Everton da Silva Coelho R.A.:101937

Implementação de um compilador em python para a linguagem C-

São José dos Campos - Brasil

Julho de 2018

Everton da Silva Coelho R.A.:101937

Implementação de um compilador em python para a linguagem C-

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

Docente: Prof. Dr. Luiz Eduardo Galvão Martins

Universidade Federal de São Paulo - UNIFESP

Instituto de Ciência e Tecnologia - Campus São José dos Campos

São José dos Campos - Brasil

Julho de 2018

Sumário

1 Introdução

Vivendo na época atual considerada a era da informação, não é difícil encontrar diversas máquinas capazes de tratar grandes quantidades de dados gerados pela interação social, por imagens, sensores, entre outros. Essas máquinas responsáveis pelo armazenamento, processamento e cálculo dos dados são chamadas de computadores. No cerne dos computadores estão os processadores, responsáveis por transformar os dados contidos na máquina em informação para o usuário.

A princípio os computadores eram programados diretamente com o código de máquina, o que é um processo lento e custoso para os programadores e com altas chances de erros. Para contornar essas dificuldades projetou-se uma maneira mais fácil de assimilar a linguagem de máquina, resultando em um compilador. O compilador faz a tradução de uma linguagem textual de fácil entendimento para uma linguagem de máquina, essa ferramenta é necessária para grande parte das aplicações de alto nível atuais.

Com o objetivo de entender o funcionamento foi desenvolvido, no Laboratório de Sistemas Computacionais: Compiladores, um compilador para C- utilizando a linguagem python. O compilador traduz um código em C- para o processador projetado em laboratórios anteriores, atentendo as particularidades do mesmo. Neste relátório são abordados os passos do desemvolvimento que estão organizados da seguinte maneira:

1. 2. O Processador: Neste tópico é apresentado o processador desenvolvido no Laboratório de sistemas computacionais: Arquitetura e Organização de computadores, suas especificações e alguns conceitos básicos para o entendimento do mesmo.
2. 3. Compilador(Fase de Análise): Neste encontra-se como foram desenvolvidas a análise léxica, sintática e semântica do compilador.
3. 4. Compilador(Fase de Síntese): Criação do código intermediário, e passagem para o assembly(objeto) e conversão do assembly para o código binário que será posto no processador.
4. 5. Exemplos Gerados: Códigos exemplos compilados.
5. 6. Conclusão: Considerações finais a respeito do desenvolvimento do projeto.

2 O Processador

2.1 Sistemas computacionais

Um sistema computacional é um conjunto de dispositivos eletrônicos que são utilizados para processar informações, compostos pela união de hardware e software. Um exemplo de sistema computacional é um celular, que possui funções para tirar fotos, gravar vídeos, sons e comunicação com outros dispositivos, ou seja, ele captura os dados dos sensores e transforma em informação, possibilitando a comunicação com o usuário(1).

Focando no Hardware de um sistema computacional, ele consiste em três principais partes:

1. O processador ou CPU: é a unidade responsável pela execução de instruções requisitadas por um sistema, como cálculos aritméticos, lógicos, operações de entrada e saída e manipulação de dados.
2. Memória principal: é o dispositivo capaz de armazenar dados gerais do sistema. Basicamente o processador acessa a memória para buscar informações e depois de processá-las manda sinais para outras partes do sistema realizando atividades pedidas.
3. Módulo de entrada e saída: responsável pela interface entre a máquina e o usuário, ele facilita a comunicação do programador com o sistema computacional.

2.2 *Central Processor Unit* (CPU)

Responsável por executar as instruções, o processador é constituído das seguintes partes:

- ALU (Unidade Lógica Aritmética): realiza cálculos como soma e subtração e exeuta comparações lógicas como *and* e *or*.
- Memória de instruções: armazena as instruções que serão executadas.
- Banco de registradores: memória de acesso rápido que armazena os operandos da instrução;
- Unidade de Controle: controla as outras partes do processador para a execução correta das instruções.
- *Program counter*: Armazena o endereço para a próxima instrução a ser executada(2).

2.3 MIPS

MIPS (*Microprocessor Without Interlocked Pipeline Stages*) é um processador de arquitetura RISC que trabalha de forma monocíclica, ou seja, ele executa uma instrução por ciclo de clock. Devido a isto todas as instruções possuem tempo igual de execução. Como é um processador RISC ele herda algumas de suas características: Quantidade reduzida de instruções, instruções de tamanho fixo, pouca redundância de instruções, data path mais simples(2).

2.4 Tipos de endereçamento

Para acessar os dados o processador precisa saber onde esses dados estão localizados, uma forma de saber o caminho desses dados é definindo um tipo de endereçamento, alguns tipos são listados a seguir:

1. Endereçamento por registrador: o operando é o conteúdo de um registrador especificado na instrução.
2. Endereçamento imediato: o conteúdo do operando está incorporado na instrução.
3. Endereçamento indireto a registrador: o operando está no endereço especificado dentro de um registrador repassado pela instrução(3).

2.5 Verilog

Verilog é uma linguagem de descrição de hardware(Hardware Description Language - HDL) que permite descrever sistemas digitais, analógicos ou híbridos em vários níveis de abstração(4).

O Verilog difere das outras linguagens pela maneira como é executado. Diferente de uma linguagem procedural um projeto Verilog consiste na separação hierárquica de módulos que contém conexões e registradores. Esses módulos contém blocos que são executados em paralelo, há também a possibilidade de executar processos sequenciais dentro de blocos "begin/end"(5).

2.6 FPGA

FPGA ou Field Programmable Gate Array é um tipo de circuito integrado que possui uma lógica programável, pode ser configurado após a sua fabricação, internamente possui um grande arranjo de células lógicas ou blocos lógicos configuráveis contidos em um único circuito integrado. É constituído basicamente de três partes:

2.7 Quartus

Quartus prime 16.1 é um software de design produzido pela Altera. Ele permite a análise e síntese de linguagem de descrição de hardware e desenhos, permite que o desenvolvedor compile seus projetos, realize análise de timing, simule a reação de um projeto a diferentes estímulos, e configure o dispositivo de destino. O Quartus inclui uma implementação do VHDL e Verilog para descrição de hardware, edição visual de circuitos lógicos, e simulação de formas de onda(6).

2.8 Processador Desenvolvido

Tomando como base a arquitetura MIPS, o processador utilizado, chamado de ProcessorE, trabalha de forma monocíclica (uma instrução por ciclo de clock) e possui instruções de tamanho fixo e realiza operações principalmente com registradores, de forma que seu endereçamento é direto por registrador. Possui instruções lógicas e aritméticas, de controle, assim como instruções para manipular a memória e de entrada e saída de dados. O formato das instruções pode ser visto na (Tabela 1).

Tabela 1 – Formato das Instruções

F1					
Opcode	RD	RS	RT	-	-
[31:26]	[25:21]	[20:16]	[15:11]	[10:6]	[5:0]
F2					
Opcode	RD	RS	IMT		
[31:26]	[25:21]	[20:16]	[15:0]		
F3					
Opcode	RD	endereço/IMT/IO			
[31:26]	[25:21]	[20:0]			
F4					
Opcode	-				
[31:26]	[25:0]				

Fonte: O Autor

Cada instrução possui o tamanho de 32 bits com 6 bits destinados à opcode, o que possibilita a criação de 64 instruções diferentes, tornando possível a expansão para futuras instruções. Dos 32 bits, blocos com 5 bits são destinados ao endereço dos registradores; os 5 bits possibilitam ter acesso aos 32 registradores de uso geral, pois com 5 bits obtemos 32 endereços distintos.

O processador possui 25 instruções básicas. Essas instruções foram classificadas de acordo com sua finalidade e estão descritas na Tabela 2.

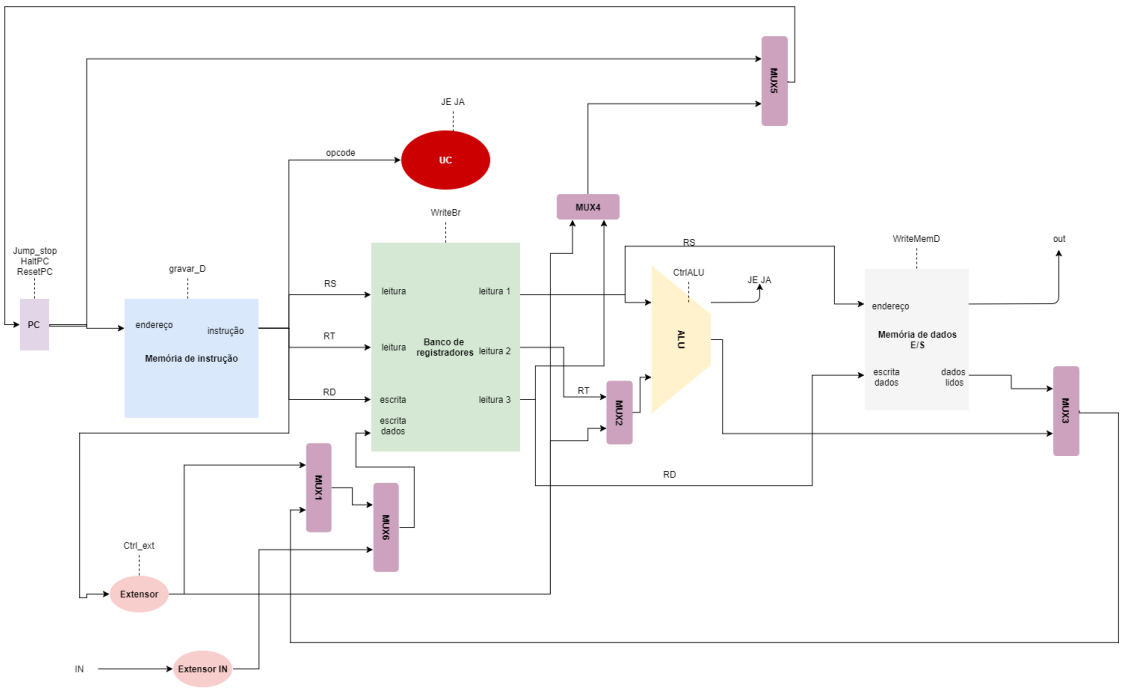
Abaixo, na (Figura 3), está o esquemático do processador referenciado.

Tabela 2 – Conjunto de instruções

Tipo	Instrução	Opcode Dec	Opcode Bin	Operação	Formato
Aritiméticas	ADD	000001	000001	RD <= RS + RT	F1
	ADDI	000002	000010	RD <= RS + IMT	F2
	SUB	000003	000011	RD <= RS - RT	F1
	SUBI	000004	000100	RD <= RS - IMT	F2
	MULT	000005	000101	RD <= RS * RT	F1
Lógicas	AND	000006	000110	RD <= RS and RT	F1
	OR	000007	000111	RD <= RS or RT	F1
	XOR	000008	001000	RD <= RS xor RT	F1
	NOT	000009	001001	RD <= not(RS)	F2
Deslocamento	SHR	000010	001010	RD <= RS »RT	F1
	SHL	000011	001011	RD <= RS «RT	F1
Movimentação de dados	LOAD	000012	001100	RD <= mem_raw[RS]	F2
	LOADI	000013	001101	RD <= IMT	F3
	STORE	000014	001110	mem_raw[RS]<= RD	F2
	MOVE	000015	001111	RD <= RS	F2
Desvios	JMPI	000016	010000	PC <= IMT	F3
	JMP	000017	010001	PC <= RD	F3
	JE	000018	010010	if RS == RT, PC <= RD	F1
	JNE	000019	010011	if RS != RT, PC <= RD	F1
	JA	000020	010100	if RS >RT, PC <= RD	F1
	JNA	000021	010101	if RS =<RT, PC <= RD	F1
Entrada e saída	IN	000022	010110	RD <= entrada de dados	F3
	OUT	000023	010111	saída de dados <= RD	F3
Controle	NOP	000024	011000	nenhuma ação	F4
	HALT	000025	011001	parar processamento	F4

Fonte: O Autor

Figura 1 – Esquematico ProcessorE



Fonte: O autor

3 Compilador

Um compilador é uma aplicação que traduz um programa escrito em uma linguagem para uma outra linguagem geralmente de mais baixo nível. O compilador projetado e desenvolvido neste laboratório faz a conversão de um programa em linguagem c- para a linguagem de máquina respectiva ao processador usado. O compilador pode ser dividido em vários blocos que consistem na Fase de Análise: Análise Léxica, Análise Sintática, Análise Semântica, e Fase de Síntese: Geração do código intermediário, Geração do código *Assembly*, Geração do Código Executável.

3.1 Fase de Análise

Na fase se análise utilizou-se o ANTLR (*ANother Tool for Language Recognition*) que é uma ferramenta de leitura e processamento de texto que a partir de uma gramática consegue contruir e analisar arvores sintáticas.

3.1.1 Análise Léxica

Na análise léxica é feita a varredura do código para a separação e verificação de *tokens*. Através de expressões regulares os *tokens* são separados e analisados, retornando se é uma sequência de caracteres válida ou não. Nessa fase é feito o reconhecimento de palavras reservadas(if, else, while), simbolos aritméticos e caracteres especiais. As expressões regulares são postas no mesmo arquivo da gramática(subseção 3.1.2) para que o ANTLR construa a arvore.

Os *tokens* gerados por um programa teste podem ser vistos a seguir:

```
1  /* programa teste */
2  void main(void)
3  {       int x; int y;
4
5      x = 12;
6      y = 15 + x;
7  }
8  /* Tokens gerados:
9  4 : void
10 4 : main
11 4 : (
12 4 : void
13 4 : )
14 5 : {
15 5 : int
16 5 : x
17 5 : ;
18 5 : int
19 5 : y
```

```
5 : ;
7 : x
7 : =
7 : 12
7 : ;
8 : y
8 : =
8 : 15
8 : +
8 : x
8 : ;
9 : }
10 : <EOF>
*/
```

3.1.2 Análise Sintática

Na análise sintática ocorre o processo em que o compilador verifica se o programa analisado obedece as regras da gramática estabelecida. Após recuperar os *tokens* fornecidos pela análise léxica o analisador sintático compara as entradas através de derivações de acordo com a gramática. Cada derivação é uma regra da gramática representada em BNF(Formalismo de Backus-Naur), o ANTLR faz a leitura da gramática no formato BNF e após faz a derivação das regras utilizando os *tokens* de entrada.

Gramática para cminus:

```
grammar cminus;


/* Parser Rules */

 programa
    : (decl+=declaracao)+
    ;


  declaracao
    : var_declaracao
    | fun_declaracao
    ;

  var_declaracao
    : tipo_especificador ID SEMI
    | tipo_especificador ID LSBRACKET NUM RSBRACKET SEMI
    ;

  tipo_especificador
    : INT
    | VOID
    ;

  fun_declaracao
    : tipo_especificador ID LPAREN params RPAREN composto_decl
    ;

  params
```

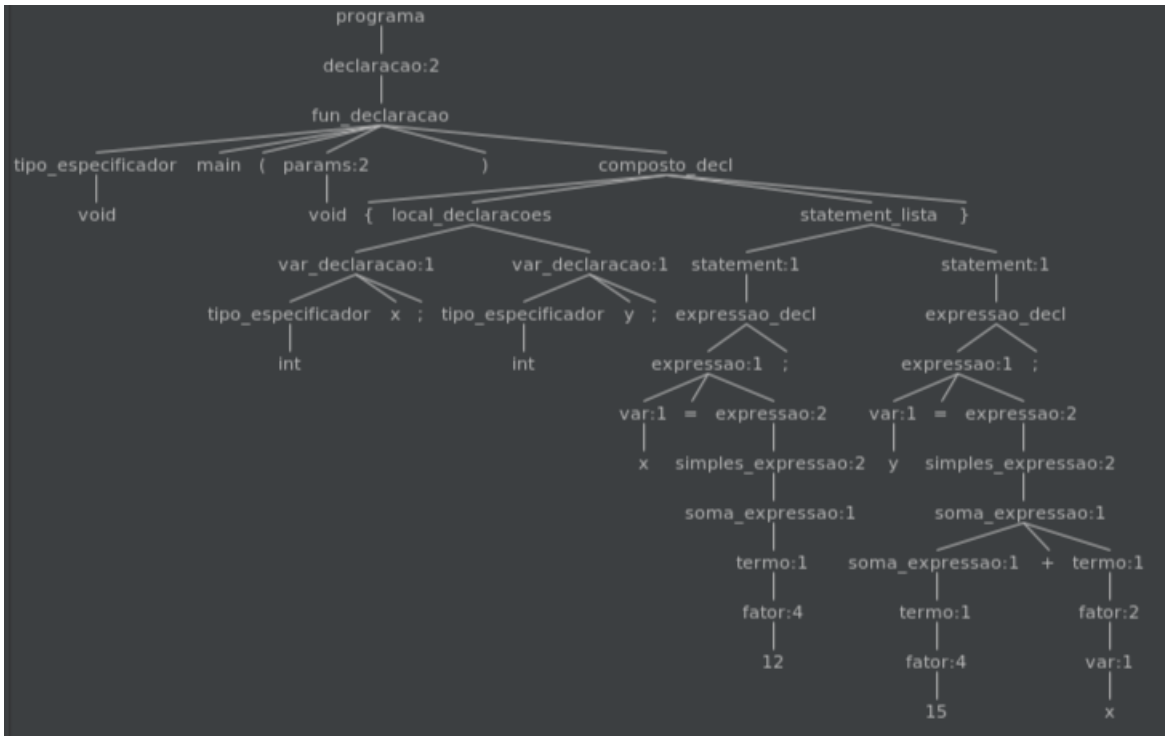
```
31    : param_lista
32    | VOID
33    ;
34
35  param_lista
36    : param_lista COMMA param
37    | param
38    ;
39
40  param
41    : tipo_especificador ID
42    | tipo_especificador ID LSBRACKET RSBRACKET
43    ;
44
45  composto_decl
46    : LCBRACKET (l_decl+=local_declaracoes)* (stm_list+=statement_lista)* RCBRACKET
47    ;
48
49  local_declaracoes
50    : (var_decl+=var_declaracao)+
51    ;
52
53  statement_lista
54    : (stms+=statement)+
55    ;
56
57  statement
58    : expressao_decl
59    | composto_decl
60    | selecao_decl
61    | iteracao_decl
62    | retorno_decl
63    ;
64
65  expressao_decl
66    : expressao? SEMI
67    ;
68
69  selecao_decl
70    : IF LPAREN condicao=expressao RPAREN LCBRACKET corpoIF+=statement* RCBRACKET (ELSE
        LCBRACKET corpoElse+=statement* RCBRACKET)?
71    ;
72
73  iteracao_decl
74    : WHILE LPAREN expressao RPAREN statement
75    ;
76
77  retorno_decl
78    : RETURN SEMI
79    | RETURN expressao SEMI
80    ;
81
82  expressao
83    : var ASSIGN expressao
84    | simples_expressao
85    ;
86
87  var
88    : ID
89    | ID LSBRACKET expressao RSBRACKET
```

```
90      ;
91
92    simples_expressao
93      : esquerda=soma_expressao relacional=(LETHAN| LT| GT| GETHAN| EQ| DF) direita=
            soma_expressao
94      | operacao=soma_expressao
95      ;
96
97    soma_expressao
98      : soma_expressao op=('+'|'-') termo
99      | termo
100     ;
101
102
103   termo
104     : termo op=('/'|'*') fator
105     | fator
106     ;
107
108
109   fator
110     : LPAREN expressao RPAREN
111     | var
112     | ativacao
113     | NUM
114     ;
115
116   ativacao
117     : ID LPAREN (arg_list+=expressao COMMA)* (arg_list+=expressao) RPAREN
118     | ID LPAREN RPAREN
119     ;
120
121     /* Lexer Rules */
122
123       //RESERVED_WORD : 'else' | 'if' | 'int' | 'return' | 'void' | 'while' ;
124       ELSE : 'else' ;
125       IF : 'if' ;
126       INT : 'int' ;
127       RETURN : 'return' ;
128       VOID : 'void' ;
129       WHILE : 'while';
130       LETHAN : '<=' ;
131       GETHAN : '>=' ;
132       ASSIGN : '=' ;
133       EQ : '==' ;
134       DF : '!=' ;
135       LT : '<' ;
136       GT : '>' ;
137       PLUS : '+' ;
138       MINUS : '-' ;
139       TIMES : '*' ;
140       OVER : '/' ;
141       LPAREN : '(';
142       RPAREN : ')';
143       SEMI : ';' ;
144       COMMA : ',' ;
145       LCBRACKET : '{' ;
146       RCBRACKET : '}' ;
147       LSBRACKET : '[';
148       RSBRACKET : ']' ;
```

```
149
150     //tokens {ELSE, IF, INT, RETURN, VOID, WHILE}
151
152     ID : [a-zA-Z]+ ; // match identifiers
153     NUM : [0-9]+ ; // match integers
154
155     BLOCK_COMMENT: '/*' .*? '*/' -> skip
156     ;
157     //LINE_COMMENT: '//' ~[\r\n]* -> skip
158     //;
159     // Whitespace
160     WS : [ \t\r\n\f]+ -> skip ;
```

A maneira que o ANTLR utiliza para representar as derivações é através de uma árvore, conhecida como árvore sintática, como as regras da gramática são recursivas uma árvore se torna uma boa forma para representar essas derivações. Se todas as derivações ocorrerem de forma a chegar num nó folha a arvore é montada e retornada, caso contrário o programa de entrada possui algum erro sintático e esse erro é retornado.

Figura 2 – Árvore de análise sintática do programa teste



Fonte: O autor

3.1.3 Análise Semântica

Após passar por duas etapas de análise o programa de entrada ainda precissa de uma análise mais específica de erros que as etapas anteriores não conseguem reconhecer. A análise semantica é responsável por reconhecer esses erros residuais, que são erros de

significado do código, ou seja, atribuições erradas, variáveis não declaradas entre outros. Os principais erros que devem ser reconhecidos na análise sintática para c- são:

- Reconhecimento de variáveis não declaradas;
- Atribuições invalidas entre tipos de dados;
- Decleração inválida de variavel como *void*;
- Declaração da mesma variável mais de uma vez;
- Chamada de função não declarada;
- Função *main()* não declarada;
- Declaração inválida de uma variável com nome de função.

A análise semântica é realizada percorrendo a árvore gerada na análize sintática de forma recursiva, esse acesso a árvore é feito utilizando um padrão de projeto comportamental chamado de *visitor pattern* que permite criar novas operações sem alterar a classe dos elementos que ele acessa. Durante a visita aos nós da árvore são inseridos elementos em uma tabela *hash*, chamada de tabela de símbolos, que auxilia na detecção dos erros. Essa tabela é acessada sempre que é preciso checar e fazer uma comparação com um parametro que já foi acessado, um exemplo de tabela é mostrado logo abaixo.

Figura 3 – Tabela de símbolos código gcd

Key	Name	Scope	Lines	Id Type	Data Type	Pos Mem	Qtd Args	Args
a	a	global	4	var[3]	int	[0, 2]		
b	b	global	5	var	int	3		
gcd	gcd	global	7, 10	funct	int	-1	2	['u', 'v']
gcd.u	u	gcd	7, 9, 10	var	int	4		
gcd.v	v	gcd	7, 9, 10	var	int	5		
dum	dum	global	14	funct	int	-1	0	[]
main	main	global	18	funct	void	-1	0	[]
main.x	x	main	19, 20	var	int	6		
main.y	y	main	19, 20	var	int	7		
input	input	main	20	sys_call	int	-1		
output	output	main	21	sys_call	int	-1		

Fonte: O autor

3.2 Fase de Síntese

3.2.1 Geração do código intermediário

Para a melhor compreensão da tradução do código em c- para a linguagem de máquina é feita a geração de um código intermediário. O código intermediário neste projeto é montado através de uma lista em que cada elemento dessa lista é uma lista com 4 elementos, de forma que o primeiro elemento é a condição de tratamento para os outros 3 elementos, os outros 3 elementos podendo ser variáveis, *labels* ou registradores temporários utilizados para realizar operações. O intermediário representa as operações na forma do código de 3 endereços e é uma forma de simplificar a lógica do programa de entrada. Nele são adicionadas *labes* e tratamentos para condicionais e saltos.

Para gerar o código intermadiário a árvore sintática é percorrida novamente, através de um *visitor*, porém realizando novas operações a cada nó e adicionando uma nova linha no intermediário quando necessário. Durante o acesso são feitas comparações com os dados da tabela de símbolos de modo a manter a coerência dos dados.

3.2.2 Geração do código *Assembly*

O código assembly é o último passo antes da geração para o código em binário, que irá ser executado no processador. Para a geração do assembly a lista de intermediários é percorrida e traduzida para o conjunto de instruções do processador específico. durante essa etapa também é feito o acesso a tabela de símbolos para a comparação dos dados e é gerado uma lista contendo as linhas do assembly.

Devido a arquitetura do processador utilizado essa etapa deve ser realizada com cautela para que a lógica não se perca durante o processo de tradução e adaptação para as respectivas instruções do processador.

3.2.3 Gerenciamento de memória

A gerenciamento de memória do código é realizado através de duas memórias contidas na memória de dados, sendo que uma representa uma pilha e a outra uma memória estática. As funções são armazenadas na memória estática e a pilha é utilizada para empilhar paramêtros utilizados quando se tem uma chamada de função, sendo assim, permitindo recursões e chamadas de função.

4 Exemplos Gerados

```
+--------+--------+---------+----------+-----------+-------------+-----------+------------+----------
| Key    | Name   | Scope   | Lines    | Id Type   | Data Type   | Pos Mem   | Qtd Args
     | Args       |
+========+========+=========+==========+===========+=============+===========+============+==========
| a      | a      | global  | 4        | var[3]    | int         | [0, 2]    |
               |            |
+--------+--------+---------+----------+-----------+-------------+-----------+------------+----------
| b      | b      | global  | 5        | var       | int         | 3         |
               |            |
+--------+--------+---------+----------+-----------+-------------+-----------+------------+----------
| gcd    | gcd    | global  | 7, 10    | funct     | int         | -1        | 2
            | ['u', 'v'] |
+--------+--------+---------+----------+-----------+-------------+-----------+------------+----------
| gcd.u  | u      | gcd     | 7, 9, 10 | var       | int         | 4         |
               |            |
+--------+--------+---------+----------+-----------+-------------+-----------+------------+----------
| gcd.v  | v      | gcd     | 7, 9, 10 | var       | int         | 5         |
               |            |
+--------+--------+---------+----------+-----------+-------------+-----------+------------+----------
| dum    | dum    | global  | 14       | funct     | int         | -1        | 0
            | []         |
+--------+--------+---------+----------+-----------+-------------+-----------+------------+----------
| main   | main   | global  | 18       | funct     | void        | -1        | 0
            | []         |
+--------+--------+---------+----------+-----------+-------------+-----------+------------+----------
| main.x | x      | main    | 19, 20   | var       | int         | 6         |
               |            |
+--------+--------+---------+----------+-----------+-------------+-----------+------------+----------
| main.y | y      | main    | 19, 20   | var       | int         | 7         |
               |            |
+--------+--------+---------+----------+-----------+-------------+-----------+------------+----------
| input  | input  | main    | 20       | sys_call  | int         | -1        |
               |            |
+--------+--------+---------+----------+-----------+-------------+-----------+------------+----------
| output | output | main    | 21       | sys_call  | int         | -1        |
               |            |
+--------+--------+---------+----------+-----------+-------------+-----------+------------+----------


testes/pudim.txt :

/* Um programa para calcular o mdc
```

```
30     segundo o algoritmo de Euclides. */
31
32  int a[3];
33  int b;
34
35  int gcd (int u, int v)
36  {
37          if (v == 0){ return u; }
38          else{ return gcd(v,u-u/v*v);}
39
40          /* u-u/v*v == u mod v */
41  }
42  int dum(void){
43
44  }
45
46  void main(void)
47  {       int x; int y;
48          x = input(); y = input();
49          output(gcd(x,y));
50  }
51
52  0  : (function, gcd, , )
53  1  : (equal_to, v, 0, t1)
54  2  : (jump_if_false, t1, L1, )
55  3  : (return, u, , )
56  4  : (go_to, L2, , )
57  5  : (label, L1, , )
58  6  : (arg, v, , )
59  7  : (division, u, v, t2)
60  8  : (multiplication, t2, v, t3)
61  9  : (subtraction, u, t3, t4)
62  10  : (arg, t4, , )
63  11  : (function_call, gcd, 2, )
64  12  : (assign_ret, t5, RT, )
65  13  : (return, t5, , )
66  14  : (label, L2, , )
67  15  : (function, dum, , )
68  16  : (function, main, , )
69  17  : (sys_call, input, , )
70  18  : (assign_ret, t6, RT, )
71  19  : (assign, x, t6, )
72  20  : (sys_call, input, , )
73  21  : (assign_ret, t7, RT, )
74  22  : (assign, y, t7, )
75  23  : (arg, x, , )
76  24  : (arg, y, , )
77  25  : (function_call, gcd, 2, )
78  26  : (assign_ret, t8, RT, )
79  27  : (arg, t8, , )
80  28  : (sys_call, output, 1, )
81
82
83       0: loadi $r0 0
84       1: loadi $stp 0
85       2: loadi $ra 0
86       3: jmp main
87  gcd:
88       4: pop $r1 $stp
89       5: loadi $rl 4
```

```
90        6: store $r1 $rl
91        7: pop $r1 $stp
92        8: loadi $rl 5
93        9: store $r1 $rl
94        10: loadi $rl 5
95        11: load $r1 $rl
96        12: loadi $r2 0
97        13: eq $r3 $r1 $r2
98        14: je $r0 $r3 L1
99        15: loadi $r1 4
100       16: load $rt $r1
101       17: jmp $ra
102       18: jmp L2
103   L1.
104       19: loadi $rl 4
105       20: load $r1 $rl
106       21: loadi $rl 5
107       22: load $r2 $rl
108       23: div $r3 $r1 $r2
109       24: loadi $rl 5
110       25: load $r1 $rl
111       26: mult $r2 $r3 $r1
112       27: loadi $rl 4
113       28: load $r1 $rl
114       29: sub $r3 $r1 $r2
115       30: push $ra $stp
116       31: addi $stp $stp 1
117       32: loadi $rl 4
118       33: load $r1 $rl
119       34: push $r1 $spt
120       35: addi $stp $stp 1
121       36: loadi $rl 5
122       37: load $r1 $rl
123       38: push $r1 $spt
124       39: addi $stp $stp 1
125       40: push $r3 $stp
126       41: addi $stp $stp 1
127       42: loadi $rl 5
128       43: load $r1 $rl
129       44: push $r1 $stp
130       45: addi $stp $stp 1
131       46: jal gcd
132       47: subi $stp $stp 1
133       48: pop $r1 $stp
134       49: loadi $rl 5
135       50: store $r1 $rl
136       51: subi $stp $stp 1
137       52: pop $r1 $stp
138       53: loadi $rl 4
139       54: store $r1 $rl
140       55: subi $stp $stp 1
141       56: pop $ra $stp
142       57: move $r1 $rt
143       58: move $rt $r1
144       59: jmp $ra
145   L2.
146   dum:
147   main:
148       60: in $rt
149       61: move $r1 $rt
```

```
150      62: loadi $r2 6
151      63: store $r1 $r2
152      64: in $rt
153      65: move $r1 $rt
154      66: loadi $r2 7
155      67: store $r1 $r2
156      68: push $ra $stp
157      69: addi $stp $stp 1
158      70: loadi $rl 6
159      71: load $r1 $rl
160      72: push $r1 $spt
161      73: addi $stp $stp 1
162      74: loadi $rl 7
163      75: load $r1 $rl
164      76: push $r1 $spt
165      77: addi $stp $stp 1
166      78: loadi $rl 7
167      79: load $r1 $rl
168      80: push $r1 $stp
169      81: addi $stp $stp 1
170      82: loadi $rl 6
171      83: load $r1 $rl
172      84: push $r1 $stp
173      85: addi $stp $stp 1
174      86: jal gcd
175      87: subi $stp $stp 1
176      88: pop $r1 $stp
177      89: loadi $rl 7
178      90: store $r1 $rl
179      91: subi $stp $stp 1
180      92: pop $r1 $stp
181      93: loadi $rl 6
182      94: store $r1 $rl
183      95: subi $stp $stp 1
184      96: pop $ra $stp
185      97: move $r1 $rt
186      98: out $r1
```

5 Conclusão

Grandes dificuldades foram emfrentadas durante o desemvolvimento do projeto. A que teve mais impacto foram as enfrentadas durante a criação do assembly, pois, soluções custosas foram necessárias como adição de novas instruções no processador e adição de pilha para tornar a recursão possível. Porém a experiência da construção de um compilador faz com que se consiga entender de uma forma bem mais profunda o funcionamento de um sitema computacional.

Referências

1 O que é um sistema computacional. Acessado em 05/04/2017. Disponível em: <https://www.portaleducacao.com.br/conteudo/artigos/informatica/o-que-e-um-sistema-computacional/46697>. Citado na página 4.

2 PATTERSON, D. A.; HENNESY, J. L. *Computer Organization and Design*. 5th edition. ed. Waltham/MA, EUA: Morgan Kaufmann, 2007. Citado 2 vezes nas páginas 4 e 5.

3 ENDEREçAMENTO de memória. Acessado em 06/04/2017. Disponível em: <http://usuarios.upf.br/~appel/arquiI/endereca.pdf>. Citado na página 5.

4 INTRODUçãO a verilog. Acessado em 05/04/2017. Disponível em: <http://www.asic-world.com/verilog/intro1.html#Introduction>. Citado na página 5.

5 VERILOG. Acessado em 07/04/2017. Disponível em: <https://pt.wikipedia.org/wiki/Verilog>. Citado na página 5.

6 ALTERA quartus. Acessado em 07/04/2017. Disponível em: <https://en.wikipedia.org/wiki/Altera_Quartus>. Citado na página 6.

Apêndices

APÊNDICE A – UnidadeC.v

Código completo da unidade de controle.