UNIVERSIDADE DE CAXIAS DO SUL
ÁREA DO CONHECIMENTO DE CIÊNCIAS EXATAS E ENGENHARIAS

JÚLIO CESAR SOTORIVA DE BASTIANI

UM *BACK END* PARA O DESENVOLVIMENTO DE COMPILADORES

CAXIAS DO SUL

2022

JÚLIO CESAR SOTORIVA DE BASTIANI

UM *BACK END* PARA O DESENVOLVIMENTO DE COMPILADORES

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação na Área do Conhecimento de Ciências Exatas e Engenharias da Universidade de Caxias do Sul.

Orientador: Prof. Dr. Ricardo Vargas Dorneles

CAXIAS DO SUL

2022

JÚLIO CESAR SOTORIVA DE BASTIANI

UM *BACK END* PARA O DESENVOLVIMENTO DE COMPILADORES

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel em Ciência da Computação na Área do Conhecimento de Ciências Exatas e Engenharias da Universidade de Caxias do Sul.

Aprovado em 28/06/2022

BANCA EXAMINADORA

_____

Prof. Dr. Ricardo Vargas Dorneles
Universidade de Caxias do Sul - UCS

_____

Prof. Dr. Andre Gustavo Adami
Universidade de Caxias do Sul - UCS

_____

Prof. Dr. Andre Luis Martinotto
Universidade de Caxias do Sul - UCS

Este trabalho é dedicado a todos que buscam nunca parar de aprender

“Because it’s there”

George Mallory

RESUMO

Um compilador traduz um programa para um programa equivalente escrito em outra linguagem, apresentando para o usuário quaisquer erros encontrados durante o processo de tradução, processo esse que pode ser dividido em duas grandes etapas: análise e síntese. O objetivo deste trabalho é a criação de uma ferramenta para facilitar a implementação de compiladores em ambientes acadêmicos. Neste trabalho foi definida uma representação intermediária em código de três endereços e implementado um compilador e biblioteca para transformar essa representação intermediária em *assembly* para processadores x86_64. Isto foi feito para que alunos da disciplina de compiladores possam focar exclusivamente na implementação da etapa de análise e ainda assim criar compiladores funcionais que geram aplicações executáveis e depuráveis.

Palavras-chave: Compiladores. x86_64. ELF.

ABSTRACT

A compiler is an application that translates programs into a form that can be executed by a computer, presenting to the user any errors found during the translation process, said process can be separated into two major steps: analysis and synthesis. The objective of this work is the creation of a framework to facilitate the implementation of compilers on academic environments. In this work, an intermediate representation based on three address code was defined, and a compiler (and library) to translate said intermediate representation into x86_64 assembly was implemented. This was done in order to enable students of the compilers subject to focus exclusively on the implementation of the analysis step and still create functional compilers that are able to produce executable applications.

Keywords: Compilers. x86_64. ELF.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ALGORITMOS

LISTA DE ABREVIATURAS E SIGLAS

RI	Representação Intermediária
AST	*Abstract Syntax Tree*
GCC	*GNU Compiler Collection*
ELF	*Executable and Linking Format*
ABI	*Application Binary Interface*
UCB	*UCS Compiler Back End*
SCC	*SOIS C Compiler*
GI	Grafo de Interferência
STL	*Standard Template Library*

SUMÁRIO

1 INTRODUÇÃO

Nos primeiros compiladores, o termo "compilador" se referia a um programa que compilava sub-rotinas em um objeto. Porém, com o passar do tempo, o termo evoluiu para significar um programa que realiza a tradução de um programa (normalmente escrito em uma linguagem amigável a seres humanos) em outro programa equivalente (normalmente escrito em uma linguagem voltada a facilidade de processamento por parte do hardware) (BAUER *et al.*, 1974, pp. 606). Com a evolução dos papéis de um compilador também houve uma explosão na complexidade dos compiladores, indo de simples traduções diretas (como em montadores) para o processamento e otimização de linguagens modernas com alto nível de expressabilidade.

O design de compiladores é uma das primeiras áreas da programação de sistemas para a qual uma forte base teórica foi desenvolvida e hoje é rotineiramente utilizada na prática. Na disciplina de compiladores, os alunos de Ciência da Computação estudam sobre as etapas de um compilador (análise léxica, sintática e semântica, geração de Representação Intermediária (RI), compreensão dos ambientes de execução, gerenciamento de recursos, otimizações, geração de código de máquina, entre outros), normalmente tendo um pequeno compilador como tarefa de implementação (AHO, 2008). Este compilador, ao menos no caso da disciplina de compiladores da Universidade de Caxias do Sul (UCS), acaba por excluir as etapas de otimização e geração de código de máquina. Isso acaba por tornar incompletos os compiladores criados pelos alunos, uma vez que eles realizam a análise léxica, sintática e semântica, e permitem a visualização da RI, mas não permitem a execução e depuração dos programas criados.

Existem outros exemplos de *frameworks* para a implementação de compiladores, dentre eles, os exemplos mais bem sucedidos são o LLVM (LATTNER, 2002) e *GNU Compiler Collection* (GCC) (STALLMAN, 2021). Entretanto, esses *frameworks* são focados na criação de compiladores eficientes para o uso abrangente de suas respectivas linguagens e não no ensino de compiladores, o que torna as suas interfaces complexas e dificulta o seu uso por parte dos alunos no espaço da disciplina. Também existem compiladores educacionais como *SOIS C Compiler* (SCC) (FOLEISS *et al.*, 2009) e VERTO (SCHEIDER; PASSERINO; OLIVEIRA, 2005), que tem como objetivo ensinar e servir como uma janela para o processo de compilação em si. Outro exemplo é o Compiler Explorer (GODBOLT, 2012) que tem como objetivo expor e explicar o código de máquina gerado por compiladores de linguagens populares. Nenhuma dessas ferramentas é focada na implementação de compiladores no âmbito acadêmico.

1.1 OBJETIVOS

Neste trabalho foi desenvolvido um *framework* educacional (chamado de *UCS Compiler Back End* (UCB)) para a implementação de compiladores, para que este seja utilizado nas

tarefas de implementação da cadeira de compiladores. Partindo deste objetivo geral, definimos os seguintes objetivos:

1. A definição da RI a ser utilizada pelo projeto;
2. A implementação de uma biblioteca de *back end* para a implementação de compiladores, biblioteca essa que iria incluir:
    a) Geradores de RI para a construção de programas dentro dos compiladores implementados;
    b) Rotinas para a aplicação de passes de otimização de programas definidos em RI;
    c) Ao menos um passe de otimização (como exemplo); e
    d) Rotinas para a tradução de RI para código de máquina.
3. A implementação de um compilador de RI, para a implementação e depuração de compiladores.

Para fins de simplicidade, essa biblioteca e compilador tiveram como alvo apenas o sistema operacional Linux em processadores x86_64, porém são extensíveis o suficiente para que outras plataformas alvo sejam adicionadas no futuro. Não foi desenvolvida uma máquina virtual para a RI, já que o objetivo do trabalho é o ensino da criação de aplicações nativas. A **??** ilustra a arquitetura proposta e implementada.

1.2 ESTRUTURA DO TRABALHO

O presente trabalho está organizado da seguinte forma:

- O Capítulo 2 apresenta etapas comuns na implementação de compiladores modernos, descreve os algoritmos a serem usados e descreve os objetos a serem gerados pela biblioteca de *back end* e pelo compilador implementados.
- O Capítulo 3 define a RI utilizada pela biblioteca de *back end* e pelo compilador.
- O Capítulo 4 apresenta a arquitetura da biblioteca de *back end* e a sua implementação.
- O Capítulo 5 apresenta as considerações finais do trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

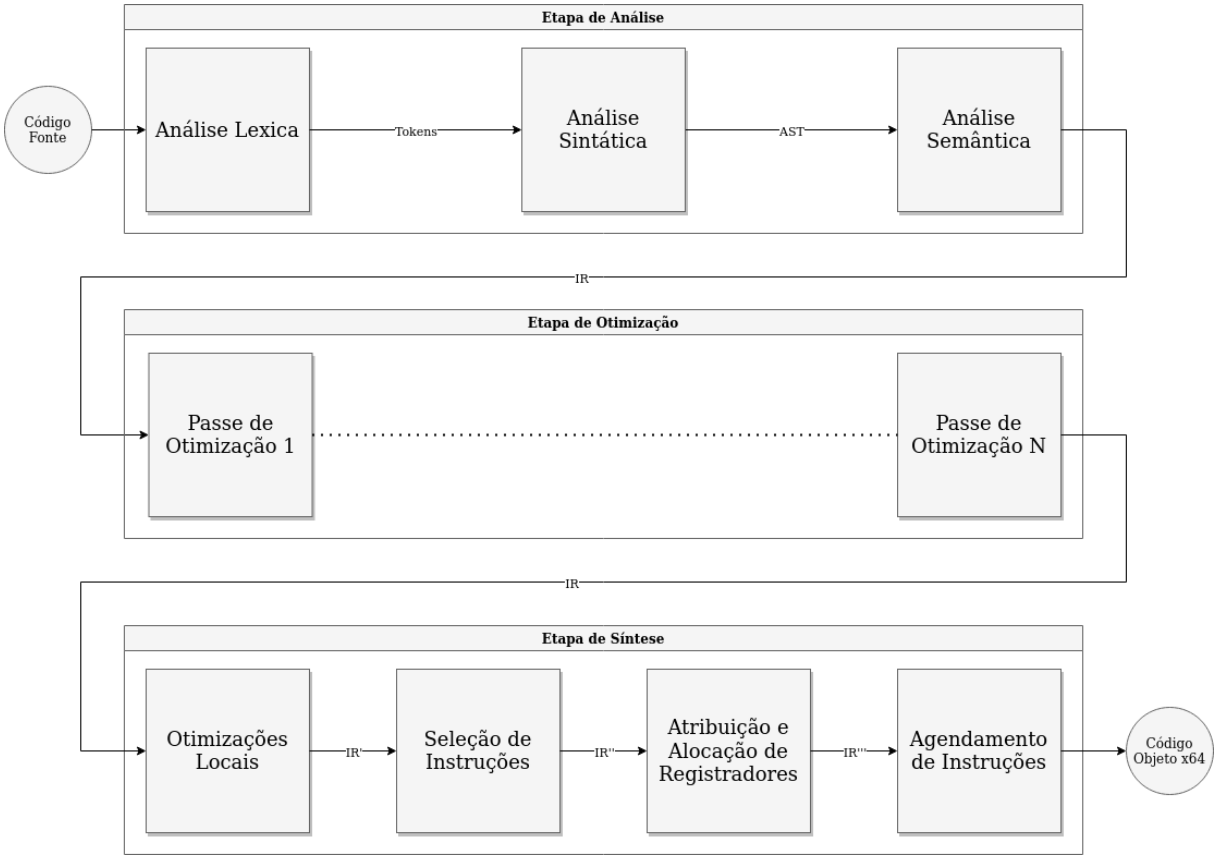
Um compilador traduz um programa para um programa equivalente escrito em outra linguagem, apresentando para o usuário quaisquer erros encontrados durante o processo de tradução. Esse processo pode ser dividido em duas grandes etapas: análise e síntese (AHO *et al.*, 2007, pp. 4).

A etapa de análise interpreta o programa inicial e verifica a sua validade sintática e semântica, caso algum erro seja encontrado o processo de compilação será interrompido e o usuário será informado dos erros no programa inicial. Uma vez que o programa é validado, este é convertido em uma RI. Esta etapa também é chamada de *front end* (AHO *et al.*, 2007, pp. 4).

A etapa de síntese cria o programa final a partir da RI criada na etapa de análise. Alguns compiladores também incluem a otimização da RI na etapa de síntese. Esta etapa também é chamada de *back end* (AHO *et al.*, 2007, pp. 4). Outros autores separam a etapa de síntese em duas etapas: otimização (otimização da RI) e *back end* (criação do programa final), criando uma arquitetura chamada de compilador de três fases (*three phase compiler*) (ROCHA, 2017, pp. 7).

As etapas de um compilador são divididas em fases, cada fase recebe como entrada uma representação do programa inicial e opera transformando esse programa, resultando em um programa equivalente podendo este ser na mesma representação recebida, ou em uma representação diferente (AHO *et al.*, 2007, pp. 4). A Figura 1 ilustra uma possível arquitetura para um compilador, o detalhamento das fases será feito a seguir.

Figura 1 – Possível arquitetura de um compilador



Fonte: Adaptado de Aho *et al.* (2007, pp. 4–5)

2.1 ETAPA DE ANÁLISE

A etapa de análise começa pela fase de análise léxica, que recebe como entrada uma série de caracteres e os transforma em *tokens*, os quais são passados para a fase seguinte. Isso é feito agrupando os caracteres recebidos em elementos textuais, sejam eles palavras, pontuação, operadores, espaços em branco e outros. Esses elementos textuais também são chamados de lexema. Neste momento, elementos textuais que não serão utilizados pela próxima fase do compilador, como espaços em branco, comentários e etc., serão eliminados e os lexemas mantidos serão classificados. A junção lexema com a sua respectiva classe semântica denomina um *token*, que é passado para a fase seguinte do compilador (AHO *et al.*, 2007, pp. 5–7). A Figura 2 ilustra a análise léxica do Algoritmo 1.

Algoritmo 1 – Programa exemplo

```
1 float res = 90 + a * 3;
```

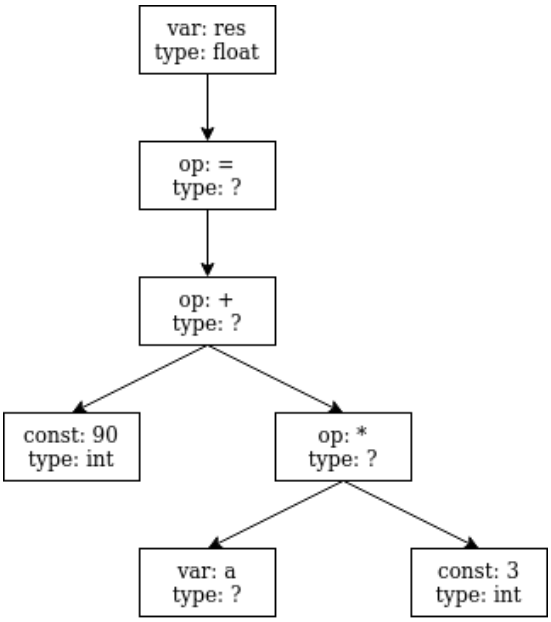
Fonte: O próprio autor

Em seguida, ocorre a fase da análise sintática. Esta transforma a sequência linear de *tokens* em uma estrutura de árvore que representa a estrutura gramatical do programa compilado. Essa representação, denominada *Abstract Syntax Tree* (AST), é uma árvore cujos nodos repre-

Figura 2 – Exemplo de tokens produzidos pela análise léxica do Algoritmo 1

Fonte: O próprio autor

Figura 3 – Exemplo de AST produzida pela análise sintática do Algoritmo 1



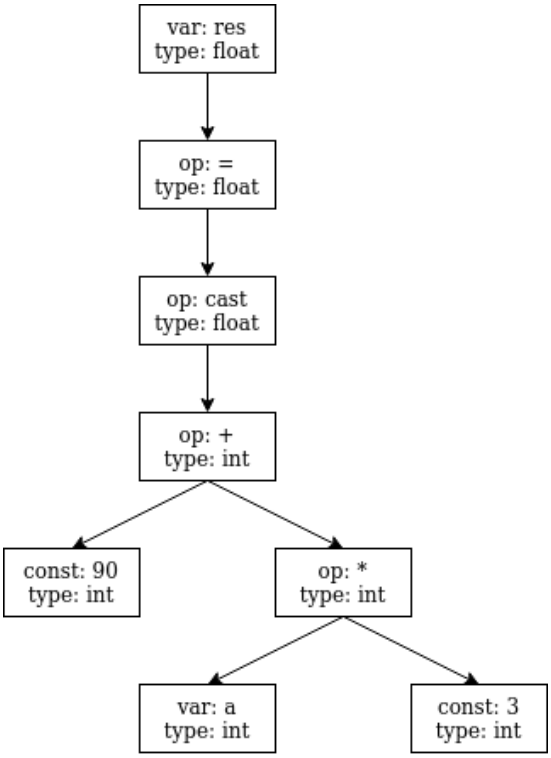
Fonte: O próprio autor

sentam estruturas da linguagem compilada (como declarações, operações, variáveis, etc) e os seus nodos subordinados representam argumentos do nodo pai. Esta estrutura é então passada para a próxima fase do compilador (AHO *et al.*, 2007, pp. 8). A Figura 3 ilustra a análise sintática do Algoritmo 1.

Em seguida, é feita a análise semântica. Esta fase se encarrega de impor as regras que não podem ser definidas na gramática, o que inclui validar se os tipos dos operandos são compatíveis com as suas respectivas operações, garantir que todas as variáveis usadas foram definidas previamente, garantir que todos os comandos *break* e *continue* estejam dentro de laços de repetição, etc. A análise semântica pode criar uma nova RI ou operar complementando a mesma AST criada na fase anterior, inferindo os tipos de quaisquer nodos que ainda não tenham um tipo definido. De qualquer forma, ao final desta fase o programa analisado estará correto (AHO *et al.*, 2007, pp. 8–9). A Figura 4 ilustra a análise semântica do Algoritmo 1.

Por fim, ocorre a fase de geração de RI. Esta fase transforma a AST, que é uma representação específica da linguagem implementada, em uma RI independente em relação à linguagem. Esta fase pode ser implementada em conjunto com a análise semântica e representa o fim da etapa de análise (AHO *et al.*, 2007, pp. 9). O Algoritmo 2 ilustra uma possível RI produzida pelo Algoritmo 1.

Figura 4 – Exemplo de AST produzida pela análise semântica do Algoritmo 1



Fonte: O próprio autor

Algoritmo 2 – Exemplo de código de três endereços produzidos pela tradução da AST do Algoritmo 1

```
1 t1 = a * 3;
2 t2 = t1 + 90;
3 res = (float) t2;
```

Fonte: O próprio autor

2.2 ETAPA DE OTIMIZAÇÃO

Etapa facultativa entre as etapas de análise e síntese, ela consiste na aplicação de um ou mais passes de otimização sobre a representação intermediária. Esses passes de otimização transformam os programas recebidos ao invés de criar programas em diferentes RIs (como em fases da etapa de análise). Isto permite que os passes de otimização sejam aplicados de forma e ordem arbitrária, de tal modo que compiladores modernos possuem diferentes configurações de otimização, que aplicam diferentes passes sobre os programas compilados (AHO *et al.*, 2007, pp. 10).

2.3 ETAPA DE SÍNTESE

A etapa de síntese irá especializar a RI, até então independente das linguagens inicial e final, para representações mais e mais similares à linguagem final. Antes da tradução da RI para a linguagem final, alguns compiladores implementam otimizações locais. Essas otimizações incluem a eliminação de sub-expressões comuns, eliminação de código morto, etc (AHO *et al.*, 2007, pp. 533).

A tradução para código objeto começa com a fase de seleção de instruções. Nesta fase, o compilador irá substituir instruções da RI por instruções da arquitetura alvo. Essa tradução nem sempre é uma tradução um para um. Arquiteturas mais complexas podem possuir instruções equivalentes a um pequeno grupo de instruções da RI. Cabe ao seletor de instruções identificar padrões e substituí-los pelas instruções da arquitetura alvo a fim de gerar o programa com o menor custo (de execução, espaço, ou qualquer outro critério usado pelo compilador) possível (AHO *et al.*, 2007, pp. 508–510).

Certas arquiteturas utilizam instruções que operam apenas sobre dois registradores (alterando um dos operandos com o resultado da operação) enquanto o código de três endereços por definição opera sobre três temporários. Pode-se resolver isso adicionando uma instrução de movimentação antes da operação que garanta que o temporário de um dos operandos seja o mesmo temporário do resultado da operação. Essa operação é ilustrada no Algoritmo 3 (LLVM COMPILER INFRASTRUCTURE, 2021).

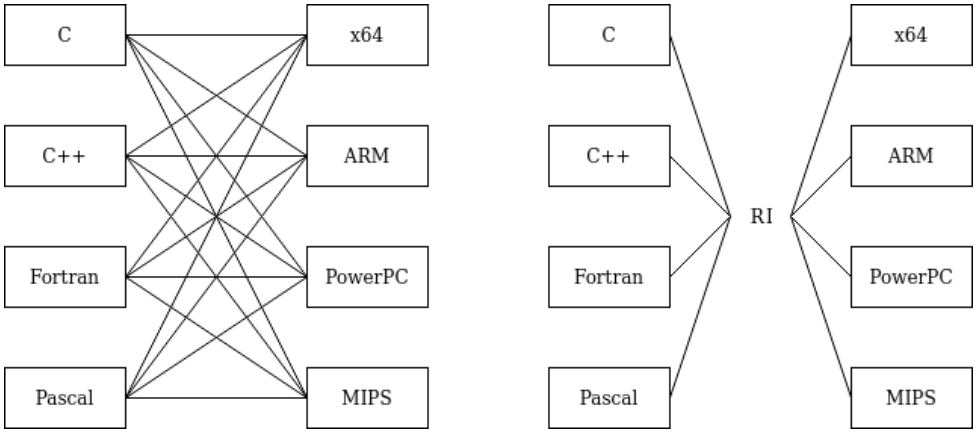
Algoritmo 3 – Instruções de dois endereços

```
1 // original
2 t3 = t1 + t2;
3
4 // adaptado
5 t3 = t1;
6 t3 = t3 + t2;
```

Fonte: O próprio autor

Em seguida ocorre a fase de alocação e atribuição de registradores. A alocação de registradores consiste em decidir quais temporários serão armazenados em registradores e quais temporários serão armazenados na pilha de execução. Idealmente, todos os temporários seriam armazenados em registradores, porém isso é impossível com um número limitado de registradores. É trabalho do alocador de registradores identificar os casos onde será necessário armazenar um temporário na pilha de execução, identificar o temporário que resultará no menor custo de acesso após ser passado para a pilha de execução e adicionar as instruções para o acesso e armazenamento desse temporário na pilha de execução, processo este chamado de *spilling*. Uma vez que o alocador de registradores tiver feito o *spilling* de uma quantidade suficiente de registradores, cabe ao atribuidor de registradores atribuir para cada temporário o seu respectivo

Figura 5 – Comparação de um compilador utilizando RI com um que não utiliza RI



Fonte: O próprio autor

registrador (AHO *et al.*, 2007, pp. 510–511). As fases de alocação e atribuição de registradores também podem ser implementadas como uma única fase.

Por último, ocorre uma fase opcional de *scheduling*. Nesta fase o compilador ordena as instruções geradas a fim de maximizar o uso do processador (AHO *et al.*, 2007, pp. 707).

2.4 REPRESENTAÇÃO INTERMEDIÁRIA

A introdução de uma representação intermediária é necessária para a generalização do compilador. Supondo que um compilador trabalhe diretamente com a linguagem da arquitetura alvo, seria necessário a re-implementação de todos os módulos do compilador para cada arquitetura alvo suportada. A introdução de uma representação intermediária permite a reutilização do *front end* entre os diferentes compiladores, com o benefício adicional de permitir que novas linguagens reutilizem os diferentes *back ends* já implementados (AHO *et al.*, 2007, pp. 357–358). A Figura 5 ilustra a modularização proporcionada por uma RI.

RIs podem ser classificadas em três classes: RIs hierárquicas, não hierárquicas e código para máquinas de pilha. RIs hierárquicas permitem o uso de estruturas encadeadas também presentes em linguagens de programação estruturada, como condicionais e laços de repetição. RIs hierárquicas possuem um nível de abstração mais elevado do que outros tipos de RI e podem ser facilmente representadas como árvores (CHOW, 2013). *Generic* e *High Gimple*, RIs utilizadas pela família de compiladores GCC, são exemplos de RIs hierárquicas utilizando árvores e código de três endereços respectivamente (STALLMAN, 2021, pp. 165, 213).

RIs não hierárquicas, também chamadas de *flat IR*, são compostas de blocos únicos de instruções a serem executadas sequencialmente, onde todo o controle de fluxo é implementado através de instruções de desvio. Representações intermediárias não hierárquicas são mais próximas a linguagens de montagem e podem ser facilmente representadas em texto através de código de três endereços (CHOW, 2013). *Low Gimple* (STALLMAN, 2021, pp. 213) e LLVM

IR, RI utilizada pela família de compiladores CLANG, são exemplos de RI não hierárquica (LATTNER, 2002).

O Algoritmo 4 apresenta um Exemplo de LLVM IR, gerada a partir da compilação do Algoritmo 1. Nela, o símbolo % denomina um temporário local e todas as instruções possuem um ou mais operandos, os quais são compostos pelo seu tipo e valor. Por exemplo, na instrução da linha 6, o valor inteiro 90 é somado ao valor contido no temporário 4 e o resultado da soma é armazenado no temporário 5. Todos os operandos (os temporários 4 e 5, e o imediato 90) são do tipo inteiro de 32 bits.

Algoritmo 4 – Exemplo de LLVM IR

```
1 %1 = alloca i32, align 4
2 %2 = alloca float, align 4
3 store i32 5, i32* %1, align 4
4 %3 = load i32, i32* %1, align 4
5 %4 = mul nsw i32 %3, 3
6 %5 = add nsw i32 90, %4
7 %6 = sitofp i32 %5 to float
8 store float %6, float* %2, align 4
9 ret i32 0
```

Fonte: O próprio autor

O código para máquinas de pilha se assemelha a RIs não hierárquicas por ter controle de fluxo explícito e ser facilmente apresentado em texto. A sua principal diferença é o uso de uma pilha de execução ao invés de registradores virtuais. Operações leem operandos do topo da pilha de execução e escrevem o seu resultado no topo da pilha (CHOW, 2013). Máquinas de pilha são amplamente utilizadas como as representações para máquinas virtuais de linguagens interpretadas, um bom exemplo disso é a máquina virtual da linguagem JAVA.

Chow (2013) define as propriedades de uma boa representação intermediária, dentre as quais podemos destacar:

- **Simplicidade**, uma boa representação deve conter o número mínimo de estruturas possível;
- **Extensibilidade**, uma boa representação intermediária deve possuir espaço para evoluir;
- **Completude**, uma representação intermediária deve poder representar corretamente todos os programas possíveis em todas as linguagens suportadas;
- **Neutralidade de hardware**, uma boa representação deve estar desacoplada das suas arquiteturas alvo, a fim de poder suportar o maior número de arquiteturas possível;
- **Diferenciação semântica**, uma boa representação não deve permitir a reconstrução do programa original a partir da representação intermediária; e

- **Ser manualmente programável**, o que facilita o desenvolvimento do compilador.

RIs podem ser apresentadas em várias formas, como árvores, grafos acíclicos dirigidos e código de três endereços. Neste trabalho, será usada uma RI não hierárquica de código de três endereços, por esta ser a forma utilizada na maior parte da literatura, e por ela ser facilmente representada em texto. O código de três endereços é uma notação para instruções onde cada instrução possui no máximo três operandos (estes também podem ser chamados de registradores virtuais). Exemplos de códigos de 3 endereços são uma operação binária seguida de uma atribuição, uma operação unária seguida de uma atribuição, etc (AHO *et al.*, 2007, pp. 91–92).

2.5 ALGORITMOS UTILIZADOS NA ETAPA DE SÍNTESE

Este trabalho consiste primariamente na implementação da etapa de síntese para a RI definida. Logo, a eficiência da implementação não é tão importante quanto a definição do espaço para a implementação dos algoritmos, para a facilidade de experimentação por parte do aluno, e a sua facilidade de compreensão. Tendo isto em mente, foram escolhidos algoritmos mais simples e menos eficientes do que os utilizados em compiladores reais, mas que realizam a mesma função. Esta seção descreve os algoritmos implementados.

2.5.1 SELEÇÃO DE INSTRUÇÕES ATRAVÉS DE PROGRAMAÇÃO DINÂMICA

O problema da seleção de instruções é classicamente descrito como um problema de reescrita de árvores (AHO *et al.*, 2007, pp. 558) (APPEL; GINSBURG, 2004, pp. 202), onde a RI é primeiramente traduzida para uma representação baseada em árvores, caso já não esteja em tal forma. Essa árvore é então reescrita de forma a substituir grupos de nodos representando instruções de RI por grupos de nodos representando instruções da arquitetura alvo, processo esse denominado *tree tiling*. Instruções são escolhidas através de heurísticas. Heurísticas comuns incluem: instruções que eliminam o número máximo de nodos de uma só vez, instruções com o menor custo de execução, instruções que ocupam o menor espaço em memória, etc. Alguns compiladores modernos realizam a seleção de instruções em grafos acíclicos dirigidos (KOES; GOLDSTEIN, 2008) que geram resultados mais próximos de resultados ótimos. Porém algoritmos de *tiling* em grafos acíclicos dirigidos não serão abordados neste trabalho devido a sua complexidade.

Um exemplo de algoritmo de *tiling* é o *maximal munch* (APPEL; GINSBURG, 2004, pp. 195). Este algoritmo realiza um único passe na árvore de instruções, operando recursivamente a partir do nodo raiz até os nodos folha, em uma abordagem *top-down*, realizando a substituição do nodo atual, ou grupo de nodos, com a instrução que substitui mais nodos de RI. Este processo

é realizado recursivamente para cada sub-árvore restante até que todas as instruções da RI sejam substituídas por instruções da arquitetura alvo.

O algoritmo de programação dinâmica (APPEL; GINSBURG, 2004, pp. 197) realiza dois passes na árvore de instruções. O primeiro passe opera dos nodos folha até o nodo raiz, em uma abordagem *bottom-up*, calculando o custo de realizar uma substituição a partir de cada nodo. Para isso, o nodo é comparado com todos os padrões e, caso seja possível substituir o nodo pelo padrão, o custo é calculado. O custo de substituição de um nodo é calculado como o custo do padrão somado ao custo das sub-árvores não englobadas pelo padrão, o padrão de menor custo será escolhido.

Após o cálculo do custo de substituição de todos os nodos, um segundo passe, também *bottom-up*, é realizado para selecionar as instruções. Para cada nodo, será primeiro realizada a seleção de todas as sub-árvores não englobadas pelo padrão escolhido, depois, os nodos englobados pelo padrão serão substituídos pelas instruções da arquitetura alvo descritas no padrão. Após esse passe, todas as instruções da RI terão sido substituídas por instruções da arquitetura alvo.

Tendo como exemplo o Algoritmo 5 e o conjunto de padrões descrito no Quadro 1 a seleção de instruções através de programação dinâmica aconteceria da seguinte maneira:

- A RI seria transformada em uma árvore de instruções, resultando na estrutura representada na Figura 6.
- O custo de cada nodo seria calculado individualmente das folhas para a raiz:
  - Ambos os nodos *load* só podem ser igualados ao padrão **a**, com custo 1.
  - O nodo *add* pode ser igualado aos padrões **b**, **c** e **d**. Caso ele seja igualado ao padrão **b**, o seu custo será o custo do padrão (1) somado ao custo dos dois operandos (1 em ambos os casos), resultando em 3. Caso ele seja igualado ao padrão **c** ou **d**, o seu custo seria apenas o custo do padrão somado ao custo de um dos dois operandos, resultando 2. Logo, um dos padrões **c** ou **d** seria escolhido, no caso deste exemplo é assumido que o padrão **d** tenha sido escolhido.
  - O nodo *ret* só pode ser igualado ao padrão **e**, com custo total de 3 (1 somado ao custo do seu operando).

  O resultado do cálculo de custos pode ser observado na Figura 7.
- Por fim, é realizada a seleção das instruções, resultando no Algoritmo 6

Algoritmo 5 – Programa exemplo para a seleção de instruções

```
1 %a = load %addr_a
2 %b = load %addr_b
```

```
3 %c = add %a, %b
4 ret %c
```

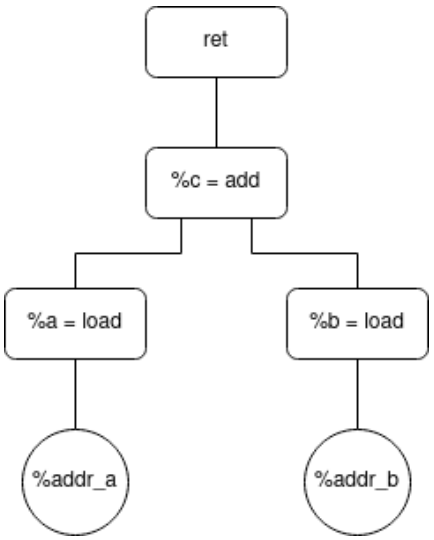
Fonte: O próprio autor

Quadro 1 – Padrões para o exemplo de seleção de instruções

	Padrão	**Instruções da arquitetura alvo**	**custo**
a	%var = load %addr	mov %var [%addr]	1
b	%res = add %lhs, %rhs	mov %res, %lhs add %res, %rhs	1
c	%lhs = load %addr %res = add %lhs, %rhs	mov %res, %rhs add %res, [%lhs]	1
d	%lrh = load %addr %res = add %lhs, %rhs	mov %res, %lhs add %res, [%rhs]	1
e	ret %res	ret %res	1

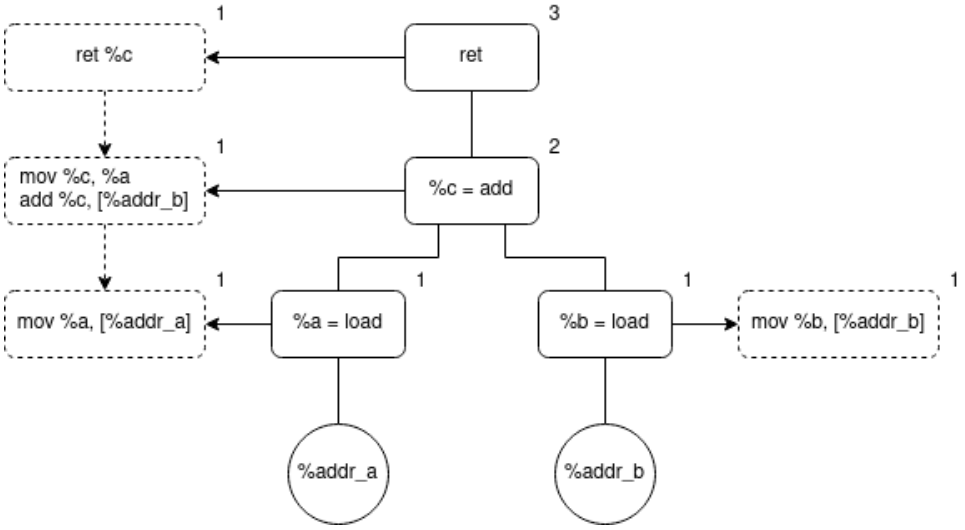
Fonte: O próprio autor

Figura 6 – Árvore gerada pelo algoritmo de programação dinâmica



Fonte: O próprio autor

Figura 7 – Árvore após o primeiro passe do algoritmo



Fonte: O próprio autor

Algoritmo 6 – Programa após a seleção de instruções

```
1 mov %a, [%addr_a]
2 mov %c, %a
3 add %c, [%addr_b]
4 ret %c
```

Fonte: O próprio autor

Neste trabalho será utilizado o algoritmo de programação dinâmica com a substituição do maior número possível de instruções como heurística. A RI será transformada em um grafo acíclico dirigido, que por sua vez será transformado em uma sequência de árvores. O algoritmo de programação dinâmica será então aplicado para cada uma dessas árvores, resultando em uma

lista de instruções.

2.5.2 ALOCAÇÃO DE REGISTRADORES POR COLORAÇÃO DE GRAFOS

A alocação de registradores pode ser abordada como um problema de coloração de grafos (AHO *et al.*, 2007, pp. 556) (APPEL; GINSBURG, 2004, pp. 235), onde as cores representam os registradores físicos e deverão ser atribuídas aos nodos de um GI. Em um GI, os nodos representam os registradores virtuais, ou valores que o compilador deseja manter em registradores, e as arestas representam uma sobreposição no tempo de vida dos registradores virtuais, indicando que os dois valores não podem ser mantidos no mesmo registrador. Neste trabalho será implementado o algoritmo de alocação de registradores apresentado em (APPEL; GINSBURG, 2004, pp. 239-242), esse algoritmo pode ser visualizado através do fluxograma na Figura 8 e consiste nas seguintes etapas:

- *Build* consiste na construção do GI dos registradores virtuais.
- *Simplify* transfere os nodos de baixo grau (nodos possuindo K vizinhos ou menos), que não podem ser coalescidos, do GI para uma pilha, repetindo até que não restem mais nodos possíveis de serem transferidos no grafo.
- *Coalesce* consiste na remoção de cópias redundantes [1]. A cópia só irá ser removida se o nodo resultante da fusão dos dois nodos fundidos tiver menos do que K vizinhos de grau significativo [2]. A remoção de uma cópia redundante implica na substituição dos dois nodos existentes no grafo de interferência por um único nodo contendo como arestas a união das arestas dos dois nodos removidos, e na remoção da instrução de movimentação. Caso um ou mais nodos tenham sido coalescidos, o algoritmo volta para a etapa *simplify*, caso contrário, o algoritmo prossegue para a etapa *freeze*.
- *Freeze* implica na desistência de coalescer dois nodos. Se ainda existirem cópias redundantes que não foram removidas por *coalesce*, a etapa de *freeze* irá escolher um par de nodos para desistir de coalescer, permitindo mais simplificações. Se um nodo tiver sido congelado, o algoritmo retorna para a etapa *simplify*, caso contrário, o algoritmo prossegue para a etapa *potential spill*.
- *Potential Spill* identifica nodos que possivelmente precisarão ser transferidos para a pilha de execução do procedimento compilado. Nesta etapa, restam apenas nodos de grau significativo no grafo de interferência. *Potential spill* irá então escolher o nodo com menor

[1] uma cópia redundante acontece quando um temporário é criado através de uma instrução de movimentação e não existe uma aresta entre os dois temporários (copiado e cópia) no grafo de interferência, isso indica que o temporário original não é utilizado após ser copiado

[2] Um nodo de grau significativo é um nodo com K ou mais vizinhos

custo de *spill* para passar para a pilha. Caso ainda existam nodos no grafo, o algoritmo voltará para a etapa *simplify*, caso contrário, o algoritmo prossegue para a etapa *select*.

- *Select* tenta colorir o grafo desempilhando todos os nodos da pilha, cada nodo é comparado com os seus vizinhos previamente desempilhados e colorido com uma cor ainda não presente entre eles. Caso não haja uma cor disponível, esse nodo é marcado como um nodo *spill* e mantido sem uma cor. Ao final da etapa *select* caso haja nodos *spill* o algoritmo passa para a etapa de *spill*, caso contrário, o algoritmo foi bem sucedido e todos os nodos do grafo estão coloridos.

- *Spill* transfere nodos que não puderam ser coloridos de registradores virtuais para a pilha de execução do procedimento compilado. Isso é feito desconsiderando as alterações no código feitas por *coalesce* e adicionando as instruções para o acesso em memória dos nodos marcados como *spill*. O algoritmo então começa novamente da etapa *build*, desconsiderando o grafo previamente construído.

Tendo como exemplo o Algoritmo 5 e o colorindo com duas cores (R1 e R2), a alocação de registradores aconteceria da seguinte maneira:

- O algoritmo começa na etapa de *build*, onde montará o GI representado na Figura 9a. O GI inicial pode ser simplificado, pois o nodo **%b** possui K vizinhos ou menos e não é potencialmente coalescível, logo o algoritmo seguirá para a etapa *simplify*.

- Na etapa *simplify*, o algoritmo irá passar o nodo **%b** para uma pilha auxiliar. O grafo resultante, representado na figura Figura 9b, não pode ser simplificado, pois os dois nodos restantes são potencialmente coalescíveis. O algoritmo então segue para a etapa *coalesce*.

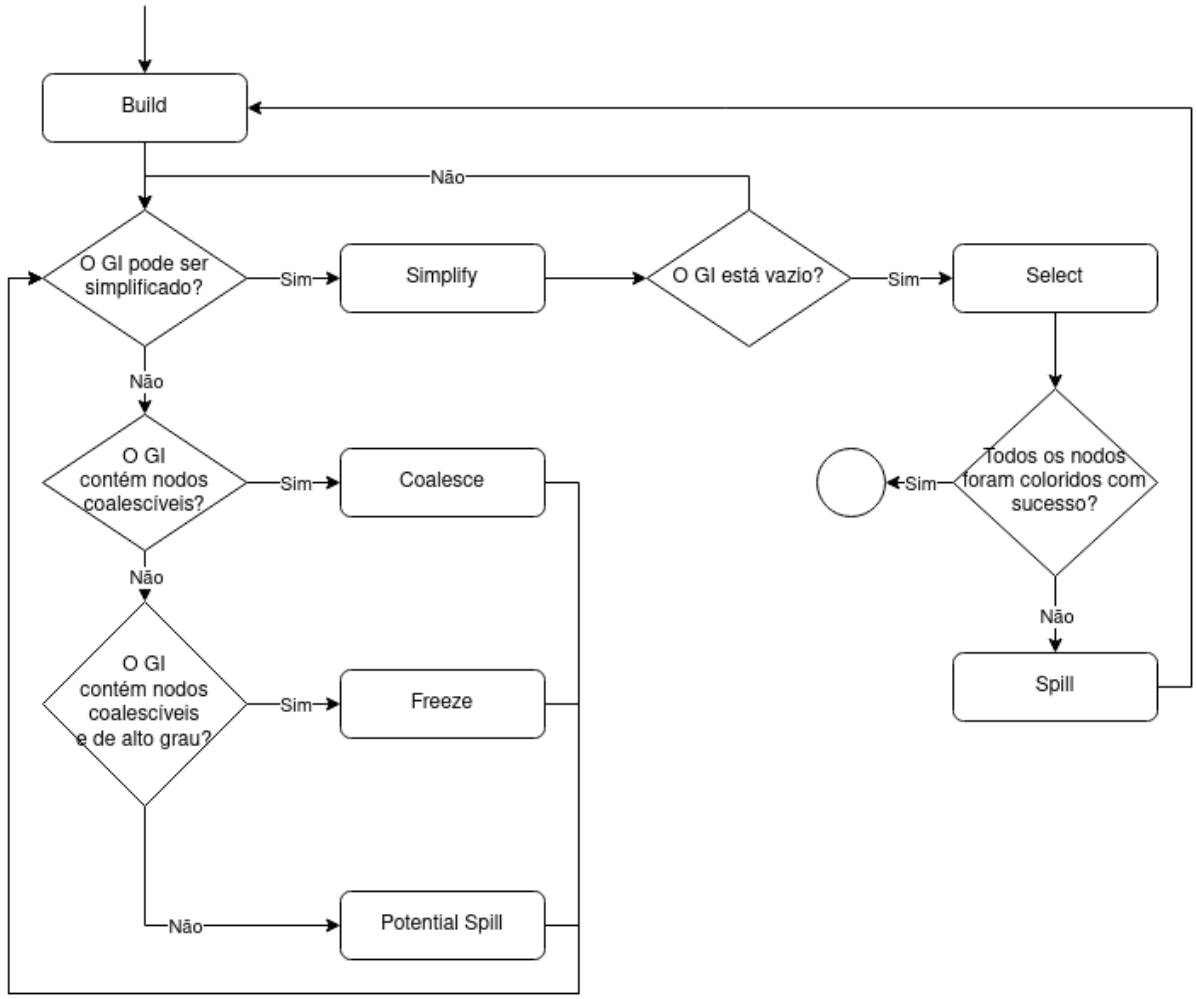
- Na etapa *coalesce*, é verificado que os nodos **%a** e **%c** são coalescíveis. A instrução na linha 2 do Algoritmo 7 é removida e os nodos **%a** e **%c** são unidos em um único nodo, resultando no grafo representado na Figura 9c. O grafo resultante pode ser simplificado, logo o algoritmo segue para a etapa *simplify*.

- Na etapa *simplify*, o algoritmo irá passar o nodo **%a/%c** para a pilha auxiliar. O grafo resultante está vazio, o que faz o algoritmo seguir para a etapa *select*.

- Na etapa *select*, o algoritmo irá desempilhar o nodo **%a/%c**. Como esse nodo ainda não possui nenhuma interferência, ele pode ser colorido com qualquer cor, este exemplo assume que ele seja colorido com R1. Em seguida, o nodo **%b** será desempilhado. Esse nodo possui uma interferência com o nodo **%a/%c**, logo ele deve ser colorido com R2.

- Todos os nodos foram desempilhados e coloridos com sucesso, o que significa que o algoritmo foi bem sucedido e os registradores virtuais podem ser substituídos pelos registradores físicos selecionados. A substituição dos registradores resultará no Algoritmo 8.

Figura 8 – Fluxograma do algoritmo de alocação de registradores



Fonte: O próprio autor

Algoritmo 7 – Programa exemplo para a alocação de registradores por coloração de grafos

```
1 live-in: %a, %b
2 mov %c, %a
3 add %c, %b
4 ret %c
```

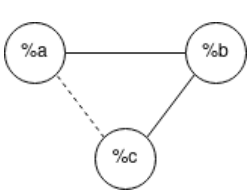
Fonte: Adaptado de (APPEL; GINSBURG, 2004)

Algoritmo 8 – Programa exemplo depois da alocação de registradores

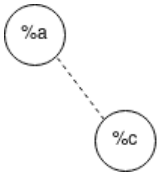
```
1 live-in: r1, r2
2 add r1, r2
3 ret r1
```

Fonte: Adaptado de (APPEL; GINSBURG, 2004)

Figura 9 – GI do Algoritmo 7



(a) GI inicial



(b) GI após o primeiro *simplify*

(c) GI após o primeiro *coalesce*

Fonte: O próprio autor

2.6 REPRESENTAÇÃO DE CÓDIGO OBJETO EXECUTÁVEL

Um compilador pode gerar vários tipos de artefatos finais. Linguagens baseadas em máquinas virtuais terminam o processo de compilação na criação da RI, que será futuramente executada pela máquina virtual da linguagem (CHOW, 2013). Aplicações nativas são executadas diretamente pelo sistema operacional e são compiladas para o código de máquina da arquitetura onde irão ser executadas. Essas aplicações nativas são denominadas arquivos objeto (TIS Committee, 1995, pp. 15). Em plataformas Linux, existem três tipos de arquivos objeto: **Arquivos executáveis**, são arquivos que podem ser executados pelo sistema operacional, ***Shared objects*** são arquivos que contêm código comum a vários executáveis e são ligados a eles dinamicamente durante a execução, **Arquivos relocáveis** são arquivos que serão combinados estaticamente para criar arquivos executáveis ou *shared objects*.

Back ends de compiladores podem operar de duas maneiras. A primeira abordagem é montando um arquivo texto contendo o código de máquina. Esse arquivo texto é então passado para outro processo, denominado montador, que irá montar o arquivo objeto ou executável. Essa abordagem tem como vantagens o aproveitamento de montadores existentes e facilitar a depuração do *back end*. A segunda abordagem é o próprio compilador montar diretamente o arquivo objeto ou executável. Essa abordagem possui como vantagens a minimização de dependências externas e maior controle sobre o artefato final. Neste projeto, será usada a primeira abordagem, a fim de reduzir a complexidade do projeto.

A fim de reduzir o escopo deste trabalho, será implementado apenas um *back end*. Esse *back end* será para sistemas Linux em arquiteturas x86_64. Sistemas Linux foram escolhidos pela sua facilidade de programação e ampla documentação. x86_64 foi escolhido pela sua grande utilização de mercado e ampla documentação.

2.6.1 FORMATO ELF

Em plataformas Linux, arquivos objeto precisam estar no formato *Executable and Linking Format* (ELF) (TIS Committee, 1995). Arquivos ELF são compostos de seções, segmentos, ou uma combinação de ambos. Segmentos contêm a informação necessária para a execução do programa, enquanto seções contêm a informação necessária para a etapa de *linking* e outros me-

tadados, como informação para a depuração do programa. Seções podem estar inclusas dentro de segmentos. Arquivos ELF esperam algumas seções pré definidas (TIS Committee, 1995, pp. 29-30), dentre as quais pode-se destacar:

- ***.text*** contém o código fonte a ser executado pela aplicação;
- ***.data*** contém as variáveis globais pré inicializadas usadas pela aplicação;
- ***.rodata*** contém as constantes globais usadas pela aplicação;
- ***.bss*** contém as variáveis globais não inicializadas, ou inicializadas com zero, usadas pela aplicação; e
- ***.debug*** seções com o prefixo debug contém informação a ser utilizada por depuradores;

Não será discutida a organização interna dos cabeçalhos de um arquivo ELF, por esta ser tratada pelo montador. Porém é preciso destacar as seções padrão, pois é necessário que estas sejam definidas corretamente no arquivo passado para o montador.

2.6.2 ARQUITETURA E ABI X86_64

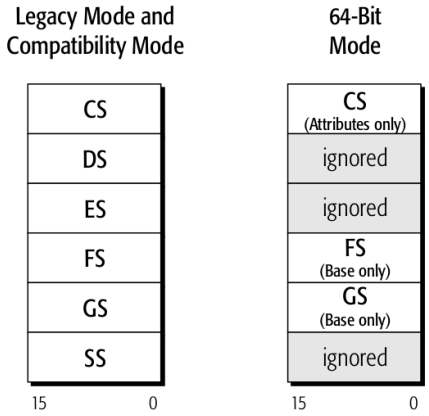
x86_64 é a arquitetura utilizada por processadores AMD e Intel de 64 bits[3]. Instruções específicas serão discutidas no Capítulo 3.

A arquitetura x86_64 possui 16 registradores de propósito geral de 64 bits cada: RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP e registradores numerados de R8 a R15. Esses registradores podem ser acessados de outras formas para realizar operações com operandos menores do que 64 bits (AMD64 Technology, 2020). Destes registradores de propósito geral, dois ainda mantêm funções específicas: RSP (*stack pointer*) aponta para o topo da pilha de execução e RBP (*base pointer*) aponta para o segmento da pilha referente à função em execução.

x86_64 também conta com 8 registradores para operações com variáveis de ponto flutuante de 80 bits, numerados de FPR0 a FPR7 (AMD64 Technology, 2020), os quais também são usados na FPU que persiste na arquitetura por questões de legado. Instruções com operandos de ponto flutuante de 64 bits devem usar os registradores MMX0 a MMX7, que fisicamente são mapeados para os registradores de ponto flutuante. Operandos de 128 bits e operandos de instruções SSE devem usar os registradores XMM0 a XMM15 para 128 bits e YMM para instruções SSE. x86_64 também conta com 6 registradores de segmento que armazenam ponteiros para diferentes segmentos do programa (dos quais apenas 3 registradores estão disponíveis para aplicações fora do modo de compatibilidade mostrado na Figura 10. EFLAGS permite inspecionar o estado de execução do programa e é usado de forma implícita em algumas instruções.

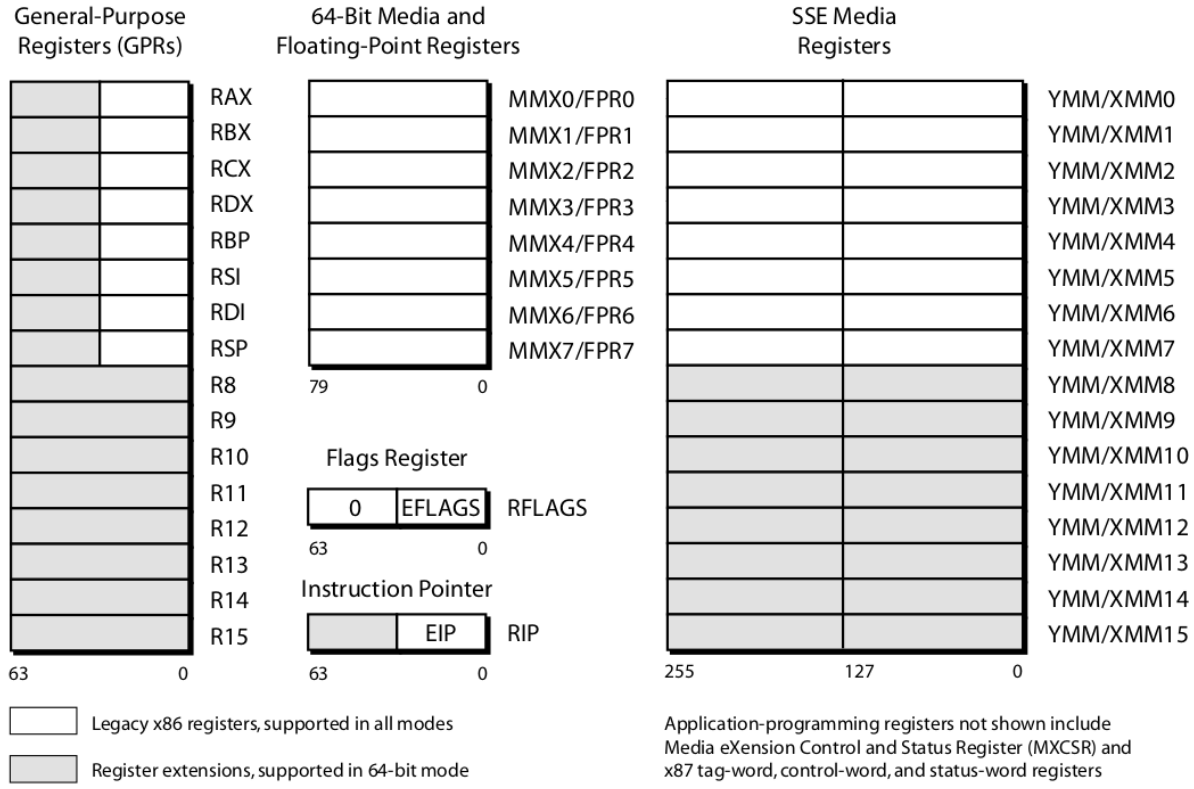
[3] x86_64 foi renomeado para AMD64 pela AMD, mas uma gama de documentos (incluindo vários documentos da Intel) ainda usam o mesmo nome, os dois nomes estão corretos e se referem à mesma arquitetura. Neste trabalho usaremos x86_64

Figura 10 – Registradores de Segmento Disponíveis em Arquiteturas x86_64



Fonte: (AMD64 Technology, 2020)

Figura 11 – Registradores Disponíveis em Arquiteturas x86_64



Fonte: (AMD64 Technology, 2020)

Por fim, RIP contém o endereço da próxima instrução a ser executada e será incrementado após cada instrução. A Figura 11 sumariza os registradores disponíveis.

Instruções x86_64 podem possuir um ou dois operandos. O número e tipo de operandos aceitos depende da instrução (AMD64 Technology, 2020). Os tipos de operandos possíveis são:

- **Registrador**: registradores possíveis dependem da instrução (registradores de propósito

geral para instruções de propósito geral, registradores MMX para instruções de ponto flutuante de 64 bits, etc);

- **Memória**: uma área de memória que pode ser acessada através de diferentes modos de endereçamento; e
- **Imediato**: um operando constante.

Gerenciamento da pilha de execução acontece através dos registradores RSP e RBP. Ao início de uma função, o conteúdo do registrador RBP é salvo no topo da pilha de execução. O conteúdo do registrador RSP é armazenado em RBP e RSP é ajustado para alocar o segmento da pilha referente à função em execução. Variáveis locais e parâmetros são então acessados através do registrador RBP. Ao final da função, o processo contrário é feito, resgatando o estado da pilha antes da chamada da função. O gerenciamento dos endereços de retorno na pilha é feito automaticamente através das instruções de chamada de função e de retorno. A manutenção dos registradores de propósito geral entre funções, a passagem de parâmetros e o retorno de resultados é organizado através de convenções (MATZ *et al.*, 2012).

Neste trabalho foi implementada a *Application Binary Interface* (ABI) Linux, que consiste no uso de registradores selecionados, em conjunto com a pilha de execução, para a passagem e retorno de parâmetros. A passagem de parâmetros é feita através dos registradores RDI, RSI, RDX, RCX, R8 e R9 para valores que podem ser passados por registradores de propósito geral, e dos registradores XMM0 a XMM7 para valores de ponto flutuante. Esses registradores, juntamente com RAX, R10, R11, MMX0 a 7 e XMM8 a 15, são registradores *caller-saved* [4]. Os registradores RBX, RBP e R12 a R15 são *callee-saved*. Caso a função chamada tenha mais parâmetros do que registradores disponíveis, ou os seus parâmetros sejam de um tipo que não possa ser passado via registrador, os valores são passados pela pilha de execução (MATZ *et al.*, 2012, pp. 17-23).

O retorno de valores acontece através de RAX e RDX para valores que podem ser passados por registradores de propósito geral e XMM0 e XMM1 para valores de ponto flutuante. Caso o valor não possa ser retornado através dos registradores descritos, ele deverá ser retornado através da memória. Nesse caso, é passado como primeiro parâmetro por RDI o endereço em memória do objeto a ser retornado, a função deverá escrever o seu resultado nesse endereço e retornar em RAX o mesmo endereço passado por RDI (MATZ *et al.*, 2012, pp. 17-23).

[4] *caller-saved* se refere a registradores que todo procedimento tem como responsabilidade salvar na pilha caso seja necessário acessar o valor deste registrador após uma chamada de procedimento. Enquanto *callee-saved* se refere a registradores que devem ser salvos na pilha de execução sempre que forem usados em um procedimento

3 REPRESENTAÇÃO INTERMEDIÁRIA

Neste capítulo, é apresentada a RI usada pelo projeto. Foi escolhida uma RI não hierárquica baseada em código de três endereços, pela sua proximidade ao código de máquina e fácil representação em texto. A linguagem final é próxima a um subconjunto da linguagem utilizada pelo LLVM. Enquanto a LLVM IR evoluiu bastante desde a sua proposta original, o conjunto de instruções da proposta original é o suficiente para os objetivos deste trabalho.

A RI proposta é, como a LLVM IR (LATTNER, 2002), fortemente tipada e fornece um número infinito de registradores virtuais. O objetivo de uma representação fortemente tipada no contexto deste trabalho é minimizar os erros de programação por parte do usuário e garantir que o programa criado pelo *front end* do aluno esteja correto antes de prosseguir com o processo de compilação. A RI proposta fornece como tipos básicos:

- **void**: um tipo sem tamanho, para ser usado como o tipo de funções que não retornam nenhum valor;
- **bool**: um tipo com tamanho de um bit, para ser usado em operações lógicas e operações de controle de fluxo;
- **i<tamanho>**: um tipo inteiro com sinal cujo o tamanho é especificado no próprio tipo;
- **u<tamanho>**: um tipo inteiro sem sinal cujo o tamanho é especificado no próprio tipo; e
- **f<tamanho>**: um tipo ponto flutuante com sinal cujo o tamanho é especificado no próprio tipo.

Os tamanhos válidos são: 8, 16, 32 e 64 bits para tipos inteiros, e 32 e 64 bits para tipos de ponto flutuante. Além de tipos básicos, a RI suporta ponteiros (definidos como ^**<tipo>**), vetores (definidos como **array <tamanho> of <tipo>** e acessados através de ponteiros) e *strings* (representadas através de vetores de inteiros). Registradores virtuais e símbolos locais são acessados e definidos através do operador **%<id>** enquanto símbolos globais são definidos e acessados através do operador **@<id>**, onde **id** é uma *string* arbitrária sem espaços. Todas as instruções AMD64 foram escolhidas através de Intel (2016)

3.1 INSTRUÇÕES

Nesta seção são definidas as instruções utilizadas pela RI apresentada e os seus equivalentes na linguagem de montagem para processadores x86_64.

3.1.1 INSTRUÇÕES ARITMÉTICAS COM DOIS OPERANDOS

A maior parte das instruções da RI se encaixa nessa categoria. São instruções que realizam operações aritméticas entre dois operandos e armazenam o resultado em um terceiro. Todos os operandos devem possuir o mesmo tipo, caso contrário é gerado um erro de compilação. Instruções aritméticas aceitam operandos dos tipos inteiro (com e sem sinal), ponto flutuante e ponteiros, sendo que ponteiros são tratados como inteiros sem sinal. O Quadro 2 apresenta as instruções de RI e as suas respectivas traduções para instruções de máquina. A linha 6 do Algoritmo 9 ilustra o uso da instrução de soma.

Quadro 2 – Instruções aritméticas com dois operandos

Operação	**Instrução de RI**	**Tipos inteiros**	**Tipos de ponto flutuante**
Soma aritmética	*add*	*ADD*	*ADDSS*, *ADDSD*
Subtração aritmética	*sub*	*SUB*	*SUBSS*, *SUBSD*
Multiplicação aritmética	*mul*	*IMUL*, *MUL*	*MULSS*, *MULSD*
Divisão aritmética	*div*	*IDIV*, *DIV*	*DIVSS*, *DIVSD*
Resto da divisão aritmética	*rem*	*IDIV*, *DIV*	N/A
Negação bit a bit	*not*	*NOT*	N/A
Conjunção bit a bit	*and*	*AND*	N/A
Disjunção bit a bit	*or*	*OR*	N/A
Disjunção exclusiva bit a bit	*xor*	*XOR*	N/A
Deslocamento bit a bit para a esquerda	*shl*	*SHL*	N/N
Deslocamento bit a bit para a direita	*shr*	*SHR*	N/N

Fonte: O próprio autor

Algoritmo 9 – Programa exemplo para a seleção de instruções

```
1  i32 @sample(i32 %a, i32 b)
2  {
3      %ref = alloc i32
4      store i32 %a ^i32 %ref
5      %c = load i32 ^i32 %ref
6      %d = add i32 %a %b
7      %e = cast i32 %a i64
8      %f = cmp le i32 %a %b
9      brc bool %f %label_1 %label_2
10
11 label_1:
12     br %label_2
13
14 label_2:
15     %g = cp i32 %d
16     %h = call i32 @foo(%g)
17     ret i32 %h
18 }
```

Fonte: O próprio autor

3.1.2 CONVERSÃO ENTRE TIPOS

A instrução *cast* realiza a conversão entre diferentes tipos. Ela recebe apenas um operando que é convertido para o tipo destino e armazenado no registrador resultado. As conversões válidas e as suas respectivas instruções de máquina são apresentadas no Quadro 3.

Quadro 3 – Tipos válidos para a instrução *cast* e suas respectivas instruções de máquina

Tipo do operando	**Tipo resultado**	**Comportamento**
Tipos inteiros	Tipos inteiros menores	Truncamento para a parte menos significativa usando instruções *MOVE* se necessário
Tipos inteiros com sinal	Tipos inteiros maiores	Serão traduzidos através da instrução *MOVSX*
Tipos inteiros sem sinal	Tipos inteiros maiores	Serão traduzidos através da instrução *MOVZX*
Tipos inteiros com sinal	f32	Serão traduzidos através da instrução *CVTSI2SS*
Tipos inteiros com sinal	f64	Serão traduzidos através da instrução *CVTSI2SD*
f32	Tipos inteiros com sinal	Serão traduzidos através da instrução *CVTSS2SI*
f64	Tipos inteiros com sinal	Serão traduzidos através da instrução *CVTSD2SI*
f32	f64	Serão traduzidos através da instrução *CVTSS2SD*
f64	f32	Serão traduzidos através da instrução *CVTSD2SS*

Fonte: O próprio autor

Conversões entre diferentes tipos de ponteiros afetam apenas o comportamento durante a de-referência do ponteiro. A conversão de qualquer tipo para *bool* não é permitida mas a conversão de *bool* para tipos inteiros é, neste caso *bool* se comporta como um inteiro sem sinal com tamanho de um bit. A linha 7 do Algoritmo 9 ilustra o uso da instrução para converter um registrador do tipo inteiro de 32 bits para um registrador do tipo inteiro de 64 bits.

3.1.3 DESVIOS CONDICIONAIS E INCONDICIONAIS

Desvios incondicionais são feitos através da instrução *br* que é traduzida diretamente para a instrução *JMP*. Desvios condicionais são feitos através da instrução *brc*, que toma como parâmetros um registrador virtual do tipo *bool* e duas *labels*, a execução do programa se direcionará para a primeira caso a condição seja verdadeira, ou para a segunda caso a condição seja falsa. A Seção 3.1.4 descreve como a instrução *brc* é traduzida caso ela seja traduzida juntamente com uma instrução *cmp*. Caso ela tenha que ser traduzida diretamente de um registrador do tipo *bool*, será adicionada uma comparação deste registrador com zero. A linha 12 do Algoritmo 9 ilustra o uso da instrução *br*, enquanto a linha 9 do mesmo algoritmo ilustra o uso da instrução *brc*.

3.1.4 COMPARAÇÃO

Comparações entre dois registradores são realizadas a partir da instrução *cmp*. Esta instrução recebe dois registradores de tipos iguais e um modo de comparação, retornando um registrador do tipo *bool*. Os modos válidos são:

- **eq** verifica se ambos os operandos são iguais;

- **ne** verifica se ambos os operandos são diferentes;
- **lt** verifica se **operando1** é menor do que **operando2**;
- **le** verifica se **operando1** é menor ou igual ao **operando2**;
- **gt** verifica se **operando1** é maior do que **operando2**; e
- **ge** verifica se **operando1** é maior ou igual ao **operando2**.

A tradução para tipos inteiros é de acordo com o Quadro 4. As traduções para f32 e f64 são feitas da mesma forma que as traduções para tipos inteiros, porém substituindo as instruções *CMP* com instruções *CMPSS* e *CMPSD* respectivamente. A linha 8 do Algoritmo 9 ilustra o uso da instrução *cmp*.

Quadro 4 – Tradução de instruções *cmp* para uso genérico do resultado e para desvios condicionais

Tipo	**Modo**	**Tradução genérica**	**Tradução em conjunto com *brc***
-	*eq*	*CMP*, *SETE*	*CMP*, *JE*, *JMP*
-	*ne*	*CMP*, *SETNE*	*CMP*, *JNE*, *JMP*
Com sinal	*lt*	*CMP*, *SETL*	*CMP*, *JL*, *JMP*
Sem sinal	*lt*	*CMP*, *SETB*	*CMP*, *JB*, *JMP*
Com sinal	*le*	*CMP*, *SETLE*	*CMP*, *JLE*, *JMP*
Sem sinal	*le*	*CMP*, *SETBE*	*CMP*, *JBE*, *JMP*
Com sinal	*gt*	*CMP*, *SETG*	*CMP*, *JG*, *JMP*
Sem sinal	*gt*	*CMP*, *SETA*	*CMP*, *JA*, *JMP*
Com sinal	*ge*	*CMP*, *SETGE*	*CMP*, *JGE*, *JMP*
Sem sinal	*ge*	*CMP*, *SETAE*	*CMP*, *JAE*, *JMP*

Fonte: O próprio autor

3.1.5 ALOCAÇÃO NA PILHA DE EXECUÇÃO

A interação com a pilha de execução e com a memória geral é realizada através das instruções de RI *load* e *store*. Ambas as instruções são traduzidas para instruções *MOVE* (ou *MOVSS* e *MOVSD* no caso de registradores f32 e f64 respectivamente) ou removidas caso a instrução que utiliza o valor carregado em memória possa acessar a memória diretamente.

A instrução *alloc* recebe um tipo, aloca espaço equivalente ao tamanho deste tipo na pilha de execução e retorna um ponteiro para o local alocado. Ela não é traduzida diretamente para nenhuma instrução, apenas afetando o tamanho da pilha de execução da função compilada, o ponteiro retornado pela instrução será substituído por um deslocamento a partir do registrador RBP. As linhas 3, 4 e 5 do Algoritmo 9 ilustram o uso das instruções *alloc*, *store* e *load* respectivamente.

3.1.6 CÓPIA DE VALORES

A instrução *cp* realiza uma cópia de um registrador e é traduzida diretamente para uma instrução *MOVE* (ou *MOVSS* e *MOVSD* no caso de registradores f32 e f64 respectivamente). A linha 15 do Algoritmo 9 ilustra o uso da instrução *cp*.

3.1.7 CHAMADA DE PROCEDIMENTOS

A chamada de procedimentos é feita através da instrução *call* que recebe o identificador do procedimento a ser chamado e um número arbitrário de parâmetros. Ela é traduzida em três partes: *stack-up*, *call* e *stack-down*:

- *stack-up* salva os registradores vivos[1] na pilha de execução e prepara a chamada de função para que esta respeite a ABI descrita na Seção 2.6.2, isto é feito através de instruções *MOVE* e *PUSH*;
- *call* é a chamada de função em si, esta é apenas uma instrução *CALL* tendo como operando o identificador proporcionado na instrução de RI; e
- *stack-down* retorna a pilha ao seu estado antes da chamada de função e move o resultado da função (se houver um) para o registrador desejado, isso é feito através de instruções *MOVE* e *POP*.

A linha 16 do Algoritmo 9 ilustra a chamada de um procedimento *foo* passando um único parâmetro do tipo inteiro de 32 bits.

3.1.8 RETORNO DE PROCEDIMENTOS

O retorno de procedimentos é realizado através da instrução *ret*, que move o valor a ser retornado (se houver um) para o registrador físico adequado e termina a execução do procedimento com uma instrução de máquina *RET*. Todo procedimento precisa de ao menos uma instrução *ret*, mesmo que não retorne nenhum valor, e todo bloco base precisa ser terminado por uma instrução de desvio (condicional ou incondicional) ou por uma instrução *ret*. Procedimentos podem ter múltiplas instruções *ret*. A linha 17 do Algoritmo 9 ilustra o uso da instrução *ret*.

[1] um registrador é considerado vivo para uma instrução caso ele tenha sido inicializado antes dessa instrução com um valor que será utilizado por outra instrução posterior

4 ARQUITETURA DO PROJETO

Neste capítulo são descritas as interfaces disponíveis para os diferentes usuários do projeto. Todo o software foi desenvolvido em C++20, com o intuito de portar a biblioteca para diferentes linguagens (como Python, Java, Rust e etc) no futuro.

UCB suporta duas maneiras distintas de desenvolver compiladores. A primeira abordagem é através de um compilador externo, permitindo que os alunos criem um arquivo texto contendo a RI. A outra abordagem é através de uma biblioteca, permitindo que os alunos criem a RI diretamente em seus compiladores.

4.1 ESTRUTURA GERAL DO PROJETO

O projeto é composto de três componentes, as bibliotecas estáticas *libucb-core* e *libucb-frontend*, e o executável *uic*. *libucb-core* contém os recursos para compilar a RI, enquanto *libucb-frontend* contém apenas o *parser* da forma textual da RI, descrita no Capítulo 3. *uic* une as duas bibliotecas estáticas em um executável.

O compilador *uic* toma como argumento o caminho para um arquivo texto contendo a representação intermediária, compila a RI e gera um arquivo contendo o código de máquina equivalente ao programa compilado. O compilador não gera código objeto isso deve ser feito através de ferramentas externas, como um montador ou até mesmo um compilador de C (como o próprio GCC ou clang).

O compilador não gera um ponto de entrada para o programa ou gera código para a interação com o sistema operacional. O que pode ser feito é a chamada das funções geradas pelo compilador através de um programa escrito na linguagem de programação C. O Algoritmo 10 ilustra uma possível implementação da função fatorial em RI, cuja compilação gera o arquivo em assembly apresentado no Algoritmo 11. O Algoritmo 12 mostra como chamar a função fatorial em um programa escrito em C, os passos para a compilação deste programa estão descritos no Algoritmo 13.

Algoritmo 10 – Função Fatorial Implementada em RI

```
1  i32  @fat(i32 %n)
2  {
3      %n.ref = alloc i32
4      store i32 %n ^i32 %n.ref
5      %n.0 = load i32 ^i32 %n.ref
6      %cnd.0 = cmp le i32 %n.0 1
7      brc bool %cnd.0 %if_then %if_else
8
9  if_then:
10     ret i32 1
11
12 if_else:
13     %n.1 = load i32 ^i32 %n.ref
14     %tmp.0 = sub i32 %n.1 1
15     %tmp.1 = call i32 @fat(%tmp.0)
16     %n.2 = load i32 ^i32 %n.ref
17     %tmp.2 = mul i32 %n.2 %tmp.1
18     ret i32 %tmp.2
19 }
```

Fonte: O próprio autor

Algoritmo 11 – Função Fatorial Compilada para Assembly

```
1         .text
2         .intel_syntax noprefix
3         .file   "../samples/factorial/fat.uir"
4         .globl  fat
5         .p2align        4, 0x90
6         .type   fat,@function
7 fat:
8 .TmpLbl0: # bblock entry
9         push    rbp
10        mov     rbp, rsp
11        sub     rsp, 4
12        mov     dword ptr [rbp - 4], edi
13        cmp     dword ptr [rbp - 4], 1
14        jle     .TmpLbl1
15        jmp     .TmpLbl2
16 .TmpLbl1: # bblock if_then
17        mov     eax, 1
18        jmp     .TmpLbl3
19 .TmpLbl2: # bblock if_else
20        mov     eax, dword ptr [rbp - 4]
21        sub     eax, 1
22        mov     edi, eax
23        call    fat
24        imul    eax, dword ptr [rbp - 4]
25        jmp     .TmpLbl3
26 .TmpLbl3: # bblock exit
27        add     rsp, 4
28        pop     rbp
29        ret
30 .TmpProcEnd0:
31        .size fat, .TmpProcEnd0-fat
```

Fonte: O próprio autor

Algoritmo 12 – Exemplo de Programa em C

```
1  #include <stdio.h>
2
3  int fat(int n);
4
5  int main()
6  {
7      int n = 5;
8      int f = fat(n);
9      printf("fat␣of␣%d␣==␣%d\n", n, f);
10     return 0;
11 }
```

Fonte: O próprio autor

Algoritmo 13 – Instruções para a compilação do Algoritmo 10

```
1 $ uic fat.uir
2 $ clang -o test_fat fat.s main.c
3 $ ./test_fat
```

Fonte: O próprio autor

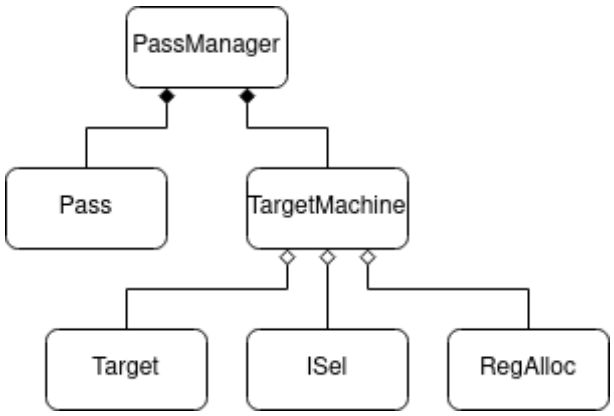
A interface da biblioteca *libucb-core* é composta das classes *CompileUnit* e *PassManager*. *CompileUnit* contém o conjunto de funções em RI a serem compiladas, a composição dessa classe será descrita mais detalhadamente na Seção 4.2. *PassManager* é a classe encarregada da compilação em si, ela é configurada através do seu construtor com um vetor de passes de otimização e uma *TargetMachine*. *PassManager* possui um único método chamado *apply* que aplica os passes de otimização configurados a uma unidade de compilação e compila a mesma para *assembly*. O Algoritmo 14 ilustra como usar o método *apply*, neste caso o programa será escrito para a saída padrão do sistema. A Figura 12 ilustra a relação da classe *PassManager* com as outras classes responsáveis pela compilação da RI.

Algoritmo 14 – Exemplo de Uso do Método *apply*

```
1 auto compile_unit = parse(...);
2 auto pm = make_pass_manager();
3 pm->apply(compile_unit, src_fname, std::cout);
```

Fonte: O próprio autor

Figura 12 – Diagrama ilustrando o relacionamento das classes responsáveis pela compilação da RI



Fonte: O próprio autor

A classe abstrata *Pass* representa um passe de otimização. *Pass* possui um único método virtual também nomeado *apply* que recebe um ponteiro a um procedimento em RI que será modificado de acordo com o passe implementado. Para implementar um passe de otimização, deve-se herdar da classe *Pass* e implementar o método *apply*, o passe então deverá ser

instanciado e um ponteiro único para ele deve ser passado para o *PassManager* que será utilizado. O processo de uso de um passe de otimização é apresentado nas linhas 3, 4 e 14 do Algoritmo 15.

A classe *TargetMachine* indica as condições sobre as quais a unidade de compilação será compilada. Ela possui apenas um método *compile* que não precisa ser utilizado pelo usuário final, ela precisa apenas ser instanciada e passada para o *PassManager*. *TargetMachine* possui apenas um construtor que toma como parâmetros ponteiros únicos para instâncias das classes abstratas *ISel*, *RegAlloc* e *Target*.

Algoritmo 15 – Exemplo da instanciação de um *PassManager*

```
1  std::unique_ptr<PassManager> make_pass_manager()
2  {
3      std::vector<std::unique_ptr<Pass>> passes;
4      passes.push_back(std::make_unique<PassA>());
5
6      auto target = std::make_shared<x64::X64Target>();
7      auto isel = std::make_unique<DynamicISel>(target);
8      auto regalloc = std::make_unique<GraphColoringRegAlloc>(target);
9
10     auto target_machine = std::make_unique<TargetMachine>(
11             std::move(isel), std::move(regalloc), target);
12
13     return std::make_unique<PassManager>(
14             std::move(passes), std::move(target_machine));
15 }
```

Fonte: O próprio autor

As classes abstratas *ISel* e *RegAlloc* representam seletores de instruções e alocadores de registradores respectivamente. Ambas as classes possuem apenas um método virtual *run_on_procedure* que recebe como parâmetro um ponteiro para um função que será alterada de acordo com o algoritmo implementado. Neste trabalho foram implementados apenas um seletor de instrução e alocador de registradores, que serão detalhados nas seções Seção 4.3.1 e Seção 4.3.2 respectivamente.

A classe abstrata *Target* representa a arquitetura alvo para a qual o programa em RI será compilado. Ela possui uma série de métodos virtuais que devem ser implementados:

- *load_pats()*: carrega os padrões que serão usados pelo seletor de instruções, o formato desses padrões será descrito na Seção 4.3.1;
- *load_reg_classes()*: carrega as classes de registradores que serão usadas pelo alocador de registradores, o formato dessas classes será descrito na Seção 4.3.2;

- *is_rr_move(MachineOpc opc)*: método de utilidade que indica se uma instrução é uma instrução de movimentação entre dois registradores;
- *dump_proc(Procedure& proc, std::ostream& out)*: método de depuração para a escrita de um procedimento na saída *out*, necessária quando o procedimento usa instruções da arquitetura alvo;
- *dump_bblock(BasicBlock& bblock, std::ostream& out)*: método de depuração para a escrita de um bloco base na saída *out*, necessária quando o bloco base usa instruções da arquitetura alvo;
- *abi_lower(Procedure& proc)*: método responsável por aplicar a ABI na entrada e saída(s) do procedimento *proc*. Este método é chamado para todo procedimento entre a seleção de instruções e alocação de registradores;
- *stack_lower(Procedure& proc)*: método responsável por adicionar o código necessário para a manutenção da pilha na entrada e saída(s) do procedimento *proc*, ele também aplica a ABI em quaisquer chamadas de função que ocorram no procedimento. Este procedimento é chamado depois da alocação de registradores, quando o tamanho necessário para a pilha já está definido; e
- *print_asm(CompileUnit& unit, std::ostream& out, const std::string& src_filename)*: escreve o *assembly* da unidade de compilação já compilada para a saída *out*. Este método é chamado uma única vez após todos os procedimentos serem compilados.

Neste trabalho foi implementada apenas uma arquitetura alvo, que será detalhada na Seção 4.3.3.

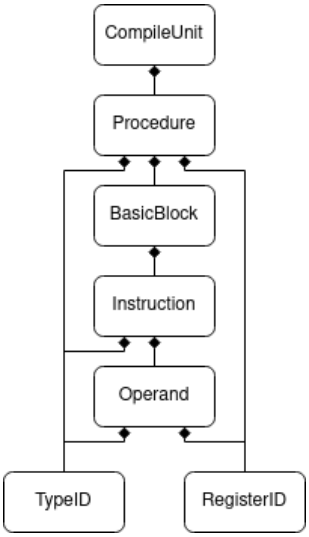
4.2 INTERFACES PARA A MANIPULAÇÃO DE RI

libucb-core manipula a RI através de uma série de classes. *TypeID* representa um tipo e possui apenas dois membros. O membro inteiro *val* indica qual o tipo representado pelo identificador, valores menores que 10 são reservados pela biblioteca para tipos simples proporcionados pelo compilador (como inteiros, ponto flutuante e outros tipos especiais), enquanto valores a partir de 10 são usados para tipos compostos (ponteiros e vetores). O membro inteiro *size* indica o tamanho em bits do tipo representado. Além destes membros, *TypeID* também conta com a implementação de operadores de comparação para o seu uso em algoritmos da *Standard Template Library* (STL).

De forma similar ao *TypeID*, *RegisterID* representa um registrador e possui apenas dois membros. O membro inteiro *val* indica qual o registrador representado pelo identificador, valores de 1 a 99 estão reservados para registradores físicos, enquanto valores a partir de 100 são usados para registradores virtuais e o valor 0 representa um registrador nulo. O membro

inteiro *size* indica o tamanho em bits do registrador. Além destes membros, *RegisterID* também conta com a implementação de operadores de comparação para o seu uso em algoritmos da STL. A Figura 13 mostra a relação das classes responsáveis pela composição da RI dentro do compilador.

Figura 13 – Diagrama ilustrando o relacionamento das classes responsáveis pela composição da RI



Fonte: O próprio autor

As instruções de RI são representadas através da classe *Intruction*, onde cada instrução é composta de um código de operação, um *TypeID*, uma lista de operandos e, no caso de instruções *call*, um identificador. Os métodos públicos da classe *Instruction* são:

- *dump(std::ostream& out)*: escreve informação de depuração para a saída *out*;
- *verify()*: valida a instrução, abortando o programa no caso de uma instrução malformada;
- *parent()*: retorna um ponteiro para o bloco base que contém a instrução;
- *op()*: retorna o código de operação da instrução;
- *ty()*: retorna o *TypeID* da instrução;
- *id()*: retorna o identificador da instrução;
- *add_operand(Operand opnd)*: cria um novo operando no final da lista de operandos;
- *opnds()*: retorna uma referência para a lista de operandos;
- *is_terminator()*: retorna verdadeiro caso o código de operação da instrução seja um terminador (*ret*, *br* ou *brc*);
- *is_store()*: retorna verdadeiro caso o código de operação da instrução seja *store*; e

- *is_side_effect()*: retorna verdadeiro caso a instrução implique em uma mudança no status do programa (caso ela seja um terminador ou *store*).

Os operandos das instruções são representados pela classe *Operand*, onde cada operando é composto de um *TypeID* (que não necessariamente deve ser o mesmo da instrução da qual o operando faz parte), uma categoria (registrador virtual, constante inteira, constante de ponto flutuante ou bloco base), uma *flag* indicando se o operando é uma leitura ou definição e o valor do operando em si. Os métodos públicos da classe *Operand* são:

- *dump(std::ostream& out)*: escreve informação de depuração para a saída *out*;
- *parent()*: retorna um ponteiro para o procedimento do qual o operando faz parte;
- *kind()*: retorna a categoria do operando;
- *ty()*: retorna o *TypeID* da instrução;
- *is_def()*: retorna verdadeiro caso o operando seja uma definição;
- *get_virtual_reg()*: retorna o *RegisterID* do operando, caso este seja um registrador virtual;
- *get_bblock_idx()*: retorna o índice do bloco base, caso o operando seja um bloco base;
- *get_integer_val()*: retorna o valor do operando, caso este seja uma constante inteira com sinal;
- *get_unsigned_val()*: retorna o valor do operando, caso este seja uma constante inteira sem sinal; e
- *get_float_val()*: retorna o valor do operando, caso este seja uma constante de ponto flutuante.

Instruções de máquina e os seus respectivos operandos são representados pelas classes *MachineInstruction* e *MachineOperand*, respectivamente. Essas classes não possuem métodos, apenas membros. *MachineOperand* é composta de um *TypeID*, uma categoria (imediato, registrador, memória, bloco base ou deslocamento da pilha de execução), um valor e duas *flags*, uma indicando se o operando é um uso e outra indicando se o mesmo é uma definição. *MachineInstruction* é composta de um código de operação, uma lista de operandos e, no caso de instruções *call*, um identificador.

A classe *BasicBlock* representa um conjunto linear de instruções, onde a última instrução é uma instrução terminadora. Um bloco base é composto de uma lista de instruções de RI, uma lista de instruções de máquina e um identificador. Os métodos públicos da classe *BasicBlock* são:

- *dump(std::ostream& out)*: escreve informação de depuração para a saída *out*;
- *parent()*: retorna um ponteiro para o procedimento que contém o bloco base;
- *id()*: retorna uma referência para o identificador do bloco base;
- *insts()*: retorna uma referência para a lista de instruções de RI;
- *machine_insts()*: retorna uma referência para a lista de instruções de máquina;
- *append_instr(ARGS... args)*: adiciona uma instrução de RI no final da lista de instruções de RI;
- *prepend_instr(ARGS... args)*: adiciona uma instrução de RI no início da lista de instruções de RI;
- *append_machine_instr(ARGS... args)*: adiciona uma instrução de máquina no final da lista de instruções de máquina;
- *prepend_machine_instr(ARGS... args)*: adiciona uma instrução de máquina no início da lista de instruções de máquina;
- *append_machine_insts(std::list<MachineInstruction> insts)*: copia as instruções da lista *insts* e as insere no final da lista de instruções de máquina;
- *compute_live_ins()*: calcula quais registradores virtuais estão vivos na entrada do bloco base, levando em conta apenas instruções de RI;
- *compute_machine_live_ins()*: calcula quais registradores virtuais estão vivos na entrada do bloco base, levando em conta apenas instruções de máquina;
- *compute_live_outs()*: calcula quais registradores estão vivos na saída do bloco base, tanto para instruções de RI quanto instruções de máquina. Para isso, os registradores *live ins* de todos os blocos base do procedimento devem ter sido previamente calculados;
- *predecessors()*: retorna uma lista de ponteiros para todos os blocos base que antecedem o bloco base corrente;
- *successors()*: retorna uma lista de ponteiros para todos os blocos base que sucedem o bloco base corrente;
- *live_ins()*: retorna uma lista contendo o *RegisterID* e *TypeID* de todos os registradores *live in*;
- *live_outs()*: retorna uma lista contendo o *RegisterID* e *TypeID* de todos os registradores *live out*;

- *clear_dataflow()*: limpa os registradores *live in*, registradores *live out*, predecessores e sucessores de um bloco base; e
- *clear_lifetimes()*: limpa apenas os registradores *live in* e *live out* de um bloco base.

A classe *Procedure* representa um procedimento, o qual pode ser resumido em uma lista de blocos base, identificador e assinatura de tipo[1]. É da responsabilidade do procedimento gerenciar os seus registradores locais e pilha de execução. Os métodos públicos da classe *Procedure* são:

- *dump(std::ostream& out)*: escreve informação de depuração para a saída *out*;
- *context()*: retorna um ponteiro para a unidade de compilação que contém o procedimento;
- *id()*: retorna o identificador do procedimento;
- *signature()*: retorna a assinatura do procedimento
- *params()*: retorna uma referência para uma lista mutável contendo os registradores virtuais referentes aos parâmetros de entrada do procedimento;
- *frame()*: retorna uma referência para uma lista mutável contendo os registradores virtuais referentes aos valores na pilha de execução do procedimento;
- *regs()*: retorna uma referência para uma lista mutável contendo os registradores virtuais restantes do procedimento;
- *bblocks()*: retorna uma referência para uma lista mutável contendo os blocos base do procedimento;
- *entry()*: retorna uma referência para o bloco base de entrada do procedimento;
- *find_bblock(const std::string& id)*: retorna o índice do bloco base com o identificador *id*;
- *get_bblock(int idx)*: retorna um ponteiro para o bloco base com o índice *idx*;
- *add_bblock(std::string id)*: cria um novo bloco base com o identificador *id* e retorna o índice do mesmo;
- *compute_predecessors()*: calcula os predecessores e sucessores de todos os blocos base do procedimento;
- *compute_machine_lifetimes()*: calcula os registradores *live in* e *live out* de todos os blocos base do procedimento;

[1] Uma assinatura de tipo se refere ao tipo retornado pelo procedimento em conjunto do número e tipos dos seus parâmetros. No caso deste projeto também são adicionados aqui os identificadores pelos quais os parâmetros serão usados no corpo do procedimento

- *find_vreg(const std::string& id)*: retorna o *RegisterID* do registrador virtual com o identificador *id*;
- *add_frame_slot(std::string id, TypeID ty)*: adiciona um novo valor na pilha do procedimento e retorna o *RegisterID* para o acesso do mesmo;
- *add_vreg(std::string id, TypeID ty)*: cria um novo registrador virtual, retornando o seu *RegisterID*;
- *operand_from_vreg(const std::string& id, bool is_def)*: método de utilidade para instanciar um operando a partir de um registrador virtual com o identificador *id*; e
- *operand_from_bblock(const std::string& id)*: método de utilidade para instanciar um operando a partir de um bloco base com o identificador *id*.

Por fim, *CompileUnit* representa uma coleção de procedimentos e será compilada para um único arquivo *assembly*. Também é responsabilidade desta classe gerenciar tipos compostos, como ponteiros e vetores. Os seus métodos são:

- *add_procedure(ProcSignature sig, std::string id)*: cria um novo procedimento com a assinatura de tipo *sig* e identificador global *id*, retornando um ponteiro compartilhado para a mesma;
- *get_procedure(const std::string& id)*: busca um procedimento com o identificador *id* e retorna um ponteiro compartilhado para o mesmo;
- *get_ptr_ty(TypeID sub)*: retorna o *TypeID* de um ponteiro para o tipo *sub*;
- *get_arr_ty(TypeID sub, std::uint64_t size)*: retorna o *TypeID* de um vetor do tipo *sub* com *size* elementos; e
- *dump(std::ostream& out)*: método de depuração para a escrita da unidade de compilação na saída *out*.

4.2.1 EXEMPLO DE CRIAÇÃO DA RI A PARTIR DA BIBLIOTECA

A biblioteca pode ser integrada diretamente no compilador do usuário. Supondo que a etapa de análise sintática crie uma estrutura para representar a AST do programa compilado (que posteriormente seja completada e validada pela etapa de análise semântica), o usuário poderá definir funções para criar RI a partir dos nodos da sua AST.

Supondo um compilador para o Algoritmo 16, parte de um possível gerador de RI é descrito no Algoritmo 17. Nele, a função **var_def_stmt_codegen** reserva espaço na pilha de execução para cada variável declarada. A função **var_write_stmt_codegen** gera uma instrução

store para cada escrita em variáveis, enquanto a função **var_read_expr_codegen** gera uma instrução **load** para cada leitura de variáveis. Por fim, a função **add_expr_codegen** gera RI para operações de soma, gerando primeiro a RI dos dois operandos e então criando uma instrução **add** com os dois registradores virtuais resultantes, o registrador virtual resultante da operação de soma é então retornado para que possa ser usado por outras instruções. Outras operações binárias, como subtração, multiplicação, conjunção lógica e etc, podem ser implementadas de maneira similar à operação de soma.

Casos que envolvem o controle de fluxo, como comandos condicionais e laços de repetição, são mais complexos e exigem a criação de novos blocos base. O Algoritmo 18 estende o gerador de RI para suportar comandos condicionais, nele primeiro são criados os blocos base adicionais a serem usados e a RI da expressão condição é gerada no bloco base de entrada (**bblock**). Após a expressão condição, é adicionado um desvio condicional para os blocos base **branch_true** e **branch_false**, as instruções para os comandos equivalentes aos dois blocos base são então adicionadas aos dois blocos individualmente, adicionando também um desvio incondicional para o bloco base **branch_end** ao final dos dois blocos base. Por fim, o bloco base **branch_end** é retornado, para que este se torne o novo ponto de inserção para novas instruções.

Algoritmo 16 – Programa exemplo

```
1  int main()
2  {
3      int a = 8;
4      int b = 10;
5      int c;
6
7      if (a < b)
8      {
9          c = a + b;
10     }
11     else
12     {
13         c = a - b;
14     }
15
16     return c;
17 }
```

Fonte: O próprio autor

Algoritmo 17 – Código em C++ para a geração de RI para expressões

```
1  static CompileUnit unit;
2  static Procedure *proc
3
4  void var_def_stmt_codegen(BasicBlock *bblock, VarDefStmtAST *ast)
5  {
6      proc->add_frame_slot(ast->id, ast->type);
7  }
8
9  void var_write_stmt_codegen(BasicBlock *bblock, VarWriteStmtAST *ast)
10 {
11     Operand ref = proc->operand_from_vreg(ast->id, false);
12
13     RegisterID val_id = expr_codegen(bblock, ast->expr);
14     Operand val(proc, val_id, false);
15
16     Instruction& inst = bblock->append_instr(OP_STORE, ast->type);
17     inst.add_operand(ref);
18     inst.add_operand(val);
19 }
20
21 RegisterID var_read_expr_codegen(BasicBlock *bblock, VarReadExprAST *ast)
22 {
23     Operand var_ref = proc->operand_from_vreg(ast->id, false);
24
25     RegisterID def_id = proc->add_vreg(ast->id, ast->type);
26     Operand def(proc, def_id, true);
27
28     Instruction& inst = bblock->append_instr(OP_LOAD, ast->type);
29     inst.add_operand(def);
30     inst.add_operand(var_ref);
31
32     return def_id;
33 }
34
35 RegisterID add_expr_codegen(BasicBlock *bblock, AddExprAST *ast)
36 {
37     RegisterID lhs_id = expr_codegen(bblock, ast->lhs);
38     Operand lhs(proc, lhs_id, false);
39
40     RegisterID rhs_id = expr_codegen(bblock, ast->rhs);
41     Operand rhs(proc, rhs_id, false);
42
43     RegisterID def_id = proc->add_vreg(ast->id, ast->type);
44     Operand def(proc, def_id, true);
45
46     Instruction& inst = bblock->append_instr(OP_ADD, ast->type);
```

```
47      inst.add_operand(def);
48      inst.add_operand(var_ref);
49
50      return def_id;
51  }
```

Fonte: O próprio autor

Algoritmo 18 – Código em C++ para a geração de RI para comandos condicionais

```
1   BasicBlock *if_stmt_codegen(BasicBlock *bblock, IfStmtAST *ast)
2   {
3       RegisterID cond_id = bool_expr_codegen(bblock, ast);
4       Operand cond(proc, cond_id, false);
5
6       int bblock_true_idx = proc->create_block(new_label_name());
7       Operand bblock_true_opnd = proc->operand_from_bblock(bblock_true_idx);
8
9       int bblock_false_idx = proc->create_block(new_label_name());
10      Operand bblock_false_opnd = proc->operand_from_bblock(bblock_false_idx);
11
12      int bblock_false_end = proc->create_block(new_label_name());
13      Operand bblock_end_opnd = proc->operand_from_bblock(bblock_end_idx);
14
15      Instruction& inst = bblock->append_instr(OP_BRC, T_STATIC_ADDRESS);
16      inst.add_operand(cond);
17      inst.add_operand(bblock_true_opnd);
18      inst.add_operand(bblock_false_opnd);
19
20      BasicBlock *bblock_true = proc->get_bblock(bblock_true_idx);
21      stmt_codegen(bblock_true, ast->branch_true());
22      Instruction& inst2 = bblock_true->append_instr(OP_BR, T_STATIC_ADDRESS);
23      inst2.add_operand(bblock_end_opnd);
24
25      BasicBlock *bblock_false = proc->get_bblock(bblock_false_idx);
26      stmt_codegen(bblock_false, ast->branch_false());
27      Instruction& inst3 = bblock_false->append_instr(OP_BR, T_STATIC_ADDRESS);
28      inst3.add_operand(bblock_end_opnd);
29
30      return proc->get_bblock(bblock_end_idx);
31  }
```

Fonte: O próprio autor

4.3 IMPLEMENTAÇÃO DO PROJETO

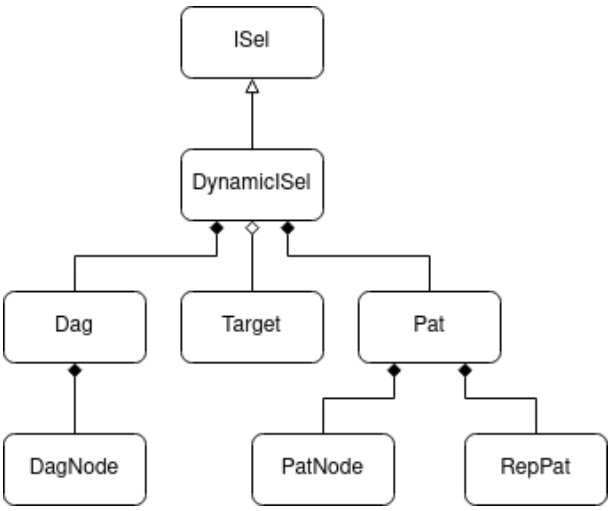
Essa seção descreve as estratégias utilizadas na implementação dos diferentes componentes do *back end*. Para cada componente serão apresentadas as estruturas de dados utilizadas,

uma visão geral da implementação, incluindo as condições sobre as quais os algoritmos são aplicados e possíveis divergências dos algoritmos apresentados no Capítulo 2, e limitações das estratégias utilizadas.

4.3.1 IMPLEMENTAÇÃO DO ALGORITMO DE SELEÇÃO DE INSTRUÇÕES

O algoritmo de seleção de instruções por programação dinâmica pode ser implementado sem grandes alterações. A maior parte da complexidade do algoritmo está nos padrões de comparação, que são isolados do algoritmo de programação dinâmica e poderão ser usados futuramente na implementação de outros algoritmos. A Figura 14 apresenta a relação das classes responsáveis pela seleção de instruções.

Figura 14 – Diagrama ilustrando o relacionamento das classes responsáveis pela seleção de instruções



Fonte: O próprio autor

4.3.1.1 ESTRUTURAS DE DADOS

O seletor de instruções depende de duas estruturas de dados principais *DAG* e *Pat*. *DAG* implementa um grafo acíclico dirigido, onde cada nodo é equivalente a uma instrução de RI. Além da instrução de RI, cada nodo contém o seu custo de substituição, uma lista de dependências diretas da instrução de RI, uma lista de instruções de máquina (selecionadas para a substituição do nodo) e uma lista de dependências da substituição. Durante a execução do algoritmo, o cálculo do custo será executado para cada nodo, onde o padrão de menor custo compatível com o nodo será escolhido. Uma vez que um padrão é escolhido, o custo do nodo e as instruções de máquina do nodo serão atualizadas de acordo com o padrão e a lista de dependências da substituição será preenchida de forma a pular as dependências do nodo que já são cobertas pelo

padrão escolhido. A classe *DAG* também mantém uma lista dos nodos raiz, que são os pontos de entrada do grafo acíclico dirigido e sempre vão gerar ao menos uma instrução de máquina.

A classe *Pat*, apresentada no Algoritmo 19, representa um padrão de substituição. Um padrão de substituição é composto do padrão (a classe *PatNode*, apresentada no Algoritmo 20) em si, o custo do padrão e o padrão de substituição (a classe *RepNode*, apresentada no Algoritmo 21).

O método *match* faz uma comparação recursiva do padrão com o nodo, onde cada nodo é comparado diretamente com um *PatNode*. Caso a comparação seja bem sucedida, o processo é repetido comparando os operandos do *PatNode* com as dependências do nodo, este processo é repetido até que sejam esgotados todos os operandos. Para que a comparação de um nodo com um padrão seja bem sucedida o nodo e o padrão devem compartilhar do mesmo tipo e número de operandos/dependências, em caso de instruções também devem compartilhar do mesmo código de operação e em casos de operandos devem compartilhar da mesma classe de operando[2]. Para facilitar o desenvolvimento de padrões foi desenvolvido um tipo auxiliar (chamado *T_SAME*) que exige que o tipo do operando seja equivalente ao primeiro tipo apresentado no padrão. O Algoritmo 22 ilustra a implementação do padrão para uma instrução de soma onde ambos os operandos são registradores.

A classe *RepNode* contém um código de operação de uma instrução de máquina e os seus operandos, onde os operandos são representados através de índices para as dependências dos nodos do grafo acíclico dirigido. As dependências são acessadas através do método *get_args* que popula um vetor com as dependências dos nodos selecionados, este vetor então é usado em conjunto com o *RepNode* para a criação das instruções de máquina. O Algoritmo 23 ilustra a implementação do padrão de substituição para o Algoritmo 22, o índice -1 indica o registrador resultado do nodo.

Algoritmo 19 – Classe *Pat*

```
1  struct Pat
2  {
3      std::uint8_t cost;
4
5      PatNode pat;
6      std::vector<RepNode> reps;
7
8      MatchResult match(std::shared_ptr<DagNode> n);
9      std::list<MachineInstruction> replace(std::shared_ptr<DagNode> n);
10 };
```

[2] Uma exceção acontece com operandos registradores que não precisam necessariamente ser operandos, podem ser outras instruções cuja execução resulte em um registrador

Algoritmo 20 – Classe *PatNode*

```
struct PatNode
{
    enum Kind
    {
        Inst,
        Opnd
    } kind;

    TypeID ty;
    InstrOpcode opc;
    OperandKind opnd;
    std::string id;
    bool is_va_pat{false};

    std::vector<PatNode> opnds;

    MatchResult match(std::shared_ptr<DagNode> n, TypeID& same_ty);
    void get_args(
        std::shared_ptr<DagNode> n,
        std::vector<std::shared_ptr<DagNode>>& args);
};
```

Algoritmo 21 – Classe *RepNode*

```
struct RepNode
{
    TypeID ty;
    MachineOpc opc;
    std::vector<std::int8_t> opnds;

    bool def_is_also_use{false};
    bool is_va_rep{false};
};
```

Algoritmo 22 – Exemplo de padrão para uma instrução de soma com dois operandos registradores

```
{
    .kind = PatNode::Inst,
    .ty = T_INT_32,
    .opc = InstrOpcode::OP,
    .opnd = OperandKind::OK_POISON,
    .opnds = {
        {
            .kind = PatNode::Opnd,
            .ty = T_SAME,
            .opc = InstrOpcode::OP_NONE,
            .opnd = OperandKind::OK_VIRTUAL_REG
        },
        {
            .kind = PatNode::Opnd,
            .ty = T_SAME,
            .opc = InstrOpcode::OP_NONE,
            .opnd = OperandKind::OK_VIRTUAL_REG
        }
    }
}
```

Algoritmo 23 – Exemplo de padrão para uma instrução de soma com dois operandos registradores

```
{
    {
        .ty = T_INT_32,
        .opc = OPC_MOVE_RR,
        .opnds = {-1, 0}
    },
    {
        .ty = T_INT_32,
        .opc = OPC_ADD,
        .opnds = {-1, 1}
    }
}
```

4.3.1.2 IMPLEMENTAÇÃO

O algoritmo é executado unicamente para cada bloco base. O algoritmo consiste na construção do grafo acíclico dirigido, cálculo de custo e seleção.

Para a construção do grafo acíclico dirigido, primeiro são criados nodos para todos os registradores *live in*. Após isso, o algoritmo irá iterar sobre as instruções criando um nodo para

cada uma delas. Nodos para operandos imediatos são criados à medida que for necessário, enquanto nodos para operandos registradores devem ter sido criados previamente. Nodos para instruções *store*, *br*, *brc* e *ret* são automaticamente marcados como nodos raiz. Outros nodos só são marcados como raiz quando o registrador contendo o seu resultado for usado múltiplas vezes. Uma vez que todas as instruções do bloco base possuem nodos no grafo, o grafo está completo.

Uma vez que o grafo esteja construído, o algoritmo de comparação é aplicado para todos os nodos, começando pelos nodos raiz e recursivamente até os nodos folha. Para cada nodo primeiro é aplicado o algoritmo de comparação nas suas dependências. Após a aplicação do algoritmo, todas as dependências terão um custo calculado e, com base nesse custo, o algoritmo é aplicado no nodo em questão, que será comparado com todos os padrões disponíveis. Uma vez que todos os padrões sejam comparados com o nodo, o padrão de menor custo é escolhido e as instruções de máquina, dependências e custo do nodo são atualizadas. Nodos que não estiverem conectados com nodos raiz são ignorados e não aparecerão no programa final.

Após o cálculo de custo, o algoritmo irá atravessar o grafo começando pelos nodos raiz e seguindo recursivamente pelas dependências atualizadas até os nodos folha. Ao utilizar as dependências atualizadas, o algoritmo irá pular nodos que representam instruções já cobertas por padrões selecionados em outros nodos, atravessando apenas os nodos necessários para a combinação de instruções com o menor custo.

4.3.1.3 LIMITAÇÕES

Atualmente, a implementação do algoritmo exige que todos os nodos sejam combináveis com ao menos um padrão, isso não é estritamente necessário desde que o nodo não combinável seja englobado por um padrão selecionado pelo seu nodo pai. Na implementação atual a não combinação de um nodo é um erro em tempo de cálculo de custo, mas seria mais correto que seja um erro em tempo de seleção.

A classe *RepNode* não é robusta o suficiente. O fato dela exigir que os operandos das instruções sejam operandos dos nodos impede a implementação de algumas instruções que exigem o uso de registradores específicos. Por causa dessas limitações não foram implementadas as instruções *div* (por esta precisar dos registradores RAX e RDX para os seus operandos), *shl* e *shr* (por estas precisarem do registrador CL) para um de seus operandos.

Outro possível ponto de contenção são os padrões variádicos. Padrões marcados como variádicos atualmente são utilizados apenas para instruções *call*, estes padrões aceitam quaisquer dependências do nodo e repassam as mesmas para a instrução de máquina. Cabe ao método *stack_lower* da classe *Target* corrigir as instruções *call*.

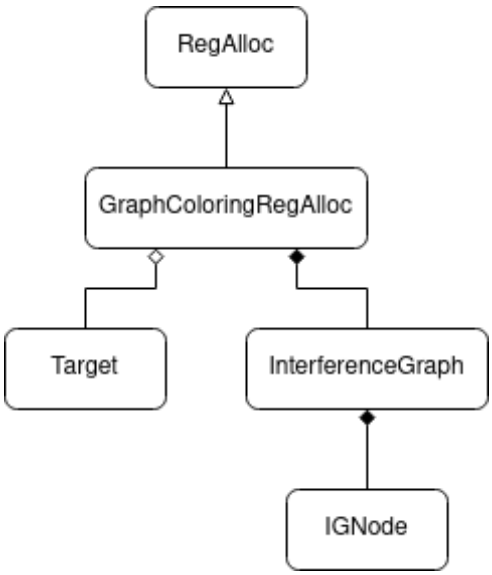
Do ponto de vista de eficiência, o seletor tem um problema grave por comparar todos os nodos com todos os padrões, resultando em múltiplas comparações desnecessárias. Uma

possível solução seria segmentar os padrões por tipo e/ou código de operação. Supondo uma segmentação rudimentar por tipos, com 4 grupos (padrões inteiros com sinal, sem sinal, ponto flutuante e outros) poderia reduzir de 3 a 4 vezes o número de comparações, supondo que os grupos tenham um número similar de padrões.

4.3.2 IMPLEMENTAÇÃO DO ALGORITMO DE ALOCAÇÃO DE REGISTRADORES

A implementação do algoritmo de alocação de registradores por coloração de grafos é mais complexa do que a implementação do algoritmo de seleção de instruções, isso acontece principalmente pelo fato deste algoritmo possuir mais etapas e um maior acoplamento com as suas estruturas de dados. Este algoritmo também foi implementado sem grandes alterações. A Figura 15 apresenta a relação das classes responsáveis pela alocação de registradores.

Figura 15 – Diagrama ilustrando o relacionamento das classes responsáveis pela alocação de registradores



Fonte: O próprio autor

4.3.2.1 ESTRUTURAS DE DADOS

A alocação de registradores por coloração de grafo conta com uma estrutura de dados principal, o grafo de interferência. No grafo de interferência, um nodo representa um registrador virtual, para o qual deve ser alocado um registrador físico, e as arestas indicam uma sobreposição no tempo de vida dos registradores virtuais, o que significa que ambos os registradores não podem compartilhar o mesmo registrador físico. A classe *IGNode* (apresentada no Algoritmo 24) implementa um nodo para um grafo de interferência, os membros dessa classe são:

- *keys* contém os registradores virtuais representados pelo nodo;
- *interferences* contém as interferências do nodo;
- *moves* contém ponteiros para as instruções de movimentação potencialmente redundantes referentes às chaves do nodo, essas instruções serão removidas do procedimento ou reconhecidas como não redundantes antes da coloração do nodo;
- *physical_register* contém o registrador físico, ou "cor", do nodo, nodos podem ser pré-coloridos; e
- *no_spill* deve ser marcado como verdadeiro no caso de nodos pré-coloridos.

Algoritmo 24 – Classe *IGNode*

```
1  struct IGNode
2  {
3      std::set<RegisterID> keys;
4      std::set<RegisterID> interferences;
5      std::set<MachineInstruction*> moves;
6
7      RegisterID physical_register{NO_REG};
8
9      bool no_spill{false};
10
11     bool interferes_with(const IGNode& other);
12
13     void dump();
14 };
```

A classe *InterferenceGraph* representa o grafo de interferência em si. Além dos nodos, os métodos principais dessa classe são:

- *is_interference(RegisterID a, RegisterID b)*: retorna verdadeiro caso haja uma interferência entre os nodos *a* e *b*;
- *add_interference(RegisterID a, RegisterID b)*: cria uma interferência entre os nodos *a* e *b*;
- *get(RegisterID key)* retorna o nodo com a chave *key*:, criando um novo nodo se necessário;
- *get_neighbor_count(IGNode& node, int k)*: retorna o número de vizinhos do nodo *node* com *k* vizinhos ou mais;
- *briggs(IGNode& a, IGNode& b, int k)*: retorna verdadeiro caso os nodos *a* e *b* possam ser coalescidos, de acordo com os critérios propostos na Seção 2.5.2;

- *can_simplify()*: retorna verdadeiro caso o grafo possa ser simplificado, de acordo com os critérios propostos na Seção 2.5.2;
- *coalesce(int k)*: tenta coalescer dois nodos, retornando verdadeiro caso dois nodos tenham sido coalescidos, esse método também retorna uma lista contendo as instruções de movimentação que devem ser removidas;
- *freeze_move()*: reconhece uma instrução de movimentação como não redundante e remove esta instrução das listas dos nodos onde ela é presente;
- *pop_node(int k)*: remove um nodo com menos de *k* vizinhos, retornando o nodo se houver um;
- *pop_highest_degree()*: remove o nodo com o maior número de vizinhos, retornando o nodo se houver um;
- void *push_node(IGNode node)*: adiciona um nodo no grafo de interferência.

Além do grafo de interferência, é importante ressaltar a classe *RegisterClass*. Essa classe não contém nenhum método, apenas uma lista de registradores físicos e uma lista de tipos. Durante a alocação de registradores, o alocador irá iterar sobre uma lista de *RegisterClass*, executando a alocação de registradores para cada uma delas. Durante a alocação de registradores serão ignorados quaisquer registradores cujos tipos não estejam presentes na lista de tipos da classe de registradores, e para a alocação de uma classe de registradores serão usados apenas os registradores físicos presentes na classe.

4.3.2.2 IMPLEMENTAÇÃO

Diferente da seleção de instruções, a alocação de registradores é realizada para o procedimento como um todo, e não unicamente para cada bloco base. Para cada procedimento, o algoritmo irá primeiro carregar as classes de registrador do alvo através do método *load_reg_classes()* da classe *Target*, o algoritmo será então executado uma vez para cada classe. Neste momento, o método *abi_lower()* já foi executado para o procedimento, fazendo com que este tenha os valores de entrada e saída nos registradores físicos apropriados.

Primeiramente, é criado o grafo de interferência para o procedimento. A criação do grafo é feita através de um *set* de registradores vivos. Para cada bloco base do procedimento, o *set* de registradores vivos é inicializado com os registradores *live out* do bloco base. As instruções do bloco base são então iteradas da última para a primeira, removendo do *set* todas as definições e adicionando todos os usos de registradores[3]. Para cada registrador adicionado no

[3] Isso precisa ser feito nessa ordem para evitar interferências entre o resultado de uma instrução e os argumentos da mesma. No caso da instrução ser um definição e um uso ao mesmo tempo, nada é alterado

set de registradores vivos é adicionada uma interferência com cada um dos outros registradores vivos.

Partindo do grafo de interferência, o algoritmo da Seção 2.5.2 é aplicado até que todos os registradores virtuais tenham sido alocados em registradores físicos. O algoritmo foi implementado com duas alterações:

- Na etapa *simplify*, registradores pré-coloridos são empilhados por último. Isso é feito pois os últimos nodos a serem empilhados na etapa *simplify* são os primeiros a serem alocados na etapa *select*, empilhar nodos pré coloridos por último evita conflitos durante a seleção.
- A etapa de *spill* não foi implementada, caso o algoritmo chegue na etapa de *spill* o programa será abortado.

Foram implementadas duas classes de registradores, uma para registradores de ponto flutuante e outra para registradores inteiros e ponteiros.

4.3.2.3 LIMITAÇÕES

A maior limitação da implementação do algoritmo é a não implementação da etapa de *spill*. O compilador ainda é funcional devido ao número de registradores de propósito geral disponível na arquitetura, mas em situações onde existem mais registradores vivos simultaneamente do que registradores físicos o compilador pode não ser capaz de compilar programas válidos.

Outra limitação é o fato dos nodos do grafo de interferência serem baseados em registradores como um todo e não nos distintos tempos de vida de um registrador, isso implica no mapeamento de cada registrador virtual para um único registrador físico, mesmo que este registrador não esteja vivo constantemente, o que aumenta desnecessariamente a pressão de registradores. Um grafo de interferência que leva em conta os distintos tempos de vida dos registradores virtuais poderia potencialmente alocar múltiplos registradores físicos para os distintos tempos de vida de um registrador virtual, potencialmente diminuindo a pressão de registradores e instruções de movimentação desnecessárias.

4.3.3 IMPLEMENTAÇÃO DA ARQUITETURA ALVO X64

Uma arquitetura alvo é implementada através da classe abstrata *Target*, é consultando essa classe que o restante do compilar saberá como transformar o programa de entrada em um programa executável. A classe Target define uma série de métodos virtuais a serem implementados, estes são:

- *load_pats()*: é o método utilizado no seletor de instruções para carregar os padrões de comparação, ele retorna uma lista de padrões;

- *load_reg_classes()*: é o método utilizado no alocador de registradores para carregar as classes de registradores, ele retorna uma lista de classes de registrador;
- *dump_proc(Procedure& proc, std::ostream& out)*: método de depuração para a escrita do procedimento *proc* na saída *out*, deve ser usado quando o procedimento contém instruções de máquina;
- *dump_bblock(BasicBlock& bblock, std::ostream& out)*: método de depuração para a escrita do bloco base *bblock* na saída *out*, deve ser usado quando o bloco base contém instruções de máquina;
- *abi_lower(Procedure& proc)*: será expandido na Seção 4.3.4;
- *stack_lower(Procedure& proc)*: será expandido na Seção 4.3.5; e
- *print_asm(CompileUnit& unit, std::ostream& out, const std::string& src_filename)*: escreve o *assembly* textual da unidade de compilação *unit* em *out*, esse método é chamado no final do processo de compilação.

4.3.4 MÉTODO *ABI_LOWER*

Esse método é chamado após a seleção de instruções para aplicar uma ABI em um procedimento. No caso do *Target* para X64 ele adiciona uma instrução *MOVE* de cada registrador físico de entrada descrito pela ABI para cada registrador virtual de entrada do procedimento. Para as saídas são adicionadas instruções *MOVE* do registrador virtual de saída para o registrador físico proposto pela ABI

O Algoritmo 10 descreve uma função para calcular o fatorial de *n* em RI. Após a seleção de instruções, ela será transformada no Algoritmo 25 que será passado para *abi_lower()*. Após *abi_lower()* o programa será transformado no Algoritmo 26 onde a função tem 3 instruções *MOVE* adicionais, na linha 2 foi adicionado um *MOVE* de *edi* para o registrador virtual *n*, e nas linhas 9 e 19 foram adicionadas instruções de *MOVE* dos dois valores de saída para *rax*.

Algoritmo 25 – Função Fatorial Após a Seleção de Instruções

```
1  fat:
2          mov_mr   32       fs(#0), reg(n)
3          cmp      32       fs(#0), imm(1)
4          jle      64       bb(if_then)
5          jmp      64       bb(if_else)
6
7  if_then:
8          ret 32   eax, imm(1)
9
10 if_else:
11         mov_rm   32       reg(n.1), fs(#0)
12         mov_rr   32       reg(tmp.0), reg(n.1)
13         sub      32       reg(tmp.0), imm(1)
14         call     32       reg(tmp.1), reg(tmp.0)
15         mov_rr   32       reg(tmp.2), reg(tmp.1)
16         imul     32       reg(tmp.2), fs(#0)
17         ret      32       reg(tmp.2)
```

Algoritmo 26 – Função Fatorial após *abi_lower*

```
1  fat:
2          mov_rr   32       reg(n), edi
3          mov_mr   32       fs(#0), reg(n)
4          cmp      32       fs(#0), imm(1)
5          jle      64       bb(if_then)
6          jmp      64       bb(if_else)
7
8  if_then:
9          mov_ri   32       eax, imm(1)
10         ret
11
12 if_else:
13         mov_rm   32       reg(n.1), fs(#0)
14         mov_rr   32       reg(tmp.0), reg(n.1)
15         sub      32       reg(tmp.0), imm(1)
16         call     32       reg(tmp.1), reg(tmp.0)
17         mov_rr   32       reg(tmp.2), reg(tmp.1)
18         imul     32       reg(tmp.2), fs(#0)
19         mov_rr   32       eax, reg(tmp.2)
20         ret
```

4.3.5 MÉTODO *STACK_LOWER*

Esse método é chamado após a alocação de registradores, para adicionar o código da manutenção da pilha de execução de um procedimento. Ela também serve para aplicar a ABI

em quaisquer chamadas de função dentro do procedimento. Ela opera da seguinte maneira:

- Primeiramente, são calculados o tamanho da pilha e os deslocamentos na pilha para todas as variáveis;
- O código para aumentar a pilha de execução é então adicionado no início do procedimento;
- Após o aumento da pilha, são salvos na pilha quaisquer registradores *callee saved* usados pelo procedimento;
- Em seguida, se o procedimento possuir múltiplos pontos de saída, todos os pontos de saída são unificados em um único bloco base;
- Antes da saída do procedimento, os registradores *callee saved* são restaurados;
- A pilha é então restaurada para o seu estado antes do procedimento;
- Tendo o código para a manutenção da pilha aplicado, os deslocamentos na pilha de todas as instruções do procedimento são atualizados; e
- Por fim, a ABI é aplicada nas chamadas de função dentro do procedimento.

O Algoritmo 27 mostra o Algoritmo 26 após a alocação de registradores. Após *stack_lower*, o Algoritmo 27 é transformado no Algoritmo 28. Nele foi adicionado o código para o aumento da pilha em 32 bits no início do procedimento. As instruções *RET* foram unificadas em um novo bloco base (*exit*, na linha 22) que contém o código para a restauração da pilha. Todos os acessos à pilha de execução (antes referidos com o prefixo *fs*) foram substituídos por deslocamentos no registrador rbp.

Algoritmo 27 – Função Fatorial Após a Alocação de Registradores

```
1  fat:
2          mov_mr   32      fs(#0), edi
3          cmp      32      fs(#0), imm(1)
4          jle      64      bb(1)
5          jmp      64      bb(2)
6
7  if_then:
8          mov_ri   32      eax, imm(1)
9          ret
10
11 if_else:
12         mov_rm   32      eax, fs(#0)
13         sub      32      eax, imm(1)
14         call     32      eax, eax
15         imul     32      eax, fs(#0)
16         ret
```

Algoritmo 28 – Função Fatorial após *stack_lower*

```
1  fat:
2          push     64      rbp
3          mov_rr   64      rbp, rsp
4          sub      64      rsp, imm(4)
5          mov_mr   32      [rbp - 4], edi
6          cmp      32      [rbp - 4], imm(1)
7          jle      64      bb(1)
8          jmp      64      bb(2)
9
10 if_then:
11         mov_ri   32      eax, imm(1)
12         jmp      64      bb(3)
13
14 if_else:
15         mov_rm   32      eax, [rbp - 4]
16         sub      32      eax, imm(1)
17         mov_rr   32      edi, eax
18         call
19         imul     32      eax, [rbp - 4]
20         jmp      64      bb(3)
21
22 exit:
23         add      64      rsp, imm(4)
24         pop      64      rbp
25         ret
26 }
```

5 CONSIDERAÇÕES FINAIS

A implementação de compiladores é uma tarefa complexa com uma ampla literatura, fatores estes que dificultam o estudo e desenvolvimento de compiladores e, por consequência, o desenvolvimento deste trabalho. Esta complexidade é especialmente evidenciada quando a literatura aborda a geração de código de máquina. Nesta etapa, é comum a literatura deixar de lado arquiteturas de processadores e máquinas virtuais reais em favor de arquiteturas hipotéticas, esperando que os alunos consigam eventualmente extrapolar o conhecimento adquirido para arquiteturas reais.

O problema da arquitetura alvo pode ser resolvido de duas maneiras: fornecendo uma máquina virtual simples como arquitetura alvo para os compiladores dos alunos, ou fornecendo ferramentas que se encarreguem da etapa de geração de código. Geradores de código e máquinas virtuais utilizados na indústria podem ser aprendidos, mas, devido ao seu grande escopo, não é realista esperar que um aluno aprenda a usá-las ao mesmo tempo que aprende a base teórica por trás de como funcionam compiladores. O maior valor de uma ferramenta feita especificamente com o propósito de ser usada em um meio acadêmico é o seu escopo reduzido, permitindo que os alunos a aprendam rapidamente e ganhem experiência em conceitos que facilitarão a sua transição para ferramentas mais complexas utilizadas na indústria (como LLVM e GCC).

5.1 OBJETIVOS ATINGIDOS

A biblioteca de *back end* e o compilador foram ambos desenvolvidos a ponto de se tornarem funcionais, compilando de forma bem sucedida programas executáveis. Esses programas respeitam a ABI Linux e podem ser lincados com outros programas compilados por outros compiladores. Isso tudo é feito com um conjunto de instruções de RI reduzido e uma base de código ordens de magnitude menor do que a de projetos utilizados na indústria, permitindo que os alunos compilem e analisem rapidamente o *back end*.

O projeto como um todo também é extensível o suficiente para que sejam desenvolvidas outras arquiteturas alvo possibilitando que, no futuro, UCB esteja disponível em outros sistemas operacionais e outras arquiteturas de computadores. A biblioteca também permite a implementação de outros algoritmos para a seleção de instruções e alocação de registradores. Isso possibilita que no futuro sejam feitas comparações de múltiplos algoritmos, a fim de testar a sua eficiência e eficácia.

5.2 OBJETIVOS NÃO ATINGIDOS

Estava previsto um sistema de *tags* para adicionar informação de depuração aos programas executados. Esta funcionalidade não foi implementada por falta de tempo.

Estava prevista a implementação de um passe de otimização. Enquanto a interface para a implementação dos passes de otimização foi desenvolvida, juntamente com as funcionalidades para aplicar os passes de otimização, não foi desenvolvido nenhum passe.

Outra falha deste projeto é a falta de testes. Infelizmente a disciplina de compiladores não foi ofertada no semestre em que este trabalho foi desenvolvido, o que privou UCB de potenciais testadores. UCB será usado nos próximos semestres, mas, no momento, não foi suficientemente testado.

5.3 OUTRAS LIMITAÇÕES

O uso de ponteiros tipados na RI é desnecessário já que o compilador, além de não usar aritmética de ponteiros, exige que o tipo do valor seja declarado no momento da de-referência. A manutenção dos tipos de ponteiros em uma unidade de compilação trouxe uma complexidade desnecessária para a implementação do trabalho.

A interface para o uso da biblioteca também não é exatamente prática, a conversão manual de *RegisterID* para operando é repetitiva e propensa a erros. Também não é claro quando é válido ou não reutilizar um operando (diferente de um *RegisterID*, que é sempre válido dentro de um procedimento). A RI por parte da biblioteca tem espaço de melhora e deveria fornecer mais métodos de utilidade para facilitar a criação de RI.

Também seria mais prático se o aluno não precisasse de código externo para executar os seus programas. UCB deveria fornecer rotinas básicas para a comunicação com o sistema operacional e abstrair o uso do *linker*, isso permitiria que os alunos trabalhassem unicamente através da RI do UCB e simplificaria o processo de compilação. Essas rotinas estavam previstas, mas não foram implementadas por falta de tempo.

5.4 TRABALHOS FUTUROS

A contribuição futura mais importante é a implementação de uma funcionalidade para adicionar informação de depuração aos programas gerados pelo UCB. Isso permitirá que os alunos executem os seus programas através de compiladores disponíveis em seus sistemas (como GDB e LLDB). Não menos importante é a simplificação da interface da biblioteca e da RI, uma representação e biblioteca mais simples facilitariam o aprendizado e adoção das mesmas.

Outras contribuições importantes seriam um porte da biblioteca (e compilador) para sistemas Windows, a fim de que os alunos possam utilizar as ferramentas em seu sistema ope-

racional de escolha; e a extensão da especificação da RI para incluir suporte a tipos estruturados. Outras contribuições importantes incluem: a implementação de novas arquiteturas de processadores, a implementação de novos passes de otimização e a implementação de portes da biblioteca para outras linguagens de programação (como Python, Java, Rust e etc).

REFERÊNCIAS

AHO, A. Teaching the compilers course. **SIGCSE Bulletin**, v. 40, p. 6–8, 11 2008.

AHO, A. *et al.* **Compilers: Principles, techniques, and tools**. 2. ed. Boston: Pearson/Addison Wesley, 2007.

AMD64 Technology. Amd64 architecture programmer's manual. 2020. Disponível em: <https://www.amd.com/system/files/TechDocs/24592.pdf>. Acesso em: 25 nov. 2021.

APPEL, A. W.; GINSBURG, M. **Modern Compiler Implementation in C**. 1. ed. Cambridge: Cambrige University Press, 2004.

BAUER, F. L. *et al.* **Compiler Construction, An Advanced Course**. Berlin: Springer-Verlag, 1974. v. 21. (Lecture Notes in Computer Science, v. 21).

CHOW, F. Intermediate representation. **Communications of the ACM**, Association for Computing Machinery (ACM), v. 56, n. 12, p. 57–62, dez. 2013. Disponível em: <https://doi.org/10.1145/2534706.2534720>.

FOLEISS, J. H. *et al.* Scc: Um compilador c como ferramenta de ensino de compiladores. In: **Anais do WEAC 2009**. Porto Alegre, RS, Brasil: SBC, 2009. p. 15–22.

GODBOLT, M. Compiler explorer. 2012. Disponível em: <https://github.com/compiler-explorer/compiler-explorer>. Acesso em: 2021-10-19.

INTEL. Intel® 64 and ia-32 architectures software developer's manual. 2016. Disponível em: <https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf>. Acesso em: 25 nov. 2021.

KOES, D. R.; GOLDSTEIN, S. C. Near-optimal instruction selection on dags. In: **Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization - CGO '08**. ACM Press, 2008. Disponível em: <https://doi.org/10.1145/1356058.1356065>.

LATTNER, C. **LLVM: An Infrastructure for Multi-Stage Optimization**. Dissertação (Mestrado) — Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. Disponível em: <https://llvm.org/pubs/2002-12-LattnerMSThesis.html>. Acesso em: 25 nov. 2021.

LLVM COMPILER INFRASTRUCTURE. **The LLVM Target-Independent Code Generator**. 2021. <https://llvm.org/docs/CodeGenerator.html>. Acesso em: 19 out. 2021.

MATZ, M. *et al.* System v application binary interface amd64 architecture processor supplement draft version 0.99.6. 2012. Disponível em: <https://refspecs.linuxbase.org/elf/x86_64-abi-0.99.pdf>. Acesso em: 25 nov. 2021.

ROCHA, R. **Online Iterative Compilation Guided by Work-based Profiling**. Tese (Doutorado) — University of Edinburgh, Edinburgh, 08 2017.

SCHEIDER, C.; PASSERINO, L. M.; OLIVEIRA, R. F. de. Compilador educativo VERTO: ambiente para aprendizagem de compiladores. **RENOTE**, Universidade Federal do Rio Grande do Sul, v. 3, n. 2, nov. 2005. Disponível em: <https://doi.org/10.22456/1679-1916.13949>.

STALLMAN, R. M. Gnu compiler collection internals. 2021. Disponível em: <https://gcc.gnu.org/onlinedocs/gccint.pdf>. Acesso em: 25 nov. 2021.

TIS Committee. Executable and linking format (elf) specification. 1995. Disponível em: <https://refspecs.linuxfoundation.org/elf/elf.pdf>. Acesso em: 25 nov. 2021.