Paulo Gabriel Ferreira
RA: 102223

Implementação do Compilador C-

(Laboratório de Sistemas Computacionais: Compiladores)

São José dos Campos - Brasil

Junho de 2019

Paulo Gabriel Ferreira
RA: 102223

Implementação do Compilador C-

(Laboratório de Sistemas Computacionais: Compiladores)

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

Junho de 2019

Resumo

No relatório em questão constam as ideias trabalhadas durante os Laboratórios de Sistemas Computacionais.

Nesse ponto de checagem, estão, além da fundamentação teorica, todo o desenvolvimento e trabalho em cima do Compilador, mostrando desde a base para o mesmo, até a integração com o Processador.

Palavras-chaves: Compilador, Processador.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

Nos dias de hoje a tecnologia avança a largos passos e caminhando lado a lado com ela temos os computadores que vem sendo compostos por arquiteturas cada vez mais complexas sendo que mesmo que essas máquinas façam parte da vida de grande maioria da população, poucos sabem os detalhes quando se trata de entender o fundamento para o funcionamento do computador.

E indo por esse caminho, uma das ferramentas que muitos usam mas poucos vão a fundo em seu funcionamento são os compiladores. Um compilador é um programa que traduz uma linguagem de um certo nível para um nível mais baixo, fazendo com que o código escrito por um programador, por exemplo, seja traduzido para um computador.

Neste cenário o objetivo desse relatório é apresentar o desenvolvimento de um compilador, passando por todas as suas fases de analise e síntese, englobando também o processador que irá ler as instruções traduzidas.

2 Processador

2.1 Arquitetura

De forma bem simples, arquitetura é um conjunto de instruções e operações lógicas (registradores e memórias), ou seja, é o que o programador precisa dominar para o desenvolvimento de seu processador.

2.2 Risc x Cisc

Quando o assunto é sistemas computacionais, existem dois tipos de arquitetura de conjunto de instruções que são trabalhados, RISC (Reduced Instruction Set Computers) e CISC (Complex Instruction Set Computers).

A arquitetura do tipo CISC possui um conjunto de instruções grandes de formatos complexos e tamanhos variados, sendo possível executar múltiplas operações quando uma única instrução é dada.

Enquanto a arquitetura CISC é fundamentada em instruções mais complexas, a RISC, busca simplificar as instruções de forma que sejam executadas mais rapidamente.(1)

Tabela 1 – RISC vs CISC

RISC	**CISC**
Arquitetura baseada em Registrador-Registrador	**Arquitetura baseada em Registrador-Memória**
Pouca variedade de tipos de dados	**Tipo de dados variados**
Acessa os dados via registradores	**Acessa os dados via memória**
Conjunto de instruções reduzido	**Grande conjunto de instruções**
Tamanho fixo das instruções	**Tamanho das instruções variam**

Fonte: O Autor(2)

Vale pontuar que desde a década de 80 a maioria dos conjuntos de instruções possuem uma arquitetura que mescla RISC e CISC, o que tem como finalidade a busca de um melhor desempenho durante a execução de suas instruções.

2.3 MIPS

MIPS, ou, Microlocation without Interlocked Pipeline Location é uma arquitetura RISC em que o processador usa apenas registradores para realizar as suas operações aritméticas e lógicas. (3)

Algumas características da arquitetura são:

- Possui execução de cinco estágios: busca, decodificação, execução, acesso à memória e escrita de resultados;

- Apenas as instruções Load e Store acessam a memória.

A arquitetura MIPS está presente, por exemplo, em vários tipos de Sistemas Embarcados, dispositivos com Windows CE e videogames como o Nintendo 64 e o Playstation 1.

2.3.1 Formatos de Instrução

A arquitetura MIPS é composta por 3 diferentes tipos de formato de instruções, sendo eles: R, I, J.

- Formato do tipo R é a função que contém todas as instruções lógicas e aritméticas.

Tabela 2 – Formato da Instruções R

6 bits	**5 bits**	**5 bits**	**5 bits**	**5 bits**	**6 bits**
Opcode	**rs**	**rt**	**rd**	**shamt**	**funct**

Fonte: O Autor

- O formato de instrução J inclui instruções de salto incondicional.

Tabela 3 – Formato da Instruções J

6 bits	**26 bits**
Opcode	**address**

Fonte: O Autor

- Já o formato de instrução do tipo I, que inclui instruções imediatas e de transferência de dados, semelhante ao do tipo R, utiliza em seus primeiros bits o opcode, sendo

também reservados um número grande de bits para seus registradores, e por último, possui 16 bits reservados para o offset.

Tabela 4 – Formato da Instruções I

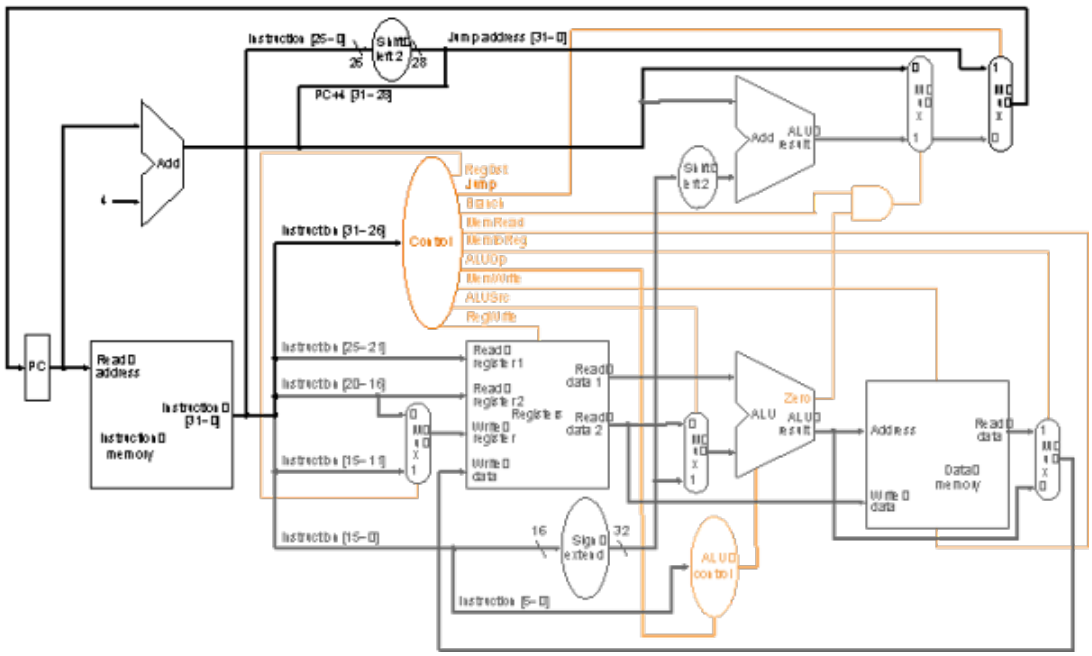
6 bits	5 bits	5 bits	16 bits
Opcode	**rs**	**rt**	**offset**

Fonte: O Autor

2.3.2 Caminho de Dados

O caminho de dados é a combinação da lógica que garante o funcionamento do processador. Nele, são incluídos todas as instruções apresentadas no relatório, além de possuir em sua estrutura componentes como multiplexadores e extensores de sinal, que tem a finalidade de alterar o tamanho da palavra para um adequado.

Figura 1 – Caminho de Dados



Fonte: Arquitectura de Computadores (4)

2.3.3 Memória

A arquitetura do tipo MIPS possui 32 bits em suas instruções, ou seja, as memórias são divididas e operam com endereços do mesmo tamanho.

Nessa arquitetura, a memória de divide em duas partes:

- Memória de Instrução que armazena todas as instruções que serão executadas pelo processador.

- Memória de Dados, componente da Unidade de Processamento responsável por armazenar e retirar informações ao longo da execução de determinada operação.

2.3.4 Program Counter

Componente inicial do processo de leitura e execução de uma instrução, é o responsável por acessar as posições seguintes na memória contando qual será a instrução utilizada no ciclo de clock.

2.3.5 Banco de Registradores

Um banco de registradores da arquitetura MIPS consiste em um conjunto de n registradores de 32 bits.

Sua principal característica é o fato de ser constituído por 32 registradores de propósito geral. Esse tipo de registrador pode ser do tipo fonte ou alvo de operações lógicas, aritméticas e de acesso a memória.

2.3.6 Unidade Lógica Aritmética (ULA)

A Unidade lógica aritmética, ou ULA, é o componente que realiza todas as operações básicas lógicas e aritméticas do computador. É nela que se tem a leitura de dados e a identificação de quais operandos e operações serão utilizados.

2.4 Hierarquia de Memória

Pensando em otimizar a manipulação das informações, é necessário estabelecer uma hierarquia de memória.

Usando a o tempo como parâmetro de comparação, por exemplo, podemos organizar a hierarquia de memória da seguinte forma:

Registrador, Cache, Memória Principal (RAM) e Memória Secundária.

Essa hierarquia é comumente vista na forma de uma pirâmide que segue a regra de quanto mais perto do topo mais rápida e cara são as memórias.

2.5 Unidade de Processamento

A unidade de processamento, ou CPU, pode ser comparada com o “cérebro” do computador, uma vez que ela é a composição das funções que irão executar toda e qualquer instrução da máquina.

2.6 Quartus

O software utilizado para o desenvolvimento do projeto é o Quartus, esse que é desenvolvido pela Altera e é um software de design de dispositivos lógicos programáveis

2.7 Verilog

Verilog é uma linguagem de descrição de hardware de fácil aprendizagem, que se assemelha a linguagens como C++.

A linguagem permite uma visualização muito mais clara da arquitetura de componentes de hardware, como por exemplo o processador descritos neste projeto.(5)

2.8 FPGA

A placa de FPGA é um dispositivo semicondutor que pode ser programado, não deixando restrito a funções de hardware pré definidas.

2.9 Desenvolvimento

2.9.1 Conjunto de Instruções

Um dos primeiros passos foi a decisão do conjunto de instruções a serem trabalhados. Dentro dos formatos R, I, J, foram escolhidas 25 instruções visando resolver os problemas que seriam apresentados, como Fibonacci e Fatorial. (6)

2.9.2 Formato de Instruções

Como foi apresentado na arquitetura MIPS, o projeto utiliza os três formatos de instruções R, I, J.

2.9.3 Modo de Endereçamento

Dentro da Arquitetura MIPS existem diversos modos de endereçamento. Dentro deles, foram selecionados 5 para a realização do projeto:

Endereçamento imediato;

Endereçamento por registrador;

Endereçamento por base ou deslocamento;

Endereçamento relativo ao PC;

Tabela 5 – Conjunto de Instruções

NOME	SIGLA	TIPO	FORMATO
Adição	**add**	**Aritmética**	**R**
Subtração	**sub**	**Aritmética**	**R**
Multiplicação	**mult**	**Aritmética**	**R**
Adição Imediata	**addi**	**Aritmética**	**R**
Subtração Imediata	**subi**	**Aritmética**	**R**
AND	**and**	**Lógicas**	**R**
OR	**or**	**Lógicas**	**R**
NOT	**not**	**Lógicas**	**R**
XOR	**xor**	**Lógicas**	**R**
Set less than	**stl**	**Lógicas**	**R**
Shift left	**shfl**	**Deslocamento**	**R**
Shift right	**shfr**	**Deslocamento**	**R**
NOP	**nop**	**Outros**	**R**
HLT	**hlt**	**Outros**	**R**
Load imediate	**li**	**Transferência**	**I**
load word	**lw**	**Transferência**	**I**
store word	**sw**	**Transferência**	**I**
branch and equal	**beq**	**Desvio condicional**	**I**
branch and not equal	**bne**	**Desvio condicional**	**I**
branch and equal zero	**beqz**	**Desvio condicional**	**I**
jump	**jump**	**Salto Incondicional**	**J**
jump to register	**jumpr**	**Salto Incondicional**	**J**
IN	**out**	**Outros**	**OUTROS**
OUT	**in**	**Outros**	**OUTROS**

Fonte: O Autor

Endereçamento pseudo direto.

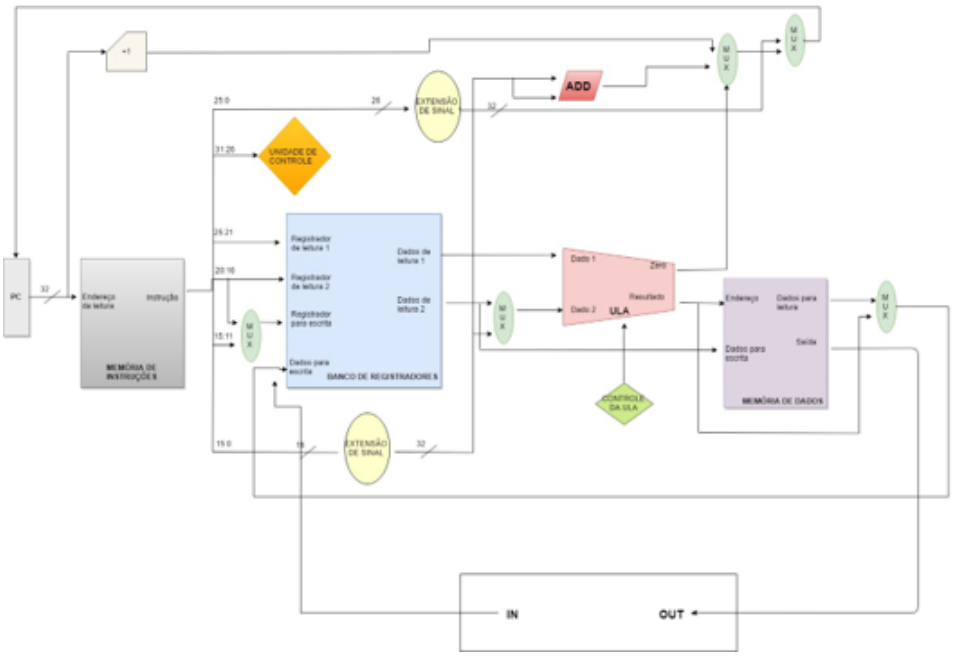
2.9.4 Arquitetura Base do Processador

O processador possui uma arquitetura e um caminho de dados semelhante ao MIPS, porém não tão complexo, sendo que as unidades compõem a arquitetura base em questão são o PC, o Banco de Registradores, Memória de Dados e Memória de Instruções.

A arquitetura da imagem 2 foi a apresentada no começo do semestre como sendo a arquitetura do projeto em questão. Porém, após um maior desnvolvimento do trabalho, e uma ajuda dos colegas, o ADD, e os dois Mux que recebiam os dados do PC, ADD e

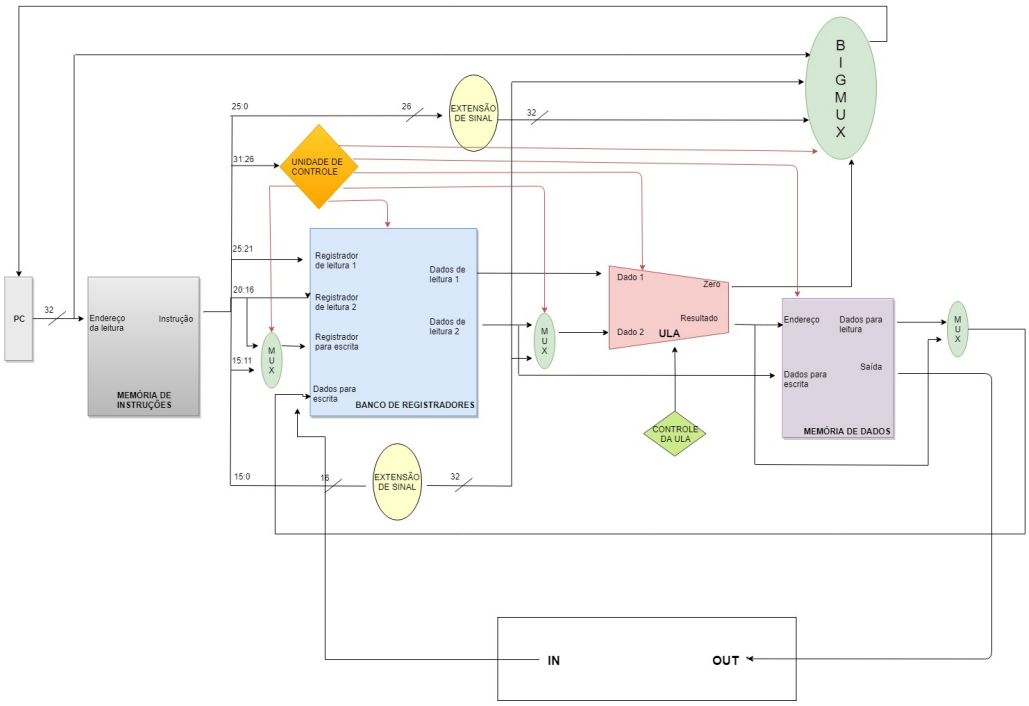
ULA, foram convertidos em uma unidade chamada BigMux, como mostra a figura 3

Figura 2 – Arquitetura Base



Fonte: O Autor

Figura 3 – Arquitetura Final



Fonte: O Autor

2.9.5 Caminho de dados

2.9.5.1 Instruções tipo R

O caminho de dados dessa instrução se inicia no PC seguindo pela memória de instruções e banco de registradores, onde entram os registradores rs, rt e rd. O resultado que sai desse banco é encaminhado para a ULA, onde são realizadas as operações (lógicas, aritméticas...) e a saída é encaminhada para o banco de registradores.

2.9.6 Unidade de Processamento

A unidade de processamento do projeto é composta por toda a arquitetura base apresentada, com exceção da unidade de controle, sendo assim, as construções dos componentes do projeto (implementadas em Verilog) são:

2.9.6.1 ULA

A unidade logica aritmetica (ULA) é o componentes responsável por realizar as operações lógicas e operações aritméticas como soma, subtração e AND.

A instrução é escolhida por meio do opcode recebido por esse componente, assim se a intrução add for escolhida, por exemplo, os dois valores que entram neste componente seram somados e seu resultado será armazenado na variavel "resultado".

2.9.6.2 PC

O Program Counter é composto por um registrador que armazena o endereço da proxima instrução a ser realizada. Esse endereço é atualizado a cada clock de subida, condição essa definida no código por meio da variável "posedge".

2.9.6.3 Memória de Instruções

A memória de instruções é composta por um vetor que possui 32 posições com a função de armazenamento. O numero de posições é atualizado de acordo com o número de etapas a serem realizadas.

No caso, as instruções no código são instruções de um código que permite escolher entre as operações de Fibonacci, Soma, Fatorial e Alocação de Memória

2.9.6.4 Memória de Dados

Outro componente da unidade de processamento é a memória de dados cujo a função é o armazenamento dos dados. As únicas instruções que utilizadam esse componente

são as instruções load e store.

Esse bloco funciona por meio de uma variável "EscreveMem"que indica que existe a necessiadade de escrita, logo, o valor é colocado dentro da memória de dados em um endereço que é previamente recebido pelo componente.

2.9.6.5 Banco de Registradores

O banco de registradores implementado trabalha com 32 registradores de propósito geral. Ele recebe como entrada a variável "RegWrite"que é utilizada para identifcar quando será armazenado um valor em um dado registrador. Para tanto ele necessita de outras entradas como o valor e o endereço do registrador onde será armazenado neste banco.

2.9.6.6 Extensores

Para o projeto foram implementados dois tipos de extensores sendo o primeiro deles o extensor de 16 bits para 32 bits e o segundo de 26 bits para 32 bits.Sua função é de estender dados que estejam em tamanhos diferentes de 32bits até que cheguem a esse valor uma vez que a arquitetura projetada necessita deste tamanho para fazer suas operações sobre os dados.

2.9.6.7 MUX

O multiplexador (mux) é o componente resposável por escolher entre duas entradas, para tanto utiliza de uma variável auxiliar "IMcontrol"como forma de mediar esta escolha.

2.9.7 Unidade de Controle

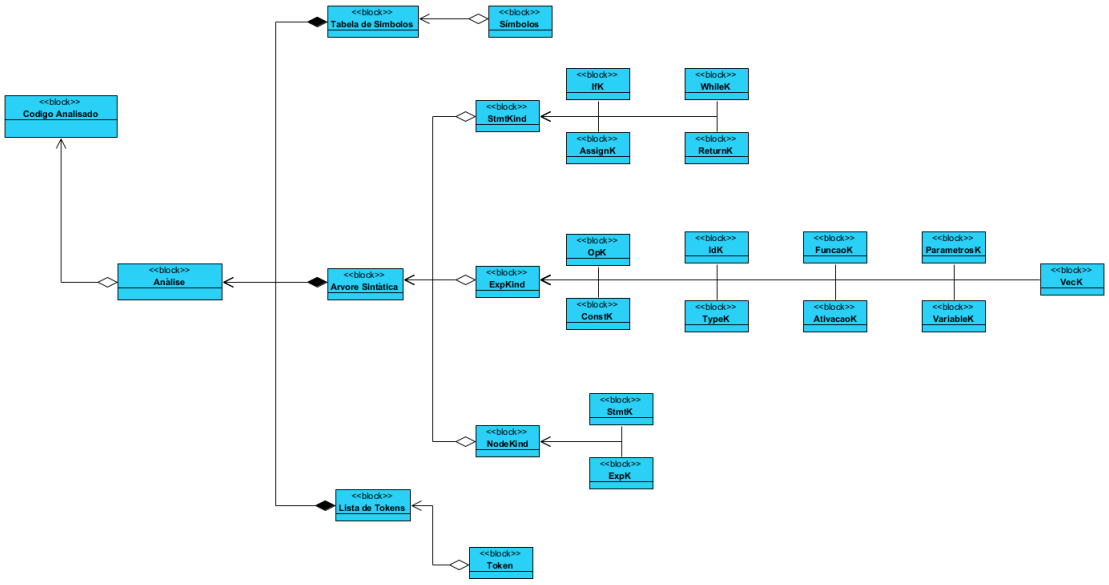
A Unidade de Controle é onde se coordenada as atividades do processador, ela foi implementada no modo hardwire, onde cada sinal fica dependente da entrada. Para cada instrução são tratados os sinais de controle que irão para os demais módulos. É atraves do opcode que tem a informação de qual instrução será executada.

3 Compilador

3.1 Modelagem

Uma maneira de entender o processo do compilador é observando os diagramas do projeto. Abaixo segue o diagrama de blocos que explica a fase de análise do compilador:

Figura 4 – Diagrama da fase de Análise

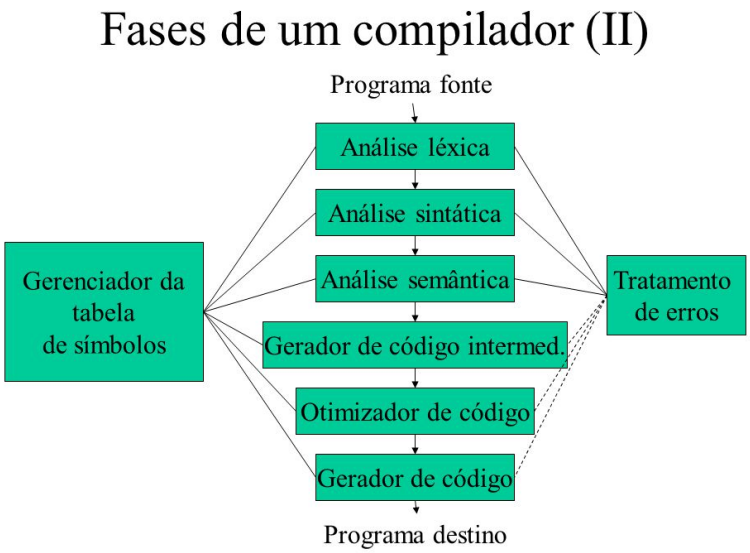


Fonte: O Autor

3.2 Compilador: Fase de Análise

Um compilador é um programa de computador que lê e transcreve um determinado código de uma determinada linguagem para outra. O compilador consiste em 3 etapas: Análise Léxica, Analise Sintática e Análise Semântica.

Figura 5 – Caminho de Dados



Fonte: Processos de Compilação (7)

3.2.1 Análise Léxica

A fase de Análise Léxica consiste em escanear e separar os "Tokens"do código fonte, ou seja:

- Palavras reservadas, que na liguagem C- são: IF - ELSE - VOID - INT - RETURN

- Símbolos Especiais: + - * / = != < > <= >= == , ; () [] /* */

- Identificadores e números.

Caso haja um erro de escrita de algum desses token, o compilador irá printar um erro de Análise Léxica.

O código abaixo mostra os tokens usados no projeto.

```
1
2  digit       [0-9]
3  number      {digit}+
4  letter      [a-zA-Z]
5  identifier  {letter}+
6  newline     \n
7  whitespace  [ \t\r]+
8
9  %%
10
11 "if"            {return IF;}
12 "else"          {return ELSE;}
13 "int"           {return INT;}
14 "return"        {return RETURN;}
15 "void"          {return VOID;}
16 "while"         {return WHILE;}
17 "+"             {return PLUS;}
```

```
18 "-"              {return MINUS;}
19 "*"              {return TIMES;}
20 "/"              {return OVER;}
21 "<"              {return LT;}
22 "<="             {return LET;}
23 ">"              {return GT;}
24 ">="             {return GET;}
25 "=="             {return EQ;}
26 "!="             {return NEQ;}
27 "="              {return ASSIGN;}
28 ";"              {return SEMI;}
29 ","              {return VIRG;}
30 "("              {return LPAREN;}
31 ")"              {return RPAREN;}
32 "["              {return CONOPEN;}
33 "]"              {return CONCLOSE;}
34 "{"              {return CHAVEOPEN;}
35 "}"              {return CHAVECLOSE;}
36 {number}         {return NUM;}
37 {identifier}     {return ID;}
38 {newline}        {lineno++;}
```

3.2.2 Análise Sintática

A Análise Sintática é a responsável por encontrar os erros advindos da junção dos "Tokens", gerando assim um erro sintático. Caso não haja nenhum erro desse tipo no código, essa etapa é onde se constrói a Árvore de Análise Sintática.

Figura 6 – Gramática para a linguagem C-

programa → declaração-lista
declaração-lista → declaração-lista declaração | declaração
declaração → var-declaração | fun-declaração
var-declaração → tipo-especificador ID ; | tipo-especificador ID [NUM] ;
tipo-especificador → int | void
fun-declaração → tipo-especificador ID (params) composto-decl
params → param-lista | void
param-lista → param-lista, param | param
param → tipo-especificador ID | tipo-especificador ID [ ]
composto-decl → { local-declarações statement-lista } | {local-declarações} | {statement-lista} | { }
local-declarações → local-declarações var-declaração | var-declaração
statement-lista → statement-lista statement | statement
statement → expressão-decl | composto-decl | seleção-decl | iteração-decl | retorno-decl
expressão-decl → expressão ; | ;
seleção-decl → if (expressão) statement | if (expressão) statement else statement
iteração-decl → while (expressão) statement
retorno-decl → return ; | return expressão;
expressão → var = expressão | simples-expressão
var → ID | ID [expressão]
simples-expressão → soma-expressão relacional soma-expressão | soma-expressão
relacional → <= | < | > | >= | == | !=
soma-expressão → soma-expressão soma termo | termo
soma → + | -
termo → termo mult fator | fator
mult → * | /
fator → (expressão) | var | ativação | NUM
ativação → ID (arg-lista) | ID ( )
arg-lista → arg-lista , expressão | expressão

Fonte: (8)

3.2.3 Análise Semântica

A última fase, Análise Semântica, é onde procura-se os error referentes a linguagem compilada. Em C- por exemplo, caso uma variável declarada como int, recebe um valor char, o erro apontado será um erro semântico. Nesta etapa também é gerada a tabela de símbolos contendo todos os símbolos do código.

Segue abaixo um trecho do código para a análise semântica.

```
1  static void insertNode( TreeNode * t)
2  { switch (t->nodekind)
3    { case StmtK:
4        switch (t->kind.stmt)
5        { case AssignK:
6            if (assigns == NULL)
7            { assigns = (LineList) malloc(sizeof(struct LineListRec));
8              assigns->lineno = t->lineno;
9              assigns->next = NULL;
10           }
11           else
12           { LineList a = assigns;
13             while (a->next != NULL) a = a->next;
14               a->next = (LineList) malloc(sizeof(struct LineListRec));
```

```
15                a->next->lineno = t->lineno;
16                a->next->next = NULL;
17              }
18              break;
19            default:
20              break;
21          }
22          break;
23        case ExpK:
24          switch (t->kind.exp)
25          { case IdK:
26              if ((st_lookup(t->nameID, t->scope) == -1) && (st_var_decl(t->nameID, t->scope)
                    !=-1))
27                st_insert(t->nameID, t->scope, t->typeID, t->typedata, t->declared, t->lineno
                      , location++, t->vet, t->params);
28              else
29                st_insert(t->nameID, t->scope, t->typeID, t->typedata, t->declared, t->lineno
                      , 0, t->vet, t->params);
30              break;
31            case VariableK:
32              if (st_lookup(t->nameID, t->scope) == -1)
33                st_insert(t->nameID, t->scope, t->typeID, t->typedata, t->declared, t->lineno
                      , location++, t->vet, t->params);
34              else
35                st_insert(t->nameID, t->scope, t->typeID, t->typedata, t->declared, t->lineno
                      , 0, t->vet, t->params);
36              break;
37            case AtivacaoK:
38              if (st_lookup(t->nameID, t->scope) == -1)
39                st_insert(t->nameID, t->scope, t->typeID, t->typedata, t->declared, t->lineno
                      , -1, t->vet, t->params);
40              else
41                st_insert(t->nameID, t->scope, t->typeID, t->typedata, t->declared, t->lineno
                      , 0, t->vet, t->params);
42              break;
43            case Funcaok:
44              if (st_lookup(t->nameID, t->scope) == -1)
45                st_insert(t->nameID, t->scope, t->typeID, t->typedata, t->declared, t->lineno
                      , -1, t->vet, t->params);
46              else
47                st_insert(t->nameID, t->scope, t->typeID, t->typedata, t->declared, t->lineno
                      , 0, t->vet, t->params);
48              break;
49            case ParametrosK:
50              if (st_lookup(t->nameID, t->scope) == -1)
51                st_insert(t->nameID, t->scope, t->typeID, t->typedata, t->declared, t->lineno
                      , location++, t->vet, t->params);
52              else
53                st_insert(t->nameID, t->scope, t->typeID, t->typedata, t->declared, t->lineno
                      , 0, t->vet, t->params);
54              break;
55            default:
56              break;
57        }
58        break;
59      default:
60        break;
61      }
62  }
```

3.3 Compilador: Fase de Síntese

3.3.1 Geração do código intermediário

O código intermediário é o código que recebe a saída da fase de analise do compilador e através das quadruplas, gera um código que facilita a formação do código assembly.

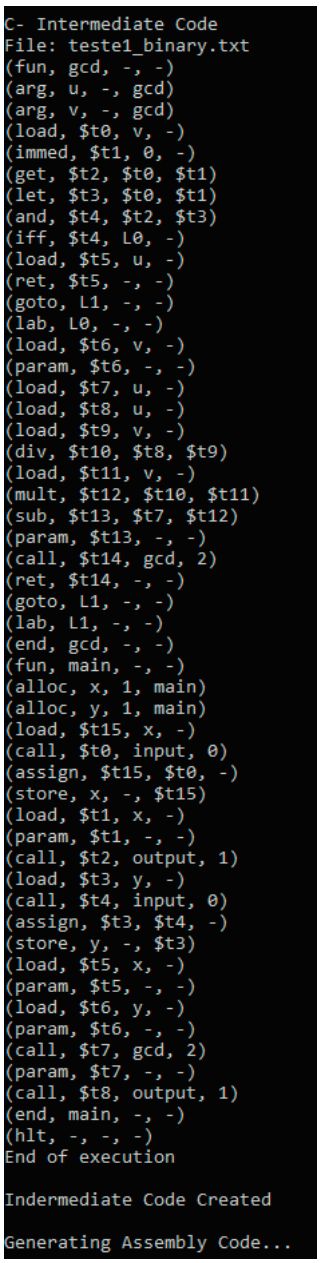
Os campos da quadrupla possuem funções distintas. O primeiro é separado para a operação a ser feita, enquanto os demais são preenchidos com os endereços para a realização da operação do primeiro campo. O último campo pode ser strings, inteiros ou vazio.

O código abaixo é o codigo que forma a quadrupla para o código intermediário.

```
1  void quad_insert (OpKind op, Address addr1, Address addr2, Address addr3) {
2
3    Quad quad;
4    quad.op = op;
5    quad.addr1 = addr1;
6    quad.addr2 = addr2;
7    quad.addr3 = addr3;
8    QuadList new = (QuadList) malloc(sizeof(struct QuadListRec));
9    new->location = location;
10   new->quad = quad;
11   new->next = NULL;
12   if (head == NULL) {
13     head = new;
14   }
15   else {
16     QuadList q = head;
17     while (q->next != NULL) q = q->next;
18     q->next = new;
19   }
20   location ++;
21 }
22
23 int quad_update(int loc, Address addr1, Address addr2, Address addr3) {
24   QuadList q = head;
25   while (q != NULL) {
26     if (q->location == loc) break;
27     q = q->next;
28   }
29   if (q == NULL)
30     return 0;
31   else {
32     q->quad.addr1 = addr1;
33     q->quad.addr2 = addr2;
34     q->quad.addr3 = addr3;
35     return 1;
36   }
37 }
```

A imagem a seguir mostra um código intermediário gerado pelo compilador. O código em questão é referente à um algoritmo de calculo do máximo divisor comum (MDC ou GCD).

Figura 7 – Código Intermediário do GCD Gerado no Terminal



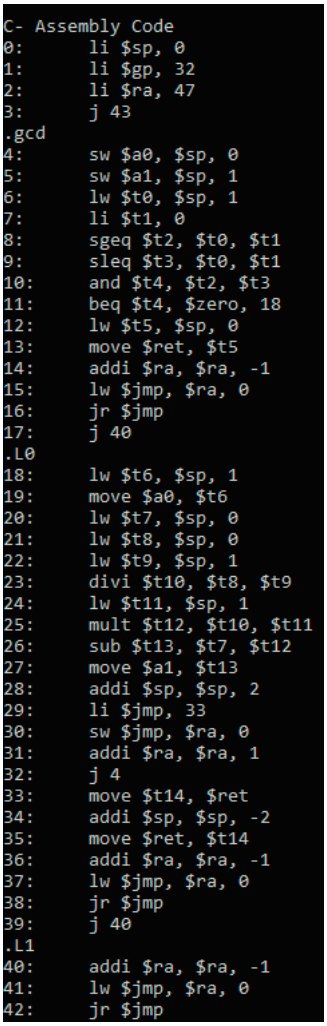
Fonte: O Autor

3.3.2 Geração do código Assembly

O passo seguinte ao intermediário é a geração do código em assembly, onde acontece a tradução das quadruplas. Este processo utilizou de um grande numero de registradores, trabalhando bastante com registradores temporários como pode ser visto nas figuras 8 e 9.

O código para geração do assembly também conta com três registradores que possuem a função de controlar a pilha. São eles o stack pointer (recursão), o global pointer (vetores globais) e o return address que controla o endereço para o retorno da função.

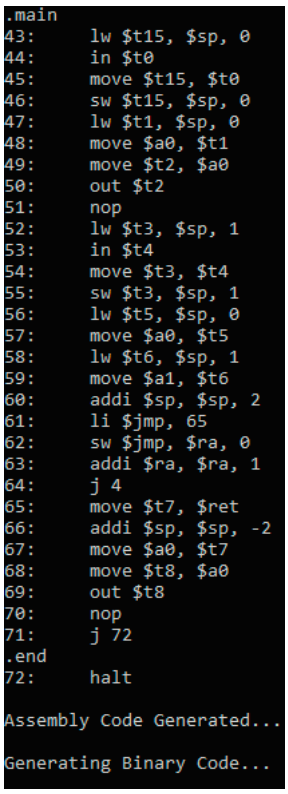
Figura 8 – Código Assembly do GCD Gerado no Terminal



```
C- Assembly Code
0:      li $sp, 0
1:      li $gp, 32
2:      li $ra, 47
3:      j 43
.gcd
4:      sw $a0, $sp, 0
5:      sw $a1, $sp, 1
6:      lw $t0, $sp, 1
7:      li $t1, 0
8:      sgeq $t2, $t0, $t1
9:      sleq $t3, $t0, $t1
10:     and $t4, $t2, $t3
11:     beq $t4, $zero, 18
12:     lw $t5, $sp, 0
13:     move $ret, $t5
14:     addi $ra, $ra, -1
15:     lw $jmp, $ra, 0
16:     jr $jmp
17:     j 40
.L0
18:     lw $t6, $sp, 1
19:     move $a0, $t6
20:     lw $t7, $sp, 0
21:     lw $t8, $sp, 0
22:     lw $t9, $sp, 1
23:     divi $t10, $t8, $t9
24:     lw $t11, $sp, 1
25:     mult $t12, $t10, $t11
26:     sub $t13, $t7, $t12
27:     move $a1, $t13
28:     addi $sp, $sp, 2
29:     li $jmp, 33
30:     sw $jmp, $ra, 0
31:     addi $ra, $ra, 1
32:     j 4
33:     move $t14, $ret
34:     addi $sp, $sp, -2
35:     move $ret, $t14
36:     addi $ra, $ra, -1
37:     lw $jmp, $ra, 0
38:     jr $jmp
39:     j 40
.L1
40:     addi $ra, $ra, -1
41:     lw $jmp, $ra, 0
42:     jr $jmp
```

Fonte: O Autor

Figura 9 – Código Assembly do GCD Gerado no Terminal



Fonte: O Autor

3.3.3 Geração do código executável

Por fim, tem-se a geração do executável, que é onde o assembly é traduzido para binário, de uma forma que possa ser interpretado pelo processador. No código é feita a conversão para o binário transformando os Opcodes de acordo com seu equivalente no processador, segue o código abaixo.

Trata-se também nesta etapa a questão do formato dos registradores, uma vez que no assembly, o registrador de destino vem primeiro e no mips o é deixando por último. Ou seja, o assembly gera um formato rd rs rt, sendo rd o registrador de destino, e o mips lê rs rt rd.

O código abaixo mostra os opcodes para a geração do binário.

```
1  const char * opcodes[] =  { "nop", "halt", "add", "sub",   //1
2                              "addi", "subi", "mult", "xor", //2
3                              "and", "not", "or", "slt",    //3
4                              "shftl", "shftr", "li", "sw", //4
5                              "beq", "bne", "j", "jr",      //5
6                              "in", "out", "sgt", "move",   //6
7                              "divi", "mod", "lw", "sgeq", //7
8                              "sleq", "beqz"};             //8
9
10 const char * opcodeBins[] =  {  "000000", "000001", "000010", "000011", //1
11                                 "000100", "000101", "000110", "000111", //2
```

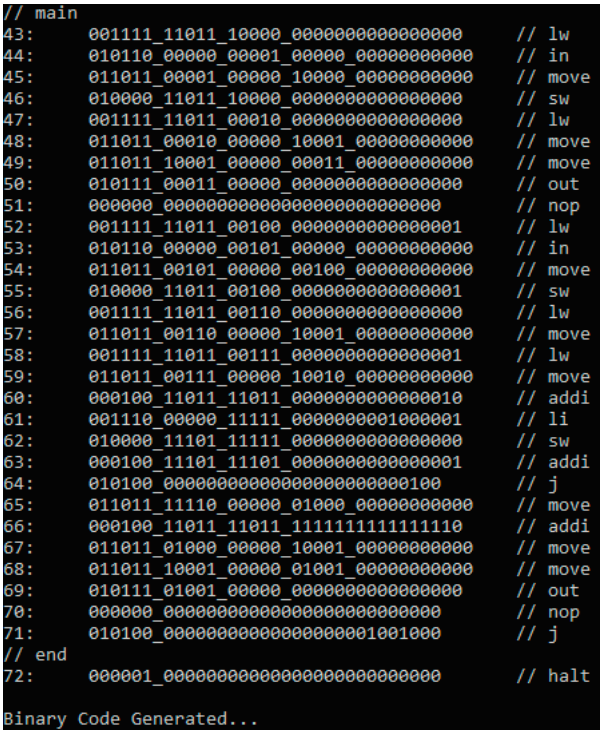
```
12                          "001000","001001", "001010", "001011", //3
13                          "001100", "001101", "001110", "010000", //4
14                          "010001", "010010", "010100", "010101", //5
15                          "010110", "010111", "011000", "011011", //6
16                          "011100", "011101", "001111", "011001",//7
17                          "011010", "010011"}; //8
```

Figura 10 – Código Binário do GCD Gerado no Terminal

```
C- Binary Code
0:      001110_00000_11011_0000000000000000     // li
1:      001110_00000_11100_0000000000100000     // li
2:      001110_00000_11101_0000000000101111     // li
3:      010100_00000000000000000000101011       // j
// gcd
4:      010000_11011_10001_0000000000000000     // sw
5:      010000_11011_10010_0000000000000001     // sw
6:      001111_11011_00001_0000000000000001     // lw
7:      001110_00000_00010_0000000000000000     // li
8:      011001_00001_00010_00011_00000000000    // sgeq
9:      011010_00001_00010_00100_00000000000    // sleq
10:     001000_00011_00100_00101_00000000000    // and
11:     010001_00101_00000_0000000000010010     // beq
12:     001111_11011_00110_0000000000000000     // lw
13:     011011_00110_00000_11110_00000000000    // move
14:     000100_11101_11101_1111111111111111     // addi
15:     001111_11101_11111_0000000000000000     // lw
16:     010101_11111_00000_0000000000000000     // jr
17:     010100_00000000000000000000101000       // j
// L0
18:     001111_11011_00111_0000000000000001     // lw
19:     011011_00111_00000_10001_00000000000    // move
20:     001111_11011_01000_0000000000000000     // lw
21:     001111_11011_01001_0000000000000000     // lw
22:     001111_11011_01010_0000000000000001     // lw
23:     011100_01001_01010_01011_00000000000    // divi
24:     001111_11011_01100_0000000000000001     // lw
25:     000110_01011_01100_01101_00000000000    // mult
26:     000011_01000_01101_01110_00000000000    // sub
27:     011011_01110_00000_10010_00000000000    // move
28:     000100_11011_11011_0000000000000010     // addi
29:     001110_00000_11111_0000000000100001     // li
30:     010000_11101_11111_0000000000000000     // sw
31:     000100_11101_11101_0000000000000001     // addi
32:     010100_00000000000000000000000100       // j
33:     011011_11110_00000_01111_00000000000    // move
34:     000100_11011_11011_1111111111111110     // addi
35:     011011_01111_00000_11110_00000000000    // move
36:     000100_11101_11101_1111111111111111     // addi
37:     001111_11101_11111_0000000000000000     // lw
38:     010101_11111_00000_0000000000000000     // jr
39:     010100_00000000000000000000101000       // j
// L1
40:     000100_11101_11101_1111111111111111     // addi
41:     001111_11101_11111_0000000000000000     // lw
42:     010101_11111_00000_0000000000000000     // jr
```

Fonte: O Autor

Figura 11 – Código Binário do GCD Gerado no Terminal



Fonte: O Autor

4 Exemplos

Aqui seguem os programas que foram apresentados para a validação do compilador.

4.1 Soma

Um algoritmo simples de soma que serviu de teste para o tratamento do segundo input.

```
1  int input(void)
2  {
3  }
4
5  void output(int x)
6  {
7  }
8
9
10 int main(void) {
11 int a;
12 int b;
13 int c;
14
15 a = input();
16 output (a);
17 b = input();
18
19
20 c = a + b;
21
22 output (c);
23
24 }
```

4.1.1 Intermediário

```
1  C- Intermediate Code
2  File: teste2_binary.txt
3  (fun, main, -, -)
4  (alloc, a, 1, main)
5  (alloc, b, 1, main)
6  (alloc, c, 1, main)
7  (load, $t0, a, -)
8  (call, $t1, input, 0)
9  (assign, $t0, $t1, -)
10 (store, a, -, $t0)
11 (load, $t2, a, -)
12 (param, $t2, -, -)
13 (call, $t3, output, 1)
14 (load, $t4, b, -)
15 (call, $t5, input, 0)
16 (assign, $t4, $t5, -)
```

```
(store, b, -, $t4)
(load, $t6, c, -)
(load, $t7, a, -)
(load, $t8, b, -)
(add, $t9, $t7, $t8)
(assign, $t6, $t9, -)
(store, c, -, $t6)
(load, $t10, c, -)
(param, $t10, -, -)
(call, $t11, output, 1)
(end, main, -, -)
(hlt, -, -, -)
End of execution
```

4.1.2 Assembly

```
C- Assembly Code
0:      li $sp, 0
1:      li $gp, 32
2:      li $ra, 47
3:      j 4
.main
4:      lw $t0, $sp, 0
5:      in $t1
6:      move $t0, $t1
7:      sw $t0, $sp, 0
8:      lw $t2, $sp, 0
9:      move $a0, $t2
10:     move $t3, $a0
11:     out $t3
12:     nop
13:     lw $t4, $sp, 1
14:     in $t5
15:     move $t4, $t5
16:     sw $t4, $sp, 1
17:     lw $t6, $sp, 2
18:     lw $t7, $sp, 0
19:     lw $t8, $sp, 1
20:     add $t9, $t7, $t8
21:     move $t6, $t9
22:     sw $t6, $sp, 2
23:     lw $t10, $sp, 2
24:     move $a0, $t10
25:     move $t11, $a0
26:     out $t11
27:     nop
28:     j 29
.end
29:     halt
```

4.1.3 Binário

```

C- Binary Code
     001110_00000_11011_0000000000000000     // li
     001110_00000_11100_0000000000100000     // li
     001110_00000_11101_0000000000101111     // li
     010100_00000000000000000000000100       // j
// main
```

```
8       001111_11011_00001_0000000000000000     // lw
9       010110_00000_00010_00000_00000000000    // in
10      011011_00010_00000_00001_00000000000    // move
11      010000_11011_00001_0000000000000000     // sw
12      001111_11011_00011_0000000000000000     // lw
13      011011_00011_00000_10001_00000000000    // move
14      011011_10001_00000_00100_00000000000    // move
15      010111_00100_00000_0000000000000000     // out
16      000000_00000000000000000000000000       // nop
17      001111_11011_00101_0000000000000001     // lw
18      010110_00000_00110_00000_00000000000    // in
19      011011_00110_00000_00101_00000000000    // move
20      010000_11011_00101_0000000000000001     // sw
21      001111_11011_00111_0000000000000010     // lw
22      001111_11011_01000_0000000000000000     // lw
23      001111_11011_01001_0000000000000001     // lw
24      000010_01000_01001_01010_00000000000    // add
25      011011_01010_00000_00111_00000000000    // move
26      010000_11011_00111_0000000000000010     // sw
27      001111_11011_01011_0000000000000010     // lw
28      011011_01011_00000_10001_00000000000    // move
29      011011_10001_00000_01100_00000000000    // move
30      010111_01100_00000_0000000000000000     // out
31      000000_00000000000000000000000000       // nop
32      010100_00000000000000000000011101       // j
33  // end
34      000001_00000000000000000000000000       // halt
35
36  Binary Code Generated...
```

4.2 Sort

Um algoritmo para realizar uma ordenação de vetor.

```
1   void sort( int a[], int low, int high)
2   {       int i; int k;
3           i = low;
4           while (i < high-1){
5                   int t;
6                   k = minloc(a,i,high);
7                   t = a[k];
8                   a[k] = a[i];
9                   a[i] = t;
10                  i = i + 1;
11          }
12  }
13
14  void main(void)
15  {
16          int a;
17          int i;
18          i = 0;
19
20
21          vet[0] = input();
22          output(0);
23          vet[1] = input();
24          output(0);
25          vet[2] = input();
```

```
26          output(0);
27          vet[3] = input();
28          output(0);
29          vet[4] = input();
30
31          sort(vet,0,5);
32
33
34          output(vet[0]);
35          output(vet[1]);
36          output(vet[2]);
37          output(vet[3]);
38          output(vet[4]);
39
40
41  }
```

4.2.1 Intermediário

```
1   C- Intermediate Code
2   File: t2_binary.txt
3   (alloc, vet, 5, Global)
4   (fun, minloc, -, -)
5   (arg, a, -, minloc)
6   (arg, low, -, minloc)
7   (arg, high, -, minloc)
8   (alloc, i, 1, minloc)
9   (alloc, x, 1, minloc)
10  (alloc, k, 1, minloc)
11  (load, $t0, k, -)
12  (load, $t1, low, -)
13  (assign, $t0, $t1, -)
14  (store, k, -, $t0)
15  (load, $t2, x, -)
16  (load, $t4, low, -)
17  (vec, $t3, a, $t4)
18  (assign, $t2, $t3, -)
19  (store, x, -, $t2)
20  (load, $t5, i, -)
21  (load, $t6, low, -)
22  (immed, $t7, 1, -)
23  (add, $t8, $t6, $t7)
24  (assign, $t5, $t8, -)
25  (store, i, -, $t5)
26  (lab, L0, -, -)
27  (load, $t9, i, -)
28  (load, $t10, high, -)
29  (lt, $t11, $t9, $t10)
30  (iff, $t11, L3, -)
31  (load, $t13, i, -)
32  (vec, $t12, a, $t13)
33  (load, $t14, x, -)
34  (lt, $t15, $t12, $t14)
35  (iff, $t15, L1, -)
36  (load, $t0, x, -)
37  (load, $t2, i, -)
38  (vec, $t1, a, $t2)
39  (assign, $t0, $t1, -)
```

```
40  (store, x, -, $t0)
41  (load, $t3, k, -)
42  (load, $t4, i, -)
43  (assign, $t3, $t4, -)
44  (store, k, -, $t3)
45  (goto, L2, -, -)
46  (lab, L1, -, -)
47  (lab, L2, -, -)
48  (load, $t5, i, -)
49  (load, $t6, i, -)
50  (immed, $t7, 1, -)
51  (add, $t8, $t6, $t7)
52  (assign, $t5, $t8, -)
53  (store, i, -, $t5)
54  (goto, L0, -, -)
55  (lab, L3, -, -)
56  (load, $t9, k, -)
57  (ret, $t9, -, -)
58  (end, minloc, -, -)
59  (fun, sort, -, -)
60  (arg, a, -, sort)
61  (arg, low, -, sort)
62  (arg, high, -, sort)
63  (alloc, i, 1, sort)
64  (alloc, k, 1, sort)
65  (load, $t10, i, -)
66  (load, $t11, low, -)
67  (assign, $t10, $t11, -)
68  (store, i, -, $t10)
69  (lab, L4, -, -)
70  (load, $t12, i, -)
71  (load, $t13, high, -)
72  (immed, $t14, 1, -)
73  (sub, $t15, $t13, $t14)
74  (lt, $t0, $t12, $t15)
75  (iff, $t0, L5, -)
76  (alloc, t, 1, sort)
77  (load, $t1, k, -)
78  (load, $t2, a, -)
79  (param, $t2, -, -)
80  (load, $t3, i, -)
81  (param, $t3, -, -)
82  (load, $t4, high, -)
83  (param, $t4, -, -)
84  (call, $t5, minloc, 3)
85  (assign, $t1, $t5, -)
86  (store, k, -, $t1)
87  (load, $t6, t, -)
88  (load, $t8, k, -)
89  (vec, $t7, a, $t8)
90  (assign, $t6, $t7, -)
91  (store, t, -, $t6)
92  (load, $t10, k, -)
93  (vec, $t9, a, $t10)
94  (load, $t12, i, -)
95  (vec, $t11, a, $t12)
96  (assign, $t9, $t11, -)
97  (store, a, $t10, $t9)
98  (load, $t14, i, -)
99  (vec, $t13, a, $t14)
```

```
100 (load, $t15, t, -)
101 (assign, $t13, $t15, -)
102 (store, a, $t14, $t13)
103 (load, $t0, i, -)
104 (load, $t1, i, -)
105 (immed, $t2, 1, -)
106 (add, $t3, $t1, $t2)
107 (assign, $t0, $t3, -)
108 (store, i, -, $t0)
109 (goto, L4, -, -)
110 (lab, L5, -, -)
111 (end, sort, -, -)
112 (fun, main, -, -)
113 (alloc, a, 1, main)
114 (alloc, i, 1, main)
115 (load, $t4, i, -)
116 (immed, $t5, 0, -)
117 (assign, $t4, $t5, -)
118 (store, i, -, $t4)
119 (immed, $t7, 0, -)
120 (vec, $t6, vet, $t7)
121 (call, $t8, input, 0)
122 (assign, $t6, $t8, -)
123 (store, vet, $t7, $t6)
124 (immed, $t9, 0, -)
125 (param, $t9, -, -)
126 (call, $t10, output, 1)
127 (immed, $t12, 1, -)
128 (vec, $t11, vet, $t12)
129 (call, $t13, input, 0)
130 (assign, $t11, $t13, -)
131 (store, vet, $t12, $t11)
132 (immed, $t14, 0, -)
133 (param, $t14, -, -)
134 (call, $t15, output, 1)
135 (immed, $t1, 2, -)
136 (vec, $t0, vet, $t1)
137 (call, $t2, input, 0)
138 (assign, $t0, $t2, -)
139 (store, vet, $t1, $t0)
140 (immed, $t3, 0, -)
141 (param, $t3, -, -)
142 (call, $t4, output, 1)
143 (immed, $t6, 3, -)
144 (vec, $t5, vet, $t6)
145 (call, $t7, input, 0)
146 (assign, $t5, $t7, -)
147 (store, vet, $t6, $t5)
148 (immed, $t8, 0, -)
149 (param, $t8, -, -)
150 (call, $t9, output, 1)
151 (immed, $t11, 4, -)
152 (vec, $t10, vet, $t11)
153 (call, $t12, input, 0)
154 (assign, $t10, $t12, -)
155 (store, vet, $t11, $t10)
156 (load, $t13, vet, -)
157 (param, $t13, -, -)
158 (immed, $t14, 0, -)
159 (param, $t14, -, -)
```

```
(immed, $t15, 5, -)
(param, $t15, -, -)
(call, $t0, sort, 3)
(immed, $t2, 0, -)
(vec, $t1, vet, $t2)
(param, $t1, -, -)
(call, $t3, output, 1)
(immed, $t5, 1, -)
(vec, $t4, vet, $t5)
(param, $t4, -, -)
(call, $t6, output, 1)
(immed, $t8, 2, -)
(vec, $t7, vet, $t8)
(param, $t7, -, -)
(call, $t9, output, 1)
(immed, $t11, 3, -)
(vec, $t10, vet, $t11)
(param, $t10, -, -)
(call, $t12, output, 1)
(immed, $t14, 4, -)
(vec, $t13, vet, $t14)
(param, $t13, -, -)
(call, $t15, output, 1)
(end, main, -, -)
(hlt, -, -, -)
End of execution
```

4.2.2 Assembly

```
C- Assembly Code
0:      li $sp, 0
1:      li $gp, 32
2:      li $ra, 47
3:      j 125
.minloc
4:      sw $a0, $sp, 0
5:      sw $a1, $sp, 1
6:      sw $a2, $sp, 2
7:      lw $t0, $sp, 5
8:      lw $t1, $sp, 1
9:      move $t0, $t1
10:     sw $t0, $sp, 5
11:     lw $t2, $sp, 4
12:     lw $t4, $sp, 1
13:     lw $t3, $sp, 0
14:     add $t4, $t4, $t3
15:     lw $t3, $t4, 0
16:     move $t2, $t3
17:     sw $t2, $sp, 4
18:     lw $t5, $sp, 3
19:     lw $t6, $sp, 1
20:     li $t7, 1
21:     add $t8, $t6, $t7
22:     move $t5, $t8
23:     sw $t5, $sp, 3
.L0
24:     lw $t9, $sp, 3
25:     lw $t10, $sp, 2
26:     slt $t11, $t9, $t10
```

```
27:     beq $t11, $zero, 54
28:     lw $t13, $sp, 3
29:     lw $t12, $sp, 0
30:     add $t13, $t13, $t12
31:     lw $t12, $t13, 0
32:     lw $t14, $sp, 4
33:     slt $t15, $t12, $t14
34:     beq $t15, $zero, 47
35:     lw $t0, $sp, 4
36:     lw $t2, $sp, 3
37:     lw $t1, $sp, 0
38:     add $t2, $t2, $t1
39:     lw $t1, $t2, 0
40:     move $t0, $t1
41:     sw $t0, $sp, 4
42:     lw $t3, $sp, 5
43:     lw $t4, $sp, 3
44:     move $t3, $t4
45:     sw $t3, $sp, 5
46:     j 47
.L1
.L2
47:     lw $t5, $sp, 3
48:     lw $t6, $sp, 3
49:     li $t7, 1
50:     add $t8, $t6, $t7
51:     move $t5, $t8
52:     sw $t5, $sp, 3
53:     j 24
.L3
54:     lw $t9, $sp, 5
55:     move $ret, $t9
56:     addi $ra, $ra, -1
57:     lw $jmp, $ra, 0
58:     jr $jmp
59:     addi $ra, $ra, -1
60:     lw $jmp, $ra, 0
61:     jr $jmp
.sort
62:     sw $a0, $sp, 0
63:     sw $a1, $sp, 1
64:     sw $a2, $sp, 2
65:     lw $t10, $sp, 3
66:     lw $t11, $sp, 1
67:     move $t10, $t11
68:     sw $t10, $sp, 3
.L4
69:     lw $t12, $sp, 3
70:     lw $t13, $sp, 2
71:     li $t14, 1
72:     sub $t15, $t13, $t14
73:     slt $t0, $t12, $t15
74:     beq $t0, $zero, 122
75:     lw $t1, $sp, 4
76:     lw $t2, $sp, 0
77:     move $a0, $t2
78:     lw $t3, $sp, 3
79:     move $a1, $t3
80:     lw $t4, $sp, 2
81:     move $a2, $t4
```

```
 82:     addi $sp, $sp, 6
 83:     li $jmp, 87
 84:     sw $jmp, $ra, 0
 85:     addi $ra, $ra, 1
 86:     j 4
 87:     move $t5, $ret
 88:     addi $sp, $sp, -6
 89:     move $t1, $t5
 90:     sw $t1, $sp, 4
 91:     lw $t6, $sp, 5
 92:     lw $t8, $sp, 4
 93:     lw $t7, $sp, 0
 94:     add $t8, $t8, $t7
 95:     lw $t7, $t8, 0
 96:     move $t6, $t7
 97:     sw $t6, $sp, 5
 98:     lw $t10, $sp, 4
 99:     lw $t9, $sp, 0
100:     add $t10, $t10, $t9
101:     lw $t9, $t10, 0
102:     lw $t12, $sp, 3
103:     lw $t11, $sp, 0
104:     add $t12, $t12, $t11
105:     lw $t11, $t12, 0
106:     move $t9, $t11
107:     sw $t9, $t10, 0
108:     lw $t14, $sp, 3
109:     lw $t13, $sp, 0
110:     add $t14, $t14, $t13
111:     lw $t13, $t14, 0
112:     lw $t15, $sp, 5
113:     move $t13, $t15
114:     sw $t13, $t14, 0
115:     lw $t0, $sp, 3
116:     lw $t1, $sp, 3
117:     li $t2, 1
118:     add $t3, $t1, $t2
119:     move $t0, $t3
120:     sw $t0, $sp, 3
121:     j 69
.L5
122:     addi $ra, $ra, -1
123:     lw $jmp, $ra, 0
124:     jr $jmp
.main
125:     lw $t4, $sp, 1
126:     li $t5, 0
127:     move $t4, $t5
128:     sw $t4, $sp, 1
129:     li $t7, 0
130:     add $t7, $t7, $gp
131:     lw $t6, $t7, 0
132:     in $t8
133:     move $t6, $t8
134:     sw $t6, $t7, 0
135:     li $t9, 0
136:     move $a0, $t9
137:     move $t10, $a0
138:     out $t10
139:     nop
```

```
140:    li $t12, 1
141:    add $t12, $t12, $gp
142:    lw $t11, $t12, 0
143:    in $t13
144:    move $t11, $t13
145:    sw $t11, $t12, 0
146:    li $t14, 0
147:    move $a0, $t14
148:    move $t15, $a0
149:    out $t15
150:    nop
151:    li $t1, 2
152:    add $t1, $t1, $gp
153:    lw $t0, $t1, 0
154:    in $t2
155:    move $t0, $t2
156:    sw $t0, $t1, 0
157:    li $t3, 0
158:    move $a0, $t3
159:    move $t4, $a0
160:    out $t4
161:    nop
162:    li $t6, 3
163:    add $t6, $t6, $gp
164:    lw $t5, $t6, 0
165:    in $t7
166:    move $t5, $t7
167:    sw $t5, $t6, 0
168:    li $t8, 0
169:    move $a0, $t8
170:    move $t9, $a0
171:    out $t9
172:    nop
173:    li $t11, 4
174:    add $t11, $t11, $gp
175:    lw $t10, $t11, 0
176:    in $t12
177:    move $t10, $t12
178:    sw $t10, $t11, 0
179:    addi $t13, $gp, 0
180:    move $a0, $t13
181:    li $t14, 0
182:    move $a1, $t14
183:    li $t15, 5
184:    move $a2, $t15
185:    addi $sp, $sp, 2
186:    li $jmp, 190
187:    sw $jmp, $ra, 0
188:    addi $ra, $ra, 1
189:    j 62
190:    move $t0, $ret
191:    addi $sp, $sp, -2
192:    li $t2, 0
193:    add $t2, $t2, $gp
194:    lw $t1, $t2, 0
195:    move $a0, $t1
196:    move $t3, $a0
197:    out $t3
198:    nop
199:    li $t5, 1
```

```
200:    add $t5, $t5, $gp
201:    lw $t4, $t5, 0
202:    move $a0, $t4
203:    move $t6, $a0
204:    out $t6
205:    nop
206:    li $t8, 2
207:    add $t8, $t8, $gp
208:    lw $t7, $t8, 0
209:    move $a0, $t7
210:    move $t9, $a0
211:    out $t9
212:    nop
213:    li $t11, 3
214:    add $t11, $t11, $gp
215:    lw $t10, $t11, 0
216:    move $a0, $t10
217:    move $t12, $a0
218:    out $t12
219:    nop
220:    li $t14, 4
221:    add $t14, $t14, $gp
222:    lw $t13, $t14, 0
223:    move $a0, $t13
224:    move $t15, $a0
225:    out $t15
226:    nop
227:    j 228
.end
228:    halt
```

4.2.3 Binário

```

C- Binary Code
    001110_00000_11011_0000000000000000     // li
    001110_00000_11100_0000000000100000     // li
    001110_00000_11101_0000000000101111     // li
    010100_00000000000000000000001111101    // j
// minloc
    010000_11011_10001_0000000000000000     // sw
    010000_11011_10010_0000000000000001     // sw
    010000_11011_10011_0000000000000010     // sw
    001111_11011_00001_0000000000000101     // lw
    001111_11011_00010_0000000000000001     // lw
    011011_00010_00000_00001_00000000000    // move
    010000_11011_00001_0000000000000101     // sw
    001111_11011_00011_0000000000000100     // lw
    001111_11011_00101_0000000000000001     // lw
    001111_11011_00100_0000000000000000     // lw
    000010_00101_00100_00101_00000000000    // add
    001111_00101_00100_0000000000000000     // lw
    011011_00100_00000_00011_00000000000    // move
    010000_11011_00011_0000000000000100     // sw
    001111_11011_00110_0000000000000011     // lw
    001111_11011_00111_0000000000000001     // lw
    001110_00000_01000_0000000000000001     // li
    000010_00111_01000_01001_00000000000    // add
    011011_01001_00000_00110_00000000000    // move
```

```
27      010000_11011_00110_0000000000000011     // sw
28  // L0
29      001111_11011_01010_0000000000000011     // lw
30      001111_11011_01011_0000000000000010     // lw
31      001011_01010_01011_01100_00000000000    // slt
32      010001_01100_00000_0000000000110110     // beq
33      001111_11011_01110_0000000000000011     // lw
34      001111_11011_01101_0000000000000000     // lw
35      000010_01110_01101_01110_00000000000    // add
36      001111_01110_01101_0000000000000000     // lw
37      001111_11011_01111_0000000000000100     // lw
38      001011_01101_01111_10000_00000000000    // slt
39      010001_10000_00000_0000000000101111     // beq
40      001111_11011_00001_0000000000000100     // lw
41      001111_11011_00011_0000000000000011     // lw
42      001111_11011_00010_0000000000000000     // lw
43      000010_00011_00010_00011_00000000000    // add
44      001111_00011_00010_0000000000000000     // lw
45      011011_00010_00000_00001_00000000000    // move
46      010000_11011_00001_0000000000000100     // sw
47      001111_11011_00100_0000000000000101     // lw
48      001111_11011_00101_0000000000000011     // lw
49      011011_00101_00000_00100_00000000000    // move
50      010000_11011_00100_0000000000000101     // sw
51      010100_00000000000000000000101111       // j
52  // L1
53  // L2
54      001111_11011_00110_0000000000000011     // lw
55      001111_11011_00111_0000000000000011     // lw
56      001110_00000_01000_0000000000000001     // li
57      000010_00111_01000_01001_00000000000    // add
58      011011_01001_00000_00110_00000000000    // move
59      010000_11011_00110_0000000000000011     // sw
60      010100_00000000000000000000011000       // j
61  // L3
62       001111_11011_01010_0000000000000101     // lw
63       011011_01010_00000_11110_00000000000    // move
64       000100_11101_11101_1111111111111111     // addi
65       001111_11101_11111_0000000000000000     // lw
66       010101_11111_00000_0000000000000000     // jr
67       000100_11101_11101_1111111111111111     // addi
68       001111_11101_11111_0000000000000000     // lw
69       010101_11111_00000_0000000000000000     // jr
70  // sort
71      010000_11011_10001_0000000000000000     // sw
72      010000_11011_10010_0000000000000001     // sw
73      010000_11011_10011_0000000000000010     // sw
74      001111_11011_01011_0000000000000011     // lw
75      001111_11011_01100_0000000000000001     // lw
76      011011_01100_00000_01011_00000000000    // move
77      010000_11011_01011_0000000000000011     // sw
78  // L4
79      001111_11011_01101_0000000000000011     // lw
80      001111_11011_01110_0000000000000010     // lw
81      001110_00000_01111_0000000000000001     // li
82      000011_01110_01111_10000_00000000000    // sub
83      001011_01101_10000_00001_00000000000    // slt
84      010001_00001_00000_0000000001111010     // beq
85      001111_11011_00010_0000000000000100     // lw
86      001111_11011_00011_0000000000000000     // lw
```

```
87      011011_00011_00000_10001_00000000000    // move
88      001111_11011_00100_0000000000000011     // lw
89      011011_00100_00000_10010_00000000000    // move
90      001111_11011_00101_0000000000000010     // lw
91      011011_00101_00000_10011_00000000000    // move
92      000100_11011_11011_0000000000000110     // addi
93      001110_00000_11111_0000000001010111     // li
94      010000_11101_11111_0000000000000000     // sw
95      000100_11101_11101_0000000000000001     // addi
96      010100_00000000000000000000000100       // j
97      011011_11110_00000_00110_00000000000    // move
98      000100_11011_11011_1111111111111010     // addi
99      011011_00110_00000_00010_00000000000    // move
100     010000_11011_00010_0000000000000100     // sw
101     001111_11011_00111_0000000000000101     // lw
102     001111_11011_01001_0000000000000100     // lw
103     001111_11011_01000_0000000000000000     // lw
104     000010_01001_01000_01001_00000000000    // add
105     001111_01001_01000_0000000000000000     // lw
106     011011_01000_00000_00111_00000000000    // move
107     010000_11011_00111_0000000000000101     // sw
108     001111_11011_01011_0000000000000100     // lw
109     001111_11011_01010_0000000000000000     // lw
110     000010_01011_01010_01011_00000000000    // add
111     001111_01011_01010_0000000000000000     // lw
112     001111_11011_01101_0000000000000011     // lw
113     001111_11011_01100_0000000000000000     // lw
114     000010_01101_01100_01101_00000000000    // add
115     001111_01101_01100_0000000000000000     // lw
116     011011_01100_00000_01010_00000000000    // move
117     010000_01011_01010_0000000000000000     // sw
118     001111_11011_01111_0000000000000011     // lw
119     001111_11011_01110_0000000000000000     // lw
120     000010_01111_01110_01111_00000000000    // add
121     001111_01111_01110_0000000000000000     // lw
122     001111_11011_10000_0000000000000101     // lw
123     011011_10000_00000_01110_00000000000    // move
124     010000_01111_01110_0000000000000000     // sw
125     001111_11011_00001_0000000000000011     // lw
126     001111_11011_00010_0000000000000011     // lw
127     001110_00000_00011_0000000000000001     // li
128     000010_00010_00011_00100_00000000000    // add
129     011011_00100_00000_00001_00000000000    // move
130     010000_11011_00001_0000000000000011     // sw
131     010100_00000000000000000001000101       // j
132 // L5
133     000100_11101_11101_1111111111111111     // addi
134     001111_11101_11111_0000000000000000     // lw
135     010101_11111_00000_0000000000000000     // jr
136 // main
137     001111_11011_00101_0000000000000001     // lw
138     001110_00000_00110_0000000000000000     // li
139     011011_00110_00000_00101_00000000000    // move
140     010000_11011_00101_0000000000000001     // sw
141     001110_00000_01000_0000000000000000     // li
142     000010_01000_11100_01000_00000000000    // add
143     001111_01000_00111_0000000000000000     // lw
144     010110_00000_01001_00000_00000000000    // in
145     011011_01001_00000_00111_00000000000    // move
146     010000_01000_00111_0000000000000000     // sw
```

```
147     001110_00000_01010_0000000000000000     // li
148     011011_01010_00000_10001_00000000000    // move
149     011011_10001_00000_01011_00000000000    // move
150     010111_01011_00000_0000000000000000     // out
151     000000_00000000000000000000000000       // nop
152     001110_00000_01101_0000000000000001     // li
153     000010_01101_11100_01101_00000000000    // add
154     001111_01101_01100_0000000000000000     // lw
155     010110_00000_01110_00000_00000000000    // in
156     011011_01110_00000_01100_00000000000    // move
157     010000_01101_01100_0000000000000000     // sw
158     001110_00000_01111_0000000000000000     // li
159     011011_01111_00000_10001_00000000000    // move
160     011011_10001_00000_10000_00000000000    // move
161     010111_10000_00000_0000000000000000     // out
162     000000_00000000000000000000000000       // nop
163     001110_00000_00010_0000000000000010     // li
164     000010_00010_11100_00010_00000000000    // add
165     001111_00010_00001_0000000000000000     // lw
166     010110_00000_00011_00000_00000000000    // in
167     011011_00011_00000_00001_00000000000    // move
168     010000_00010_00001_0000000000000000     // sw
169     001110_00000_00100_0000000000000000     // li
170     011011_00100_00000_10001_00000000000    // move
171     011011_10001_00000_00101_00000000000    // move
172     010111_00101_00000_0000000000000000     // out
173     000000_00000000000000000000000000       // nop
174     001110_00000_00111_0000000000000011     // li
175     000010_00111_11100_00111_00000000000    // add
176     001111_00111_00110_0000000000000000     // lw
177     010110_00000_01000_00000_00000000000    // in
178     011011_01000_00000_00110_00000000000    // move
179     010000_00111_00110_0000000000000000     // sw
180     001110_00000_01001_0000000000000000     // li
181     011011_01001_00000_10001_00000000000    // move
182     011011_10001_00000_01010_00000000000    // move
183     010111_01010_00000_0000000000000000     // out
184     000000_00000000000000000000000000       // nop
185     001110_00000_01100_0000000000000100     // li
186     000010_01100_11100_01100_00000000000    // add
187     001111_01100_01011_0000000000000000     // lw
188     010110_00000_01101_00000_00000000000    // in
189     011011_01101_00000_01011_00000000000    // move
190     010000_01100_01011_0000000000000000     // sw
191     000100_11100_01110_0000000000000000     // addi
192     011011_01110_00000_10001_00000000000    // move
193     001110_00000_01111_0000000000000000     // li
194     011011_01111_00000_10010_00000000000    // move
195     001110_00000_10000_0000000000000101     // li
196     011011_10000_00000_10011_00000000000    // move
197     000100_11011_11011_0000000000000010     // addi
198     001110_00000_11111_0000000010111110     // li
199     010000_11101_11111_0000000000000000     // sw
200     000100_11101_11101_0000000000000001     // addi
201     010100_00000000000000000000111110       // j
202     011011_11110_00000_00001_00000000000    // move
203     000100_11011_11011_1111111111111110     // addi
204     001110_00000_00011_0000000000000000     // li
205     000010_00011_11100_00011_00000000000    // add
206     001111_00011_00010_0000000000000000     // lw
```

```
207     011011_00010_00000_10001_00000000000    // move
208     011011_10001_00000_00100_00000000000    // move
209     010111_00100_00000_0000000000000000     // out
210     000000_00000000000000000000000000       // nop
211     001110_00000_00110_0000000000000001     // li
212     000010_00110_11100_00110_00000000000    // add
213     001111_00110_00101_0000000000000000     // lw
214     011011_00101_00000_10001_00000000000    // move
215     011011_10001_00000_00111_00000000000    // move
216     010111_00111_00000_0000000000000000     // out
217     000000_00000000000000000000000000       // nop
218     001110_00000_01001_0000000000000010     // li
219     000010_01001_11100_01001_00000000000    // add
220     001111_01001_01000_0000000000000000     // lw
221     011011_01000_00000_10001_00000000000    // move
222     011011_10001_00000_01010_00000000000    // move
223     010111_01010_00000_0000000000000000     // out
224     000000_00000000000000000000000000       // nop
225     001110_00000_01100_0000000000000011     // li
226     000010_01100_11100_01100_00000000000    // add
227     001111_01100_01011_0000000000000000     // lw
228     011011_01011_00000_10001_00000000000    // move
229     011011_10001_00000_01101_00000000000    // move
230     010111_01101_00000_0000000000000000     // out
231     000000_00000000000000000000000000       // nop
232     001110_00000_01111_0000000000000100     // li
233     000010_01111_11100_01111_00000000000    // add
234     001111_01111_01110_0000000000000000     // lw
235     011011_01110_00000_10001_00000000000    // move
236     011011_10001_00000_10000_00000000000    // move
237     010111_10000_00000_0000000000000000     // out
238     000000_00000000000000000000000000       // nop
239     010100_00000000000000000011100100       // j
240 // end
241     000001_00000000000000000000000000       // halt
242 Binary Code Generated...
```

4.3 GCD

Um algoritmo que calcula o maximo divisor comum.

```
1  int input(void)
2  {
3  }
4
5  void output(int x)
6  {
7  }
8
9  int gcd (int u, int v)
10 {
11         if (v == 0) return u ;
12         else return gcd(v,u-u/v*v);
13         /* u-u/v*v == u mod v */
14 }
15
16 void main(void)
17 {       int x; int y;
18         x = input();
```

```
19    output(x);
20          y = input();
21          output(gcd(x,y));
22  }
```

4.3.1 Intermediário

```
1   C- Intermediate Code
2   File: teste1_binary.txt
3   (fun, gcd, -, -)
4   (arg, u, -, gcd)
5   (arg, v, -, gcd)
6   (load, $t0, v, -)
7   (immed, $t1, 0, -)
8   (get, $t2, $t0, $t1)
9   (let, $t3, $t0, $t1)
10  (and, $t4, $t2, $t3)
11  (iff, $t4, L0, -)
12  (load, $t5, u, -)
13  (ret, $t5, -, -)
14  (goto, L1, -, -)
15  (lab, L0, -, -)
16  (load, $t6, v, -)
17  (param, $t6, -, -)
18  (load, $t7, u, -)
19  (load, $t8, u, -)
20  (load, $t9, v, -)
21  (div, $t10, $t8, $t9)
22  (load, $t11, v, -)
23  (mult, $t12, $t10, $t11)
24  (sub, $t13, $t7, $t12)
25  (param, $t13, -, -)
26  (call, $t14, gcd, 2)
27  (ret, $t14, -, -)
28  (goto, L1, -, -)
29  (lab, L1, -, -)
30  (end, gcd, -, -)
31  (fun, main, -, -)
32  (alloc, x, 1, main)
33  (alloc, y, 1, main)
34  (load, $t15, x, -)
35  (call, $t0, input, 0)
36  (assign, $t15, $t0, -)
37  (store, x, -, $t15)
38  (load, $t1, x, -)
39  (param, $t1, -, -)
40  (call, $t2, output, 1)
41  (load, $t3, y, -)
42  (call, $t4, input, 0)
43  (assign, $t3, $t4, -)
44  (store, y, -, $t3)
45  (load, $t5, x, -)
46  (param, $t5, -, -)
47  (load, $t6, y, -)
48  (param, $t6, -, -)
49  (call, $t7, gcd, 2)
50  (param, $t7, -, -)
51  (call, $t8, output, 1)
```

```
(end, main, -, -)
(hlt, -, -, -)
End of execution
```

4.3.2 Assembly

```
C- Assembly Code
0:      li $sp, 0
1:      li $gp, 32
2:      li $ra, 47
3:      j 43
.gcd
4:      sw $a0, $sp, 0
5:      sw $a1, $sp, 1
6:      lw $t0, $sp, 1
7:      li $t1, 0
8:      sgeq $t2, $t0, $t1
9:      sleq $t3, $t0, $t1
10:     and $t4, $t2, $t3
11:     beq $t4, $zero, 18
12:     lw $t5, $sp, 0
13:     move $ret, $t5
14:     addi $ra, $ra, -1
15:     lw $jmp, $ra, 0
16:     jr $jmp
17:     j 40
.L0
18:     lw $t6, $sp, 1
19:     move $a0, $t6
20:     lw $t7, $sp, 0
21:     lw $t8, $sp, 0
22:     lw $t9, $sp, 1
23:     divi $t10, $t8, $t9
24:     lw $t11, $sp, 1
25:     mult $t12, $t10, $t11
26:     sub $t13, $t7, $t12
27:     move $a1, $t13
28:     addi $sp, $sp, 2
29:     li $jmp, 33
30:     sw $jmp, $ra, 0
31:     addi $ra, $ra, 1
32:     j 4
33:     move $t14, $ret
34:     addi $sp, $sp, -2
35:     move $ret, $t14
36:     addi $ra, $ra, -1
37:     lw $jmp, $ra, 0
38:     jr $jmp
39:     j 40
.L1
40:     addi $ra, $ra, -1
41:     lw $jmp, $ra, 0
42:     jr $jmp
.main
43:     lw $t15, $sp, 0
44:     in $t0
45:     move $t15, $t0
46:     sw $t15, $sp, 0
47:     lw $t1, $sp, 0
```

```
48:     move $a0, $t1
49:     move $t2, $a0
50:     out $t2
51:     nop
52:     lw $t3, $sp, 1
53:     in $t4
54:     move $t3, $t4
55:     sw $t3, $sp, 1
56:     lw $t5, $sp, 0
57:     move $a0, $t5
58:     lw $t6, $sp, 1
59:     move $a1, $t6
60:     addi $sp, $sp, 2
61:     li $jmp, 65
62:     sw $jmp, $ra, 0
63:     addi $ra, $ra, 1
64:     j 4
65:     move $t7, $ret
66:     addi $sp, $sp, -2
67:     move $a0, $t7
68:     move $t8, $a0
69:     out $t8
70:     nop
71:     j 72
.end
72:     halt
```

4.3.3 Binário

```

C- Binary Code
    001110_00000_11011_0000000000000000     // li
    001110_00000_11100_0000000000100000     // li
    001110_00000_11101_0000000000101111     // li
    010100_00000000000000000000101011       // j
// gcd
     010000_11011_10001_0000000000000000     // sw
     010000_11011_10010_0000000000000001     // sw
     001111_11011_00001_0000000000000001     // lw
     001110_00000_00010_0000000000000000     // li
     011001_00001_00010_00011_00000000000    // sgeq
     011010_00001_00010_00100_00000000000    // sleq
     001000_00011_00100_00101_00000000000    // and
     010001_00101_00000_0000000000010010     // beq
     001111_11011_00110_0000000000000000     // lw
     011011_00110_00000_11110_00000000000    // move
     000100_11101_11101_1111111111111111     // addi
     001111_11101_11111_0000000000000000     // lw
     010101_11111_00000_0000000000000000     // jr
     010100_00000000000000000000101000       // j
// L0
     001111_11011_00111_0000000000000001     // lw
     011011_00111_00000_10001_00000000000    // move
     001111_11011_01000_0000000000000000     // lw
     001111_11011_01001_0000000000000000     // lw
     001111_11011_01010_0000000000000001     // lw
     011100_01001_01010_01011_00000000000    // divi
     001111_11011_01100_0000000000000001     // lw
```

```
31        000110_01011_01100_01101_00000000000    // mult
32        000011_01000_01101_01110_00000000000    // sub
33        011011_01110_00000_10010_00000000000    // move
34        000100_11011_11011_0000000000000010     // addi
35        001110_00000_11111_0000000000100001     // li
36        010000_11101_11111_0000000000000000     // sw
37        000100_11101_11101_0000000000000001     // addi
38        010100_00000000000000000000000100       // j
39        011011_11110_00000_01111_00000000000    // move
40        000100_11011_11011_1111111111111110     // addi
41        011011_01111_00000_11110_00000000000    // move
42        000100_11101_11101_1111111111111111     // addi
43        001111_11101_11111_0000000000000000     // lw
44        010101_11111_00000_0000000000000000     // jr
45        010100_00000000000000000000101000       // j
46  // L1
47        000100_11101_11101_1111111111111111     // addi
48        001111_11101_11111_0000000000000000     // lw
49        010101_11111_00000_0000000000000000     // jr
50  // main
51        001111_11011_10000_0000000000000000     // lw
52        010110_00000_00001_00000_00000000000    // in
53        011011_00001_00000_10000_00000000000    // move
54        010000_11011_10000_0000000000000000     // sw
55        001111_11011_00010_0000000000000000     // lw
56        011011_00010_00000_10001_00000000000    // move
57        011011_10001_00000_00011_00000000000    // move
58        010111_00011_00000_0000000000000000     // out
59        000000_00000000000000000000000000       // nop
60        001111_11011_00100_0000000000000001     // lw
61        010110_00000_00101_00000_00000000000    // in
62        011011_00101_00000_00100_00000000000    // move
63        010000_11011_00100_0000000000000001     // sw
64        001111_11011_00110_0000000000000000     // lw
65        011011_00110_00000_10001_00000000000    // move
66        001111_11011_00111_0000000000000001     // lw
67        011011_00111_00000_10010_00000000000    // move
68        000100_11011_11011_0000000000000010     // addi
69        001110_00000_11111_0000000001000001     // li
70        010000_11101_11111_0000000000000000     // sw
71        000100_11101_11101_0000000000000001     // addi
72        010100_00000000000000000000000100       // j
73        011011_11110_00000_01000_00000000000    // move
74        000100_11011_11011_1111111111111110     // addi
75        011011_01000_00000_10001_00000000000    // move
76        011011_10001_00000_01001_00000000000    // move
77        010111_01001_00000_0000000000000000     // out
78        000000_00000000000000000000000000       // nop
79        010100_00000000000000000001001000       // j
80  // end
81        000001_00000000000000000000000000       // halt
```

5 Conclusão

A primeira etapa da disciplina foi relativamente simples sendo proposto a retomada do que fora desenvolvido durante a matéria de compiladores. Porém, a partir do segundo ponto de checagem as dificuldades aumentaram gradativamente. O primeiro grande problema foi a elaboração do código intermediário, devido ao fato da árvore ter sido trabalhada de maneira simples deixando de ter fatores importantes que seriam usados nesta etapa. Todas as fases seguintes que dependiam da fase anterior (da árvore) para sua execução acabavam por resultar em erro. Por esse motivo, o código intermediário que gerava quadruplas parcialmente completas porém sem os nós que se enquadravam no tipo identificador. Foi necessário revisitar toda a árvore e reestrutura-la para que fosse feita as devidas correções dos erros que eram encontrados nesta fase do projeto.

Após essa parte, o projetou estagnou em um erro de entrada, onde o processador realizava apenas a primeira leitura. Porém, após perceber que o erro era por conta da forma quer foi tratado o clock ficou mais fácil dar sequência ao compilador e apresentar o projeto.

De modo geral, a elaboração do Compilador e o fato de precisar revisitar varias vezes a arquitetura do mesmo ajudou a desenvolver bem as habilidade de programação e criatividade na hora de resolver problemas, uma vez que por mais que se tenha ajuda, é um projeto seu, e quem precisa entender e saber consertar os erros apresentados é o próprio aluno.

Referências

1 STALLINGS, W. Arquitetura e Organização de Computadores. 8th. ed. EUA: Prentice-Hall Brasil, 2010. Citado na página 9.

2 RISC x CISC. Disponível em: <http://producao.virtual.ufpb.br/books/edusantana/introducao-a-arquitetura-de-computadores-livro/livro/livro.chunked/ch04s04.html>. Acesso em: 09/04/2018. Citado na página 9.

3 MIPS. Disponível em: <http://www.ic.unicamp.br/~pannain/mc542/aulas/ch3_arq.pdf>. Acesso em: 09/04/2018. Citado na página 10.

4 ARQUITECTURA de Computadores. Disponível em: <http://gec.di.uminho.pt/discip/TextoAC/cap12.html>. Acesso em: 08/03/2018. Citado na página 11.

5 TUTORIAL de Verilog - Operadores. Disponível em: <https://www.embarcados.com.br/tutorial-de-verilog-operadores/>. Acesso em: 15/05/2018. Citado na página 13.

6 O Conjunto de Instruções do Processador. Disponível em: <http://www.facom.ufu.br/~claudio/Cursos/AOC2/Slides/aoc2_aula4.0_2p.pdf>. Acesso em: 12/04/2018. Citado na página 13.

7 PROCESSOS de Compilação. Disponível em: <https://gbritoo.wordpress.com/2014/09/19/processo-de-compilacao/>. Acesso em: 04/07/2019. Citado na página 20.

8 LOUDEN, K. C. Compiler construction.Principles and Practice, 1997. Citado na página 22.