Matheus Augusto de Castro Santos

Implementação do Compilador C-

São José dos Campos - Brasil

Outubro de 2020

Matheus Augusto de Castro Santos

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: Prof. Dr. Luiz Eduardo Galvão Martins

Universidade Federal de São Paulo - UNIFESP

Instituto de Ciência e Tecnologia - Campus São José dos Campos

São José dos Campos - Brasil

Outubro de 2020

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

Com o avanço das tecnologias para construção de computadores, instruções de máquina cada vez mais complexas foram sendo implementadas nos processadores, que permitem a execução de programas altamente complexos. Contudo apenas a existência dessas instruções não torna viável a implementação de programas pois a escrita em código de máquina é algo bastante trabalhoso. Por esse motivo surgiram os compiladores, ferramentas capazes de transformar um código mais alto nível e de mais fácil entendimento e escrita para a linguagem entendida pelas máquinas, o assembly. Segundo (1) compiladores são programas que traduzem um programa escrito em uma linguagem para outra. Eles recebem como entrada um arquivo que contém um código escrito na linguagem fonte e fazem a tradução para um código equivalente escrito na linguagem alvo. Por esses motivos os compiladores são um dos principais motivos pelos quais a computação existe como a conhecemos atualmente, onde podemos escrever um programa em uma linguagem de alto nível sem a necessidade de se preocupar em como o computador irá executa-lo ou quais instruções serão necessárias. Isto também permite que um código escrito possa ser executado em máquinas diferentes.

O principal objetivo deste projeto consiste no desenvolvimento de um compilador da linguagem C-, que trata-se de um subconjunto da linguagem C, para o processador desenvolvido na disciplina de Laboratorios de Sistemas Computacionais: Arquitetura e Organização de Computadores que é baseado na Arquietura MIPS Monociclo com conjunto de instruções RISC. Será desenvolvido o módulo de análise de código fonte, analisando os aspectos léxicos, sintáticos e semânticos. Após a fase de análise será executada a fase síntese, na qual o código fonte será então traduzido para o assembly da arquitetura desenvolvida.

No Capítulo 2 será apresentado o processador desenvolvido, seus componentes, o conjunto de instruções e a organização de sua memória. No Capítulo 3 será apresentado o desenvolvimento do módulo de análise do compilador, compreendendo a análise léxica, sintática e semântica. No Capítulo 4 será apresentada toda de síntese do compilador desde o código intermediário até o código binário, em seguida, no Capítulo 5 são apresentados exemplos de códigos compilados e por fim no Capítulo 6 a conclusão do projeto.

2 O Processador

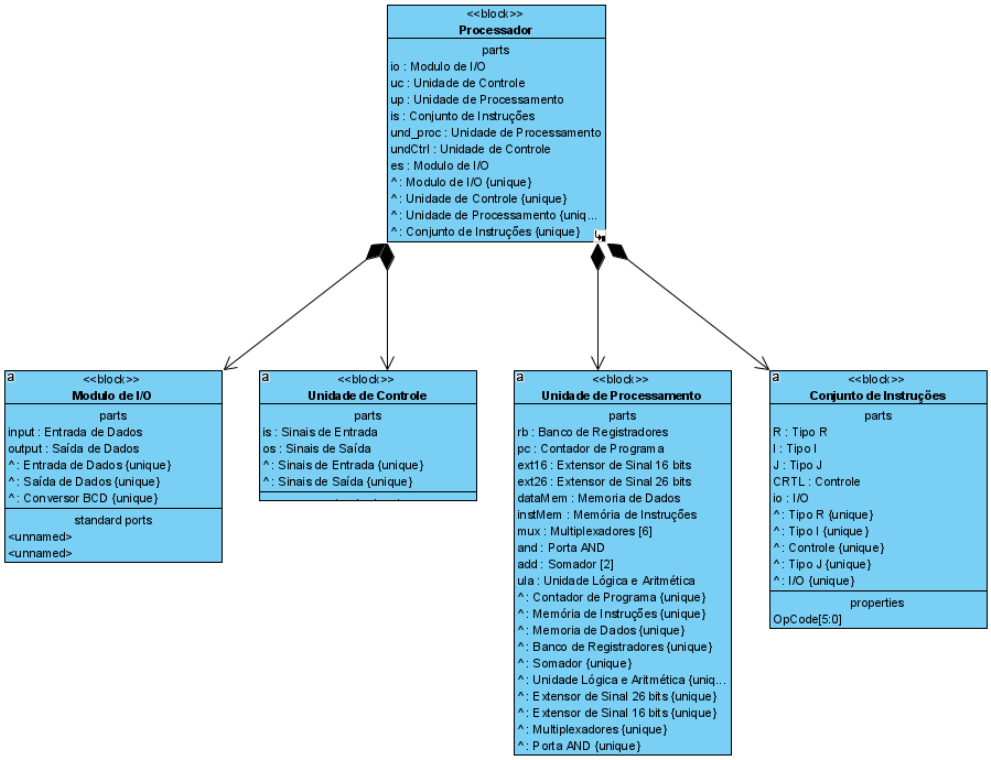
Neste capitulo será apresentada a arquitetura e o conjunto de instruções do processador desenvolvido na disciplina de Laboratório de Sistemas Computacionais: Arquitetura e Organização de Computadores. O processador implementado é baseado na arquitetura MIPS Uniciclo na qual cada instrução é executada em um ciclo de clock e a duração deste ciclo é definida pelo instrução mais longa. Seu conjunto de instrução segue a abordagem RISC, na qual é implementado número reduzido de instruções com modos de endereçamento simplificados(2).

2.1 Diagrama de Blocos do Processador

Nesta seção serão apresentados os diagramas de blocos do processador. Na Figura 1 é apresentado o diagrama de blocos do processador com os seus componentes, o módulo de IO, a Unidade de Controle, a Unidade de Processamento e o Conjunto de Instruções.

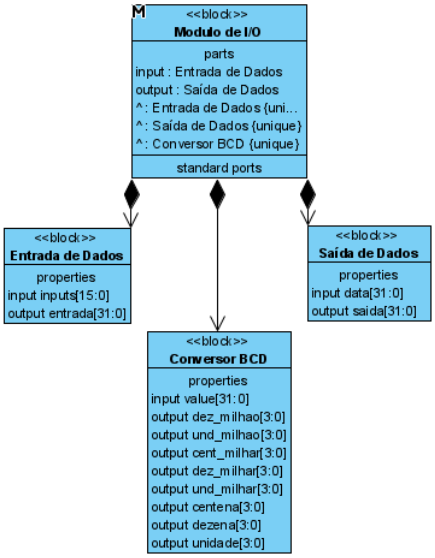
A Figura 2 apresenta composição do Módulo de Entrada e Saída do processador responsável por fazer a comunicação com o ambiente externo. Na Figura 3 é apresentado o diagrama de blocos com os componentes da unidade de processamento. Por fim, a Figura 4 apresenta a composição da Unidade de Controle na forma de blocos.

Figura 1 – Diagrama de Blocos do Processador



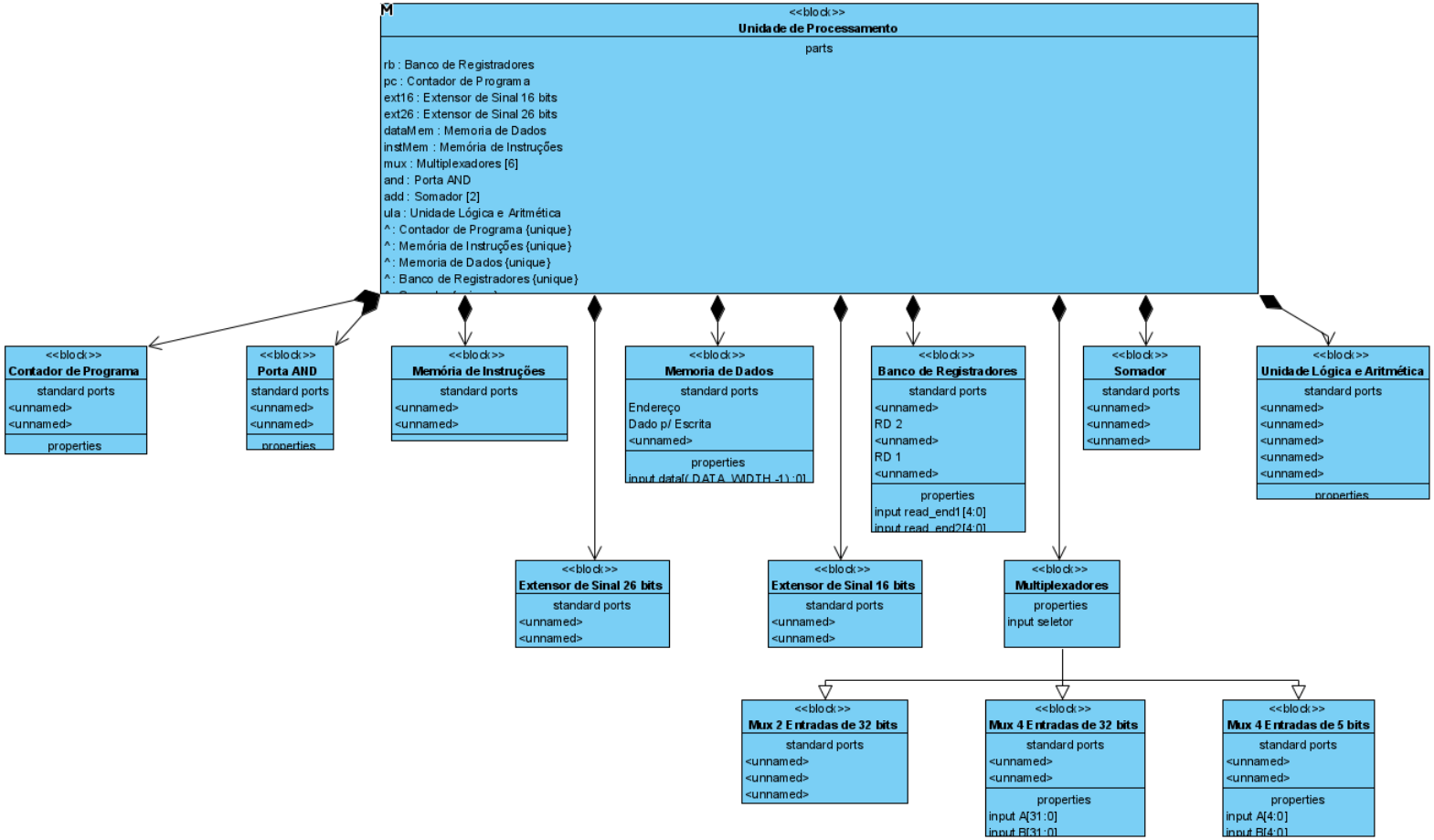
Fonte: Autoria Própria

Figura 2 – Diagrama de Blocos do Módulo de IO



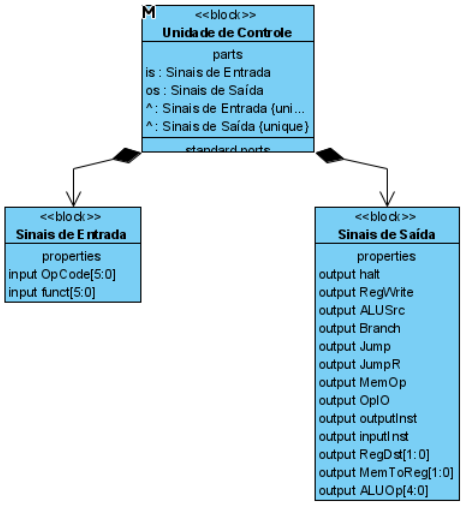
Fonte: Autoria Própria

Figura 3 – Diagrama de Blocos da Unidade de Processamento



Fonte: Autoria Própria

Figura 4 – Diagrama de Blocos da Unidade de Controle

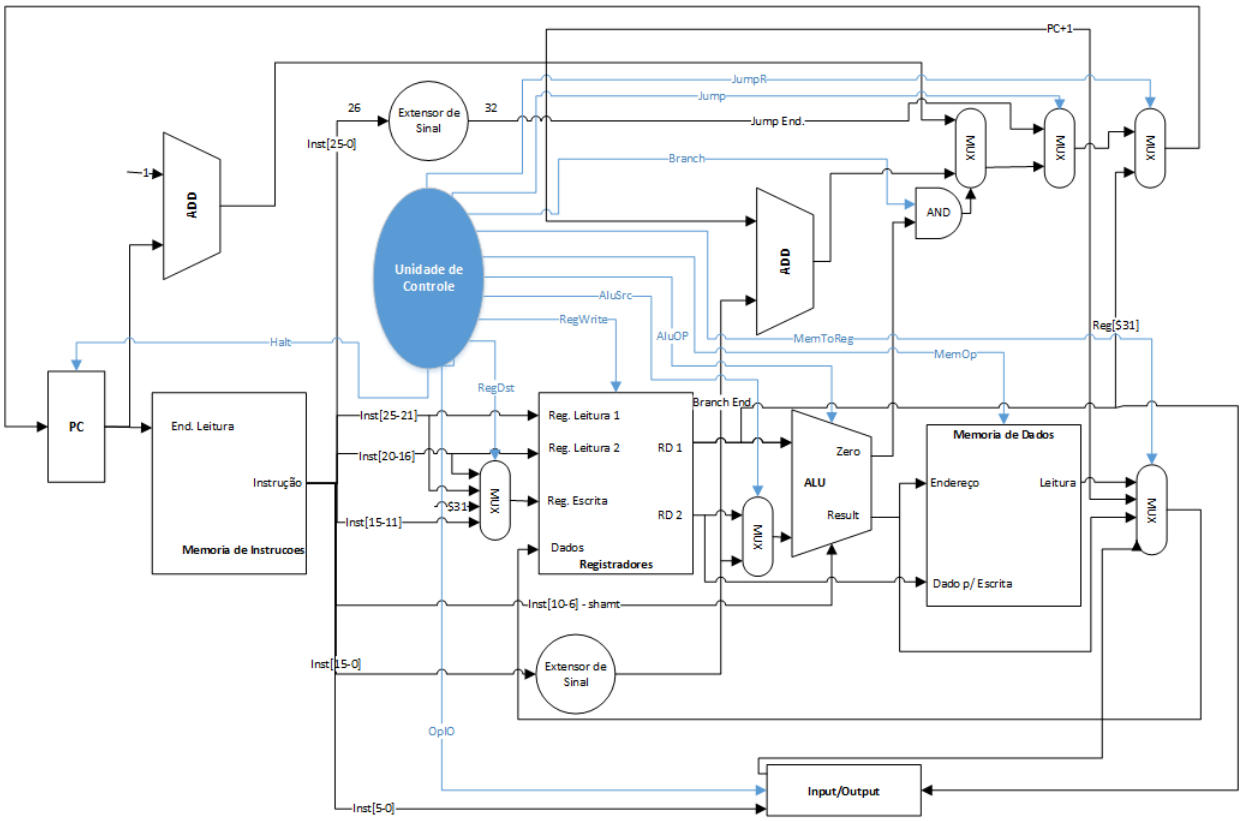


Fonte: Autoria Própria

2.2 Caminho de Dados

Para a construção do processador foi projetado o caminho de dados apresentado na Figura 5. No diagrama estão presentes todos os módulos necessários para a execução das instruções, como a ULA, a Unidade de Controle, memória de instrução para armazenar as instruções que serão executadas, a memória de dados, somadores, multiplexadores, contador de programa que indica qual instrução será executada, o banco de registradores, o módulo de entrada e saída e os extensores de sinal. Na Figura 5 são destacados na cor azul os sinais de controle enviados pelo Unidade de Controle para gerenciar as operações realizadas por cada componente.

Figura 5 – Caminho de dados do Processador



Fonte: Autoria Própria

2.3 Componentes

Nesta seção serão apresentados os componentes do caminho de dados do processador bem como seu funcionamento. Os componentes foram implementados por meio da linguagem de descrição de circuitos *Verilog* através da ferramenta *Quartus Prime*.

2.3.1 Banco de Registradores

O banco de registradores possui 32 registradores de 32 bits. Este módulo possui duas entradas que recebem os endereços de dois registradores especificados pela instrução atual que serão lidos e seus valores enviados para as duas saídas, e uma entrada para especificar o endereço do registrador de escrita, este que pode ser especificado pela instrução corrente ou ser o registrador 31 reservado para armazenar o endereço de retorno utilizado pela instrução *Jump Register*, no qual será escrito o valor recebido na entrada de

dados. Durante um ciclo de execução de uma instrução os valores lidos são disponibilizados nas saídas de maneira assíncrona. Já no caso da escrita, durante a borda de subida do sinal de *clock* o módulo recebe o endereço de escrita para que na borda de descida do sinal de *clock* seja de fato realizada a escrita.

2.3.2 Unidade Lógica e Aritmética

A Unidade Lógica e Aritmética(ULA) é responsável por executar as instruções lógicas, aritméticas, comparação e deslocamento de bits. O módulo recebe através de suas entradas os dois operandos que serão utilizados e o sinal de controle *AluOp* que especifica qual operação deve ser realizada sobre os operados. Após a execução da operação seu resultado é disponibilizado na saída *Zero* no caso de uma instrução desvio condicional para indicar se o desvio deve ou não ser efetuado e para caso das demais instruções o resultado enviado para a saída *result*. No caso das instruções de deslocamento a ULA também recebe através da porta *shamt* a quantidade de bits a serem deslocados. Em todas as operações aritméticas podem ser utilizados números positivos e negativos de 32 bits, contudo no caso da operação de multiplicação a quantidade de bits fica limitada a 16 bits, devido ao fato de que na multiplicação de dois números com mais de 16 bits não seria possível armazenar o resultado em um registrador. A Tabela 1 apresenta os possíveis valores de *AluOp* e suas funções.

Tabela 1 – OpCodes da ULA

ALUOp	Operação
00000	Soma
00001	Subtração
00010	Divisão
00011	Multiplicação
00100	And bit a bit
00101	Or bit a bit
00110	Nor
00111	Menor que(1 se sim, 0 se não)
01000	Desloca para esquerda
01001	Desloca para direita
01010	Menor que zero(1 se sim, 0 se não)
01011	Maior que zero(1 se sim, 0 se não)
01100	Igual a zero(1 se sim, 0 se não)
01101	Se igual(1 se sim, 0 se não)
01110	Se diferente(1 se sim, 0 se não)
01111	Resto
10000	Concatena 16 bits superiores de um operando com os 16 inferiores do outro
10001	Maior que(1 se sim, 0 se não)
10010	Menor/igual que(1 se sim, 0 se não)
10011	Maior/igual que(1 se sim, 0 se não)
10100	Operandos iguais(1 se sim, 0 se não)
10101	Operandos diferentes(1 se sim, 0 se não)

Fonte: Autoria Própria

2.3.3 Contador de Programa

O Contador de Programa(PC) consiste de um registrador de propósito especifico para armazenar o endereço da instrução que deve ser lida e executada pelo processador. Este módulo módulo recebe como entrada o endereço da próxima instrução a ser executada, podendo ser o valor atual incrementado em 1 ou então um valor especificado por uma instrução de desvio ou salto, que será armazenado e disponibilizado na saída. Através desse módulo também realizada a paralisação processador que ocorre quando são executadas as instruções *input*, *output* ou *halt*. Neste caso o PC recebe um sinal chamado *halt* que quando em estado alto indica que o endereço da instrução não deve ser atualizado até que o sinal *halt* retorne para estado baixo.

2.3.4 Somadores

Os Somadores são utilizados para fazer um incremento do contador de programa e também nas instruções de desvio em que o valor especificado no campo imediato é somado ao PC para indicar o endereço da próxima instrução.

2.3.5 Módulo de Entrada e Saída

O Módulo de Entrada e Saída é responsável por receber e enviar dados do ambiente externo. Este módulo recebe como entrada o sinal de controle OpIO enviado pela Unidade de Controle para que se possa identificar se irá ocorrer uma operação de saída ou entrada, no caso de entrada o campo funct da instrução é utilizado para identificar a origem da entrada. O módulo de E/S é conectado aos *push-buttons* e chaves do kit FPGA. No caso de uma operação de entrada a execução da instrução é interrompida para aguardar que o usuário informe o valor de entrada. O valor de entrada especificado é armazenado em um registrador interno do módulo e quando o usuário confirma a entrada este registrador é lido e tem seu valor armazenado no banco de registradores. Para o caso de uma instrução de saída o módulo recebe um valor lido do banco de registradores que é escrito em um registrador interno durante a borda de descida do sinal de *clock* e enviado ao conversor BCD de modo assíncrono.

2.3.5.1 Conversor BCD

Este módulo é responsável por fazer a conversão do valor binário enviado pelo módulo de entrada e saída para a representação em casas decimais. O módulo contém as saídas dez_milhao, und_milhao, cent_milhar, dez_milhar, und_milhar, centena, dezena, unidade as quais irão fornecer o valor binário de cada digito do valor recebido como entrada pelo módulo. Antes de realizar a conversão é necessário verificar se o valor recebido é positivo ou negativo, para o caso de ser negativo uma flag é ativada indicando ao usuário que o valor apresentado é negativo e é feita a conversão do formato de Complemento de 2 para a forma binária normal.

2.3.6 Display de Sete Segmentos

Este módulo é responsável por realizar a conversão de uma valor binário de 4 bits entre 0 e 9 para a representação em sete segmentos. Para cada display presente no kit FPGA foi implementado um conversor que recebe uma das saídas do conversor BCD e envia ao display a representação correspondente em sete segmentos.

2.3.7 Extensores de Sinal

Alguns dados e endereços que percorrem o caminho dados possuem tamanhos inferiores a 32 bits, o que faz com que seja necessaria a extensão do sinal desses componentes. Por exemplo para o caso do imediato de 16 bits nas instruções do formato I é necessário verificar qual o valor do bit mais significativo e estende-lo para o 16 bits superiores para que tenhamos um dado de 32 bits. O mesmo ocorre com endereço de 26 bits nas instruções *jal* e *jump* que também é extendido de 26 bits para 32 bits. Os extensores tem como entrada um dado de 16 ao 26 bits e como saída o sinal extendido para 32 bits.

2.3.8 Multiplexadores

Os multiplexadores são componentes que recebem múltiplas sinais de entrada e de acordo com um sinal seletor um dos sinais de entrada é enviado para a saída.

2.3.9 Memória de Dados

Para a implementação do módulo da Memória de Dados foi utilizado um *template* disponível no *software* Quartus Prime, pois o uso deste template implica na diminuição do tempo de compilação do projeto no Quartus e também permite que quando o projeto for mapeado no Kit FPGA seja utilizada a memória interna do dispositivo. O tamanho dos dados armazenados é de 32 bits com 1024 posições que podem ser endereçadas.

2.3.10 Memoria de Instruções

Para armazenar as instruções à serem executadas pelo processador foi implementada a Memoria de Instruções, também utilizando um *template* utilizado no Quartus Prime. A memória é constituida por 256 posições de 32 bits.

2.3.11 Unidade de Controle

A Unidade Controle é o módulo do processador responsável por gerenciar o funcionamento de todas os demais módulos. Para fazer esse gerenciamento a Unidade de Controle recebe como entrada os campos *OpCode* e *funct* da instrução que será executada, em seguida de acordo com o valor dessas entradas são envidados para as saídas do módulo os sinais de controle. As principais funções desses sinais de controle serão especificar qual operação lógica/aritmetica a ULA deverá executar, gerenciar qual será o endereço da

próxima instrução que será executada, habilitar a entrada e saída de dados no Módulo de E/S, especificar a origem do dado a ser escrito no banco de registradores e também de onde virá o endereço do registrador de escrita. A Figura 6 apresenta uma tabela como nome, destino e possíveis valores de cada sinal de controle.

Figura 6 – Sinais de Controle

Sinal	Módulo de destino do sinal	Valor
Halt	Contador de Programa	0 - PC continua a contagem de instrução 1- A contagem de instruções é interrompida
RegDst	Banco de Registradores	00 - Registrador de destino vem do campo [20-16] 01 - Registrador de destino vem do campo [25-21] 10 - O registrador de destino é o 31 11 - Registrador de destino vem do campo [15-11]
RegWrite	Banco de Registradores	0 - Não permite escrita no banco de registradores 1 - Permite a escrita no banco de registradores
ALUSrc	Mux da entrada do segundo operando da ULA	0 - Segundo operando vem do banco de registradores 1 - Segundo operando vem do campo imediato da instrução
ALUOp	ULA	00000 à 01111
MemOp	Memória de Dados	0 - Não permite escrita na memória 1 - Permite a escrita na memória
Branch	Porta AND	1 - Instrução de desvio condicional 0 - Demais instruções
MemToReg	Mux da entrada de dados no banco de registradores	00 - Dado para escrita no banco de registradores vem da memória 01 - PC+1 é escrito no banco de registradores 10 - Dado para escrita no banco de registradores vem da ULA 11 - Dado para escrita no banco de registradores vem do módulo de E/S
Jump	Indica uma instrução Jump	0 - Endereço da próxima instrução é PC+1 1 - Endereço da próxima instrução é o imediato extendido
JumpR	Indica uma instrução JumpR	0 - Endereço da próxima instrução é PC+1 1 - Endereço da próxima instrução vem do banco de registradores
OpIO	Permite ou não a escrita no registrador do módulo de E/S	0 - Instrução de entrada de dados 1 - Instrução de saída de dados

Fonte: Autoria Própria

2.4 Conjunto de Instruções

O conjunto de instruções do processador implementado é composto por 45 instruções com tamanho de 32 bits apresentadas na Tabela 3. Esse conjunto foi divido em subconjuntos de instruções de 4 formatos diferentes, sendo eles os formatos R, I, J e Instruções E/S. Os campos de cada formato e seus respectivos tamanhos são apresentados na Tabela 2.

Tabela 2 – Formato das Instruções

Tamanho do Campo	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
Formato R	opcode	rs	rt	rd	shamt	funct
Formato I	opcode	rs	rt	imediato		
Formato J	opcode	endereço				
Instruções de E/S	opcode	rs	Não Utilizado			funct

Fonte: Autoria Própria

O Formato R compreende as instruções lógicas, aritiméticas e de deslocamento, que possuem um único opcode e são especificadas por meio do campo funct. Neste formato os campos rt e rs armazenam os endereços dos registradores que contém os operandos que serão utilizados e o campo rd especifica o registrador que receberá o resultado da operação. O campo shamt é utilizado em operações de deslocamento para especificar a quantidade de bits que devem ser deslocados.

O Formato I compreende as instruções logicas e aritméticas com imediato, instruções de desvio condicional e movimentação de dados. No caso de instruções lógicas e aritméticas o campo rs contém o endereço do registrador que armazena o primeiro operando, o campo rt contém o endereço do registrador que receberá o resultado e o campo imediato contém o segundo operando. Já no caso de instruções de desvio o imediato é utilizado é para calcular o endereço de destino para caso o desvio seja tomado. Por fim para as instruções de movimentação de dados existem duas possibilidades, a primeira é para o caso das instruções *sw* e *lw* em que o imediato e o campo rs são utilizados para calcular o endereço de acesso a memoria e o rt contém o endereço de um registrador que contém um valor a ser armazenado na memoria ou receberá um valor vindo da memoria, já no caso da instrução *lui* o rs contém o endereço do registrador que recebera o valor imediato nos seus 16 bits superiores.

As instruções do formato J são *jump* e *jal* que são utilizadas para saltos e o campo endereço contém o destino do salto.

Por fim, o formato Instruções de E/S compreende as instruções utilizadas para efetuar operações de entrada e saída de dados do processador. Para instrução *output* o campo rs especifica o registrador que contém o valor que deverá ser apresentado nos displays de sete segmentos. Já no caso da instrução *input* o campo rs indica o registrador que irá receber o valor lido do ambiente externo, nesse caso também é necessária a especificação do campo funct para indicar se a leitura será de apenas uma chave ou *push-button*(funct = 0-22), ou das primeiras 8 chaves(funct = 22) ou das 16 chaves(funct

= 23) do kit FPGA.

Tabela 3 – Conjunto de Instruções

OpCode	**Funct**	**Instrução**	**Função**	**Expressão**
000000	000001	add	Adição	R[rd] = R[rs] + R[rt]
000000	000010	sub	Subtração	R[rd] = R[rs] - R[rt]
000000	000011	mult	Multiplicação	R[rd] = R[rs] * R[rt]
000000	000100	div	Divisão	R[rd] = R[rs] / R[rt]
000000	000101	and	AND bit a bit	R[rd] = R[rs] & R[rt]
000000	000110	or	OR bit a bit	R[rd] = R[rs] | R[rt]
000000	000111	nor	NOR bit a bit	R[rd] = (R[rs] | R[rt])
000000	001000	slt	Verifica se um operando é menor	R[rd] = (R[rs]<R[rt]) ? 1 : 0
000000	001001	sll	Deslocamento de bits para esquerda	R[rd] = R[rs]«shamt
000000	001010	srl	Deslocamento de bits para direita	R[rd] = R[rs]»shamt
000000	001011	mod	Resto da Divisão	R[rd] = R[rs] % R[rt]
000000	001100	jr	Salto para R[31]	PC = R[31]
000000	001101	sgt	Verifica se um operando é maior	R[rd] = (R[rs]>R[rt]) ? 1 : 0
000000	001110	slet	Verifica se um operando é menor/igual	R[rd] = (R[rs]<=R[rt]) ? 1 : 0
000000	001111	sget	Verifica se um operando é maior/igual	R[rd] = (R[rs]>=R[rt]) ? 1 : 0
000000	010000	set	Verifica se um operando é igual	R[rd] = (R[rs]==R[rt]) ? 1 : 0
000000	010001	sdt	Verifica se um operando é diferente	R[rd] = (R[rs]!=R[rt]) ? 1 : 0
000001	-	addi	Adição com um imediato	R[rt] = R[rs] + ImSinExt
000010	-	multi	Multiplicação com imediato	R[rt] = R[rs] * ImSinExt
000011	-	divi	Divisão com imediato	R[rt] = R[rs] / ImSinExt
000100	-	andi	AND bit a bit com imediato	R[rt] = R[rs] & ImSinExt
000101	-	bltz	Desvio se menor que zero	if(R[rs]<0) PC = novoPC
000110	-	bgtz	Desvio se maior que zero	if(R[rs]>0) PC = novoPC
000111	-	beqz	Desvio se igual a zero	if(R[rs]==0) PC = novoPC
001000	-	beq	Desvio se igual	if(R[rt]==R[rs]) PC = novoPC
001001	-	bne	Desvio se diferente	if(R[rt]!=R[rs]) PC = novoPC
001010	-	lw	Carrega da memória	R[rt] = M[R[rs]+ImSinExt]
001011	-	sw	Armazena na memoria	M[R[rs]+ImSinExt] = R[rt]
001100	-	ori	OR bit a bit com imediato	R[rt] = R[rs] I ImSinExt
001101	-	slti	Verifica se um operando é menor	R[rt] = (R[rs]<ImSinExt]) ? 1 : 0
001110	-	modi	Resto da divisão por imediato	R[rt] = R[rs] % ImSinExt
010101	-	lui	Carrega imediato nos 16 bits superiores do Reg.	R[rs] = {Imediato, 16'b0}
010110	-	sgti	Verifica se um operando é maior	R[rt] = (R[rs]>ImSinExt]) ? 1 : 0
010111	-	sleti	Verifica se um operando é menor/igual	R[rt] = (R[rs]<=ImSinExt]) ? 1 : 0
011000	-	sgeti	Verifica se um operando é maior/igual	R[rt] = (R[rs]>=ImSinExt]) ? 1 : 0
011001	-	seti	Verifica se um operando é igual	R[rt] = (R[rs]==ImSinExt]) ? 1 : 0
011010	-	sdti	Verifica se um operando é diferente	R[rt] = (R[rs]!=ImSinExt]) ? 1 : 0
001111	-	jump	Salto incondicional	PC = novoPc
010000	-	jal	Realiza salto e salva endereço	R[31] = PC+1; PC = novoPC
010001	-	nop	Ciclo vazio	-
010010	-	halt	Para o processador	-
010011	0-21	input	Entrada de Dados	R[rs] = Módulo de E/S[funct]
010011	10110	input	Entrada de Dados	R[rs] = Módulo de E/S[funct]
010011	10111	input	Entrada de Dados	R[rs] = Módulo de E/S[funct]
010100	-	output	Saída de Dados	Módulo de ES = R[rs]

Fonte: Autoria Própria

2.5 Organização da Memória

A arquitetura desenvolvida tem como base a arquitetura Harvard, desta forma são implementados dois blocos distintos de memória: a memória de instruções e a memória de dados.

Na memória de instruções são armazenadas as instruções do programa em execução e esta é endereçada pela registrador de propósito específico PC que define qual a instrução que deverá ser decodificada e executada.

Já a memoria de dados armazena informações de 32 bits em cada posição e é acessada por meio das instruções *Load Word* e *Store Word* e o seu endereçamento e feito através da soma de um endereço contido em um registrador base especificado com um valor de deslocamento.

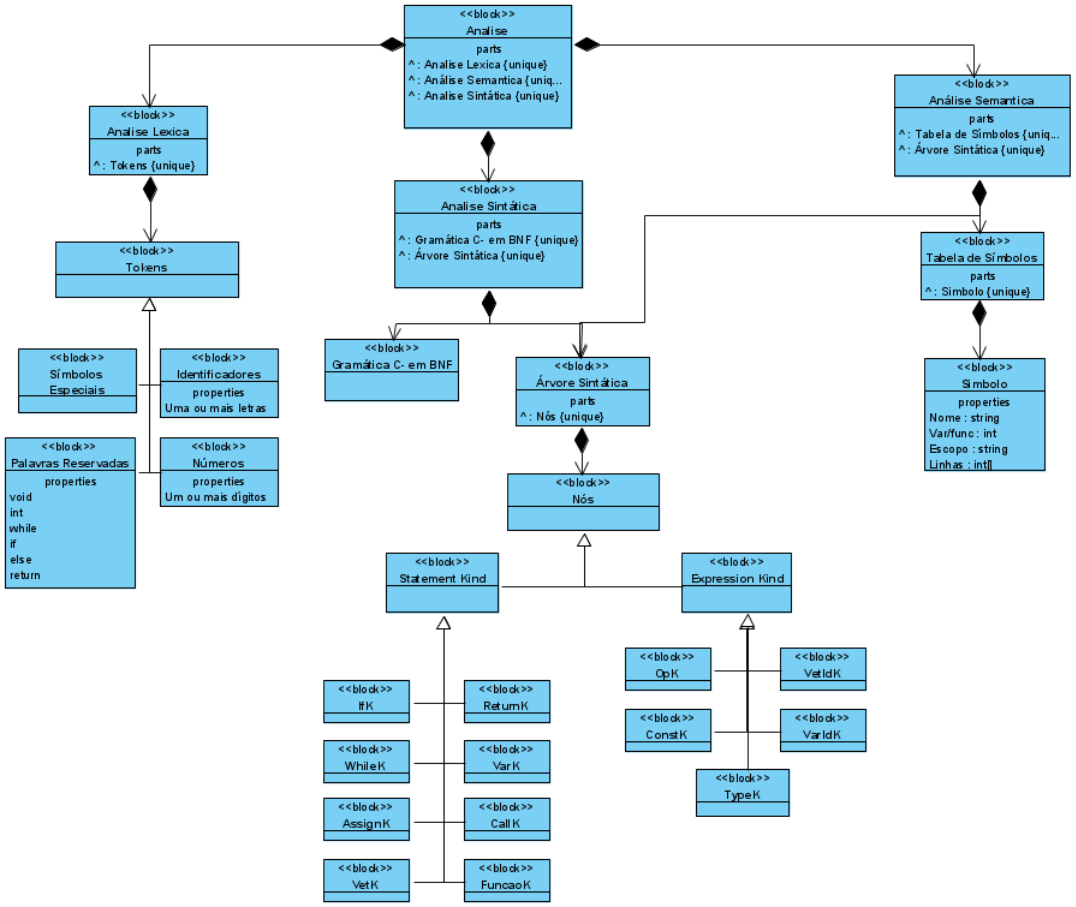
3 Compilador: Fase de Análise

Neste capítulo será apresentada a implementação dos módulos da fase de análise do compilador. Inicialmente serão apresentados diagramas SysML utilizados na modelagem e em seguida a implmentação do módulos de análise léxica, sintática e semântica.

3.1 Modelagem

Para realizar o modelagem do projeto foram empregados os digramas *SysML* de blocos e atividades. A Figura 7 apresenta o diagrama de blocos da Fase de Análise, nele são especificados os componentes de cada uma das etapas. Para a análise léxica seu componente principal são os *tokens* que podem ser agrupados em quatro diferentes tipos, os símbolos especiais, os identificadores, as palavras reservadas e os números. Para análise sintática seus componentes são a gramatica C- em BNF e a arvore de análise sintática composta por nós divididos em dois tipos principais cada um com seus subtipos já apresentados na seção 3.3. A análise semântica possui como componentes a tabela de símbolos e também árvore sintática, já que esta será utilizada para realizar a construção da tabela de símbolos.

Figura 7 – Diagrama de Blocos - Fase Análise

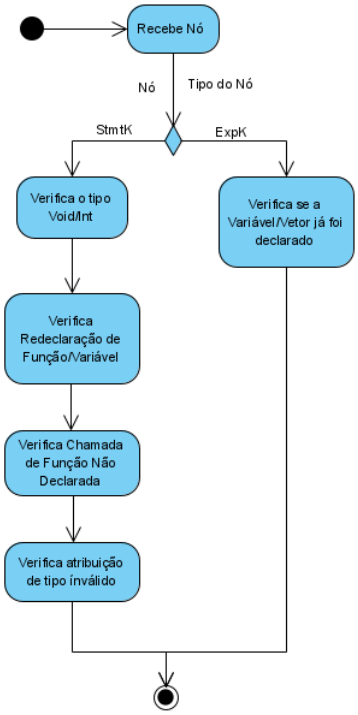


Fonte: Autoria Própria

Na Figura 9 é apresentado o diagrama de atividades da fase de análise sintática. Neste diagrama é apresentado o passo a passo da analise, que tem início com a abertura do arquivo que contém o código em C-, em seguida analisador léxico lê um *token* do arquivo e verifica se é um *token* válido, caso seja ele é enviado ao analisador sintático, caso contrário um aviso de erro e também sintático é emitido e a analise é interrompida. O analisador sintático irá verificar se o *token* recebido é de fim de arquivo, neste caso a arvore sintática é enviada ao analisador semântico e também impressa, caso contrário sera verificado se ele pode satisfazer alguma das regras gramaticais, ou seja, se é um *token* esperado. Nesse caso existem duas opções, caso não seja um *token* esperado é emitido um aviso de erro sintático e a analise é encerrada, caso contrário ele será inserido na pilha e então será verificado se há o casamento dos *tokens* com alguma regra da gramática. Em caso afirmativo os *tokens* são retirados da pilha e um nó é inserido na árvore e um

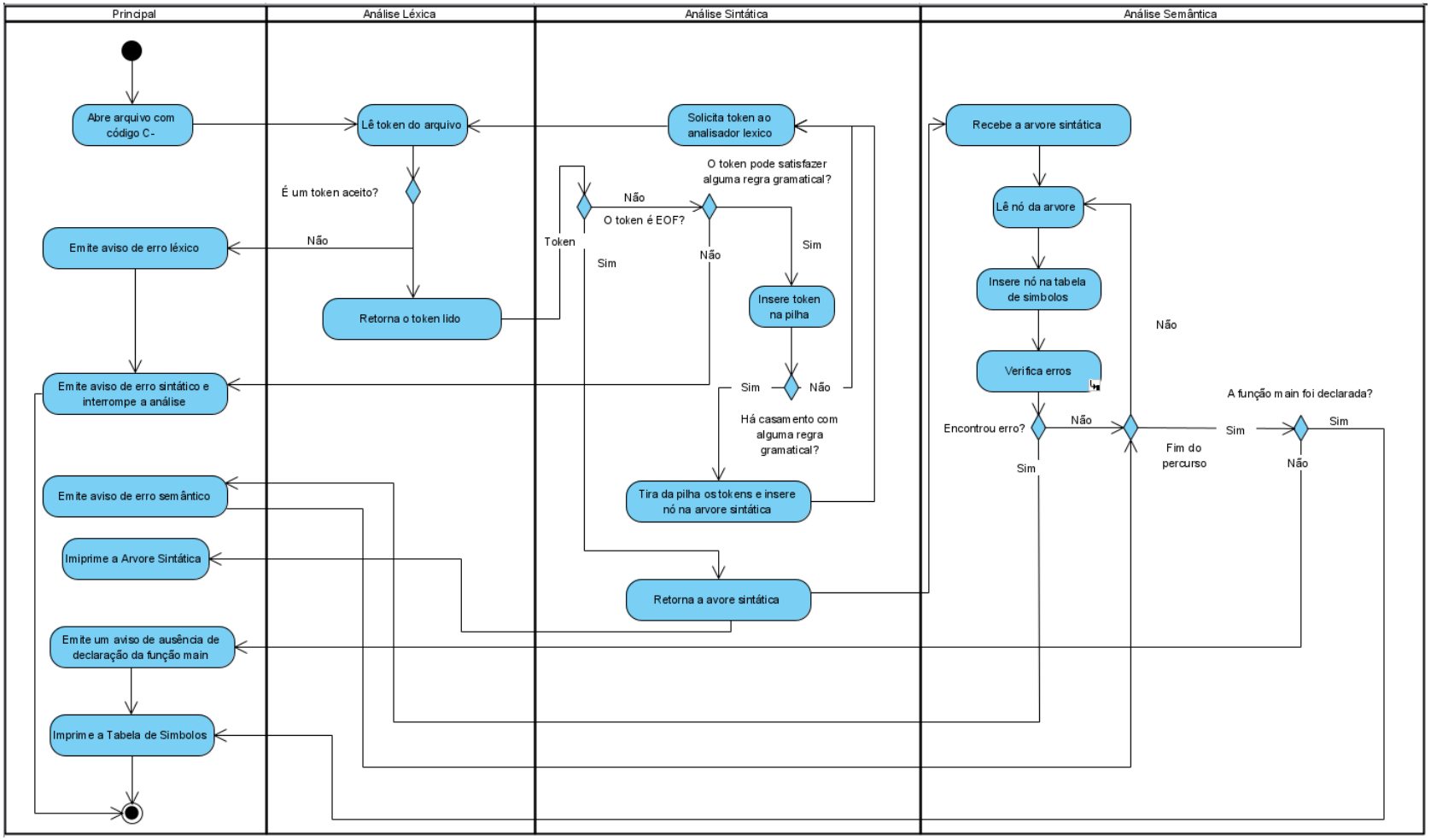
novo *token* é solicitado ao analisador léxico. Da mesma forma se não houver casamento com alguma das regras gramaticais também será solicitado o próximo *token*. Por fim, ao concluir a analise sintática tem início a análise semântica com percurso da árvore sintática. Nesta etapa conforme o percurso na árvore é realizado são verificados os erros semânticos, para esta parte do fluxo foi criado o diagrama apresentado na Figura 8 para apresentar de maneira mais direta a busca pelos erros já apresentados na seção 3.4. Caso seja encontrado algum dos erros um aviso de erro é emitido e a analise prossegue. Ao final do percurso da árvore é verificado se há a declaração da função *main* e então a tabela de símbolos é impressa.

Figura 8 – Diagrama de Atividades - Erros Semânticos



Fonte: Autoria Própria

Figura 9 – Diagrama de Atividades - Fase Análise



Fonte: Autoria Própria

3.2 Análise Léxica

A Análise Léxica é a fase na qual o compilador estará fazendo a leitura dos caracteres do código fonte do programa e irá separá-los em *tokens*. Esses *tokens* representam diferentes informações contidas dentro do código, podendo ser uma palavra-chave da linguagem como por exemplo *while* ou *if*, podem ser identificadores de variáveis ou funções que são constituídos por letras e números e são definidos pelo usuário, e também podem ser símbolos especiais como <, >, ;, =, ! entre outros.

O processo de analise léxica em um compilador é efetuado através da implementação de autômatos finitos a partir de expressões regulares que são empregados no reconhecimento dos *tokens* dada um sequência de caracteres de entrada. A implementação do módulo de análise léxica pode ser efetuada de maneira mais simples através da utilização de um gerador de sistemas de varredura Lex, como por exemplo o Flex que recebe como entrada um arquivo contendo as expressões regulares e tem como saída um arquivo em C com código do procedimento *yylex* que é responsável por retornar os *tokens* do codigo de entrada conforme ele é lido. A implementação desse procedimento é baseada em autômatos finitos determinísticos correspondentes as expressões regulares recebidas como entrada(1).

Para a implementação do módulo de análise léxica foi utilizada a ferramenta Flex abordada anteriormente, na qual as expressões regulares da linguagem C- especificadas no livro (1) apresentadas a seguir foram fornecidas como entrada para o gerador.

- Palavras-chaves: **else if return void while**
- Simbolos Especiais: + - * / < <= > >= == != = ; , () [] /* */
- Marcadores **ID** e **NUM** definidos pelas expressoões regulares:
  - **ID = letra letra***
  - **NUM = digito digito***
  - **letra = a|..|z|A|..|Z|**
  - **digito = 0|..|9**

A especificação das expressões regulares no Flex é apresentada na Figura 10, podemos observar que para cada expressão especificada um *token* é retornado quando há casamento com uma cadeia de caracteres. Os *tokens* reconhecidos são recebidos pelo analisador sintático que dará continuidade à análise.

Figura 10 – Sinais de Controle

```
DIGITO [0-9]
LETRA [a-zA-Z]
ESPACO [ \r\t]+
%%
"void"    return VOID;
"int"     return INT;
"while"   return WHILE;
"if"      return IF;
"else"    return ELSE;
"return"  return RETURN;
"\n" {++lineno;}
"+"   return SOMA;
"-"   return SUB;
"*"   return MUL;
"/"   return DIV;
"="   return ATRIB;
"=="  return IGL;
"!="  return DIF;
">="  return MAIGL;
"<="  return MEIGL;
">"   return MAI;
"<"   return MEN;
","   return VIRG;
";"   return PV;
"("   return APR;
")"   return FPR;
"["   return ACOL;
"]"   return FCOL;
"{"   return ACH;
"}"   return FCH;
"/*"        {
                int c;
                while ( (c = input()) != '*' && c != EOF )
                {
                  if(c=='\n')
                  {
                    lineno = lineno +1;
                  }
                }
                if ( c == '*' )
                {
                    while ( (c = input()) == '*' );
                    if ( c == '/' ) break;
                }
                if ( c == EOF )
                    break;
            }
{LETRA}+   return ID;
{DIGITO}+ return NUM;
{ESPACO}  {};
<<EOF>>   return 0;
({LETRA}|{DIGITO})+ {return ERRO;}
.     {return ERRO;}
```

Fonte: Autoria Própria

Na especificação do arquivo de entrada para o Flex também é definida a função *getToken*, apresentada na Figura 11 na qual será feita a abertura do arquivo que contém o código fonte que será compilado, em seguida é feita a aquisição do *token* através da chamada da função *yylex* e o armazenamento da cadeia de caracteres que casou com a expressão regular, é verificado então se é um *token* de *ERRO*, caso seja o erro é informado ao usuário, e por fim o *token* lido é retornado.

Figura 11 – Função GetToken do Analisador Léxico

```
TokenType getToken(void)
{
  static int primeiraExecucao = 1;
  TokenType tokenAtual;
  if(primeiraExecucao)
  {
    primeiraExecucao = 0;
    lineno++;
    yyin = source;
  }
  tokenAtual = yylex();
  strncpy(tokenString, yytext, MAXTOKENLEN);
  if(tokenAtual == ERRO)
  {
    printf("ERRO LÉXICO: %s Linha: %d \n",yytext,(lineno));
  }
  return tokenAtual;
}
```

Fonte: Autoria Própria

3.3 Análise Sintática

A fase de Análise Sintática de um compilador é voltada para a verificação da estrutura do programa. Isto feito através da aquisição dos *tokens* e a verificação se a combinação deles atende a um conjunto de regras gramaticais definidas por uma Gramática Livre de Contexto. As regras definidas pelas gramáticas livres de contexto são em geral recursivas o que faz com que o processo de análise também seja recursivo. Ao longo desse processo conforme os *tokens* são recebidos as regras gramaticais são verificadas e conforme essas forem atendidas o produto final da analise sintática é construído, a Árvore de Análise Sintática. Os *tokens* recebidos são inseridos em uma pilha e conforme ocorra o casamento de um conjunto de *tokens* presentes nesta pilha são inseridos os nós na arvore sintática. Nesta estrutura de arvore os nós são organizados conforme a estrutura hierárquica presente no código recebido como entrada(1).

Para o desenvolvimento do analisador sintático neste projeto foi utilizado o gerador de *parser* YACC-Bison. Para isto foi necessária a especificação dá gramatica da linguagem C- como entrada para o gerador. A Figura 12 apresenta a Gramática da linguagem C- utilizada. Com a gramatica especificada o YACC-Bison irá gerar o código para a função *yyparse* que será utilizada para iniciar o processo de análise sintática. Para fazer a aquisição dos *tokens* o *parser* utilizará a função *yylex* que invocará a função *getToken* definida no módulo de análise léxica.

Figura 12 – Gramática da Linguagem C-

1. *programa* → *declaração-lista*
2. *declaração-lista* → *declaração-lista declaração* | *declaração*
3. *declaração* → *var-declaração* | *fun-declaração*
4. *var-declaração* → *tipo-especificador* **ID** **;** | *tipo-especificador* **ID** **[** **NUM** **]** **;**
5. *tipo-especificador* → **int** | **void**
6. *fun-declaração* → *tipo-especificador* **ID** **(** *params* **)** *composto-decl*
7. *params* → *param-lista* | **void**
8. *param-lista* → *param-lista* **,** *param* | *param*
9. *param* → *tipo-especificador* **ID** | *tipo-especificador* **ID** **[]**
10. *composto-decl* → **{** *local-declarações statement-lista* **}**
11. *local-declarações* → *local-declarações var-declaração* | *vazio*
12. *statement-lista* → *statement-lista statement* | *vazio*
13. *statement* → *expressão-decl* | *composto-decl* | *seleção-decl* | *iteração-decl* | *retorno-decl*
14. *expressão-decl* → *expressão* **;** | **;**
15. *seleção-decl* → **if** **(** *expressão* **)** *statement* | **if** **(** *expressão* **)** *statement* **else** *statement*
16. *iteração-decl* → **while** **(** *expressão* **)** *statement*
17. *retorno-decl* → **return** **;** | **return** *expressão* **;**
18. *expressão* → *var* **=** *expressão* | *simples-expressão*
19. *var* → **ID** | **ID** **[** *expressão* **]**
20. *simples-expressão* → *soma-expressão relacional soma-expressão* | *soma-expressão*
21. *relacional* → **<=** | **<** | **>** | **>=** | **==** | **!=**
22. *soma-expressão* → *soma-expressão soma termo* | *termo*
23. *soma* → **+** | **-**
24. *termo* → *termo mult fator* | *fator*
25. *mult* → ***** | **/**
26. *fator* → **(** *expressão* **)** | *var* | *ativação* | **NUM**
27. *ativação* → **ID** **(** *args* **)**
28. *args* → *arg-lista* | *vazio*
29. *arg-lista* → *arg-lista* **,** *expressão* | *expressão*

Fonte: (1)

Os nós da árvore sintática possuem diferentes tipos que são utilizados para identificar diferentes trechos do código compilado, sendo eles:

- **IfK**: Referente a declaração da estrutura condicional *if*.
- **WhileK**: Referente a declaração do laço de repetição *while*.
- **AssignK**: Referente a operação de atribuição.
- **ReturnK**: Referente ao retorno de valores em uma função.
- **CallK**: Referente a chamada de função dentro do código.
- **Vark**: Referente a declaração de variáveis dentro do código.
- **VetK**: Referente a declaração de vetor dentro do código.
- **FuncaoK**: Referente a declaração de função.
- **OpK**: Referente a execução de operações logicas/aritiméticas.
- **ConstK**: Referente ao uso de constantes no código.

- **IdK**: Referentes aos identificadores.
- **VarIdK**: Referente ao uso de uma variável já declarada.
- **VetIdK**: Referente ao uso de um vetor já declarado.
- **TypeK**: Referente a especificação de tipo em declarações de funções ou variáveis.

Como produto desta etapa da análise teremos a arvore de análise sintática que será utilizada nos passos seguintes da compilação. Como exemplo a Figura 13 apresenta a árvore sintática gerada para o código 3.1.

```
0  int gcd(int u, int v)
1  {
2          if (v==0) return u;
3          else return gcd(v, u-u/v*v);
4  }
5  void main(void)
6  {
7          int x; int y;
8          x = input();
9          y = input();
10         output(gcd(x,y));
11 }
```

Listing 3.1 – Código GCD

Figura 13 – Árvore de Análise Sintática GCD

```
Tipo: int
  Funcão - int - gcd - Linha: 1 Escopo: global
    Tipo: int
      Var u Linha: 1 Escopo: gcd
    Tipo: int
      Var v Linha: 1 Escopo: gcd
    If
      Op: ==
        VarID: v Line: 3 Escopo: gcd
        Const: 0
      Return
        VarID: u Line: 3 Escopo: gcd
      Return
        Call: gcd Linha: 4
          VarID: v Line: 4 Escopo: gcd
          Op: -
            VarID: u Line: 4 Escopo: gcd
            Op: *
              Op: /
                VarID: u Line: 4 Escopo: gcd
                VarID: v Line: 4 Escopo: gcd
              VarID: v Line: 4 Escopo: gcd
Tipo: void
  Funcão - void - main - Linha: 6 Escopo: global
    Tipo: int
      Var x Linha: 8 Escopo: main
    Tipo: int
      Var y Linha: 8 Escopo: main
    Assign
      VarID: x Line: 9 Escopo: main
      Call: input Linha: 9
    Assign
      VarID: y Line: 10 Escopo: main
      Call: input Linha: 10
    Call: output Linha: 11
      Call: gcd Linha: 11
        VarID: x Line: 11 Escopo: main
        VarID: y Line: 11 Escopo: main
```

Fonte: Autoria Própria

3.4 Analise Semântica

A Análise Semântica compreende a última etapa da fase de análise do compilador. Nesta etapa , a partir da árvore de analise sintática construída na etapa anterior é realizada a construção da tabela de símbolos. A tabela de simbolos irá conter informações a respeito de funções e variáveis declaradas no código, geralmente são armazenados identificadores, tipos, número da linha em que aparece no código e para o caso de variáveis o escopo. Nesta ponto da análise são verificados também erros através da coleta de informações que fogem às regras gramaticais e expressões regulares, como por exemplo a verificação de tipos em operações de atribuição, unicidade na declaração de funções e variáveis entre outros.

Neste projeto a análise semântica é realizada através do percurso da arvore de análise sintática. Durante esse percurso as informações são inseridas na tabela de

símbolos conforme o tipo do nó encontrado e são verificadas a ocorrência dos seguintes erros semânticos:

- 1 - Variável não declarada.
- 2 - Atribuição inválida.
- 3 - Declaração inválida de variável.
- 4 - Redeclaração de variável.
- 5 - Chamada de função não declarada.
- 6 - Ausência de declaração da função *main*.
- 7 - Declaração de variável com o mesmo nome de função anteriormente declarada.

Para a verificação do erro 1 é verificado se o identificador da variável que está sendo utilizada já foi inserido na tabela de símbolos, caso não tenha sido inserido um aviso de erro é emitido. Para a verificação do erro 2 são analisados os nós de operação de atribuição da árvore observando se os filhos desse nó possuem o mesmo tipo, caso não possuam um aviso de erro é emitido, esse erro ocorre quando temos a atribuição de uma chamada de uma função do tipo *void* a uma variável. Para o erro 3 é verificado a o tipo atribuído a variável está correto, neste caso apenas é permitida a declaração de variáveis do tipo inteiro. Para o erro 4, ao detectar uma declaração de variável é feita verificação se um identificador com o mesmo nome já foi inserido na tabela de símbolos, em caso afirmativo é emitido um aviso de erro, este procedimento também trata o erro 7. No caso do erro 5 quando um nó referente à uma chamada de função é encontrado na árvore é verificado se o identificador da função que está sendo chamada já foi inserido na tabela de símbolos, caso não tenha sido um aviso de erro será emitido. Por fim, para a verificação do erro 7 basta que seja verificado se o identificador *main* foi inserido na tabela de símbolos, em caso negativo um aviso de erro é emitido.

A Figura 14 apresenta como exemplo a tabela de símbolos gerada para o código GCD apresentado no bloco de código 3.2, nela é possível observar todos os identificadores de funções e varáveis presentes no código bem como seu tipo e o número da linha em que aparecem no código.

```
0 int gcd(int u, int v)
1 {
2         if (v==0) return u;
```

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

Listing 3.2 – Código GCD

Figura 14 – Tabela de Símbolos GCD

Nome	Var/func	Tipo	Escopo	Número de linhas
main	funcao	void	global	6,
u	variável	int	gcd	1, 3, 4, 4,
y	variável	int	main	8, 10, 11,
gcd	funcao	int	global	1, 4, 11,
v	variável	int	gcd	1, 3, 4, 4, 4,
x	variável	int	main	8, 9, 11,

Fonte: Autoria Própria

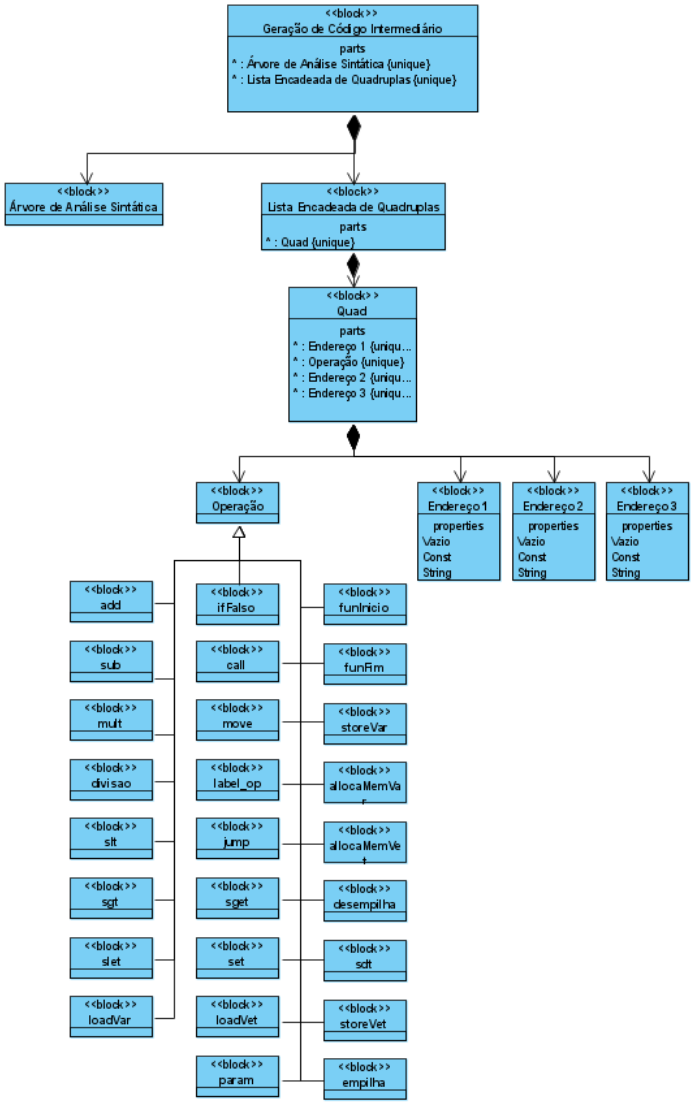
4 Compilador: Fase de Síntese

A Fase de Síntese é a etapa da compilação em que será feita a tradução do código em C- para o código executável da máquina alvo. Neste capítulo será inicialmente apresentada a modelagem do módulos de síntese empregando os diagramas de blocos e atividades SysML, em seguida a geração do código intermediário, a geração do *assembly*, do código binário executável e por fim será apresentado o gerenciamento de memória no processo de tradução.

4.1 Modelagem

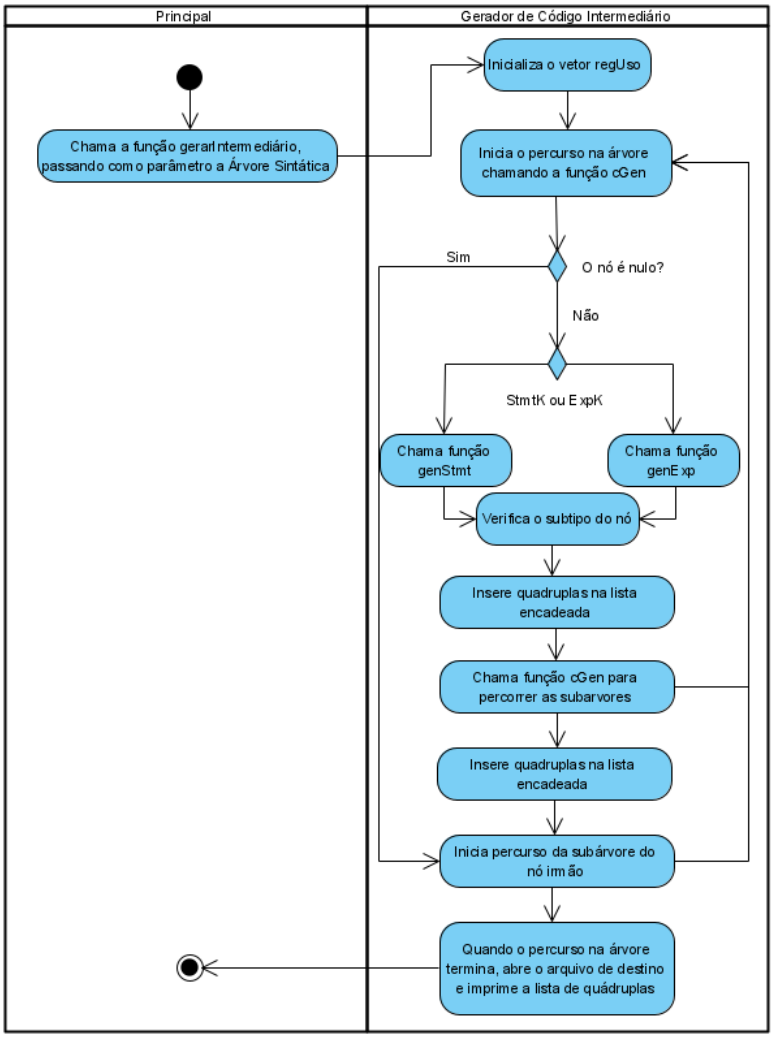
Nesta seção serão apresentados os diagramas SysML criados para a modelagem dos módulos de síntese do compilador. A Figura 15 apresenta o diagrama de blocos onde são mostrados os componentes do gerador de código intermediário que são a árvore de analise sintática e as quádruplas definidas. O passo-a-passo da geração do código intermediário é apresentado na Figura 16 por meio de um diagrama de atividades, que tem início com a chamada da função *geraIntermediario* que recebe como parâmetro a árvore sintática construída durante a fase de análise, em seguida o vetor *regUso* é inicializado e então o percurso da árvore se inicia até que todos os nós sejam visitados e as quádruplas geradas possam ser impressas no arquivo.

Figura 15 – Diagrama de Blocos - Gerador de Código Intermediário



Fonte: Autoria Própria

Figura 16 – Diagrama de Atividades - Gerador de Código Intermediário

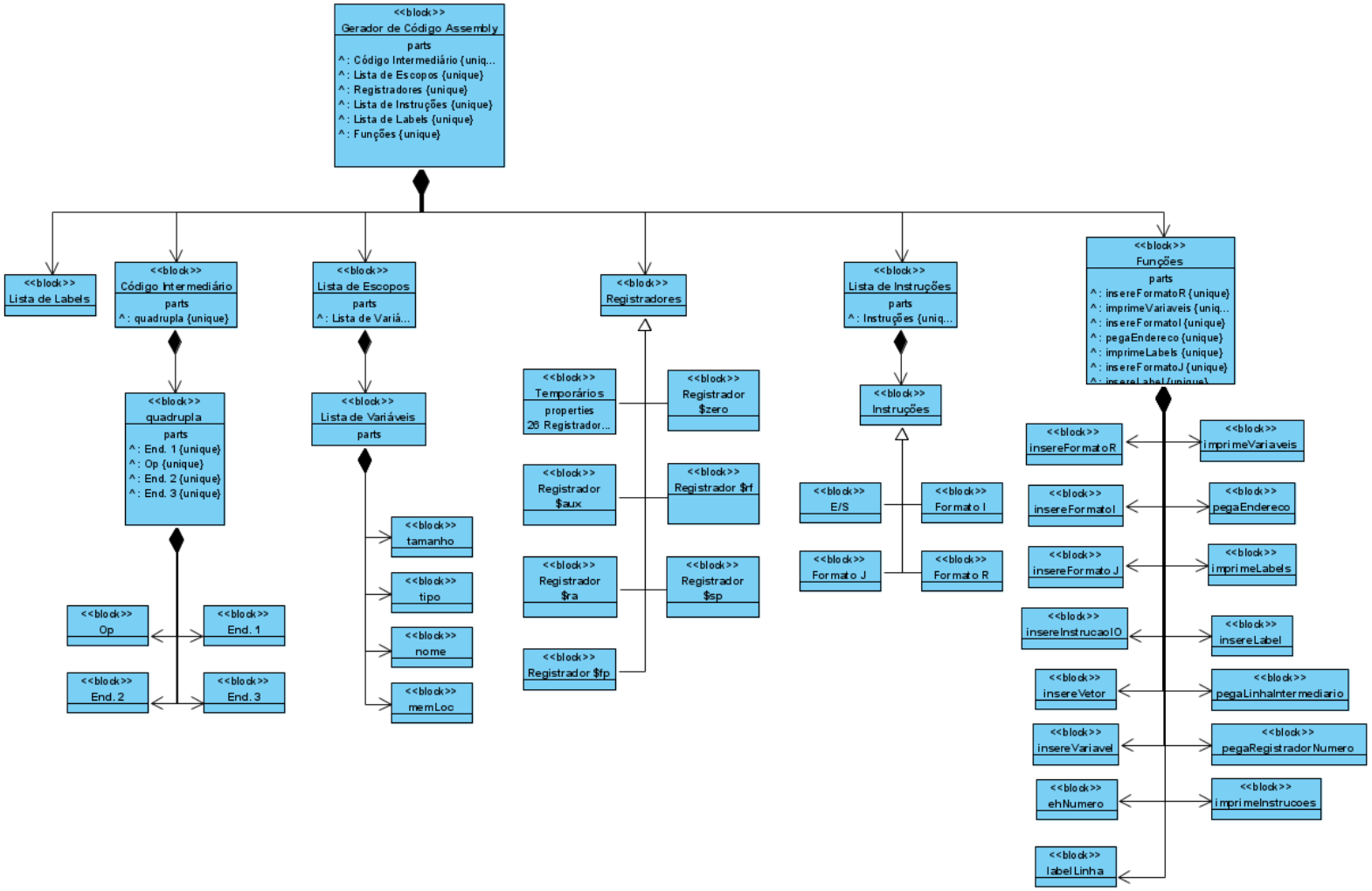


Fonte: Autoria Própria

Para o gerador de código assembly, a Figura 17 apresenta o diagrama de blocos com a composição do gerador, onde podem ser encontrado o código intermediário anteriormente gerado, as listas de labels, de escopo, de variáveis, o conjunto de registradores, a lista de instruções e as funções que auxiliam a geração do assembly. O diagrama de atividades deste módulo teve que ser divido em três figuras devido a grande quantidade de açoes que ele representa. As Figuras 18, 19, 20 apresentam o diagrama de atividades da geração de código assembly. Como pode-se observar no diagrama, esta etapa tem início com a abertura do arquivo que contém o código intermediário, em seguida a inserção de uma instrução *jump* e tem início então a leitura das quádruplas e conforme sua operação são

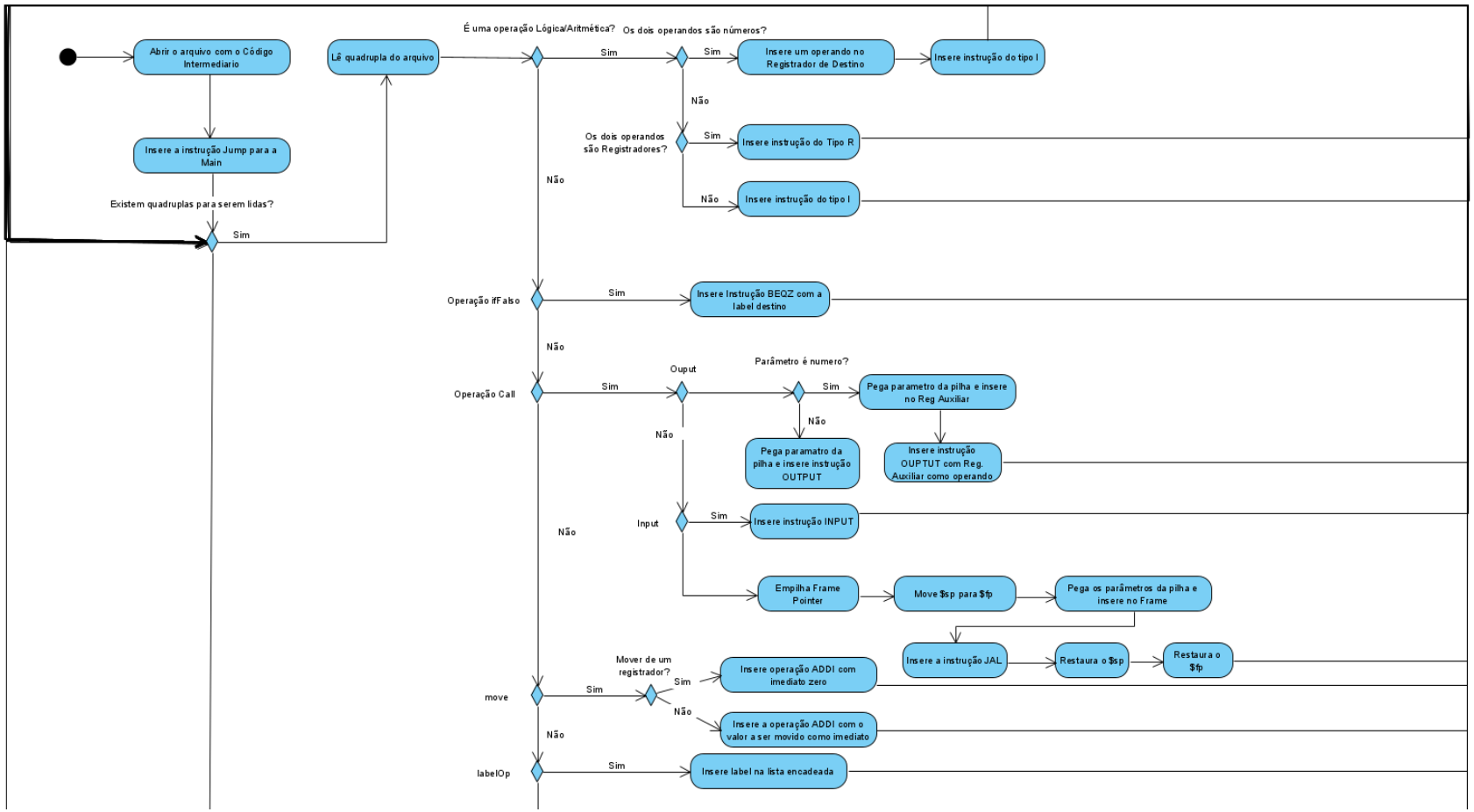
executadas diferentes ações no processo de tradução.

Figura 17 – Diagrama de Blocos - Gerador de Código Assembly



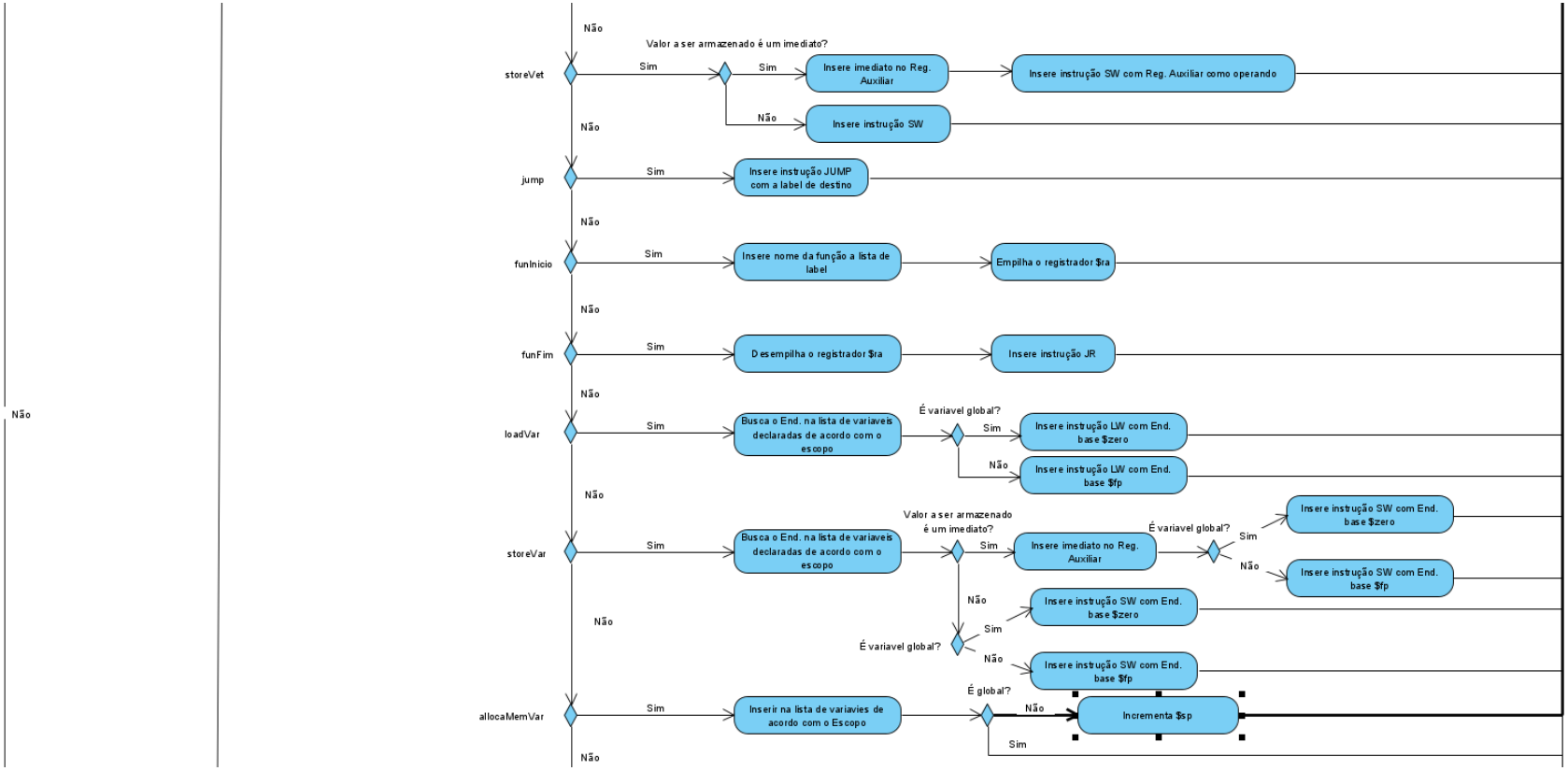
Fonte: Autoria Própria

Figura 18 – Diagrama de Atividades - Parte 1 - Gerador de Código Assembly



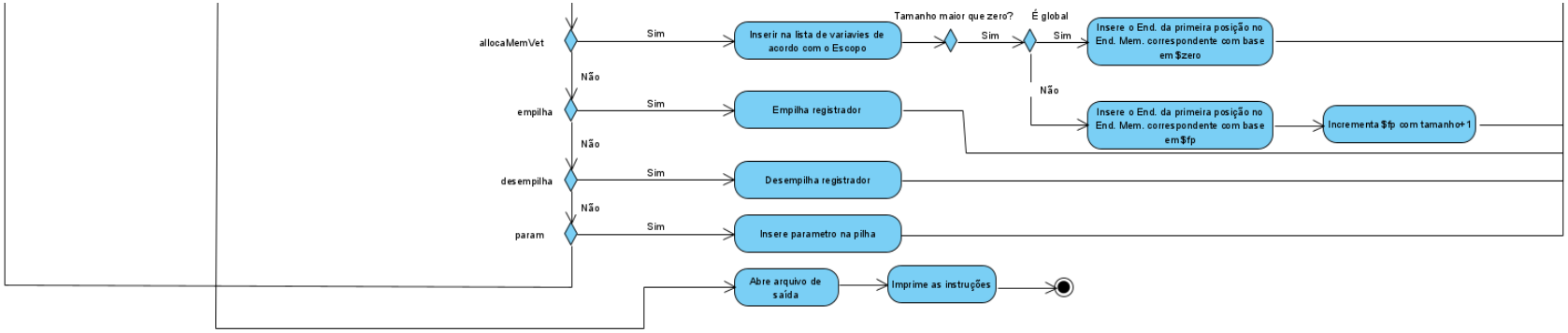
Fonte: Autoria Própria

Figura 19 – Diagrama de Atividades - Parte 2 - Gerador de Código Assembly



Fonte: Autoria Própria

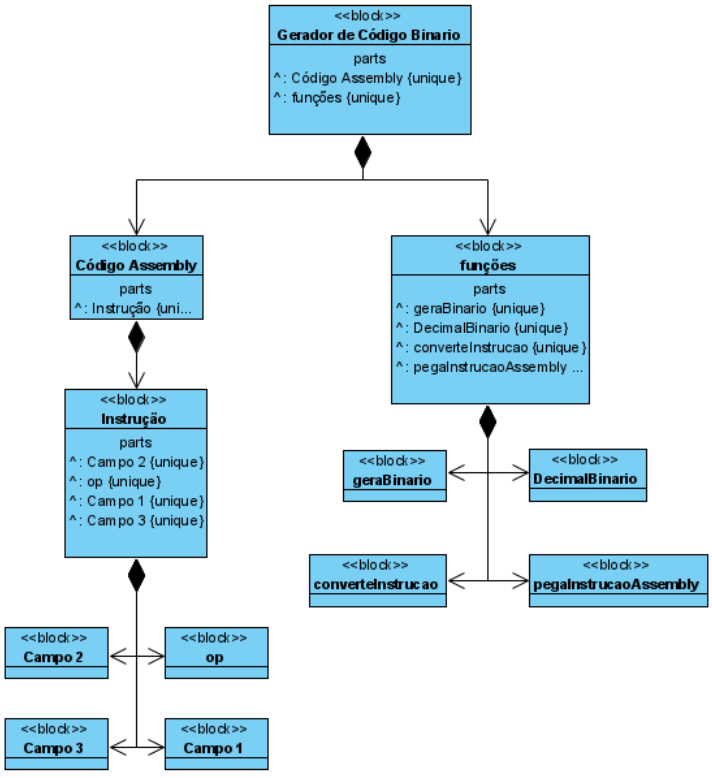
Figura 20 – Diagrama de Atividades - Parte 3 - Gerador de Código Assembly



Fonte: Autoria Própria

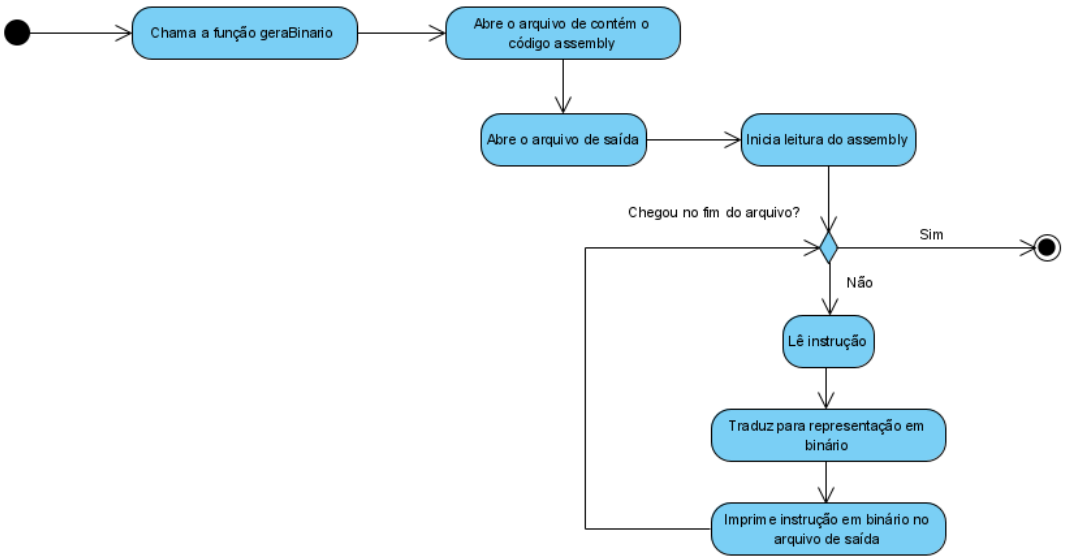
Por fim, as Figuras 21 e 22 apresentam os diagramas de blocos e atividades respectivamente para do gerador de código binário. Como pode-se observar no diagrama de blocos o gerador de binário possui poucos componentes sendo eles o código assembly de entrada e as funções que realizam a tradução, sendo a função *geraBinario* a função principal, a função *DecimalBinario* utilizada para converter valores decimais para sua representação em binário, a função *pegaInstrucaoAssembly* utilizada para ler uma instrução do arquivo e a função *converteInstrucao* a responsável por converter a instrução lida para a sua representação em binário. No diagrama de atividades temos o fluxo do processo de geração do binário que tem ínicio com a chamada da função *geraBinario*, em seguida é feita abertura do arquivo que contém o código assembly e para cada instrução lida é feita a tradução até que seja atingido o fim do arquivo e não existam mais instruções para serem lidas.

Figura 21 – Diagrama de Blocos - Gerador de Código Binário



Fonte: Autoria Própria

Figura 22 – Diagrama de Atividades - Gerador de Código Binário



Fonte: Autoria Própria

4.2 Geração do código intermediário

A geração do código intermediário é a etapa inicial da Fase de Síntese na qual ocorre a linearização da árvore de análise sintática através do seu percurso. Nesta etapa temos como produto o uma forma de representação intermediaria que se assemelha ao código-alvo. O código intermediário pode representar de maneira abstrata o código de entrada ou então incorporar características mais específicas da arquitetura alvo, como é o caso deste projeto em que características como operações LOAD-STORE e o uso e a quantidade de registradores da arquitetura alvo são levados em consideração.

Uma forma bastante usual de se representar o código intermediário é através do código de três endereços em que uma operação é efetuada sobre um conjunto de três operandos. Para a representação e armazenamento desse código são utilizadas estruturas chamadas quadruplas. Essas estruturas são compostas por quatro campos em que o primeiro armazena a operação e os outros três os operandos que serão utilizados. Nesta forma também são aceitas quadruplas que contenham menos de três operandos.

Neste projeto, para o armazenamento das quádruplas e geração do código intermediário foram definidos alguns tipos e estruturas conforme é apresentado na Figura 23. O primeiro tipo chamado Operacao é um enumerável utilizado para representar as quádruplas definidas. O tipo enumerável TipoEnd define o tipo do endereço presente na quádrupla, podendo ser vazio, constante ou uma *string*. A estrutura Endereco armazena o conteúdo(numérico ou texto) e o tipo de um endereço. A estrutura QuadListNo consiste em uma quádrupla que será armazenada em uma lista encadeada e contém três variáveis do tipo Endereco e uma do tipo Operacao utilizadas para armazenar os operandos e a operação respectivamente.

Figura 23 – Estruturas e tipos utilizados na geração do código intermediário

```
typedef enum {add,sub,mult,divisao,ifFalso, call, move,label_op,
storeVet,jump,slt,sgt,slet,sget,set,sdt, funInicio, funFim,
loadVar,loadVet, storeVar, allocaMemVar, allocaMemVet, param,
empilha,desempilha}Operacao;

typedef enum {Vazio,Const,String} TipoEnd;

typedef struct{
    TipoEnd tipo;
    union{
        int val;
        char *nome;
    }conteudo;
}Endereco;

typedef struct QuadListaNo{
    Operacao op;
    Endereco end1, end2, end3;
    struct QuadListaNo *prox;
}QuadLista;
```

Fonte: Autoria Própria

As quádruplas definidas para a geração do código intermediário e a funcionalidade de cada uma é apresentada na Tabela 4.

Quádrupla	**Função**
add	Operação de adição
sub	Operação de subtração
mult	Operação de multiplicação
divisao	Operação de divisão
ifFalso	Verifica se um valor armazenado no registrador temporário é falso, em caso afirmativo um salto deverá ser realizado para a label especificada
call	Representa a chamada de uma função. Nesta quádrupla são especificados o nome da função chamada e a quantidade de parâmetros necessários
move	Utilizada para mover um valor para um registrador, podendo ser um valor armazenado em outro registrador ou um imediato
label_op	Especifica uma label dentro do código
storeVet	Atribuição de um valor para uma dada posição de um vetor
jump	Salto incondicional para a label especificada
slt	Verifica se o primeiro operando é menor que o segundo
sgt	Verifica se o primeiro operando é maior que o segundo
slet	Verifica se o primeiro operando é menor/igual ao segundo
sget	Verifica se o primeiro operando é maior/igual ao segundo
set	Verifica se os operandos são iguais
sdt	Verifica se os operandos são diferentes
funInicio	Indica o início de uma função
funFim	Indica o fim de uma função
loadVar	Carregamento do valor de uma variável da memoria para um registrador
loadVet	Carregamento do valor de um vetor da memoria para um registrador
storeVar	Atribuição de um valor para uma variável
allocaMemVar	Representa a declaração de uma variável
allocaMemVet	Representa a declaração de um vetor
param	Representa um parâmetro de uma função
empilha	Especifica um registrador que deve ser empilhado antes de uma chamada de função
desempilha	Especifica um registrador que deve ser desempilhado após uma chamada de função

Tabela 4 – Quadruplas definidas

Fonte: Autoria Própria

A geração do código intermediário tem início com o percurso da árvore sintática através da chamada da função *cGen*, apresentada na Figura 24, que recebe como parâmetro

o nó raiz da árvore e verifica o tipo deste nó, se é do tipo *StmtK* ou do tipo *ExpK* chamando a função apropriada para cada um dos tipos e em seguida a função *cGen* é chamada passando como parâmetro o nó irmão do nó raiz para realizar o percurso. Nas funções *genStmt* e *genExp* é verificado qual o subtipo do nó e é feita a inserção das quádruplas apropriadas na lista encadeada.

Determinadas quadruplas necessitam de temporários para armazenar o resultado da operação, para isso são utilizados 26 registradores da arquitetura-alvo. O gerenciamento do uso de desses registradores é realizado por meio de um vetor de 26 posições que indica se um registrar está ou não em uso. Quando um registrador temporário é solicitado é verificado qual registrador está disponível para ser utilizado com base no vetor de status. Este mesmo vetor também é utilizado para definir quais registradores devem ser empilhados antes de uma chamada de função e desempilhados após o retorno.

Figura 24 – Função *cGen* para percurso da árvore sintática.

```
static void cGen(TreeNode *t)
{
    if(t != NULL)
    {
        switch(t->nodeKind)
        {
            case StmtK:
                genStmt(t);
                break;
            case ExpK:
                genExp(t);
                break;
            default:
                break;
        }
        cGen(t->sibling);
    }
}
```

Fonte: Autoria Própria

A Figura 25 apresenta o código intermediário gerado para o código apresentado em 3.2. No código gerado podem ser observadas as quadruplas geradas conforme a árvore sintática apresentada Figura 13. Para essa árvore temos como raiz um nó do tipo *TypeK* em seguida temos o início da declaração da função GCD que por esse motivo no código intermediário temos primeiro a geração do código para esta função iniciando com a quádrupla *funInicio* seguida pelas quadruplas referentes à alocação das variáveis tidas como argumentos e assim por diante conforme a árvore de entrada. Em seguida às quadruplas geradas para a função GCD temos a geração do código para a função *main*, isto se deve ao fato de que o nó que inicia a declaração da função ser um nó irmão do nó raiz da árvore.

Figura 25 – Código intermediário gerado para o programa GCD.

```
funInicio,gcd,___,___
allocaMemVar,gcd,u,___
allocaMemVar,gcd,v,___
loadVar,gcd,v,t0
set,t0,0,t1
ifFalso,t1,l0,___
loadVar,gcd,u,t0
move,t0,$rf,___
jump,l1,___,___
label_op,l0,___,___
loadVar,gcd,v,t0
param,t0,___,___
loadVar,gcd,u,t1
loadVar,gcd,u,t2
loadVar,gcd,v,t3
divisao,t2,t3,t4
loadVar,gcd,v,t2
mult,t4,t2,t3
sub,t1,t3,t2
param,t2,___,___
empilha,0,___,___
empilha,2,___,___
call,gcd,2,___
desempilha,2,___,___
desempilha,0,___,___
move,$rf,$rf,___
label_op,l1,___,___
funFim,gcd,___,___
funInicio,main,___,___
allocaMemVar,main,x,___
allocaMemVar,main,y,___
call,input,0,___
storeVar,$rf,x,main
call,input,0,___
storeVar,$rf,y,main
loadVar,main,x,t0
param,t0,___,___
loadVar,main,y,t1
param,t1,___,___
empilha,0,___,___
empilha,1,___,___
call,gcd,2,___
desempilha,1,___,___
desempilha,0,___,___
param,$rf,___,___
call,output,1,___
funFim,main,___,___
```

Fonte: Autoria Própria

4.3 Geração Código Assembly

A Geração do Código Assembly consiste na tradução do código intermediário para o código assembly da arquitetura alvo. Durante essa etapa da fase de síntese também é necessária fazer o gerenciamento das variáveis alocadas em memória, ajustar os endereços dos saltos e desvios dentro do programa e também gerenciar a forma como são realizadas as chamadas de função. Para este projeto a tradução do código intermediário para o assembly se dá, de uma maneira geral, através da tradução de cada uma das quadruplas, em que para cada uma são geradas as instruções necessárias.

O processo de tradução tem início com a abertura do arquivo que contém o código intermediário, em seguida para cada linha deste arquivo é verificado qual o tipo da quádrupla e são inseridas então as instruções em assembly na lista encadeada de instruções conforme a operação especificada pela quádrupla. Os nós da lista encadeada utilizada para armazenar o código assembly são definidos conforme a Figura 26, o campo *inst* é do tipo enumerável *instrucoes*(Figura 27) e armazena o identificador da instrução, os campos *rs*, *rt*, *rd* do tipo *registradores*(Figura 27) armazenam o enumerável que identifica os registradores utilizados, a *string imediato* é utilizada pelas instruções que operam com valores imediatos, a *string endereço* é utilizada em instruções de salto para armazenar o endereço de destino e o campo *tipo* idetifica o tipo de instrução(1 - Formato R, 2 - Formato I, 3 - Formato J e 4 - Entrada e Saída)

Figura 26 – Definição do nó da lista de instruções.

```
class nodeInstrucao
{
    public:
    instrucoes inst;
    registradores rs;
    registradores rt;
    registradores rd;
    string imediato;
    string endereco;
    int tipo;
    nodeInstrucao *prox;
};
```

Fonte: Autoria Própria

Figura 27 – Tipos enumeráveis para registradores e instruções.

```
enum instrucoes:int {ADD, SUB, MULT, DIV, AND, OR, NOR,
                     SLT, SGT,SLET, SGET, SET, SDT, SGTI,
                     SLETI, SGETI, SETI, SDTI, SLL, SRL,
                     MOD,JR,ADDI,MULTI,DIVI,ANDI,BLTZ,
                     BGTZ,BEQZ,BEQ, BNE, LW, SW, ORI, SLTI,
                      MODI, JUMP,JAL,NOP,HALT,INPUT,OUTPUT,LUI};

enum registradores:int {$zero, $t0, $t1, $t2, $t3, $t4,
                        $t5, $t6, $t7, $t8, $t9, $t10, $t11,
                        $t12, $t13, $t14, $t15, $t16, $t17,
                        $t18, $t19, $t20, $t21, $t22, $t23,
                        $t24, $t25, $aux, $rf, $fp, $sp,$ra};
```

Fonte: Autoria Própria

Para a geração do código assembly são utilizados os 32 registradores disponíveis na arquitetura alvo. O conjunto de registradores é composto pelos registradores *$zero* que contém o valor zero armazenado que será utilizado em determinadas instruções, os

registradores *$t0 a $t25* são os registradores temporários, o registrador auxiliar *$aux*, *$rf* utilizado para armazenar o retorno de funções, *$fp* utilizado para identificar o início do frame da função em execução, *$sp* que aponta para o topo da pilha e *$ra* que armazena o endereço da próxima instrução a ser executada após uma chamada de função.

Durante o processo de tradução, conforme as quadruplas *allocaMemVar* e *allocaMemVet* são encontradas é necessário fazer o gerenciamento dos endereços das variáveis alocadas em memória. Para isso será utilizada uma lista encadeada para armazenar os escopos do código e dentro de cada nó da lista de escopo estará contido o nome e a posição atual na memória que indica em qual endereço a próxima variável encontrada nesse escopo deve ser alocada e uma lista encadeada de variáveis que irá armazenar o nome, tipo(0 - Variável e 1 - Vetor), localização dentro da memória e o tamanho para o caso dos vetores. A definição das classes para criação dos nós das listas é apresentada na Figura 28. Quando uma quádrupla do tipo *allocaMemVar* é encontrada no código será feita inserção na lista de escopos, que é percorrida em busca do escopo da variável indicado na quádrupla, caso ele não seja encontrado um novo nó será adicionado à lista. Com o escopo encontrado será então adicionada a variável à sua lista encadeada de variáveis setando sua posição dentro da memória como sendo a posição atual de memória do escopo. Para o caso de uma quádrupla *allocaMemVet* temos o mesmo processo, contudo após a atribuição da posição de memória ao vetor será necessário incrementar a posição atual de memória do escopo com o tamanho do vetor mais um, devido a forma como o vetor é alocado em memória, este tópico será abordado com mais profundidade na seção 4.5.

Para gerenciar as *labels* de destino utilizadas pelas quádruplas de salto e desvio é utilizada um lista encadeada que armazenará o nome das *labels* especificadas pela quadrupla *label_op* e o número da linha dentro do código assembly ao qual ela se refere de acordo com a contagem atual de instruções inseridas. Durante o processo de impressão do código assembly para o arquivo as *labels* indicadas nas instruções serão substituídas pelo número da linha ou quantidade de linhas a serem desviadas conforme especificada na lista de labels.

Figura 28 – Definição dos nós da listas de escopos e variáveis.

```
class nodeEscopo
{
    public:
    int posAtualMem;
    string nome;
    nodeVariavel *var;
    nodeEscopo *prox;
};

class nodeVariavel
{
    public:
    int memLoc;
    string nome;
    int tipo;
    int tamanho;
    nodeVariavel *prox;
};
```

Fonte: Autoria Própria

Para fazer o gerenciamento dos parâmetros passados durante a chamada de uma função foi criada uma pilha na qual sempre que uma quádrupla *param* é encontrada o parâmetro especificado por ela é inserido na pilha. Desta forma quando temos uma chamada de função esses parâmetros são desempilhados de modo que possam ser inseridos nas posições de memória dentro do *frame* da função que está sendo chamada para que durante sua execução ela possa acessa-los de maneira correta. Para isso os valores imediatos e registradores presentes na pilha de parâmetros são armazenados dentro do *frame* da função por meio da instrução *StoreWord*.

A Figura 29 apresenta o código assembly gerado para o código intermediário apresentado na Figura 25. Pode-se observar que a primeira instrução inserida é um *jump* para linha 38, isso feito para que a execução do código tenha início com a função *main*.

Figura 29 – Código Assembly GCD.

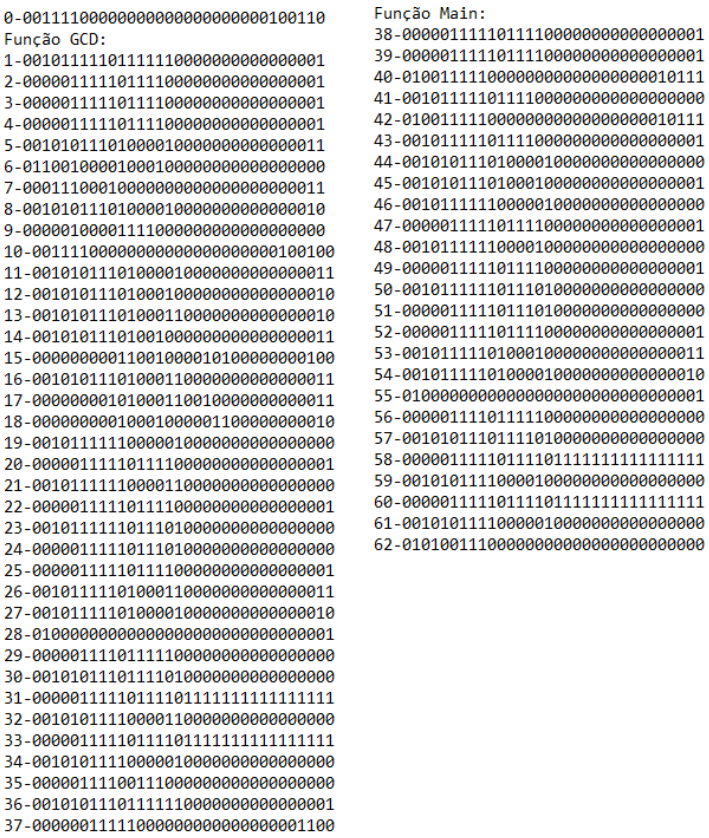
```
0-jump 38
Função GCD:
1-sw r29 r31 1
2-addi r30 r30 1
3-addi r30 r30 1
4-addi r30 r30 1
5-lw r29 r1 3
6-seti r1 r2 0
7-beqz r2 r0 3
8-lw r29 r1 2
9-addi r1 r28 0
10-jump 36
11-lw r29 r1 3
12-lw r29 r2 2
13-lw r29 r3 2
14-lw r29 r4 3
15-div r3 r4 r5
16-lw r29 r3 3
17-mult r5 r3 r4
18-sub r2 r4 r3
19-sw r30 r1 0
20-addi r30 r30 1
21-sw r30 r3 0
22-addi r30 r30 1
23-sw r30 r29 0
24-addi r30 r29 0
25-addi r30 r30 1
26-sw r29 r3 3
27-sw r29 r1 2
28-jal 1
29-addi r29 r30 0
30-lw r29 r29 0
31-addi r30 r30 -1
32-lw r30 r3 0
33-addi r30 r30 -1
34-lw r30 r1 0
35-addi r28 r28 0
36-lw r29 r31 1
37-jr r31 r0 r0
Função Main:
38-addi r30 r30 1
39-addi r30 r30 1
40-input r28
41-sw r29 r28 0
42-input r28
43-sw r29 r28 1
44-lw r29 r1 0
45-lw r29 r2 1
46-sw r30 r1 0
47-addi r30 r30 1
48-sw r30 r2 0
49-addi r30 r30 1
50-sw r30 r29 0
51-addi r30 r29 0
52-addi r30 r30 1
53-sw r29 r2 3
54-sw r29 r1 2
55-jal 1
56-addi r29 r30 0
57-lw r29 r29 0
58-addi r30 r30 -1
59-lw r30 r2 0
60-addi r30 r30 -1
61-lw r30 r1 0
62-output r28
```

Fonte: Autoria Própria

4.4 Geração do Código Binário

A geração do Código Binário é etapa final do processo de compilação, ela consiste basicamente da tradução do assembly para o seu correspondente em binário, em que o nome da instrução é substituído pelo seu *OpCode* binário e, para o caso das instruções do Formato R e E/S também utiliza-se o código *funct* binário. O numero dos registradores, endereços e imediatos também são convertidos da sua representação decimal para números binários. Esse processo tem início com a abertura do arquivo que contém o código assembly gerado e para cada linha é feita a tradução conforme o tipo da instrução e em seguida o correspondente binário é impresso em arquivo. Para a tradução das instruções foram utilizados os *OpCodes* e *funct* apresentados na Tabela 3 e para os números decimais foi implementada uma função que recebe como parâmetro a representação decimal e a quantidade de bits que devem utilizados na representação e retorna o binário correspondente. A Figura 30 apresenta o código binário refente a tradução do código assembly apresentado na Figura 29.

Figura 30 – Código Binário GCD.



Fonte: Autoria Própria

4.5 Gerenciamento de Memória

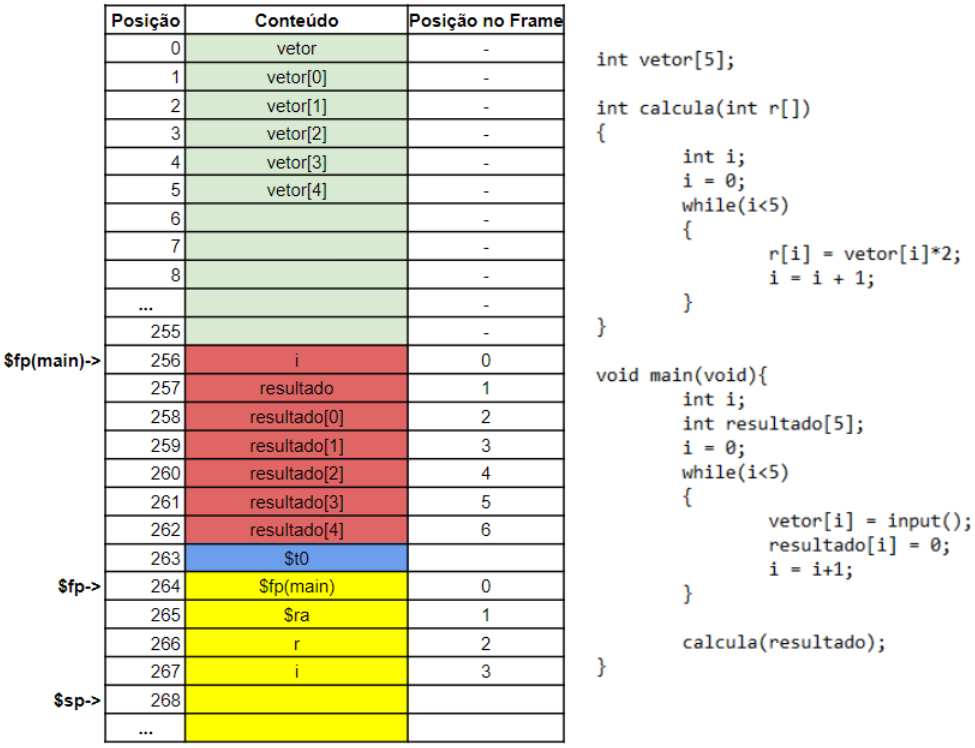
Para fazer a tradução do código fonte para o código assembly da arquitetura alvo é necessário também fazer o gerenciamento da memória e da forma como as variáveis serão dispostas dentro dela. Para este projeto foi utilizada a abordagem baseada em pilhas apresentada no livro (1). Nesta abordagem para cada ativação de função um *frame* que contém as variáveis alocadas da função que foi chamada é inserido na pilha. Desta forma foi reservado um registrador chamado de registrador *$fp* que contém o endereço de memória do início do *frame* da função atualmente em execução. Com isso o endereçamento de variáveis dentro de uma função é feito de maneira relativa através da soma do endereço da variável dentro do *frame* com o valor do registrador *$fp*. Por esse motivo durante a fase de geração do código assembly quando as variáveis locais são alocadas o endereço atribuído à elas é o endereço dentro do frame. Para a função *main* os endereços atribuídos às variaveis começam a partir da posição 0, já para as

demais funções a posição 0 do *frame* é reservada para armazenar o *$fp* da função que fez a chamada e a posição 1 é reservada para armazenar o valor presente no registrador *$ra* que indica para onde a execução do programa deve retornar quando a função terminar, desta forma os endereços atribuídos às variáveis tem início a partir da posição 2. Para as variáveis globais o endereçamento ocorre de maneira direta, pois são alocadas na parte da memória reservada unicamente para este fim, desta forma quando as quádruplas de alocação de variáveis e vetores globais são encontradas durante a geração do assembly o endereço atribuído a elas já é o seu endereço final, que tem início a partir posição 0 da memória.

A memoria de dados foi divida em duas partes, a primeira que contém 256 posições será utilizada para armazenar as variáveis globais do programa, e a segunda parte a partir da posição 256 será utilizada como uma pilha com o registrador *$sp* apontado para o topo.

Na Figura 31 é apresentado um código em C- e a forma como as variáveis estão dispostas na memória durante a execução deste programa até o momento em que é feita a declaração das variáveis da função *calcula*. A parte representada em verde corresponde a parte da memória reservada para as variáveis globais, a parte em vermelho corresponde ao *frame* da função *main*, a parte em azul corresponde ao registrador que precisou ser empilhado antes da chamada da função e a parte em amarelo corresponde ao *frame* da função *calcula*. Nesta figura pode-se observar que o vetor global foi alocado nas primeiras seis posições da área global da memoria com a posição 0 sendo utilizada para armazenar o endereço da primeira posição do vetor, ou seja na posição de memória 0 estará contido o valor 1. Na função *main* a variável *i* é alocada na posição 0 do *frame* e o vetor *resultado* alocado na posição 1 que corresponde a posição de memoria 257 que irá armazenar o valor 258 referente ao endereço da primeira posição do vetor, desta forma, por exemplo quando, é necessário acessar a posição 3 do vetor o primeiro passo é carregar o valor contido em *$fp(main)* somado ao valor 1(posição do vetor dentro do *frame*), o valor carregado será então somado à 3 resultando no endereço de memória que deverá ser acessado. Quando um vetor é passado como parâmetro em uma função o endereço da posição 0 do vetor será copiado para o local de memória atribuído ao vetor da função chamada, por esse motivo o vetor *r* da função *calcula* ocupa apenas uma posição na memória que irá conter o endereço da primeira posição do vetor *resultado* declarado na *main*. Desta forma quando for feita uma alteração no vetor *r* o vetor *resultado* será alterado.

Figura 31 – Representação da memória de dados durante a execução de um programa.



	Posição	Conteúdo	Posição no Frame
	0	vetor	-
	1	vetor[0]	-
	2	vetor[1]	-
	3	vetor[2]	-
	4	vetor[3]	-
	5	vetor[4]	-
	6		-
	7		-
	8		-
	...		-
	255		-
$fp(main)->	256	i	0
	257	resultado	1
	258	resultado[0]	2
	259	resultado[1]	3
	260	resultado[2]	4
	261	resultado[3]	5
	262	resultado[4]	6
	263	$t0	
$fp->	264	$fp(main)	0
	265	$ra	1
	266	r	2
	267	i	3
$sp->	268		
	...		

```
int vetor[5];

int calcula(int r[])
{
        int i;
        i = 0;
        while(i<5)
        {
                r[i] = vetor[i]*2;
                i = i + 1;
        }
}

void main(void){
        int i;
        int resultado[5];
        i = 0;
        while(i<5)
        {
                vetor[i] = input();
                resultado[i] = 0;
                i = i+1;
        }

        calcula(resultado);
}
```

Fonte: Autoria Própria

5 Exemplos

Neste capítulo serão apresentados o código intermediário, o código assembly e o código binário para três exemplos de código C-.

5.1 Exemplo 1 - GCD

Este primeiro exemplo apresenta os códigos intermediário, assembly e binário gerados para o código em C- do programa GCD. Será apresentado também um comparativo entre os códigos mostrando a relação entre cada linha do código fonte com o código intermedário e com o código assembly.

```
0  int gcd(int u, int v)
1  {
2          if (v==0) return u;
3          else return gcd(v, u-u/v*v);
4  }
5  void main(void)
6  {
7          int x;
8          int y;
9          x = input();
10         y = input();
11         output(gcd(x,y));
12 }
```

Listing 5.1 – Código C- do GCD

```
0  funInicio,gcd,___,___
1  allocaMemVar,gcd,u,___
2  allocaMemVar,gcd,v,___
3  loadVar,gcd,v,t0
4  set,t0,0,t1
5  ifFalso,t1,l0,___
6  loadVar,gcd,u,t0
7  move,t0,$rf,___
8  jump,l1,___,___
9  label_op,l0,___,___
10 loadVar,gcd,v,t0
11 param,t0,___,___
12 loadVar,gcd,u,t1
13 loadVar,gcd,u,t2
14 loadVar,gcd,v,t3
15 divisao,t2,t3,t4
16 loadVar,gcd,v,t2
17 mult,t4,t2,t3
```

```
18 sub,t1,t3,t2
19 param,t2,___,___
20 empilha,0,___,___
21 empilha,2,___,___
22 call,gcd,2,___
23 desempilha,2,___,___
24 desempilha,0,___,___
25 move,$rf,$rf,___
26 label_op,l1,___,___
27 funFim,gcd,___,___
28 funInicio,main,___,___
29 allocaMemVar,main,x,___
30 allocaMemVar,main,y,___
31 call,input,0,___
32 storeVar,$rf,x,main
33 call,input,0,___
34 storeVar,$rf,y,main
35 loadVar,main,x,t0
36 param,t0,___,___
37 loadVar,main,y,t1
38 param,t1,___,___
39 empilha,0,___,___
40 empilha,1,___,___
41 call,gcd,2,___
42 desempilha,1,___,___
43 desempilha,0,___,___
44 param,$rf,___,___
45 call,output,1,___
46 funFim,main,___,___
```

Listing 5.2 – Código Intermediário - GCD

```
0  jump 38
1  sw r29 r31 1
2  addi r30 r30 1
3  addi r30 r30 1
4  addi r30 r30 1
5  lw r29 r1 3
6  seti r1 r2 0
7  beqz r2 r0 3
8  lw r29 r1 2
9  addi r1 r28 0
10 jump 36
11 lw r29 r1 3
12 lw r29 r2 2
13 lw r29 r3 2
14 lw r29 r4 3
15 div r3 r4 r5
16 lw r29 r3 3
17 mult r5 r3 r4
18 sub r2 r4 r3
19 sw r30 r1 0
20 addi r30 r30 1
21 sw r30 r3 0
```

```
22 addi r30 r30 1
23 sw r30 r29 0
24 addi r30 r29 0
25 addi r30 r30 1
26 sw r29 r3 3
27 sw r29 r1 2
28 jal 1
29 addi r29 r30 0
30 lw r29 r29 0
31 addi r30 r30 -1
32 lw r30 r3 0
33 addi r30 r30 -1
34 lw r30 r1 0
35 addi r28 r28 0
36 lw r29 r31 1
37 jr r31 r0 r0
38 addi r30 r30 1
39 addi r30 r30 1
40 input r28
41 sw r29 r28 0
42 input r28
43 sw r29 r28 1
44 lw r29 r1 0
45 lw r29 r2 1
46 sw r30 r1 0
47 addi r30 r30 1
48 sw r30 r2 0
49 addi r30 r30 1
50 sw r30 r29 0
51 addi r30 r29 0
52 addi r30 r30 1
53 sw r29 r2 3
54 sw r29 r1 2
55 jal 1
56 addi r29 r30 0
57 lw r29 r29 0
58 addi r30 r30 -1
59 lw r30 r2 0
60 addi r30 r30 -1
61 lw r30 r1 0
62 output r28
```

Listing 5.3 – Código Assembly - GCD

```
0 00111100000000000000000000100110
1 00101111101111110000000000000001
2 00000111110111110000000000000001
3 00000111110111110000000000000001
4 00000111110111110000000000000001
5 00101011101000010000000000000011
6 01100100001000100000000000000000
7 00011100010000000000000000000011
8 00101011101000010000000000000010
9 00000100001111000000000000000000
```

```
10 00111100000000000000000000100100
11 00101011101000010000000000000011
12 00101011101000100000000000000010
13 00101011101000110000000000000010
14 00101011101001000000000000000011
15 00000000011001000010100000000100
16 00101011101000110000000000000011
17 00000000101000110010000000000011
18 00000000010001000001100000000010
19 00101111110000010000000000000000
20 00000111110111100000000000000001
21 00101111110000110000000000000000
22 00000111110111100000000000000001
23 00101111110111010000000000000000
24 00000111110111010000000000000000
25 00000111110111100000000000000001
26 00101111101000110000000000000011
27 00101111101000010000000000000010
28 01000000000000000000000000000001
29 00000111101111100000000000000000
30 00101011101111010000000000000000
31 00000111110111101111111111111111
32 00101011110000110000000000000000
33 00000111110111101111111111111111
34 00101011110000010000000000000000
35 00000111100111000000000000000000
36 00101011101111110000000000000001
37 00000011111000000000000000001100
38 00000111110111100000000000000001
39 00000111110111100000000000000001
40 01001111100000000000000000010111
41 00101111101111000000000000000000
42 01001111100000000000000000010111
43 00101111101111000000000000000001
44 00101011101000010000000000000000
45 00101011101000100000000000000001
46 00101111110000010000000000000000
47 00000111110111100000000000000001
48 00101111110000100000000000000000
49 00000111110111100000000000000001
50 00101111110111010000000000000000
51 00000111110111010000000000000000
52 00000111110111100000000000000001
53 00101111101000100000000000000011
54 00101111101000010000000000000010
55 01000000000000000000000000000001
56 00000111101111100000000000000000
57 00101011101111010000000000000000
58 00000111110111101111111111111111
59 00101011111000010000000000000000
60 00000111110111101111111111111111
61 00101011111000001000000000000000
62 01010011100000000000000000000000
```

63

Nas Figuras 32 e 33 é apresentado um comparativo entre o código C-, o intermediário e o assembly, mostrando a correspondência das linhas entre eles. É possível observar que a quantidade de linhas de código cresce consideravelmente quando comparamos o código fonte com o código intermediário, o que não é observado na relação entre o intermediário e o assembly já que a maioria das quadruplas se transforma em poucas instruções do código assembly. Como no código intermediário a característica LOAD-STORE da arquitetura alvo foi incorporada é possível observar que as quadruplas que fazem acesso às variáveis são traduzidas de maneira mais direta para o assembly. Neste comparativo torna-se visível como uma simples chamada de função com apenas dois argumentos vem a se tornar múltiplas instruções devido a necessidade de alguns preparativos antes de executar de fato a função chamada. É visto também que algo que não indicado através de um código no C- que é o fim de uma função vem a se tornar uma quadrupla e duas instruções no código assembly, já que é necessário indicar que ao fim da execução da função a execução do programa deve retornar para função que fez a chamada.

Figura 32 – Comparativo GCD - Parte 1.

Código C-	Código Intermediário	Código Assembly
		jump 38
int gcd(int u, int v)	funInicio,gcd,___,___	sw r29 r31 1 addi r30 r30 1
	allocaMemVar,gcd,u,___	addi r30 r30 1
	allocaMemVar,gcd,v,___	addi r30 r30 1
if (v==0) return u;	loadVar,gcd,v,t0	lw r29 r1 3
	set,t0,0,t1	seti r1 r2 0
	ifFalso,t1,l0,___	beqz r2 r0 3
	loadVar,gcd,u,t0	lw r29 r1 2
	move,t0,$rf,___	addi r1 r28 0
	jump,l1,___,___	jump 36
else return gcd(v, u-u/v*v);	label_op,l0,___,___	
	loadVar,gcd,v,t0	lw r29 r1 3
	param,t0,___,___	
	loadVar,gcd,u,t1	lw r29 r2 2
	loadVar,gcd,u,t2	lw r29 r3 2
	loadVar,gcd,v,t3	lw r29 r4 3
	divisao,t2,t3,t4	div r3 r4 r5
	loadVar,gcd,v,t2	lw r29 r3 3
	mult,t4,t2,t3	mult r5 r3 r4
	sub,t1,t3,t2	sub r2 r4 r3
	param,t2,___,___	
	empilha,0,___,___	sw r30 r1 0 addi r30 r30 1
	empilha,2,___,___	sw r30 r3 0 addi r30 r30 1
	call,gcd,2,___	sw r30 r29 0 addi r30 r29 0 addi r30 r30 1 sw r29 r3 3 sw r29 r1 2 jal 1 addi r29 r30 0 lw r29 r29 0
	desempilha,2,___,___	addi r30 r30 -1 lw r30 r3 0
	desempilha,0,___,___	addi r30 r30 -1 lw r30 r1 0
	move,$rf,$rf,___	addi r28 r28 0
	label_op,l1,___,___	
Fim da função "gcd"	funFim,gcd,___,___	lw r29 r31 1 jr r31 r0 r0

Fonte: Autoria Própria

Figura 33 – Comparativo GCD - Parte 2.

Código C-	**Código Intermediário**	**Código Assembly**
void main(void)	funInicio,main,___,___	
int x;	allocaMemVar,main,x,___	addi r30 r30 1
int y;	allocaMemVar,main,y,___	addi r30 r30 1
x = input();	call,input,0,___	input r28
	storeVar,$rf,x,main	sw r29 r28 0
y = input();	call,input,0,___	input r28
	storeVar,$rf,y,main	sw r29 r28 1
output(gcd(x,y));	loadVar,main,x,t0	lw r29 r1 0
	param,t0,___,___	
	loadVar,main,y,t1	lw r29 r2 1
	param,t1,___,___	
	empilha,0,___,___	sw r30 r1 0 addi r30 r30 1
	empilha,1,___,___	sw r30 r2 0 addi r30 r30 1
	call,gcd,2,___	sw r30 r29 0 addi r30 r29 0 addi r30 r30 1 sw r29 r2 3 sw r29 r1 2 jal 1 addi r29 r30 0 lw r29 r29 0
	desempilha,1,___,___	addi r30 r30 -1 lw r30 r2 0
	desempilha,0,___,___	addi r30 r30 -1 lw r30 r1 0
	param,$rf,___,___	
	call,output,1,___	output r28
Fim da função "main"	funFim,main,___,___	

Fonte: Autoria Própria

5.2 Exemplo 2 - Sort

Nesta seção é apresentado o código intermediário, o assembly e o binário gerados pelo compilador para o programa Sort apresentado no Listing 5.4.

```
0 int vet[ 10 ];
1
2 int minloc ( int a[], int low, int high )
```

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

Listing 5.4 – Código C- do Sort

```
0  allocaMemVet,global,vet,10
1  funInicio,minloc,___,___
2  allocaMemVet,minloc,a,-1
3  allocaMemVar,minloc,low,___
4  allocaMemVar,minloc,high,___
5  allocaMemVar,minloc,i,___
6  allocaMemVar,minloc,x,___
7  allocaMemVar,minloc,k,___
8  loadVar,minloc,low,t0
```

```
9  storeVar,t0,k,minloc
10 loadVar,minloc,a,t0
11 loadVar,minloc,low,t1
12 add,t0,t1,t2
13 loadVet,t2,t0,___
14 storeVar,t0,x,minloc
15 loadVar,minloc,low,t0
16 add,t0,1,t1
17 storeVar,t1,i,minloc
18 label_op,l0,___,___
19 loadVar,minloc,i,t0
20 loadVar,minloc,high,t1
21 slt,t0,t1,t2
22 ifFalso,t2,l1,___
23 loadVar,minloc,a,t0
24 loadVar,minloc,i,t1
25 add,t0,t1,t2
26 loadVet,t2,t0,___
27 loadVar,minloc,x,t1
28 slt,t0,t1,t2
29 ifFalso,t2,l2,___
30 loadVar,minloc,a,t0
31 loadVar,minloc,i,t1
32 add,t0,t1,t2
33 loadVet,t2,t0,___
34 storeVar,t0,x,minloc
35 loadVar,minloc,i,t0
36 storeVar,t0,k,minloc
37 label_op,l2,___,___
38 loadVar,minloc,i,t0
39 add,t0,1,t1
40 storeVar,t1,i,minloc
41 jump,l0,___,___
42 label_op,l1,___,___
43 loadVar,minloc,k,t0
44 move,t0,$rf,___
45 funFim,minloc,___,___
46 funInicio,sort,___,___
47 allocaMemVet,sort,a,-1
48 allocaMemVar,sort,low,___
49 allocaMemVar,sort,high,___
50 allocaMemVar,sort,i,___
51 allocaMemVar,sort,k,___
52 loadVar,sort,low,t0
53 storeVar,t0,i,sort
54 label_op,l3,___,___
55 loadVar,sort,i,t0
56 loadVar,sort,high,t1
57 sub,t1,1,t2
58 slt,t0,t2,t1
59 ifFalso,t1,l4,___
60 allocaMemVar,sort,t,___
61 loadVar,sort,a,t0
62 param,t0,___,___
```

```
63  loadVar,sort,i,t1
64  param,t1,___,___
65  loadVar,sort,high,t2
66  param,t2,___,___
67  empilha,0,___,___
68  empilha,1,___,___
69  empilha,2,___,___
70  call,minloc,3,___
71  desempilha,2,___,___
72  desempilha,1,___,___
73  desempilha,0,___,___
74  storeVar,$rf,k,sort
75  loadVar,sort,a,t0
76  loadVar,sort,k,t3
77  add,t0,t3,t4
78  loadVet,t4,t0,___
79  storeVar,t0,t,sort
80  loadVar,sort,a,t0
81  loadVar,sort,i,t3
82  add,t0,t3,t4
83  loadVet,t4,t0,___
84  loadVar,sort,a,t3
85  loadVar,sort,k,t4
86  add,t3,t4,t5
87  storeVet,t0,t5,___
88  loadVar,sort,t,t0
89  loadVar,sort,a,t3
90  loadVar,sort,i,t4
91  add,t3,t4,t5
92  storeVet,t0,t5,___
93  loadVar,sort,i,t0
94  add,t0,1,t3
95  storeVar,t3,i,sort
96  jump,l3,___,___
97  label_op,l4,___,___
98  funFim,sort,___,___
99  funInicio,main,___,___
100 allocaMemVar,main,i,___
101 storeVar,0,i,main
102 label_op,l5,___,___
103 loadVar,main,i,t0
104 slt,t0,10,t1
105 ifFalso,t1,l6,___
106 call,input,0,___
107 loadVar,global,vet,t0
108 loadVar,main,i,t1
109 add,t0,t1,t2
110 storeVet,$rf,t2,___
111 loadVar,main,i,t0
112 add,t0,1,t1
113 storeVar,t1,i,main
114 jump,l5,___,___
115 label_op,l6,___,___
116 loadVar,global,vet,t0
```

```
117 param,t0,___,___
118 param,0,___,___
119 param,10,___,___
120 empilha,0,___,___
121 call,sort,3,___
122 desempilha,0,___,___
123 storeVar,0,i,main
124 label_op,l7,___,___
125 loadVar,main,i,t1
126 slt,t1,10,t2
127 ifFalso,t2,l8,___
128 loadVar,global,vet,t1
129 loadVar,main,i,t2
130 add,t1,t2,t3
131 loadVet,t3,t1,___
132 param,t1,___,___
133 call,output,1,___
134 loadVar,main,i,t2
135 add,t2,1,t3
136 storeVar,t3,i,main
137 jump,l7,___,___
138 label_op,l8,___,___
139 funFim,main,___,___
```

Listing 5.5 – Código Intermediário - Sort

```
0  jump 111
1  addi r0 r27 1
2  sw r0 r27 0
3  sw r29 r31 1
4  addi r30 r30 1
5  addi r30 r30 1
6  addi r30 r30 1
7  addi r30 r30 1
8  addi r30 r30 1
9  addi r30 r30 1
10 addi r30 r30 1
11 lw r29 r1 3
12 sw r29 r1 7
13 lw r29 r1 2
14 lw r29 r2 3
15 add r1 r2 r3
16 lw r3 r1 0
17 sw r29 r1 6
18 lw r29 r1 3
19 addi r1 r2 1
20 sw r29 r2 5
21 lw r29 r1 5
22 lw r29 r2 4
23 slt r1 r2 r3
24 beqz r3 r0 18
25 lw r29 r1 2
26 lw r29 r2 5
27 add r1 r2 r3
```

```
28 lw r3 r1 0
29 lw r29 r2 6
30 slt r1 r2 r3
31 beqz r3 r0 7
32 lw r29 r1 2
33 lw r29 r2 5
34 add r1 r2 r3
35 lw r3 r1 0
36 sw r29 r1 6
37 lw r29 r1 5
38 sw r29 r1 7
39 lw r29 r1 5
40 addi r1 r2 1
41 sw r29 r2 5
42 jump 21
43 lw r29 r1 7
44 addi r1 r28 0
45 lw r29 r31 1
46 jr r31 r0 r0
47 sw r29 r31 1
48 addi r30 r30 1
49 addi r30 r30 1
50 addi r30 r30 1
51 addi r30 r30 1
52 addi r30 r30 1
53 addi r30 r30 1
54 lw r29 r1 3
55 sw r29 r1 5
56 lw r29 r1 5
57 lw r29 r2 4
58 addi r2 r3 -1
59 slt r1 r3 r2
60 beqz r2 r0 48
61 addi r30 r30 1
62 lw r29 r1 2
63 lw r29 r2 5
64 lw r29 r3 4
65 sw r30 r1 0
66 addi r30 r30 1
67 sw r30 r2 0
68 addi r30 r30 1
69 sw r30 r3 0
70 addi r30 r30 1
71 sw r30 r29 0
72 addi r30 r29 0
73 addi r30 r30 1
74 sw r29 r3 4
75 sw r29 r2 3
76 sw r29 r1 2
77 jal 3
78 addi r29 r30 0
79 lw r29 r29 0
80 addi r30 r30 -1
81 lw r30 r3 0
```

```
82  addi r30 r30 -1
83  lw r30 r2 0
84  addi r30 r30 -1
85  lw r30 r1 0
86  sw r29 r28 6
87  lw r29 r1 2
88  lw r29 r4 6
89  add r1 r4 r5
90  lw r5 r1 0
91  sw r29 r1 7
92  lw r29 r1 2
93  lw r29 r4 5
94  add r1 r4 r5
95  lw r5 r1 0
96  lw r29 r4 2
97  lw r29 r5 6
98  add r4 r5 r6
99  sw r6 r1 0
100 lw r29 r1 7
101 lw r29 r4 2
102 lw r29 r5 5
103 add r4 r5 r6
104 sw r6 r1 0
105 lw r29 r1 5
106 addi r1 r4 1
107 sw r29 r4 5
108 jump 56
109 lw r29 r31 1
110 jr r31 r0 r0
111 addi r30 r30 1
112 addi r0 r27 0
113 sw r29 r27 0
114 lw r29 r1 0
115 slti r1 r2 10
116 beqz r2 r0 9
117 input r28
118 lw r0 r1 0
119 lw r29 r2 0
120 add r1 r2 r3
121 sw r3 r28 0
122 lw r29 r1 0
123 addi r1 r2 1
124 sw r29 r2 0
125 jump 114
126 lw r0 r1 0
127 sw r30 r1 0
128 addi r30 r30 1
129 sw r30 r29 0
130 addi r30 r29 0
131 addi r30 r30 1
132 addi r0 r27 10
133 sw r29 r27 4
134 addi r0 r27 0
135 sw r29 r27 3
```

```
136 sw r29 r1 2
137 jal 47
138 addi r29 r30 0
139 lw r29 r29 0
140 addi r30 r30 -1
141 lw r30 r1 0
142 addi r0 r27 0
143 sw r29 r27 0
144 lw r29 r2 0
145 slti r2 r3 10
146 beqz r3 r0 9
147 lw r0 r2 0
148 lw r29 r3 0
149 add r2 r3 r4
150 lw r4 r2 0
151 output r2
152 lw r29 r3 0
153 addi r3 r4 1
154 sw r29 r4 0
155 jump 144
```

Listing 5.6 – Código Assembly - Sort

```
0  00111100000000000000000001101111
1  00000100000110110000000000000001
2  00101100000110110000000000000000
3  00101111101111110000000000000001
4  00000111110111100000000000000001
5  00000111110111100000000000000001
6  00000111110111100000000000000001
7  00000111110111100000000000000001
8  00000111110111100000000000000001
9  00000111110111100000000000000001
10 00000111110111100000000000000001
11 00101011101000010000000000000011
12 00101111101000010000000000000111
13 00101011101000010000000000000010
14 00101011101000100000000000000011
15 00000000001000100001100000000001
16 00101000011000010000000000000000
17 00101111101000010000000000000110
18 00101011101000010000000000000011
19 00000100001000100000000000000001
20 00101111101000100000000000000101
21 00101011101000010000000000000101
22 00101011101000100000000000000100
23 00000000001000100001100000001000
24 00011100011000000000000000010010
25 00101011101000010000000000000010
26 00101011101000100000000000000101
27 00000000001000100001100000000001
28 00101000011000010000000000000000
29 00101011101000100000000000000110
30 00000000001000100001100000001000
```

```
00011100011000000000000000000111
00101011101000010000000000000010
00101011101000100000000000000101
00000000001000100001100000000001
00101000011000010000000000000000
00101111101000010000000000000110
00101011101000010000000000000101
00101111101000010000000000000111
00101011101000010000000000000101
00000100001000100000000000000001
00101111101000100000000000000101
00111100000000000000000000010101
00101011101000010000000000000111
00000100001111000000000000000000
00101011101111111000000000000001
00000011111000000000000000001100
00101111101111111000000000000001
00000111111011110000000000000001
00000111111011110000000000000001
00000111111011110000000000000001
00000111111011110000000000000001
00000111111011110000000000000001
00000111111011110000000000000001
00101011101000010000000000000011
00101111101000010000000000000101
00101011101000010000000000000101
00101011101000100000000000000100
00000100010000111111111111111111
00000000001000110001000000001000
00011100010000000000000000110000
00000111111011110000000000000001
00101011101000010000000000000010
00101011101000100000000000000101
00101011101000110000000000000100
00101111110000010000000000000000
00000111111011110000000000000001
00101111110000100000000000000000
00000111111011110000000000000001
00101111110000110000000000000000
00000111111011110000000000000001
00101111110111010000000000000000
00000111111011101000000000000000
00000111111011110000000000000001
00101111101000110000000000000100
00101111101000100000000000000011
00101111101000010000000000000010
01000000000000000000000000000011
00000111101111100000000000000000
00101011101111010000000000000000
00000111111011111011111111111111
00101011111000011000000000000000
00000111111011111011111111111111
00101011111000010000000000000000
00000111111011111011111111111111
```

```
85  00101011110000010000000000000000
86  00101111101111000000000000000110
87  00101011101000010000000000000010
88  00101011101001000000000000000110
89  00000000001001000010100000000001
90  00101000101000010000000000000000
91  00101111101000010000000000000111
92  00101011101000010000000000000010
93  00101011101001000000000000000101
94  00000000001001000010100000000001
95  00101000101000010000000000000000
96  00101011101001000000000000000010
97  00101011101001010000000000000110
98  00000000100001010011000000000001
99  00101100110000010000000000000000
100 00101011101000010000000000000111
101 00101011101001000000000000000010
102 00101011101001010000000000000101
103 00000000100001010011000000000001
104 00101100110000010000000000000000
105 00101011101000010000000000000101
106 00000100001001000000000000000001
107 00101111101001000000000000000101
108 00111100000000000000000000111000
109 00101011101111110000000000000001
110 00000011111000000000000000001100
111 00000111110111100000000000000001
112 00000100000110110000000000000000
113 00101111101110110000000000000000
114 00101011101000010000000000000000
115 00110100001000100000000000001010
116 00011100010000000000000000001001
117 01001111100000000000000000010111
118 00101000000000010000000000000000
119 00101011101000100000000000000000
120 00000000001000100001100000000001
121 00101100011111000000000000000000
122 00101011101000010000000000000000
123 00000100001000100000000000000001
124 00101111101000100000000000000000
125 00111100000000000000000001110010
126 00101000000000010000000000000000
127 00101111110000010000000000000000
128 00000111110111100000000000000001
129 00101111110111010000000000000000
130 00000111110111010000000000000000
131 00000111110111100000000000000001
132 00000100000110110000000000001010
133 00101111101110110000000000000100
134 00000100000110110000000000000000
135 00101111101110110000000000000011
136 00101111101000010000000000000010
137 01000000000000000000000000101111
138 00000111101111100000000000000000
```

```
00101011101111010000000000000000
00000111110111101111111111111111
00101011110000010000000000000000
00000100000110110000000000000000
00101111101110110000000000000000
00101011110100010000000000000000
00110100010000110000000000001010
00011100011000000000000000001001
00101000000000100000000000000000
00101011101000110000000000000000
00000000010000110010000000000001
00101000100000100000000000000000
01010000010000000000000000000000
00101011101000110000000000000000
00000100011001000000000000000001
00101111101001000000000000000000
00111100000000000000000010010000
```

Listing 5.7 – Código Binário - Sort

5.3 Exemplo 3 - Fatorial

Por fim, o último exemplo trás os códigos intermediário, assembly e binário gerados para um programa de cálculo de fatorial na forma recursiva apresentado no Listing 5.8.

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

Listing 5.8 – Código C- Fatorial

```
0  funInicio,fatorial,___,___
1  allocaMemVar,fatorial,n,___
2  loadVar,fatorial,n,t0
3  set,t0,1,t1
4  ifFalso,t1,l0,___
```

```
5  move,1,$rf,___
6  jump,l1,___,___
7  label_op,l0,___,___
8  loadVar,fatorial,n,t0
9  loadVar,fatorial,n,t1
10 sub,t1,1,t2
11 param,t2,___,___
12 empilha,0,___,___
13 empilha,2,___,___
14 call,fatorial,1,___
15 desempilha,2,___,___
16 desempilha,0,___,___
17 mult,t0,$rf,t1
18 move,t1,$rf,___
19 label_op,l1,___,___
20 funFim,fatorial,___,___
21 funInicio,main,___,___
22 allocaMemVar,main,n,___
23 allocaMemVar,main,fat,___
24 call,input,0,___
25 storeVar,$rf,n,main
26 loadVar,main,n,t0
27 param,t0,___,___
28 empilha,0,___,___
29 call,fatorial,1,___
30 desempilha,0,___,___
31 storeVar,$rf,fat,main
32 loadVar,main,fat,t0
33 param,t0,___,___
34 call,output,1,___
35 funFim,main,___,___
```

Listing 5.9 – Código Intermediário - Fatorial

```
0  jump 31
1  sw r29 r31 1
2  addi r30 r30 1
3  addi r30 r30 1
4  lw r29 r1 2
5  seti r1 r2 1
6  beqz r2 r0 2
7  addi r0 r28 1
8  jump 29
9  lw r29 r1 2
10 lw r29 r2 2
11 addi r2 r3 -1
12 sw r30 r1 0
13 addi r30 r30 1
14 sw r30 r3 0
15 addi r30 r30 1
16 sw r30 r29 0
17 addi r30 r29 0
18 addi r30 r30 1
19 sw r29 r3 2
```

```
jal 1
addi r29 r30 0
lw r29 r29 0
addi r30 r30 -1
lw r30 r3 0
addi r30 r30 -1
lw r30 r1 0
mult r1 r28 r2
addi r2 r28 0
lw r29 r31 1
jr r31 r0 r0
addi r30 r30 1
addi r30 r30 1
input r28
sw r29 r28 0
lw r29 r1 0
sw r30 r1 0
addi r30 r30 1
sw r30 r29 0
addi r30 r29 0
addi r30 r30 1
sw r29 r1 2
jal 1
addi r29 r30 0
lw r29 r29 0
addi r30 r30 -1
lw r30 r1 0
sw r29 r28 1
lw r29 r1 1
output r1
```

Listing 5.10 – Código Assembly - Fatorial

```
0  00111100000000000000000000011111
1  00101111101111110000000000000001
2  00000111110111100000000000000001
3  00000111110111100000000000000001
4  00101011101000010000000000000010
5  01100100001000100000000000000001
6  00011100010000000000000000000010
7  00000100000111000000000000000001
8  00111100000000000000000000011101
9  00101011101000010000000000000010
10 00101011101000100000000000000010
11 00000100010000111111111111111111
12 00101111110000010000000000000000
13 00000111110111100000000000000001
14 00101111110000110000000000000000
15 00000111110111100000000000000001
16 00101111110111010000000000000000
17 00000111110111010000000000000000
18 00000111110111100000000000000001
19 00101111101000110000000000000010
20 01000000000000000000000000000001
```

```
21 00000111101111100000000000000000
22 00101011101111010000000000000000
23 00000111110111101111111111111111
24 00101011110000110000000000000000
25 00000111110111101111111111111111
26 00101011110000010000000000000000
27 00000000001111000001000000000011
28 00000100010111000000000000000000
29 00101011101111110000000000000001
30 00000011111000000000000000001100
31 00000111110111100000000000000001
32 00000111110111100000000000000001
33 01001111100000000000000000010111
34 00101111101111000000000000000000
35 00101011101000010000000000000000
36 00101111110000010000000000000000
37 00000111110111100000000000000001
38 00101111110111010000000000000000
39 00000111110111010000000000000000
40 00000111110111100000000000000001
41 00101111101000010000000000000010
42 01000000000000000000000000000001
43 00000111101111100000000000000000
44 00101011101111010000000000000000
45 00000111110111101111111111111111
46 00101011110000010000000000000000
47 00101111101111000000000000000001
48 00101011101000010000000000000001
49 01010000001000000000000000000000
```

Listing 5.11 – Código Binário - Fatorial

6 Conclusão

Com desenvolvimento desse projeto, foi possível conhecer todas as etapas do processo de compilação, desde a análise até a síntese do código de máquina para a arquitetura desenvolvida, o que proporcionou uma grande aquisição de conhecimento, já que foi possível entender claramente como um código em uma linguagem de programação de mais alto nível torna-se um código executável. Os objetivos propostos foram atingidos, pois conseguiu-se realizar a análise do código de entrada de maneira satisfatória encontrando erros de caráter léxico, sintático e semântico, além da construção dos produtos finais da fase de análise que são a tabela de símbolos e a árvore sintática, esta última que constitui uma peça chave para a fase de síntese. Na fase de síntese, foi possível fazer a geração do código intermediário de maneira adequada a partir da árvore sintática de modo que o código gerado contivesse informações suficientes para a geração do código assembly mas que ainda assim mantivesse as características de uma representação intermediária. Na geração do código assembly conseguiu-se traduzir de maneira apropriada as quádruplas empregando as instruções e registradores disponíveis na arquitetura alvo cobrindo desde a síntese de códigos mais simples até códigos que contivessem uso de vetores e chamadas recursivas. Por fim, na geração do código binário foi possível fazer uma conversão bastante direta, com base no nome das instruções e seus códigos binários e também da conversão da representação decimal do operandos, endereços e registradores para a representação binária.

Acredito que a maior dificuldade encontrada no desenvolvimento do projeto foi durante a geração do código assembly, onde é necessário fazer o gerenciamento da alocação das varáveis dentro da memória de dados do processador, sendo necessário atribuir às variáveis declaradas um endereço. Isto é uma tarefa bastante direta quando se trata de variáveis globais, pois estas possuem um espaço reservado dentro da memória, mas no caso das variáveis locais o tratamento é mais complexo, pois é necessário levar em consideração as ativações de função que podem ocorrer durante a execução do programa sendo necessário associar as variáveis alocadas à execução corrente, por esse motivo foi adotado o gerenciamento da memória utilizando a abordagem por pilha.

Por fim, pode-se concluir que o desenvolvimento do projeto atendeu aos requisitos propostos, permitindo que a arquitetura anteriormente desenvolvida seja melhor explorada através da implementação de programas mais complexos que antes não eram possíveis

em linguagem de máquina, com apoio do compilador C- desenvolvido.

Referências

1 LOUDEN, K.; SILVA, F. *Compiladores - Princípios e Práticas*. Pioneira Thomson Learning, 2004. ISBN 9788522104222. Disponível em: <https://books.google.com.br/books?id=a4tuyCBcM_MC>. Citado 5 vezes nas páginas 7, 26, 28, 29 e 52.

2 PATTERSON, D. A.; HENNESY, J. L. *Computer Organization and Design*. 5th edition. ed. Waltham/MA, EUA: Morgan Kaufmann, 2007. Citado na página 8.