Implementação de um compilador para a linguagem C-

Andrew Medeiros de Campos
RA: 111775

São José dos Campos
Outubro de 2020

Andrew Medeiros de Campos
RA: 111775

Implementação de um compilador para a linguagem C-

Relatório apresentado à Universidade Federal de São Paulo como parte dos requisitos para aprovação na disciplina de Laboratório de Sistemas Computacionais: Compiladores.

Docente: Prof. Dr. Luiz Eduardo Galvão Martins
Universidade Federal de São Paulo
Instituto de Ciência e Tecnologia

São José dos Campos
Outubro de 2020

Resumo

Nesse relatório serão apresentadas as ideias trabalhadas no projeto da disciplina Laboratório de Sistemas Computacionais: Compiladores, onde foi proposto o desenvolvimento de um compilador para a linguagem *C-minus* (C-) que gere um arquivo com um código binário executável de instruções provenientes do processador desenvolvido no Laboratório de Arquitetura e Organização de Computadores.

O compilador também deve ser capaz de gerar e apresentar ao usuário as estruturas de análise e síntese utilizadas durante o processo de compilação do código fonte, sendo elas a árvore sintática, a tabela de simbolos, o conjunto de quádruplas de código intermediário e o código assembly referente.

O compilador gerado foi baseado no código de um compilador adaptado para a linguagem *Tiny* presente no livro Compiladores: princípios e práticas, de Kenneth Louden (1), e foi implementado na linguagem C com o auxílio dos softwares Flex e YACC-Bison, que são ferramentas que facilitam a criação de alguns elementos de compiladores.

À seguir será apresentado, além de um resumo da arquitetura e organização do processador, todo o desenvolvimento do projeto descrevendo o funcionamento do compilador.

Palavras-chaves: Compilador. Computadores. Instruções. Análise. Síntese. Código Intermediário. Assembly. C-minus. YACC-Bison. Flex

Lista de ilustrações

Figura 1 – Diagrama Básico do Processador 10
Figura 2 – Diagrama Básico do Processador 11
Figura 3 – Diagrama de Blocos da Fase de Análise 17
Figura 4 – Diagrama de Atividades da Fase de Análise 19
Figura 5 – CFG para o compilador 21
Figura 6 – Exemplo de Árvore Sintática 22
Figura 7 – Exemplo de Tabela de Símbolos 23
Figura 8 – Diagrama de Blocos do Gerador de Código Intermediário 25
Figura 9 – Diagrama de Atividades do Gerador de Código Intermediário 26

Lista de tabelas

Tabela 1 – Lista de Instruções 14
Tabela 2 – Formato das Instruções 15

Sumário

1 INTRODUÇÃO 8

2 O PROCESSADOR 10
2.1 Introdução 10
2.2 Módulos 10
2.3 Instruções 13
2.4 Memória 15

3 COMPILADOR: ANÁLISE 17
3.1 Modelagem 17
3.2 Léxica 20
3.3 Sintática 20
3.4 Semântica 22

4 COMPILADOR: SÍNTESE 24
4.1 Introdução 24
4.2 Geração de CI 24
4.3 Assembly 27
4.4 Binário 27
4.5 Gerenciamento de memória 27

5 EXEMPLOS 29
5.1 Introdução 29
5.2 Código 1 29
5.2.1 Intermediário 29
5.2.2 Assembly 31
5.2.3 Binário 32
5.3 Código 2 33
5.3.1 Intermediário 34
5.3.2 Assembly 35
5.3.3 Binário 36
5.4 Código 3 36
5.4.1 Intermediário 37
5.4.2 Assembly 37
5.4.3 Binário 38

6 CONCLUSÃO 40

REFERÊNCIAS 41

1 Introdução

A vida atual sem a tecnologia é algo impensado, pois muito dos avanços e dos confortos atuais não se existiriam sem os recursos compitacionais disponíveis hoje em dia (2). Para isso ser possível é de extrema importância o desenvolvimento de computadores dotados de processadores, ou dispositivos portando microcontroladores, que automatizam grande parte do esforço braçal, repetitivo ou matemático que é preciso para muitas atividades e ainda com o bônus de fazê-los de uma forma muito mais rápida e com uma precisão muito maior do que se fossem feitas por um humano comum. Além disso também existe a vantagem da distância, uma vez que atividades que antes precisavam ser feitas presencialmente agora podem ser realizadas remotamente, o que foi possível após o desenvolvimento das redes de computadores, que tornaram o mundo muito menor e possibilitou a troca de informação mais rápida entre distâncias cada vez maiores.

Da exploração espacial à facilidade de comprar um produto do outro lado do mundo e recebê-lo em sua casa, tudo isso se tornou possível com o avanço da computação e do desenvolvimento de circuitos embarcados. Porém assim como é importante a produção dos processadores e microcontroladores, que são os cérebros das máquinas, também é conseguir informar aos mesmos instruções de operação. Nesse ponto se dá a importância do desenvolvimento de compiladores e linguagens de programação uma vez que a comunicação com as máquinas se dá por meio de sinais binários, o que não é nada intuitivo para um ser humano e acaba dificultando a utilização dos mesmos. O papel do compilador é justamente esse, traduzir uma linguagem de mais alto nível para outra (1) que possa ser entendida pelos computadores.

Visto toda a importância da computação nos dias atuais e a necessidade da criação de compiladores para fazer a linguagem das máquinas se tornar algo próximo da nossa, foi proposto na disciplina de Laboratório de Sistemas Computacionais: Compiladores, da Universidade Federal de São Paulo, a criação de um compilador para a linguagem didática C- proposta no livro de Kenneth Louden (1) para ajudar no entendimento do funcionamento de compiladores. Ao final, o compilador desenvolvido deve processar o código fonte e gerar o um código binário executável referente à um processador já desenvolvido anteriormente e apresentado na Seção 1.2 desse relatório. Além do código binário o compilador também deve ser capaz de mostrar ao usuário os passos intermediários da compilação como a árvore sintática, a tabela de símbolos, o código intermediário e o código assembly referentes ao código fonte. Porém essas informações devem ser geradas apenas caso não haja nenhum erro no código fonte, caso contrário o compilador precisa retornar o tipo do erro (léxico, sintático ou semântico), o numero da linha que o erro se encontra e uma descrição do erro contendo o elemento do código referente.

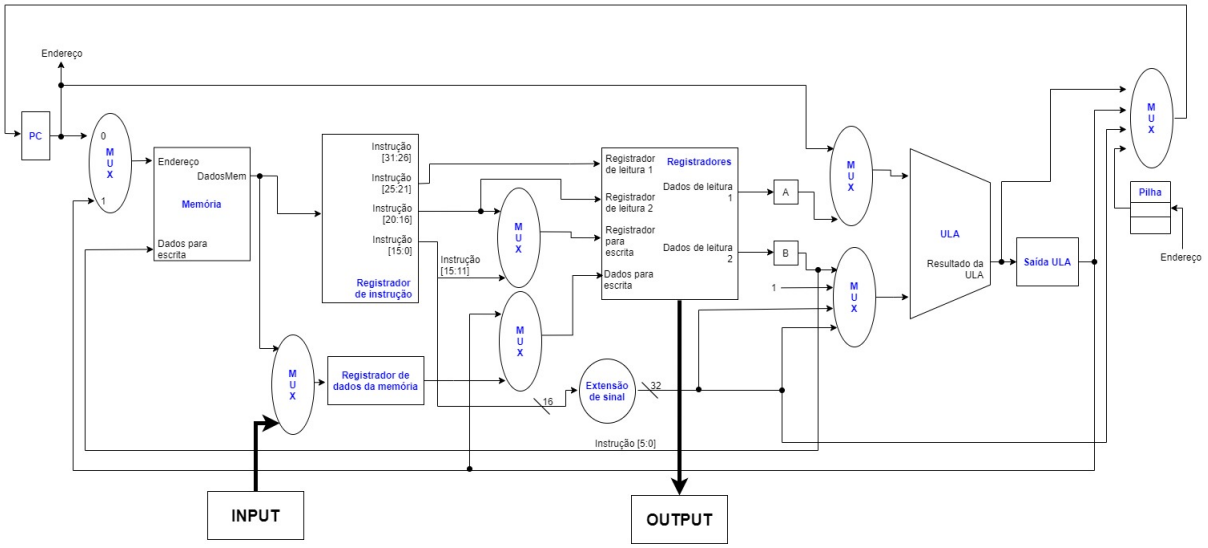
Nesse relatório será comentado todo desenvolvimento desse compilador, começando pela organização e arquitetura do processador para que é voltado, seguindo para o passo de Análise e terminando com o passo de Síntese. Por fim alguns códigos compilados serão apresentados e comparados para demonstrar a funcionalidade do compilador.

2 O Processador

2.1 Introdução

Durante a disciplina de Laboratório de Sistemas Compulacionais: Arquitetura e Organização de Computadores foi proposto o desenvolvimento de uma unidade de processamento juntamente com seu conjunto de instruções. O projeto foi baseado no modelo de organização MIPS multiciclo (2), contendo um conjunto de instruções RISC de 32 bits e que apesar de baseado nessa estrutura de unidade de processamento contém algumas alterações significativas quando comparado ao mesmo, tanto em relão à sua estrutura quanto à seu conjunto de instruções. O *datapath* simplificado do processador pode ser visto abaixo na Figura 1.

Figura 1 – Diagrama Básico do Processador



Fonte: O Autor

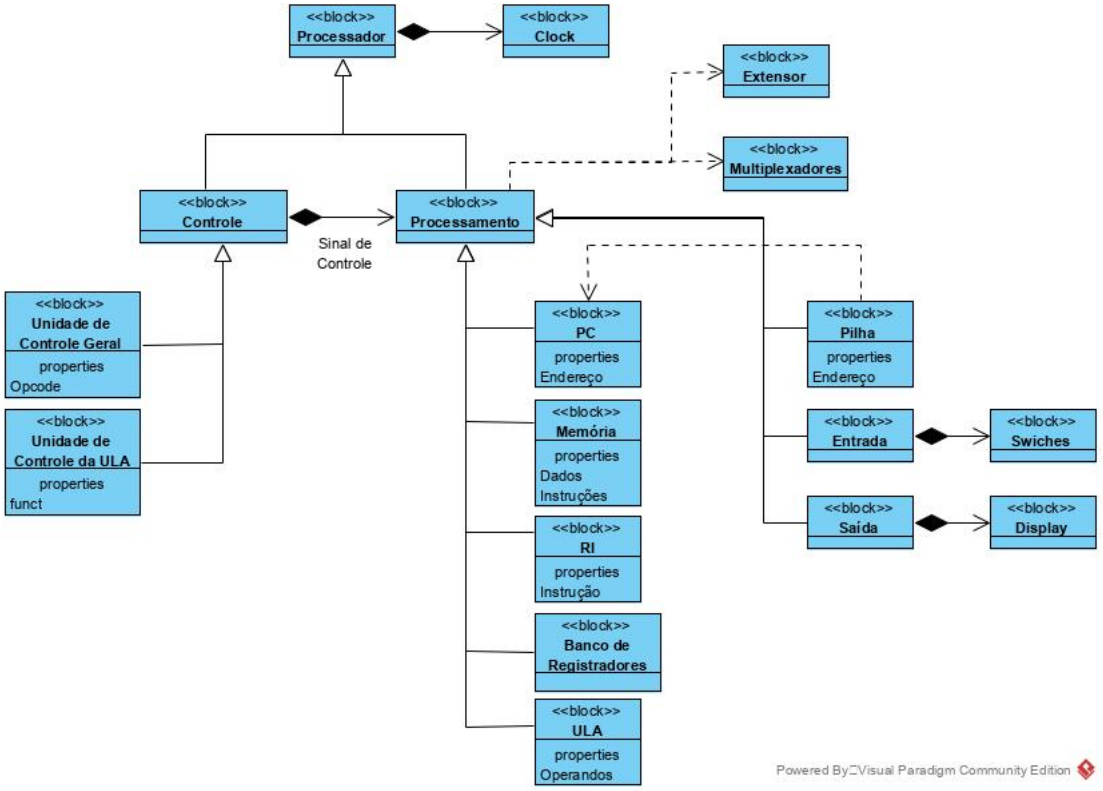
2.2 Módulos

Como pôde ser visto na Figura 1, no processador desenvolvido existem sete módulos diferenciados, sete multiplexadores e seis registradores únicos. A estrutura dos módulos diferenciados, sendo eles Memória, Banco de Registradores, *Input*, *Output*, Extensor de Sinal, ULA e Pilha de Endereços, assim como sua funcionalidade e o propósito dos registradores únicos, *Program Counter*, Registrador de Instrução, Registrador de Dados de Memória, Registradores A e B e Registrador de Saída da ULA, estão explicados à

seguir. Além dos módulos vistos na figura ainda existem os módulos de controle que tem a finalidade de enviar os sinais de funcionamento para todos os demais módulos.

Abaixo na Figura 2 pode ser visto um diagrama de blocos onde é possível analizar o relacionamento entre os módulos do processador. É importante notar que os módulos podem ser divididos em duas categorias, sendo elas Controle e Processamento.

Figura 2 – Diagrama Básico do Processador



Fonte: O Autor

Iniciando pelos módulos diferenciados o primeiro citado foi o **Módulo de memória** que consiste em um banco de registradores de 32 bits com 512 posições de endereço e é utilizado para armazenar tanto as instruções do programa quanto os valores salvos nas variáveis recebendo como valor de endereço o Registrador de Saída da ULA (Saída ULA) ou o valor salvo no *Program Counter* (PC) uma vez que o primeiro contém o valor do imediato contido nas instruções *ldi* e *sti* e o segundo contém o endereço da instrução próxima que será processada, enquanto o valor de escrita é adiquirido do Registrador B.

Em seguida tem-se o **Banco de Registradores**, que assim como o módulo de Memória conta com registradores de 32 bits, porém nesse caso apenas com valores de variáveis ou constantes (como $zero citado mais à frente) que serão utilizados em operações em um futuro próximo.

A Unidade Lógica e Aritimética, ou **ULA**, é o terceiro módulo diferenciado e tem

como premissa utilizar os valores dos registradores A, ou PC, e B, ou um imediato, para realizar operações que serão enviadas parao registrador Saída ULA ou de volta para o PC. A ULA é o único módulo que é controlado por uma unidade de controle específica, sendo ela a Unidade de Controle da ULA.

Os módulos de ***Input*** e ***Output*** são utilizados como mediadores da interface Máquina-Usuário, processando dados para serem exibidos em displays de 7 segmentos ou compilando sinais de *switches* para como um valor de 32 bits para serem carregados no banco de registradores e utilizados pelo programa.

Alguns dados a serem processados, como valores imediatos e endereços, não estão no tamanho padrão de 32 bits portanto são necessários **Extensores de Sinal** que replicam o bit mais significativo do número até o mesmo alcançar 32 bits. Além do extensor de 16 bits para 32 bits, utilizado para extender imediatos, mostrado no datapah e com o funcionamento análogo ao explicado ainda existem outras partes do processador que necessitam de extensões específicas, como para a instução *jmp*, que funcionam pela concatenação de n bits 0 na frente do número para que o mesmo alcance 32 bits.

Por fim, a **Pilha de Endereços** é mais um conjunto de dez registradores de 32 bits porém com um funcionamento e propósito diferente. A Pilha tem a exclusiva função de armazenar endereços salvos pela instrução *jal* quando existe uma chamada de função e retornar esse endereço ao PC com a execussão da instrução *jst*. O acesso aos valores da pilha também se da de forma diferente uma vez que os endereços salvos são inseridos e removidos sempre em forma de pilha, ou seja, sempre empilhando valores no topo e retirando do mesmo lugar. Para isso a pilha conta com um ponteiro dentro de sua estrutura que é incrementado sempre que um novo valor é adicionado e decrementado quando um valor é removido.

Já sobre os registradores únicos, todos tem a mesma estrutura sendo registradores de 32 bits com um sinal de controle para escrita com a diferença sendo apenas no tipo de dado que guardam.

Os registradores mais notáveis são o **PC** e o **Registrador de Instrução**, por serem os mais em destaque. Esses registradores tem respectivamente as funções de guardar o endereço da próxima instrução a ser processada e guardar a instrução propriamente dita. Os registradores **A**, **B** e **Saída ULA** por outro lado são registradores intermediários que guardam valores para operações na ULA (respectivamente os dois operandos e o resultado) para que os mesmo não sejam perdidos ou alterados entre um ciclo e outro de processamento. E por fim o **Registrador de Dados de Memória** é responsavel por guardar dados que serão carregados no banco de registradores, sejam eles vindos da Memória de Dados ou do módulo de entrada.

Por último é importante saber sobre o controle do processador, sendo ele dividido

entre **Unidade de Controle** e **Unidade de Controle da ULA**, onde a primeira é baseada em uma máquina de estados que analiza o quais módulos devem estar ativos em cada ciclo de *clock* e qual seu comportamento, com excessão da ULA que é controlada exclusivamente pela sua unidade específica que, diferente da Unidade de Controle geral se trata de apenas um decodficador de sinais que cruza os sinais de *opcode* e *funct* para enviar um sinal referente à qual operação deve ser realizada.

2.3 Instruções

Para o processador foi optado um conjunto de 34 instruções do tipo RISC, como dito acima e as instruções escolhidas foram baseadas nas instruções presentes no conjunto apresentado no livro Organização e Projeto de Computadores: A Interface Hardware/-Software (2) porém com o nome e os formatos de algumas instruções diferentes. Abaixo na Tabela 1 as instruções escolhidas podem ser vistas com seus respectivos nomes, *opcodes*, formato e representação assembly enquanto a Tabela 2 apresenta a organização de cada formato de instrução.

Tabela 1 – Lista de Instruções

Nome	**Nome**	**Formato**	**Assembly**	**Opcode**	**Funct**
add	Soma	R	add $rf $r1 $r2	000000	000000
sub	Subtração	R	sub $rf $r1 $r2	000000	000001
mult	Multiplicação	R	mult $rf $r1 $r2	000000	000010
div	Divisão	R	div $rf $r1 $r2	000000	000011
and	*And*	R	and $rf $r1 $r2	000000	000100
or	*Or*	R	or $rf $r1 $r2	000000	000101
nand	*Nand*	R	nand $rf $r1 $r2	000000	000110
nor	*Nor*	R	nor $rf $r1 $r2	000000	000111
sle	*Set Less or Equal*	R	sle $rf $r1 $r2	000000	001000
slt	*Set Less Than*	R	slt $rf $r1 $r2	000000	001001
sge	*Set Greater or Equal*	R	sge $rf $r1 $r2	000000	001010
addi	Soma com Imediato	I	addi $rf $r1 xxx	000001	—
subi	Subtração com Imediato	I	subi $rf $r1 xxx	000010	—
divi	Divisão com Imediato	I	divi $rf $r1 xxx	000011	—
multi	Multiplicação com Imediato	I	multi $rf $r1 xxx	000100	—
andi	*Add* com Imediato	I	andi $rf $r1 xxx	000101	—
ori	*Or* com Imediato	I	ori $rf $r1 xxx	000110	—
nori	*Nor* com Imediato	I	nori $rf $r1 xxx	000111	—
slei	*sle* com Imediato	I	slei $rf $r1 xxx	001000	—
slti	*slt* com Imediato	I	slti $rf $r1 xxx	001001	—
beq	*Branch on Equal*	I	beq $r1 $r2 xxx	001010	—
bne	*Branch on Not Equal*	I	bne $r1 $r2 xxx	001011	—
blt	*Branch Less Than*	I	blt $r1 $r2 xxx	001100	—
bgt	*Branch Greater Than*	I	bgt $r1 $r2 xxx	001101	—
sti	*Store* com Imediato	I	sti $rf xxx	001110	—
ldi	*Load* com Imediato	I	ldi $rf xxx	001111	—
str	*Store* com Registrador	I	str $rf xxx	001000	—
ldr	*Load* com Registrador	I	ldr $rf xxx	001001	—
hlt	*Halt*	SYS	hlt	001010	—
in	*Input*	SYS	in $rf	001011	—
out	*Output*	SYS	out $rf	001100	—
jmp	*Jump*	J	jmp xxx	001101	—
jal	*Jump and Link*	J	jal xxx	001110	—
jst	*Jump from Stack*	J	jst	001111	—

Fonte: O Autor

Tabela 2 – Formato das Instruções

Tipo	**Operandos**					
R	op	r1	r2	rd**	dc*	funct
bits	6	5	5	5	5	6
I	op	r1	rd**	Imediato		
bits	6	5	5	16		
SYS	op	dc*	rd**	dc*		
bits	6	5	5	16		
J	op	dc*				
bits	6	26				

Fonte: O Autor

**dc:* don't care*, bits insignificantes para a funcionalidade da instrução.*
**rd: registrador destino, registrador onde o resultado da operação será guardado.*

Como é possível notar, apesar dos formatos de instrução serem bem definidos existem três instruções que não se encaixam totalmente em seus tipos, sendo elas *hlt*, *sti* e *ldi*, uma vez que usam um registrador a menos do que o estipulado para seu formato. Também é importante comentar que o símbolo "xxx"representa um valor inteiro referente à um número imediato que pode ser utilizado tanto para operações quanto como endereço de memória.

2.4 Memória

Como pode ser visto na Figura 1 e na Seção 1.2.2, apenas um módulo de memória foi implementado no projeto desse processador, ou seja, foi adotado o princípio de arquitetura Von Neumann onde a memória de instruções se encontra junta com a memória de dados, com uma proporção de aproximadamente 70:30. Para esse projeto optou-se por utilizar de apenas 5 modos de endereçamento, sendo eles endereçamento direto, utilizado nas instruções do tipo J, SYS e *branches*, endereçamento por registrador, utilizado nas instruções do tipo R, I e SYS, endereçamento imediato, utilizado nas instruções do tipo I, endereçamento por pilha, exclusivo das instruções *jst* e *jal* e endereçamento por deslocamento, exclusivo das instruções *ldr* e *str*. Com isso tem-se uma limitação para o tamanho da memória, uma vez que o maior endereço que a instrução *jmp* alcança é 67.108.863 por existirem apenas 26 bits direcionados para endereço, ou seja, existe um limite para o tamanho máximo de memória de aproximadamente 8 MB, porém essa limitação não afeta o projeto uma vez que, para simplificação de compilação e otimização do uso da memória da FPGA durante o teste foi utilizado uma memória com 512 posições, ou seja, são necessários apenas 9 bits de endereço para acessar toda a memória.

Como dito antes, a memória foi dividida em um bloco de aproximadamente 70% e outro de 30%, onde os endereços entre 0 e 360 foram atribuídos para instruções e os endereços entre 361 e 511 foram atribuídos para dados, sendo esse segundo divido entre varíaveis globais (poições entre 361 e 461) e variáveis locais (posições entre 461 e 511).

Assim como a memória, o banco de registradores também foi dividido. Os 32 registradores foram divididos em cinco categorias sendo elas Zero, Variáveis, Parâmetros, Retorno e Ponteiro, onde o primeiro registrador foi definido como categoria Zero, uma vez que o mesmo guarda apenas o valor zero e leva o nome $zero, os registradores entre 1 e 19 foram definidos como registradores para armazenamento de Variáveis durante operações, sendo nomeados com valores entre $r1 e $r19, os entre 20 e 29 são de categoria Parâmetros e são utilizados para a passagem de parâmetros entre funções e nomeados de $p1 à $p10, e os dois últimos registradores foram categorizados respectivamente como Retorno e Ponteiro, onde o primeiro nomeado como $ret carrega o valor de retorno de uma função e o segundo nomeado como $lp é utilizado para guardar endereços de memória de inicio de escopos.

3 Compilador: Análise

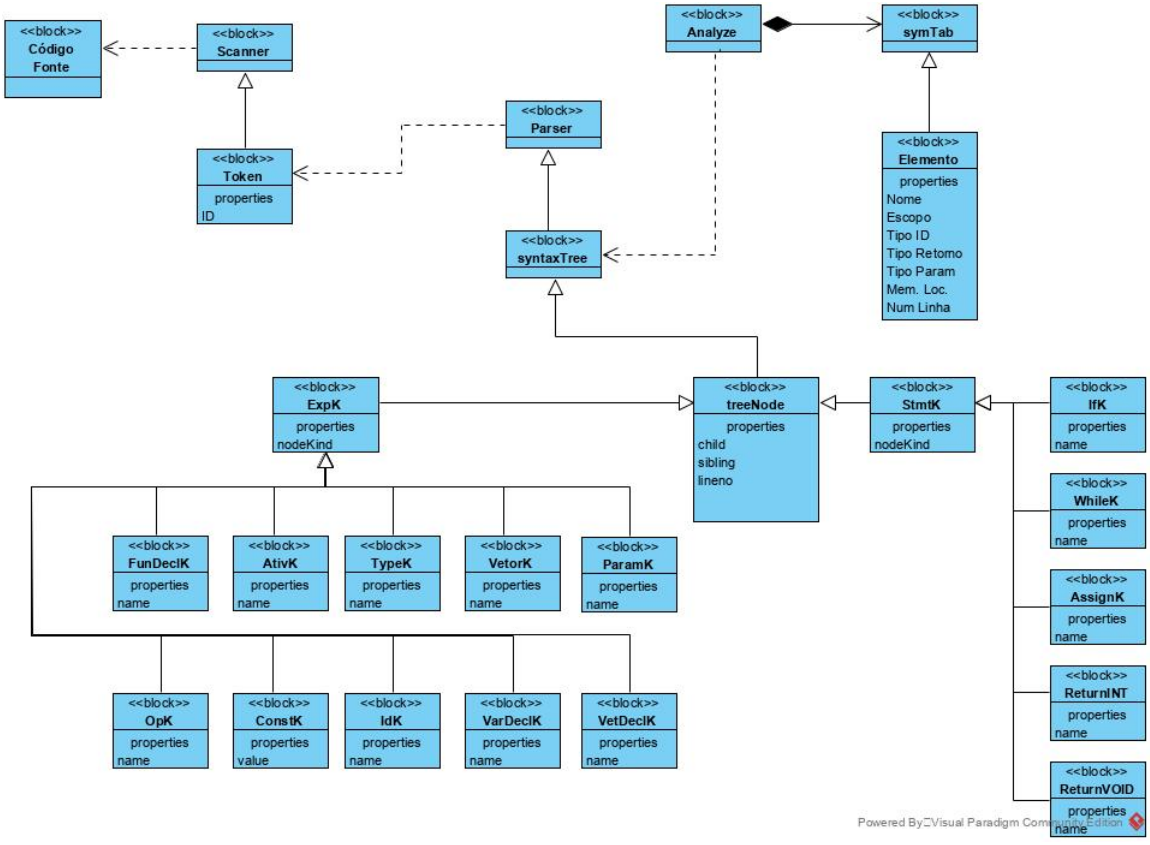
3.1 Modelagem

A Fase de Análise é o primeiro processo da compilação do código fonte, onde o compilador percorre todo o código com a finalidade de encontrar erros que podem comprometer o funcionamento do mesmo.

Para o melhor entendimento do funcionamento do compilador, abaixo nas Figuras 3 e 4 estão apresentados os diagramas de blocos e de atividades da fase de análise.

Analisando primeiramente do diagrama de blocos é possível ver a composição desse módulo, contendo nele três sub módulos: *Scanner*, *Parser* e *Analyze*.

Figura 3 – Diagrama de Blocos da Fase de Análise

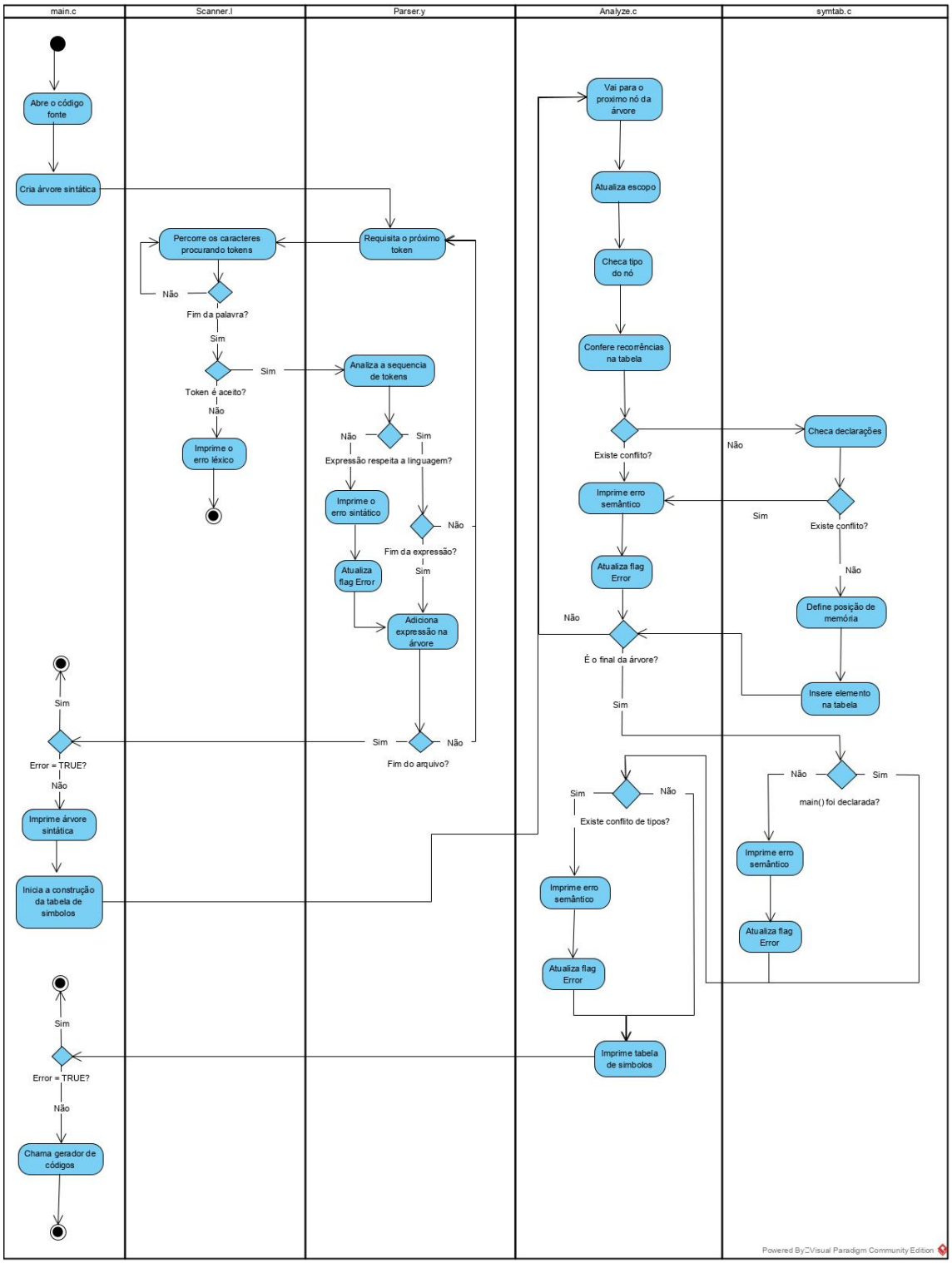


Fonte: O Autor

No diagrama acima é possível ver os três sub módulos de análise e suas generalizações que serão explicadas mais à frente. Outro detalhe que pode ser notado é a interação entre os três sub módulos, onde é possível notar que, iniciando pelo código fonte, existe

uma dependência do *Scanner* para com o código fonte, assim como o *Parser* tem uma dependência para com cada *token* gerado pelo *Scanner* e o sub módulo *Analyze* depende da *syntaxTree* gerada pelo *Parser*. Essa dependência e as ações tomadas por cada módulo pode ser visto na diagrama da Figura 4, onde o arquivo main.c se trata do módulo principal que une todos os sub módulos do compilador e os demais arquivos são referentes aos sub módulos propriamente ditos.

Figura 4 – Diagrama de Atividades da Fase de Análise



Fonte: O Autor

3.2 Léxica

A primeira fase da análise é a Análise Léxica feita pelo elemento chamado de *Scanner*, que se trata da verifiação de cada palavra do código fonte (chamadas de Lexemas). O trabalho do *Scanner* é verificar se os lexemas do código fonte respeitam a linguagem definida e para isso são criadas expressões regulares que definem cada lexema. Para a linguagem C- foram definidos o lexemas para pontuação (parênteses, colchetes, chaves, etc), variáveis (chamados de IDs e sendo qualquer conjunto de letras que não contenha números e não seja uma palavra reservada), operações, comentários e palavras reservadas (int, void, return, etc).

Para a implementação do *Scanner* foi utilizado o software Flex para gerar a classificação dos lexemas, que é um software gerador de analisador que se baseia em contantes ou expressões regulares para criar autômatos para verificar a validade de cada lexema e em seguida, caso o mesmo for válido, retorna um token para que possa ser analizado futuramente.

Abaixo é possível ver os caracteres especiais, palavras reservadas e expressões regulares informadas ao Flex.

- **Expressões**

  digito+ : Números

  letra+ : Identificador

  (letra | digito)+ : Erro

- **Caracteres Especiais**

  + - * / < <= > >= == != = ; , ( ) { } [ ] /* */

- **Palavras reservadas**

  int, void, return, while, if, else

3.3 Sintática

O próximo passo de análise é o Analizador Sintático, chamado aqui de *Parser*, que é a fase respnsável por analizar toda a estrutura do código fonte verificando se a forma do código se encaixa nos padrões da linguagem. Para isso foi utilizado o software YACC-Bison que recebe uma gramática livre de contexto (CFG) (1) e com base nela analisa as estruturas do código fonte, baseando-se nos tokens gerados pelo *Scanner*, para conferir se o código respeita a sintaxe da linguagem. Abaixo pode ser vista a CGF utilizada no projeto do compilador.

Figura 5 – CFG para o compilador

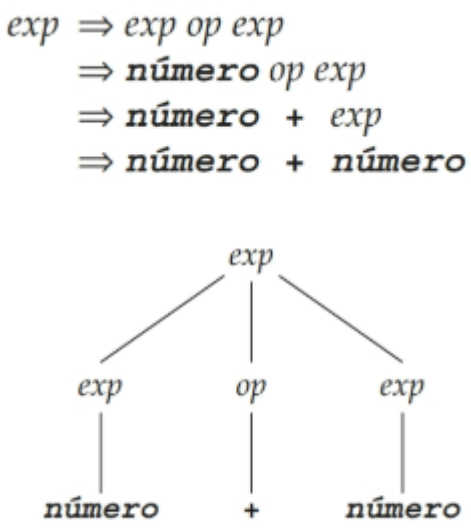
init → lista-dec
lista → lista-dec | declaracao
declaracao → var-dec | fun-dec
var-dec → tipo ID PEV | tipo fun-id ACO tam FCO PEV | error
tam → NUM
tipo → INT | VOID
fun-id → ID
fun-dec → tipo fun-id APR parametros FPR escopo
parametros → VOID | lista-parametros
lista-parametros → lista-parametros VIRG lista-parametros | tipo-parametro
tipo-parametro → tipo ID | tipo ID ACO FCO
escopo → ACH dec-loacis lista-dec-locais FCH | ACH FCH | ACH dec-locais FCH | ACH lista-dec-locais FCH
dec-locais → dec-locais var-dec | var-dec
lista-dec-locais → lista-dec-locais dec-interna | dec-interna
dec-interna → exp-dec | escopo | sel-dec | iteracao-dec |retorno-dec
exp-dec → exp PEV | PEV
sel-dec → IF APR exp FPR dec-interna | IF APR exp FPR dec-interna ELSE dec-interna
iteracao-dec → WHI APR exp FPR dec-interna
retorno-dec → RET exp PEV | RET PEV
exp → var ATR exp | exp-simples
var → ID | fun-id ACO exp FCO
exp-simples → exp-soma relacional exp-soma | exp-soma
relacional → IGL | MENO | MAIO | MAIG | MEIG | DIF
exp-soma → exp-soma soma termo | termo
soma → SOM | SUB
termo → termo mult fator | fator
mult → APR exp FPR | var | ativacao | NUM
ativacao → fun-id APR args FPR | fun-id APR FPR
args → args VIRG exp | exp

Fonte: O Autor

Essa CGF foi baseada na presente do livro Compiladores: princípios e práticas (1) porém com algumas alterações para melhor se adaptar ao projeto.

Outra função do módulo de análise é construir a árvore sintática, que consiste em um estrutura que guarda as derivações da análise do código (1). Essa árvore sintática servirá de base para a análise semântica do código fonte além de guardar informações sobre os elementos. Como pode ser visto na Figura 3, as informações armazenadas pela árvore são o nome do elemento, seu tipo, podendo ser StmtK ou ExpK, um ponteiro que indica o primeiro de seus filhos, a linha do código que se trata e um ponteiro para seu próximo irmão. Um exemplo de árvore sintática pode ser visto abaixo na Figura 6.

Figura 6 – Exemplo de Árvore Sintática



Fonte: Compiladores: Princípios e Práticas (1)

3.4 Semântica

Por fim a análise semântica, último sub módulo de análise, é o responsável por conferir detalhes do código como por exemplo variáveis declaradas multiplas vezes ou não declaradas, conferência de tipos, entre outros erros relacionados ao "contexto"do código. Para tal, essa parte do compilador cria uma estrutura chamada Tabela de Simbolos (*SymTab*), que pode ser vista na Figura 3. Sempre que uma função, um retorno ou uma variável aparecem no código, a mesma é inserida na tabela contendo algumas características sendo elas: Nome, Escopo, Tipo, Tipo de Retorno, Tipo de Parâmetro, Localização na Memória e Número da linha. No momento em que o elemento é inserido, caso seja uma variável ou uma função, é conferido se o mesmo já foi declarado em busca de duplas declarações, nomes já utilizados ou de chamadas de elementos não declarados. Quando todos os elementos ja estiverem inseridos uma busca na tabela é feita com a intenção de buscar a declaração da função main() e por fim uma checagem de tipos é feita para conferir se os retornos das funções estão presentes e corretos assim como se as atribuições feitas no código respeitam os tipos necessários. Um exemplo de Tabela de Símbolos está mostrada abaixo na Figura 7.

Figura 7 – Exemplo de Tabela de Símbolos

Nome	Escopo	Tipo ID	Tipo Retorno	Tipo Param	Mem. Loc.	Num da linha
main	global	fun	INT	VOID	-	1;
vet	main	var	INT	null	361	2;

Fonte: O Autor

4 Compilador: Síntese

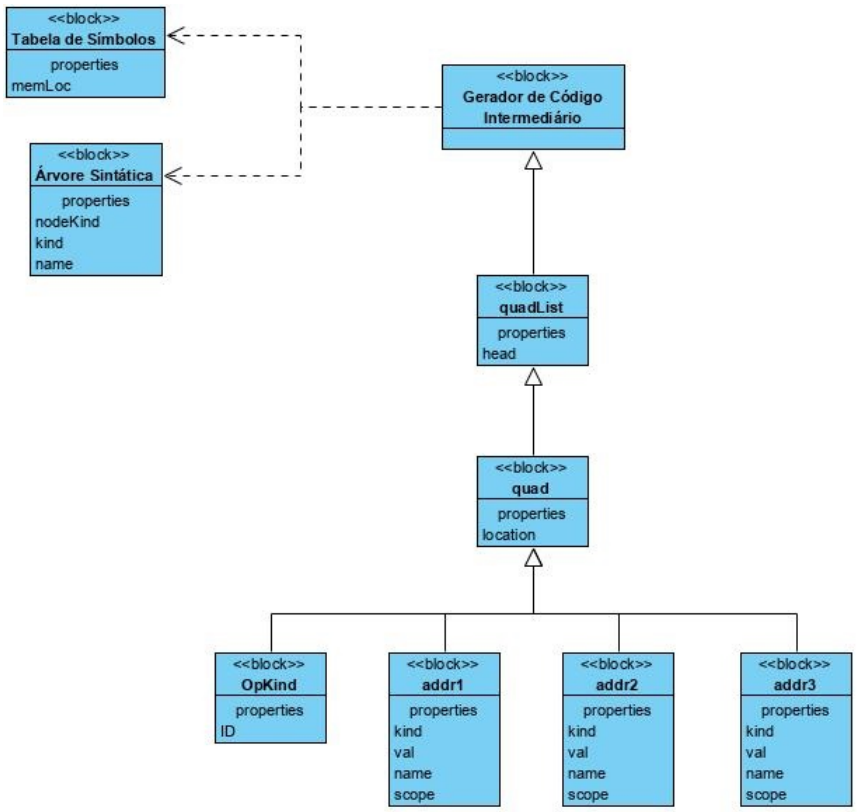
4.1 Introdução

A segunda fase do compilador é a fase se síntese, onde à partir das estruturas geradas na fase anterior, o código fonte será convertido em uma série de instruões entendíveis pelo processador. Porém para isso ser feito são necessários alguns passos para facilitar a conversão sendo eles a geração de código intermediário e geração de código assembly, para assim então ser gerado o código binário. O código intermediário é formado por quádruplas de 3 endereções e um identificador da operação.

4.2 Geração de CI

O código intermediário consiste em uma lista encadeada, onde cada item guarda os três endereços, podendo ser um valor, uma *string* ou um endereço vazio, a localização da quadrupla, que funciona como identificação da mesma, o escopo dos endereços. Detalhes sobre a estrutura do código intermediário pode ser vistos na Figura 3.

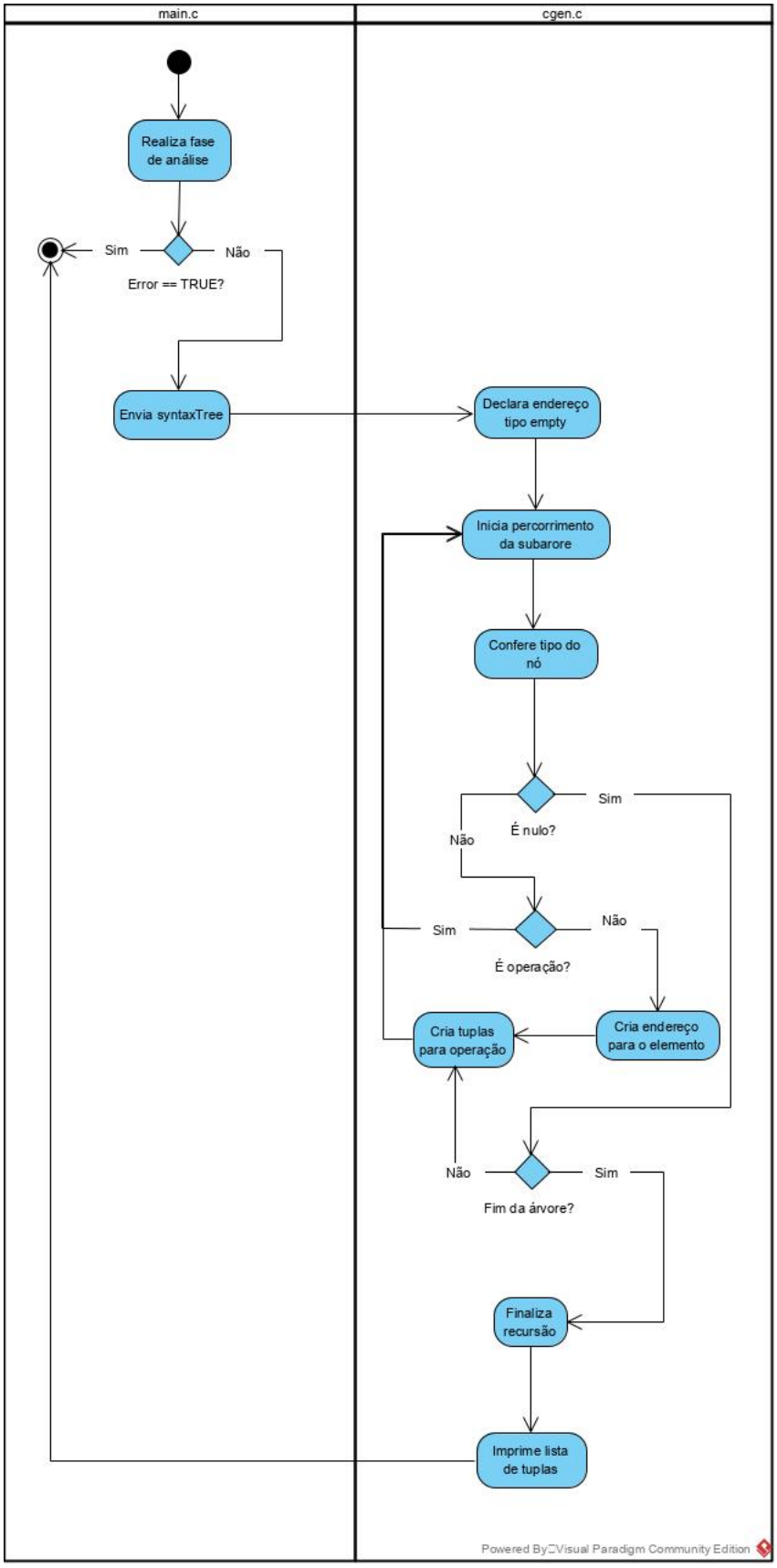
Figura 8 – Diagrama de Blocos do Gerador de Código Intermediário



Fonte: O Autor

A geração do código é feita pelo percorrimento recursivo da árvore sintática e pela análise do tipo de cada nó e caso o nó seja uma variável uma busca na tabela de símbolos para adquirir a posição de memória que a mesma será salva no processador. Assim que o tipo do nó é identificado as tuplas para cada tipo de operação necessárias para realizar o comando vindo do código fonte são criadas e inseridas na lista. O processo mais detalhado da geração de código intermediário pode ser visto na Figura 9 abaixo que contém o diagrama de atividades do módulo.

Figura 9 – Diagrama de Atividades do Gerador de Código Intermediário



Fonte: O Autor

Para o desenvolvimento do projeto foi optado pro um conjunto de 30 tuplas que definem algumas operações realizadas, sendo elas:

opADD, opSUB, opMULT, opDIV, opBLT, opSLE, opBGT, opSGE, opBEQ, opBNE, opAND, opOR, opASSIGN, opALLOC, opADDI, opSUBI, opLOAD, opSTORE, opVEC, opGOTO, opRET, opFUN, opEND, opPARAM, opCALL, opARG, opLABEL, opHLT.

4.3 Assembly

O segundo passo da fase de síntese é a geração de código assembly, que vai aproximar o máximo possível o código fonte do código binário, facilitando tanto o entendimento do código quanto a conversão para binário. Para isso, uma vez que ja se tem o código intermediário, a geração do código assembly consiste em conferir o operando da quádrupla referente à operação da mesma e convertê-la em uma ou mais instruções assembly referentes. As quadras "opALLOC"e "opLABEL"são as únicas que não se referem à instruções assembly, sendo respectivamente relativas à inserção de uma nova variável, junto com seu escopo e sua posição de memória, à uma lista para o acesso mais fácil do módulo e a *labels* que indicam endereços de desvio, condicional ou não.

4.4 Binário

Por fim, o ultimo módulo à ser executado é o gerador de código binário. Esse módulo do compilador é o mais simples de todos, uma vez que todo o trabalho de conversão de código já foi feito nos módulos de Código Intermediário e Assembly, necessitando apenas fazer a conversão direta de cada linha do código Assembly para binário, convertendo os *opcodes* das operações, adicionando os registradores, imediatos e *funct* das instruções que necessitam do mesmo e incluindo *offsets* em todos os campos de instruções que contenham *don't care*.

4.5 Gerenciamento de memória

Como pode ser visto na Seção 1.2.4, a memória foi dividida em três partes: Instruções, Variáveis Globais e Variáveis Locais. O controle da divisão entre as instruções e os dados é feito simplesmente pela variável *location* na estrutura interna do compilador, sendo essa variável responsável por definir as posições de memória de cada variável global. Ao iniciar uma nova função a variável *location* é zerada e passa a armazenar a posição de memória das variáveis no escopo local, e para o cálculo da posição real ser feita o registrador *%lp* é utilizado para ser somado o imediato "MemLoc"presente na tabela de simbolos. A função do registrador *$lp* é armazenar o endereço da posição de memória inicial

da cada função e ele é atualizado toda vez que né feita uma *call* ou o após a finalização da função, a memória possa ser utilizada sem sobrescrever os dados de outras funções. Esse comportamento é justamente o que possibilita o processo de recursão.

Outro comportamento particular que merece destaque é o caso da passagem de vetores como parâmetro em funções. No caso de variáveis, seu valor é simplesmente carregado nos registradores de parâmetro quando existe uma *call* e no escopo da função esse valor é armazenado na posição de memória indicada pela tabela de simbolos. Esse comportamento não é válido para vetores, uma vez que os mesmos não tem seu tamanho definido quando são declarados como parâmetro, portanto recebem apenas uma posição de memória na tabela de simbolos. Isso é contornado passando o vetor em questão por referência na chamada da função, ou seja, diferentemente de variáveis comuns que seu valor é carregado no registrador de parâmetro, no caso de vetores seu endereço é carregado no registrador de parâmetros e salvo na posição de memória alocada para a função. Quando é necessário o uso do mesmo, um deslocamento é feito para a posição original e seu valor é carregado em um registrador para uso.

5 Exemplos

5.1 Introdução

Após a explicação do funcionamento do compilador é importante verificar seu comportamento. Para isso, a seguir, estão três códigos testes que tem o intuito de testar os aspectos de compilação e conferir a integridade da conversão de código.

5.2 Código 1

```
int x;
int vet[5];

int soma (int total[],int a){

    total[2] = vet[3] + x;
    return total;
}

int teste (int a, int b){
    int total;

    total = a + b;
    return total;
}

int main(void){
    int j;
    int a[5];
    j = 14;
    a[j] = 38;

    if(a > 20){
        a = teste(j, 12);
    }
    else{
        a = soma(a, j+16);
    }

    return 0;
}
```

5.2.1 Intermediário

```
(alloc, x, 1, global)
(alloc, vet, 5, global)
(fun, soma, _, _)
(arg, total, _, soma)
```

```
5  (arg, a, _, soma)
6  (addi, $r2, $zero, 2)
7  (vec, $r1, total, $r2)
8  (addi, $r4, $zero, 3)
9  (vec, $r3, vet, $r4)
10 (load, $r5, x, _)
11 (add, $r6, $r3, $r5)
12 (atrib, $r1, $r6, _)
13 (store, $r1, total, $r2)
14 (load, $r7, total, _)
15 (ret, $r7, _, _)
16 (end, soma, _, _)
17 (fun, teste, _, _)
18 (arg, a, _, teste)
19 (arg, b, _, teste)
20 (alloc, total, 1, teste)
21 (load, $r8, total, _)
22 (load, $r9, a, _)
23 (load, $r10, b, _)
24 (add, $r11, $r9, $r10)
25 (atrib, $r8, $r11, _)
26 (store, $r8, total, _)
27 (load, $r12, total, _)
28 (ret, $r12, _, _)
29 (end, teste, _, _)
30 (fun, main, _, _)
31 (alloc, j, 1, main)
32 (alloc, a, 5, main)
33 (load, $r13, j, _)
34 (addi, $r14, $zero, 14)
35 (atrib, $r13, $r14, _)
36 (store, $r13, j, _)
37 (load, $r16, j, _)
38 (vec, $r15, a, $r16)
39 (addi, $r17, $zero, 38)
40 (atrib, $r15, $r17, _)
41 (store, $r15, a, $r16)
42 (load, $r18, a, _)
43 (addi, $r19, $zero, 20)
44 (lequal, $r20, $r18, $r19)
45 (addi, $r1, $zero, 1)
46 (beq, $r20, $r1, L0)
47 (load, $r2, a, _)
48 (load, $r3, j, _)
49 (param, $r3, _, _)
50 (addi, $r4, $zero, 12)
51 (param, $r4, _, _)
52 (call, $ret, teste, 2)
53 (atrib, $r2, $ret, _)
54 (store, $r2, a, _)
55 (goto, L1, _, _)
56 (label, L0, _, _)
57 (load, $r5, a, _)
58 (addi, $r6, $lp, 2)
59 (param, $r6, _, _)
60 (load, $r7, j, _)
61 (addi, $r8, $zero, 16)
62 (add, $r9, $r7, $r8)
63 (param, $r9, _, _)
64 (call, $ret, soma, 2)
```

```
(atrib, $r5, $ret, _)
(store, $r5, a, _)
(label, L1, _, _)
(addi, $r10, $zero, 0)
(ret, $r10, _, _)
(end, main, _, _)
(hlt, _, _, _)
```

5.2.2 Assembly

```
0:      addi $lp $zero 461
1:      jmp 31
.soma
2:      str $p1 $lp 1
3:      str $p2 $lp 2
4:      addi $r2 $zero 2
5:      ldr $r1 $lp 1
6:      add $r2 $r2 $r1
7:      ldr $r1 $r2 0
8:      addi $r4 $zero 3
9:      ldr $r3 $r4 362
10:     ldi $r5 361
11:     add $r6 $r3 $r5
12:     addi $r1 $r6 0
13:     add $r2 $r2 $lp
14:     str $r1 $r2 1
15:     ldr $r7 $lp 1
16:     addi $ret $r7 0
17:     jst
18:     jst
.teste
19:     str $p1 $lp 1
20:     str $p2 $lp 2
21:     ldr $r8 $lp 3
22:     ldr $r9 $lp 1
23:     ldr $r10 $lp 2
24:     add $r11 $r9 $r10
25:     addi $r8 $r11 0
26:     str $r8 $lp 3
27:     ldr $r12 $lp 3
28:     addi $ret $r12 0
29:     jst
30:     jst
.main
31:     ldr $r13 $lp 1
32:     addi $r14 $zero 14
33:     addi $r13 $r14 0
34:     str $r13 $lp 1
35:     ldr $r16 $lp 1
36:     ldr $r15 $r16 2
37:     addi $r17 $zero 38
38:     addi $r15 $r17 0
39:     add $r16 $r16 $lp
40:     str $r15 $r16 2
41:     ldr $r18 $lp 2
42:     addi $r19 $zero 20
43:     sle $zero $r18 $r19
44:     addi $r1 $zero 1
```

```
45:     beq $zero $r1 57
46:     ldr $r2 $lp 2
47:     ldr $r3 $lp 1
48:     addi $p1 $r3 0
49:     addi $r4 $zero 12
50:     addi $p2 $r4 0
51:     addi $lp $lp 6
52:     jal 19
53:     subi $lp $lp 6
54:     addi $r2 $ret 0
55:     str $r2 $lp 2
56:     jmp 69
.L0
57:     ldr $r5 $lp 2
58:     addi $r6 $lp 2
59:     addi $p1 $r6 0
60:     ldr $r7 $lp 1
61:     addi $r8 $zero 16
62:     add $r9 $r7 $r8
63:     addi $p2 $r9 0
64:     addi $lp $lp 6
65:     jal 2
66:     subi $lp $lp 6
67:     addi $r5 $ret 0
68:     str $r5 $lp 2
.L1
69:     addi $r10 $zero 0
70:     addi $ret $r10 0
71:     jmp 73
72:     jmp 73
.end
73:     hlt
```

5.2.3 Binário

```
ram[0] = 000001 00000 11111 0000000111001101   // addi
ram[1] = 010101 00000000000000000000011111   // jmp
ram[2] = 010000 11111 10100 0000000000000001   // str
ram[3] = 010000 11111 10101 0000000000000010   // str
ram[4] = 000001 00000 00010 0000000000000010   // addi
ram[5] = 010001 11111 00001 0000000000000001   // ldr
ram[6] = 000000 00010 00001 00010 00000 000000   // add
ram[7] = 010001 00010 00001 0000000000000000   // ldr
ram[8] = 000001 00000 00100 0000000000000011   // addi
ram[9] = 010001 00100 00011 0000000101101010   // ldr
ram[10] = 001111 00000 00101 0000000101101001   // ldi
ram[11] = 000000 00011 00101 00110 00000 000000   // add
ram[12] = 000001 00110 00001 0000000000000000   // addi
ram[13] = 000000 00010 11111 00010 00000 000000   // add
ram[14] = 010000 00010 00001 0000000000000001   // str
ram[15] = 010001 11111 00111 0000000000000001   // ldr
ram[16] = 000001 00111 11110 0000000000000000   // addi
ram[17] = 010111 00000000000000000000000000   // jst
ram[18] = 010111 00000000000000000000000000   // jst
ram[19] = 010000 11111 10100 0000000000000001   // str
ram[20] = 010000 11111 10101 0000000000000010   // str
ram[21] = 010001 11111 01000 0000000000000011   // ldr
ram[22] = 010001 11111 01001 0000000000000001   // ldr
```

```
ram[23] = 010001 11111 01010 0000000000000010   // ldr
ram[24] = 000000 01001 01010 01011 00000 000000   // add
ram[25] = 000001 01011 01000 0000000000000000   // addi
ram[26] = 010000 11111 01000 0000000000000011   // str
ram[27] = 010001 11111 01100 0000000000000011   // ldr
ram[28] = 000001 01100 11110 0000000000000000   // addi
ram[29] = 010111 00000000000000000000000000   // jst
ram[30] = 010111 00000000000000000000000000   // jst
ram[31] = 010001 11111 01101 0000000000000001   // ldr
ram[32] = 000001 00000 01110 0000000000001110   // addi
ram[33] = 000001 01110 01101 0000000000000000   // addi
ram[34] = 010000 11111 01101 0000000000000001   // str
ram[35] = 010001 11111 10000 0000000000000001   // ldr
ram[36] = 010001 10000 01111 0000000000000010   // ldr
ram[37] = 000001 00000 10001 0000000000100110   // addi
ram[38] = 000001 10001 01111 0000000000000000   // addi
ram[39] = 000000 10000 11111 10000 00000 000000   // add
ram[40] = 010000 10000 01111 0000000000000010   // str
ram[41] = 010001 11111 10010 0000000000000010   // ldr
ram[42] = 000001 00000 10011 0000000000010100   // addi
ram[43] = 000000 10010 10011 00000 00000 001000   // sle
ram[44] = 000001 00000 00001 0000000000000001   // addi
ram[45] = 001010 00001 00000 0000000000111001   // beq
ram[46] = 010001 11111 00010 0000000000000010   // ldr
ram[47] = 010001 11111 00011 0000000000000001   // ldr
ram[48] = 000001 00011 10100 0000000000000000   // addi
ram[49] = 000001 00000 00100 0000000000001100   // addi
ram[50] = 000001 00100 10101 0000000000000000   // addi
ram[51] = 000001 11111 11111 0000000000000110   // addi
ram[52] = 010110 00000000000000000000010011   // jal
ram[53] = 000010 11111 11111 0000000000000110   // subi
ram[54] = 000001 11110 00010 0000000000000000   // addi
ram[55] = 010000 11111 00010 0000000000000010   // str
ram[56] = 010101 00000000000000000001000101   // jmp
ram[57] = 010001 11111 00101 0000000000000010   // ldr
ram[58] = 000001 11111 00110 0000000000000010   // addi
ram[59] = 000001 00110 10100 0000000000000000   // addi
ram[60] = 010001 11111 00111 0000000000000001   // ldr
ram[61] = 000001 00000 01000 0000000000010000   // addi
ram[62] = 000000 00111 01000 01001 00000 000000   // add
ram[63] = 000001 01001 10101 0000000000000000   // addi
ram[64] = 000001 11111 11111 0000000000000110   // addi
ram[65] = 010110 00000000000000000000000010   // jal
ram[66] = 000010 11111 11111 0000000000000110   // subi
ram[67] = 000001 11110 00101 0000000000000000   // addi
ram[68] = 010000 11111 00101 0000000000000010   // str
ram[69] = 000001 00000 01010 0000000000000000   // addi
ram[70] = 000001 01010 11110 0000000000000000   // addi
ram[71] = 010101 00000000000000000001001001   // jmp
ram[72] = 010101 00000000000000000001001001   // jmp
ram[73] = 010010 00000000000000000000000000   // hlt
```

5.3 Código 2

```
int fatorial(int n){
    int vfat;

```

```
4     if ( n <= 1 )
5       return (1);
6     else{
7       vfat = n * fatorial(n - 1);
8       return (vfat);
9     }
10  }
11
12  int main(void){
13    int numero;
14    int f;
15
16    numero = 5;
17
18    f = fatorial(numero);
19
20    return 0;
21  }
```

5.3.1 Intermediário

```
1   (fun, fatorial, _, _)
2   (arg, n, _, fatorial)
3   (alloc, vfat, 1, fatorial)
4   (load, $r1, n, _)
5   (addi, $r2, $zero, 1)
6   (bgt, $r1, $r2, L0)
7   (addi, $r4, $zero, 1)
8   (ret, $r4, _, _)
9   (goto, L1, _, _)
10  (label, L0, _, _)
11  (load, $r5, vfat, _)
12  (load, $r6, n, _)
13  (load, $r7, n, _)
14  (addi, $r8, $zero, 1)
15  (sub, $r9, $r7, $r8)
16  (param, $r9, _, _)
17  (call, $ret, fatorial, 1)
18  (mult, $r10, $r6, $ret)
19  (atrib, $r5, $r10, _)
20  (store, $r5, vfat, _)
21  (load, $r11, vfat, _)
22  (ret, $r11, _, _)
23  (label, L1, _, _)
24  (end, fatorial, _, _)
25  (fun, main, _, _)
26  (alloc, numero, 1, main)
27  (alloc, f, 1, main)
28  (load, $r12, numero, _)
29  (addi, $r13, $zero, 5)
30  (atrib, $r12, $r13, _)
31  (store, $r12, numero, _)
32  (load, $r14, f, _)
33  (load, $r15, numero, _)
34  (param, $r15, _, _)
35  (call, $ret, fatorial, 1)
36  (atrib, $r14, $ret, _)
37  (store, $r14, f, _)
```

```
(addi, $r16, $zero, 0)
(ret, $r16, _, _)
(end, main, _, _)
(hlt, _, _, _)
```

5.3.2 Assembly

```
0:      addi $lp $zero 461
1:      jmp 26
.fatorial
2:      str $p1 $lp 1
3:      ldr $r1 $lp 1
4:      addi $r2 $zero 1
5:      bgt $r1 $r2 10
6:      addi $r4 $zero 1
7:      addi $ret $r4 0
8:      jst
9:      jmp 25
.L0
10:     ldr $r5 $lp 2
11:     ldr $r6 $lp 1
12:     ldr $r7 $lp 1
13:     addi $r8 $zero 1
14:     sub $r9 $r7 $r8
15:     addi $p1 $r9 0
16:     addi $lp $lp 2
17:     jal 2
18:     subi $lp $lp 2
19:     mult $r10 $r6 $ret
20:     addi $r5 $r10 0
21:     str $r5 $lp 2
22:     ldr $r11 $lp 2
23:     addi $ret $r11 0
24:     jst
.L1
25:     jst
.main
26:     ldr $r12 $lp 1
27:     addi $r13 $zero 5
28:     addi $r12 $r13 0
29:     str $r12 $lp 1
30:     ldr $r14 $lp 2
31:     ldr $r15 $lp 1
32:     addi $p1 $r15 0
33:     addi $lp $lp 2
34:     jal 2
35:     subi $lp $lp 2
36:     addi $r14 $ret 0
37:     str $r14 $lp 2
38:     addi $r16 $zero 0
39:     addi $ret $r16 0
40:     jmp 42
41:     jmp 42
.end
42:     hlt
```

5.3.3 Binário

```
ram[0] = 000001 00000 11111 0000000111001101   // addi
ram[1] = 010101 00000000000000000000011010   // jmp
ram[2] = 010000 11111 10100 0000000000000001   // str
ram[3] = 010001 11111 00001 0000000000000001   // ldr
ram[4] = 000001 00000 00010 0000000000000001   // addi
ram[5] = 001101 00010 00001 0000000000001010   // bgt
ram[6] = 000001 00000 00100 0000000000000001   // addi
ram[7] = 000001 00100 11110 0000000000000000   // addi
ram[8] = 010111 00000000000000000000000000   // jst
ram[9] = 010101 00000000000000000000011001   // jmp
ram[10] = 010001 11111 00101 0000000000000010   // ldr
ram[11] = 010001 11111 00110 0000000000000001   // ldr
ram[12] = 010001 11111 00111 0000000000000001   // ldr
ram[13] = 000001 00000 01000 0000000000000001   // addi
ram[14] = 000000 00111 01000 01001 00000 000001   // sub
ram[15] = 000001 01001 10100 0000000000000000   // addi
ram[16] = 000001 11111 11111 0000000000000010   // addi
ram[17] = 010110 00000000000000000000000010   // jal
ram[18] = 000010 11111 11111 0000000000000010   // subi
ram[19] = 000000 00110 11110 01010 00000 000010   // mult
ram[20] = 000001 01010 00101 0000000000000000   // addi
ram[21] = 010000 11111 00101 0000000000000010   // str
ram[22] = 010001 11111 01011 0000000000000010   // ldr
ram[23] = 000001 01011 11110 0000000000000000   // addi
ram[24] = 010111 00000000000000000000000000   // jst
ram[25] = 010111 00000000000000000000000000   // jst
ram[26] = 010001 11111 01100 0000000000000001   // ldr
ram[27] = 000001 00000 01101 0000000000000101   // addi
ram[28] = 000001 01101 01100 0000000000000000   // addi
ram[29] = 010000 11111 01100 0000000000000001   // str
ram[30] = 010001 11111 01110 0000000000000010   // ldr
ram[31] = 010001 11111 01111 0000000000000001   // ldr
ram[32] = 000001 01111 10100 0000000000000000   // addi
ram[33] = 000001 11111 11111 0000000000000010   // addi
ram[34] = 010110 00000000000000000000000010   // jal
ram[35] = 000010 11111 11111 0000000000000010   // subi
ram[36] = 000001 11110 01110 0000000000000000   // addi
ram[37] = 010000 11111 01110 0000000000000010   // str
ram[38] = 000001 00000 10000 0000000000000000   // addi
ram[39] = 000001 10000 11110 0000000000000000   // addi
ram[40] = 010101 00000000000000000000101010   // jmp
ram[41] = 010101 00000000000000000000101010   // jmp
ram[42] = 010010 00000000000000000000000000   // hlt
```

5.4 Código 3

```
int gcd(int u, int v){
    if(v == 0){
        return u;
    }
    else{
        return gcd(v, u-u/v*v);;
    }
}
```

```

void  main ( void ){
    int  x;
    int  y;
    x = input();
    y = input();
    output(gcd(x ,y));
}
```

5.4.1 Intermediário

```
(fun, gcd, _, _)
(arg, u, _, gcd)
(arg, v, _, gcd)
(load, $r1, v, _)
(addi, $r2, $zero, 0)
(bne, $r1, $r2, L0)
(load, $r4, u, _)
(ret, $r4, _, _)
(goto, L1, _, _)
(label, L0, _, _)
(load, $r5, v, _)
(param, $r5, _, _)
(load, $r6, u, _)
(load, $r7, u, _)
(load, $r8, v, _)
(div, $r9, $r7, $r8)
(load, $r10, v, _)
(mult, $r11, $r9, $r10)
(sub, $r12, $r6, $r11)
(param, $r12, _, _)
(call, $ret, gcd, 2)
(ret, $ret, _, _)
(label, L1, _, _)
(end, gcd, _, _)
(fun, main, _, _)
(alloc, x, 1, main)
(alloc, y, 1, main)
(load, $r13, x, _)
(call, _, input, 0)
(atrib, $r13, $ret, _)
(store, $r13, x, _)
(load, $r14, y, _)
(call, _, input, 0)
(atrib, $r14, $ret, _)
(store, $r14, y, _)
(addi, $p1, $ret, 0)
(end, main, _, _)
(hlt, _, _, _)
```

5.4.2 Assembly

```
0:      addi $lp $zero 461
1:      jmp 27
.gcd
2:      str $p1 $lp 1
```

```
3:      str $p2 $lp 2
4:      ldr $r1 $lp 2
5:      addi $r2 $zero 0
6:      bne $r1 $r2 11
7:      ldr $r4 $lp 1
8:      addi $ret $r4 0
9:      jst
10:     jmp 26
.L0
11:     ldr $r5 $lp 2
12:     addi $p1 $r5 0
13:     ldr $r6 $lp 1
14:     ldr $r7 $lp 1
15:     ldr $r8 $lp 2
16:     div $r9 $r7 $r8
17:     ldr $r10 $lp 2
18:     mult $r11 $r9 $r10
19:     sub $r12 $r6 $r11
20:     addi $p2 $r12 0
21:     addi $lp $lp 2
22:     jal 2
23:     subi $lp $lp 2
24:     addi $ret $ret 0
25:     jst
.L1
26:     jst
.main
27:     ldr $r13 $lp 1
28:     in $ret
29:     addi $r13 $ret 0
30:     str $r13 $lp 1
31:     ldr $r14 $lp 2
32:     in $ret
33:     addi $r14 $ret 0
34:     str $r14 $lp 2
35:     addi $p1 $ret 0
36:     jmp 37
.end
37:     hlt
```

5.4.3 Binário

```
ram[0] = 000001 00000 11111 0000000111001101   // addi
ram[1] = 010101 00000000000000000000011011   // jmp
ram[2] = 010000 11111 10100 0000000000000001   // str
ram[3] = 010000 11111 10101 0000000000000010   // str
ram[4] = 010001 11111 00001 0000000000000010   // ldr
ram[5] = 000001 00000 00010 0000000000000000   // addi
ram[6] = 001011 00010 00001 0000000000001011   // bne
ram[7] = 010001 11111 00100 0000000000000001   // ldr
ram[8] = 000001 00100 11110 0000000000000000   // addi
ram[9] = 010111 00000000000000000000000000   // jst
ram[10] = 010101 00000000000000000000011010   // jmp
ram[11] = 010001 11111 00101 0000000000000010   // ldr
ram[12] = 000001 00101 10100 0000000000000000   // addi
ram[13] = 010001 11111 00110 0000000000000001   // ldr
ram[14] = 010001 11111 00111 0000000000000001   // ldr
ram[15] = 010001 11111 01000 0000000000000010   // ldr
```

```
17 ram[16] = 000000 00111 01000 01001 00000 000011   // div
18 ram[17] = 010001 11111 01010 0000000000000010   // ldr
19 ram[18] = 000000 01001 01010 01011 00000 000010   // mult
20 ram[19] = 000000 00110 01011 01100 00000 000001   // sub
21 ram[20] = 000001 01100 10101 0000000000000000   // addi
22 ram[21] = 000001 11111 11111 0000000000000010   // addi
23 ram[22] = 010110 00000000000000000000000010   // jal
24 ram[23] = 000010 11111 11111 0000000000000010   // subi
25 ram[24] = 000001 11110 11110 0000000000000000   // addi
26 ram[25] = 010111 00000000000000000000000000   // jst
27 ram[26] = 010111 00000000000000000000000000   // jst
28 ram[27] = 010001 11111 01101 0000000000000001   // ldr
29 ram[28] = 010011 00000 11110 0000000000000000   // in
30 ram[29] = 000001 11110 01101 0000000000000000   // addi
31 ram[30] = 010000 11111 01101 0000000000000001   // str
32 ram[31] = 010001 11111 01110 0000000000000010   // ldr
33 ram[32] = 010011 00000 11110 0000000000000000   // in
34 ram[33] = 000001 11110 01110 0000000000000000   // addi
35 ram[34] = 010000 11111 01110 0000000000000010   // str
36 ram[35] = 000001 11110 10100 0000000000000000   // addi
37 ram[36] = 010101 00000000000000000000100101   // jmp
38 ram[37] = 010010 00000000000000000000000000   // hlt
```

6 Conclusão

Ao final da análise dos códigos gerados pelo compilador é possível chegar à conclusão que os mesmo cumpre o objetivo do projeto, uma vez que os três códigos apresentados na Seção 1.4 contêm todas as estruturas propostas à linguagem, sendo elas: variáveis locais e globais, uso de vetores tanto declarados quanto passados por parâmetro, chamadas de funções (inclusive com o tratamento de recursão) e estruturas de laço e desvio *while* e *if*. Com relação às instruções do processador, a maioria delas são usadas e as que não são usadas se dão pelo fato de serem redundantes graças a presença de outras que realizam o mesmo processo, como por exemplo a instrução *sgt* que é substituída pela instrução *bgt* para instruções de desvio (único escopo em que essas instruções são utilizadas).

As dificuldades encontradas durante a realização do projeto se deram principalmente em relação à passagem de vetores como parâmetro, que foi reslvida após pesquisas sobre como compiladores para outras linguagens tratam esse comportamento, e o tratamento de recursão que pode ser solucionado graças à análise do relatório feito por Ricardo Elizeu Neto (3) sobre a implementação de seu compilador para a linguagem C-, disponibilizado pelo docente da matéria.

O durante o processo de desenvolvimento do compilador foi possível também encontrar vários pontos onde o processador poderia ser otimizado, por exemplo melhor organização dos *opcodes* e *functs* das intruções, a remoção ou a inserção de instruções devido à necessidade e a re-organização dos registradores e da divisão da memória.

Entre as melhorias futuras encontram-se a adaptação do código para que as instruções com o campo imediato como *multi* e *divi* sejam melhores utilizadas e que instruções de *load* e *store* não seja feitas em momentos de uso recorrente de alguma variável. Essas mudanças tem a finalidade de diminuir a quantidade de instruções desnecessárias geradas e o código final se torne mais otimizado.

Referências

1 LOUDEN K. C.; SILVA, F. S. C. *Compiladores: Princípios e Práticas*. [S.l.]: São Paulo Cengage Learning, 2004. Citado 5 vezes nas páginas 3, 8, 20, 21 e 22.

2 PATTERSON, D. A.; HENNESSY, L. J. *Organização e Projeto de Computadores: A Interface Hardware/Software*. [S.l.]: Elsevier Editora Ltda., 2005. v. 3. Citado 3 vezes nas páginas 8, 10 e 13.

3 NETO, R. E. Implementação do compilador c-. p. 28–29, 2018. Citado na página 40.