

Eduardo Veríssimo Faccio  
148859

## **Implementação do Compilador C-**

São José dos Campos - Brasil

Julho de 2023

Eduardo Veríssimo Faccio  
148859

## **Implementação do Compilador C-**

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

Docente: Thaína Aparecida Azevedo Tosta  
Universidade Federal de São Paulo - UNIFESP  
Instituto de Ciência e Tecnologia - Campus São José dos Campos

São José dos Campos - Brasil  
Julho de 2023

# Listas de ilustrações

Figura 1 – Caminho de dados para o processador desenvolvido.	9
Figura 2 – Diagrama de blocos para a fase de análise	14
Figura 3 – Diagrama de atividades para a fase de análise	15
Figura 4 – Diagrama de blocos para a fase de síntese	19
Figura 5 – Diagrama de atividades para a fase de síntese	20
Figura 6 – Estrutura de dados para o código intermediário, presente no arquivo “codIterm.h”	22
Figura 7 – Função para gerar o código intermediário, com base nos possíveis nós da arvore sintática, no “codIterm.c”	22
Figura 8 – Função para gerar o código intermediário ao encontrar um operador aritmético no “codIterm.c”	23
Figura 9 – Estrutura para armazenar o código assembly gerado, presente no arquivo “assembly.h”	24
Figura 10 – Saída do terminal para a memória de uma função genérica, para um programa compilado	26
Figura 11 – Estrutura de dados para armazenar e gerenciar a memória da pilha de funções, presente no arquivo “memoria.h”	26
Figura 12 – Estrutura de dados para armazenar e gerenciar as <i>labels</i> presentes no código <i>assembly</i> “label.h”	27
Figura 13 – Função “assembly” utilizada para inicializar a geração do código <i>assembly</i> para o processador MIPS em “assembly.c”	27
Figura 14 – Função “geraAssembly” utilizada para gerar efetivamente o código <i>assembly</i> para o processador MIPS, presente no arquivo “assembly.c”	28
Figura 15 – Verificação para a instrução ALLOC presente em “assembly.c”	29
Figura 16 – Verificação para a instrução ARG presente em “assembly.c”	29
Figura 17 – Verificação para a instrução LOAD para vetores alocados no escopo da função, presente em “assembly.c”	30
Figura 18 – Verificação para a instrução LOAD para vetores passados como argumento e variáveis inteiras, presente em “assembly.c”	30
Figura 19 – Verificação para a instrução PARAM, presente em “assembly.c”	32
Figura 20 – Geração de código <i>assembly</i> para a instrução intermediária CALL - parte 1 - presente no arquivo “assembly.c”	33
Figura 21 – Geração de código <i>assembly</i> para a instrução intermediária CALL - parte 2 - presente no arquivo “assembly.c”	33
Figura 22 – Geração de código <i>assembly</i> para a instrução intermediária CALL - parte 3 - presente no arquivo “assembly.c”	34

Figura 23 – Geração de código <i>assembly</i> para a instrução intermediária FUN, presente no arquivo “assembly.c” . . . . .	35
Figura 24 – Geração de código <i>assembly</i> para a instrução intermediária END presente no arquivo “assembly.c” . . . . .	35
Figura 25 – Geração de código <i>assembly</i> para a instrução intermediária RET presente no arquivo “assembly.c” . . . . .	36
Figura 26 – Estruturas para armazenar os valores binários do código, presente no arquivo “binario.h” . . . . .	36
Figura 27 – Função “binario” utilizada para converter o código <i>assembly</i> para binário, implementado no arquivo “binario.c” . . . . .	37
Figura 28 – Funções específicas para a conversão do código <i>assembly</i> em formatos do tipo R, I e J, respectivamente. Implementado no arquivo “binario.c”	38
Figura 29 – Funções para conversão do nome da instrução em seu opcode e funct (para tipo R) em binário. Presente no arquivo “binario.c” . . . . .	38
Figura 30 – Função para mostrar o valor binário para um arquivo esterno. Presente no arquivo “binario.c” . . . . .	39
Figura 31 – Inserindo o valor decimal 6 para o cálculo do fatorial na placa FPGA . .	44
Figura 32 – Resultado obtido para o cálculo do fatorial de 6 na placa FPGA . . . .	44
Figura 33 – Inserindo como primeiro valor o número 15 para o programa do máximo divisor comum . . . . .	48
Figura 34 – Inserindo como segundo valor o número 20 para o programa do máximo divisor comum . . . . .	49
Figura 35 – Número 5 obtido como resultado após o cálculo do máximo divisor comum	49

# **Lista de tabelas**

Tabela 1 – Formato de Instruções do tipo R . . . . .	10
Tabela 2 – Formato de Instruções do tipo I . . . . .	10
Tabela 3 – Formato de Instruções do tipo J . . . . .	11
Tabela 4 – Instruções do tipo R . . . . .	11
Tabela 5 – Instruções do tipo I . . . . .	12
Tabela 6 – Instruções do tipo J . . . . .	12
Tabela 7 – Relação entre símbolos e tokens para o compilador . . . . .	16
Tabela 8 – Sintaxe da linguagem C- utilizada no projeto . . . . .	17
Tabela 9 – Erros semânticos presentes na linguagem C- . . . . .	18
Tabela 10 – Instruções do código intermediário . . . . .	21
Tabela 11 – Valores dos registradores reservados e de uso geral do programa e do compilador . . . . .	25
Tabela 12 – Tabela para a memória da pilha de funções . . . . .	25
Tabela 13 – Geração de códigos da fase de síntese para o programa factorial recursivo	40
Tabela 14 – Resultado da geração de códigos para o programa máximo denominador comum . . . . .	45
Tabela 15 – Código Intermediário . . . . .	50
Tabela 16 – Geração dos códigos assembly e binário . . . . .	57

# Sumário

<b>1</b>	<b>INTRODUÇÃO</b>	<b>7</b>
<b>2</b>	<b>PROCESSADOR MIPS</b>	<b>8</b>
<b>2.1</b>	<b>Arquitetura Básica</b>	<b>8</b>
<b>2.2</b>	<b>Formato de Instruções</b>	<b>9</b>
<b>2.3</b>	<b>Conjunto de Instruções</b>	<b>11</b>
<b>2.4</b>	<b>Modos de Endereçamento</b>	<b>12</b>
2.4.1	A Registrador	12
2.4.2	Imediato	12
2.4.3	Base-Deslocamento	13
2.4.4	Absoluto	13
<b>3</b>	<b>COMPILADOR PARA A LINGUAGEM C-</b>	<b>14</b>
<b>3.1</b>	<b>Fase de Análise</b>	<b>14</b>
3.1.1	Modelagem	14
3.1.1.1	Diagrama de Blocos	14
3.1.1.2	Diagrama de Atividades	15
3.1.2	Análise Léxica	15
3.1.3	Análise Sintática	16
3.1.4	Análise Semântica	18
<b>3.2</b>	<b>Fase de Síntese</b>	<b>19</b>
3.2.1	Modelagem	19
3.2.1.1	Diagrama de Blocos	19
3.2.1.2	Diagrama de Atividades	20
3.2.2	Código Intermediário	20
3.2.3	Código Assembly	24
3.2.3.1	Variáveis Inteiras e Vetores	28
3.2.3.2	Chamada de Funções	31
3.2.4	Código Binário	36
<b>4</b>	<b>RESULTADOS</b>	<b>40</b>
<b>4.1</b>	<b>Fatorial Recursivo</b>	<b>40</b>
4.1.1	Código Fonte	40
4.1.2	Relação entre Códigos	40
4.1.3	Teste FPGA	43
<b>4.2</b>	<b>Máximo Denominador Comum</b>	<b>44</b>

4.2.1	Código Fonte . . . . .	44
4.2.2	Relação entre Códigos . . . . .	45
4.2.3	Teste FPGA . . . . .	48
<b>4.3</b>	<b>Ordenação de Vetores . . . . .</b>	<b>49</b>
4.3.1	Código Fonte . . . . .	49
4.3.2	Código Intermediário . . . . .	50
4.3.3	Código Assembly e Binário . . . . .	56
<b>5</b>	<b>CONSIDERAÇÕES FINAIS . . . . .</b>	<b>65</b>
	<b>REFERÊNCIAS . . . . .</b>	<b>66</b>

# 1 Introdução

Antes da criação dos primeiros compiladores, os códigos para os computadores eram escritos em código binário pelos programadores, o que ocasionava nos seguintes problemas: 1) dificuldade de programação; 2) falta de portabilidade; 3) baixa produtividade; 4) manutenção complexa; e 5) curva de aprendizado íngreme. Isso significava em uma demanda maior de tempo do programador para escrever e buscar por erros no programa que foi desenvolvido, já que códigos binários não são de fácil leitura e precisam ser analisados cuidadosamente. Além disso, era necessário ter em mente toda a arquitetura da máquina em que o código seria escrito, já que o código binário seria diferente e, consequentemente, o programador deveria reaprender tudo novamente.

Sabendo desses problemas, o primeiro compilador foi inventado na década de 1950, que foi idealizado e implementado por uma equipe da IBM para a linguagem de programação FORTRAN ([LOUDEN, 1997](#)). Sua principal característica é trazer ao programador uma abstração maior, sem que esteja preocupado com o hardware em questão, permitindo um escrita rápida de um código e de fácil entendimento por demais leitores, o que ocasionou em um aumento significativo de produtividade e de manutenção futura do mesmo. Ademais, o compilador é capaz de observar erros que o programador não tivesse observado, o que pode poupar tempo em depurações futuras do código.

Dessa forma, o compilador possui dois principais procedimentos: 1) Fase de análise; e 2) Fase de síntese. A primeira retrata a verificação dos erros do código escrito pelo programador e se ele está coerente com o pedido por sua gramática e, portanto, não é dependente das especificações da máquina. Já a segunda é o momento de geração do código binário para o *hardware* em específico, em que leva em consideração em qual máquina o código será executado. Nota-se a relação entre as duas fases, já que sem a primeira não seria possível linearizar o código e transformá-lo em um código legível pelo processador.

Nesse viés, a disciplina de Laboratório de Sistemas Computacionais: Compiladores tem como objetivo principal trazer para o aluno, com a implementação de um compilador com sua fase de análise e síntese, a importância do seu uso e suas motivações de criação. Cabe ressaltar que o compilador deve ser desenvolvido para gerar o código fonte de uma unidade de processamento desenvolvida anteriormente em outras disciplinas, ressaltando a interconexão entre o desenvolvimento de *hardware* e *software*.

## 2 Processador MIPS

O processador MIPS (Microprocessador Sem Estágios Intertravados de Pipeline) é uma família de processadores criada pela empresa MIPS Computer System, em 1985. Seu modelo de arquitetura é um ótimo para ser utilizado como estudo, já que possui grande popularidade e por ser de fácil entendimento para quem está iniciando os estudos sobre arquitetura e organização de computadores (PATTERSON; HENNESSY, 2006). Dessa maneira, o projeto do processador desenvolvido na disciplina anterior do curso foi idealizado com as mesmas arquiteturas e organização do MIPS e, por conta disso, será descrito informações relevantes sobre a sua implementação, que poderão ser úteis para sua explicação.

### 2.1 Arquitetura Básica

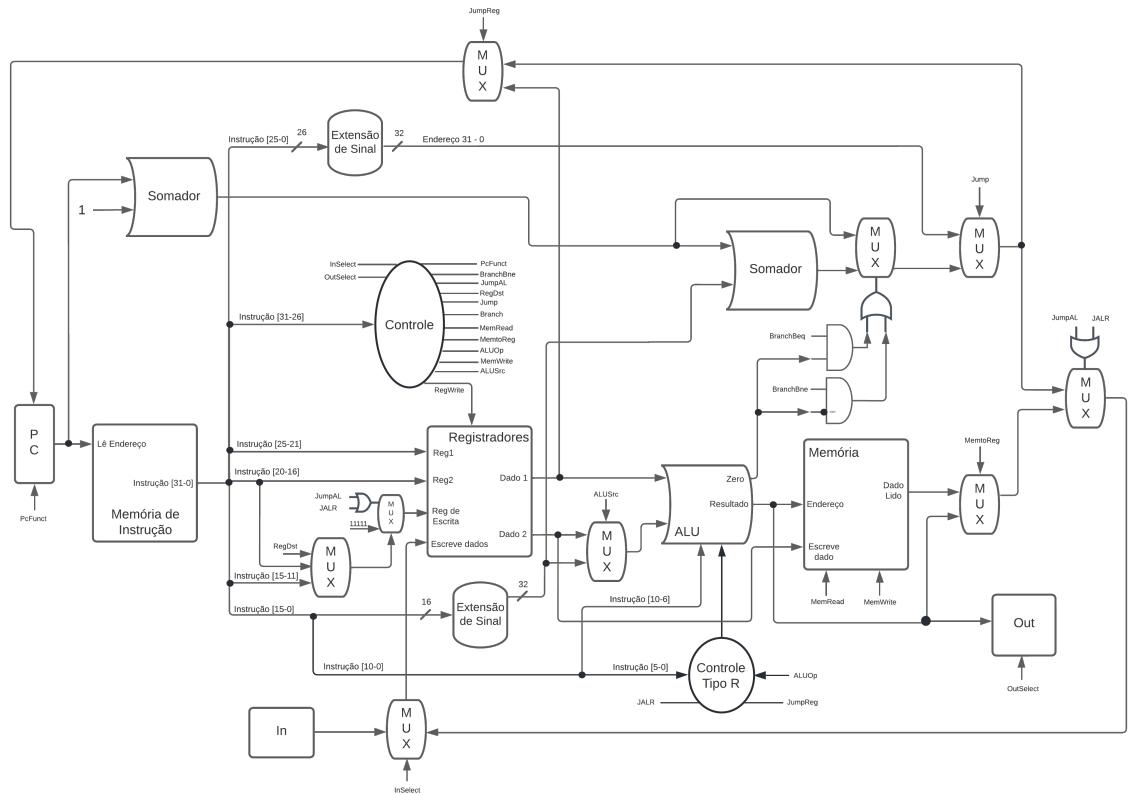
Um processador MIPS possuí estruturas indispensáveis para o seu correto funcionamento. As mais importantes são o registrador PC, o banco de registradores, a memória de instrução, memória principal e a unidade lógica e aritmética (PATTERSON, 2017). Além disso, na arquitetura MIPS existem duas memórias separadas, ou seja, é implementada uma arquitetura de Harvard. Como as figuras do caminho de dados em livros didáticos não são capazes de ilustrar todas as instruções a serem desenvolvidas no projeto, um novo caminho de dados foi desenvolvido, que está ilustrado na [Figura 1](#).

Nesse diagrama do processador é importante ressaltar alguns componentes que possuem maiores relevâncias para o projeto. Primeiramente, temos a memória de instrução, que sua única entrada será o valor vindo do registrador PC, indicando qual será a instrução que será enviada para os demais segmentos. Em seguida, essa saída será dividida em diversos segmentos, em que cada parte será enviada a um componente diferente. Cada segmento será um dos campos dos tipos de instruções.

O banco de registradores irá possuir cinco entradas distintas, sendo elas: o endereço do primeiro registrador de entrada (rs); o endereço do segundo registrador de entrada (rt); o endereço do registrador de destino (rd ou rt); o dado a ser escrito em um dos registradores; e um bit de controle, para ativar e desativar a escrita no banco. Já as saídas serão os valores armazenados nos registradores indicados nos endereços de entrada.

As duas saídas do banco de registradores serão enviadas diretamente para a unidade lógica e aritmética, em que servirão como valores de entrada para realizar alguma de suas operações. Ademais, o valor da segunda entrada também poderá ser o valor de um imediato de 16bits de uma instrução do tipo I. Este valor deverá ser estendido para 32bits

Figura 1 – Caminho de dados para o processador desenvolvido.



Fonte: O Autor

para que seja possível realizar sua operação na ULA. Além disso, temos mais duas outras entradas: o valor do “shamt” da instrução do tipo R e os *bits* de controle, que dirão qual instrução será realizada. Como saída, será enviada por um cabo o valor do resultado obtido pela operação. Um outro bit também será emitido caso a resposta seja zero ou não.

Por fim, temos a memória principal. Ela recebe como entrada: o resultado do ULA, que será o endereço do dado a ser lido ou armazenado; o valor do dado a ser armazenado; e dois *bits* de controle, sendo eles um para escrita e um para leitura dos dados. Como saída, ele emite o valor do dado desejado, que será armazenado no banco de registradores.

Ao realizar as instruções, o fluxo a ser realizado será sequencial, ou seja, da mesma maneira em que elas foram dispostas na memória de instruções. Porém, caso seja realizada uma instrução de pulo condicional ou incondicional, o valor do endereço obtido será encaminhado ao registrador PC e ele será atualizado. Dessa forma, é possível alterar a ordem em que as instruções serão efetuadas dentro do processador projetado.

## 2.2 Formato de Instruções

Para transmitir uma instrução para o processador, é preciso possuir uma série de sinais eletrônicos altos e baixos, que em binário são representados com os números

1 e 0, respectivamente (PATTERSON, 2017). Cada valor unitário do sistema binário é chamado de bit. O conjunto desses números formam as instruções, que serão interpretadas pelo processador. A partir disso, são selecionados campos específicos desses números para passarem informações importantes. Esses segmentos terão informações diferentes dependendo de qual instrução será realizada e o seu valor irá impactar de alguma forma o funcionamento geral do que será efetuado.

Dessa forma, o processador implementa a arquitetura RISC (*Reduced Instruction Set Computer*) de 32 bits e, consequentemente, possui o mesmo número de *bits* para todos os formatos de instruções.

O MIPS possui apenas três tipos de instruções diferentes, sendo eles: registrador (R); imediato (I); e de desvio incondicional (J). Cada um deles será segmentado de forma a armazenar informações diferentes. Na Tabela 1 é possível verificar as segmentações para o formato para o tipo R.

Tabela 1 – Formato de Instruções do tipo R

Tamanho (bits)	6	5	5	5	5	6
Campo	OPcode	Reg1 (rs)	Reg2 (rt)	Reg3 (rd)	Shamt	Funct
Bits	31 - 26	25 - 21	20 - 16	15 - 11	10 - 6	5 - 0

Fonte: O Autor

O campo “opcode” indica ao processador a operação básica da instrução, segmento que será importante para os três tipos de instruções. Sem essa informação, não é possível diferenciar as instruções uma das outras. Essa informação é encaminhada para a unidade de controle, em que irá selecionar quais funcionalidades deverão ser ativadas ou desativadas para que a instrução seja efetuada corretamente. Além disso, o campo “funct” também servirá para o propósito de controle, porém voltado apenas para a unidade lógica e aritmética, escolhendo qual operação deverá ser realizada.

Ademais, temos três campos para armazenar o endereço de um dos registradores, sendo eles o primeiro registrador de origem (rs), o segundo registrador de origem (rt) e o registrador de destino (rd). Então, ao realizar uma operação entre “rs” e “rt”, o valor final será armazenado em “rd”.

Por fim, temos o campo “shamt”, que possui o propósito de realizar instruções que envolvam deslocar valores de *bits*. Esse deslocamento pode ser realizado para a esquerda ou para a direita, dependendo de qual função será utilizada.

Tabela 2 – Formato de Instruções do tipo I

Tamanho (bits)	6	5	5	16
Campo	OPcode	Reg1 (rs)	Reg2 (rt)	Imediato
Bits	31 - 26	25 - 21	20 - 16	15 - 0

Fonte: O Autor

Na [Tabela 2](#) está ilustrado o formato de instrução para o tipo I. É possível verificar que as únicas diferenças para o tipo R seriam que os campos “rd”, “shamt” e “funct” foram alterados para comportar um imediato de 16 *bits*. Esse imediato pode possuir diversas funções, seja para suportar um endereço de memória ou um valor para realizar uma operação aritmética.

Em determinadas instruções do tipo I, será preciso acessar ou armazenar dados em registradores. Assim, “rs” será o registrador para leitura das informações e “rt” o registrador destino.

Tabela 3 – Formato de Instruções do tipo J

<b>Tamanho (bits)</b>	6	26
<b>Campo</b>	OPcode	Immediato
<b>Bits</b>	31 - 26	25 - 0

Fonte: O Autor

Por último, na [Tabela 3](#), o formato de instruções para o tipo I. Neste, os únicos campos existentes será o *opcode* e o imediato de 26 *bits*. Esse imediato será importante para armazenar o endereço de uma instrução na memória de instruções, permitindo que seja possível realizar desvios incondicionais.

## 2.3 Conjunto de Instruções

Com os formatos definidos no seção anterior, é possível descrever quais serão as instruções que serão implementadas para esse processador. Dessa forma, nas Tabelas [4](#), [5](#) e [6](#) estão representadas essas instruções, em conjunto com os seus nomes, números de identificação e qual a sua funcionalidade.

Tabela 4 – Instruções do tipo R

OPcode	Funct	Nome	Função
000000	100000	ADD	$rd \leftarrow rs + rt$
000000	100010	SUB	$rd \leftarrow rs - rt$
000000	100100	AND	$rd \leftarrow rs \& rt$
000000	100101	OR	$rd \leftarrow rs   rt$
000000	101101	XOR	$rd \leftarrow rs \oplus rt$
000000	001000	JR	$PC \leftarrow rs$
000000	001001	JALR	$ra \leftarrow \text{endereço}; PC \leftarrow rs$
000000	101010	SLT	$rd \leftarrow (rs < rt); 0 \text{ se falso; } 1 \text{ se verdadeiro}$
000000	100111	NOR	$rd \leftarrow rs \sim   rt$
000000	000000	SLL	$rd \leftarrow rt \ll \text{shamt}$
000000	000010	SRL	$rd \leftarrow rt \gg \text{shamt}$
000000	011010	DIV	$rd \leftarrow rs / rt$
000000	011000	MULT	$rd \leftarrow rs * rt$

Fonte: O Autor

Tabela 5 – Instruções do tipo I

OPcode	Nome	Função
100011	LW	$rt \leftarrow \text{Mem}[\text{base} + \text{offset}]$
101011	SW	$\text{Mem}[\text{base} + \text{offset}] \leftarrow rt$
001000	ADDI	$rt \leftarrow rs + \text{Imm}$
001001	SUBI	$rt \leftarrow rs - \text{Imm}$
001100	ANDI	$rt \leftarrow rs \& \text{Imm}$
001101	ORI	$rt \leftarrow rs   \text{Imm}$
101101	XORI	$rt \leftarrow rs \oplus \text{Imm}$
000100	BEQ	Se ( $rs == rt$ ) então branch
000101	BNE	Se ( $rs != rt$ ) então branch
001010	SLTI	$rt \leftarrow (rs < \text{Imm}); 0 \text{ se falso; } 1 \text{ se verdadeiro}$
011111	IN	$rt \leftarrow \text{Imm}$
011110	OUT	saída $\leftarrow rs$

Fonte: O Autor

Tabela 6 – Instruções do tipo J

OPcode	Nome	Função
000010	J	Jump para o endereço
000011	JAL	$ra \leftarrow PC + 1; \text{Jump para o endereço}$
111111	HALT	Parar o processador

Fonte: O Autor

## 2.4 Modos de Endereçamento

Os meios de acesso aos operandos das instruções são chamados de modos de endereçamento, que são salvos na memória do processador, seja na memória principal, de instrução ou o banco de registradores (PATTERSON; HENNESSY, 2006). Em uma arquitetura MIPS, existem cinco modos de endereçamentos, porém na implementação do projeto foram utilizados apenas quatro delas, que serão explicadas a seguir.

### 2.4.1 A Registrador

Neste primeiro modo de endereçamento temos que, em um dos segmentos da instrução, existirá um termo que indicará o endereço de um registrador no banco de registradores e, assim, podemos obter o operando armazenado dentro dele. É possível observar que na instrução do tipo R (Tabela 1) será passado o endereço de três registradores (rs, rt e rd). Já nas instruções do tipo I (Tabela 2) temos apenas dois endereços (rs e rt).

### 2.4.2 Imediato

Neste caso, teremos que o endereço de um certo operando na memória principal será referenciado por um valor imediato, com um valor máximo de 16bits. O único tipo de

instrução a utilizar esse modo será a do tipo I ([Tabela 2](#)), que possui um campo justamente para esse valor imediato.

### 2.4.3 Base-Deslocamento

Para esse tipo, o valor do endereço a ser escolhido na memória principal será relativo a um vetor. Ou seja, utilizamos um valor como base, que será o inicio do vetor, e depois deslocamos uma quantidade de *bits* para encontrar a posição desejada nesta lista. É um modo de endereçamento utilizado por instruções de *load* e *store*, que são do tipo I ([Tabela 2](#)), em que o valor armazenado em “rs” será o valor base e o imediato o valor do deslocamento.

### 2.4.4 Absoluto

Para finalizar, o último modo, da mesma forma que o relativo ao PC, também influência apenas na memória de instrução. Neste caso, o valor do endereço é totalmente passado pelo imediato de 26 *bits*. Dizemos que ele é absoluto por ser o maior valor possível que podemos passar por meio de uma instrução, já que os demais 6 *bits* são destinados ao opcode. Dessa forma, para indicar isso, ele também pode ser chamado de pseudo-absoluto. O único tipo de instrução capaz de realizar esse modo de endereçamento é a do tipo J ([Tabela 3](#)).

# 3 Compilador para a linguagem C-

Como qualquer outro software, um compilador de uma nova linguagem deve, preferencialmente, ser escrito em uma linguagem de programação que já possua um compilador ou interpretador implementado, para que assim seja possível obter todos os benefícios de se trabalhar com uma linguagem de alto nível. Nesse projeto, a linguagem de programação escolhido foi a C.

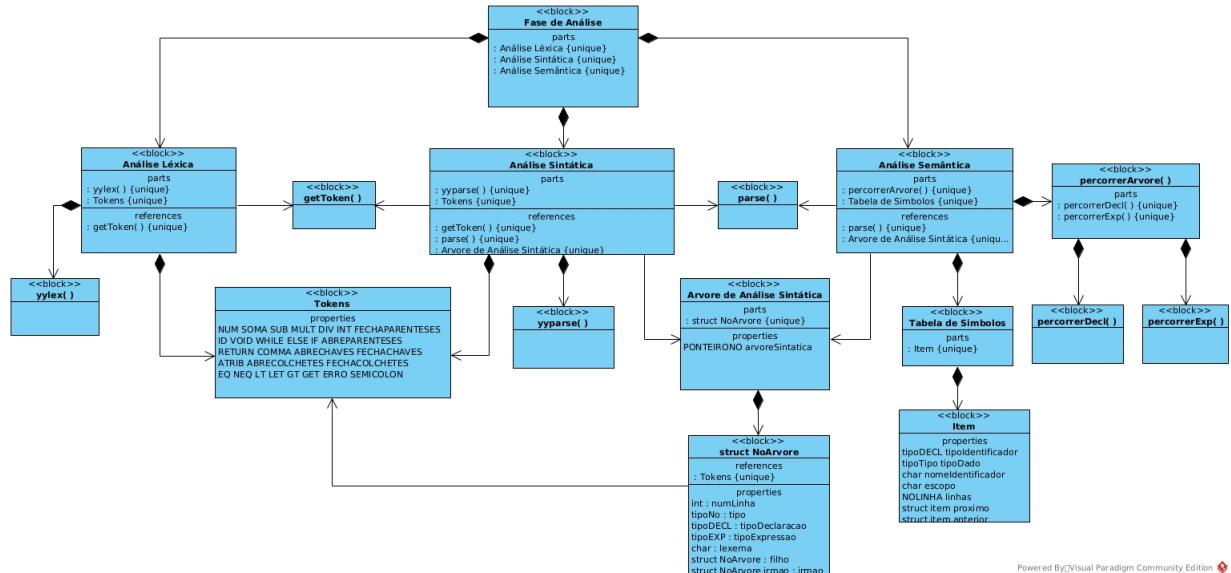
## 3.1 Fase de Análise

A fase de análise é o momento em que o compilador irá analisar se o que foi escrito pelo programador está coerente com as regras propostas pela linguagem. Caso algum erro tenha sido cometido, o compilador deve avisá-lo e parar seus procedimentos, sem continuar para a fase de síntese, para que assim o programador possa arrumá-los.

### 3.1.1 Modelagem

#### 3.1.1.1 Diagrama de Blocos

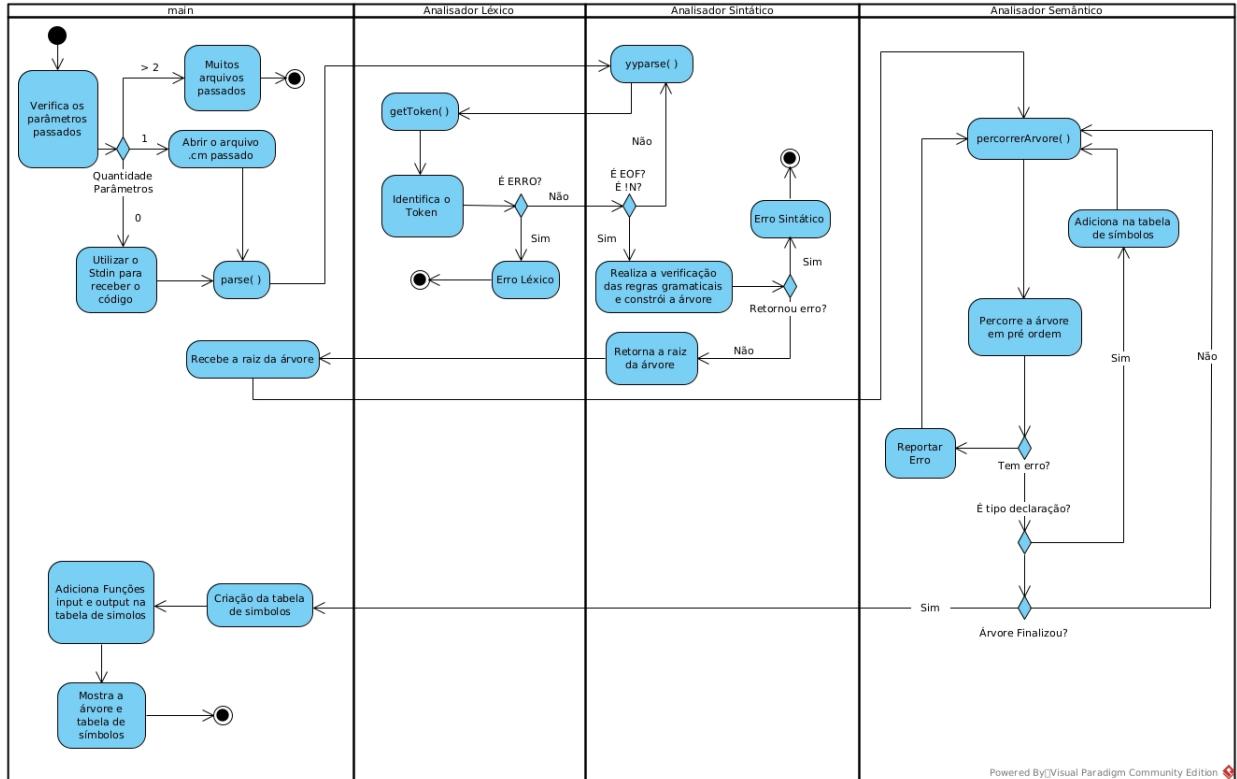
Figura 2 – Diagrama de blocos para a fase de análise



Fonte: O Autor

### 3.1.1.2 Diagrama de Atividades

Figura 3 – Diagrama de atividades para a fase de análise



Fonte: O Autor

### 3.1.2 Análise Léxica

A análise léxica é a primeira verificação que todo compilador irá realizar. Sua importância é encontrar os “tokens” da linguagem, ou seja, se as palavras colocadas no código são de alguma forma interpretadas pelo compilador e, em caso afirmativo, agrupá-las em seu significado geral. Caso um símbolo ou palavra não for reconhecido pelo analisador, um erro léxico deverá ocorrer.

Os símbolos com os seus respectivos tokens podem ser observados na [Tabela 7](#).

Tabela 7 – Relação entre símbolos e tokens para o compilador

Simbolo	Token
Números (0-9)	NUM
Nomes de Variáveis/Funções	ID
void	VOID
if	IF
int	INT
else	ELSE
return	RETURN
while	WHILE
(	ABREPARENTESES
)	FECHAPARENTESES
[	ABRECOLCHETES
]	FECHACOLCHETES
{	ABRECHAVES
}	FECHACHAVES
=	ATRIB
,	COMMA
;	SEMICOLON
+	SOMA
-	SUB
*	MULT
==	EQ
!=	NEQ
<	LT
>	GT
<=	LET
>=	GET
Diferente dos símbolos acima	ERRO

Fonte: O Autor

### 3.1.3 Análise Sintática

O analisador sintático tem a função de verificar se o que foi escrito pelo programador está coeso, ou seja, se os tokens estão em uma ordem possível de se ocorrer. Essa verificação é importante por manter regras que permitem o compilador linearizar o código escrito em um código sequencial, que será facilmente entendido pelo processador. Caso um token encontrado não esteja definido para estar presentes naquele momento, um erro sintático deverá ser apresentado para o usuário.

As regras da análise sintática são definidas por gramáticas livre de contexto (CFG). Cada linguagem de programação possui a sua gramática definida a priori pelos desenvolvedores do compilador. Dessa forma, na [Tabela 8](#) é possível verificar a gramática definida para linguagem C-.

Ao verificar as regras da gramática, o compilador realiza a geração de uma estrutura muito relevantes para as fases seguintes do projeto, chamada de árvore de análise sintática. Ela é uma estrutura do tipo árvore que cada nó pode possuir um irmão e no máximo três filhos, dessa forma cada um deverá representar alguma informação relevante sobre o que o

Tabela 8 – Sintaxe da linguagem C- utilizada no projeto

programa	declaracao_lista
declaracao_lista	declaracao_lista declaracao
declaracao	declaracao
var_declaracao	var_declaracao fun_declaracao
var_declaracao	tipo_especificador ID SEMICOLON tipo_especificador ID ABRECOLCHETES NUM FECHACOLCHETES SEMICOLON
tipo_especificador	INT VOID
fun_declaracao	tipo_especificador fun_id ABREPARENTES params FECHAPARENTES composto_decl
fun_id	ID
params	param_lista VOID
param_lista	param_lista COMMA param param
param	tipo_especificador ID tipo_especificador ID ABRECOLCHETES FECHACOLCHETES
composto_decl	ABRECHAVES local_declaracoes statement_lista FECHACHAVES
local_declaracoes	local_declaracoes var_declaracao   %empty
statement_lista	statement_lista statement   %empty
statement	expressao_decl composto_decl selecao_decl iteracao_decl retorno_decl
expressao_decl	expressao SEMICOLON SEMICOLON
selecao_decl	IF ABREPARENTES expressao FECHAPARENTES statement fatoracao
fatoracao	ELSE statement   %empty
iteracao_decl	WHILE ABREPARENTES expressao FECHAPARENTES statement
retorno_decl	RETURN SEMICOLON RETURN expressao SEMICOLON
expressao	var ATRIB expressao simples_expressao
var	ID ID ABRECOLCHETES expressao FECHACOLCHETES
simples_expressao	soma_expressao_relacional soma_expressao soma_expressao
relacional	operador_relacional
operador_relacional	EQ   NEQ   LT   GT   LET   GET
soma_expressao	soma_expressao soma termo
soma	SOMA   SUB
termo	termo mult fator fator
mult	MULT   DIV
fator	ABREPARENTES expressao FECHAPARENTES var ativacao NUM
ativacao	fun_id ABREPARENTES args FECHAPARENTES
args	arg_lista   %empty
arg_lista	arg_lista COMMA expressao expressao

Fonte: O Autor

programador está fazendo naquele momento, como por exemplo: declaração de variáveis e funções, operações aritméticas, utilização das variáveis e demais outras. Sem essa estrutura, não seria possível realizar a análise seguinte e a geração do código intermediário, já que as duas precisam percorrer-la para extrair as informações necessárias.

### 3.1.4 Análise Semântica

Caso o código não possua erros após as duas verificações anteriores, então concluímos que ele está escrito corretamente e padronizado nas regras da linguagem, ou seja, não viola sua sintaxe. Porém, ainda assim é possível que essas expressões não façam sentido ou estejam logicamente incorretas, o que poderia ocasionar em um comportamento anormal do código fonte, caso eles não fossem alterados. Esses erros dependem do projetista da linguagem, já que eles podem estar presentes em uma e não em outra.

Na linguagem de programação C- os erros semânticos são semelhantes aos da linguagem original C. Na [Tabela 9](#) estão presentes os erros semânticos que foram tratados no projeto deste compilador e a explicação do que ele indica.

Tabela 9 – Erros semânticos presentes na linguagem C-

Erro	Explicação
Declaração variável como void	Não é possível declarar uma variável do tipo void, já que seria de um tipo inexistente e não conseguiria ser usado pelo processador.
Declaração de função duplicada	Quando duas funções são nomeadas com o mesmo ID, impedindo o compilador de identificar qual a certa a ser chamada.
Declaração de Variável duplicada	Quando duas variáveis, no mesmo escopo, estão definidas com o mesmo ID.
Declaração de função com nome de variável	Quando uma função é declarada, mas uma variável já possuía esse mesmo ID anteriormente.
Declaração de variável com nome de função	Quando uma variável é declarada, mas uma função já possuía esse mesmo ID anteriormente.
Variável não declarada	Quando uma variável é utilizada, mas ela não foi declarada. Isso impede o compilador de alocar um espaço na memória para aquela variável em questão.
Função não declarada	Ocorre quando uma função é utilizada sem ser declarada, impedindo o compilador de realizar o desvio do código fonte, já que ela não existe.
Atribuições de funções do tipo Void	Como funções declaradas com o tipo Void não retornam valores, não é possível realizar atribuições a variáveis com ela.
Função “main” não declarada	Caso essa função não exista, não existe um procedimento inicial para iniciar o código fonte.
Vetor não declarado	Quando uma variável é utilizada como vetor, mas ela não foi declarada.
Chamada de função não realizada	Quando o ID de uma função é escrito, porém não foi utilizado () no final do mesmo.

Fonte: O Autor

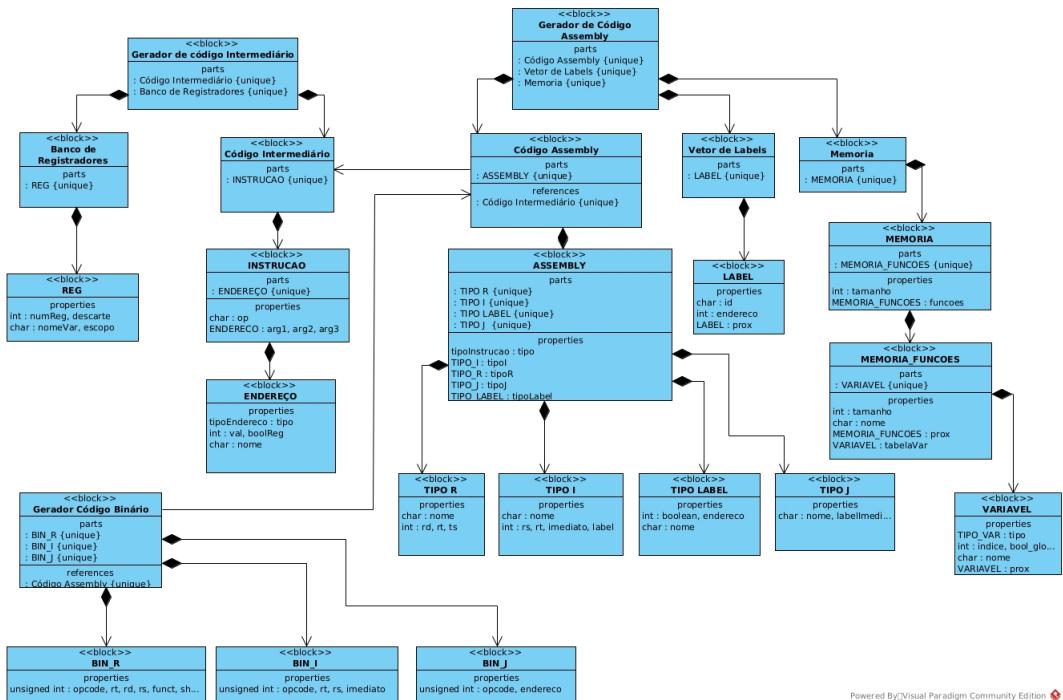
## 3.2 Fase de Síntese

Nesse momento, o compilador já realizou todas as verificações cabíveis e está pronto a iniciar a geração do código fonte. Caso algum erro tenha sido levantado na fase anterior, o compilador irá concluir sua execução, já que não faz sentido a geração de um código com erros envolvidos e cabe ao programador corrigi-los e realizar uma nova instância do programa. Essa fase é dividida em três partes: código intermediário, código *assembly* e código binário, que serão descritas nas próximas seções.

### 3.2.1 Modelagem

#### 3.2.1.1 Diagrama de Blocos

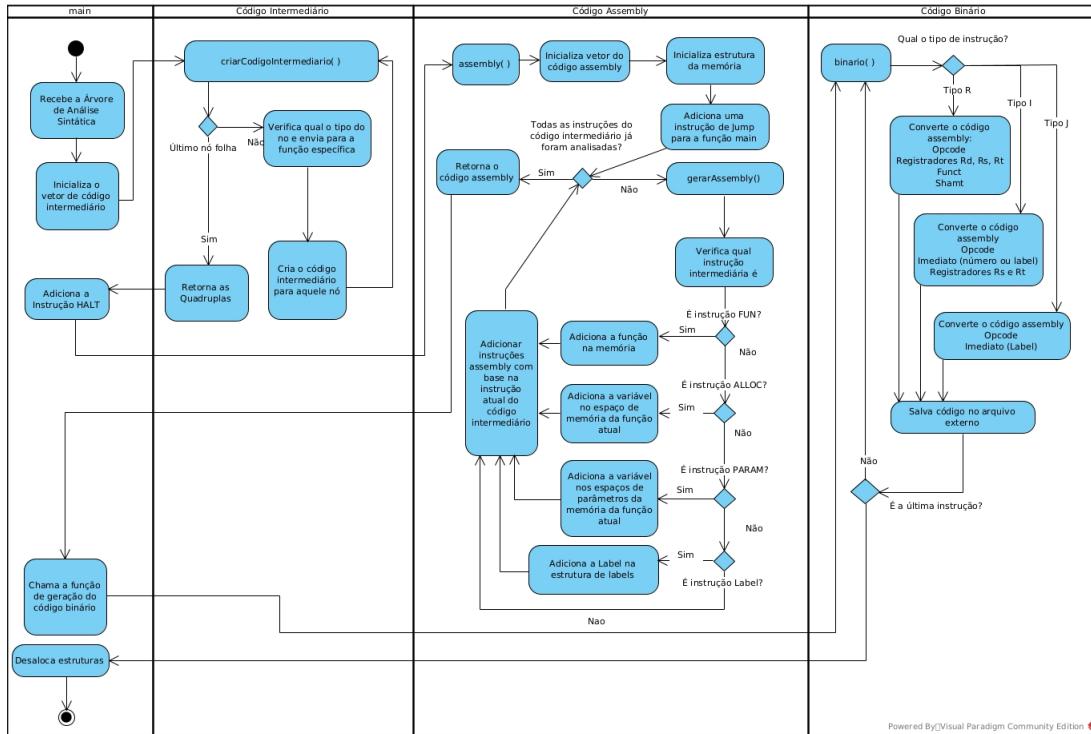
Figura 4 – Diagrama de blocos para a fase de síntese



Fonte: O Autor

### 3.2.1.2 Diagrama de Atividades

Figura 5 – Diagrama de atividades para a fase de síntese



Fonte: O Autor

### 3.2.2 Código Intermediário

O código intermediário é o primeiro mecanismo após o código ter sido totalmente verificado na fase de análise. O seu principal propósito é transformar uma estrutura não linear em linear, que nesse caso, é o código escrito na linguagem C-, representado pela árvore de análise sintática, e transformada em pseudo instruções sequenciais.

Essas pseudo-instruções são generalizadas e podem representar diversos códigos de máquinas distintos. Isso é importante já que o compilador pode realizar esse procedimento até esse ponto para qualquer máquina, pois as instruções do código intermediário serão as mesmas. Dessa forma, caso o projetista do compilador queira realizar a geração do código *assembly* diretamente, sem passar por essa fase em questão, não teria problema para aquela máquina em questão, porém seria mais complexo realizar a portabilidade daquele programa para demais máquinas.

Como visto anteriormente, a árvore de análise sintática é extremamente importante, pois é com ela que toda a estrutura do código fonte foi armazenado para indicar o que aquele código deve realizar. Assim, o gerador de código intermediário deve percorrer os nós da árvore e adicionar as instruções que realizam a operação desejada. Essas instruções foram descritas na Tabela 10.

Tabela 10 – Instruções do código intermediário

Pseudo-Instrução	Explicação
IFF	Uma instrução para verificar se o valor passado por ele é igual a 0. Em caso afirmativo, um desvio deve ser realizado para a label indicada (desvio condicional).
LABEL	Indica que deve existir uma label naquele ponto do código. Elas são utilizadas para marcações de pontos de interesse, em que o código pode realizar saltos condicionais ou incondicionais. Exemplos: nome de funções e finalização de uma função.
GOTO	Realiza um desvio incondicional para a label indicada.
FUN	Indica o início de uma função.
END	Indica a finalização de uma função.
ARG	Indica quais são os argumentos que devem ser passados para essa função.
LOAD	Acessa a memória de dados para trazer o valor armazenado de uma variável para o registrador indicado (memória para registrador).
ALLOC	Realiza a alocação de uma nova variável no escopo daquela função (ou global).
RET	Realiza o retorno para a função acima na pilha, ou seja, a função anterior.
ADD - SUB - MULT - DIV	Realiza as operações aritméticas de soma, subtração, multiplicação e divisão, respectivamente.
LOADI	Salva o valor de um imediato inteiro dentro do registrador especificado.
EQ - NEQ - GT - LT - GET - LET	Realiza operações relacionais entre dois registradores, retornando verdadeiro (1) ou falso (0). Elas são: igualdade, não igualdade, maior que, menor que, maior ou igual que e menor ou igual que, respectivamente.
CALL	Realiza a chamada para a função indicada.
PARAM	Utilizado para passar parâmetros como argumentos de funções. Esses parâmetros devem ser enviados antes de realizar uma instrução de CALL.
ASSIGN	Realiza uma atribuição de valor para uma variável (registrador para registrador).
STORE	Armazena o dado da variável na memória de dados (registrador para a memória).

Fonte: O Autor

Primeiramente, é necessário a construção de uma estrutura de dados que consiga armazenar essas instruções e, dessa forma, as duas *structs*, representadas na Figura 6 foram criadas para tal. A *struct* “INSTRUCAO” armazena o nome da operação a ser realizada em conjunto com os seus três argumentos, já que é valor máximo de argumentos que um código intermediário possui. Esses argumentos são armazenados em *structs* chamadas “ENDERECO” utilizada, em que cada um pode ser do tipo *string*, inteiro ou vazia (caso não possua valor). Por fim, todas as instruções são armazenadas no vetor “codigoIntermediario”.

Assim, basta percorrer os nós da árvore e analisando o seus tipos, já que cada um corresponderá a uma instrução distinta. Além disso, os nós filhos também podem conter informações relevantes para aquela instrução, então isso deve ser levado em consideração ao ser criado o código intermediário. A Figura 7 ilustra o código principal para a geração desse

Figura 6 – Estrutura de dados para o código intermediário, presente no arquivo “codIterm.h”

```

16 //ENUM com os tipos de enderecos
17 typedef enum {Vazio, IntConst, String} tipoEndereco;
18
19 /* Estrutura de enderecos, em que armazena o seu tipo e o
20 valor determinado pelo mesmo, ou seja, numero ou char*/
21 typedef struct endereco{
22     tipoEndereco tipo;
23     int val;
24     int boolReg; //Diz se o endereco eh um numero (0), Reg (1) ou label (2)
25     char* nome;
26 } ENDERECO;
27
28 /* Estrutura para as intrucoes, armazenado em quadruplas */
29 typedef struct instrucao{
30     char* op; //Operador
31     ENDERECO* arg1;
32     ENDERECO* arg2;
33     ENDERECO* arg3;
34 } INSTRUCAO;
35
36 //Vetor que armazena o codigo intermediario
37 extern INSTRUCAO** codigoIntermediario;

```

Fonte: O Autor

código, em que cada tipo irá possuir o seu próprio procedimento. Note que essa função é recursiva e, dessa forma, um nó pode chamar novamente a função “criarCodigoIntermediario” para percorrer os seus nós filhos.

Figura 7 – Função para gerar o código intermediário, com base nos possíveis nós da árvore sintática, no “codIterm.c”

```

635 void criarCodigoIntermediario(PONTEIRONO arvoreSintatica, PONTEIROITEM tabelaHash[], int boolean){
636     if(arvoreSintatica == NULL){
637         return;
638     }
639
640     if(arvoreSintatica->tipo == DECLARACAO){
641         if(arvoreSintatica->tipoDeclaracao == FunDecK){
642             codIntDeclFunc(arvoreSintatica, tabelaHash);
643         }
644         else if(arvoreSintatica->tipoDeclaracao == VarDeclK || arvoreSintatica->tipoDeclaracao == VetDeclK){
645             codIntDeclVarDecl(arvoreSintatica, tabelaHash);
646         }
647         else if(arvoreSintatica->tipoDeclaracao == IfK){
648             codIntDeclIf(arvoreSintatica, tabelaHash);
649         }
650         else if(arvoreSintatica->tipoDeclaracao == WhileK){
651             codIntDeclWhile(arvoreSintatica, tabelaHash);
652         }
653         else if(arvoreSintatica->tipoDeclaracao == ReturnINT || arvoreSintatica->tipoDeclaracao == ReturnVOID){
654             codIntDeclReturn(arvoreSintatica, tabelaHash);
655         }
656     }
657     else if (arvoreSintatica->tipo == EXPRESSAO){
658         if(arvoreSintatica->tipoExpressao == OpK){
659             codIntExpOp(arvoreSintatica, tabelaHash);
660         }
661         else if(arvoreSintatica->tipoExpressao == ConstK){
662             codIntExpConst(arvoreSintatica, tabelaHash);
663         }
664         else if(arvoreSintatica->tipoExpressao == OpRel){
665             codIntExpOpRel(arvoreSintatica, tabelaHash);
666         }
667         else if(arvoreSintatica->tipoExpressao == IdK || arvoreSintatica->tipoExpressao == VtorK){
668             codIntExpId(arvoreSintatica, tabelaHash);
669         }
670         else if(arvoreSintatica->tipoExpressao == AtivK){
671             codIntExpCall(arvoreSintatica, tabelaHash);
672         }
673         else if(arvoreSintatica->tipoExpressao == AssignK){
674             codIntExpAtrib(arvoreSintatica, tabelaHash);
675         }
676     }
677
678     if(boolean == 1){
679         criarCodigoIntermediario(arvoreSintatica->irmao, tabelaHash, 1);
680     }
681 }

```

Fonte: O Autor

Para exemplificar esse procedimento, será descrito uma das funções utilizadas para a geração do código para exemplificar como as estruturas são criadas e o funcionamento da recursividade da passagem pelos nós. Na [Figura 8](#) está ilustrado a função que cria

o código intermediário para operadores aritméticos (soma, subtração, multiplicação e divisão) e, dessa forma, sempre que uma dessas operações seja encontrada na árvore de análise sintática, a função representada deve tratá-las. Dessa forma, a função primeiro verifica qual operação aritmética é, para poder atribuir o nome para a operação, com base na [Tabela 10](#).

Figura 8 – Função para gerar o código intermediário ao encontrar um operador aritmético no “codIterm.c”

```

351     void codIntExpOp(PONTEIROONO arvoreSintatica, PONTEIROITEM tabelaHash[]){
352         INSTRUCAO* op = NULL;
353         char NomeOp[MAXLEXEMA];
354
355         strcpy(NomeOp, arvoreSintatica->lexema);
356
357         if(strcmp(NomeOp, "+") == 0){
358             op = criaInstrucao("ADD");
359         }
360         else if(strcmp(NomeOp, "-") == 0){
361             op = criaInstrucao("SUB");
362         }
363         else if(strcmp(NomeOp, "*") == 0){
364             op = criaInstrucao("MULT");
365         }
366         else if(strcmp(NomeOp, "/") == 0){
367             op = criaInstrucao("DIV");
368         }
369
370         criarCodigoIntermediario(arvoreSintatica->filho[0], tabelaHash, 1);
371         op->arg1 = criaEndereco(IntConst, numReg, NULL, 1);
372
373         criarCodigoIntermediario(arvoreSintatica->filho[1], tabelaHash, 1);
374         op->arg2 = criaEndereco(IntConst, numReg, NULL, 1);
375
376         numReg = verificaRegistradores(NULL, NULL, 1);
377
378         op->arg3 = criaEndereco(IntConst, numReg, NULL, 1);
379
380         codigoIntermediario[indiceVetor] = op;
381         indiceVetor++;
382
383     }

```

Fonte: O Autor

A função “criaInstrucao” retorna uma *struct* INSTRUCAO com todos os campos de argumentos vazios, permitindo que o programador decida posteriormente o que deve conter nesses espaços. Com isso, para o primeiro e segundo argumento, dois registradores devem ser utilizados para serem os operandos. Dessa maneira, antes é realizada uma chamada recursiva nos dois nós filhos, já que mais operações podem estar encadeadas, e seu registrador com o valor resultante das operações anteriores estará presente na variável global “numReg” e, dessa forma, basta colocar esse valor como argumento, por meio da função “criaEndereco”. Por fim, é necessário escolher um registrador temporário que não esteja em uso no momento para poder armazenar o resultado da operação, por meio da instrução “verificaRegistradores”, e armazenar o seu índice no terceiro argumento da instrução. Basta adicionar a nova instrução no vetor de código intermediário e incrementar o índice da próxima instrução.

### 3.2.3 Código Assembly

Diferentemente do código intermediário, a geração do código *assembly* deve ser feito considerando as especificações da máquina alvo, já que cada processador possui organizações e arquiteturas distintas que podem impactar em seu uso, como quantidade e tipos de instruções, números de registradores e quais possuem uso específico, especificações do acesso a memória e entre demais outras informações que devem ser levadas em consideração pelo programador do projeto. Nesse caso, a arquitetura e os tipos de instruções já foram descritas na seção anterior “Processador MIPS”.

A geração é feita analisando cada instrução do código intermediário, traduzindo a sua funcionalidade para o código assembly da máquina, utilizando os três parâmetros como informações relevantes do que fazer a seguir. Cada código intermediário pode possuir uma ou mais instruções em código assembly para atingir o objetivo e funcionamento desejado (1 para N).

A estrutura desse código pode ser visualizada na [Figura 9](#), em que existem cinco *structs* distintas, sendo três para cada tipo de instrução do código do processador (tipo R, I e J, respectivamente nas [Tabelas 1, 2, 3](#)), outra para armazenar apenas valores de *label* e, por fim, uma que possui uma instância de cada uma dentro dela, para poder ser criado uma lista única de código *assembly* com os quatro tipos distintos descritos acima.

Figura 9 – Estrutura para armazenar o código assembly gerado, presente no arquivo “assembly.h”

```

11  typedef enum{
12      tipoR, // Instrucoes do tipo R
13      tipoI, // Instrucoes do tipo I
14      tipoJ, // Instrucoes do tipo J
15      tipoLabel // Label de funcoes ou de pulsos
16  } tipoInstrucao;
17
18  /* Struct para armazenar as informacoes
19  de instrucoes do tipo R */
20  typedef struct tipoR{
21      char * nome; // Nome da instrucao
22      int rd; // Registrador destino
23      int rt; // Registrador fonte
24      int rs; // Registrador fonte
25  } TIPO_R;
26
27  /* Struct para armazenar as informacoes
28  de instrucoes tipo I */
29  typedef struct tipoI{
30      char * nome; // Nome da instrucao
31      int rd; // Registrador destino
32      int rt; // Registrador destino
33      int imediato; // Valor imediato
34      int label; // Label para o branch
35  } TIPO_I;
36
37  /* Struct para armazenar as informacoes
38  de instrucoes do tipo J */
39  typedef struct tipoJ{
40      char * nome; // Nome da instrucao
41      char * labelImediato; // Nome da Label para o jump
42  } TIPO_J;
43
44  typedef struct tipoLabel{
45      int boolean; // Booleano para verificar se eh label(1) ou funcao(0)
46      char * nome; // Nome da label
47      int endereco; // Endereco da label
48  } TIPO_LABEL;
49
50
51  typedef struct assembly{
52      tipoInstrucao tipo; // Tipo da instrucao
53      TIPO_I * tipoI; // Ponteiro para a struct do tipo I
54      TIPO_R * tipoR; // Ponteiro para a struct do tipo R
55      TIPO_J * tipoJ; // Ponteiro para a struct do tipo J
56      TIPO_LABEL * tipoLabel; // Ponteiro para a struct do tipo Label
57  } ASSEMBLY;
58

```

Fonte: O Autor

Ademais, o processador possui internamente 32 registradores, porém alguns deles são reservados para uso específico do compilador. Eles estão descritos na [Tabela 11](#) em conjunto com o seu nome e sua motivação de uso.

Tabela 11 – Valores dos registradores reservados e de uso geral do programa e do compilador

Registradores	Número	Nome	Uso
\$zero	31	Zero	Armazena sempre o valor 0
\$ra	30	Return Adress	Armazena o endereço de retorno após uma chamada
\$fp	29	Frame Pointer	Armazena o índice do início da memória de dados daquela função em específico
\$sp	28	Stack Pointer	Armazena o índice do fim da memória de dados daquela função em específico
\$temp	27	Temporário	Registrador temporário para uso do compilador
\$pilha	26	Ponteiro para Pilha de parâmetros	Ponteiro para a região da pilha de parâmetros
\$t0 - \$t25	0 - 25	Registradores Diversos	Registradores para uso das operações do programa

Fonte: O Autor

Além disso, foi preciso criar uma estrutura capaz de gerenciar a memória da pilha de funções, já que será nela que dados importantes serão armazenados para aquele escopo em específico. Essa estrutura está ilustrada na [Figura 11](#). Cada função terá a possibilidade de armazenar até 25 valores inteiros de 32 bits dentro da memória, com alguns índices já pré selecionados para armazenar informações importantes para o compilador. A [Tabela 12](#) representa um exemplo para uma função genérica.

Tabela 12 – Tabela para a memória da pilha de funções

Ponteiros	Índice	Nome
\$fp	0 até M	Parâmetros
	M + 1	Vínculo de Controle
	M + 2	Endereço Retorno
	M + 3	Valor de retorno
	M + 4	Registrador Temporário
	M + 5	Registrador \$fp
	M + 6	Registrador \$sp
\$sp	M + 7 até 24	Variáveis estáticas

Fonte: O Autor

Primeiro, a memória precisa armazenar o seu “vínculo de controle” com a função anterior que a chamou, para que seja possível alterar algum valor, se necessário. O “endereço de retorno” é o valor do índice da memória de instruções em que a função atual foi chamada, assim é possível realizar um salto incondicional de volta para aquele procedimento anterior. O “valor de retorno” é o local em que as funções chamadas pelo procedimento atual armazenam os seus valores retornados. O “registrador temporário” é um valor inteiro reservado para uso do compilador, se necessário. Os índices dos “registradores \$fp e \$sp” armazenam os seus valores para caso ocorra uma chamada de uma outra função,

permitindo salvar o contexto daquela função. Por fim, os valores de “parâmetro” são o local em que as variáveis passadas como parâmetro são armazenadas e “variáveis estáticas” as demais variáveis alocadas pela própria função. Note que nem todos os espaços da memória foram utilizados durante a implementação do código *assembly*, porém os seus espaços serão mantidos para caso eles sejam precisos em momentos futuros.

Os registradores \$fp e \$sp devem apontar, respectivamente, para o início do quadro da função e para o final. Como chamadas de outras funções ocorrem, o compilador e o processador precisam reconhecer em que área da memória as variáveis estão presentes. Essa memória é fundamental para ser possível encontrar a relação entre esses registradores para as demais variáveis presentes nela, permitindo que o compilador saiba como acessá-las em instruções de acesso a memória. Na [Figura 10](#) é possível visualizar a relação entre os dois ponteiros para cada variável presente, onde elas serão armazenadas e quais os seus tipos.

Figura 10 – Saída do terminal para a memória de uma função genérica, para um programa compilado

```
=====
gcd: 8
=====
$fp -> 0: u [$fp + 0] [$sp - 24] : inteiroArg local
      1: v [$fp + 1] [$sp - 23] : inteiroArg local
      2: Vinculo Controle [$fp + 2] [$sp - 22] : controle local
      3: Endereco Retorno [$fp + 3] [$sp - 21] : retorno local
      4: Valor Retorno [$fp + 4] [$sp - 20] : retorno local
      5: Registrador Temporario [$fp + 5] [$sp - 19] : inteiro local
      6: Registrador $fp [$fp + 6] [$sp - 18] : inteiro local
      7: Registrador $sp [$fp + 7] [$sp - 17] : inteiro local
$sp -> 24:
```

Fonte: O Autor

Figura 11 – Estrutura de dados para armazenar e gerenciar a memória da pilha de funções, presente no arquivo “memoria.h”

```
6  typedef enum tipo{
7    |  inteiro,
8    |  vetor,
9    |  inteiroArg,
10   |  vetorArg,
11   |  controle,
12   |  retorno,
13   |  temp
14 } TIPO_VAR;
15
16 typedef struct variavel{
17   TIPO_VAR tipo;
18   int indice;
19   int bool_global;
20   char *nome;
21   struct variavel *prox;
22 } VARIABEL;
23
24 typedef struct memoria_funcoes{
25   int tamanho;
26   char* nome;
27   struct memoria_funcoes *prox;
28   VARIABEL *tabelaVar;
29 } MEMORIA_FUNCOES;
30
31 typedef struct{
32   int tamanho;
33   MEMORIA_FUNCOES *funcoes;
34 } MEMORIA;
```

Fonte: O Autor

Outra estrutura que será importante para a geração do código binário é as *labels* e os seus endereços de onde estão no código *assembly*, como ilustrado na Figura 12. A geração dessa estrutura facilita a busca dos valores dos endereços, já que não é mais preciso percorrer todo o vetor de código *assembly* para encontrar onde ela está localizada.

Figura 12 – Estrutura de dados para armazenar e gerenciar as *labels* presentes no código *assembly* “label.h”

```

4  typedef struct label{
5      char* id;
6      int endereco;
7      struct label *prox;
8  } LABEL;
9
10 typedef struct vetorLabel{
11     LABEL *vetor;
12     int tamanho;
13 } VETOR_LABEL;
14
15 extern VETOR_LABEL *vetorLabel;
--
```

Fonte: O Autor

Antes de iniciar a geração do código *assembly* é necessário inicializar a memória com duas pilhas iniciais: 1) “variáveis globais”, em que todas as variáveis que não estiverem dentro de um escopo de uma função serão adicionadas nessa partição da memória, e 2) parâmetros, já que uma pilha separada na memória deverá ser criada para armazenar as passagens de parâmetros antes da realização de uma chamada de função.

Após a inicialização do vetor do código, a primeira instrução a ser adicionada deverá ser um jump para a função “main”, já que a execução do código deve iniciar por ela. Assim, o método “assembly”, ilustrado na Figura 13, irá percorrer todas as instruções do código intermediário e para cada uma terá uma conversão para o código “assembly”, que será realizada na função “geraAssembly”.

Figura 13 – Função “assembly” utilizada para inicializar a geração do código *assembly* para o processador MIPS em “assembly.c”

```

14 void geraAssembly(INSTRUCAO* instrucao);
15
16 void assembly(){
17     inicializaAssembly();
18
19     /* Criar uma função Jump para a main */
20     ASSEMBLY *novaInstrucao = criarNoAssembly(typeJ, "j");
21     novaInstrucao->tipoJ->labelImediato = strdup("main");
22     instrucoesAssembly[indiceAssembly++] = novaInstrucao;
23
24     for(int i = 0; i < indiceVetor; i++){
25         geraAssembly(codigoIntermediario[i]);
26     }
27 }
28
```

Fonte: O Autor

A função “geraAssembly” deve filtrar qual a operação atual daquele código in-

termiário, para que assim ele possa converter corretamente. A Figura 14 ilustra esse processo para algumas instruções. Para criar uma nova instrução, basta utilizar a função “criarNoAssembly”, atribuindo como parâmetros os valores do tipo da instrução e o seu nome, que ela irá retornar um ponteiro para uma estrutura ASSEMBLY com o tipo dela inicializada internamente. Dessa forma, basta atribuir os demais valores diretamente a instrução a depender do que deve ser realizado. Essas informações que serão inseridas já são provenientes do código intermediário, como, por exemplo, os valores de quais registradores serão utilizados e quais são as variáveis envolvidas.

Para a maior parte das instruções de código intermediário basta adicionar apenas uma única instrução para a máquina MIPS e, portanto, são mais diretas de serem interpretadas. Instruções que demandam passos diferentes serão explicadas em suas devidas seções a seguir.

Figura 14 – Função “geraAssembly” utilizada para gerar efetivamente o código *assembly* para o processador MIPS, presente no arquivo “assembly.c”

```

162 void geraAssembly(INSTRUCAO* instrucao){
163     ASSEMBLY* novaInstrucao = NULL;
164
165     if(opAritmeticos(instrucao, &novaInstrucao)){
166         instrucoesAssembly[indiceAssembly++] = novaInstrucao;
167     }
168     else if(opRelacionais(instrucao, &novaInstrucao)){
169         instrucoesAssembly[indiceAssembly++] = novaInstrucao;
170     }
171     else if(!strcmp(instrucao->op, "ASSIGN")){
172         novaInstrucao = criarNoAssembly(typeR, "add");
173         novaInstrucao->tipoR->rd = instrucao->arg1->val;
174         novaInstrucao->tipoR->rs = $zero;
175         novaInstrucao->tipoR->rt = instrucao->arg2->val;
176         instrucoesAssembly[indiceAssembly++] = novaInstrucao;
177     }
178     else if(!strcmp(instrucao->op, "LOADI")){
179         novaInstrucao = criarNoAssembly(typeI, "ori");
180         novaInstrucao->tipoI->rt = instrucao->arg1->val;
181         novaInstrucao->tipoI->rs = $zero;
182         novaInstrucao->tipoI->imediato = instrucao->arg2->val;
183         instrucoesAssembly[indiceAssembly++] = novaInstrucao;
184     }
185     else if(!strcmp(instrucao->op, "ALLOC")){
186         MEMORIA_FUNCOES* funcao = buscar_funcao(&vetorMemoria, instrucao->arg2->nome);
187         int count = 0;
188
189         /* Continua ...*/
190 }
```

Fonte: O Autor

### 3.2.3.1 Variáveis Inteiras e Vetores

Para realizar a alocação de memória das variáveis existem duas instruções distintas de código intermediário: “ALLOC” e “ARG”. A primeira é responsável por separar um espaço na memória para variáveis estáticas declaradas internamente naquele escopo, seja da função atual ou global (Figura 15). Já a segunda é responsável por separar um espaço para as variáveis que foram adicionadas com parâmetro na chamada daquela função (Figura 16).

É possível verificar pelas duas figuras que são utilizadas duas funções: 1) “buscar\_funcao” realiza uma busca na estrutura da memória para encontrar a função desejada e 2) “insere\_variavel” para inserir a nova variável no espaço livre da função encontrada anteriormente. Para a alocação de vetores, basta adicionar a quantidade de índices passada na declaração do mesmo. Note que nenhuma instrução do código *assembly* foi gerada por enquanto, apenas as configurações internas do compilador.

Figura 15 – Verificação para a instrução ALLOC presente em “assembly.c”

```

186     else if(!strcmp(instrucao->op, "ALLOC")){
187         MEMORIA_FUNCOES* funcao = buscar_funcao(&vetorMemoria, instrucao->arg2->nome);
188         int count = 0;
189
190         if(!instrucao->arg3){
191             printf(ANSI_COLOR_RED "Erro: " ANSI_COLOR_RESET);
192             printf("NULL no argumento 3\n");
193             return;
194         }
195
196         if(instrucao->arg3->tipo == Vazio){
197             insere_variavel(funcao, instrucao->arg1->nome, inteiro);
198             count = 1;
199         }
200         else{
201             for(int i = 0; i < instrucao->arg3->val; i++){
202                 insere_variavel(funcao, instrucao->arg1->nome, vetor);
203             }
204             count = instrucao->arg3->val;
205         }
206
207         int static flag = 0;
208         if(funcao->tamanho > get_sp(funcao) && !flag){
209             printf(ANSI_COLOR_RED "Erro: " ANSI_COLOR_RESET);
210             printf("Memória insuficiente na função %s\n", funcao->nome);
211             printf("Aumente a memória alocada pelo compilador ou diminua as variáveis da função\n");
212             flag = 1;
213         }
214     }
215 }
```

Fonte: O Autor

Figura 16 – Verificação para a instrução ARG presente em “assembly.c”

```

221     else if(!strcmp(instrucao->op, "ARG")){
222         MEMORIA_FUNCOES* funcao = buscar_funcao(&vetorMemoria, instrucao->arg3->nome);
223
224         if(!strcmp(instrucao->arg1->nome, "INT"))
225             insere_variavel(funcao, instrucao->arg2->nome, inteiroArg);
226         else insere_variavel(funcao, instrucao->arg2->nome, vetorArg);
227     }
228 }
```

Fonte: O Autor

Para utilizar as variáveis presentes nessas memórias, é preciso recorrer as instruções de acesso a memória, seja de escrita ou leitura. A explicação a seguir será apenas para as instruções de “LOAD”, mas as instruções de “STORE” é um caso análogo a esse.

Para a realização de uma leitura a um valor inteiro basta utilizar o ponteiro \$fp como valor de referência e avançar o valor inteiro em relação a ele, por meio da função “get\_fp\_relation”. Por meio do exemplo da [Figura 10](#), caso queira ser acessado o valor da variável “v”, basta passar como registrador base o valor de \$fp e incrementar 1. A implementação pode ser visualizada na [Figura 18](#) no último *else*.

Figura 17 – Verificação para a instrução LOAD para vetores alocados no escopo da função, presente em “assembly.c”

```

384     else if(!strcmp(instrucao->op, "LOAD")){
385         if(!instrucao->arg3){
386             printf(ANSI_COLOR_RED "Erro: " ANSI_COLOR_RESET);
387             printf("NULL no argumento 3\n");
388             return;
389         }
390
391         VARIAVEL* var = get_variavel(funcaoAtual, instrucao->arg2->nome);
392
393         if(instrucao->arg3->tipo != Vazio){
394             // Load de um indice de um vetor
395
396             if(var->tipo == vetor){
397                 // Vetor alocado no escopo dessa função
398                 novaInstrucao = criarNoAssembly(typeI, "addi");
399                 novaInstrucao->tipoI->rt = $temp;
400                 novaInstrucao->tipoI->rs = (var->bool_global) ? $zero : $fp;
401                 novaInstrucao->tipoI->imediato = get_fp_relation(funcaoAtual, var);
402                 instrucoesAssembly[indiceAssembly++] = novaInstrucao;
403
404                 novaInstrucao = criarNoAssembly(typeR, "add");
405                 novaInstrucao->tipoR->rd = $temp;
406                 novaInstrucao->tipoR->rs = $temp;
407                 novaInstrucao->tipoR->rt = instrucao->arg3->val;
408                 instrucoesAssembly[indiceAssembly++] = novaInstrucao;
409
410                 novaInstrucao = criarNoAssembly(typeI, "lw");
411                 novaInstrucao->tipoI->rt = instrucao->arg1->val;
412                 novaInstrucao->tipoI->rs = $temp;
413                 novaInstrucao->tipoI->imediato = 0;
414                 instrucoesAssembly[indiceAssembly++] = novaInstrucao;
415             } /* Continua ... */

```

Fonte: O Autor

Figura 18 – Verificação para a instrução LOAD para vetores passados como argumento e variáveis inteiros, presente em “assembly.c”

```

415 } /* Continua ... */
416 else{
417     // Vetor passado como parametro
418     novaInstrucao = criarNoAssembly(typeI, "lw");
419     novaInstrucao->tipoI->rt = $temp;
420     novaInstrucao->tipoI->rs = (var->bool_global) ? $zero : $fp;
421     novaInstrucao->tipoI->imediato = get_fp_relation(funcaoAtual, var);
422     instrucoesAssembly[indiceAssembly++] = novaInstrucao;
423
424     novaInstrucao = criarNoAssembly(typeR, "add");
425     novaInstrucao->tipoR->rd = $temp;
426     novaInstrucao->tipoR->rs = $temp;
427     novaInstrucao->tipoR->rt = instrucao->arg3->val;
428     instrucoesAssembly[indiceAssembly++] = novaInstrucao;
429
430     novaInstrucao = criarNoAssembly(typeI, "lw");
431     novaInstrucao->tipoI->rt = instrucao->arg1->val;
432     novaInstrucao->tipoI->rs = $temp;
433     novaInstrucao->tipoI->imediato = 0;
434     instrucoesAssembly[indiceAssembly++] = novaInstrucao;
435 }
436
437 }
438 else{
439     // Load de um inteiro
440     novaInstrucao = criarNoAssembly(typeI, "lw");
441     novaInstrucao->tipoI->rt = instrucao->arg1->val;
442     novaInstrucao->tipoI->rs = (var->bool_global) ? $zero : $fp;
443     novaInstrucao->tipoI->imediato = get_fp_relation(funcaoAtual, var);
444     instrucoesAssembly[indiceAssembly++] = novaInstrucao;
445 }
446

```

Fonte: O Autor

Para o caso da variável ser um vetor, é preciso modificar a lógica utilizada para acessá-lo. Obtemos da instrução intermediário “LOAD” o valor do índice a ser acessado dentro de um registrador, contudo como o processador implementado apenas suporta utilizar como deslocamento da instrução “lw” um valor imediato, é preciso, primeiramente, realizar uma soma entre o valor desse índice a ser acessado com o valor do ponteiro \$fp. Isso é realizado na implementação na [Figura 17](#) nas instruções “addi” e “add”. Por fim, como o valor atual do registrador \$temp é o endereço exato da memória daquele índice do vetor, basta realizar um “lw” com o valor imediato igual a zero.

Dessa forma, para vetores passados como argumento existe apenas uma única modificação que é a necessidade de realizar um “lw” no ponteiro armazenado na área de parâmetros da função, que aponta diretamente para o início do vetor na outra função que o alocou. Isso está ilustrado na [Figura 18](#) no primeiro *else*.

Por fim, para o caso de variáveis globais, basta trocar o valor do ponteiro \$fp pelo registrador \$zero, já que todos as variáveis globais são armazenadas no início da memória de dados.

### 3.2.3.2 Chamada de Funções

Existem cinco instruções importantes para que seja possível realizar a chamada de uma função, sendo elas: “FUN”, “END”, “PARAM”, “CALL” e “RET”. Elas serão explicadas separadamente, em conjunto com do código gerado em *assembly*.

Para adicionar um valor como parâmetro primeiro é preciso verificar qual o tipo da variável, já que caso ela seja um vetor terá algumas peculiaridades que devem ser levadas em consideração. Ao ser encontrada uma instrução “PARAM” ([Figura 19](#)), considerando uma variável inteira, basta adicionar o seu valor na pilha de parâmetros, por meio do registrador \$pilha. Caso seja um vetor, não é o valor que será passado como argumento, mas sim o ponteiro para o início dos seus valores, dessa forma caso ele tenha sido declarado dentro do escopo da função atual (ou seja global), primeiro é preciso somar o valor do ponteiro de referência desejado (\$fp ou \$zero) com a sua relação entre eles e, assim, basta salvar esse valor na pilha. Caso esse vetor já seja um vetor argumento - passado por uma outra função - basta recuperar e salvar na pilha o valor desse ponteiro.

Figura 19 – Verificação para a instrução PARAM, presente em “assembly.c”

```

340     else if(!strcmp(instrucao->op, "PARAM")){
341         if(!strcmp(instrucao->arg2->nome, "VET")){
342             VARIAVEL* var = get_variavel(funcaoAtual, instrucao->arg3->nome);
343
344             if(var->tipo == vetorArg){
345                 novaInstrucao = criarNoAssembly(typeI, "lw");
346                 novaInstrucao->tipol->rt = instrucao->arg1->val;
347                 novaInstrucao->tipol->rs = (var->bool_global) ? $zero : $fp;
348                 novaInstrucao->tipol->imediato = get_fp_relation(funcaoAtual, var);
349                 instrucoesAssembly[indiceAssembly++] = novaInstrucao;
350
351                 novaInstrucao = criarNoAssembly(typeI, "sw");
352                 novaInstrucao->tipol->rs = $pilha;
353                 novaInstrucao->tipol->rt = instrucao->arg1->val;
354                 novaInstrucao->tipol->imediato = buscar_funcao(&vetorMemoria, "parametros")->tamanho;
355                 instrucoesAssembly[indiceAssembly++] = novaInstrucao;
356             }
357             else{
358                 novaInstrucao = criarNoAssembly(typeI, "add");
359                 novaInstrucao->tipol->rt = instrucao->arg1->val;
360                 novaInstrucao->tipol->rs = (var->bool_global) ? $zero : $fp;
361                 novaInstrucao->tipol->imediato = get_fp_relation(funcaoAtual, var);
362                 instrucoesAssembly[indiceAssembly++] = novaInstrucao;
363
364                 novaInstrucao = criarNoAssembly(typeI, "sw");
365                 novaInstrucao->tipol->rs = $pilha;
366                 novaInstrucao->tipol->rt = instrucao->arg1->val;
367                 novaInstrucao->tipol->imediato = buscar_funcao(&vetorMemoria, "parametros")->tamanho;
368                 instrucoesAssembly[indiceAssembly++] = novaInstrucao;
369             }
370         }
371     }
372 }
373 else{
374     novaInstrucao = criarNoAssembly(typeI, "sw");
375     novaInstrucao->tipol->rs = $pilha;
376     novaInstrucao->tipol->rt = instrucao->arg1->val;
377     novaInstrucao->tipol->imediato = buscar_funcao(&vetorMemoria, "parametros")->tamanho;
378     instrucoesAssembly[indiceAssembly++] = novaInstrucao;
379 }
380 ...

```

Fonte: O Autor

Ao realizar uma chamada de função ‘CALL’ primeiro é preciso realizar a retirada da pilha de parâmetros para poder armazená-los no ínicio da moldura da próxima função, ou seja, no valor do \$sp + k, em que k é o valor atual do parâmetro a ser salvo (primeiro parâmetro é k igual a 1, segundo é k igual a 2 e assim por diante).

Após realizado esse procedimento dos parâmetros, é preciso salvar o valor de controle da função atual na moldura da próxima, no campo de “vinculo de controle”. Dessa forma, a próxima função poderá acessar sempre que necessário a função que a chamou.

Como os valores de início e fim da moldura serão distintos nessa nova função, também é necessário alterar os valor de \$fp e \$sp para ficarem condizentes. Dessa forma, como as molduras são padronizadas em 25 índices, basta somar esse valor em cada um dos ponteiros.

Por fim, para ir até o inicio do código de máquina da próxima função, basta realizar um *jump and link* (jal), que é a função responsável em desviar para a *label* da nova função e salvar o endereço de retorno para a função que a chamou em um registrador específico \$ra. Esse endereço de retorno é importante já que quando a função for realizar um retorno ela precisa saber exatamente para onde ela precisa voltar para continuar com o andamento do código.

Cabe ressaltar que existem dois tipos de funções que podem ser chamadas, porém não precisam estar declaradas internamente no código de programação do usuário: “output” e “input”. Cada uma delas possuem instruções binárias específicas e com funcionalidades já programadas pelo processador em questão. Dessa forma, basta converter as chamadas

dessas funções em suas respectivas instruções *assembly* “out” e “in”, realizando a passagem dos parâmetros desejados, advindos do código intermediário.

As Figuras 20, 21 e 22 ilustram o código para a conversão do código intermediário para *assembly* para o “CALL” explicado anteriormente.

Figura 20 – Geração de código *assembly* para a instrução intermediária CALL - parte 1 - presente no arquivo “assembly.c”

```

484 |     else if(!strcmp(instrucao->op, "CALL")){
485 |         if(!instrucao->arg3){
486 |             printf(ANSI_COLOR_RED "Erro: " ANSI_COLOR_RESET);
487 |             printf("NULL no argumento 3\n");
488 |             return;
489 |         }
490 |
491 |         if(!strcmp(instrucao->arg1->nome, "output")){
492 |             apagar_temp(buscar_funcao(&vetorMemoria, "parametros")); // Apaga os temporários usados na chamada
493 |
494 |             novaInstrucao = criarNoAssembly(typeI, "lw");
495 |             novaInstrucao->tipoI->rt = $temp;
496 |             novaInstrucao->tipoI->rs = $pilha;
497 |             novaInstrucao->tipoI->imediato = buscar_funcao(&vetorMemoria, "parametros")->tamanho;
498 |             instrucoesAssembly[indiceAssembly++] = novaInstrucao;
499 |
500 |             // Mostra o valor do $temp para o usuário
501 |             novaInstrucao = criarNoAssembly(typeI, "out");
502 |             novaInstrucao->tipoI->rt = $temp;
503 |             novaInstrucao->tipoI->rs = $zero;
504 |             novaInstrucao->tipoI->imediato = 0;
505 |             instrucoesAssembly[indiceAssembly++] = novaInstrucao;
506 |
507 |             return; // Não precisa fazer mais nada
508 |
509 |         }
510 |         else if(!strcmp(instrucao->arg1->nome, "input")){
511 |             // Como essa função não tem param, basta colocar o input no registrador
512 |             novaInstrucao = criarNoAssembly(typeI, "in");
513 |             novaInstrucao->tipoI->rt = instrucao->arg3->val;
514 |             novaInstrucao->tipoI->rs = $zero;
515 |             novaInstrucao->tipoI->imediato = 0;
516 |             instrucoesAssembly[indiceAssembly++] = novaInstrucao;
517 |
518 |             return; // Não precisa fazer mais nada
519 |
520 |         }
521 |     }
522 |
523 |     /* Continua */

```

Fonte: O Autor

Figura 21 – Geração de código *assembly* para a instrução intermediária CALL - parte 2 - presente no arquivo “assembly.c”

```

520 |     /* Continua */
521 |
522 |     for(int i = instrucao->arg2->val; i > 0; i--) {
523 |         // Salva o valor do param no $temp para ser usado no output
524 |         apagar_temp(buscar_funcao(&vetorMemoria, "parametros")); // Apaga os temporários usados na chamada
525 |
526 |         novaInstrucao = criarNoAssembly(typeI, "lw");
527 |         novaInstrucao->tipoI->rt = $temp;
528 |         novaInstrucao->tipoI->rs = $pilha;
529 |         novaInstrucao->tipoI->imediato = buscar_funcao(&vetorMemoria, "parametros")->tamanho;
530 |         instrucoesAssembly[indiceAssembly++] = novaInstrucao;
531 |
532 |         novaInstrucao = criarNoAssembly(typeI, "sw");
533 |         novaInstrucao->tipoI->rs = $$p;
534 |         novaInstrucao->tipoI->rt = $temp;
535 |         novaInstrucao->tipoI->imediato = i;
536 |         instrucoesAssembly[indiceAssembly++] = novaInstrucao;
537 |
538 |         // Armazena o valor de $fp dessa função em $temp para poder ser armazenado no novo frame
539 |         novaInstrucao = criarNoAssembly(typeR, "add");
540 |         novaInstrucao->tipoR->rt = $zero;
541 |         novaInstrucao->tipoR->rs = $temp;
542 |         novaInstrucao->tipoR->rd = $temp;
543 |         instrucoesAssembly[indiceAssembly++] = novaInstrucao;
544 |
545 |         // Faz com que $temp aponte para "Vínculo Controle" para poder ser armazenado no novo frame
546 |         novaInstrucao = criarNoAssembly(typeI, "addi");
547 |         novaInstrucao->tipoI->rt = $temp;
548 |         novaInstrucao->tipoI->rs = $temp;
549 |         novaInstrucao->tipoI->imediato = get_fp_relation(funcaoAtual, get_variavel(funcaoAtual, "Vínculo Controle"));
550 |         instrucoesAssembly[indiceAssembly++] = novaInstrucao;
551 |
552 |         // Armazena o valor de controle no seu respectivo local no novo frame
553 |         novaInstrucao = criarNoAssembly(typeI, "sw");
554 |         novaInstrucao->tipoI->rt = $temp;
555 |         novaInstrucao->tipoI->rs = $$p;
556 |         novaInstrucao->tipoI->imediato = instrucao->arg2->val + 1;
557 |         instrucoesAssembly[indiceAssembly++] = novaInstrucao;
558 |
559 |
560 |
561 |
562 |
563 |
564 |
565 |
566 |
567 |
568 |
569 |
570 |
571 |
572 |
573 |
574 |
575 |
576 |
577 |
578 |
579 |
580 |
581 |
582 |
583 |
584 |
585 |
586 |
587 |
588 |
589 |
590 |
591 |
592 |
593 |
594 |
595 |
596 |
597 |
598 |
599 |
600 |
601 |
602 |
603 |
604 |
605 |
606 |
607 |
608 |
609 |
610 |
611 |
612 |
613 |
614 |
615 |
616 |
617 |
618 |
619 |
620 |
621 |
622 |
623 |
624 |
625 |
626 |
627 |
628 |
629 |
630 |
631 |
632 |
633 |
634 |
635 |
636 |
637 |
638 |
639 |
640 |
641 |
642 |
643 |
644 |
645 |
646 |
647 |
648 |
649 |
650 |
651 |
652 |
653 |
654 |
655 |
656 |
657 |
658 |
659 |
660 |
661 |
662 |
663 |
664 |
665 |
666 |
667 |
668 |
669 |
670 |
671 |
672 |
673 |
674 |
675 |
676 |
677 |
678 |
679 |
680 |
681 |
682 |
683 |
684 |
685 |
686 |
687 |
688 |
689 |
690 |
691 |
692 |
693 |
694 |
695 |
696 |
697 |
698 |
699 |
700 |
701 |
702 |
703 |
704 |
705 |
706 |
707 |
708 |
709 |
710 |
711 |
712 |
713 |
714 |
715 |
716 |
717 |
718 |
719 |
720 |
721 |
722 |
723 |
724 |
725 |
726 |
727 |
728 |
729 |
730 |
731 |
732 |
733 |
734 |
735 |
736 |
737 |
738 |
739 |
740 |
741 |
742 |
743 |
744 |
745 |
746 |
747 |
748 |
749 |
750 |
751 |
752 |
753 |
754 |
755 |
756 |
757 |
758 |
759 |
760 |
761 |
762 |
763 |
764 |
765 |
766 |
767 |
768 |
769 |
770 |
771 |
772 |
773 |
774 |
775 |
776 |
777 |
778 |
779 |
780 |
781 |
782 |
783 |
784 |
785 |
786 |
787 |
788 |
789 |
790 |
791 |
792 |
793 |
794 |
795 |
796 |
797 |
798 |
799 |
800 |
801 |
802 |
803 |
804 |
805 |
806 |
807 |
808 |
809 |
810 |
811 |
812 |
813 |
814 |
815 |
816 |
817 |
818 |
819 |
820 |
821 |
822 |
823 |
824 |
825 |
826 |
827 |
828 |
829 |
830 |
831 |
832 |
833 |
834 |
835 |
836 |
837 |
838 |
839 |
840 |
841 |
842 |
843 |
844 |
845 |
846 |
847 |
848 |
849 |
850 |
851 |
852 |
853 |
854 |
855 |
856 |
857 |
858 |
859 |
860 |
861 |
862 |
863 |
864 |
865 |
866 |
867 |
868 |
869 |
870 |
871 |
872 |
873 |
874 |
875 |
876 |
877 |
878 |
879 |
880 |
881 |
882 |
883 |
884 |
885 |
886 |
887 |
888 |
889 |
890 |
891 |
892 |
893 |
894 |
895 |
896 |
897 |
898 |
899 |
900 |
901 |
902 |
903 |
904 |
905 |
906 |
907 |
908 |
909 |
910 |
911 |
912 |
913 |
914 |
915 |
916 |
917 |
918 |
919 |
920 |
921 |
922 |
923 |
924 |
925 |
926 |
927 |
928 |
929 |
930 |
931 |
932 |
933 |
934 |
935 |
936 |
937 |
938 |
939 |
940 |
941 |
942 |
943 |
944 |
945 |
946 |
947 |
948 |
949 |
950 |
951 |
952 |
953 |
954 |
955 |
956 |
957 |
958 |
959 |
960 |
961 |
962 |
963 |
964 |
965 |
966 |
967 |
968 |
969 |
970 |
971 |
972 |
973 |
974 |
975 |
976 |
977 |
978 |
979 |
980 |
981 |
982 |
983 |
984 |
985 |
986 |
987 |
988 |
989 |
990 |
991 |
992 |
993 |
994 |
995 |
996 |
997 |
998 |
999 |
1000 |
1001 |
1002 |
1003 |
1004 |
1005 |
1006 |
1007 |
1008 |
1009 |
1010 |
1011 |
1012 |
1013 |
1014 |
1015 |
1016 |
1017 |
1018 |
1019 |
1020 |
1021 |
1022 |
1023 |
1024 |
1025 |
1026 |
1027 |
1028 |
1029 |
1030 |
1031 |
1032 |
1033 |
1034 |
1035 |
1036 |
1037 |
1038 |
1039 |
1040 |
1041 |
1042 |
1043 |
1044 |
1045 |
1046 |
1047 |
1048 |
1049 |
1050 |
1051 |
1052 |
1053 |
1054 |
1055 |
1056 |
1057 |
1058 |
1059 |
1060 |
1061 |
1062 |
1063 |
1064 |
1065 |
1066 |
1067 |
1068 |
1069 |
1070 |
1071 |
1072 |
1073 |
1074 |
1075 |
1076 |
1077 |
1078 |
1079 |
1080 |
1081 |
1082 |
1083 |
1084 |
1085 |
1086 |
1087 |
1088 |
1089 |
1090 |
1091 |
1092 |
1093 |
1094 |
1095 |
1096 |
1097 |
1098 |
1099 |
1100 |
1101 |
1102 |
1103 |
1104 |
1105 |
1106 |
1107 |
1108 |
1109 |
1110 |
1111 |
1112 |
1113 |
1114 |
1115 |
1116 |
1117 |
1118 |
1119 |
1120 |
1121 |
1122 |
1123 |
1124 |
1125 |
1126 |
1127 |
1128 |
1129 |
1130 |
1131 |
1132 |
1133 |
1134 |
1135 |
1136 |
1137 |
1138 |
1139 |
1140 |
1141 |
1142 |
1143 |
1144 |
1145 |
1146 |
1147 |
1148 |
1149 |
1150 |
1151 |
1152 |
1153 |
1154 |
1155 |
1156 |
1157 |
1158 |
1159 |
1160 |
1161 |
1162 |
1163 |
1164 |
1165 |
1166 |
1167 |
1168 |
1169 |
1170 |
1171 |
1172 |
1173 |
1174 |
1175 |
1176 |
1177 |
1178 |
1179 |
1180 |
1181 |
1182 |
1183 |
1184 |
1185 |
1186 |
1187 |
1188 |
1189 |
1190 |
1191 |
1192 |
1193 |
1194 |
1195 |
1196 |
1197 |
1198 |
1199 |
1200 |
1201 |
1202 |
1203 |
1204 |
1205 |
1206 |
1207 |
1208 |
1209 |
1210 |
1211 |
1212 |
1213 |
1214 |
1215 |
1216 |
1217 |
1218 |
1219 |
1220 |
1221 |
1222 |
1223 |
1224 |
1225 |
1226 |
1227 |
1228 |
1229 |
1230 |
1231 |
1232 |
1233 |
1234 |
1235 |
1236 |
1237 |
1238 |
1239 |
1240 |
1241 |
1242 |
1243 |
1244 |
1245 |
1246 |
1247 |
1248 |
1249 |
1250 |
1251 |
1252 |
1253 |
1254 |
1255 |
1256 |
1257 |
1258 |
1259 |
1260 |
1261 |
1262 |
1263 |
1264 |
1265 |
1266 |
1267 |
1268 |
1269 |
1270 |
1271 |
1272 |
1273 |
1274 |
1275 |
1276 |
1277 |
1278 |
1279 |
1280 |
1281 |
1282 |
1283 |
1284 |
1285 |
1286 |
1287 |
1288 |
1289 |
1290 |
1291 |
1292 |
1293 |
1294 |
1295 |
1296 |
1297 |
1298 |
1299 |
1300 |
1301 |
1302 |
1303 |
1304 |
1305 |
1306 |
1307 |
1308 |
1309 |
1310 |
1311 |
1312 |
1313 |
1314 |
1315 |
1316 |
1317 |
1318 |
1319 |
1320 |
1321 |
1322 |
1323 |
1324 |
1325 |
1326 |
1327 |
1328 |
1329 |
1330 |
1331 |
1332 |
1333 |
1334 |
1335 |
1336 |
1337 |
1338 |
1339 |
1340 |
1341 |
1342 |
1343 |
1344 |
1345 |
1346 |
1347 |
1348 |
1349 |
1350 |
1351 |
1352 |
1353 |
1354 |
1355 |
1356 |
1357 |
1358 |
1359 |
1360 |
1361 |
1362 |
1363 |
1364 |
1365 |
1366 |
1367 |
1368 |
1369 |
1370 |
1371 |
1372 |
1373 |
1374 |
1375 |
1376 |
1377 |
1378 |
1379 |
1380 |
1381 |
1382 |
1383 |
1384 |
1385 |
1386 |
1387 |
1388 |
1389 |
1390 |
1391 |
1392 |
1393 |
1394 |
1395 |
1396 |
1397 |
1398 |
1399 |
1400 |
1401 |
1402 |
1403 |
1404 |
1405 |
1406 |
1407 |
1408 |
1409 |
1410 |
1411 |
1412 |
1413 |
1414 |
1415 |
1416 |
1417 |
1418 |
1419 |
1420 |
1421 |
1422 |
1423 |
1424 |
1425 |
1426 |
1427 |
1428 |
1429 |
1430 |
1431 |
1432 |
1433 |
1434 |
1435 |
1436 |
1437 |
1438 |
1439 |
1440 |
1441 |
1442 |
1443 |
1444 |
1445 |
1446 |
1447 |
1448 |
1449 |
1450 |
1451 |
1452 |
1453 |
1454 |
1455 |
1456 |
1457 |
1458 |
1459 |
1460 |
1461 |
1462 |
1463 |
1464 |
1465 |
1466 |
1467 |
1468 |
1469 |
1470 |
1471 |
1472 |
1473 |
1474 |
1475 |
1476 |
1477 |
1478 |
1479 |
1480 |
1481 |
1482 |
1483 |
1484 |
1485 |
1486 |
1487 |
1488 |
1489 |
1490 |
1491 |
1492 |
1493 |
1494 |
1495 |
1496 |
1497 |
1498 |
1499 |
1500 |
1501 |
1502 |
1503 |
1504 |
1505 |
1506 |
1507 |
1508 |
1509 |
1510 |
1511 |
1512 |
1513 |
1514 |
1515 |
1516 |
1517 |
1518 |
1519 |
1520 |
1521 |
1522 |
1523 |
1524 |
1525 |
1526 |
1527 |
1528 |
1529 |
1530 |
1531 |
1532 |
1533 |
1534 |
1535 |
1536 |
1537 |
1538 |
1539 |
1540 |
1541 |
1542 |
1543 |
1544 |
1545 |
1546 |
1547 |
1548 |
1549 |
1550 |
1551 |
1552 |
1553 |
1554 |
1555 |
1556 |
1557 |
1558 |
1559 |
1560 |
1561 |
1562 |
1563 |
1564 |
1565 |
1566 |
1567 |
1568 |
1569 |
1570 |
1571 |
1572 |
1573 |
1574 |
1575 |
1576 |
1577 |
1578 |
1579 |
1580 |
1581 |
1582 |
1583 |
1584 |
1585 |
1586 |
1587 |
1588 |
1589 |
1590 |
1591 |
1592 |
1593 |
1594 |
1595 |
1596 |
1597 |
1598 |
1599 |
1600 |
1601 |
1602 |
1603 |
1604 |
1605 |
1606 |
1607 |
1608 |
1609 |
1610 |
1611 |
1612 |
1613 |
1614 |
1615 |
1616 |
1617 |
1618 |
1619 |
1620 |
1621 |
1622 |
1623 |
1624 |
1625 |
1626 |
1627 |
1628 |
1629 |
1630 |
1631 |
1632 |
1633 |
1634 |
1635 |
1636 |
1637 |
1638 |
1639 |
1640 |
1641 |
1642 |
1643 |
1644 |
1645 |
1646 |
1647 |
1648 |
1649 |
1650 |
1651 |
1652 |
1653 |
1654 |
1655 |
1656 |
1657 |
1658 |
1659 |
1660 |
1661 |
1662 |
1663 |
1664 |
1665 |
1666 |
1667 |
1668 |
1669 |
1670 |
1671 |
1672 |
1673 |
1674 |
1675 |
1676 |
1677 |
1678 |
1679 |
1680 |
1681 |
1682 |
1683 |
1684 |
1685 |
1686 |
1687 |
1688 |
1689 |
1690 |
1691 |
1692 |
1693 |
1694 |
1695 |
1696 |
1697 |
1698 |
1699 |
1700 |
1701 |
1702 |
1703 |
1704 |
1705 |
1706 |
1707 |
1708 |
1709 |
1710 |
1711 |
1712 |
1713 |
1714 |
1715 |
1716 |
1717 |
1718 |
1719 |
1720 |
1721 |
1722 |
1723 |
1724 |
1725 |
1726 |
1727 |
1728 |
1729 |
1730 |
1731 |
1732 |
1733 |
1734 |
1735 |
1736 |
1737 |
1738 |
1739 |
1740 |
1741 |
1742 |
1743 |
1744 |
1745 |
1746 |
1747 |
1748 |
1749 |
1750 |
1751 |
1752 |
1753 |
1754 |
1755 |
1756 |
1757 |
1758 |
1759 |
1760 |
1761 |
1762 |
1763 |
1764 |
1765 |
1766 |
1767 |
1768 |
1769 |
1770 |
1771 |
1772 |
1773 |
1774 |
1775 |
1776 |
1777 |
1778 |
1779 |
1780 |
1781 |
1782 |
1783 |
1784 |
1785 |
1786 |
1787 |
1788 |
1789 |
1790 |
1791 |
1792 |
1793 |
1794 |
1795 |
1796 |
1797 |
1798 |
1799 |
1800 |
1801 |
1802 |
1803 |
1804 |
1805 |
1806 |
1807 |
1808 |
1809 |
1810 |
1811 |
1812 |
1813 |
1814 |
1815 |
1816 |
1817 |
1818 |
1819 |
1820 |
1821 |
1822 |
1823 |
1824 |
1825 |
1826 |
1827 |
1828 |
1829 |
1830 |
1831 |
1832 |
1833 |
1834 |
1835 |
1836 |
1837 |
1838 |
1839 |
1840 |
1841 |
1842 |
1843 |
1844 |
1845 |
1846 |
1847 |
1848 |
1849 |
1850 |
1851 |
1852 |
1853 |
1854 |
1855 |
1856 |
1857 |
1858 |
1859 |
1860 |
1861 |
1862 |
1863 |
1864 |
1865 |
1866 |
1867 |
1868 |
1869 |
1870 |
1871 |
1872 |
1873 |
1874 |
1875 |
1876 |
1877 |
1878 |
1879 |
1880 |
1881 |
1882 |
1883 |
1884 |
1885 |
1886 |
1887 |
1888 |
1889 |
1890 |
1891 |
1892 |
1893 |
1894 |
1895 |
1896 |
1897 |
1898 |
1899 |
1900 |
1901 |
1902 |
1903 |
1904 |
1905 |
1906 |
1907 |
1908 |
1909 |
1910 |
1911 |
1912 |
1913 |
1914 |
1915 |
1916 |
1917 |
1918 |
1919 |
1920 |
1921 |
1922 |
1923 |
1924 |
1925 |
1926 |
1927 |
1928 |
1929 |
1930 |
1931 |
1932 |
1933 |
1934 |
1935 |
1936 |
1937 |
1938 |
1939 |
1940 |
1941 |
1942 |
1943 |
1944 |
1945 |
1946 |
1947 |
1948 |
1949 |
1950 |
1951 |
1952 |
1953 |
1954 |
1955 |
1956 |
1957 |
1958 |
1959 |
1960 |
1961 |
1962 |
1963 |
1964 |
1965 |
1966 |
1967 |
1968 |
1969 |
1970 |
1971 |
1972 |
1973 |
1974 |
1975 |
1976 |
1977 |
1978 |
1979 |
1980 |
1981 |
1982 |
1983 |
1984 |
1985 |
1986 |
1987 |
1988 |
1989 |
1990 |
1991 |
1992 |
1993 |
1994 |
1995 |
1996 |
1997 |
1998 |
1999 |
2000 |
2001 |
2002 |
2003 |
2004 |
2005 |
2006 |
2007 |
2008 |
2009 |
2010 |
2011 |
2012 |
2013 |
2014 |
2015 |
2016 |
2017 |
2018 |
2019 |
2020 |
2021 |
2022 |
2023 |
2024 |
2025 |
2026 |
2027 |
2028 |
2029 |
2030 |
2031 |
2032 |
2033 |
2034 |
2035 |
2036 |
2037 |
2038 |
2039 |
2040 |
2041 |
2042 |
2043 |
2044 |
2045 |
2046 |
2047 |
2048 |
2049 |
2050 |
2051 |
2052 |
2053 |
2054 |
2055 |
2056 |
2057 |
2058 |
2059 |
2060 |
2061 |
2062 |
2063 |
2064 |
2065 |
2066 |
2067 |
2068 |
2069 |
2070 |
2071 |
2072 |
2073 |
2074 |
2075 |
2076 |
2077 |
2078 |
2079 |
2080 |
2081 |
2082 |
2083 |
2084 |
2085 |
2086 |
2087 |
2088 |
2089 |
2090 |
2091 |
2092 |
2093 |
2094 |
2095 |
2096 |
2097 |
2098 |
2099 |
2100 |
2101 |
2102 |
2103 |
2104 |
2105 |
2106 |
2107 |
2108 |
2109 |
2110 |
2111 |
2112 |
2113 |
2114 |
2115 |
2116 |
2117 |
2118 |
2119 |
2120 |
2121 |
2122 |
2123 |
2124 |
2125 |
2126 |
2127 |
2128 |
2129 |
2130 |
2131 |
2132 |
2133 |
2134 |
2135 |
2136 |
2137 |
2138 |
2139 |
2140 |
2141 |
2142 |
2143 |
2144 |
2145 |
2146 |
2147 |
2148 |
2149 |
2150 |
2151 |
2152 |
2153 |
2154 |
2155 |
2156 |
2157 |
2158 |
2159 |
2160 |
2161 |
2162 |
2163 |
2164 |
2165 |
2166 |
2167 |
2168 |
2169 |
2170 |
2171 |
2172 |
2173 |
2174 |
2175 |
2176 |
2177 |
2178 |
2179 |
2180 |
2181 |
2182 |
2183 |
2184 |
2185 |
2186 |
2187 |
2188 |
2189 |
2190 |
2191 |
2192 |
2193 |
2194 |
2195 |
2196 |
2197 |
2198 |
2199 |
2200 |
2201 |
2202 |
2203 |
2204 |
2205 |
2206 |
2207 |
2208 |
2209 |
2210 |
2211 |
2212 |
2213 |
2214 |
2215 |
2216 |
2217 |
2218 |
2219 |
2220 |
2221 |
2222 |
2223 |
2224 |
2225 |
2226 |
2227 |
2228 |
2229 |
2230 |
2231 |
2232 |
2233 |
2234 |
2235 |
2236 |
2237 |
2238 |
2239 |
2240 |
2241 |
2242 |
2243 |
2244 |
2245 |
2246 |
2247 |
2248 |
2249 |
2250 |
2251 |
2252 |
2253 |
2254 |
2255 |
2256 |
2257 |
2258 |
2259 |
2260 |
2261 |
2262 |
2263 |
2264 |
2265 |
2266 |
2267 |
2268 |
2269 |
2270 |
2271 |
2272 |
2273 |
2274 |
2275 |
2276 |
2277 |
2278 |
2279 |
2280 |
2281 |
2282 |
2283 |
2284 |
2285 |
2286 |
2287 |
2288 |
2289 |
2290 |
2291 |
2292 |
2293 |
2294 |
2295 |
2296 |
2297 |
2298 |
2299 |
2300 |
2301 |
2302 |
2303 |
2304 |
2305 |
2306 |
2307 |
2308 |
2309 |
2310 |
2311 |
2312 |
2313 |
2314 |
2315 |
2316 |
2317 |
2318 |
2319 |
2320 |
2321 |
2322 |
2323 |
2324 |
2325 |
2326 |
2327 |
2328 |
2329 |
2330 |
2331 |
2332 |
2333 |
2334 |
2335 |
2336 |
2337 |
2338 |
2339 |
2340 |
2341 |
2342 |
2343 |
2344 |
2345 |
2346 |
2347 |
2348 |
2349 |
2350 |
2351 |
2352 |
2353 |
2354 |
2355 |
2356 |
2357 |
2358 |
2359 |
2360 |
2361 |
2362 |
2363 |
2364 |
2365 |
2366 |
2367 |
2368 |
2369 |
2370 |
2371 |
2372 |
2373 |
2374 |
2375 |
2376 |
2377 |
2378 |
2379 |
2380 |
2381 |
2382 |
2383 |
2384 |
2385 |
2386 |
2387 |
2388 |
2389 |
2390 |
2391 |
2392 |
2393 |
2394 |
2395 |
2396 |
2397 |
2398 |
2399 |
2400 |
2401 |
2402 |
2403 |
2404 |
2405 |
2406 |
2407 |
2408 |
2409 |
2410 |
2411 |
2412 |
2413 |
2414 |
2415 |
2416 |
2417 |
2418 |
2419 |
2420 |
2421 |
2422 |
2423 |
2424 |
2425 |
2426 |
2427 |
2428 |
2429 |
2430 |
2431 |
2432 |
2433 |
2434 |
2435 |
2436 |
2437 |
2438 |
2439 |
2440 |
2441 |
2442 |
2443 |
2444 |
2445 |
2446 |
2447 |
2448 |
2449 |
2450 |
2451 |
2452 |
2453 |
2454 |
2455 |
2456 |
2457 |
2458 |
2459 |
2460 |
2461 |
2462 |
2463 |
2464 |
2465 |
2466 |
2467 |
2468 |
2469 |
2470 |
2471 |
2472 |
2473 |
2474 |
2475 |
2476 |
2477 |
2478 |
2479 |
2480 |
2481 |
2482 |
2483 |
2484 |
2485 |
2486 |
2487 |
2488 |
2489 |
2490 |
2491 |
2492 |
2493 |
2494 |
2495 |
2496 |
2497 |
2498 |
2499 |
2500 |
2501 |
2502 |
25
```

Figura 22 – Geração de código *assembly* para a instrução intermediária CALL - parte 3 - presente no arquivo “assembly.c”

```

560 // Incrementa o valor de $fp e $sp para o novo frame da função
561 novaInstrucao = criarNoAssembly(typeI, "addi");
562 novaInstrucao->tipoI->r = $fp;
563 novaInstrucao->tipoI->s = $sp;
564 novaInstrucao->tipoI->imediato = get_sp(funcaoAtual) + 1;
565 instrucoesAssembly[indiceAssembly++] = novaInstrucao;
566
567 novaInstrucao = criarNoAssembly(typeI, "addi");
568 novaInstrucao->tipoI->r = $sp;
569 novaInstrucao->tipoI->s = $fp;
570 novaInstrucao->tipoI->imediato = get_sp(funcaoAtual) + 1;
571 instrucoesAssembly[indiceAssembly++] = novaInstrucao;
572
573 // Pulando para a função chamada
574 novaInstrucao = criarNoAssembly(typeJ, "jal");
575 novaInstrucao->tipoJ->labelMediato = strdup(instrucao->arg1->nome);
576 instrucoesAssembly[indiceAssembly++] = novaInstrucao;
577
578 // Volta os valores de $fp
579 novaInstrucao = criarNoAssembly(typeI, "subi");
580 novaInstrucao->tipoI->r = $fp;
581 novaInstrucao->tipoI->s = $fp;
582 novaInstrucao->tipoI->imediato = 25;
583 instrucoesAssembly[indiceAssembly++] = novaInstrucao;
584
585 // Volta os valores de $sp
586 novaInstrucao = criarNoAssembly(typeI, "subi");
587 novaInstrucao->tipoI->r = $sp;
588 novaInstrucao->tipoI->s = $sp;
589 novaInstrucao->tipoI->imediato = 25;
590 instrucoesAssembly[indiceAssembly++] = novaInstrucao;
591
592 if(instrucao->arg3->tipo != Vazio){
593     // Armazena o valor do retorno da função anterior no registrador passado como argumento
594     novaInstrucao = criarNoAssembly(typeI, "lw");
595     novaInstrucao->tipoI->r = instrucao->arg3->val;
596     novaInstrucao->tipoI->s = $fp;
597     novaInstrucao->tipoI->imediato = get_fp_relation(funcaoAtual, get_variavel(funcaoAtual, "Valor Retorno"));
598     instrucoesAssembly[indiceAssembly++] = novaInstrucao;
599 }
600

```

Fonte: O Autor

No momento que a função que foi chamada for finalizada ela irá voltar na instrução seguinte ao “jal”. Como a função foi alterada novamente, os valores dos ponteiros devem voltar ao que eram antes. Nesse caso, basta subtrair o valor 25. Para finalizar o procedimento, caso a função tenha retornado algo, basta realizar uma leitura na área da memória “valor de retorno”. O motivo de realizar esse procedimento será explicado adiante, durante a função “RET”.

A instrução de código intermediário “FUN” tem a fundamental importância de sinalizar o inicio de uma nova função no código digitado pelo usuário. Ao convertê-la para o código *assembly*, basta realizar uma única ação, que é armazenar o valor presente no registrador \$ra dentro do seu *frame*, no campo “Valor de Retorno”. Assim, quando o procedimento seja finalizado, uma instrução de *jump* incondicional será realizada para quem a chamou. Caso a função seja a “main”, alguns procedimento a mais terão que ser realizados: 1) Inicializar os valores de \$fp e \$sp considerando os valores das variáveis globais já separadas na memória; e 2) inicializar o registrador \$pilha para o valor fixo para a pilha de parâmetros. A Figura 23 representa o código para a verificação anterior.

Figura 23 – Geração de código *assembly* para a instrução intermediária FUN, presente no arquivo “assembly.c”

```

247 |     else if(!strcmp(instrucao->op, "FUN")){
248 |         novaInstrucao = criarNoAssembly(typeLabel, instrucao->arg2->nome);
249 |         novaInstrucao->tipolabel->boolean = 0;
250 |
251 |         adicionarLabel(instrucao->arg2->nome, indiceAssembly);
252 |         instrucoesAssembly[indiceAssembly++] = novaInstrucao;
253 |
254 |         funcaoAtual = inserirFuncao(&vetorMemoria, instrucao->arg2->nome);
255 |
256 |         if(!strcmp(instrucao->arg2->nome, "main")){
257 |             /* Carrega os registradores $fp e $sp com seus valores iniciais */
258 |             novaInstrucao = criarNoAssembly(typeI, "ori");
259 |             novaInstrucao->tipoi->rt = $fp;
260 |             novaInstrucao->tipoi->rs = $zero;
261 |             novaInstrucao->tipoi->imediato = buscarFuncao(&vetorMemoria, "global")->tamanho + get_fp(funcaoAtual);
262 |             instrucoesAssembly[indiceAssembly++] = novaInstrucao;
263 |             printf("fp: %d\n", novaInstrucao->tipoi->imediato);
264 |
265 |             novaInstrucao = criarNoAssembly(typeI, "ori");
266 |             novaInstrucao->tipoi->rt = $sp;
267 |             novaInstrucao->tipoi->rs = $zero;
268 |             novaInstrucao->tipoi->imediato = buscarFuncao(&vetorMemoria, "global")->tamanho + get_sp(funcaoAtual);
269 |             instrucoesAssembly[indiceAssembly++] = novaInstrucao;
270 |             printf("sp: %d\n", novaInstrucao->tipoi->imediato);
271 |
272 |             // Inicia o ponteiro de memória para os parametros
273 |             novaInstrucao = criarNoAssembly(typeI, "ori");
274 |             novaInstrucao->tipoi->rt = $pilha;
275 |             novaInstrucao->tipoi->rs = $zero;
276 |             novaInstrucao->tipoi->imediato = MEM_PARAM; // Valor fixo para a localização dos parametros
277 |             instrucoesAssembly[indiceAssembly++] = novaInstrucao;
278 |         }
279 |     else{
280 |         // Guarda o valor de controle para a função anterior
281 |         novaInstrucao = criarNoAssembly(typeI, "sw");
282 |         novaInstrucao->tipoi->rt = $ra;
283 |         novaInstrucao->tipoi->rs = $fp;
284 |         novaInstrucao->tipoi->imediato = get_fp_relation(funcaoAtual, get_variavel(funcaoAtual, "Endereco Retorno")) + instrucao->arg3->val;
285 |         instrucoesAssembly[indiceAssembly++] = novaInstrucao;
286 |     }
287 | }

```

Fonte: O Autor

Por fim, basta exemplificar o que será feito nas instruções de “RET” (Figura 25) e “END” (Figura 24), já que suas funcionalidades estão interligadas. Na primeira, será preciso retornar um valor para a função anterior na pilha da memória, já que ela está aguardando esse valor para continuar a sua rotina, dessa forma foi programado para que todos os resultados a serem retornados devem ser armazenados no campo “valor de retorno”, como citado anteriormente. Dessa forma, basta carregar o valor que aponta para *frame* anterior e armazenar o valor indicado no registrador. Já a segunda instrução, basta ler da memória o endereço de retorno, armazená-lo em um registrador e, por fim, realizar um *jump register*. Caso a instrução de finalização seja da função “main”, nada será feito, já que o programa será finalizado na instrução “HALT” seguinte.

Figura 24 – Geração de código *assembly* para a instrução intermediária END presente no arquivo “assembly.c”

```

601 |     else if(!strcmp(instrucao->op, "END")){
602 |         if(!strcmp(instrucao->arg1->nome, "main")){
603 |             return; // Não precisa fazer mais nada, já que a próxima instrução eh o HALT
604 |         }
605 |
606 |         // Restaura o valor de $ra da função anterior
607 |         novaInstrucao = criarNoAssembly(typeI, "lw");
608 |         novaInstrucao->tipoi->rs = $fp;
609 |         novaInstrucao->tipoi->rt = $ra;
610 |         novaInstrucao->tipoi->imediato = get_fp_relation(funcaoAtual, get_variavel(funcaoAtual, "Endereco Retorno"));
611 |         instrucoesAssembly[indiceAssembly++] = novaInstrucao;
612 |
613 |         // Pula para a instrução que fez a chamada da função
614 |         novaInstrucao = criarNoAssembly(typeR, "jr");
615 |         novaInstrucao->tipor->rs = $ra;
616 |         novaInstrucao->tipor->rd = $zero;
617 |         novaInstrucao->tipor->rt = $zero;
618 |         instrucoesAssembly[indiceAssembly++] = novaInstrucao;
619 |
620 |     }

```

Fonte: O Autor

Figura 25 – Geração de código *assembly* para a instrução intermediária RET presente no arquivo “assembly.c”

```

288 |     else if(!strcmp(instrucao->op, "RET")){
289 |         if(!strcmp(funcaoAtual->nome, "main")) return;
290 |
291 |         // Acessa o valor de controle da função anterior
292 |         novaInstrucao = criarNoAssembly(typeI, "lw");
293 |         novaInstrucao->tipol->rt = $temp;
294 |         novaInstrucao->tipol->rs = $fp;
295 |         novaInstrucao->tipol->imediato = get_fp_relation(funcaoAtual, get_variavel(funcaoAtual, "Vinculo Controle"));
296 |         instrucoesAssembly[indiceAssembly++] = novaInstrucao;
297 |
298 |         // Salva o valor do retorno no frame da função anterior
299 |         novaInstrucao = criarNoAssembly(typeI, "sw");
300 |         novaInstrucao->tipol->rs = $temp;
301 |         novaInstrucao->tipol->rt = instrucao->arg1->val;
302 |         novaInstrucao->tipol->imediato = 2; // 2 para avançar para "Valor de Retorno"
303 |         instrucoesAssembly[indiceAssembly++] = novaInstrucao;
304 |
305 |     }

```

Fonte: O Autor

### 3.2.4 Código Binário

Após obter o código *assembly*, basta convertê-lo para o valor binário correspondente para cada campo designado. Por ser apenas um processo direto de conversão um para um ele não possuí muitas complicações de implementação no projeto do compilador. A Figura 26 ilustra as estruturas para armazenar os valores binários antes de realizar a conversão para um arquivo externo.

Figura 26 – Estruturas para armazenar os valores binários do código, presente no arquivo “binario.h”

```

10  typedef struct {
11      unsigned int funct:6;
12      unsigned int shamt:5;
13      unsigned int rd:5;
14      unsigned int rt:5;
15      unsigned int rs:5;
16      unsigned int opcode:6;
17  } BIN_R;
18
19  typedef struct {
20      unsigned int immediate:16;
21      unsigned int rt:5;
22      unsigned int rs:5;
23      unsigned int opcode:6;
24  } BIN_I;
25
26  typedef struct {
27      unsigned int address:26;
28      unsigned int opcode:6;
29  } BIN_J;

```

Fonte: O Autor

A estrutura foi implementada seguindo o padrão já mencionado das instruções nas Tabelas 1, 2 e 3. Os dois pontos após a declaração das variáveis representam a quantidade de bits que serão utilizados para armazenar os valores para aquele campo em específico e, dessa forma, a soma dos bits utilizados deve ser igual a 32, o equivalente ao tamanho de um “*unsigned int*”. Com isso, todas as *structs* geradas irão armazenar exatamente 32 bits e não será preciso realizar nenhum tratamento especial para a exclusão de valores a mais

de variáveis maiores e, portanto, ao realizar a impressão no arquivo basta mostrar todos os bits armazenados dentro dessa estrutura.

Figura 27 – Função “binario” utilizada para converter o código *assembly* para binário, implementado no arquivo “binario.c”

```

131 void binario(FILE* arquivo){
132     BIN_I* binI;
133     BIN_J* binJ;
134     BIN_R* binR;
135
136     for(int i = 0; i < indiceAssembly; i++){
137         switch (instrucoesAssembly[i]->tipo)
138         {
139             case typeR:
140                 binR = binarioR(instrucoesAssembly[i]);
141                 printBits(sizeof(*binR), &(*binR), arquivo);
142                 free(binR);
143                 break;
144             case typeI:
145                 binI = binarioI(instrucoesAssembly[i]);
146                 printBits(sizeof(*binI), &(*binI), arquivo);
147                 free(binI);
148                 break;
149             case typeJ:
150                 binJ = binarioJ(instrucoesAssembly[i]);
151                 printBits(sizeof(*binJ), &(*binJ), arquivo);
152                 free(binJ);
153                 break;
154             case typeLabel:
155                 binR = binarioNop();
156                 printBits(sizeof(*binR), &(*binR), arquivo);
157                 free(binR);
158             }
159         fprintf(arquivo, "\n");
160     }
161 }
162 }
```

Fonte: O Autor

Na Figura 27 a função “binario” irá realizar um laço para percorrer todas as instruções *assembly*, separando elas por tipo e enviando para métodos específicos para tratar cada uma delas, representados pela Figura 28. As funções “get\_opcode” e “get\_funct”, ilustradas na Figura 29, tem o propósito de retornar o *opcode* e o *funct* das instruções, com base no código binário descrito nas Tabelas 1, 2 e 3. Já as demais funções apenas retornam o mesmo valor passado como parâmetro, já que os valores de endereços, registradores e imediatos não precisam de conversão.

Por fim, para imprimir o valor do código binário em um arquivo, a função “printBits” foi implementada (Figura 30) para receber o endereço de uma estrutura qualquer em conjunto com o seu tamanho e, assim, acessar os seus valores armazenados, mostrando bit a bit em um arquivo externo, que também é passado como argumento.

Figura 28 – Funções específicas para a conversão do código *assembly* em formatos do tipo R, I e J, respectivamente. Implementado no arquivo “binario.c”

```

78  BIN_R* binarioR(ASSEMBLY* instrucao){
79      BIN_R* bin = (BIN_R*)malloc(sizeof(BIN_R));
80      bin->opcode = get_opcode(instrucao->tipoR->nome, instrucao->tipo);
81      bin->rs = get_register(instrucao->tipoR->rs);
82      bin->rt = get_register(instrucao->tipoR->rt);
83      bin->rd = get_register(instrucao->tipoR->rd);
84      bin->shamt = get_shamt(0);
85      bin->funct = get_funct(instrucao->tipoR->nome);
86      return bin;
87  }
88
89
90  BIN_I* binarioI(ASSEMBLY* instrucao){
91      BIN_I* bin = (BIN_I*)malloc(sizeof(BIN_I));
92      bin->opcode = get_opcode(instrucao->tipoI->nome, instrucao->tipo);
93      bin->rs = get_register(instrucao->tipoI->rs);
94      bin->rt = get_register(instrucao->tipoI->rt);
95
96      if(!strcmp(instrucao->tipoI->nome, "bne") || !strcmp(instrucao->tipoI->nome, "beq")){
97          // Converte numero para string
98          char label[26];
99          sprintf(label, "Label %d", instrucao->tipoI->label);
100         bin->immediate = get_address(label);
101     }
102     else{
103         bin->immediate = get_immediate(instrucao->tipoI->imediato);
104     }
105
106     return bin;
107 }
108
109 BIN_J* binarioJ(ASSEMBLY* instrucao){
110     BIN_J* bin = (BIN_J*)malloc(sizeof(BIN_J));
111     bin->opcode = get_opcode(instrucao->tipoJ->nome, instrucao->tipo);
112     bin->address = get_address(instrucao->tipoJ->labelImmediato);
113     return bin;
114 }
```

Fonte: O Autor

Figura 29 – Funções para conversão do nome da instrução em seu opcode e funct (para tipo R) em binário. Presente no arquivo “binario.c”

```

8  unsigned int get_opcode(char* nome, tipoInstrucao tipo){
9      int opcode = -1;
10
11      if(tipo == typeR) opcode = 0b000000;
12      else if(!strcmp(nome, "lw")) opcode = 0b100011;
13      else if(!strcmp(nome, "sw")) opcode = 0b101011;
14      else if(!strcmp(nome, "addi")) opcode = 0b001000;
15      else if(!strcmp(nome, "subi")) opcode = 0b001001;
16      else if(!strcmp(nome, "andi")) opcode = 0b001100;
17      else if(!strcmp(nome, "ori")) opcode = 0b001101;
18      else if(!strcmp(nome, "beq")) opcode = 0b000100;
19      else if(!strcmp(nome, "bne")) opcode = 0b000101;
20      else if(!strcmp(nome, "slti")) opcode = 0b001010;
21      else if(!strcmp(nome, "in")) opcode = 0b011111;
22      else if(!strcmp(nome, "out")) opcode = 0b011110;
23      else if(!strcmp(nome, "jr")) opcode = 0b000010;
24      else if(!strcmp(nome, "jal")) opcode = 0b000011;
25      else if(!strcmp(nome, "halt")) opcode = 0b111111;
26      else if(!strcmp(nome, "xori")) opcode = 0b101101;
27
28      return opcode;
29  }
30
31  unsigned int get_funct(char* nome){
32      int funct = -1;
33
34      if(!strcmp(nome, "add")) funct = 0b100000;
35      else if(!strcmp(nome, "sub")) funct = 0b100010;
36      else if(!strcmp(nome, "and")) funct = 0b100100;
37      else if(!strcmp(nome, "or")) funct = 0b100101;
38      else if(!strcmp(nome, "jr")) funct = 0b001000;
39      else if(!strcmp(nome, "sll")) funct = 0b101010;
40      else if(!strcmp(nome, "srl")) funct = 0b000000;
41      else if(!strcmp(nome, "nor")) funct = 0b100111;
42      else if(!strcmp(nome, "sll")) funct = 0b000000;
43      else if(!strcmp(nome, "srl")) funct = 0b000000;
44      else if(!strcmp(nome, "div")) funct = 0b011010;
45      else if(!strcmp(nome, "mult")) funct = 0b011100;
46      else if(!strcmp(nome, "xor")) funct = 0b101101;
47
48      return funct;
49 }
```

Fonte: O Autor

Figura 30 – Função para mostrar o valor binário para um arquivo esterno. Presente no arquivo “binario.c”

```
117 void printBits(size_t const size, void const * const ptr, FILE* arquivo)
118 {
119     unsigned char *b = (unsigned char*) ptr;
120     unsigned char byte;
121     int i, j;
122
123     for (i = size-1; i >= 0; i--) {
124         for (j = 7; j >= 0; j--) {
125             byte = (b[i] >> j) & 1;
126             fprintf(arquivo, "%u", byte);
127         }
128     }
129 }
```

Fonte: O Autor

## 4 Resultados

Para demonstrar o funcionamento do compilador, três exemplos serão testados e com os resultados obtidos de cada uma das três etapas da fase de geração de código fonte.

## 4.1 Fatorial Recursivo

### 4.1.1 Código Fonte

```
1 int fatorial(int n){
2     int vfat;
3
4     if (n == 1){
5         return (1);
6     }
7     else{
8         vfat = fatorial(n-1)*n;
9         return (vfat);
10    }
11 }
12
13 int main(void){
14     int x;
15     x = input();
16
17     output(fatorial(x));
18
19     return 0;
20 }
```

#### 4.1.2 Relação entre Códigos

Após adicionar o arquivo do código fonte como entrada no compilador finalizado, os resultados para o código intermediário, *assembly* e binário pode ser observado na [Tabela 13](#).

Tabela 13: Geração de códigos da fase de síntese para o programa fatorial recursivo

Continued on next page

Tabela 13: Geração de códigos da fase de síntese para o programa factorial recursivo (Continued)

ALLOC, vfat, fatorial, -	-	-
LOAD, \$t0, n, -	lw \$t0 0(\$fp)	10001111010000000000000000000000
LOADI, \$t1, 1, -	ori \$t1 \$zero 1	00110111110001000000000000000001
EQ, \$t0, \$t1, \$t2	slt \$t2 \$t0 \$t1	0000000000000001000100000101010
	slt \$t2 \$t1 \$t0	0000000000100000000100000101010
	xori \$t2 \$t2 1	10110100010000100000000000000001
IFF, \$t2, L1, -	beq \$t2 \$zero Label 1	00010011110001000000000000000001111
LOADI, \$t3, 1, -	ori \$t3 \$zero 1	00110111110001100000000000000001
RET, \$t3, -, -	lw \$temp 1(\$fp)	10001111011011000000000000000001
	sw \$t3 2(\$temp)	101011101100011000000000000000010
GOTO, L0, -, -	j Label 0	0000100000000000000000000000000101001
GOTO, L2, -, -	j Label 2	0000100000000000000000000000000101000
LABEL, L1, -, -	Label 1:	000000111111111111100000100000
LOAD, \$t4, vfat, -	lw \$t4 7(\$fp)	1000111101001000000000000000000111
LOAD, \$t0, n, -	lw \$t0 0(\$fp)	100011110100000000000000000000000
LOADI, \$t5, 1, -	ori \$t5 \$zero 1	00110111110010100000000000000001
SUB, \$t0, \$t5, \$t6	sub \$t6 \$t0 \$t5	0000000000001010011000000100010
PARAM, \$t6, INT, -	sw \$t6 0(\$pilha)	10101110100011000000000000000000
CALL, fatorial, 1, \$t7	lw \$temp 0(\$pilha)	100011101011011000000000000000000
	sw \$temp 1(\$sp)	10101111001101100000000000000001
	add \$temp \$fp \$zero	0000001101111111011000001000000
	addi \$temp \$temp 1	00100011011110110000000000000001
	sw \$temp 2(\$sp)	101011110011011000000000000000010
	addi \$fp \$fp 25	001000110111101000000000000000011001
	addi \$sp \$sp 25	001000110011100000000000000000011001
	jal fatorial	000011000000000000000000000000000000000001
	subi \$fp \$fp 25	001001110111101000000000000000011001
	subi \$sp \$sp 25	001001110011100000000000000000011001
	lw \$t7 3(\$fp)	1000111101001110000000000000000011

Continued on next page

Tabela 13: Geração de códigos da fase de síntese para o programa factorial recursivo (Continued)

---

Continued on next page

Tabela 13: Geração de códigos da fase de síntese para o programa factorial recursivo (Continued)

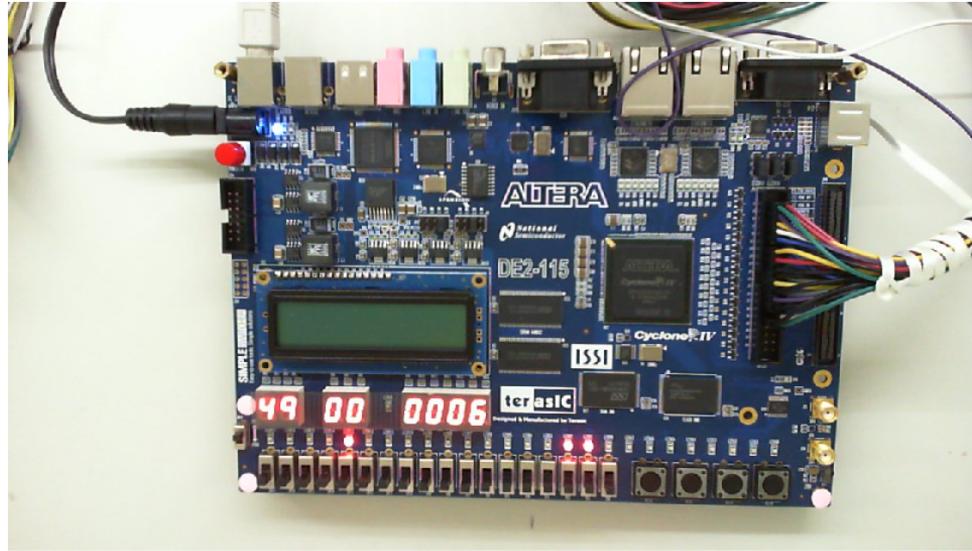
	addi \$sp \$sp 25	0010001110011100000000000000000011001
	jal factorial	00001100000000000000000000000000000001
	subi \$fp \$fp 25	0010011101110100000000000011001
	subi \$sp \$sp 25	001001110011100000000000000011001
	lw \$t11 2(\$fp)	1000111101010110000000000000000010
PARAM, \$t11, INT, -	sw \$t11 0(\$pilha)	10101110100101100000000000000000000
CALL, output, 1, -	lw \$temp 0(\$pilha)	10001110101101100000000000000000000
	out \$zero \$temp 0	0111101101111110000000000000000000
RET, \$t31, -, -	-	-
GOTO, L3, -, -	j Label 3	00001000000000000000000000001000101
LABEL, L3, -, -	Label 3:	00000011111111111111000000100000
END, main, -, -	-	-
HALT, -, -, -	halt \$zero	11111111111111111111111111111111111111

Fonte: O Autor

#### 4.1.3 Teste FPGA

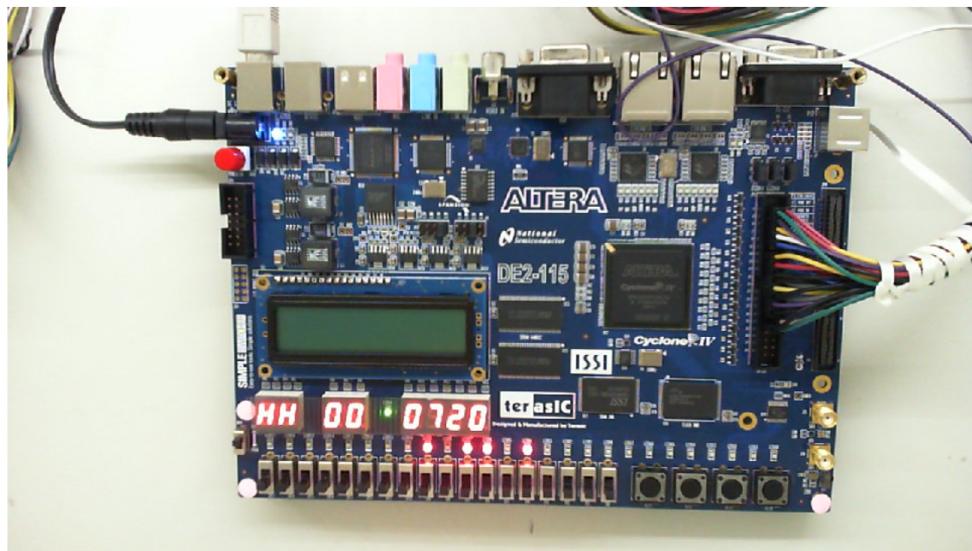
Para finalizar, um teste foi realizada com o envio do processador programado para operar o programa anteriormente compilado. O valor de entrada do usuário será de 6, portanto o valor resultante deverá ser 720.

Figura 31 – Inserindo o valor decimal 6 para o cálculo do fatorial na placa FPGA



Fonte: O Autor

Figura 32 – Resultado obtido para o cálculo do fatorial de 6 na placa FPGA



Fonte: O Autor

Considerando os testes realizados nas Figuras 31 e 32, é constatado que o programa está calculando corretamente o fatorial do número 6, já que o valor corresponde é 720.

## 4.2 Máximo Denominador Comum

### 4.2.1 Código Fonte

```
1 int gcd (int u, int v){
```

```

2     if (v == 0) {return u;}
3     else return gcd(v,u-u/v*v);
4 }
5
6 void main(void){
7     int x; int y;
8
9     x = input(); y = input();
10    output(gcd(x,y));
11 }
```

#### 4.2.2 Relação entre Códigos

Tabela 14: Resultado da geração de códigos para o programa máximo denominador comum

Código Intermediário	Código Assembly	Código Binário
GOTO, main, -, -	j main	00001000000000000000000000000000101110
FUN, INT, gcd, 2	gcd:	0000001111111111111100000100000
	sw \$ra 3(\$fp)	10101111011111000000000000000011
ARG, INT, u, gcd	-	-
ARG, INT, v, gcd	-	-
LOAD, \$t0, u, -	lw \$t0 0(\$fp)	10001111010000000000000000000000
LOAD, \$t1, v, -	lw \$t1 1(\$fp)	10001111010000100000000000000001
LOAD, \$t1, v, -	lw \$t1 1(\$fp)	10001111010000100000000000000001
EQ, \$t1, \$t31, \$t2	slt \$t2 \$t1 \$zero	0000000000111110001000000101010
	slt \$t2 \$zero \$t1	00000011111000010001000000101010
	xori \$t2 \$t2 1	10110100010000100000000000000001
IFF, \$t2, L1, -	beq \$t2 \$zero Label 1	00010011111000100000000000000001111
LOAD, \$t0, u, -	lw \$t0 0(\$fp)	10001111010000000000000000000000
RET, \$t0, -, -	lw \$temp 2(\$fp)	1000111101110110000000000000000010
	sw \$t0 2(\$temp)	1010111101100000000000000000000010
GOTO, L0, -, -	j Label 0	00001000000000000000000000000000101011
GOTO, L2, -, -	j Label 2	00001000000000000000000000000000101010
LABEL, L1, -, -	Label 1:	0000001111111111111100000100000
LOAD, \$t1, v, -	lw \$t1 1(\$fp)	10001111010000100000000000000001
PARAM, \$t1, INT, -	sw \$t1 0(\$pilha)	10101111010000010000000000000000

Continued on next page

Tabela 14: Resultado da geração de códigos para o programa máximo denominador comum (Continued)

LOAD, \$t0, u, -	lw \$t0 0(\$fp)	10001111010000000000000000000000
LOAD, \$t0, u, -	lw \$t0 0(\$fp)	10001111010000000000000000000000
LOAD, \$t1, v, -	lw \$t1 1(\$fp)	10001111010000100000000000000001
DIV, \$t0, \$t1, \$t3	div \$t3 \$t0 \$t1	00000000000000010001100000011010
LOAD, \$t1, v, -	lw \$t1 1(\$fp)	10001111010000100000000000000001
MULT, \$t3, \$t1, \$t4	mult \$t4 \$t3 \$t1	00000000011000010010000000011000
SUB, \$t0, \$t4, \$t5	sub \$t5 \$t0 \$t4	0000000000000001000010100000100010
PARAM, \$t5, INT, -	sw \$t5 1(\$pilha)	10101110100010100000000000000001
CALL, gcd, 2, \$t6	lw \$temp 1(\$pilha)	10001110101101100000000000000001
	sw \$temp 2(\$sp)	10101110011011000000000000000010
	lw \$temp 0(\$pilha)	10001110101101100000000000000000
	sw \$temp 1(\$sp)	10101110011011000000000000000001
	add \$temp \$fp \$zero	000000110111111011000001000000000
	addi \$temp \$temp 2	0010001101111011000000000000000010
	sw \$temp 3(\$sp)	101011100110110000000000000000011
	addi \$fp \$fp 25	0010001101111010000000000000000011001
	addi \$sp \$sp 25	0010001100111000000000000000000011001
	jal gcd	00001100000000000000000000000000000001
	subi \$fp \$fp 25	0010011101111010000000000000000011001
	subi \$sp \$sp 25	0010011100111000000000000000000011001
	lw \$t6 4(\$fp)	10001111010011000000000000000000100
RET, \$t6, -, -	lw \$temp 2(\$fp)	100011110111011000000000000000010
	sw \$t6 2(\$temp)	101011101100110000000000000000010
GOTO, L0, -, -	j Label 0	00001000000000000000000000000000101011
LABEL, L2, -, -	Label 2:	00000011111111111111100000100000
LABEL, L0, -, -	Label 0:	00000011111111111111100000100000
END, gcd, -, -	lw \$ra 3(\$fp)	1000111101111100000000000000000011
	jr \$zero \$ra \$zero	000000111011111111100000001000
FUN, VOID, main, 0	main:	0000001111111111111100000100000

Continued on next page

Tabela 14: Resultado da geração de códigos para o programa máximo denominador comum (Continued)

Continued on next page

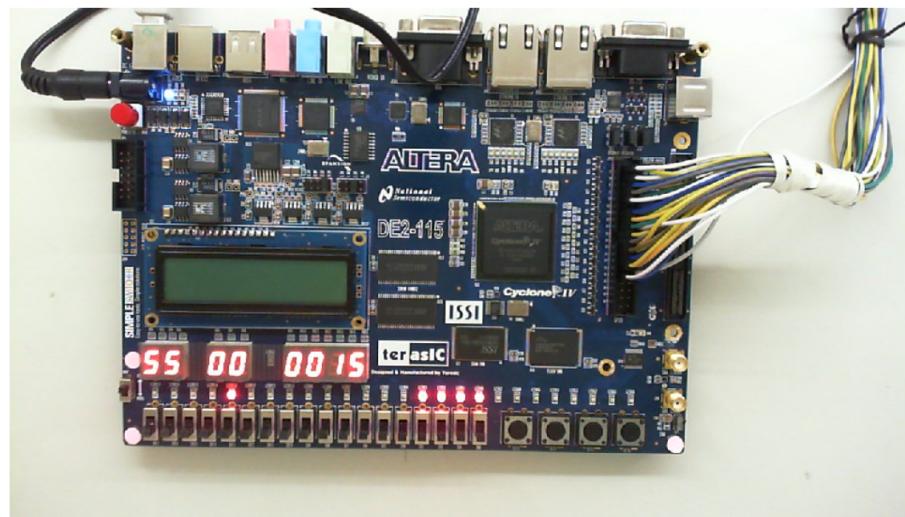
Tabela 14: Resultado da geração de códigos para o programa máximo denominador comum (Continued)

	lw \$t11 2(\$fp)	10001111010101100000000000000010
PARAM, \$t11, INT, -	sw \$t11 0(\$pilha)	10101111010010110000000000000000
CALL, output, 1, -	lw \$temp 0(\$pilha)	10001111010110110000000000000000
	out \$zero \$temp 0	01111011011111100000000000000000
LABEL, L3, -, -	Label 3:	0000001111111111111100000100000
END, main, -, -	-	-
HALT, -, -, -	halt \$zero	11111111111111111111111111111111

Fonte: O Autor

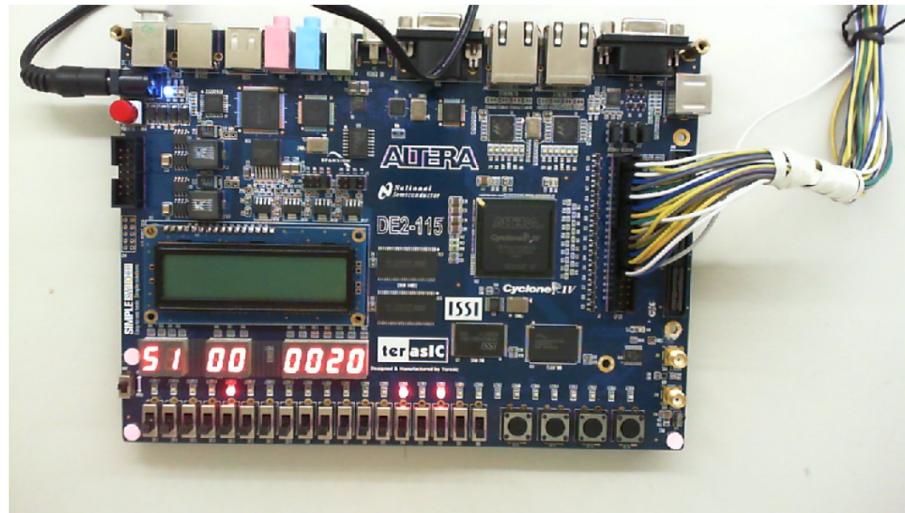
#### 4.2.3 Teste FPGA

Figura 33 – Inserindo como primeiro valor o número 15 para o programa do máximo divisor comum



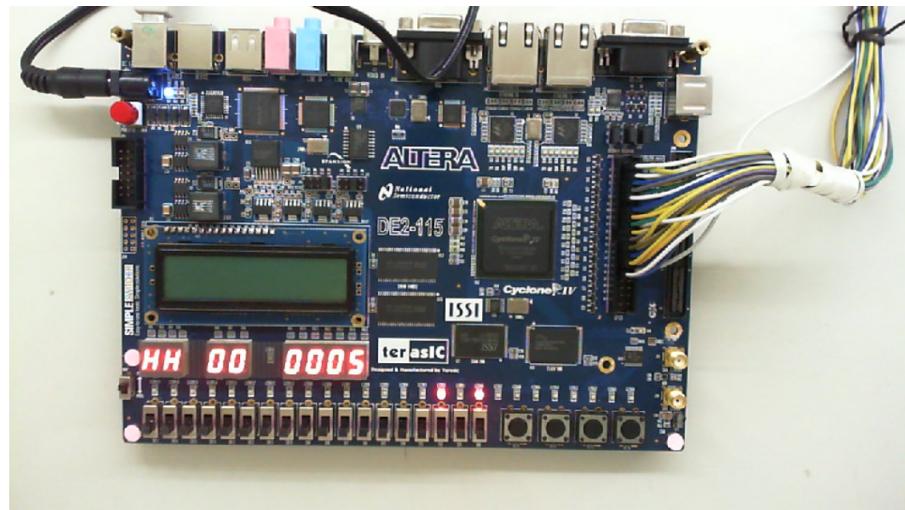
Fonte: O Autor

Figura 34 – Inserindo como segundo valor o número 20 para o programa do máximo divisor comum



Fonte: O Autor

Figura 35 – Número 5 obtido como resultado após o cálculo do máximo divisor comum



Fonte: O Autor

## 4.3 Ordenação de Vetores

### 4.3.1 Código Fonte

```

1 int vet[5];
2
3 int minloc (int a[], int low, int high){
4     int i; int x; int k;
5     k = low;
6     x = a[low];
7     i = low + 1;

```

```

8     while (i < high){
9         if (a[i] < x){
10             x = a[i];
11             k = i;
12         }
13         i = i + 1;
14     }
15     return k;
16 }
17
18 void sort(int a[], int low, int high){
19     int i; int k;
20     i = low;
21     while (i < high-1){
22         int t;
23
24         k = minloc(a,i,high);
25         t = a[k];
26         a[k] = a[i];
27         a[i] = t;
28         i = i + 1;
29     }
30 }
31
32 void main(void){
33     int i;
34
35     i = 0;
36     while(i < 5){
37         vet[i] = input();
38         i = i + 1;
39     }
40
41     sort(vet,0,5);
42
43     while (1){
44         output(vet[0]);
45         output(vet[1]);
46         output(vet[2]);
47         output(vet[3]);
48         output(vet[4]);
49     }
50 }
```

### 4.3.2 Código Intermediário

Tabela 15: Código Intermediário

ALLOC, vet, global, 5
FUN, INT, minloc, 3
ARG, VET, a, minloc

Continued on next page

Tabela 15: Código In-  
termediário  
(Continued)

ARG, INT, low, minloc
ARG, INT, high, minloc
LOAD, \$t0, a, -
LOAD, \$t1, low, -
LOAD, \$t2, high, -
ALLOC, i, minloc, -
ALLOC, x, minloc, -
ALLOC, k, minloc, -
LOAD, \$t3, k, -
LOAD, \$t1, low, -
ASSIGN, \$t3, \$t1, -
STORE, k, \$t3, -
LOAD, \$t4, x, -
LOAD, \$t1, low, -
LOAD, \$t5, a, \$t1
ASSIGN, \$t4, \$t5, -
STORE, x, \$t4, -
LOAD, \$t6, i, -
LOAD, \$t1, low, -
LOADI, \$t7, 1, -
ADD, \$t1, \$t7, \$t8
ASSIGN, \$t6, \$t8, -
STORE, i, \$t6, -
LABEL, L1, -, -
LOAD, \$t6, i, -
LOAD, \$t2, high, -
LT, \$t6, \$t2, \$t9
IFF, \$t9, L2, -

Continued on next page

Tabela 15: Código Intermediário  
(Continued)

LOAD, \$t6, i, -
LOAD, \$t10, a, \$t6
LOAD, \$t4, x, -
LT, \$t10, \$t4, \$t11
IFF, \$t11, L3, -
LOAD, \$t4, x, -
LOAD, \$t6, i, -
LOAD, \$t12, a, \$t6
ASSIGN, \$t4, \$t12, -
STORE, x, \$t4, -
LOAD, \$t3, k, -
LOAD, \$t6, i, -
ASSIGN, \$t3, \$t6, -
STORE, k, \$t3, -
LABEL, L3, -, -
LOAD, \$t6, i, -
LOAD, \$t6, i, -
LOADI, \$t13, 1, -
ADD, \$t6, \$t13, \$t14
ASSIGN, \$t6, \$t14, -
STORE, i, \$t6, -
GOTO, L1, -, -
LABEL, L2, -, -
LOAD, \$t3, k, -
RET, \$t3, -, -
GOTO, L0, -, -
LABEL, L0, -, -
END, minloc, -, -

Continued on next page

Tabela 15: Código Intermediário  
(Continued)

FUN, VOID, sort, 3
ARG, VET, a, sort
ARG, INT, low, sort
ARG, INT, high, sort
LOAD, \$t15, a, -
LOAD, \$t16, low, -
LOAD, \$t17, high, -
ALLOC, i, sort, -
ALLOC, k, sort, -
LOAD, \$t18, i, -
LOAD, \$t16, low, -
ASSIGN, \$t18, \$t16, -
STORE, i, \$t18, -
LABEL, L6, -, -
LOAD, \$t18, i, -
LOAD, \$t17, high, -
LOADI, \$t19, 1, -
SUB, \$t17, \$t19, \$t20
LT, \$t18, \$t20, \$t21
IFF, \$t21, L7, -
ALLOC, t, sort, -
LOAD, \$t22, k, -
PARAM, \$t23, VET, a
LOAD, \$t18, i, -
PARAM, \$t18, INT, -
LOAD, \$t17, high, -
PARAM, \$t17, INT, -
CALL, minloc, 3, \$t24

Continued on next page

Tabela 15: Código In-  
termediário  
(Continued)

ASSIGN, \$t22, \$t24, -
STORE, k, \$t22, -
LOAD, \$t25, t, -
LOAD, \$t22, k, -
LOAD, \$t5, a, \$t22
ASSIGN, \$t25, \$t5, -
STORE, t, \$t25, -
LOAD, \$t22, k, -
LOAD, \$t7, a, \$t22
LOAD, \$t18, i, -
LOAD, \$t8, a, \$t18
ASSIGN, \$t7, \$t8, -
STORE, a, \$t7, \$t22
LOAD, \$t18, i, -
LOAD, \$t9, a, \$t18
LOAD, \$t25, t, -
ASSIGN, \$t9, \$t25, -
STORE, a, \$t9, \$t18
LOAD, \$t18, i, -
LOAD, \$t18, i, -
LOADI, \$t10, 1, -
ADD, \$t18, \$t10, \$t11
ASSIGN, \$t18, \$t11, -
STORE, i, \$t18, -
GOTO, L6, -, -
LABEL, L7, -, -
LABEL, L5, -, -
END, sort, -, -

Continued on next page

Tabela 15: Código Intermediário  
(Continued)

FUN, VOID, main, 0
ALLOC, i, main, -
LOAD, \$t12, i, -
ASSIGN, \$t12, \$t31, -
STORE, i, \$t12, -
LABEL, L9, -, -
LOAD, \$t12, i, -
LOADI, \$t13, 5, -
LT, \$t12, \$t13, \$t14
IFF, \$t14, L10, -
LOAD, \$t12, i, -
LOAD, \$t19, vet, \$t12
CALL, input, 0, \$t20
ASSIGN, \$t19, \$t20, -
STORE, vet, \$t19, \$t12
LOAD, \$t12, i, -
LOAD, \$t12, i, -
LOADI, \$t21, 1, -
ADD, \$t12, \$t21, \$t23
ASSIGN, \$t12, \$t23, -
STORE, i, \$t12, -
GOTO, L9, -, -
LABEL, L10, -, -
PARAM, \$t24, VET, vet
PARAM, \$t31, INT, -
LOADI, \$t5, 5, -
PARAM, \$t5, INT, -
CALL, sort, 3, -

Continued on next page

Tabela 15: Código Intermediário  
(Continued)

LABEL, L11, -, -
LOADI, \$t7, 1, -
IFF, \$t7, L12, -
LOAD, \$t8, vet, \$t31
PARAM, \$t8, INT, -
CALL, output, 1, -
LOADI, \$t9, 1, -
LOAD, \$t10, vet, \$t9
PARAM, \$t10, INT, -
CALL, output, 1, -
LOADI, \$t11, 2, -
LOAD, \$t13, vet, \$t11
PARAM, \$t13, INT, -
CALL, output, 1, -
LOADI, \$t14, 3, -
LOAD, \$t19, vet, \$t14
PARAM, \$t19, INT, -
CALL, output, 1, -
LOADI, \$t20, 4, -
LOAD, \$t21, vet, \$t20
PARAM, \$t21, INT, -
CALL, output, 1, -
GOTO, L11, -, -
LABEL, L12, -, -
LABEL, L8, -, -
END, main, -, -
HALT, -, -, -

Fonte: O Autor

#### 4.3.3 Código Assembly e Binário

Tabela 16: Geração dos códigos assembly e binário

Continued on next page

Tabela 16: Geração dos códigos assembly e binário (Continued)

add \$temp \$temp \$t6	00000011011001101101100000100000
lw \$t10 0(\$temp)	10001111011010100000000000000000
lw \$t4 10(\$fp)	10001111010010000000000000000000
slt \$t11 \$t10 \$t4	00000001010001000101100000101010
beq \$t11 \$zero Label 3	00010011110101100000000000000000
lw \$t4 10(\$fp)	10001111010010000000000000000000
lw \$t6 9(\$fp)	10001111010011000000000000000000
lw \$temp 0(\$fp)	10001111011011000000000000000000
add \$temp \$temp \$t6	00000011011001101101100000100000
lw \$t12 0(\$temp)	10001111011011000000000000000000
add \$t4 \$zero \$t12	00000011110110000100000000000000
sw \$t4 10(\$fp)	10101111010010000000000000000000
lw \$t3 11(\$fp)	10001111010001100000000000000000
lw \$t6 9(\$fp)	10001111010011000000000000000000
add \$t3 \$zero \$t6	0000001111001100001100000100000
sw \$t3 11(\$fp)	10101111010001100000000000000000
Label 3:	0000001111111111111100000100000
lw \$t6 9(\$fp)	10001111010011000000000000000000
lw \$t6 9(\$fp)	10001111010011000000000000000000
ori \$t13 \$zero 1	00110111110110100000000000000000
add \$t14 \$t6 \$t13	00000000110011010111000000100000
add \$t6 \$zero \$t14	0000001111011100011000000100000
sw \$t6 9(\$fp)	10101111010011000000000000000000
j Label 1	00001000000000000000000000000000
Label 2:	0000001111111111111100000100000
lw \$t3 11(\$fp)	10001111010001100000000000000000
lw \$temp 3(\$fp)	10001111011101100000000000000000
sw \$t3 2(\$temp)	10101111011000110000000000000000
j Label 0	00001000000000000000000000000000
Label 0:	0000001111111111111100000100000

Continued on next page

Tabela 16: Geração dos códigos assembly e binário (Continued)

Continued on next page

Tabela 16: Geração dos códigos assembly e binário (Continued)

Continued on next page

Tabela 16: Geração dos códigos assembly e binário (Continued)

Continued on next page

Tabela 16: Geração dos códigos assembly e binário (Continued)

Continued on next page

Tabela 16: Geração dos códigos assembly e binário (Continued)

Continued on next page

Tabela 16: Geração dos códigos assembly e binário (Continued)

lw \$temp 0(\$pilha)	10001111010110110000000000000000
out \$zero \$temp 0	01111011011111100000000000000000
ori \$t14 \$zero 3	00110111110111000000000000000011
addi \$temp \$zero 0	00100011111101100000000000000000
add \$temp \$temp \$t14	00000011011011101100000100000
lw \$t19 0(\$temp)	10001111011100110000000000000000
sw \$t19 0(\$pilha)	10101111010100110000000000000000
lw \$temp 0(\$pilha)	10001111010110110000000000000000
out \$zero \$temp 0	01111011011111100000000000000000
ori \$t20 \$zero 4	0011011111101000000000000000100
addi \$temp \$zero 0	00100011111101100000000000000000
add \$temp \$temp \$t20	00000011011101001101100000100000
lw \$t21 0(\$temp)	10001111011101010000000000000000
sw \$t21 0(\$pilha)	10101111010101010000000000000000
lw \$temp 0(\$pilha)	10001111010110110000000000000000
out \$zero \$temp 0	01111011011111100000000000000000
j Label 11	0000100000000000000000000000000010111101
Label 12:	00000011111111111111000000100000
Label 8:	00000011111111111111000000100000
halt \$zero	11111111111111111111111111111111

Fonte: O Autor

## 5 Considerações Finais

A realização de projetos em arquitetura e organização de computadores são de extrema importância para conseguirmos colocar em prática muito dos conceitos teóricos vistos em aula, permitindo uma evolução do pensamento técnico do aluno. Como engenheiros da computação, é imprescindível que sejamos capazes de planejar, desde o *design* de uma arquitetura de processador, até a sua devida implementação. Além disso, ao sair de uma zona de conforto, o aluno deve buscar os conhecimentos teóricos que faltam para conseguir realizar tal desafio.

Além disso, ao projetar um compilador desde sua fase de análise até a de síntese é possível entender profundamente como o programa que é utilizado diariamente pelos estudantes na graduação de computação realiza a análise e gera os códigos executáveis dos computadores atualmente. O desenvolvimento desse *software* é extremamente trabalhoso e demanda do aluno a conexão e melhoramento de diversos outros projetos realizados em disciplinas passadas, porém o seu resultado é grandioso para o desenvolvimento das habilidades de programação e de entendimento do que o código escrito na linguagem de programação realmente significa para a máquina.

Por fim, nota-se após a realização de testes, com a utilização da placa FPGA disponibilizada de forma online e presencial, que o compilador consegue transformar o código de alto nível para código binário capaz de efetuar as operações pedidas pelo usuário, considerando também as limitações impostas pelo próprio processador e pela linguagem de programação C-.

## Referências

LOUDEN, K. C. *Compiler construction: principles and practice*. 1. ed. [S.l.]: Cengage Learning, 1997. Citado na página [7](#).

PATTERSON, D. *Organização e projeto de computadores: a interface hardware/software*. 5. ed. Rio de Janeiro: GEN LTC, 2017. Citado 2 vezes nas páginas [8](#) e [9](#).

PATTERSON, D. A.; HENNESSY, J. L. *Computer architecture: a quantitative approach*. 3. ed. [S.l.]: Morgan Kaufmann, 2006. Citado 2 vezes nas páginas [8](#) e [12](#).