_loop:
      calcula_tecp(N, acumulador_te)
  senão:
    para todo vizinho N de node:
      calcula_tecp(N, acumulador_te)

  te_memo.remove(node)
  retorne
```

```
4*et_jmp + 10*et_mem + 29*et_arl +
(5 + a) * (et_call + print_char() + 2*et_jmp + 5*et_arl) +
(a + b - c) * (et_call + et_jmp + print_char() + 4*et_arl) -
(10 - b) * (et_call + print_char() + 2*et_jmp + 5*et_arl)
```

Onde:

- *et_arl, et_mem, et_jmp e et_call*: São os custos de cada tipo de instrução da plataforma RISCO.
- *print_char()*: É o custo de uma execução da rotina `print_char`.
- *a, b e c*: São os valores dos parâmetros da rotina.

Este é um exemplo construído cuidadosamente para mostrar o funcionamento ideal da ferramenta. Na prática, para a maioria dos programas do conjunto de testes, a análise `LoopPass` da plataforma LLVM não consegue determinar o número de iterações de todos os seus laços, ou então o programa contêm laços que não sejam naturais.

8 Considerações finais

Este documento relatou a criação de uma plataforma de desenvolvimento de software para sistemas embarcados que utilizam o processador RISCO. Foram apresentadas a fundamentação teórica para o entendimento do projeto, a motivação por trás do trabalho e os detalhes das decisões de projeto e de implementação que foram necessárias. É possível apostar que, com a conclusão deste trabalho, surgem novas oportunidades para a utilização do RISCO em ambiente acadêmico, podendo um dia contar com ferramentas mais robustas que possam suportar o uso do RISCO em âmbito comercial. O trabalho também poderá ser utilizado como ferramenta de ensino em arquitetura de computadores e compiladores, estimulando a utilização do LLVM neste sentido. A descrição da implementação, neste texto, junto com o código fonte disponibilizado são uma excelente fonte de aprendizado do projeto LLVM. Alguns trabalhos futuros que podem ser realizados tendo este como base são:

- Melhorias na implementação do módulo `risco-llvm`, possibilitando melhorar os resultados de densidade de código que foram obtidos no estudo descrito no capítulo 7.
- Aumento do conjunto de programas teste do RISCO, incluindo casos de uso relacionados a software para sistemas embarcados.
- Aplicação de algumas técnicas descritas em (WILHELM et al., 2008) para melhorar a eficiência da ferramenta `risco-cfg`.
- Levando em conta a crescente popularidade de linguagens de mais alto nível para o desenvolvimento de software embarcado (HIGUERA-TOLEDANO; ISSARNY, 2000; CLAUSEN et al., 2000), uma direção interessante seria estudar a possibilidade da execução do VMKit (GEOFFRAY et al., 2010) sob a plataforma RISCO, possibilitando a compilação de código Java no módulo `risco-llvm`.

Referências

ADVE, V. et al. LLVA: A Low-level Virtual Instruction Set Architecture. In: *Proceedings of the 36th annual ACM/IEEE international symposium on Microarchitecture (MICRO-36)*. San Diego, California: [s.n.], 2003.

AHO, A. V. et al. *Compilers: principles, techniques, and tools*. Second. Boston, MA, USA: Pearson/Addison Wesley, 2007. ISBN 0-321-48681-1.

APPEL, A. W.; GINSBURG, M. *Modern Compiler Implementation in C*. [S.l.]: Press Syndicate of the University of Cambridge, 1998. ISBN 052158390X.

ARAUJO, R. Computação ubíqua: Princípios, tecnologias e desafios. In: *XXI Simpósio Brasileiro de Redes de Computadores*. [S.l.: s.n.], 2003.

ARNOUT, G. SystemC standard. In: *Proceedings of the 2000 Asia and South Pacific Design Automation Conference*. New York, NY, USA: ACM, 2000. (ASP-DAC '00), p. 573–578. ISBN 0-7803-5974-7.

ASHENDEN, P. J. *The Designer's Guide to VHDL*. 1st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000. ISBN 1558602704.

AYERS, A.; YODAIKEN, B. V. Introducing real-time Linux. *Linux J.*, Specialized Systems Consultants, Inc., Seattle, WA, USA, p. 5, 1997. ISSN 1075-3583.

BACKUS, J. Report on the algorithmic language ALGOL 60. *Communications of the ACM*, v. 2, p. 299–314, 1960.

BAZZICHI, F.; SPADAFORA, I. An automatic generator for compiler testing. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 8, p. 343–353, July 2006. ISSN 0098-5589.

BECK, K. *Test Driven Development: By Example*. [S.l.]: Addison-Wesley Professional, 2002. Paperback. ISBN 0321146530.

BRANOVIC, I.; GIORGI, R.; MARTINELLI, E. WebMIPS: a new web-based MIPS simulation environment for computer architecture education. In: *WCAE '04: Proceedings of the 2004 workshop on Computer architecture education*. New York, NY, USA: ACM, 2004. p. 19.

CANTU, E. M. *Geração de código para a máquina virtual LLVM a partir de programas escritos na linguagem de programação Java*. Dissertação (Mestrado) — Universidade Federal de Santa Catarina, 2008.

CARRO, L.; SUZIM, A. A RISC architecture to explore HW/SW parallelism in HW/SW co-design. In: *Proceedings of the 96' IEEE Symposium and Workshop on Engineering of Computer-Based Systems*. [S.l.: s.n.], 1996.

CLAUSEN, L. R. et al. Java bytecode compression for low-end embedded systems. *ACM Trans. Program. Lang. Syst.*, ACM, New York, NY, USA, v. 22, p. 471–489, May 2000. ISSN 0164-0925.

DANDAMUDI, S. P. *Guide to RISC Processors: for Programmers and Engineers*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005. ISBN 0387210172.

DOUGLASS, B. P. *Real-Time UML: Developing Efficient Objects for Embedded Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN 0201325799.

FANDREY, D. Clang/LLVM Maturity Report. In: *Proceedings of the Summer 2010 Research Seminar*. [S.l.: s.n.].

FILIPPI, E. et al. Intellectual property re-use in embedded system co-design: an industrial case study. In: *ISSS '98: Proceedings of the 11th international symposium on System synthesis*. Washington, DC, USA: IEEE Computer Society, 1998. p. 37–42. ISBN 0-8186-8623-5.

FURBER, S.; EDWARDS, D.; GARSIDE, J. AMULET3: A 100 MIPS asynchronous embedded processor. *International Conference on Computer Design*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 329, 2000. ISSN 1063-6404.

GCC, the GNU Compiler Collection. 1987. Free Software Foundation. Disponível em: <`http://gcc.gnu.org/`>. Acesso em 23 de Novembro, 2010.

GEOFFRAY, N. et al. VMKit: a Substrate for Managed Runtime Environments. In: *Virtual Execution Environment Conference (VEE 2010)*. Pittsburgh, USA: ACM Press, 2010.

HECKMANN, R. et al. Worst-case execution time prediction by static program analysis. In: *In 18th International Parallel and Distributed Processing Symposium (IPDPS 2004*. [S.l.]: IEEE Computer Society, 2004. p. 26–30.

HENNESSY, J. L. et al. *MIPS: a VLSI processor architecture*. Stanford, CA, USA, 1981.

HENNESSY, J. L.; PATTERSON, D. A. *Computer Architecture: A Quantitative Approach*. 3. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003. ISBN 1558607242.

HIGUERA-TOLEDANO, M. T.; ISSARNY, V. Java embedded real-time systems: An overview of existing solutions. In: *Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*. Washington, DC, USA: IEEE Computer Society, 2000. (ISORC '00), p. 392–. ISBN 0-7695-0607-0.

HYDE, R. *The Art of Assembly Language*. 2nd. ed. San Francisco, CA, USA: No Starch Press, 2010. ISBN 1593272073, 9781593272074.

JUNQUEIRA, A.; SUZIM, A. *Microprocessador RISC CMOS de 32 bits*. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Sul, 1993.

KERNIGHAN, B. W.; RITCHIE, D. M. *The C programming Language*. 1988.

KERRISK, M. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. 1st. ed. San Francisco, CA, USA: No Starch Press, 2010. ISBN 1593272200, 9781593272203.

KOES, D. R.; GOLDSTEIN, S. C. Near-optimal instruction selection on dags. In: *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. New York, NY, USA: ACM, 2008. (CGO '08), p. 45–54. ISBN 978-1-59593-978-4.

LARUS, J. *SPIM S20: A MIPS R2000 simulator*. Madison, WI, USA, 2004.

LATTNER, C. *LLVM: An Infrastructure for Multi-Stage Optimization*. Dissertação (Mestrado) — Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002.

LATTNER, C. *LLVM and Clang: Next Generation Compiler Technology*. BSDCan 2008: The BSD Conference. Ottawa, Canada, May, 2008.

LATTNER, C.; ADVE, V. LLVM: A compilation framework for lifelong program analysis and transformation. *IEEE/ACM International Symposium on Code Generation and Optimization*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 75, 2004.

LATTNER, C.; ADVE, V. *The LLVM Compiler Framework and Infrastructure*. September 22, 2004, 2004.

LEFURGY, C. et al. Improving code density using compression techniques. In: *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 1997. (MICRO 30), p. 194–203. ISBN 0-8186-7977-8.

LENGAUER, T.; TARJAN, R. E. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, ACM, New York, NY, USA, v. 1, p. 121–141, January 1979. ISSN 0164-0925.

LEVINE, J. *Flex & Bison*. 1st. ed. [S.l.]: O'Reilly Media, Inc., 2009. ISBN 0596155972, 9780596155971.

LEVINE, J. R. *Linkers and Loaders*. 1st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999. ISBN 1558604960.

LIMA, D. *Montador RISCO*. Porto Alegre, RS, 1993.

LINDHOLM, T.; YELLIN, F. *Java Virtual Machine Specification*. 2nd. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN 0201432943.

LLVM: the LLVM Compiler Infrastructure. 2002. Disponível em: <`http://llvm.org/`>. Acesso em 23 de Novembro, 2010.

LOPES, B. C. Object code emission and llvm-mc. In: *LDM '09: LLVM developers meeting*. [S.l.: s.n.], 2009.

LOPES, B. C. The LLVM MIPS and ARM backends. In: *ELC '09: Embedded Linux Conference*. [S.l.: s.n.], 2009.

LOPES, B. C. Understanding and Writing an LLVM compiler backend. In: *ELC '09: Embedded Linux Conference*. [S.l.: s.n.], 2009.

LU, H. ELF: From the programmer's perspective. *NYNEX Science & Technology Inc*, p. 95, 1995.

MACHADO, A. *Análise de Tempo de Execução em alto nível para Sistemas de Tempo Real utilizando-se o framework LLVM*. Dissertação (Mestrado) — Universidade Federal de Santa Catarina, 2008.

MARK, D.; LAMARCHE, J. *Beginning iPhone 3 Development: Exploring the iPhone SDK*. Berkely, CA, USA: Apress, 2009. ISBN 1430224592, 9781430224594.

MUCHNICK, S. S. *Advanced compiler design and implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. ISBN 1-55860-320-4.

NOERGAARD, T. *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers*. [S.l.]: Newnes, 2005. ISBN 0750677929.

OHF: The Open Hardware Foundation. 2010. Disponível em: <`http://openhardwarefoundation.org/`>. Acesso em 23 de Novembro, 2010.

PIGUET, C. *Electronic Computer History: 1940-2000*. Centre Suisse d'Electronique et de Microtechnique SA. Maladière 71, 2000 Neuchâtel, Switzerland.

QNX Neutrino Realtime OS. 1982. QNX Software Systems. Disponível em: <`http://www.qnx.com/products/os/neutrino.html`>. Acesso em 20 Julho, 2010.

REVILLA, S. M. M.; LIU, R. Competitive Learning in Informatics: The UVA Online Judge Experience. In: *IOI '08: Proceedings of the Olympiads in Informatics*. Vilnius, Lithuania: Institute of Mathematics and Informatics, 2008. p. 131–148.

ROSSUM, G. *Python reference manual*. CWI (Centre for Mathematics and Computer Science). Amsterdam, The Netherlands, 1995.

RYZHYK, L. *The ARM Architecture*. 2006.

SALOMON, D. *Assemblers and loaders*. Upper Saddle River, NJ, USA: Ellis Horwood, 1992. ISBN 0-13-052564-2.

SEBESTA, R. W. *Concepts of Programming Languages*. 6. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN 0321193628.

SHANLEY, T. *x86 Instruction Set Architecture*. [S.l.]: Mindshare Press, 2010. ISBN 0977087859, 9780977087853.

SPARC. *The SPARC Architecture Manual Version 9*. Prentice Hall PTR, 1993. Paperback. ISBN 0130992275. Disponível em: <http://www.worldcat.org/isbn/0130992275>.

STROUSTRUP, B. The C++ programming language. Addison-Wesley, 1986. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=?doi=10.1.1.26.9545>.

SYMPY: a Python Library for Symbolic Mathematics. 2010. Disponível em: <`http://sympy.org/`>. Acesso em 23 de Novembro, 2010.

VAHID, F.; GAJSKI, D. D. Specification and design of embedded hardware-software systems. *IEEE Des. Test*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 12, n. 1, p. 53–67, 2010. ISSN 0740-7475.

VAHID, F.; GIVARGIS, T. *Embedded System Design: A Unified Hardware/Software Introduction*. New York, NY, USA: John Wiley & Sons, Inc., 2001. ISBN 0471386782.

VILELA, G. *RISCO LLVM backend*. 2010. Disponível em: <`http://code.google.com/p/risco-llvm`>. Acesso em 23 Dezembro, 2010.

VOLLMAR, K.; SANDERSON, P. MARS: an education-oriented MIPS assembly language simulator. In: *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*. New York, NY, USA: ACM, 2006. p. 239–243. ISBN 1-59593-259-3.

WEISS, A. Open source hardware: freedom you can hold? *netWorker*, ACM, New York, NY, USA, v. 12, p. 26–33, September 2008. ISSN 1091-3556.

WILHELM, R. et al. The worst-case execution-time problem: overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, ACM, New York, NY, USA, v. 7, p. 36:1–36:53, May 2008. ISSN 1539-9087.

WILLIAMS, J.; CURTIS, L. Green: The new computing coat of arms? *IT Professional*, IEEE Educational Activities Department, Piscataway, NJ, USA, v. 10, p. 12–16, January 2008. ISSN 1520-9202.

YOSHIDA, Y. et al. An object code compression approach to embedded processors. In: *ISLPED '97: Proceedings of the 1997 international symposium on Low power electronics and design*. New York, NY, USA: ACM, 1997. p. 265–268. ISBN 0-89791-903-3.

YOSHIKAWA, T.; SHIMURA, K.; OZAWA, T. Random Program Generator for Java JIT Compiler Test System. In: *Proceedings of the Third International Conference on Quality Software*. Washington, DC, USA: IEEE Computer Society, 2003. (QSIC '03), p. 20–. ISBN 0-7695-2015-4.

APÊNDICE A – Conjunto completo de instruções do RISCO

Nas seções seguintes é detalhado o conjunto de instruções do RISCO.

A.1 Instruções aritmético-lógicas

São as instruções que realizam cálculos aritméticos ou lógicos nos seus operandos, guardando o resultado em um dos registradores. O RISCO implementa apenas 25 instruções deste tipo, das 32 possíveis de acordo com o campo C4..C0 da instrução. Elas são apresentadas na tabela 5.

A.2 Instruções de acesso à memória

São as instruções que realizam leitura ou escrita em uma posição da memória. Toda instrução trabalha com um endereço de 4 bytes, indicando a posição de um byte na memória. A tabela 6 descreve estas instruções.

A.3 Instruções de salto

São instruções que implementam uma soma condicional em um registrador, não necessariamente envolvendo R31 (PC). O esquema geral das instruções de salto é:

```
jmp[cond] DST, FT1, FT2  <=>  se cond então DST <- FT1 + FT2
```

Os sufixos indicando as possíveis condições de salto são apresentados na tabela 7.

Mnemônico	**Semântica**	**Comentário**
`add`	DST ← FT1 + FT2	adição
`addc`	DST ← FT1 + FT2 + C	adição com carry
`sub`	DST ← FT1 + FT2	subtração
`subc`	DST ← FT1 - FT2 - C	subtração com carry
`subr`	DST ← FT2 - FT1	subtração reversa
`subrc`	DST ← FT2 - FT1 - C	subtração reversa com carry
`and`	DST ← FT1 and FT2	E lógico
`or`	DST ← FT1 or FT2	OU lógico
`xor`	DST ← FT1 xor FT2	OU exclusivo lógico
`rll`	DST ← FT1 rot FT2	rotação lógica à esq.
`rllc`	DST ← FT1 rot FT2 c/ C	rotação lógica à esq. c/ carry
`rla`	DST ← FT1 rot FT2	rotação aritmética à esq.
`rlac`	DST ← FT1 rot FT2 c/ C	rotação aritmética à esq. c/ carry
`rrl`	DST ← FT1 rot FT2	rotação lógica à dir.
`rrlc`	DST ← FT1 rot FT2 c/ C	rotação lógica à dir. c/ carry
`rra`	DST ← FT1 rot FT2	rotação aritmética à dir.
`rrac`	DST ← FT1 rot FT2 c/ C	rotação aritmética à dir. c/ carry
`sll`	DST ← FT1 desl FT2	deslocamento lógico à esq.
`sllc`	DST ← FT1 desl FT2 c/ C	deslocamento lógico à esq. c/ carry
`sla`	DST ← FT1 desl FT2	deslocamento aritmético à esq.
`slac`	DST ← FT1 desl FT2 c/ C	deslocamento aritmético à esq. c/ carry
`srl`	DST ← FT1 desl FT2	deslocamento lógico à dir.
`srlc`	DST ← FT1 desl FT2 c/ C	deslocamento lógico à dir. c/ carry
`sra`	DST ← FT1 desl FT2	deslocamento aritmético à dir.
`srac`	DST ← FT1 desl FT2 c/ C	deslocamento aritmético à dir. c/ carry

Tabela 5: Instruções aritmético-lógicas do RISCO

Mnemônico	**Semântica**	**Comentário**
`ld`	DST ← M[FT1 + FT2]	leitura simples
`ldpri`	FT2 ← FT2 + 1; DST ← M[FT1 + FT2]	leitura e pré-incremento
`ldpoi`	DST ← M[FT1 + FT2]; FT2 ← FT2 + 1	leitura e pós-incremento
`ldpod`	DST ← M[FT1 + FT2]; FT2 ← FT2 - 1	leitura e pós-decremento
`st`	M[FT1 + FT2] ← DST	escrita simples
`stpri`	FT2 ← FT2 + 1; M[FT1 + FT2] ← DST	escrita e pré-incremento
`stpoi`	M[FT1 + FT2] ← DST; FT2 ← FT2 + 1	escrita e pós-incremento
`stpod`	M[FT1 + FT2] ← DST; FT2 ← FT2 - 1	escrita e pós-decremento

Tabela 6: Instruções de acesso à memória do RISCO

Mnemônico	Comentário
ns	número negativo
cs	carry setado
os	overflow
zs	zero
ge	maior ou igual
gt	maior
eq	igualdade
fl	sempre falso
nn	número não negativo
nc	carry não setado
no	não houve overflow
nz	diferente de zero
lt	menor
le	menor ou igual
ne	desigualdade

Tabela 7: Sufixos de condição do RISCO

A.4 Instruções de chamada de sub-rotina

São as instruções que agrupam, sob uma condição: decremento de registrador, escrita em memória do valor do PC e salvo para outra instrução. Elas implementam eficientemente o procedimento de chamada de sub-rotina no RISCO. O esquema geral das instruções de chamada de subrotina é:

```
sr[cond] DST, FT1, FT2  <=> se cond então:
                                DST <- DST - 1
                                M[DST] <- PC
                                PC <- FT1 + FT2
```

A instrução pode ser utilizada com as condições da tabela 7, assim como as instruções de salto. Note que não há uma instrução específica para o retorno de subrotina. O retorno se dá com uma instrução "`ldpoi $pc, [$zero + $sp]`".