João Victor de Mesquita Cândido dos Santos

Implementação do Compilador C-

São José dos Campos - Brasil

Junho de 2019

João Victor de Mesquita Cândido dos Santos

Implementação do Compilador C-

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

Docente: Prof. Dr. Luiz Eduardo Galvão Martins

Universidade Federal de São Paulo - UNIFESP

Instituto de Ciência e Tecnologia - Campus São José dos Campos

São José dos Campos - Brasil

Junho de 2019

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

Segundo a definição do dicionário, o computador é um aparelho eletrônico usado para processar, guardar e tornar acessível uma informação de variados tipos (1). Os computadores vem se tornando cada vez complexos, poderosos e desenvolvidos para desenvolver as mais diferenciadas tarefas. Com o avanço no desenvolvimento de tecnologias os computadores trazem um grande impacto e uma evolução na sociedade em diversas áreas. O uso de computadores na era das máquinas em que vivemos vem se tornando indispensável em várias atividades que são realizadas diariamente, fazendo com que a demanda por novas tecnologias ágeis, rápidas, com uma alta performance e baixo custo se torne cada vez mais crescente para agradar os usuários, entretanto para que isso ocorra é necessário entender qual é o funcionamento por trás dos sistemas computacionais, como são desenvolvidos e como cada componente é importante para que tudo flua da melhor maneira possível.Assim o desenvolvimento de sistemas computacionais traz consigo uma enorme importância para o mundo todo já que isso contribui para que a sociedade siga em constante evolução.

A especificação de um computador consiste na descrição do seu nível mais baixo,sua arquitetura,organização e conjunto de instruções(2). O conjunto de instruções é tratado por componente à nível de *hardware* obedecendo programas gravados nas unidades de memória do processador.

Para que um programa possa ser executado de acordo com o conjunto definido se faz necessário o uso de um compilador que é um código capaz de ler um programa definido por uma linguagem de programação e traduzindo o mesmo em linguagem de máquina gerando assim um arquivo em binário compatível ao *hardware* da máquina em que o mesmo será executado. O compilador realiza esse processo de tradução em duas fases, a primeira fase é conhecida como Fase de Análise que realiza as etapas de análise léxica, sintática e semântica. A segunda etapa é conhecida como Fase de Síntese que tem por objetivo final, gerar de fato o código alvo em binário.

Sendo assim, o presente relatório tem como foco abordar desenvolvimento e implementação de um compilador que seja capaz de realizar a tradução de códigos escritos na linguage de programação C- proposta por (3) para um código em binário respeitando o conjunto de instruções definido pelo processador AKIRAPROC©desenvolvido na disciplina de Laboratório de Sistemas Computacionais: Arquitetura e Organização de Computadores e validando os resultados através de testes no FPGA.

1.1 Descrição

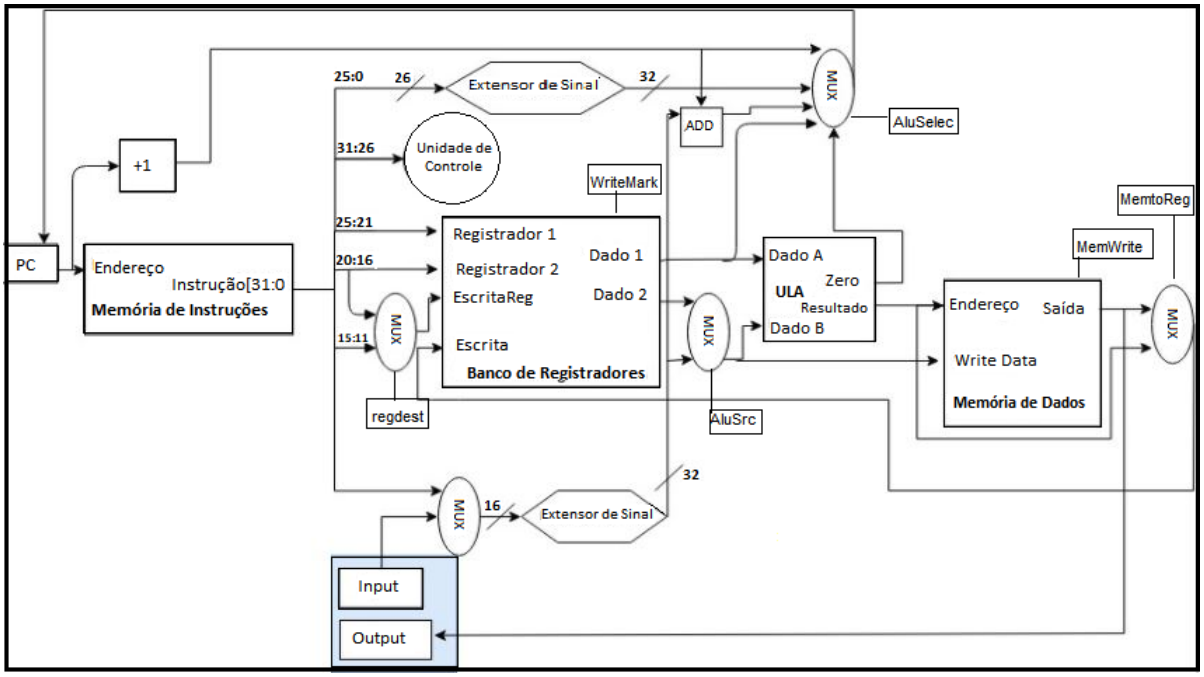
A lista a seguir apresenta uma descrição sucinta do conteúdo dos capítulos seguintes deste trabalho:

- Capítulo 2: Aborda a Arquitetura do Processador.
- Capítulo 3: Aborda a explicação de cada um dos componentes que compõem o Processador.
- Capítulo 4: Aborda as Instruções bem como os formatos e o conjunto de instruções utilizado no processador.
- Capítulo 5: Aborda a Organização da Memória no Processador.
- Capítulo 6: Aborda a descrição da modelagem de todas as etapas na Fase de Análise.
- Capítulo 7: Aborda a descrição da modelagem de todas as etapas na Fase de Síntese.
- Capítulo 8: Apresenta exemplos de três códigos e os resultados obtidos na Fase de Síntese
- Capítulo 9: Inclui a conclusão do projeto, citando as dificuldades, desafios e os destaques relacionados ao desenvolvimento do projeto.

2 Arquitetura do Processador

A arquitetura utilizada apresenta um *Datapath*, ou caminho de dados, semelhante ao da arquitetura *MIPS* como mostra a Figura 2. No processador desenvolvido se encontram alguns componentes característicos com o objetivo de se obter seu devido funcionamento. Os componentes principais do processador são o Contador de Programa (*Program Counter*), Memória de Instruções, Banco de Registradores, Unidade Lógica e Aritmética, Memória de Dados e foi adicionado um Bloco de *Input*/*Output* responsável pela entrada e saída de dados do processador. Além dos componentes principais o processador também apresenta seus componentes auxiliares tais como os multiplexadores e extensores de sinais, ambos os componentes podem ser visualizados na Figura 2.

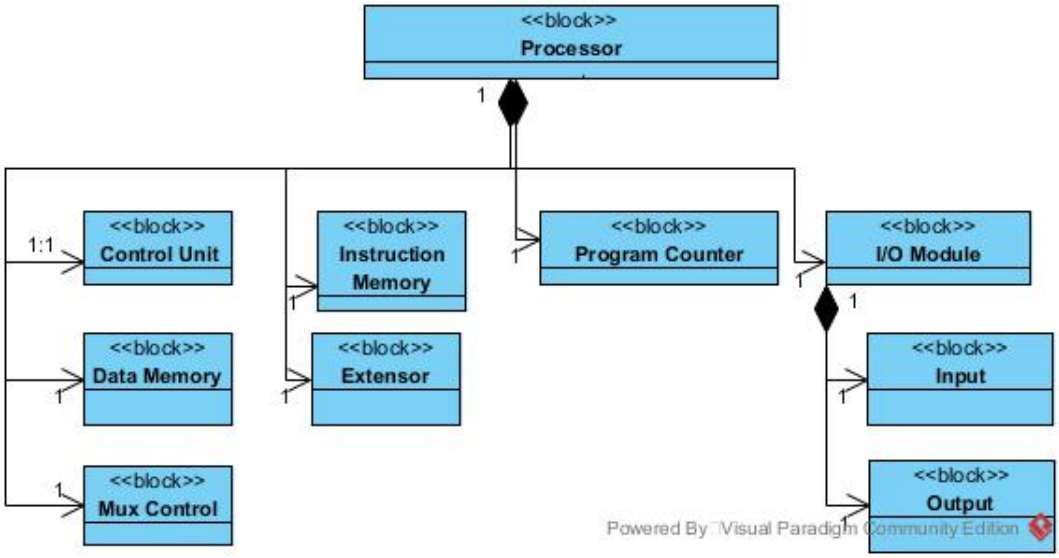
Figura 1 – Caminho de Dados



Fonte: O Autor

2.1 Diagrama de Blocos Processador

Figura 2 – Diagrama de Blocos do Processador



Fonte: O Autor

3 Explicação dos Componentes do Processador

3.1 Contador de Programa

O contador de programa é responsável pelo fluxo das instruções no processado, nele há um registrador que tem a responsabilidade de armazenar o endereço da instrução atual que está em execução, assim sempre que há uma subida no *clock*, o valor é atualizado de acordo com alguns sinais de controle.

Há trés possibilidades para que ocorra a atualização do valor do registrador. Na primeira situação o valor atual do Pc é incrementado com uma unidade para que haja a execução da próxima instrução, na segunda situação caso ocorra algum desvio seja ele de *branch* ou *jumo* é o valor do PC é atualizado com o valor de endereço de desvio. Na terceira situação caso o sinal de *reset* seja igual a 1, o valor do registrador Pc é zerado inciando a contagem novamente. Também há uma situação que utiliza um outro sinal de *halt*, caso esse sinal seja igual a 1, ocorre uma parada na contagem do Pc até que o usuário decida retomar a contagem do mesmo.

3.2 Memória de Instruções

A Memória de Instruções, apresenta a capacidade de armazenar até 256 instruções e como segue o padrão *MIPS*, cada instrução tem um tamanho fixo de 32 *bits*. A entrada consiste em um valor representando um endereço referente a posição de memória aonde a instrução está, tal posição de memória é representado por um índice de um vetor.

3.3 Banco de Registradores

O Banco de Registradores, é responsável por armazenar os 32 registradores de propósito geral utilizados em todas as operações para o armazenamento de dados do programa, dados de entrada ou endereços.

O banco apresenta quatro diferentes entradas e duas saídas, sendo três delas possíveis endereços de registradores de onde se deseja ler ou escrever dados. A primeira entrada é dada pelo endereço do primeiro registrador fonte (RS), a segunda entrada se refere ao endereço do segundo registrador fonte (RT), a terceira entrada de acordo com a Figura 2 pode receber o endereço do registrador de destino (RD) no caso de operações lógicas e aritméticas ou recebe um endereço para o registrador de onde se deseja inserir um

valor de imediato. A quarta entrada se refere aos dados provenientes da memória de dados que serão armazenados no registrador de escrita. As duas saídas se referem aos dados que estão armazenados e serão lidos através do endereço dos dois primeiros registradores fonte.

A escrita em algum registrador é feita de acordo com um sinal de controle que caso tenha seu valor em nível alto (valor 1) sinaliza a escrita no registrados, caso contrário é feito a leitura.

3.4 Unidade Lógica e Aritmética

A ULA (Unidade Lógica e Aritmética) é responsável por todas as operações realizadas pelo processador, sejam elas operações lógicas ou operações aritméticas utilizando dos dados fornecidos pelo processador. A ULA apresenta um funcionamento independente do *clock* fazendo com que funcione apenas com os seus sinais de entrada.

A ULA pode realizar diferentes operações, assim para que possa haver uma diferenciação dessas operações o componente apresenta um sinal de controle de cinco *bits* chamado de *opcode* que irá selecionar a devida operação de acordo com o seu valor como é visto na Tabela 1.

Tabela 1 – Conjunto de Operações Realizadas pela ULA

opcode	instrução
00000	adição
00001	subtração
00010	*set on less than*
00011	multiplicação
00100	*branch if equal*
00101	divisão
00110	*shift left*
00111	*shift right*
01000	negação
01001	AND *bit* a *bit*
01010	OR *bit* a *bit*
01011	XOR *bit* a *bit*
01100	*set on less than equal*
01101	*set on grater than equal*
01110	*set on greater than*
01111	*MOD*

Fonte: O Autor

É importante citar que no componente também há um sinal de controle chamado de *zero* utilizado nas instruções de saltos condicionais como *branch if equal* e *branch if not equal*, tal sinal de controle verifica se o valor das entradas são iguais ou diferentes para que haja um desvio ou não.

3.5 Memória de dados

A Memória de Dados, é um dos últimos componentes que compõem o caminho de dados do processador em questão e é acessada apenas nas instruções de *load* e *store*. Tal componente é responsável pelo armazenamento de dados do sistema, funciona de maneira similar ao banco de registradores, apresentando um sinal de controle para controlar a escrita ou leitura da memória. Entretanto a diferença está nas entradas em que há apenas duas, a primeira é a do endereço que se deseja armazenar o dado e a segunda entrada é o dado que de fato será armazenado.

A escrita é feita na memória através da subida de cada *clock* a fim de evitar que haja a leitura de um dado que ainda não foi calculado. A saída da memória de dados é transmitida em todo o ciclo de *clock*, entretanto como na sua saída há um multiplexador que irá decidir se vai transmitir o dado lido da memória ou o resultado de uma operação realizada pela ULA, tal dado pode ser escrito ou não no banco de registradores.

3.6 Multiplexador

Os multiplexadores são controlados através dos sinais de controle e com isso é possível determinar qual é a informação de dados que um respectivo bloco irá receber, assim os multiplexadores ajudam no controle do fluxo dos bits no processador.

3.7 Extensor de Sinal

O extensor de sinal é muito importante pois pode ocorrer a existência de palavras que não tenham um tamanho 32 *bits* fazendo com haja um problema com a comunicação de dados. Assim, o papel do extensor é fazer com que seja extraído o subconjunto de interesse afim de transformar o dado em um dado de 32 *bits*

3.8 Módulo de Entrada

O módulo de entrada, como o próprio nome já diz, é responsável pela entrada de algum dado no processador que é selecionado pelo usuário através a leitura das chaves(*switches*) no FPGA, tal dado pode ser utilizado posteriormente em outras instruções.

3.9 Módulo de Saída

O módulo de saída é responsável por mostrar algum valor de saída para o usuário através dos *displays* do FPGA.

3.10 Unidade de Controle

A Unidade de Controle é responsável pela configuração de todos os sinais de controle no processador. Existem dois tipos de Unidade de Controle, a *Hardwired* e a Microprogramada, no caso do projeto em questão é utilizada a *Hardwired* em que os sinais de controle são gerados a partir da identificação de qual instrução está sendo executada, sendo assim, através da análise do *opcode* é capaz de configurar as especificações necessárias para que ocorra o fluxo de dados de maneira correta na Unidade de Processamento. Tais sinais podem ser vistos na Tabela 2 abaixo.

Tabela 2 – Sinais de Controle

Sinais de Controle	**Módulo**	**Função**
WriteMark	Banco de Registradores	Escrita no Banco
AluSrce	MUX ULA	Escolha entre imediato ou dado do banco
AluOp	Operação da ULA	ULA
AluSelec	MUX PC	Escolha do próximo endereço de PC
MemReg	MUX pós Memória de Dados	Escolha entre dado da memória ou resultado da ULA
RegDest	MUX pré Banco de Registradores	Escolhe registrador de destino
MemWrite	Memória de Dados	Escrita na Memória
beq	ULA,MUXPC e PC	verifica *branch*
bne	ULA,MUXPC e PC	verifica *branch*
hlt	PC	Para a contagem do PC
rst	PC	Reseta a contagem do PC
flag	Módulo de Saída	Mostra algum valor no *display*
flagIN	Módulo de Entrada	Lê algum dado para o Processador
checkin	Módulo de Entrada	Para o processador para fazer a leitura do dado

Fonte: O Autor

Os únicos dois sinais que funcionam de maneira diferente são, o sinal que refere à Unidade Lógica e Aritmética que recebe um *opcode* determinando a operação a ser realizada e o sinal referente multiplexador que controla o próximo endereço de PC pois este apresenta quatro possibilidades diferentes de seleção. Além disso há um sinal de entrada chamado *checkin* tal sinal é selecionado pelo usuário através de um *switch* do FPGA indicando o fim da seleção de um dado que será lido pelo registrador, enquanto esse *switch* não for selecionado o sinal de controle *hlt* permanece ligado impedindo que o processador vá para a próxima instrução.

Dentre todos os sinais de controle existentes nesse módulo, há dois deles que valem ser citados pois estes monitoram as instruções de *Input* e *Output*.

O primeiro sinal de controle é a *flagIN* que é utilizado apenas na instrução de *Input*,

tal sinal de controle é responsável pela seleção de um multiplexador que irá optar por fazer a leitura das chaves(*switches*) realizando assim a entrada de algum valor escolhido pelo usuário no processador ou então selecionará um valor que virá do Banco de Registradores representando um endereço que é utilizado pela instrução de *jump*.

O segundo sinal é o *flag* tal sinal é utilizado apenas na instrução de *Output* para que quando seja ativado possa mostrar o valor de saída desejado em um *display*.

4 Instruções

4.1 Formato de Instruções

Como a arquitetura que foi desenvolvida no projeto é semelhante a arquitetura *MIPS* monociclo o projeto apresenta características semelhantes ao processador real bem como seus formatos de instruções. Tais formatos de instruções que foram implementados no desenvolvimento do processador podem ser vistos a seguir.

O primeiro tipo de instrução de acordo com a Tabela 3 é o tipo R que simboliza o uso de registradores e é nesse tipo que se encontram todas as instruções de operações lógicas e aritméticas tais como adição, subtração multiplicação.

Tabela 3 – Formato de Instruções do tipo R

Tamanho (bits)	6	5	5	5	5	6
Campo	Opcode	RS	RT	RD	*shamt*	não utilizado
Bits	31 - 26	25 - 21	20 - 16	15 - 11	10 - 6	5 - 0

Fonte: O Autor

O segundo tipo de instrução conforme a Tabela 4 abaixo é o tipo I que simboliza instruções que executam operações lógicas e aritméticas com dados do campo de *offset*, executam saltos condicionais e realizam acesso à memória.

Tabela 4 – Formato de Instruções do tipo I

Tamanho (bits)	6	5	5	16
Campo	*Opcode*	RS	RT	Endereço de Desvio
Bits	31 - 26	25 - 21	20 - 16	15 - 0

Fonte: O Autor

O tipo de instruções J como pode ser visto na Tabela 5 é o tipo mais simples e representa instruções de saltos.

Tabela 5 – Formato de Instruções do tipo J

Tamanho (bits)	6	26
Campo	*Opcode*	Endereço de Salto
Bits	31 - 26	25 - 0

Fonte: O Autor

4.2 Conjunto de Instruções

Como este projeto é baseado na arquitetura *MIPS*, foram escolhidas 25 instruções para o desenvolvimento do projeto, afim de suprir as necessidades básicas do processador. As instruções são similares aos três formatos da arquitetura *MIPS* (R, I e J) respeitando suas especificações definidas pela Tabela 3, Tabela 4 e Tabela 5 já apresentadas anteriormente. Todas as instruções apresentam um tamanho fixo de 32 *bits* e podem realizar operações lógicas, aritméticas, de salto e acesso à memória, como pode ser visto na Tabela 6. Abaixo segue uma legenda para facilitar na compreensão dos dados apresentados na tabela:

- RS e RT representam os registradores fonte;
- RD representa o registrador de destino;
- IM representado um valor de imediato;
- PC representa o *Program Counter* ou Contador de Programa;
- M representa um acesso a memória;
- R representa o uso de Registradores;

Tabela 6 – Conjunto de Instruções

Instrução	abreviação	Opcode	Operação
adição	add	000000	R[RD] ← R[RS]+R[RT]
subtração	sub	000001	R[RD]←R[RS]-R[RT]
adição com imediato	addi	000010	R[RD]←R[RS]+IM
subtração com imediato	subi	000011	R[RD]←R[RS]-IM
multiplicação	mult	000100	R[RD]←R[RS] x R[RT]
divisão	div	000101	R[RD]←R[RS] / R[RT]
set on less than	slt	000110	if(R[RS] < R[RT])R[RD]=1
not	NOT	000111	R[RD]←!R[RS]
and	AND	001000	R[RD]←R[RS] & R[RT]
or	OR	001001	R[RD]←R[RS] | R[RT]
xor	XOR	001010	R[RD]←R[RS] ∧ R[RT]
load word	lw	001011	R[RT]←M[IM+ R[RS]]
load imediato	ldi	001100	R[RD]←IM
store word	sw	001101	M[R[RS]+IM]←R[RT]
branch if equal	beq	001110	IF(R[RS] == R[RT]) PC←IM
branch if not equal	bne	001111	IF(R[RS] != R[RT]) PC←IM
jump	jmp	010000	PC ← IM
jump to register	jmpr	010001	PC←R[RS]
not an operation	NOP	010010	não realiza operação
halt	HALT	010011	para o processador (contagem do pc)
input	IN	010100	R[RS]←IN
output	OUT	010101	OUT←R[RS]
move	MOVE	010110	R[RD]←R[RS]
shift left	shl	010111	R[RD]←sl(shamt)R[RS]
shift right	shr	011000	R[RD]←sr(shamt)R[RS]
set on less than equal	sle	011001	if(R[RS] <= R[RT])R[RD]=1
set on great than equal	sge	011010	if(R[RS] >= R[RT])R[RD]=1
set on great than	sgt	011011	if(R[RS] > R[RT])R[RD]=1
mod	mod	011100	R[RD] ← R[RS]%R[RT]

Fonte: O Autor

5 Organização da Memória

A organização da memória foi implementada de tal forma que respeita a arquitetura *Harvard* da memória de instruções. Ambas as memórias são organizadas de forma semelhante a uma matriz, cada linha corresponde a uma posição da memória, sendo estas compostas por um tamanho de 32 *bits*. Ao tratar da implementação da memória de dados esta foi feita de tal forma que as instruções *load word* e *store word* acessam um endereço específico utilizando endereçamento por deslocamento onde o endereço é calculado através da soma do conteúdo de um registrador base com um valor que está no campo de imediato da instrução. É importante citar que ao se tratar de mudanças de contexto, chamadas de funções e funções recursivas se fez necessário a implementação de uma pilha no processador. A pilha foi implementada na memória de dados, de tal modo que existe um registrador do banco de registradores *$sp* que salva a posição do topo dessa pilha, fazendo com que o acesso da mesma seja feita através de instruções de *load* onde o mesmo realiza instruções *push* com valores de imediato negativos, e *pop* com valores positivos.

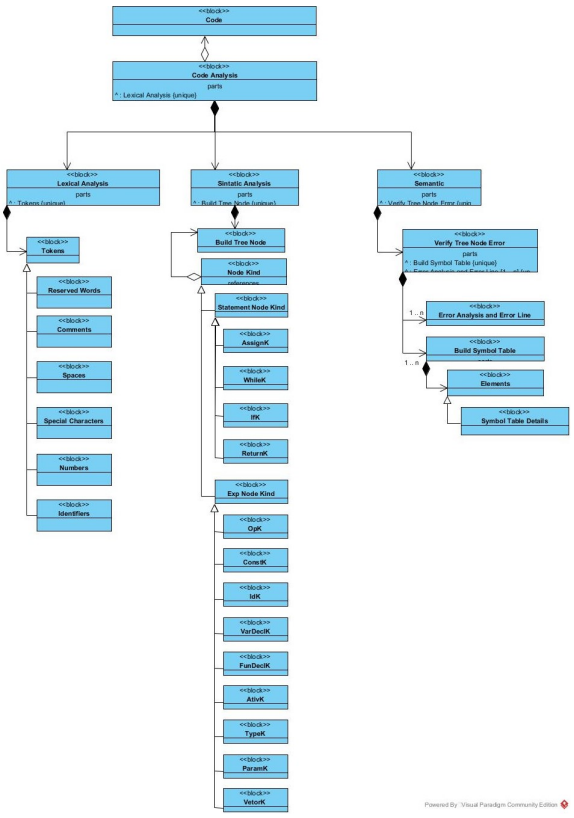
6 Compilador: Fase Análise

6.1 Modelagem

A construção da fase de análise de um compilador é feita a partir da entrada de um arquivo código escrito em C-, essa construção é chamada de fase de análise da qual é composta por três etapas. A primeira parte se refere a análise léxica, esta tem por objetivo identificar cada um dos caracteres do código e produzir uma sequência de *tokens* que podem ser utilizados posteriormente em outras etapas. A segunda etapa se refere a análise sintática que a partir dos *tokens* gerados na etapa anterior, é criada uma árvore de análise sintática de acordo com os *tokens* gerados faz uma análise hierárquica através da análise da estrutura gramatical. A última parte se refere a análise semântica que tem como objetivo gerar uma tabela de símbolos a partir da verificação de regras específicas da gramática que devem ser respeitadas para averiguar que não há existência de erros semânticos.

6.1.1 Diagrama de Blocos (SysML)

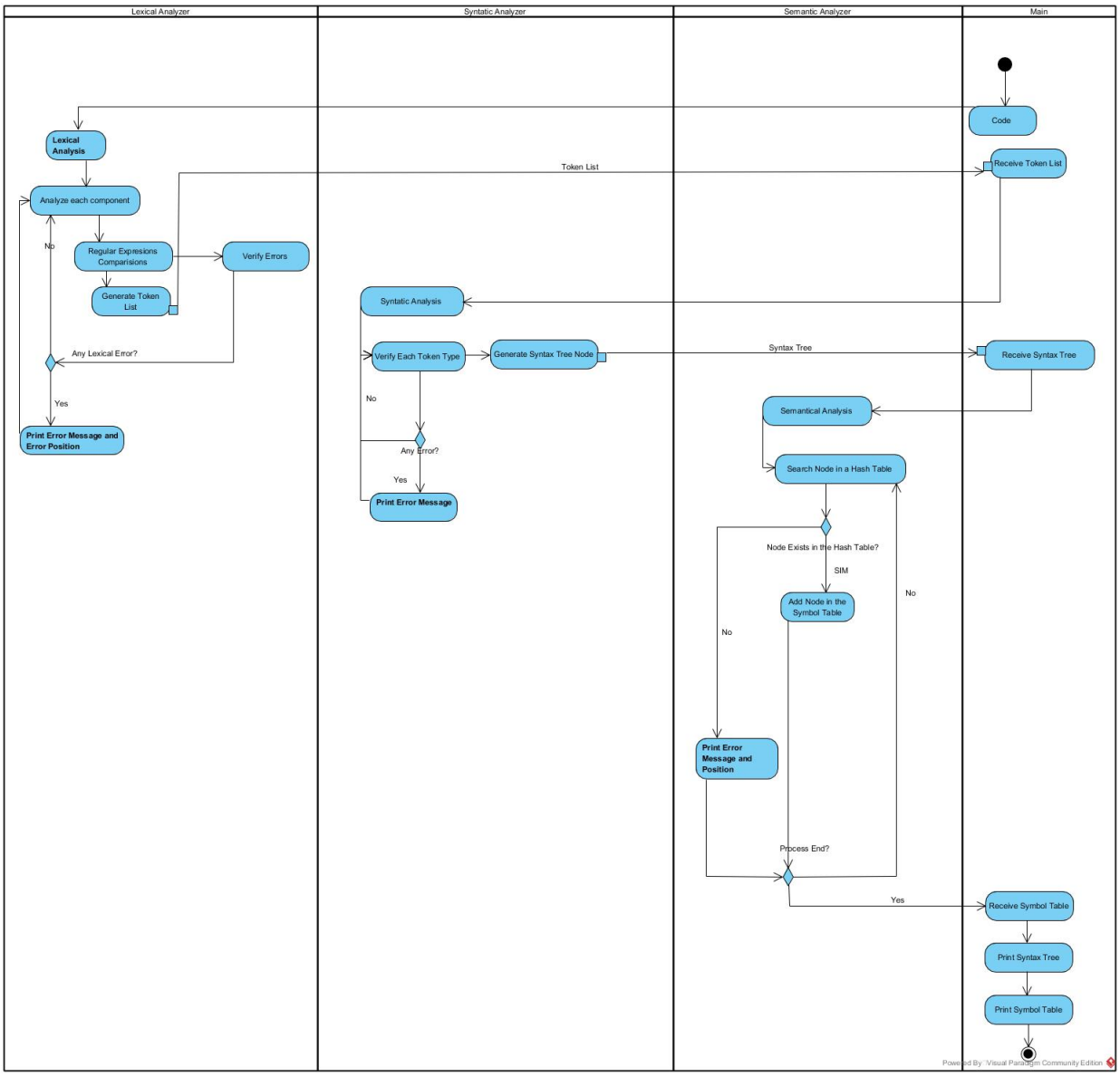
Figura 3 – Diagrama de Blocos



Fonte: O Autor

6.1.2 Diagramas de Atividades (SysML)

Figura 4 – Diagrama de Atividades



Fonte: O Autor

6.2 Análise Léxica

Como dito anteriormente, a primeira parte da fase de análise do compilador é a fase de análise léxica (3), onde é feita uma varredura no código de entrada, tal código é lido com uma cadeia de caracteres que são separados em *tokens* que são definidos por uma sequência de caracteres ou símbolos que representam alguma informação, tal informação representa unidades lógicas do arquivo fonte que serão utilizados nas próximas etapas da fase de análise.

Os *tokens* podem ser representados por palavras reservadas, identificadores, símbolos especiais e números. Para a análise léxica, a geração dos *tokens* é feita com base em expressões regulares e autômatos finitos definidos por uma tabela de convenções léxicas para a linguagem C- como mostra a Tabela 7 abaixo.

Tabela 7 – Convenção léxica C-

Convenção léxica C-	
Identificadores	Iniciado por uma letra, podendo ser seguido por mais letras ou números.
Números	Inteiros, sem sinal.
Comentários	/* Comentários */.
Palavras Reservadas	*if*,*else*, *void*, *return*, *int*, *while*.
Símbolos	+, -, *, /, ;, {, }, (,), [,], =, ==, !=, <, <=, >, >=.
Espaço	Podem ser brancos, \n (quebra de linha) ou \t (tabulação)

A elaboração do analisador foi feita utilizando o gerador de varreduras *Flex(Fast Lex)* na linguagem C, que de maneira simples consegue definir os *tokens*. O processo parte do algoritmo em C- que divide os caracteres e expressões transformando-os em *tokens* isso é feito a partir da geração de um arquivo em C que através de uma rotina de varredura denominada de **yylex** gera um arquivo final denominado de **lex.yy.c** que será compilado gerando um executável que posteriormente itera sobre um arquivo desejado que se queira realizar uma análise. Para que a análise fosse feita de maneira correta algumas expressões regulares foram escolhidas para serem utilizadas no *Flex*, de acordo com o Algoritmo 1 abaixo.

Algoritmo 1 - Expressões Regulares

```
1 digit       [0-9]
2 number      {digit}+
3 letter      [a-zA-Z]
4 identifier  {letter}+
5 newline     \n
6 whitespace  [ \t\r]+
```

De acordo com o que foi dito anteriormente, o Algoritmo 2 abaixo representa um trecho de um código de algoritmo em C-.

Algoritmo 2 - Exemplo de Código em C-

```
1 int x;
2 y = 2;
```

Através da varredura a saída da análise leva em consideração o lexema obtido e em qual linha o mesmo se encontra.

1 **Palavra Reservada:** int

1 **Identificador:** x

1 **Símbolo Especial:** ;

2 **Identificador:** y

2 **Número:** 2

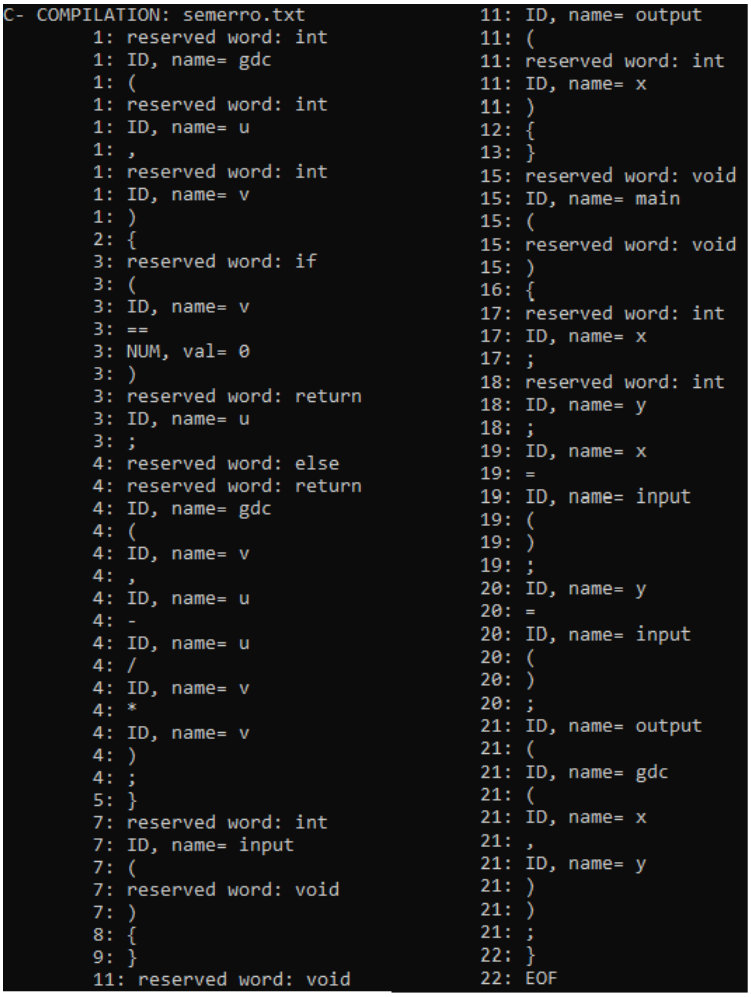
2 **Símbolo Especial:** ;

O Algoritmo 3 abaixo representa um código que foi submetido a uma análise léxica e o resultado da análise pode ser visto na Figura 5 em seguida.

Algoritmo 3 - Análise Léxica

```
1
2  int gdc (int u, int v)
3  {
4      if (v == 0) return u;
5      else return gdc(v,u-u/v*v);
6  }
7
8  int input(void)
9  {
10 }
11
12 void output(int x)
13 {
14 }
15
16 void main(void)
17 {
18    int x;
19    int y;
20    x = input();
21    y = input();
22    output(gdc(x,y));
23 }
```

Figura 5 – Análise Léxica - Parte 1

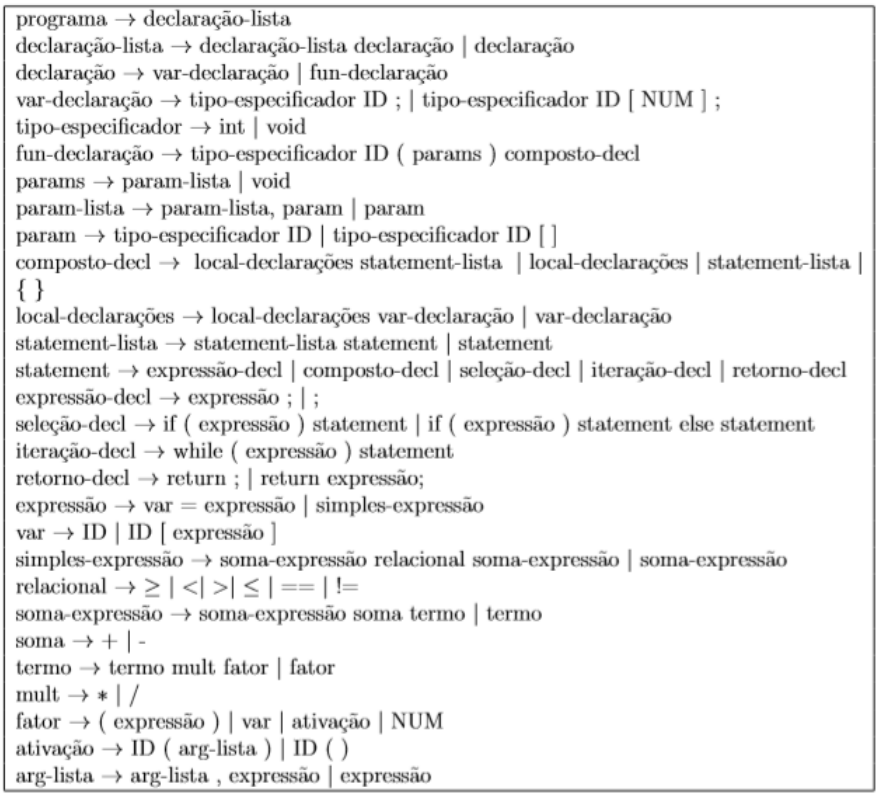


Fonte: O Autor

6.3 Análise Sintática

A segunda etapa da Fase de Análise é a fase de Análise Sintática, comumente conhecida como *parser* (3) apresenta como função analisar os *tokens* gerados na fase de análise léxica e formar uma árvore sintática. Para que a análise seja feita de maneira eficaz o compilador leva em conta uma gramática livre de contexto que determina como a estrutura será construída, caso a análise não encontre nenhum erro sintático a árvore de análise sintática é gerada. Para que a análise seja feita de maneira eficiente, o analisador léxico recebe os *tokens* e os compara com uma gramática BNF, tal gramática possui padrões pré-definidos para a linguagem C-. Como pode ser vista na Figura 6 abaixo.

Figura 6 – Gramática para Linguagem C-



Fonte: O Autor

Para que tal análise fosse analisada de maneira eficiente foi utilizado o gerador *parser Bison* em C que gera uma subrotina responsável pela análise sintática de um código fonte. O *Bison* é feito de uma maneira recursiva sobre uma gramática LR que a partir da verificação de um único símbolo este consegue obter o próximo *token* através de uma derivação à esquerda, ou seja, a partir de uma regra de início ele consegue acesar as próximas regras através de substituições onde cada termo da expressão que forma regra pode implicar em uma outra regra ou em um terminal.

O Algoritmo 4 abaixo apresenta um código que passou pelo procedimento de análise léxica.

Algoritmo 4 - Fatorial

```
1  int fatorial(int x) {
2          if(x == 1) {
3                  return 1;
4          } else {
5                  return fatorial(x-1)*x;
6          }
7  }
8   int main (void){
9          int x;
10         int z;
11         x = input();
```

```
12          z = fatorial(x);
13      output(z);
14  }
```

O algoritmo acima gerou uma árvore sintática que pode ser vista na Figura 7 abaixo.

Figura 7 – Árvore Sintática

```
TINY COMPILATION: fatorial.txt

Syntax tree:
  Tipo: INT
    Func: fatorial
      Tipo: INT
        Id: x
      If
        Op: ==
          Id: x
          Const: 1
        Return
          Const: 1
        Return
          Op: *
            Chamada da Função: fatorial
              Op: -
                Id: x
                Const: 1
            Id: x
  Tipo: INT
    Func: main
      Tipo: VOID
      Tipo: INT
        Var: x
      Tipo: INT
        Var: z
      Assign:
        Id: x
        Chamada da Função: input
      Assign:
        Id: z
        Chamada da Função: fatorial
          Id: x
      Chamada da Função: output
        Id: z
```

Fonte: O Autor

6.4 Análise Semântica

A última etapa da fase de análise corresponde a análise semântica que tem como objetivo encontrar erros e inconsistências que seriam muito complexos para serem identificados na fase de análise sintática, sendo assim se faz necessário analisar esses possíveis problemas nessa etapa, Além de realizar as devidas análises, essa etapa também é responsável por gerar uma tabela de símbolos que servirá de apoio para a segunda fase de

construção do compilador que é a fase de síntese. De todos os possíveis erros que um código em C- pode apresentar, foram selecionados 7 deles para serem analisados conforme a lista a seguir.

1. Variável não declarada no escopo;
2. Atribuição inválida (Atribuir retorno de uma função void a uma variável);
3. Declaração inválida (Declaração de variável como void);
4. Declaração inválida (Variável já declarada no escopo);
5. Chamada de função não declarada;
6. Função **main** não declarada;
7. Declaração inválida (Nome de variável já utilizado em nome de função).

A análise semântica foi feita de tal forma que recebe os nós da árvore sintática criada na etapa anterior e aplica as regras propostas ao mesmo tempo que monta a tabela de símbolos que é construída por meio de um tabela *Hash* onde ao percorrer a árvore sintática, insere para cada símbolo as seguintes propriedades na tabela:

1. Identificador;
2. Tipo (Se é variável, função ou chamada de função);
3. Escopo onde aquele simbolo se encontra;
4. Tipo do símbolo (Se é int ou void);
5. Linha do código fonte que o símbolo aparece

A Figura 8 apresenta a tabela de símbolos que foi gerada a partir do Algoritmo 4.

Figura 8 – Tabela de Símbolos

```
Building Symbol Table...

Symbol table:

Variable Name  Escopo  Tipo ID  Tipo dado  Line Numbers
-------------  ------  -------  ---------  ------------
main           global  fun      INT        16;
input          main    call     null       12;
output         main    call     null       14;
x              main    var      INT        10;  12;  13;
x              fatorial  var      INT          1;   2;   5;   5;
z              main    var      INT        11;  13;  14;
fatorial       global  fun      INT         7;   5;  13;

Checking Types...

Type Checking Finished
```

Fonte: O Autor

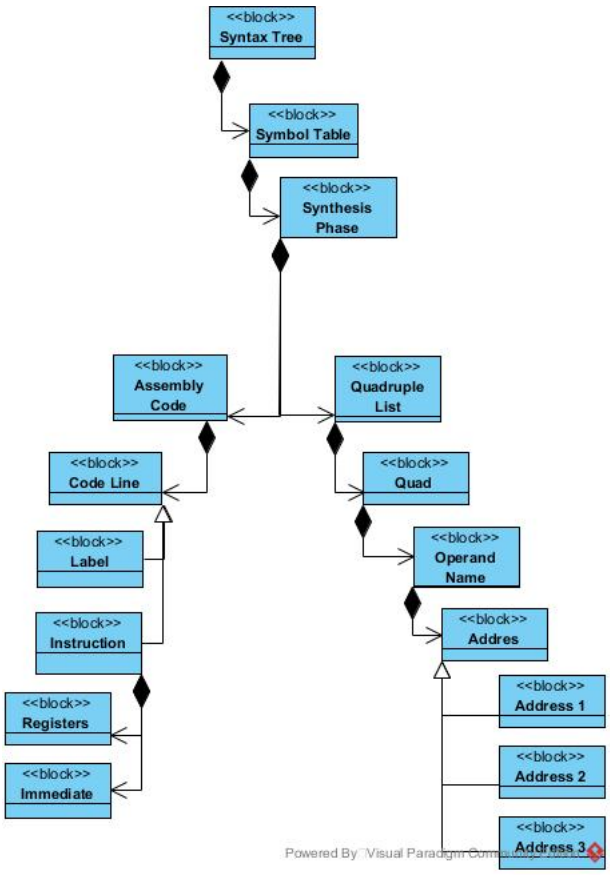
7 Compilador: Fase de Síntese

Este capítulo diz a respeito da fase de síntese modo compilador onde será apresentado o procedimento de elaboração do código *assembly* gerado através da análise de um código fonte bem como o código em binário que diz respeito as instruções que serão executadas no processador AKIRAPROC baseado em MIPS conforme foi apresentado anteriormente.

7.1 Modelagem

7.1.1 Diagrama de Blocos (SysML)

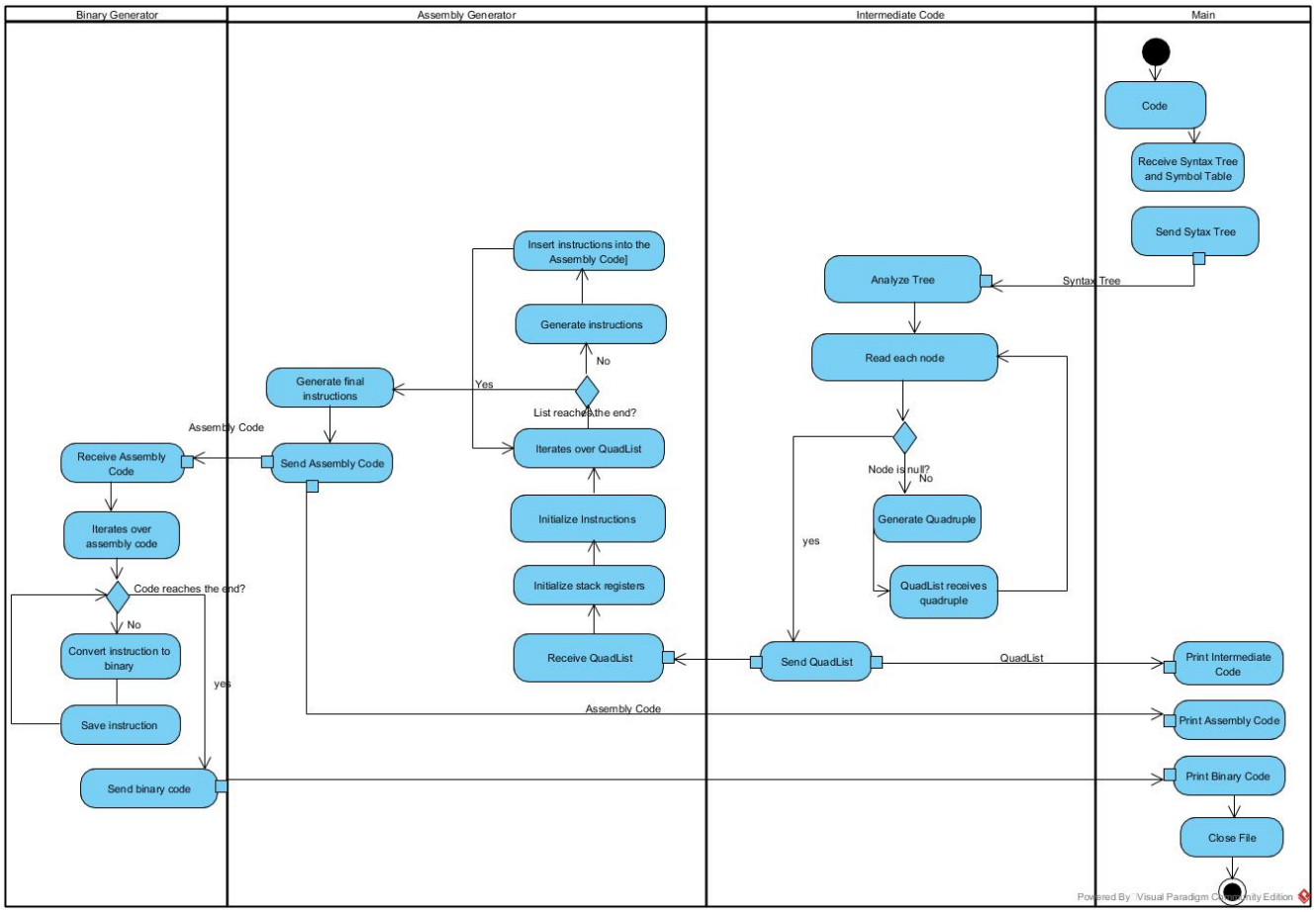
Figura 9 – Diagrama de Blocos da Fase de Síntese



Fonte: O Autor

7.1.2 Diagramas de Atividades (SysML)

Figura 10 – Diagrama de Atividades da Fase de Síntese



Fonte: O Autor

7.2 Geração do código intermediário

A primeira parte da fase de síntese do compilador se inicia a partir da geração do código intermediário, essa etapa se faz necessária para que a execução da geração de ambos código *assembly* e binário possam ser gerados com mais facilidade. Tal etapa recebe como entrada um tipo de estrutura proposta por (3) chamada de quádrupla, contendo três endereços que dão forma a cada uma das instruções. O primeiro campo contém a operação que será realizada e os demais campos contém os três endereços necessários para a realização da operação. Além de seguir uma regra definida de acordo com a tabela acima, cada um dos endereços é definido por:

1. *String* ao utilizar variáveis temporárias ou rótulos;
2. *IntConst* para valores de imediato;

3. *Empty* ao relacionar um endereço vazio;

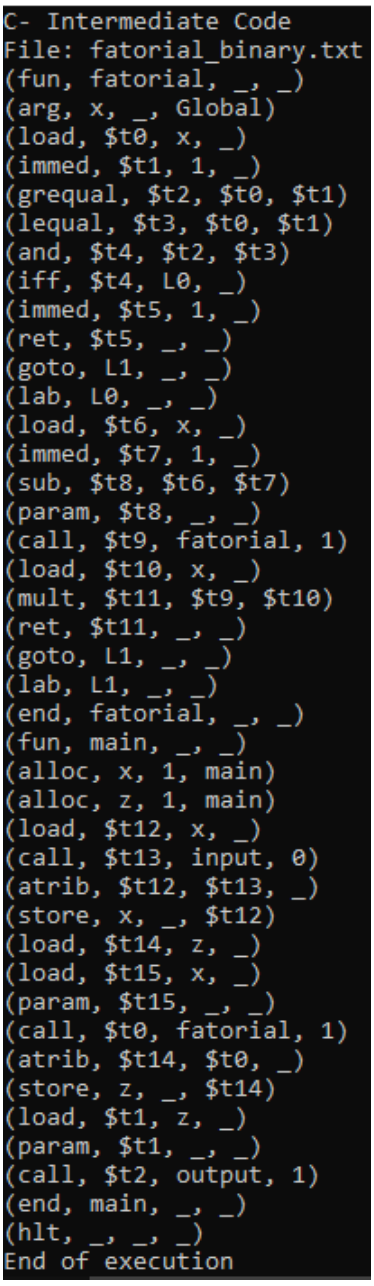
Cada endereço é armazenado em uma estrutura de dados pode ser vista no código a seguir:

```
typedef struct {
  AddrKind kind;
  union {
    int val;
    struct {
      char * name;
      char * scope;
    } var;
  } contents;
} Address;

typedef struct {
  OpKind op;
  Address addr1, addr2, addr3;
} Quad;

typedef struct QuadListRec {
  int location;
  Quad quad;
  struct QuadListRec * next;
} * QuadList;
```

A construção de cada uma das quádruplas, o código recebe a árvore de análise sintática e percorre a mesma analisando cada tipo de nó é inserida uma quadrupla, do código acima pode ser feita a análise da inserção dessa quadrupla, na estrutura Quad se encontram os quatro campos, o primeiro, op é a instrução de fato e os demais campos são os endereços.

Utilizando o mesmo Algoritmo 4 apresentado anteriormente e o submetendo a fase de geração de código intermediário foram geradas suas respectivas quádruplas que pode ser visto na Figura 11 a seguir.

Figura 11 – Quádrupla Gerada



Fonte: O Autor

7.3 Geração do código Assembly

A geração do código em *Assembly* foi elaborada em C e recebe como entrada a tabela de símbolos gerada na etapa de análise semântica e o código intermediário gerado anteriormente. Essa etapa visita cada uma das quádruplas e realiza a tradução de cada uma delas em uma sequência de instruções de acordo com a estrutura do processador AKIRAPROC.

É importante citar que algumas das instruções que foram inseridas nas quádruplas

não estão presentes na estrutura do processador e servem apenas de auxílio, ou seja, não são traduzidas em instruções da arquitetura do processador, como por exemplo, instrução *alloc* que serve apenas para alocar um espaço da memória que uma função irá utilizar e a instrução *call* que apenas referencia a chamada de função.

Em relação aos registradores, como já dito anteriormente é utilizado um banco de registradores onde nele existem 32 registradores que foram divididos em: $zero que armazena apenas um valor zero e o mesmo não é modificado pelo usuário, a vantagem de utilizar ele é que o mesmo pode ser útil para instruções de comparação como por exemplo *branch if equal*; outros 16 registradores são registradores temporários utilizados como suporte para instruções, tais registradores correspondem por $t0 até $t15; 10 registradores foram utilizados para armazenamento de argumentos de funções, estes correspondem aos registradores de $a0 até $a9; Existem três registradores especial que fazem controle de pilha, estes são o $sp (*stack pointer*) que faz o controle da pilha a medida que uma função é iniciada, $gp (*global pointer*) que corresponde ao ponteiro global para variáveis e vetores globais, o último corresponde ao registrador $ra (*return address*) que controla a pilha de endereços de retorno; Por fim há mais dois registradores estes são $ret que armazena os valores de retorno de uma função e $jmp que armazena o endereço de salto de uma instrução.

Para a construção do código em *assembly* é utilizada uma estrutura de lista que apresenta a funcionalidade de armazenar cada uma das linhas de instrução do código. Esta lista armazena o número da linha correspondente, o tipo da instrução (se é uma instrução ou rótulo de alguma rotina) e os respectivos dados necessários para tal instrução. Caso o tipo de instrução seja um rótulo de rotina é armazenado apenas o identificador do mesmo, caso o tipo for de fato uma instrução é utilizada uma estrutura de dados que armazena os campos de *opcode* da instrução, registradores e caso for necessário, um valor de imediato. Essa estrutura de lista pode ser visto no Algoritmo abaixo.

Estrutura Código Executável

```
1  typedef enum {  nop, halt, add, addi, sub, mult, divi, mod, and, or, not, xor, slt, sgt,
2                  sle, sge, shl, shr, move,
3                  ldi, beq, bne, jmp, in, out, str, load, jr } InstrKind;
4  typedef enum {  format1, format2, format3, format4 } InstrFormat;
5  typedef enum {  instr, lbl } LineKind;
6  typedef enum {  simple, vector, address } VarKind;
7  typedef enum {  $zero, $t0, $t1, $t2, $t3, $t4, $t5, $t6, $t7, $t8, $t9, $t10, $t11, $t12
       , $t13, $t14,
8                  $t15, $a0, $a1, $a2, $a3, $a4, $a5, $a6, $a7, $a8, $a9, $sp, $gp, $ra,
                       $ret, $jmp } Reg;
9
10 typedef struct {
11     InstrFormat format;
12     InstrKind opcode;
13     Reg reg1;
14     Reg reg2;
```

```
15      Reg reg3;
16      int im;
17      char * imlbl;
18  } Instruction;
19
20  typedef struct AssemblyCodeRec {
21      int lineno;
22      LineKind kind;
23      union {
24          Instruction instruction;
25          char * label;
26      } line;
27      struct AssemblyCodeRec * next;
28  } * AssemblyCode;
```

Utilizando o algoritmo acima, cada uma das quádruplas são percorridas gerando as devidas instruções e um código na linguagem *assembly* que será traduzido em linguagem de máquina (binário) na próxima etapa da fase de Síntese.

7.4 Geração do código executável

A última etapa da fase de síntese do compilador diz respeito ao processo mais simples tendo a função de traduzir o código em *assembly* gerado na etapa anterior em um código de máquina (binário) de acordo com a arquitetura do processador. Essa etapa recebe como entrada a lista gerada após percorrer todas as quádruplas, percorre cada uma das instruções transcrevendo cada um dos nomes da instruções em *opcodes* e cada um de registradores bem como os valores de imediato para valores em binário. É importante citar que antes de montar a instrução em binário, o código verifica qual é o formato da instrução (R, I ou J) para poder alocar os registradores RS, RT e RD de maneira que respeite o formato de instruções do processador, a partir disso popula a instrução concatenando cada uma das partes através de chamadas de estruturas contendo dicionários que irão traduzir item a item da instrução em valores binários.

7.5 Gerenciamento de memória

Ao falar de Gerenciamento de memória, foi escolhida uma abordagem que simula uma pilha, entretanto a mesma não é explicita em sua implementação.Isso é feito de tal maneira que para cada função que é iniciada aloque dinamicamente um espaço na memória de dados para sua execução. O projeto apresenta três blocos de pilhas, o primeiro bloco é controlado pelo registrador $sp que faz o acesso desse bloco na memória. A medida que uma função é iniciada é necessário reservar um espaço na memória para realizar com exatidão as instruções presentes nesse escopo, assim esse espaço é alocado na memória através da soma da quantidade de memória necessária com o valor presente no registrador $sp, quando a função é encerrada é decrementado do registrador o valor de memória que foi

utilizada e o mesmo de fato libera o espaço utilizado da memória de dados.Essa abordagem facilita na implementação de chamada de funções recursivas sem necessidade de alocar um espaço a mais na memória. Vale citar que esse registrador aponta sempre para o topo dessa "pilha", ou seja, irá desalocar o espaço na memória a partir da última função que foi chamada. O segundo bloco de memória semelhante ao primeiro bloco é controlado por um registrador, nesse caso $gp, este tem por finalidade alocar espaço na memória de dados para variáveis e vetores globais, semelhante ao bloco anterior, este aponta para o topo da "pilha"desalocando o espaço a partir da ultima variável armazenada, ou no caso de um vetor, desaloca a partir da última posição. O terceiro e último bloco é acessado pelo registrador de retorno $ra que a partir do momento que uma chamada de função é realizada, o endereço atual, ou seja, a instrução que está esperando um valor de retorno é armazenado no bloco por intermédio desse registrador. Os três blocos trabalham juntos de modo a fazer com que todas as instruções pudessem ser executadas sem dificuldades no processador.

8 Exemplos

Para averiguar o funcionamento completo do compilador foram elencados três exemplos, cada um deles passa por todas as etapas desde a Fase de Análise até a Fase de Síntese. Entretanto, devido a complexidade de alguns códigos, as etapas de análise sintática, léxica e semântica foram descartadas para apresentação nesse relatório devido a uma montagem muito grande e as mesmas etapas são necessárias para que seja possível a realização da fase de síntese. Sendo assim, serão apresentados os códigos exemplos, a geração do código intermediário, código em *assembly* e o código executável em binário.

8.1 Exemplo 1 - GCD

8.1.1 Código Fonte

Algoritmo Fonte - GCD

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

8.1.2 Código Intermediário

Código Intermediário - GCD

```
1  C- Intermediate Code
2  File: teste1_binary.txt
3  (fun, gcd, _, _)
4  (arg, u, _, Global)
5  (arg, v, _, Global)
6  (load, $t0, v, _)
7  (immed, $t1, 0, _)
8  (grequal, $t2, $t0, $t1)
9  (lequal, $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, _, _)
```

```
(load, $t6, v, _)
(param, $t6, _, _)
(load, $t7, u, _)
(load, $t8, u, _)
(load, $t9, v, _)
(div, $t10, $t8, $t9)
(load, $t11, v, _)
(mult, $t12, $t10, $t11)
(sub, $t13, $t7, $t12)
(param, $t13, _, _)
(call, $t14, gcd, 2)
(ret, $t14, _, _)
(goto, L1, _, _)
(lab, L1, _, _)
(end, gcd, _, _)
(fun, main, _, _)
(alloc, x, 1, main)
(alloc, y, 1, main)
(load, $t15, x, _)
(call, $t0, input, 0)
(atrib, $t15, $t0, _)
(store, x, _, $t15)
(load, $t1, y, _)
(call, $t2, input, 0)
(atrib, $t1, $t2, _)
(store, y, _, $t1)
(load, $t3, x, _)
(param, $t3, _, _)
(load, $t4, y, _)
(param, $t4, _, _)
(call, $t5, gcd, 2)
(param, $t5, _, _)
(call, $t6, output, 1)
(end, main, _, _)
(hlt, _, _, _)
End of execution
```

8.1.3 Código Assembly

Código Assembly - GCD

```
C- Assembly Code
0:      ldi $sp, 0
1:      ldi $gp, 32
2:      ldi $ra, 47
3:      jmp 43
.gcd
4:      str $a0, $sp, 0
5:      str $a1, $sp, 1
6:      load $t0, $sp, 1
7:      ldi $t1, 0
8:      sge $t2, $t0, $t1
9:      sle $t3, $t0, $t1
10:     and $t4, $t2, $t3
11:     beq $t4, $zero, 18
12:     load $t5, $sp, 0
13:     move $ret, $t5
14:     addi $ra, $ra, -1
15:     load $jmp, $ra, 0
```

```
16:     jr $jmp
17:     jmp 40
.L0
18:     load $t6, $sp, 1
19:     move $a0, $t6
20:     load $t7, $sp, 0
21:     load $t8, $sp, 0
22:     load $t9, $sp, 1
23:     divi $t10, $t8, $t9
24:     load $t11, $sp, 1
25:     mult $t12, $t10, $t11
26:     sub $t13, $t7, $t12
27:     move $a1, $t13
28:     addi $sp, $sp, 2
29:     ldi $jmp, 33
30:     str $jmp, $ra, 0
31:     addi $ra, $ra, 1
32:     jmp 4
33:     move $t14, $ret
34:     addi $sp, $sp, -2
35:     move $ret, $t14
36:     addi $ra, $ra, -1
37:     load $jmp, $ra, 0
38:     jr $jmp
39:     jmp 40
.L1
40:     addi $ra, $ra, -1
41:     load $jmp, $ra, 0
42:     jr $jmp
.main
43:     load $t15, $sp, 0
44:     in $t0
45:     move $t15, $t0
46:     str $t15, $sp, 0
47:     load $t1, $sp, 1
48:     in $t2
49:     move $t1, $t2
50:     str $t1, $sp, 1
51:     load $t3, $sp, 0
52:     move $a0, $t3
53:     load $t4, $sp, 1
54:     move $a1, $t4
55:     addi $sp, $sp, 2
56:     ldi $jmp, 60
57:     str $jmp, $ra, 0
58:     addi $ra, $ra, 1
59:     jmp 4
60:     move $t5, $ret
61:     addi $sp, $sp, -2
62:     move $a0, $t5
63:     move $t6, $a0
64:     out $t6
65:     nop
66:     jmp 67
.end
67:     halt
```

8.1.4 Código executável

Código Executável - GCD

```
instrmem[0] = 32'b001100_00000_11011_0000000000000000;// ldi
instrmem[1] = 32'b001100_00000_11100_0000000000100000;// ldi
instrmem[2] = 32'b001100_00000_11101_0000000000101111;// ldi
instrmem[3] = 32'b010000_00000000000000000000101011;// jmp
//gcd
instrmem[4] = 32'b001101_11011_10001_0000000000000000;// str
instrmem[5] = 32'b001101_11011_10010_0000000000000001;// str
instrmem[6] = 32'b001011_11011_00001_0000000000000001;// load
instrmem[7] = 32'b001100_00000_00010_0000000000000000;// ldi
instrmem[8] = 32'b011010_00001_00010_00011_00000000000;// sge
instrmem[9] = 32'b011001_00001_00010_00100_00000000000;// sle
instrmem[10] = 32'b001000_00011_00100_00101_00000000000;// and
instrmem[11] = 32'b001110_00101_00000_0000000000010010;// beq
instrmem[12] = 32'b001011_11011_00110_0000000000000000;// load
instrmem[13] = 32'b010110_00110_00000_11110_00000000000;// move
instrmem[14] = 32'b000010_11101_11101_1111111111111111;// addi
instrmem[15] = 32'b001011_11101_11111_0000000000000000;// load
instrmem[16] = 32'b010001_11111_00000_0000000000000000;// jr
instrmem[17] = 32'b010000_00000000000000000000101000;// jmp
//L0
instrmem[18] = 32'b001011_11011_00111_0000000000000001;// load
instrmem[19] = 32'b010110_00111_00000_10001_00000000000;// move
instrmem[20] = 32'b001011_11011_01000_0000000000000000;// load
instrmem[21] = 32'b001011_11011_01001_0000000000000000;// load
instrmem[22] = 32'b001011_11011_01010_0000000000000001;// load
instrmem[23] = 32'b000101_01001_01010_01011_00000000000;// divi
instrmem[24] = 32'b001011_11011_01100_0000000000000001;// load
instrmem[25] = 32'b000100_01011_01100_01101_00000000000;// mult
instrmem[26] = 32'b000001_01000_01101_01110_00000000000;// sub
instrmem[27] = 32'b010110_01110_00000_10010_00000000000;// move
instrmem[28] = 32'b000010_11011_11011_0000000000000010;// addi
instrmem[29] = 32'b001100_00000_11111_0000000000100001;// ldi
instrmem[30] = 32'b001101_11101_11111_0000000000000000;// str
instrmem[31] = 32'b000010_11101_11101_0000000000000001;// addi
instrmem[32] = 32'b010000_00000000000000000000000100;// jmp
instrmem[33] = 32'b010110_11110_00000_01111_00000000000;// move
instrmem[34] = 32'b000010_11011_11011_1111111111111110;// addi
instrmem[35] = 32'b010110_01111_00000_11110_00000000000;// move
instrmem[36] = 32'b000010_11101_11101_1111111111111111;// addi
instrmem[37] = 32'b001011_11101_11111_0000000000000000;// load
instrmem[38] = 32'b010001_11111_00000_0000000000000000;// jr
instrmem[39] = 32'b010000_00000000000000000000101000;// jmp
//L1
instrmem[40] = 32'b000010_11101_11101_1111111111111111;// addi
instrmem[41] = 32'b001011_11101_11111_0000000000000000;// load
instrmem[42] = 32'b010001_11111_00000_0000000000000000;// jr
//main
instrmem[43] = 32'b001011_11011_10000_0000000000000000;// load
instrmem[44] = 32'b010100_00000_00000_00001_00000000000;// in
instrmem[45] = 32'b010110_00001_00000_10000_00000000000;// move
instrmem[46] = 32'b001101_11011_10000_0000000000000000;// str
instrmem[47] = 32'b001011_11011_00010_0000000000000001;// load
instrmem[48] = 32'b010100_00000_00000_00011_00000000000;// in
instrmem[49] = 32'b010110_00011_00000_00010_00000000000;// move
instrmem[50] = 32'b001101_11011_00010_0000000000000001;// str
```

```
instrmem[51] = 32'b001011_11011_00100_0000000000000000;// load
instrmem[52] = 32'b010110_00100_00000_10001_00000000000;// move
instrmem[53] = 32'b001011_11011_00101_0000000000000001;// load
instrmem[54] = 32'b010110_00101_00000_10010_00000000000;// move
instrmem[55] = 32'b000010_11011_11011_0000000000000010;// addi
instrmem[56] = 32'b001100_00000_11111_0000000000111100;// ldi
instrmem[57] = 32'b001101_11101_11111_0000000000000000;// str
instrmem[58] = 32'b000010_11101_11101_0000000000000001;// addi
instrmem[59] = 32'b010000_00000000000000000000000100;// jmp
instrmem[60] = 32'b010110_11110_00000_00110_00000000000;// move
instrmem[61] = 32'b000010_11011_11011_1111111111111110;// addi
instrmem[62] = 32'b010110_00110_00000_10001_00000000000;// move
instrmem[63] = 32'b010110_10001_00000_00111_00000000000;// move
instrmem[64] = 32'b010101_00111_00000_0000000000000000;// out
instrmem[65] = 32'b010010_00000000000000000000000000;// nop
instrmem[66] = 32'b010000_00000000000000000001000011;// jmp
//end
instrmem[67] = 32'b010011_00000000000000000000000000;// halt
```

8.2 Exemplo 2 - Selection Sort

8.2.1 Código Fonte

Código Fonte - Selection Sort

```
int vet[ 10 ];

int minloc ( int a[], int low, int high )
{       int i; int x; int k;
        k = low;
        x = a[low];
        i = low + 1;
        while (i < high){
                if (a[i] < x){
                        x = a[i];
                        k = i;
                }
                i = i + 1;
        }
        return k;
}

void sort( int a[], int low, int high)
{       int i; int k;
        i = low;
        while (i < high-1){
                int t;
                k = minloc(a,i,high);
                t = a[k];
                a[k] = a[i];
                a[i] = t;
                i = i + 1;
        }
}

void main(void)
{
```

```
33          int i;
34          i = 0;
35          while (i < 10){
36                  vet[i] = input();
37                  i = i + 1;
38          }
39          sort(vet,0,10);
40          i = 0;
41          while (i < 10){
42                  output(vet[i]);
43                  i = i + 1;
44          }
45  }
```

8.2.2 Código Intermediário

Código Intermediário - Selection Sort

```
1   C- Intermediate Code
2   File: sortteste_binary.txt
3   (alloc, vet, 10, Global)
4   (fun, minloc, _, _)
5   (arg, a, _, Global)
6   (arg, low, _, Global)
7   (arg, high, _, Global)
8   (alloc, i, 1, Global)
9   (alloc, x, 1, Global)
10  (alloc, k, 1, Global)
11  (load, $t0, k, _)
12  (load, $t1, low, _)
13  (atrib, $t0, $t1, _)
14  (store, k, _, $t0)
15  (load, $t2, x, _)
16  (load, $t4, low, _)
17  (vec, $t3, a, $t4)
18  (atrib, $t2, $t3, _)
19  (store, x, _, $t2)
20  (load, $t5, i, _)
21  (load, $t6, low, _)
22  (immed, $t7, 1, _)
23  (add, $t8, $t6, $t7)
24  (atrib, $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  (atrib, $t0, $t1, _)
40  (store, x, _, $t0)
41  (load, $t3, k, _)
```

```
(load, $t4, i, _)
(atrib, $t3, $t4, _)
(store, k, _, $t3)
(goto, L2, _, _)
(lab, L1, _, _)
(lab, L2, _, _)
(load, $t5, i, _)
(load, $t6, i, _)
(immed, $t7, 1, _)
(add, $t8, $t6, $t7)
(atrib, $t5, $t8, _)
(store, i, _, $t5)
(goto, L0, _, _)
(lab, L3, _, _)
(load, $t9, k, _)
(ret, $t9, _, _)
(end, minloc, _, _)
(fun, sort, _, _)
(arg, a, _, sort)
(arg, low, _, sort)
(arg, high, _, sort)
(alloc, i, 1, sort)
(alloc, k, 1, sort)
(load, $t10, i, _)
(load, $t11, low, _)
(atrib, $t10, $t11, _)
(store, i, _, $t10)
(lab, L4, _, _)
(load, $t12, i, _)
(load, $t13, high, _)
(immed, $t14, 1, _)
(sub, $t15, $t13, $t14)
(lt, $t0, $t12, $t15)
(iff, $t0, L5, _)
(alloc, t, 1, sort)
(load, $t1, k, _)
(load, $t2, a, _)
(param, $t2, _, _)
(load, $t3, i, _)
(param, $t3, _, _)
(load, $t4, high, _)
(param, $t4, _, _)
(call, $t5, minloc, 3)
(atrib, $t1, $t5, _)
(store, k, _, $t1)
(load, $t6, t, _)
(load, $t8, k, _)
(vec, $t7, a, $t8)
(atrib, $t6, $t7, _)
(store, t, _, $t6)
(load, $t10, k, _)
(vec, $t9, a, $t10)
(load, $t12, i, _)
(vec, $t11, a, $t12)
(atrib, $t9, $t11, _)
(store, a, $t10, $t9)
(load, $t14, i, _)
(vec, $t13, a, $t14)
(load, $t15, t, _)
(atrib, $t13, $t15, _)
```

```
(store, a, $t14, $t13)
(load, $t0, i, _)
(load, $t1, i, _)
(immed, $t2, 1, _)
(add, $t3, $t1, $t2)
(atrib, $t0, $t3, _)
(store, i, _, $t0)
(goto, L4, _, _)
(lab, L5, _, _)
(end, sort, _, _)
(fun, main, _, _)
(alloc, i, 1, main)
(load, $t4, i, _)
(immed, $t5, 0, _)
(atrib, $t4, $t5, _)
(store, i, _, $t4)
(lab, L6, _, _)
(load, $t6, i, _)
(immed, $t7, 10, _)
(lt, $t8, $t6, $t7)
(iff, $t8, L7, _)
(load, $t10, i, _)
(vec, $t9, vet, $t10)
(call, $t11, input, 0)
(atrib, $t9, $t11, _)
(store, vet, $t10, $t9)
(load, $t12, i, _)
(load, $t13, i, _)
(immed, $t14, 1, _)
(add, $t15, $t13, $t14)
(atrib, $t12, $t15, _)
(store, i, _, $t12)
(goto, L6, _, _)
(lab, L7, _, _)
(load, $t0, vet, _)
(param, $t0, _, _)
(immed, $t1, 0, _)
(param, $t1, _, _)
(immed, $t2, 10, _)
(param, $t2, _, _)
(call, $t3, sort, 3)
(load, $t4, i, _)
(immed, $t5, 0, _)
(atrib, $t4, $t5, _)
(store, i, _, $t4)
(lab, L8, _, _)
(load, $t6, i, _)
(immed, $t7, 10, _)
(lt, $t8, $t6, $t7)
(iff, $t8, L9, _)
(load, $t10, i, _)
(vec, $t9, vet, $t10)
(param, $t9, _, _)
(call, $t11, output, 1)
(load, $t12, i, _)
(load, $t13, i, _)
(immed, $t14, 1, _)
(add, $t15, $t13, $t14)
(atrib, $t12, $t15, _)
(store, i, _, $t12)
```

```
(goto, L8, _, _)
(lab, L9, _, _)
(end, main, _, _)
(hlt, _, _, _)
```

8.2.3 Código Assembly

Código Assembly - Selection Sort

```
0:      ldi $sp, 0
1:      ldi $gp, 32
2:      ldi $ra, 47
3:      jmp 125
.minloc
4:      str $a0, $sp, 10
5:      str $a1, $sp, 11
6:      str $a2, $sp, 12
7:      load $t0, $sp, 15
8:      load $t1, $sp, 11
9:      move $t0, $t1
10:     str $t0, $sp, 15
11:     load $t2, $sp, 14
12:     load $t4, $sp, 11
13:     load $t3, $sp, 10
14:     add $t4, $t4, $t3
15:     load $t3, $t4, 0
16:     move $t2, $t3
17:     str $t2, $sp, 14
18:     load $t5, $sp, 13
19:     load $t6, $sp, 11
20:     ldi $t7, 1
21:     add $t8, $t6, $t7
22:     move $t5, $t8
23:     str $t5, $sp, 13
.L0
24:     load $t9, $sp, 13
25:     load $t10, $sp, 12
26:     slt $t11, $t9, $t10
27:     beq $t11, $zero, 54
28:     load $t13, $sp, 13
29:     load $t12, $sp, 10
30:     add $t13, $t13, $t12
31:     load $t12, $t13, 0
32:     load $t14, $sp, 14
33:     slt $t15, $t12, $t14
34:     beq $t15, $zero, 47
35:     load $t0, $sp, 14
36:     load $t2, $sp, 13
37:     load $t1, $sp, 10
38:     add $t2, $t2, $t1
39:     load $t1, $t2, 0
40:     move $t0, $t1
41:     str $t0, $sp, 14
42:     load $t3, $sp, 15
43:     load $t4, $sp, 13
44:     move $t3, $t4
45:     str $t3, $sp, 15
46:     jmp 47
.L1
```

```
.L2
47:     load $t5, $sp, 13
48:     load $t6, $sp, 13
49:     ldi $t7, 1
50:     add $t8, $t6, $t7
51:     move $t5, $t8
52:     str $t5, $sp, 13
53:     jmp 24
.L3
54:     load $t9, $sp, 15
55:     move $ret, $t9
56:     addi $ra, $ra, -1
57:     load $jmp, $ra, 0
58:     jr $jmp
59:     addi $ra, $ra, -1
60:     load $jmp, $ra, 0
61:     jr $jmp
.sort
62:     str $a0, $sp, 0
63:     str $a1, $sp, 1
64:     str $a2, $sp, 2
65:     load $t10, $sp, 3
66:     load $t11, $sp, 1
67:     move $t10, $t11
68:     str $t10, $sp, 3
.L4
69:     load $t12, $sp, 3
70:     load $t13, $sp, 2
71:     ldi $t14, 1
72:     sub $t15, $t13, $t14
73:     slt $t0, $t12, $t15
74:     beq $t0, $zero, 122
75:     load $t1, $sp, 4
76:     load $t2, $sp, 0
77:     move $a0, $t2
78:     load $t3, $sp, 3
79:     move $a1, $t3
80:     load $t4, $sp, 2
81:     move $a2, $t4
82:     addi $sp, $sp, 6
83:     ldi $jmp, 87
84:     str $jmp, $ra, 0
85:     addi $ra, $ra, 1
86:     jmp 4
87:     move $t5, $ret
88:     addi $sp, $sp, -6
89:     move $t1, $t5
90:     str $t1, $sp, 4
91:     load $t6, $sp, 5
92:     load $t8, $sp, 4
93:     load $t7, $sp, 0
94:     add $t8, $t8, $t7
95:     load $t7, $t8, 0
96:     move $t6, $t7
97:     str $t6, $sp, 5
98:     load $t10, $sp, 4
99:     load $t9, $sp, 0
100:    add $t10, $t10, $t9
101:    load $t9, $t10, 0
102:    load $t12, $sp, 3
```

```
111  103:    load $t11, $sp, 0
112  104:    add $t12, $t12, $t11
113  105:    load $t11, $t12, 0
114  106:    move $t9, $t11
115  107:    str $t9, $t10, 0
116  108:    load $t14, $sp, 3
117  109:    load $t13, $sp, 0
118  110:    add $t14, $t14, $t13
119  111:    load $t13, $t14, 0
120  112:    load $t15, $sp, 5
121  113:    move $t13, $t15
122  114:    str $t13, $t14, 0
123  115:    load $t0, $sp, 3
124  116:    load $t1, $sp, 3
125  117:    ldi $t2, 1
126  118:    add $t3, $t1, $t2
127  119:    move $t0, $t3
128  120:    str $t0, $sp, 3
129  121:    jmp 69
130  .L5
131  122:    addi $ra, $ra, -1
132  123:    load $jmp, $ra, 0
133  124:    jr $jmp
134  .main
135  125:    load $t4, $sp, 0
136  126:    ldi $t5, 0
137  127:    move $t4, $t5
138  128:    str $t4, $sp, 0
139  .L6
140  129:    load $t6, $sp, 0
141  130:    ldi $t7, 10
142  131:    slt $t8, $t6, $t7
143  132:    beq $t8, $zero, 146
144  133:    load $t10, $sp, 0
145  134:    add $t10, $t10, $gp
146  135:    load $t9, $t10, 0
147  136:    in $t11
148  137:    move $t9, $t11
149  138:    str $t9, $t10, 0
150  139:    load $t12, $sp, 0
151  140:    load $t13, $sp, 0
152  141:    ldi $t14, 1
153  142:    add $t15, $t13, $t14
154  143:    move $t12, $t15
155  144:    str $t12, $sp, 0
156  145:    jmp 129
157  .L7
158  146:    addi $t0, $gp, 0
159  147:    move $a0, $t0
160  148:    ldi $t1, 0
161  149:    move $a1, $t1
162  150:    ldi $t2, 10
163  151:    move $a2, $t2
164  152:    addi $sp, $sp, 1
165  153:    ldi $jmp, 157
166  154:    str $jmp, $ra, 0
167  155:    addi $ra, $ra, 1
168  156:    jmp 62
169  157:    move $t3, $ret
170  158:    addi $sp, $sp, -1
```

```
159:    load $t4, $sp, 0
160:    ldi $t5, 0
161:    move $t4, $t5
162:    str $t4, $sp, 0
.L8
163:    load $t6, $sp, 0
164:    ldi $t7, 10
165:    slt $t8, $t6, $t7
166:    beq $t8, $zero, 181
167:    load $t10, $sp, 0
168:    add $t10, $t10, $gp
169:    load $t9, $t10, 0
170:    move $a0, $t9
171:    move $t11, $a0
172:    out $t11
173:    nop
174:    load $t12, $sp, 0
175:    load $t13, $sp, 0
176:    ldi $t14, 1
177:    add $t15, $t13, $t14
178:    move $t12, $t15
179:    str $t12, $sp, 0
180:    jmp 163
.L9
181:    jmp 182
.end
182:    halt
```

8.2.4 Código Executável

Código Executável - Selection Sort

```
instrmem[0] = 32'b001100_00000_11011_0000000000000000;// ldi
instrmem[1] = 32'b001100_00000_11100_0000000000100000;// ldi
instrmem[2] = 32'b001100_00000_11101_0000000000101111;// ldi
instrmem[3] = 32'b010000_00000000000000000001111101;// jmp
//minloc
instrmem[4] = 32'b001101_11011_10001_0000000000001010;// str
instrmem[5] = 32'b001101_11011_10010_0000000000001011;// str
instrmem[6] = 32'b001101_11011_10011_0000000000001100;// str
instrmem[7] = 32'b001011_11011_00001_0000000000001111;// load
instrmem[8] = 32'b001011_11011_00010_0000000000001011;// load
instrmem[9] = 32'b010110_00010_00000_00001_00000000000;// move
instrmem[10] = 32'b001101_11011_00001_0000000000001111;// str
instrmem[11] = 32'b001011_11011_00011_0000000000001110;// load
instrmem[12] = 32'b001011_11011_00101_0000000000001011;// load
instrmem[13] = 32'b001011_11011_00100_0000000000001010;// load
instrmem[14] = 32'b000000_00101_00100_00101_00000000000;// add
instrmem[15] = 32'b001011_00101_00100_0000000000000000;// load
instrmem[16] = 32'b010110_00100_00000_00011_00000000000;// move
instrmem[17] = 32'b001101_11011_00011_0000000000001110;// str
instrmem[18] = 32'b001011_11011_00110_0000000000001101;// load
instrmem[19] = 32'b001011_11011_00111_0000000000001011;// load
instrmem[20] = 32'b001100_00000_01000_0000000000000001;// ldi
instrmem[21] = 32'b000000_00111_01000_01001_00000000000;// add
instrmem[22] = 32'b010110_01001_00000_00110_00000000000;// move
instrmem[23] = 32'b001101_11011_00110_0000000000001101;// str
//L0
instrmem[24] = 32'b001011_11011_01010_0000000000001101;// load
```

```
28 instrmem[25] = 32'b001011_11011_01011_0000000000001100;// load
29 instrmem[26] = 32'b000110_01010_01011_01100_00000000000;// slt
30 instrmem[27] = 32'b001110_01100_00000_0000000000110110;// beq
31 instrmem[28] = 32'b001011_11011_01110_0000000000001101;// load
32 instrmem[29] = 32'b001011_11011_01101_0000000000001010;// load
33 instrmem[30] = 32'b000000_01110_01101_01110_00000000000;// add
34 instrmem[31] = 32'b001011_01110_01101_0000000000000000;// load
35 instrmem[32] = 32'b001011_11011_01111_0000000000001110;// load
36 instrmem[33] = 32'b000110_01101_01111_10000_00000000000;// slt
37 instrmem[34] = 32'b001110_10000_00000_0000000000101111;// beq
38 instrmem[35] = 32'b001011_11011_00001_0000000000001110;// load
39 instrmem[36] = 32'b001011_11011_00011_0000000000001101;// load
40 instrmem[37] = 32'b001011_11011_00010_0000000000001010;// load
41 instrmem[38] = 32'b000000_00011_00010_00011_00000000000;// add
42 instrmem[39] = 32'b001011_00011_00010_0000000000000000;// load
43 instrmem[40] = 32'b010110_00010_00000_00001_00000000000;// move
44 instrmem[41] = 32'b001101_11011_00001_0000000000001110;// str
45 instrmem[42] = 32'b001011_11011_00100_0000000000001111;// load
46 instrmem[43] = 32'b001011_11011_00101_0000000000001101;// load
47 instrmem[44] = 32'b010110_00101_00000_00100_00000000000;// move
48 instrmem[45] = 32'b001101_11011_00100_0000000000001111;// str
49 instrmem[46] = 32'b010000_00000000000000000000101111;// jmp
50 //L1
51 //L2
52 instrmem[47] = 32'b001011_11011_00110_0000000000001101;// load
53 instrmem[48] = 32'b001011_11011_00111_0000000000001101;// load
54 instrmem[49] = 32'b001100_00000_01000_0000000000000001;// ldi
55 instrmem[50] = 32'b000000_00111_01000_01001_00000000000;// add
56 instrmem[51] = 32'b010110_01001_00000_00110_00000000000;// move
57 instrmem[52] = 32'b001101_11011_00110_0000000000001101;// str
58 instrmem[53] = 32'b010000_00000000000000000000011000;// jmp
59 //L3
60 instrmem[54] = 32'b001011_11011_01010_0000000000001111;// load
61 instrmem[55] = 32'b010110_01010_00000_11110_00000000000;// move
62 instrmem[56] = 32'b000010_11101_11101_1111111111111111;// addi
63 instrmem[57] = 32'b001011_11101_11111_0000000000000000;// load
64 instrmem[58] = 32'b010001_11111_00000_0000000000000000;// jr
65 instrmem[59] = 32'b000010_11101_11101_1111111111111111;// addi
66 instrmem[60] = 32'b001011_11101_11111_0000000000000000;// load
67 instrmem[61] = 32'b010001_11111_00000_0000000000000000;// jr
68 //sort
69 instrmem[62] = 32'b001101_11011_10001_0000000000000000;// str
70 instrmem[63] = 32'b001101_11011_10010_0000000000000001;// str
71 instrmem[64] = 32'b001101_11011_10011_0000000000000010;// str
72 instrmem[65] = 32'b001011_11011_01011_0000000000000011;// load
73 instrmem[66] = 32'b001011_11011_01100_0000000000000001;// load
74 instrmem[67] = 32'b010110_01100_00000_01011_00000000000;// move
75 instrmem[68] = 32'b001101_11011_01011_0000000000000011;// str
76 //L4
77 instrmem[69] = 32'b001011_11011_01101_0000000000000011;// load
78 instrmem[70] = 32'b001011_11011_01110_0000000000000010;// load
79 instrmem[71] = 32'b001100_00000_01111_0000000000000001;// ldi
80 instrmem[72] = 32'b000001_01110_01111_10000_00000000000;// sub
81 instrmem[73] = 32'b000110_01101_10000_00001_00000000000;// slt
82 instrmem[74] = 32'b001110_00001_00000_0000000001111010;// beq
83 instrmem[75] = 32'b001011_11011_00010_0000000000000100;// load
84 instrmem[76] = 32'b001011_11011_00011_0000000000000000;// load
85 instrmem[77] = 32'b010110_00011_00000_10001_00000000000;// move
86 instrmem[78] = 32'b001011_11011_00100_0000000000000011;// load
87 instrmem[79] = 32'b010110_00100_00000_10010_00000000000;// move
```

```
88  instrmem[80] = 32'b001011_11011_00101_0000000000000010;// load
89  instrmem[81] = 32'b010110_00101_00000_10011_00000000000;// move
90  instrmem[82] = 32'b000010_11011_11011_0000000000000110;// addi
91  instrmem[83] = 32'b001100_00000_11111_0000000001010111;// ldi
92  instrmem[84] = 32'b001101_11101_11111_0000000000000000;// str
93  instrmem[85] = 32'b000010_11101_11101_0000000000000001;// addi
94  instrmem[86] = 32'b010000_00000000000000000000000100;// jmp
95  instrmem[87] = 32'b010110_11110_00000_00110_00000000000;// move
96  instrmem[88] = 32'b000010_11011_11011_1111111111111010;// addi
97  instrmem[89] = 32'b010110_00110_00000_00010_00000000000;// move
98  instrmem[90] = 32'b001101_11011_00010_0000000000000100;// str
99  instrmem[91] = 32'b001011_11011_00111_0000000000000101;// load
100 instrmem[92] = 32'b001011_11011_01001_0000000000000100;// load
101 instrmem[93] = 32'b001011_11011_01000_0000000000000000;// load
102 instrmem[94] = 32'b000000_01001_01000_01001_00000000000;// add
103 instrmem[95] = 32'b001011_01001_01000_0000000000000000;// load
104 instrmem[96] = 32'b010110_01000_00000_00111_00000000000;// move
105 instrmem[97] = 32'b001101_11011_00111_0000000000000101;// str
106 instrmem[98] = 32'b001011_11011_01011_0000000000000100;// load
107 instrmem[99] = 32'b001011_11011_01010_0000000000000000;// load
108 instrmem[100] = 32'b000000_01011_01010_01011_00000000000;// add
109 instrmem[101] = 32'b001011_01011_01010_0000000000000000;// load
110 instrmem[102] = 32'b001011_11011_01101_0000000000000011;// load
111 instrmem[103] = 32'b001011_11011_01100_0000000000000000;// load
112 instrmem[104] = 32'b000000_01101_01100_01101_00000000000;// add
113 instrmem[105] = 32'b001011_01101_01100_0000000000000000;// load
114 instrmem[106] = 32'b010110_01100_00000_01010_00000000000;// move
115 instrmem[107] = 32'b001101_01011_01010_0000000000000000;// str
116 instrmem[108] = 32'b001011_11011_01111_0000000000000011;// load
117 instrmem[109] = 32'b001011_11011_01110_0000000000000000;// load
118 instrmem[110] = 32'b000000_01111_01110_01111_00000000000;// add
119 instrmem[111] = 32'b001011_01111_01110_0000000000000000;// load
120 instrmem[112] = 32'b001011_11011_10000_0000000000000101;// load
121 instrmem[113] = 32'b010110_10000_00000_01110_00000000000;// move
122 instrmem[114] = 32'b001101_01111_01110_0000000000000000;// str
123 instrmem[115] = 32'b001011_11011_00001_0000000000000011;// load
124 instrmem[116] = 32'b001011_11011_00010_0000000000000011;// load
125 instrmem[117] = 32'b001100_00000_00011_0000000000000001;// ldi
126 instrmem[118] = 32'b000000_00010_00011_00100_00000000000;// add
127 instrmem[119] = 32'b010110_00100_00000_00001_00000000000;// move
128 instrmem[120] = 32'b001101_11011_00001_0000000000000011;// str
129 instrmem[121] = 32'b010000_00000000000000000001000101;// jmp
130 //L5
131 instrmem[122] = 32'b000010_11101_11101_1111111111111111;// addi
132 instrmem[123] = 32'b001011_11101_11111_0000000000000000;// load
133 instrmem[124] = 32'b010001_11111_00000_0000000000000000;// jr
134 //main
135 instrmem[125] = 32'b001011_11011_00101_0000000000000000;// load
136 instrmem[126] = 32'b001100_00000_00110_0000000000000000;// ldi
137 instrmem[127] = 32'b010110_00110_00000_00101_00000000000;// move
138 instrmem[128] = 32'b001101_11011_00101_0000000000000000;// str
139 //L6
140 instrmem[129] = 32'b001011_11011_00111_0000000000000000;// load
141 instrmem[130] = 32'b001100_00000_01000_0000000000001010;// ldi
142 instrmem[131] = 32'b000110_00111_01000_01001_00000000000;// slt
143 instrmem[132] = 32'b001110_01001_00000_0000000010010010;// beq
144 instrmem[133] = 32'b001011_11011_01011_0000000000000000;// load
145 instrmem[134] = 32'b000000_01011_11100_01011_00000000000;// add
146 instrmem[135] = 32'b001011_01011_01010_0000000000000000;// load
147 instrmem[136] = 32'b010100_00000_00000_01100_00000000000;// in
```

```
148 instrmem[137] = 32'b010110_01100_00000_01010_00000000000;// move
149 instrmem[138] = 32'b001101_01011_01010_0000000000000000;// str
150 instrmem[139] = 32'b001011_11011_01101_0000000000000000;// load
151 instrmem[140] = 32'b001011_11011_01110_0000000000000000;// load
152 instrmem[141] = 32'b001100_00000_01111_0000000000000001;// ldi
153 instrmem[142] = 32'b000000_01110_01111_10000_00000000000;// add
154 instrmem[143] = 32'b010110_10000_00000_01101_00000000000;// move
155 instrmem[144] = 32'b001101_11011_01101_0000000000000000;// str
156 instrmem[145] = 32'b010000_00000000000000000010000001;// jmp
157 //L7
158 instrmem[146] = 32'b000010_11100_00001_0000000000000000;// addi
159 instrmem[147] = 32'b010110_00001_00000_10001_00000000000;// move
160 instrmem[148] = 32'b001100_00000_00010_0000000000000000;// ldi
161 instrmem[149] = 32'b010110_00010_00000_10010_00000000000;// move
162 instrmem[150] = 32'b001100_00000_00011_0000000000001010;// ldi
163 instrmem[151] = 32'b010110_00011_00000_10011_00000000000;// move
164 instrmem[152] = 32'b000010_11011_11011_0000000000000001;// addi
165 instrmem[153] = 32'b001100_00000_11111_0000000010011101;// ldi
166 instrmem[154] = 32'b001101_11101_11111_0000000000000000;// str
167 instrmem[155] = 32'b000010_11101_11101_0000000000000001;// addi
168 instrmem[156] = 32'b010000_00000000000000000000111110;// jmp
169 instrmem[157] = 32'b010110_11110_00000_00100_00000000000;// move
170 instrmem[158] = 32'b000010_11011_11011_1111111111111111;// addi
171 instrmem[159] = 32'b001011_11011_00101_0000000000000000;// load
172 instrmem[160] = 32'b001100_00000_00110_0000000000000000;// ldi
173 instrmem[161] = 32'b010110_00110_00000_00101_00000000000;// move
174 instrmem[162] = 32'b001101_11011_00101_0000000000000000;// str
175 //L8
176 instrmem[163] = 32'b001011_11011_00111_0000000000000000;// load
177 instrmem[164] = 32'b001100_00000_01000_0000000000001010;// ldi
178 instrmem[165] = 32'b000110_00111_01000_01001_00000000000;// slt
179 instrmem[166] = 32'b001110_01001_00000_0000000010110101;// beq
180 instrmem[167] = 32'b001011_11011_01011_0000000000000000;// load
181 instrmem[168] = 32'b000000_01011_11100_01011_00000000000;// add
182 instrmem[169] = 32'b001011_01011_01010_0000000000000000;// load
183 instrmem[170] = 32'b010110_01010_00000_10001_00000000000;// move
184 instrmem[171] = 32'b010110_10001_00000_01100_00000000000;// move
185 instrmem[172] = 32'b010101_01100_00000_0000000000000000;// out
186 instrmem[173] = 32'b010010_00000000000000000000000000;// nop
187 instrmem[174] = 32'b001011_11011_01101_0000000000000000;// load
188 instrmem[175] = 32'b001011_11011_01110_0000000000000000;// load
189 instrmem[176] = 32'b001100_00000_01111_0000000000000001;// ldi
190 instrmem[177] = 32'b000000_01110_01111_10000_00000000000;// add
191 instrmem[178] = 32'b010110_10000_00000_01101_00000000000;// move
192 instrmem[179] = 32'b001101_11011_01101_0000000000000000;// str
193 instrmem[180] = 32'b010000_00000000000000000010100011;// jmp
194 //L9
195 instrmem[181] = 32'b010000_00000000000000000010110110;// jmp
196 //end
197 instrmem[182] = 32'b010011_00000000000000000000000000;// halt
```

8.3 Exemplo 3 - Fatorial

8.3.1 Código Fonte

Código Fonte - Fatorial

```
int fatorial(int x) {
        if(x == 1) {
                return 1;
        } else {
                return fatorial(x-1)*x;
        }
}
 int main (void){
        int x;
        int z;
        x = input();
        z = fatorial(x);
   output(z);
}
```

8.3.2 Código Intermediário

Código Intermediário - Fatorial

```
(fun, fatorial, _, _)
(arg, x, _, Global)
(load, $t0, x, _)
(immed, $t1, 1, _)
(grequal, $t2, $t0, $t1)
(lequal, $t3, $t0, $t1)
(and, $t4, $t2, $t3)
(iff, $t4, L0, _)
(immed, $t5, 1, _)
(ret, $t5, _, _)
(goto, L1, _, _)
(lab, L0, _, _)
(load, $t6, x, _)
(immed, $t7, 1, _)
(sub, $t8, $t6, $t7)
(param, $t8, _, _)
(call, $t9, fatorial, 1)
(load, $t10, x, _)
(mult, $t11, $t9, $t10)
(ret, $t11, _, _)
(goto, L1, _, _)
(lab, L1, _, _)
(end, fatorial, _, _)
(fun, main, _, _)
(alloc, x, 1, main)
(alloc, z, 1, main)
(load, $t12, x, _)
(call, $t13, input, 0)
(atrib, $t12, $t13, _)
(store, x, _, $t12)
(load, $t14, z, _)
(load, $t15, x, _)
(param, $t15, _, _)
(call, $t0, fatorial, 1)
(atrib, $t14, $t0, _)
(store, z, _, $t14)
(load, $t1, z, _)
(param, $t1, _, _)
(call, $t2, output, 1)
(end, main, _, _)
```

```
41 (hlt, _, _, _)
```

8.3.3 Código Assembly

Código Assembly - Fatorial

```
1  0:      ldi $sp, 0
2  1:      ldi $gp, 32
3  2:      ldi $ra, 47
4  3:      jmp 38
5  .fatorial
6  4:      str $a0, $sp, 0
7  5:      load $t0, $sp, 0
8  6:      ldi $t1, 1
9  7:      sge $t2, $t0, $t1
10 8:      sle $t3, $t0, $t1
11 9:      and $t4, $t2, $t3
12 10:     beq $t4, $zero, 17
13 11:     ldi $t5, 1
14 12:     move $ret, $t5
15 13:     addi $ra, $ra, -1
16 14:     load $jmp, $ra, 0
17 15:     jr $jmp
18 16:     jmp 35
19 .L0
20 17:     load $t6, $sp, 0
21 18:     ldi $t7, 1
22 19:     sub $t8, $t6, $t7
23 20:     move $a0, $t8
24 21:     addi $sp, $sp, 1
25 22:     ldi $jmp, 26
26 23:     str $jmp, $ra, 0
27 24:     addi $ra, $ra, 1
28 25:     jmp 4
29 26:     move $t9, $ret
30 27:     addi $sp, $sp, -1
31 28:     load $t10, $sp, 0
32 29:     mult $t11, $t9, $t10
33 30:     move $ret, $t11
34 31:     addi $ra, $ra, -1
35 32:     load $jmp, $ra, 0
36 33:     jr $jmp
37 34:     jmp 35
38 .L1
39 35:     addi $ra, $ra, -1
40 36:     load $jmp, $ra, 0
41 37:     jr $jmp
42 .main
43 38:     load $t12, $sp, 0
44 39:     in $t13
45 40:     move $t12, $t13
46 41:     str $t12, $sp, 0
47 42:     load $t14, $sp, 1
48 43:     load $t15, $sp, 0
49 44:     move $a0, $t15
50 45:     addi $sp, $sp, 2
51 46:     ldi $jmp, 50
52 47:     str $jmp, $ra, 0
53 48:     addi $ra, $ra, 1
```

```
49:     jmp 4
50:     move $t0, $ret
51:     addi $sp, $sp, -2
52:     move $t14, $t0
53:     str $t14, $sp, 1
54:     load $t1, $sp, 1
55:     move $a0, $t1
56:     move $t2, $a0
57:     out $t2
58:     nop
59:     jmp 60
.end
60:     halt
```

8.3.4 Código Executável

Código Executável - Fatorial

```
instrmem[0] = 32'b001100_00000_11011_0000000000000000;// ldi
instrmem[1] = 32'b001100_00000_11100_0000000000100000;// ldi
instrmem[2] = 32'b001100_00000_11101_0000000000101111;// ldi
instrmem[3] = 32'b010000_00000000000000000000100110;// jmp
//fatorial
instrmem[4] = 32'b001101_11011_10001_0000000000000000;// str
instrmem[5] = 32'b001011_11011_00001_0000000000000000;// load
instrmem[6] = 32'b001100_00000_00010_0000000000000001;// ldi
instrmem[7] = 32'b011010_00001_00010_00011_00000000000;// sge
instrmem[8] = 32'b011001_00001_00010_00100_00000000000;// sle
instrmem[9] = 32'b001000_00011_00100_00101_00000000000;// and
instrmem[10] = 32'b001110_00101_00000_0000000000010001;// beq
instrmem[11] = 32'b001100_00000_00110_0000000000000001;// ldi
instrmem[12] = 32'b010110_00110_00000_11110_00000000000;// move
instrmem[13] = 32'b000010_11101_11101_1111111111111111;// addi
instrmem[14] = 32'b001011_11101_11111_0000000000000000;// load
instrmem[15] = 32'b010001_11111_00000_0000000000000000;// jr
instrmem[16] = 32'b010000_00000000000000000000100011;// jmp
//L0
instrmem[17] = 32'b001011_11011_00111_0000000000000000;// load
instrmem[18] = 32'b001100_00000_01000_0000000000000001;// ldi
instrmem[19] = 32'b000001_00111_01000_01001_00000000000;// sub
instrmem[20] = 32'b010110_01001_00000_10001_00000000000;// move
instrmem[21] = 32'b000010_11011_11011_0000000000000001;// addi
instrmem[22] = 32'b001100_00000_11111_0000000000011010;// ldi
instrmem[23] = 32'b001101_11101_11111_0000000000000000;// str
instrmem[24] = 32'b000010_11101_11101_0000000000000001;// addi
instrmem[25] = 32'b010000_00000000000000000000000100;// jmp
instrmem[26] = 32'b010110_11110_00000_01010_00000000000;// move
instrmem[27] = 32'b000010_11011_11011_1111111111111111;// addi
instrmem[28] = 32'b001011_11011_01011_0000000000000000;// load
instrmem[29] = 32'b000100_01010_01011_01100_00000000000;// mult
instrmem[30] = 32'b010110_01100_00000_11110_00000000000;// move
instrmem[31] = 32'b000010_11101_11101_1111111111111111;// addi
instrmem[32] = 32'b001011_11101_11111_0000000000000000;// load
instrmem[33] = 32'b010001_11111_00000_0000000000000000;// jr
instrmem[34] = 32'b010000_00000000000000000000100011;// jmp
//L1
instrmem[35] = 32'b000010_11101_11101_1111111111111111;// addi
instrmem[36] = 32'b001011_11101_11111_0000000000000000;// load
instrmem[37] = 32'b010001_11111_00000_0000000000000000;// jr
```

```
42 //main
43 instrmem[38] = 32'b001011_11011_01101_0000000000000000;// load
44 instrmem[39] = 32'b010100_00000_00000_01110_00000000000;// in
45 instrmem[40] = 32'b010110_01110_00000_01101_00000000000;// move
46 instrmem[41] = 32'b001101_11011_01101_0000000000000000;// str
47 instrmem[42] = 32'b001011_11011_01111_0000000000000001;// load
48 instrmem[43] = 32'b001011_11011_10000_0000000000000000;// load
49 instrmem[44] = 32'b010110_10000_00000_10001_00000000000;// move
50 instrmem[45] = 32'b000010_11011_11011_0000000000000010;// addi
51 instrmem[46] = 32'b001100_00000_11111_0000000000110010;// ldi
52 instrmem[47] = 32'b001101_11101_11111_0000000000000000;// str
53 instrmem[48] = 32'b000010_11101_11101_0000000000000001;// addi
54 instrmem[49] = 32'b010000_00000000000000000000000100;// jmp
55 instrmem[50] = 32'b010110_11110_00000_00001_00000000000;// move
56 instrmem[51] = 32'b000010_11011_11011_1111111111111110;// addi
57 instrmem[52] = 32'b010110_00001_00000_01111_00000000000;// move
58 instrmem[53] = 32'b001101_11011_01111_0000000000000001;// str
59 instrmem[54] = 32'b001011_11011_00010_0000000000000001;// load
60 instrmem[55] = 32'b010110_00010_00000_10001_00000000000;// move
61 instrmem[56] = 32'b010110_10001_00000_00011_00000000000;// move
62 instrmem[57] = 32'b010101_00011_00000_0000000000000000;// out
63 instrmem[58] = 32'b010010_00000000000000000000000000;// nop
64 instrmem[59] = 32'b010000_00000000000000000000111100;// jmp
65 //end
66 instrmem[60] = 32'b010011_00000000000000000000000000;// halt
```

9 Conclusão

O desenvolvimento do projeto foi de suma importância para o aprendizado em sistemas computacional tanto, instigando o pensamento lógico e fazendo com que o aluno tem uma maior imersão no projeto e construção destes sistemas. Com o projeto foi possível se obter êxito no desenvolvimento de um compilador em C- fazendo com que se tenha um aprofundamento no que diz respeito à compiladores, isso foi alcançado através da construção das fases de Síntese e Análise que trabalhando em conjunto são cruciais para o funcionamento de computadores.

9.1 Dificuldades Encontradas

Sobre dificuldades encontradas uma das principais é a busca por um aprofundamento de carga de teórica relacionado à compiladores na medida que o projeto foi se desenvolvendo. Com relação a fase de síntese a grande dificuldade foi encontrada a partir do desenvolvimento da árvore pois o trabalho de definir as regras da gramática se dá através de um trabalho braçal e suscetível a erros que poderiam impedir não só a construção da árvore mas a tabela de símbolos também. Sobre a fase de síntese, essa foi bem mais trabalho do que a fase de análise devido ao fato de se ter que mexer com muitas estruturas de dados e armazenamento de características de cada um dos tokens. A segunda grande dificuldade nessa fase se dá devido ao fato de construir o código em assembly de forma a respeitar as instruções existentes no processador e ao gerar o código executável se atentar ao fato da posição dos registradores de fonte e destino para cada uma das instruções de acordo com o formato de instrução que a mesma se encaixa.

9.2 Destaques

De destaque as futuras implementações é trabalhar na criação de novas instruções que possam diminuir o tamanho do código em binário gerado fazendo com que se reduza o custo computacional e evite problemas de compilação para as próximas disciplinas de Laboratório de Sistemas Computacionais.

Referências

1 AURELIO, D. do. *Computador*. Disponível em: <https://dicionariodoaurelio.com/computador>. Citado na página 6.

2 PATTERSON, D. A.; HENNESY, J. L. *Computer Organization and Design*. 5th edition. ed. Waltham/MA, EUA: Morgan Kaufmann, 2007. Citado na página 6.

3 KENNETH, C. L. Compiler construction. *Principles and Practice*, 1997. Citado 4 vezes nas páginas 6, 20, 23 e 29.