RA: 120513

Implementação do Compilador C-

São José dos Campos - Brasil

Outubro de 2020

RA: 120513

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.

Dicente: Luiz Felipe Raveduti Zafiro

Docente: Prof. Dr. Luiz Eduardo Galvão Martins

Universidade Federal de São Paulo - UNIFESP

Instituto de Ciência e Tecnologia - Campus São José dos Campos

São José dos Campos - Brasil

Outubro de 2020

Lista de ilustrações

Figura 1 – Diagrama de Bloco. Fonte: Autor. 9
Figura 2 – Diagrama de Atividade - Seleção de Operação. Fonte: Autor. 9
Figura 3 – Diagrama de Caso de Uso - Leitura e Escrita. Fonte: Autor. 10
Figura 4 – Diagrama de Requerimento - ULA. Fonte: Autor. 10
Figura 5 – Diagrama de Requerimento - Memórias. Fonte: Autor. 11
Figura 6 – Datapath referente ao sistema de processamento ZAFx32. Fonte: Autor 12
Figura 7 – Datapath referente á Unidade de Processamento do processador ZAFx32. Fonte: Autor 13
Figura 8 – Diagrama de Blocos - Fase de Análise. Fonte: Autor. 19
Figura 9 – Diagrama de Atividade - Fase de Análise. Fonte: Autor. 20
Figura 10 – Gramática BNF Livre de Contexto para a linguegem C-. Fonte: (1). . . 22
Figura 11 – Tabela de Símbolos e árvore sintática gerada a a partir do código de exemplo GCD. 23
Figura 12 – Lista de tokens gerada para exemplo de entrada. Fonte: Autor. 24
Figura 13 – Árvore sintática gerada para exemplo de entrada. Fonte: Autor. 25
Figura 14 – Tabela de símbolos gerada para exemplo de entrada. Fonte: Autor. . . 25
Figura 15 – Diagrama de Blocos - Fase de Síntese. Fonte: Autor. 27
Figura 16 – Diagrama de Atividade - Fase de Síntese. Fonte: Autor. 28
Figura 17 – Estruturas definidas para a geração de Código Intermediário. Fonte: Autor. 30
Figura 18 – Mapeamento da memória. Fonte: Autor. 36
Figura 19 – Memory Frame. Fonte: Autor. 37

Lista de tabelas

Tabela 1 – Tabela referente aos tipos de instrução utilizados. Fonte: Autor. 16
Tabela 2 – Conjunto de instruções e seus respectivos *OpCodes*. Fonte: Autor. . . . 17
Tabela 3 – Tipos de Quadruplas Suportadas. Fonte: Autor. 31

Sumário

1 INTRODUÇÃO 7

2 O PROCESSADOR 9
2.1 Modelagem 9
2.2 DataPath 12
2.3 Componentes do Processador 13
2.3.1 Memória de Instruções 13
2.3.2 Banco de Registradores 13
2.3.3 Memória de Dados 14
2.3.4 Program Counter (PC) 14
2.3.5 Unidade Lógica e Aritmética (ALU) 14
2.3.6 Extensores de Bits 14
2.3.7 Multiplexadores 15
2.3.8 Unidade de Controle 15
2.3.9 Unidade de Entrada e Saída (IO) 15
2.3.10 Módulo CPU (*Central Processing Unit*) 15
2.3.11 Módulo Principal 15
2.3.12 Módulos Auxiliares 16
2.4 Conjunto de Instruções 16
2.5 Organização da Memória 17

3 COMPILADOR: FASE DE ANÁLISE 19
3.1 Modelagem 19
3.2 Análise Léxica 21
3.3 Análise Sintática 21
3.4 Análise Semântica 22
3.5 Exemplo de Saída da Fase de Análise 23

4 COMPILADOR: FASE DE SÍNTESE 27
4.1 Modelagem 27
4.2 Geração do Código Intermediário 29
4.3 Gerador de Código Assembly 32
4.4 Gerador de Código Executável 34
4.5 Gerenciamento de Memória 35

5 EXEMPLOS 39

5.1 GCD . . . 39
5.1.1 Código em C- . . . 39
5.1.2 Código Intermediário . . . 40
5.1.3 Código Assembly . . . 40
5.1.4 Código Executável . . . 42
5.2 Sort . . . 43
5.2.1 Código em C- . . . 43
5.2.2 Código Intermediário . . . 44
5.2.3 Código Assembly . . . 47
5.2.4 Código Executável . . . 51
5.3 Fatorial . . . 55
5.3.1 Código em C- . . . 55
5.3.2 Código Intermediário . . . 55
5.3.3 Código Assembly . . . 56
5.3.4 Código Executável . . . 57
5.4 Fibonacci . . . 58
5.4.1 Código em C- . . . 58
5.4.2 Código Intermediário . . . 59
5.4.3 Código Assembly . . . 60
5.4.4 Código Executável . . . 62

6 CONCLUSÃO . . . 65

REFERÊNCIAS . . . 67

1 Introdução

Quando pensamos em tecnologia e inovação, logo pensamos em celulares, computadores, relógios inteligentes e afins. Todos esses dispositivos citados são desenvolvidos baseados em sistemas digitais e munidos de grandes quantidades de *Software*.

Todos esses *Softwares* são escritos em uma linguagem de programação, independente de qual seja a aplicação. Segundo (1) e (2), linguagens de programação são notações para descrever computações para pessoas e para máquinas. Os computadores e dispositivos, por sua vez, não conseguem interpretar um código da maneira textual como é redigida pelo ser humano e por isso este deve ser convertido para uma representação que os dispositivos sejam capazes de interpretar e que sem perda de generalidade, execute exatamente o que foi programado pelo programador.

Num geral os processos de compilação são divididos e duas principais fases denominadas fase de análise e síntese. Durante a fase de análise são executados três principais procedimentos: análise léxica, sintática e semântica. Nessa primeira fase também é o momento onde os possíveis erros léxicos, sintáticos e semânticos são apontados. A próxima fase do processo de compilação é a fase de síntese, onde as estruturas geradas a partir do código fonte na fase de análise são convertidas para uma representação intermediária (não necessária), em seguida convertidas para a representação em linguagem de montagem (Assembly) da arquitetura alvo e por fim convertida para sua representação binária.

O projeto em questão se trata da implementação de um compilador completo, suportando assim as fases de análise e síntese, da linguagem C- (cMinus) proposta por (1). A arquitetura alvo da fase de síntese (códigos Assembly e Binário) é a arquiterura de *Soft Processor ZAFx32*, desenvolvida para execução em *chips* de lógica programável, FPGA (*Field Programmable Gate Array*). O projeto do processador em questão foi realizado pelo mesmo autor durante a Unidade Curricular Laboratório de Sistemas Computacionais: Arquitetura e Organização de Computadores.

No capítulo "O Processador"é abordado todas as definições à respeito da arquitetura do processador utilizado (arquitetura ZAFx32). Nele são mostrados o conjunto de instruções, *datapath*, tipos de instruções e explicações sobre todas as unidades presentes. No capítulo "Compilador: Fase de Análise"é abordada a fase de análise do compilador (análise léxica, sintática e semântica) e suas definições. No capítulo "Compilador: Fase de Síntese"é abordada todos os processos referentes ao processo de síntese (geração de código intermediário, assembly e executável) trazendo também as especificações sobre o gerenciamento e mapeamento de memória utilizados. No capítulo "Exemplos"são expostos exemplos de códigos compilados pelo compilador projetado com todas as saídas geradas

por eles. Por fim no capítulo "Conclusão"é apresentada a conclusão do projeto desenvolvido bem como as dificuldades encontradas e perspectivas futuras.

2 O Processador

2.1 Modelagem

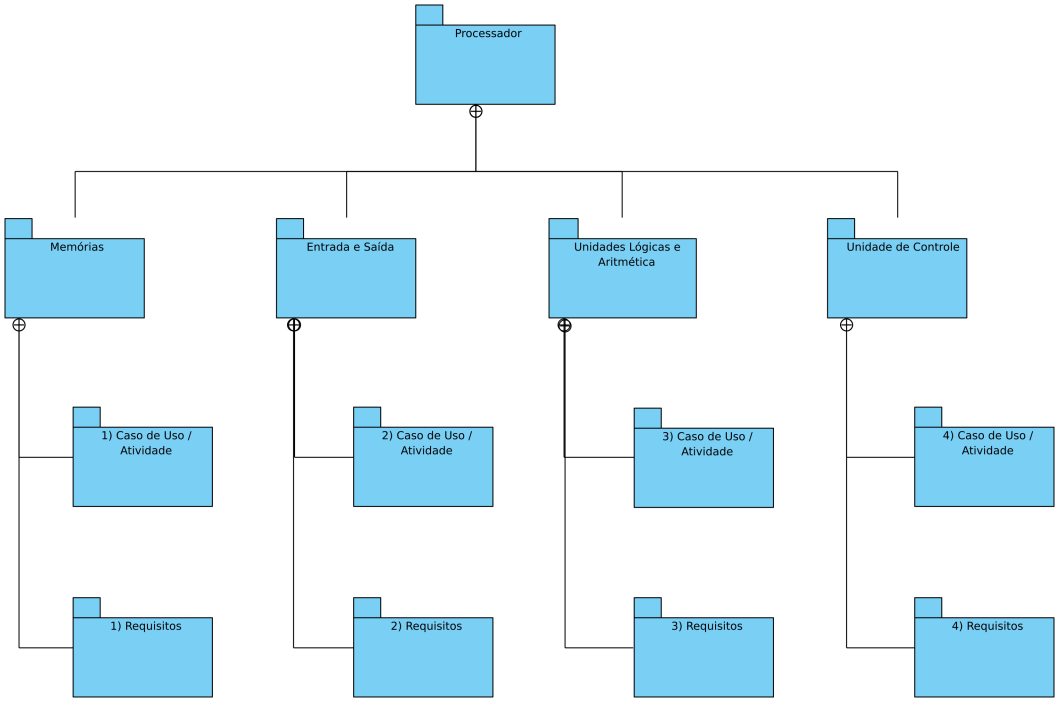

Figura 1 – Diagrama de Bloco. Fonte: Autor.

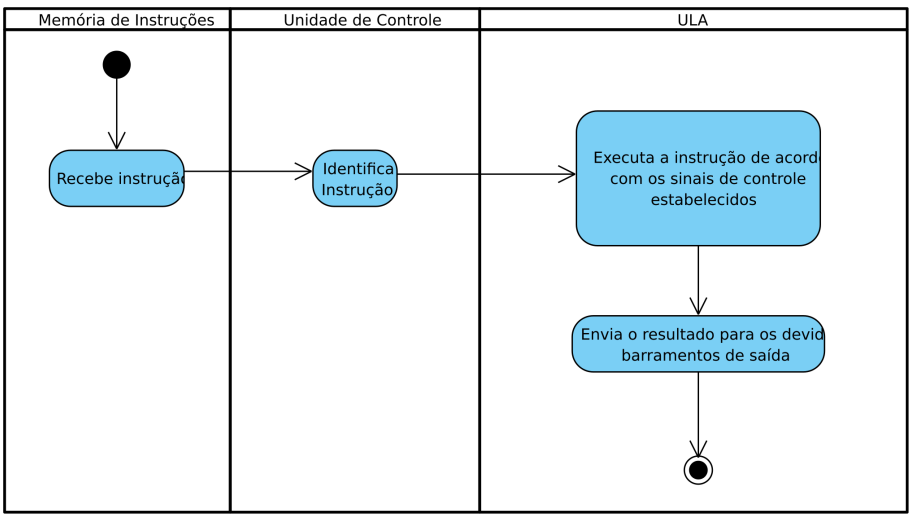

Figura 2 – Diagrama de Atividade - Seleção de Operação. Fonte: Autor.

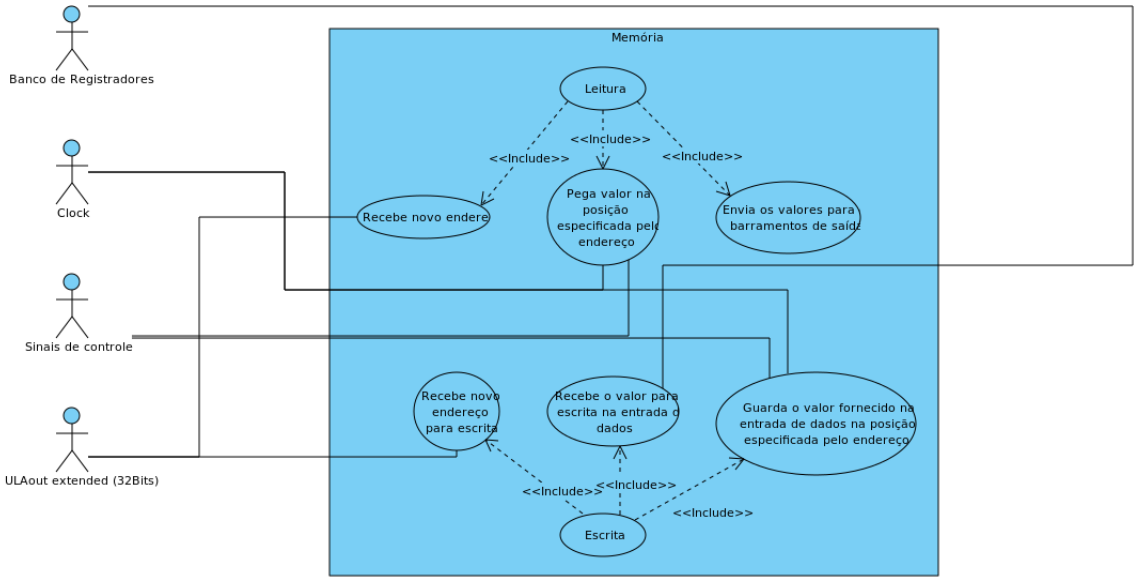

Figura 3 – Diagrama de Caso de Uso - Leitura e Escrita. Fonte: Autor.

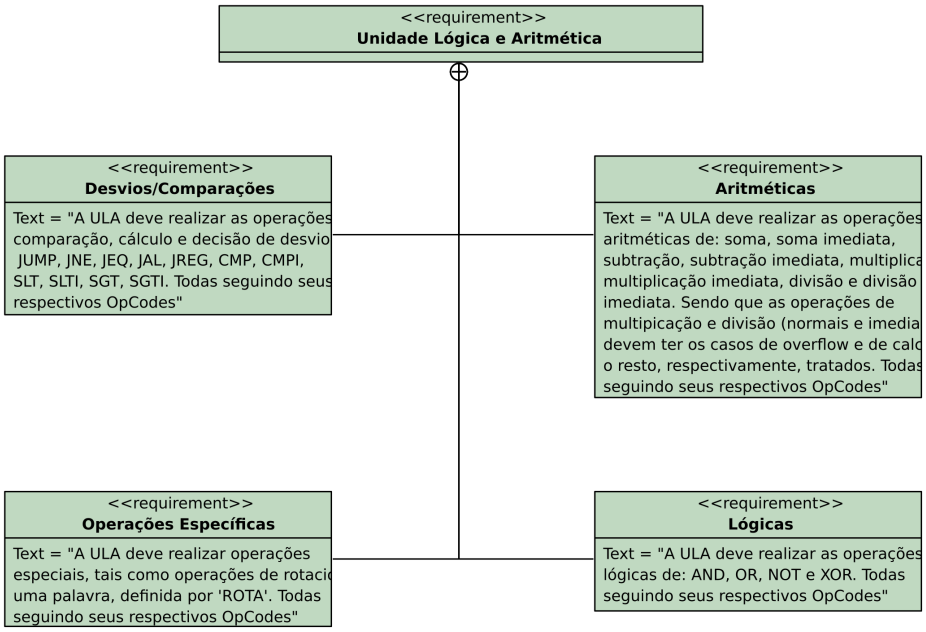

Figura 4 – Diagrama de Requerimento - ULA. Fonte: Autor.

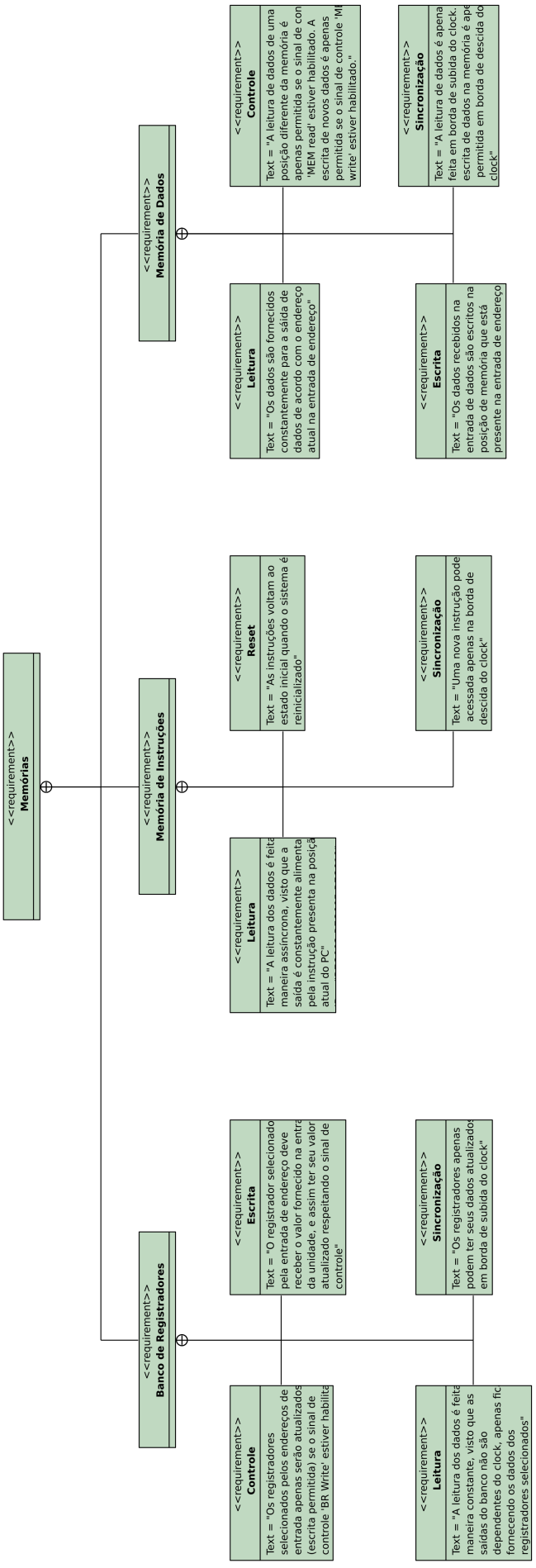

Figura 5 – Diagrama de Requerimento - Memórias. Fonte: Autor.

2.2 DataPath

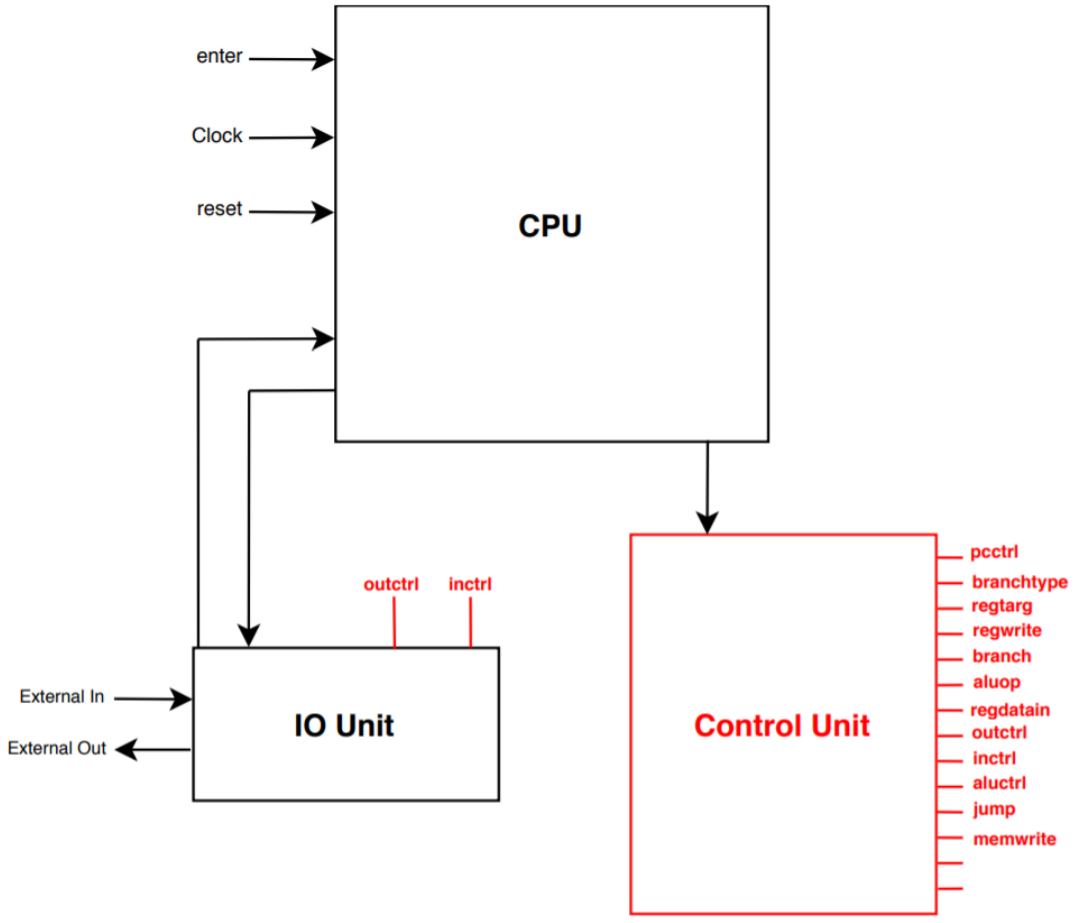

Figura 6 – Datapath referente ao sistema de processamento ZAFx32. Fonte: Autor

.

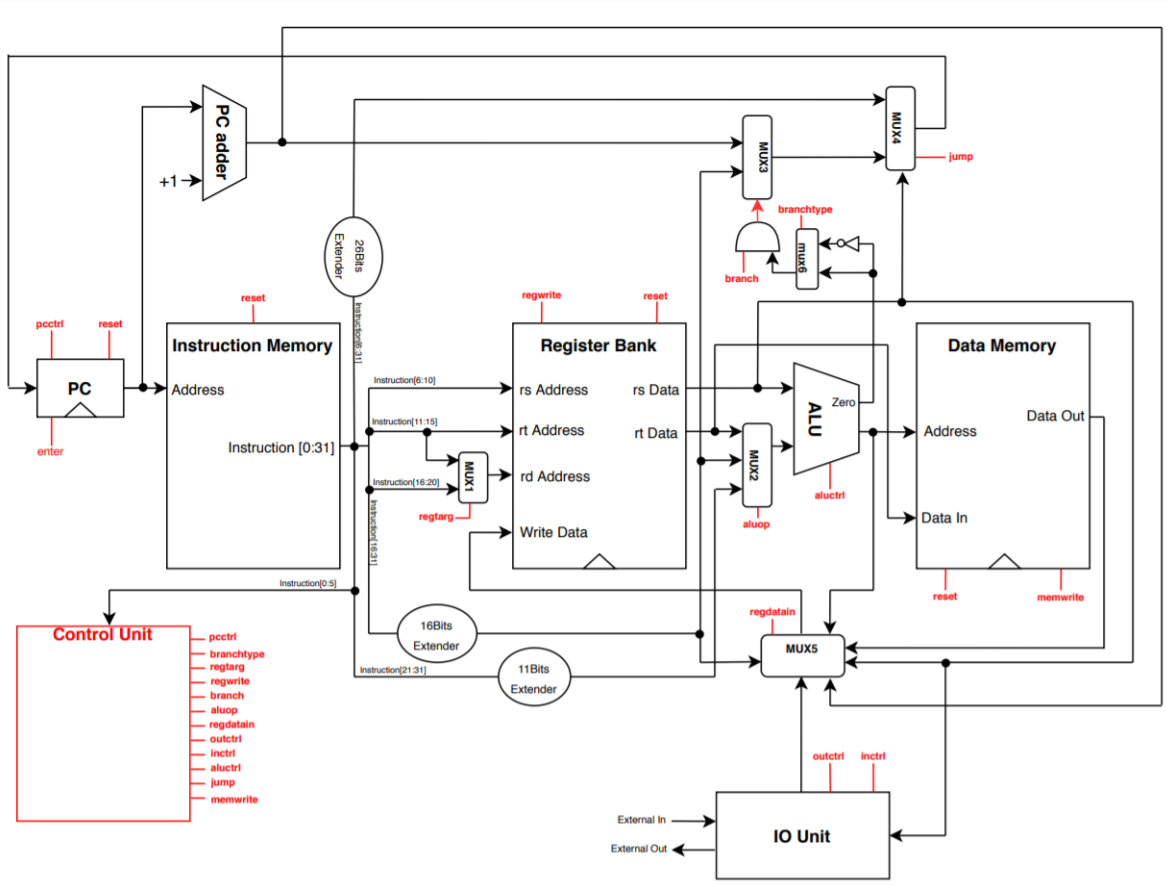

Figura 7 – Datapath referente á Unidade de Processamento do processador ZAFx32. Fonte: Autor

.

2.3 Componentes do Processador

2.3.1 Memória de Instruções

A memória de instruções é uma unidade de armazenamento digital. Nesta implementação ela possui um tamanho definido de 1024x32, ou seja, é capaz de armazenar até 1024 instruções de 32 Bits. As instruções são acessadas por um endereço que é passado como entrada da unidade, e sua saída consiste em uma palavra de 32 Bits referente à instrução do endereço passado. A unidade é independente de um sinal de *clock*.

2.3.2 Banco de Registradores

O banco de registradores é uma unidade de armazenamento digital. Nesta implementação ela é composta por 32 registradores de 32 Bits cada. As funcionalidades da unidade se resumem em: fornecer os dados armazenados no registradores localizados nos endereços de entrada *rs, rt*, e armazenar um valor de 32 Bits passado para a unidade no registrador de endereço *rd*. Uma diferença dessa unidade de armazenamento é que ela é

dependente de um sinal de *clock*, pois as escritas e leituras são feitas em diferentes bordas, para que se evite conflitos.

2.3.3 Memória de Dados

A memória de dados é uma unidade de armazenamento digital. Nesta implementação ela possui um tamanho definido de 1024x32, ou seja, é capaz de armazenar 1024 dados de 32 Bits. As funcionalidades presentes na unidade se resumem em: escrever um dado valor de 32 Bits no espaço de endereço passado como entrada e realizar a leitura de um dado dado um endereço de entrada. Essa unidade é dependente do sinal de *clock* para que os dados sejam acessados em diferentes bordas e assim evite conflitos.

2.3.4 Program Counter (PC)

O *Program Counter* ou PC é um registrador que armazena o endereço da instrução que está sendo acessada na Memória de Instruções. É ele quem coordena a execução do programa carregado na memória. Nesta implementação ele é composto por um registrador de 32 Bits, dependente do clock do sistema e com a funcionalidade de resetar seu valor para zero ou destravá-lo, como por exemplo é feito nas instruções de *halt*.

2.3.5 Unidade Lógica e Aritmética (ALU)

A Unidade Lógica e Aritmética (ALU) é composta por um circuito combinacional capaz de realizar todas as operações matemáticas necessárias para o funcionamento do processador. Sua funcionalidade se resume em receber dois valores de 32 Bits nas entradas de dados e realizar a operação especificada pelo sinal de controle passado para ela. Em sua saída tem-se o resultado da operação e existe ainda uma outra saída chamada 'zero'. A saída 'zero' possui valor lógico constante igual a 0, caso o resultado da operação executada pela ALU seja 0, a saída 'zero' tem seu valor invertido para 1. Esse procedimento com a saída 'zero' é utilizado para instruções de tomada de decisão para desvios, por exemplo.

2.3.6 Extensores de Bits

Os extensores estão presentes com três diferentes tamanhos de entrada, extensor de 11 Bits, 16 Bits e 26 Bits. Sua funcionalidade consistem em basicamente identificar o bit mais significativo do valor passado como entrada e replicá-lo para todos os Bits faltantes até que se complete 32 Bits. Sua importância se dá pelo fato de que todos os valores identificáveis pela arquitetura são de 32 Bits, um valor com menos Bits poderia causar erros.

2.3.7 Multiplexadores

Os multiplexadores são unidade combinacionais capazes de realizar o controle de qual sinal dos que recebe como entrada será propagado em sua saída. Tal controle é feito pelo sinal de controle que ele recebe. Sus funcionalidade é indispensável para a coordenação e controle dos sinais dentro do sistema. Os multiplexadores estão presentes em diversos tamanhos no processador, desde unidades com 2 sinais de entrada até sinais com 6 sinais de entrada.

2.3.8 Unidade de Controle

A Unidade de Controle é uma unidade combinacional. Sua funcionalidade é de controlar todo o fluxo de dados e ativação de unidades de maneira correta para que a instrução em execução seja executada de maneira correta. Ela possui como entrada o valor dos 6 bits mais significativos da instrução atual (diretamente da Memória de Instruções) denominados OpCodes, e a partir disso gera como saída 12 sinais de controle.

2.3.9 Unidade de Entrada e Saída (IO)

A Unidade de Entrada e Saída (IO) é a unidade responsável pela comunicação do sistema com dispositivos de entrada e saída. É utilizada pelas instruções *in, out*, capazes de enviar valores externos ao sistema e retornar valores do sistema para o meio externo, respectivamente. Um exemplo de conexão realizada é de receber valores das chaves do Kit FPGA DE2-115 (utilizado para o projeto do processador) e enviar os valores para os codificadores de *display* de sete segmentos que serão exibidos pelos *displays* na placa.

2.3.10 Módulo CPU (*Central Processing Unit*)

O módulo CPU é responsável por realizar toda a interconexão das unidades exceto as Unidades de Controle e de Entrada e Saída. Capaz de receber e enviar dados para a unidade de Entrada e Saída, enviar os OpCodes das instruções para a unidade de Controle e receber dela todos os correspondentes sinais de controle. Ainda possui como entrada um sinal de *reset*, permitindo que volte a posição do *Program Counter* para a posição 0, um sinal de *enter* para liberar a atualização do *Program Counter* e segui o fluxo do programa e um sinal de *clock* do sistema.

2.3.11 Módulo Principal

O módulo principal consiste em reunir todos os módulos que compõe o funcionamento do processador. Em linhas gerais ele irá "unir"o módulo CPU, a unidade de Entrada e Saída e a unidade de Controle.

2.3.12 Módulos Auxiliares

Os módulos auxiliares são responsáveis por permitir a conexão do Módulo Principal com o ambiente físico em questão. No caso deste projeto, os módulos auxiliares são responsáveis pela conexão com o Kit de Desenvolvimento FPGA DE2-115 da Altera. As principais funcionalidades apresentadas pelos módulos auxiliares são de:

- Divisor de frequência: Capaz de reduzir a frequência de 50 MHz fornecida pela placa para uma frequência de 20 MHz que é utilizada pelo processador;
- Debounce: Capaz de tratar o efeito de *bounce* gerados pelos contatos físicos de botões e chaves presentes na placa;
- Codificador Display de Sete Segmentos: Capaz de representar um valor binário de 32 Bits em sua representação para *display* de sete segmentos, em nosso caso para 4 *displays*;

2.4 Conjunto de Instruções

O Conjunto de Instruções bem como os modos de endereçamento e os tipos de instruções são de suma importância para o projeto em questão, sendo eles:

- Instruções do Tipo R: possuem um campo de 6 Bits referente ao OpCode da instrução, três campos de 5 bits cada para os operandos e 11 bits de *Shift Amount* utilzado para as instruções de *Left Shift e Right Shift*;
- Instruções do Tipo I: possuem um campo de 6 Bits referente ao OpCode da instrução, dois campos de 5 bits cada para os operandos e um campo de 16 bits utilizado para armazenar um imediato, utilizado em instruções que utilizam esse tipo de dado;
- Instruções do Tipo J: possuem um campo de 6 Bits referente ao OpCode da instrução, e um campo de 26 bits que armazenará um valor de desvio para instruções de desvio incondicional, tais como a *j* e a *jal* (*Jump and link*);

Tipo R	OpCode(6Bits)	rs(5Bits)	rt(5Bits)	rd(5Bits)	Shamt(11Bits)
Tipo I	OpCode(6Bits)	rs(5Bits)	rt(5Bits)	Imediato(16Bits)	
Tipo J	OpCode(6Bits)	Desvio(26Bits)			

Tabela 1 – Tabela referente aos tipos de instrução utilizados. Fonte: Autor.

Conjunto de instruções, ou seja, o conjunto de operações que nosso processador, inicialmente (pois futuras alterações poderão surgir de acordo com a necessidade), irá ser capaz de processar.

Instruções (32 Bits) - Referências	Tipo da instrução	OpCode(6 Bits)
add - Add	A	0 0 0 0 0 0
addi - Add Immidiate	C	0 0 0 0 0 1
SUB - Subtract	A	0 0 0 0 1 0
mul - Multiply	C	0 0 0 0 1 1
div - Divide	A	0 0 0 1 0 0
mod - Remaining of Division	C	0 0 0 1 0 1
and - And	A	0 0 0 1 1 0
or - Or	C	0 0 0 1 1 1
xor - Exclusive Or	A	0 0 1 0 0 0
not - Not	A	0 0 1 0 0 1
slt - Set Less Then	A	0 0 1 0 1 0
sgt - Set Greater Then	A	0 0 1 0 1 1
slet - Set Less and Equal Then	C	0 0 1 1 0 0
sget - Set Greather and Equal Then	B	0 0 1 1 0 1
lsh - Left Shift	B	0 0 1 1 1 0
rsh - Right Shift	B	0 0 1 1 1 1
mov - Move	D	0 1 0 0 0 0
li - Load Immidiate	B	0 1 0 0 0 1
beq - Branch if Equal	B	0 1 0 0 1 0
bne - Branch if Not Equal	B	0 1 0 0 1 1
j - Jump	B	0 1 0 1 0 0
in - Input	B	0 1 0 1 0 1
out - Output	B	0 1 0 1 1 0
load - Load	B	0 1 0 1 1 1
store - Store	A	0 1 1 0 0 0
jr - Jump Register	B	0 1 1 0 0 1
jal - Jamp and Link	A	0 1 1 0 1 0
halt - Halt	B	0 1 1 0 1 1
eq - Equal	D	0 1 1 1 0 0
neq - Not Equal	D	0 1 1 1 0 1

Tabela 2 – Conjunto de instruções e seus respectivos *OpCodes*. Fonte: Autor.

2.5 Organização da Memória

Como citado na seção dos componentes do processador, temos ao todo duas unidades de armazenamento principais no sistema, Memória de Instruções e a Memória de Dados. A Memória de Instruções é organizada como uma matriz bidimensional de tamanho 1024x32, sendo assim é capaz de armazenar até 1024 instruções de 32 Bits cada. Seu endereçamento é feito de maneira direta, onde cada posição da memória é acessada pelo valor passado de entrada para esta unidade. A Memória de Dados também foi implementada como uma matriz bidimensional de tamanho 1024x32, fazendo com que seja possível o armazenamento de 1024 dados de 32 Bits cada.

Vale a ressalva de que os tamanhos das memórias podem ser alteradas a partir do código das mesmas em *Verilog*, isso pode ser feito de acordo com necessidades futuras.

Alguns detalhes dobre a forma que a Memória de Dados é organizada para que fosse possível o suporte de Recursão no processador serão abordados mais à frente.

3 Compilador: Fase de Análise

A primeira fase do processo de compilação é denominada fase de análise a qual é subdividida em três principais etapas: Análise Léxica, Análise Sintática e Análise Semântica. Nessa fase ocorre o processamento do texto de entrada (código fonte), verificação de erros e geração de estruturas referentes ao texto, estruturas essas que são utilizadas ao longo do processo todo de compilação.

3.1 Modelagem

Para facilitar a compreensão da fase de análise, foram criados dois diagramas *UML*, um diagrama de blocos e um de atividades. No diagrama de blocos são especificadas todas as estruturas presentes na implementação e suas relações. No diagrama de atividades são mostradas as relações a nível de execução dessas estruturas.

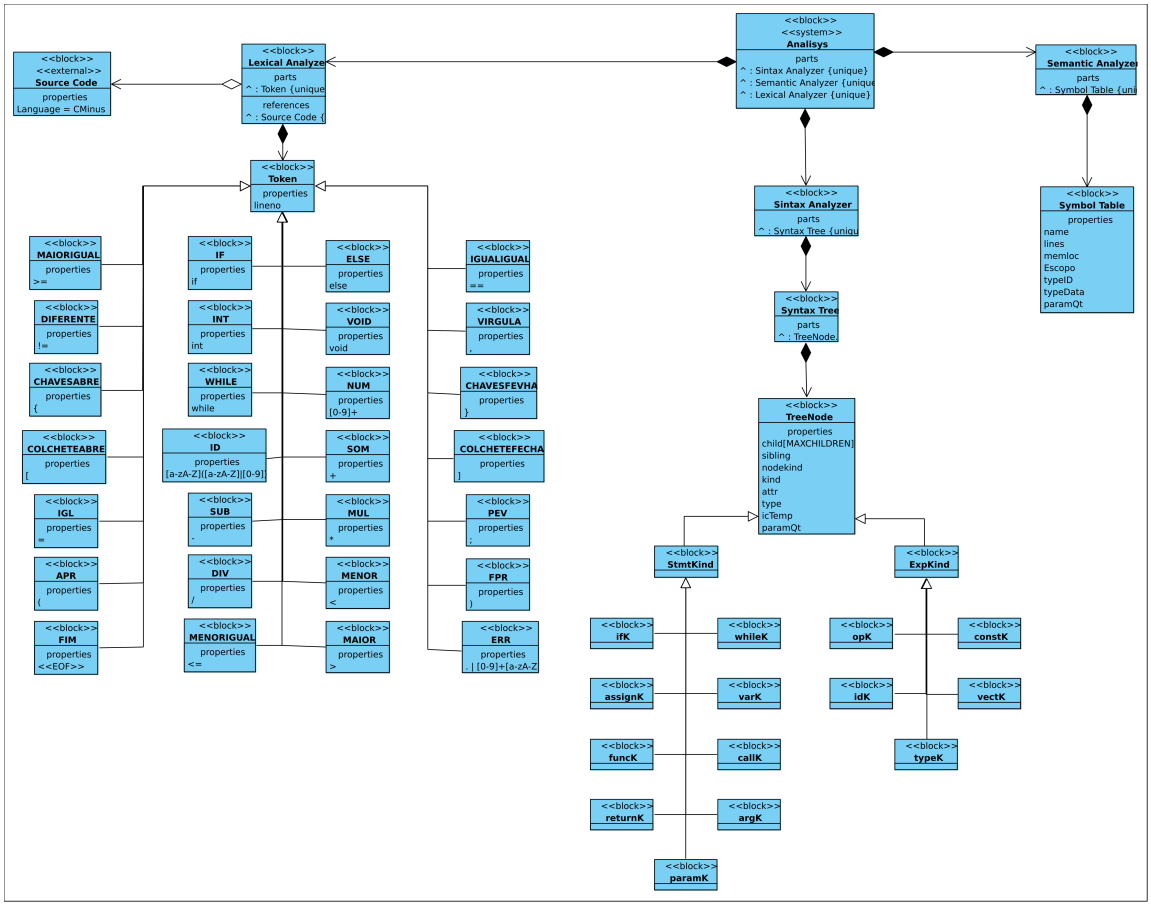

Figura 8 – Diagrama de Blocos - Fase de Análise. Fonte: Autor.

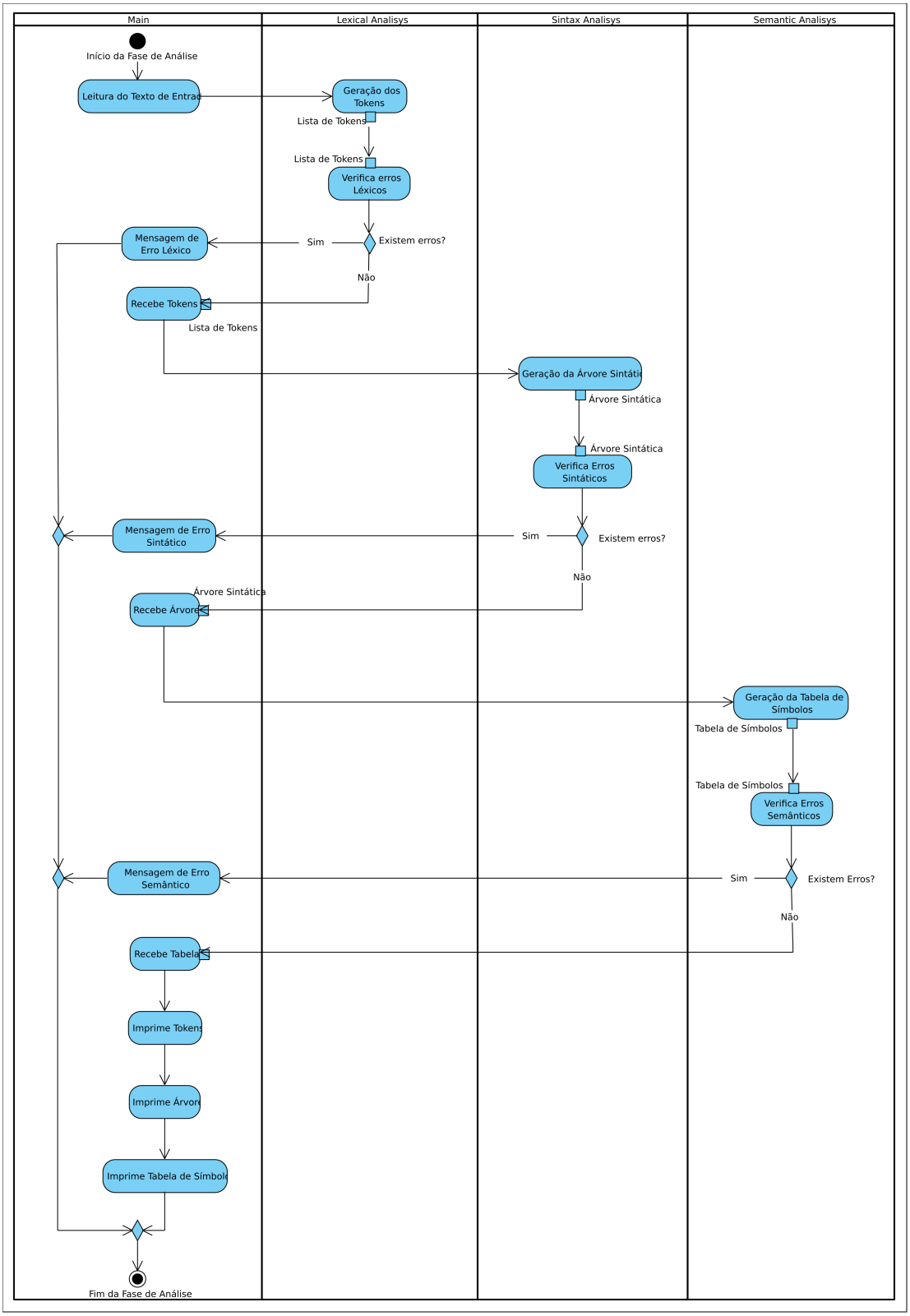

Figura 9 – Diagrama de Atividade - Fase de Análise. Fonte: Autor.

3.2 Análise Léxica

A fase de análise léxica é responsável pela identificação e classificação dos tokens do texto de entrada (código fonte). Esse processo é feito por meio de expressões regulares, as quais são definidas para identificar palavras reservadas, identificadores, valores numéricos e cometários. Durante a fase de análise léxica há a criação de uma estrutura de dados em forma de lista, que armazena os tokens gerados e a posição (linha do código) onde apareceu.

Os comentários dentro da especificação do projeto, para o C- (cMinus) é realizado pelos caracteres "/*"para iniciar um comentário, e "*/"para finalizar o comentário, sendo assim, tudo que estiver presente entre esses dois conjunto de elementos será considerado como comentário.

Seguindo as convenções propostas pelo C- (cMinus), temos que as expressões regulares e palavras reservadas da linguagem são as seguintes.

- Palavras chave (reservadas): **else, if, int, void, while, return**
- \+ - * / < <= > >= == != = ; , ( ) [ ] /* */
- **ID = letra letra***
- **NUM = digito digito***
- **letra = a|b|...|z|A|B|...|Z**
- **digito = 0|1...|9**

Para o projeto, o analisador léxico foi desenvolvido baseado no gerador de analisador léxico *flex*, da *GNU*. Estruturas auxiliares foram utilizadas para garantir o funcionamento e estruturação correta do analisador.

3.3 Análise Sintática

O processo de análise sintática consiste na validação da estrutura em que os tokens estão organizados dentro do código fonte, este procedimento é realizado baseado na gramática definida para a linguagem. Durante o processo de análise sintática que são avaliados os erros sintáticos e a construção da árvore sintática.

No caso do compilador desenvolvido, a gramática é do tipo BNF e livre de contexto, onde suas regras são definidas de acordo com a figura 10.

```
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
```

Figura 10 – Gramática BNF Livre de Contexto para a linguegem C-. Fonte: (1).

Para o projeto, o analisador sintático foi desenvolvido baseado no gerador de parser *bison*, da *GNU*. Estruturas auxiliares foram utilizadas para garantir o funcionamento e estruturação correta do analisador, como por exemplo a estrutura da árvore sintática gerada no processo.

3.4 Análise Semântica

O processo de análise semântica se mostra responsável por avaliar se as estruturas e operações definidas no código fonte fazem sentido dentro da gramática especificada.

Realiza a verificação de tipos de variáveis declaradas, verifica se uma variável foi declarada no escopo antes de uso, se os tipos de retorno de função condizem com os tipos retornados, verificam se funções do tipo *void* não possuem retorno com valor, expressões de "else"sem uma expressão de "if"a antecedendo, se funções foram declaradas antes de serem chamadas, se há dupla declaração de funções, declaração obrigatória da função

main, conflitos em declaração de nomos de variáveis e nomes de funções, declaração de variável do tipo *void*

Durante o processo de análise semântica ocorre a verificação e indicação de erros semânticos e a construção da tabela de símbolos. A tabela de símbolos é uma estrutura de dados, tabela *Hash*, que armazena informações importantes sobre os tokens apresentados, como mostrado na figura a seguir para o exemplo do cógigo *GCD*, em C-.

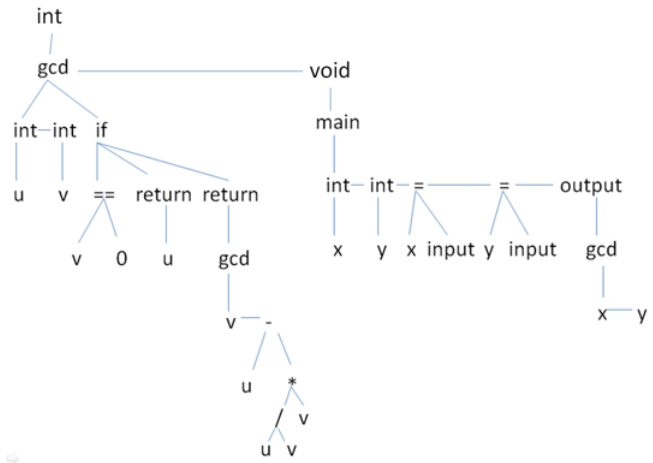

Tabela de Símbolos

Programa de entrada:

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

Entrada	Nome ID	Escopo	Tipo ID	Tipo dado	n. Linha
...					
12	u	gcd	var	int	1,2,3
...					
136	main		fun	void	5
...					
200	gcd		fun	int	1,3,8

Figura 11 – Tabela de Símbolos e árvore sintática gerada a a partir do código de exemplo GCD.

3.5 Exemplo de Saída da Fase de Análise

Uma brave exemplificação do funcionamento dos códigos desenvolvidos para a fase de análise. A seguir o código utilizado como exemplo seguido da sua lista de tokens, árvore sintática e tabela de símbolos.

```
1 /* C digo de exemplo da fase de an lise */
2
3 void main(void){
4     int a;
5     int b;
6     int c;
7
```

```
8       a = input();
9       b = input();
10      c = (a + b) * (a - b);
11      output(c);
12  }
```

Listing 3.1 – Código Exemplo, Fonte: Autor.

```
3: Palavra reservada: VOID, Lexema: void
3: ID, name= main
3: (
3: Palavra reservada: VOID, Lexema: void
3: )
3: {
4: Palavra reservada: INT, Lexema: int
4: ID, name= a
4: ;
5: Palavra reservada: INT, Lexema: int
5: ID, name= b
5: ;
6: Palavra reservada: INT, Lexema: int
6: ID, name= c
6: ;
8: ID, name= a
8: =
8: ID, name= input
8: (
8: )
8: ;
9: ID, name= b
9: =
9: ID, name= input
9: (
9: )
9: ;
10: ID, name= c
10: =
10: (
10: ID, name= a
10: +
10: ID, name= b
10: )
10: *
10: (
10: ID, name= a
10: -
10: ID, name= b
10: )
10: ;
11: ID, name= output
11: (
11: ID, name= c
11: )
11: ;
12: }
```

Figura 12 – Lista de tokens gerada para exemplo de entrada. Fonte: Autor.

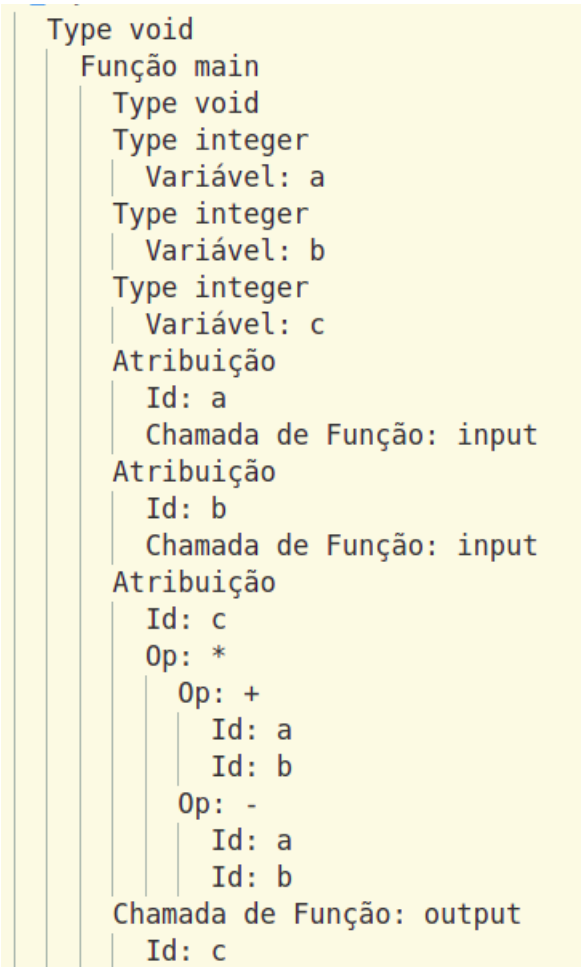

Figura 13 – Árvore sintática gerada para exemplo de entrada. Fonte: Autor.

Location	Name	Escopo	TypeID	TypeData	Param Numb.	Line Numbers
1	a	main	variable	integer	0	4 8 10 10
0	main	global	function	void	1	3
3	c	main	variable	integer	0	6 10 11
2	b	main	variable	integer	0	5 9 10 10

Figura 14 – Tabela de símbolos gerada para exemplo de entrada. Fonte: Autor.

4 Compilador: Fase de Síntese

A fase de síntese do compilador é responsável por traduzir as estruturas criadas e já processadas pela fase de análise e sintetizá-las para um código executável (código de máquina). Para tornar o processo mais simples, duas conversões prévias, antes do código executável, são geradas, a representação em código Intermediário e a representação em código Assembly para a máquina alvo.

Vale a ressalva de que a fase de geração de código intermediário não é obrigatória mas facilita a etapa de geração de código Assembly.

4.1 Modelagem

Para facilitar a compreensão da fase de síntese, foram criados dois diagramas *UML*, um diagrama de blocos e um de atividades. No diagrama de blocos são especificadas todas as estruturas presentes na implementação e suas relações. No diagrama de atividades são mostradas as relações a nível de execução dessas estruturas.

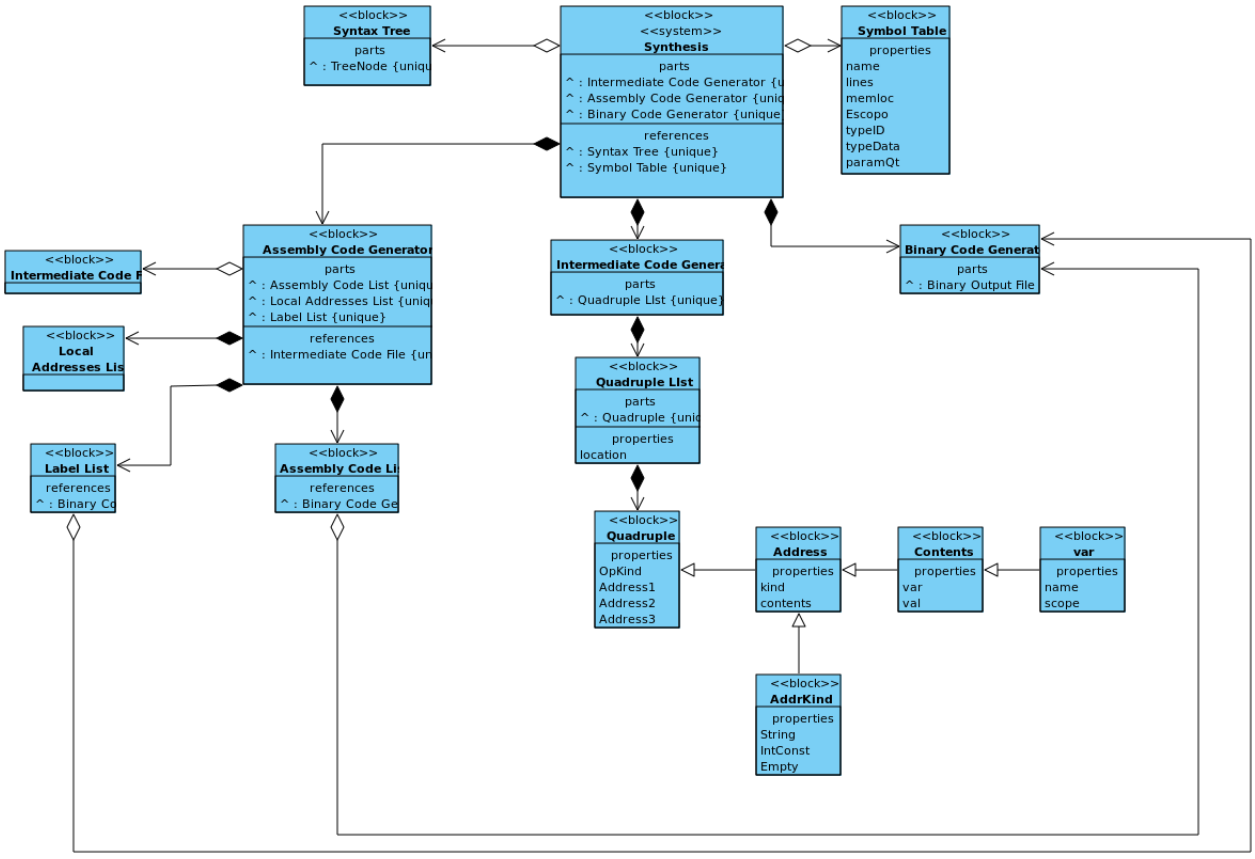

Figura 15 – Diagrama de Blocos - Fase de Síntese. Fonte: Autor.

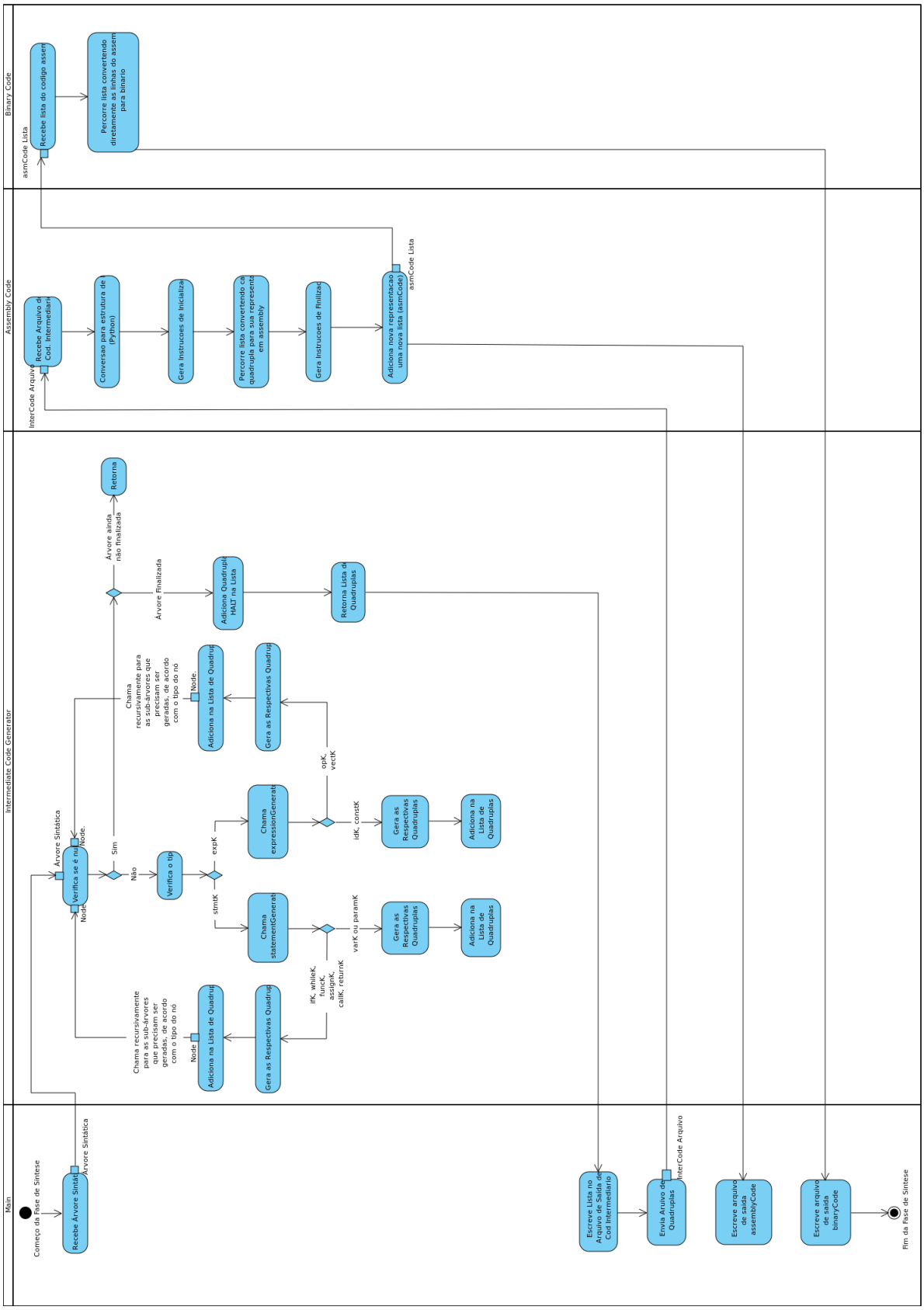

Figura 16 – Diagrama de Atividade - Fase de Síntese. Fonte: Autor.

4.2 Geração do Código Intermediário

Tal etapa opcional busca gerar uma representação do tipo "três endereços"onde cada procedimento é representado por uma tripla ou quadrupla de elementos, como proposto por (1). No caso do projeto desenvolvido a representação por quadruplas foi adotada onde cada uma delas foi adicionada à uma estrutura de lista encadeada para armazenar o código gerado, sequencialmente.

O modelo de quadrupla é composto por quatro campos cada:

- Campo 1: Identificador da operação;
- Campo 2: Endereço 1;
- Campo 3: Endereço 2;
- Campo 4: Endereço 3;

Nesse tipo de representação cada um dos endereços pode ser de alguns diferentes tipos:

- String: Parar armazenar valores de identificadores (variáveis, temporários e rótulos de desvio e funções);
- IntConst: Para armazenar valores numéricos constantes (imediatos desvios);
- Empty: Para identificar que o endereço é vazio (sem algum valor relacionado à ele);

As estruturas definidas para a representação em código das quadruplas são as seguintes (toda geração de código intermediário foi realizada em linguagem C):

```c
/* All structures definitions for the intermidiate code generation - Quadruples */
////////////////////////////////////////////////////////////////////////////////

/* Structure that defines the type of the operator for each quadruple */
typedef enum{ADDc = 0, SUBc, MULc, DIVc, EQc, DIFc, LTc, LETc, GTc, GETc, ANDc, ORc,
ASSIGNc, ALLOCc, IMMEc, LOADc, STOREc, ARRc, GOTOc, IFFc, RETc, FUNCc,
ENDc, PARAMc, CALLc, ARGc, LABc, HALTc} OpKind;

/*
    * String = Variables, temporaries, labbels
    * IntConst = Immidiates
    * Empty = for an ampty address
    */
/* Structure that defines the kind of each address */
typedef enum{String, IntConst, Empty} AddrKind;

/* Structure that defines and describes all relevant info present
in the address*/
typedef struct address{
    AddrKind kind;
    union{
        int val;
        struct{
            char *name;
            char *scope;
        } var;
    } contents;
} *Address;

/* Structure thar defines a quadruple (op, addr, addr, addr) */
typedef struct quadruple{
    OpKind op;
    Address addr1, addr2, addr3;
} *Quadruple;

/* Structure thar defines a unidirectional list of quadruples */
typedef struct QuadListElem{
    int location;
    Quadruple quadruple;
    struct QuadListElem *next;
} *QuadrupleListElem;
////////////////////////////////////////////////////////////////////////////////
```

Figura 17 – Estruturas definidas para a geração de Código Intermediário. Fonte: Autor.

A geração do código intermediário é feita baseada completamente nas informações presentes na árvore sintática e na tabela de símbolos. O procedimento é realizado percorrendo a árvore sintática em pré ou em ordem, dependendo do tipo de nó. De acordo com o tipo de nó encontrado, o procedimento recursivo é responsável por gerar todas as quadruplas dos nós presentes nas possíveis sub-árvores e também pela geração das quadruplas do nó atual. A tabela de símbolos é utilizada para o fornecimento de algumas informações auxiliares, como da quantidade de parâmetros necessária para determinada função e o escopo em que se encontra determinada variável ou função. O controle do número de variáveis temporárias é feita por uma função auxiliar no código, em nosso caso

ela possibilita a criação de 22 temporários.

As quadruplas suportadas pelo gerador de código intermediário são mostradas em Tipos de Quadruplas Suportadas. Fonte: Autor.

Operador	**Descrição**
ADD	Soma
SUB	Subtração
MUL	Multiplicação
DIV	Divisão
EQ	Igual
DIF	Diferente
LT	Menor
LET	Menor e Igual
GT	Maior
GET	Maior e Igual
AND	e lógico
OR	ou lógico
ASSIGN	Atribuição
ALLOC	Alocação
IMME	Imediato
LOAD	Carregar (Memória)
STORE	Guardar (Memória)
ARR	Vetor
GOTO	Deslocamento
IFF	Se Falso
RET	Retorno
FUNC	Chamada de Função
END	Fim de Função
PARAM	Parâmetro de Função
CALL	Chamada de Função
ARG	Argumento de Função
LAB	Rótulo
HALT	Parada

Tabela 3 – Tipos de Quadruplas Suportadas. Fonte: Autor.

Para o código exemplo apresentado em Compilador: Fase de Análise temos a geração do seguinte código intermediário.

```
..::Intermediate Code for ZAFx32 Processor::..

(FUNC, main, -, 1)
(ALLOC, a, 1, main)
(ALLOC, b, 1, main)
(ALLOC, c, 1, main)
(LOAD, $t0, a, -)
(CALL, $t1, input, 0)
(ASSIGN, $t0, $t1, -)
(STORE, a, $t0, -)
```

```
(LOAD, $t2, b, -)
(CALL, $t3, input, 0)
(ASSIGN, $t2, $t3, -)
(STORE, b, $t2, -)
(LOAD, $t4, c, -)
(LOAD, $t5, a, -)
(LOAD, $t6, b, -)
(ADD, $t7, $t5, $t6)
(LOAD, $t8, a, -)
(LOAD, $t9, b, -)
(SUB, $t10, $t8, $t9)
(MUL, $t11, $t7, $t10)
(ASSIGN, $t4, $t11, -)
(STORE, c, $t4, -)
(LOAD, $t12, c, -)
(ARG, $t12, -, -)
(CALL, $t13, output, 1)
(END, main, -, -)
(HALT, -, -, -)
```

Listing 4.1 – Código Intermediário do Código Exemplo, Fonte: Autor.

Para mais exemplos do funcionamento do gerador de Código Intermediário vide Exemplos.

4.3 Gerador de Código Assembly

Após a geração da estrutura geral do Código Intermediário, o gerador de Código Assembly é responsável pela conversão das quadruplas em representações equivalentes e sem perda de generalidade para a linguagem da máquina alvo, em nosso caso o *Soft Processor ZAFx32*. Vale a ressalva de que existe a conversão direta de alguns tipos de quadruplas para sua representação em Assembly, mas algumas, precisam de algumas linhas de Código Assembly para serem representadas e respeitar por completo sua funcionalidade.

A implementação do gerador do Código Assembly foi realizada em Python (versão 3.8). Tal escolha foi baseada no fato de que o processo de tradução do Código Intermediário para Assembly não é, no caso do projeto em questão, dependente de nenhuma estrutura construída nos códigos em C, apenas da tradução feita em cima do Código Intermediário gerado. Como Python se mostra como uma linguagem alto nível para tratamento de strings, optou-se por utilizá-la.

Como o Assembly é uma representação exclusiva da arquitetura algumas definições à respeito da mesma devem ser adotadas para garantir o funcionamento da abordagem em tal arquitetura alvo.

Uma definição que deve ser levada em consideração é em relação aos registradores disponíveis na arquitetura. Como definido em Banco de Registradores a arquitetura possui um total de 32 registradores, organizados da seguinte forma.

- Registrador 0: $zero - Armazena sempre o valor 0;
- Registrador 1: $sp - Stack Pointer;
- Registrador 2: $fp - Frame Pointer;
- Registrador 3: $gp - Global Pointer;
- Registrador 4: $ra - Endereço de Retorno;
- Registrador 5: $ret - Valor de Retorno;
- Registradores 6 - 9: $a0 - $a3 - Argumentos de Função;
- Registradores 10 - 31: $t0 - $t21 - Registradores de Uso Geral;

A ordem apresentada na tabela corresponde a ordem que tais registradores são dispostos no Bando de Registradores da arquitetura alvo.

A detalhe de implementação, o texto produzido pelo gerador de Código Intermediário é lido. Linha a linha o Código Intermediário é convertida para sua estrutura em Assembly, onde cada uma das linhas dessa nova representação é salva em uma estrutura de lista. Essa lista será utilizada para a produção do texto de saída do Código Assembly bem como para a conversão do mesmo para Código Executável (Código de Máquina). Vale a ressalva de que a geração do Código Assembly e sua conversão (basicamente direta) para Código Executável foram realizadas no mesmo código em Python.

Para o código exemplo apresentado em Compilador: Fase de Análise temos a geração do seguinte código Assembly.

```
..::Assembly Code for ZAFx32 Processor::..

0:          li sp, -1
1:          li fp, 0
2:          li zero, 0
3:          li gp, 249
4:          j main
.main
5:          addi sp, sp, 1
6:          addi sp, sp, 1
7:          addi sp, sp, 1
8:          load t0, fp, 0
9:          in t1
10:         mov t0, t1
11:         store t0, fp, 0
12:         load t2, fp, 1
13:         in t3
14:         mov t2, t3
15:         store t2, fp, 1
16:         load t4, fp, 2
17:         load t5, fp, 0
18:         load t6, fp, 1
19:         add t7, t5, t6
```

```
20:         load t8, fp, 0
21:         load t9, fp, 1
22:         sub t10, t8, t9
23:         mul t11, t7, t10
24:         mov t4, t11
25:         store t4, fp, 2
26:         load t12, fp, 2
27:         mov a0, t12
28:         mov t13, a0
29:         out t13
30:         j end
.end
31:         halt
```

Listing 4.2 – Código Assembly do Código Exemplo, Fonte: Autor.

Para mais exemplos do funcionamento do gerador de Código Assembly vide Exemplos.

4.4 Gerador de Código Executável

O gerador de Código Executável tem a função de converter a representação em Assembly gerada para sua representação à nível de máquina, para que essa possa ser carregada na Memória de Instruções do processador e assim executada.

A detalhe de implementação, o código foi produzido percorrendo-se a lista com as linhas do Código Assembly gerada no passo anterior, e convertendo cada uma das informações presentes em cada linha da lista, para sua representação binária e escrita assim no arquivo de saída.

Vale a ressalva de que optou-se pela utilização de uma representação suportada pela *HDL - Verilog* em que os valores explicitamente binários não precisam ser escritos. A representação é feita pela concatenação de partes, por exemplo, para gerarmos o código binário representado por uma instrução de *j* (Jump) de OpCode 20 (decimal) e com valor de desvio de 65 (decimal), fazemos apenas: {6'd20, 26'd65}. Isso nos gera uma palavra binária de 32 Bits composta pelo número 20 decimal representado em 6 Bits concatenado pelo número 65 decimal representado em 26 Bits. Tal abordagem facilita a compreensão do código para leitura, visto que os valores em decimal referentes a cada campo de cada instrução ficam evidentes, e sua leitura é mais fácil quando comparada à leitura em binário.

Para o código exemplo apresentado em Compilador: Fase de Análise temos a geração do seguinte código Executável.

```
..::Binary Code for ZAFx32 Processor::..

MemInst[0] = { 6'd17, 5'd0, 5'd1, 16'd-1 }      //li
MemInst[1] = { 6'd17, 5'd0, 5'd2, 16'd0 }      //li
MemInst[2] = { 6'd17, 5'd0, 5'd0, 16'd0 }      //li
MemInst[3] = { 6'd17, 5'd0, 5'd3, 16'd249 }      //li
MemInst[4] = { 6'd20, 26'd5 }                 //j
```

```
8  //main
9  MemInst[5] = { 6'd1, 5'd1, 5'd1, 16'd1 }    //addi
10 MemInst[6] = { 6'd1, 5'd1, 5'd1, 16'd1 }    //addi
11 MemInst[7] = { 6'd1, 5'd1, 5'd1, 16'd1 }    //addi
12 MemInst[8] = { 6'd23, 5'd2, 5'd10, 16'd0 }    //load
13 MemInst[9] = { 6'd21, 26'd11 }                //in
14 MemInst[10] = { 6'd16, 5'd11, 5'd10, 16'd0 }     //mov
15 MemInst[11] = { 6'd24, 5'd2, 5'd10, 16'd0 }   //store
16 MemInst[12] = { 6'd23, 5'd2, 5'd12, 16'd1 }    //load
17 MemInst[13] = { 6'd21, 26'd13 }                //in
18 MemInst[14] = { 6'd16, 5'd13, 5'd12, 16'd0 }     //mov
19 MemInst[15] = { 6'd24, 5'd2, 5'd12, 16'd1 }   //store
20 MemInst[16] = { 6'd23, 5'd2, 5'd14, 16'd2 }    //load
21 MemInst[17] = { 6'd23, 5'd2, 5'd15, 16'd0 }    //load
22 MemInst[18] = { 6'd23, 5'd2, 5'd16, 16'd1 }    //load
23 MemInst[19] = { 6'd0, 5'd17, 5'd15, 5'd16, 11'd0 }     //add
24 MemInst[20] = { 6'd23, 5'd2, 5'd18, 16'd0 }    //load
25 MemInst[21] = { 6'd23, 5'd2, 5'd19, 16'd1 }    //load
26 MemInst[22] = { 6'd2, 5'd20, 5'd18, 5'd19, 11'd0 }     //sub
27 MemInst[23] = { 6'd3, 5'd21, 5'd17, 5'd20, 11'd0 }     //mul
28 MemInst[24] = { 6'd16, 5'd21, 5'd14, 16'd0 }     //mov
29 MemInst[25] = { 6'd24, 5'd2, 5'd14, 16'd2 }   //store
30 MemInst[26] = { 6'd23, 5'd2, 5'd22, 16'd2 }    //load
31 MemInst[27] = { 6'd16, 5'd22, 5'd6, 16'd0 }     //mov
32 MemInst[28] = { 6'd16, 5'd6, 5'd23, 16'd0 }     //mov
33 MemInst[29] = { 6'd22, 26'd23 }               //out
34 MemInst[30] = { 6'd20, 26'd31 }                 //j
35 //end
36 MemInst[31] = { 6'd27, 26'd0 }              //halt
```

Listing 4.3 – Código Executável do Código Exemplo, Fonte: Autor.

Para mais exemplos do funcionamento do gerador de Código Executável vide Exemplos.

Um ultimo adendo é de que para cada linha gerada, a instrução correspondente foi explicitada a sua direta, bem como a identificação de rótulos de deslocamento e de funções, em forma de comentário (para a *HDL - Verilog*). Isso foi feito para facilitar a leitura do Código Executável.

4.5 Gerenciamento de Memória

Uma parte crucial do projeto é em relação ao gerenciamento de memória. Nosso objetivo foi abordar uma estratégia que possibilitasse a alocação de memória de forma correta para qualquer caso suportado pela estrutura da linguagem definida, isto é, permitir a alocação de variáveis e vetores locais à funções, alocação de variáveis e vetores globais, recursão e passagem de variáveis por valor para funções e passagem de vetores por referência para funções.

Desta forma o método de gerenciamento de memória adotado foi baseado no método por pilha apresentado na Arquitetura MIPS. Nesse método cada função possui

um espaço de memória referente á ela chamado *Memory Frame* que são delimitados por dois registradores, o $sp que armazena o endereço de memória do topo do frame atual e o registrador $fp que armazena o endereço de memória do início do frame atual. Dentro de cada frame é realizado o armazenamento de todas as declarações locais à função.

A medida que as funções são chamadas, é armazenado no começo do frame o valor de $fp anterior e o valor de $ra anterior. Tais salvamentos são importantes para localizarmos exatamente o frame da função que chamou a função atual. Esse procedimento é indispensável para o suporte à recursão, visto que as chamadas serão empilhadas e valores locais serão mantidos na memória para cada uma das chamadas.

Em relação às alocações globais, temos que é alocado na memória um espaço de 50 posições para esse tipo de armazenamento, *Global Frame*, onde o endereço da base desse frame especial é armazenado em $gp.

Em realação a como a memória é mapeada, os frames das funções (incluindo a main) são armazenados partindo do endereço 0 da memória (considerando que a memória tem enderço 0 em sua base e cresce até o valor limite) e empilhados de forma crescente. Já o *Global Frame* é mapeado na porção superior da memória indo da posição $(MEM\_SIZE - 1) - 50$ até $MEM\_SIZE - 1$. A figura 18 ilustra a proposta.

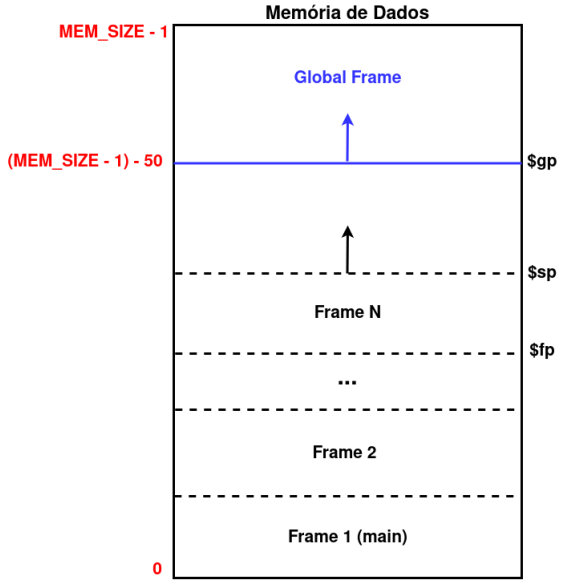

Figura 18 – Mapeamento da memória. Fonte: Autor.

Um *Memory Frame* armazena as seguintes informações, na respectiva ordem (de baixo para cima).

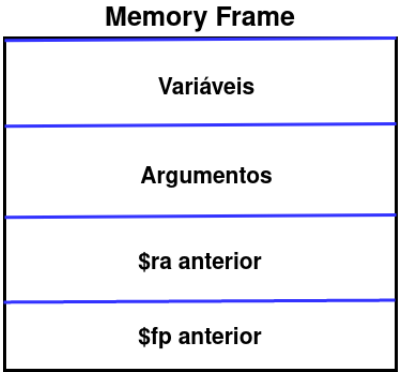

Figura 19 – Memory Frame. Fonte: Autor.

Em relação à alocação de vetores e passada dos mesmos para funções temos que quando são alocados, é reservado no *Memory Frame* atual um espaço do tamanho correspondente ao vetor, quando sua utilização é feita localmente o cálculo da posição do elemento que estamos buscando é feito baseado no $fp atual ou $gp, caso seja um acesso direto ao vetor global. Quando o acesso é feito a um vetor não local, passado para uma função como referência, o valor que é passado para função é exatamente da posição base da memória onde a alocação foi feita, sendo assim os cálculos são feitos em relação à esse valor base (o qual permanece armazenado no *Memory Frame* na região de argumentos passados à função).

5 Exemplos

Ao todo foram realizados quatro exemplos para a verificação do funcionamento do compilador proposto. Os exemplos procuram englobar a maioria das estruturas propostas, como por exemplo:

- Laços de repetição;
- Estruturas Condicionais;
- Declaração de variáveis;
- Declaração e uso de funções;
- Declaração e uso de vetores (localmente/globalmente e passado por referência em funções);
- Declarações locais e globais;
- Recursão;
- Estruturas aritméticas e lógicas;

A seguir é exposto cada um dos exemplos em C- seguidos dos respectivos códigos Intermediários, Asembly e Executável.

5.1 GCD

5.1.1 Código em C-

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

Listing 5.1 – Código C- GCD, Fonte: Autor.

5.1.2 Código Intermediário

```
..::Intermediate Code for ZAFx32 Processor::..

(FUNC, gcd, -, 2)
(PARAM, u, -, gcd)
(PARAM, v, -, gcd)
(LOAD, $t0, v, -)
(IMME, $t1, 0, -)
(EQ, $t2, $t0, $t1)
(IFF, $t2, L0, -)
(LOAD, $t3, u, -)
(RET, $t3, -, -)
(GOTO, L1, -, -)
(LAB, L0, -, -)
(LOAD, $t4, v, -)
(ARG, $t4, -, -)
(LOAD, $t5, u, -)
(LOAD, $t6, u, -)
(LOAD, $t7, v, -)
(DIV, $t8, $t6, $t7)
(LOAD, $t9, v, -)
(MUL, $t10, $t8, $t9)
(SUB, $t11, $t5, $t10)
(ARG, $t11, -, -)
(CALL, $t12, gcd, 2)
(RET, $t12, -, -)
(LAB, L1, -, -)
(END, gcd, -, -)
(FUNC, main, -, 1)
(ALLOC, x, 1, main)
(ALLOC, y, 1, main)
(LOAD, $t13, x, -)
(CALL, $t14, input, 0)
(ASSIGN, $t13, $t14, -)
(STORE, x, $t13, -)
(LOAD, $t15, y, -)
(CALL, $t16, input, 0)
(ASSIGN, $t15, $t16, -)
(STORE, y, $t15, -)
(LOAD, $t17, x, -)
(ARG, $t17, -, -)
(LOAD, $t18, y, -)
(ARG, $t18, -, -)
(CALL, $t19, gcd, 2)
(ARG, $t19, -, -)
(CALL, $t20, output, 1)
(END, main, -, -)
(HALT, -, -, -)
```

Listing 5.2 – Código Intermediário - GCD. Fonte: Autor.

5.1.3 Código Assembly

```
..::Assembly Code for ZAFx32 Processor::..

0:          li sp, -1
1:          li fp, 0
2:          li zero, 0
3:          li gp, 249
```

```
4:          j main
.gcd
5:          addi sp, sp, 2
6:          store ra, sp, 0
7:          store fp, sp, -1
8:          mov fp, sp
9:          addi fp, fp, 1
10:         addi sp, sp, 1
11:         store a0, sp, 0
12:         addi sp, sp, 1
13:         store a1, sp, 0
14:         load t0, fp, 1
15:         li t1, 0
16:         eq t2, t0, t1
17:         beq t2, zero, L0
18:         load t3, fp, 0
19:         mov ret, t3
20:         addi fp, fp, -2
21:         load ra, fp, 1
22:         mov sp, fp
23:         addi sp, sp, -1
24:         load fp, fp, 0
25:         jr ra
26:         j L1
.L0
27:         load t4, fp, 1
28:         mov a0, t4
29:         load t5, fp, 0
30:         load t6, fp, 0
31:         load t7, fp, 1
32:         div t8, t6, t7
33:         load t9, fp, 1
34:         mul t10, t8, t9
35:         sub t11, t5, t10
36:         mov a1, t11
37:         jal gcd
38:         mov t12, ret
39:         mov ret, t12
40:         addi fp, fp, -2
41:         load ra, fp, 1
42:         mov sp, fp
43:         addi sp, sp, -1
44:         load fp, fp, 0
45:         jr ra
.L1
46:         addi fp, fp, -2
47:         load ra, fp, 1
48:         mov sp, fp
49:         addi sp, sp, -1
50:         load fp, fp, 0
51:         jr ra
.main
52:         addi sp, sp, 1
53:         addi sp, sp, 1
54:         load t13, fp, 0
55:         in t14
56:         mov t13, t14
57:         store t13, fp, 0
58:         load t15, fp, 1
59:         in t16
```

```
60:         mov t15, t16
61:         store t15, fp, 1
62:         load t17, fp, 0
63:         mov a0, t17
64:         load t18, fp, 1
65:         mov a1, t18
66:         jal gcd
67:         mov t19, ret
68:         mov a0, t19
69:         mov t20, a0
70:         out t20
71:         j end
.end
72:         halt
```

Listing 5.3 – Código Assembly - GCD. Fonte: Autor.

5.1.4 Código Executável

```
..::Binary Code for ZAFx32 Processor::..

MemInst[0] = { 6'd17, 5'd0, 5'd1, 16'd-1 }      //li
MemInst[1] = { 6'd17, 5'd0, 5'd2, 16'd0 }      //li
MemInst[2] = { 6'd17, 5'd0, 5'd0, 16'd0 }      //li
MemInst[3] = { 6'd17, 5'd0, 5'd3, 16'd249 }      //li
MemInst[4] = { 6'd20, 26'd52 }                  //j
//gcd
MemInst[5] = { 6'd1, 5'd1, 5'd1, 16'd2 }    //addi
MemInst[6] = { 6'd24, 5'd1, 5'd4, 16'd0 }   //store
MemInst[7] = { 6'd24, 5'd1, 5'd2, -16'd1 }   //store
MemInst[8] = { 6'd16, 5'd1, 5'd2, 16'd0 }     //mov
MemInst[9] = { 6'd1, 5'd2, 5'd2, 16'd1 }    //addi
MemInst[10] = { 6'd1, 5'd1, 5'd1, 16'd1 }    //addi
MemInst[11] = { 6'd24, 5'd1, 5'd6, 16'd0 }   //store
MemInst[12] = { 6'd1, 5'd1, 5'd1, 16'd1 }    //addi
MemInst[13] = { 6'd24, 5'd1, 5'd7, 16'd0 }   //store
MemInst[14] = { 6'd23, 5'd2, 5'd10, 16'd1 }    //load
MemInst[15] = { 6'd17, 5'd0, 5'd11, 16'd0 }      //li
MemInst[16] = { 6'd28, 5'd12, 5'd10, 5'd11, 11'd0 }      //eq
MemInst[17] = { 6'd18, 5'd12, 5'd0, 16'd27 }     //beq
MemInst[18] = { 6'd23, 5'd2, 5'd13, 16'd0 }    //load
MemInst[19] = { 6'd16, 5'd13, 5'd5, 16'd0 }     //mov
MemInst[20] = { 6'd1, 5'd2, 5'd2, -16'd2 }    //addi
MemInst[21] = { 6'd23, 5'd2, 5'd4, 16'd1 }    //load
MemInst[22] = { 6'd16, 5'd2, 5'd1, 16'd0 }     //mov
MemInst[23] = { 6'd1, 5'd1, 5'd1, -16'd1 }    //addi
MemInst[24] = { 6'd23, 5'd2, 5'd2, 16'd0 }    //load
MemInst[25] = { 6'd25, 26'd4 }                //jr
MemInst[26] = { 6'd20, 26'd46 }                 //j
//L0
MemInst[27] = { 6'd23, 5'd2, 5'd14, 16'd1 }    //load
MemInst[28] = { 6'd16, 5'd14, 5'd6, 16'd0 }     //mov
MemInst[29] = { 6'd23, 5'd2, 5'd15, 16'd0 }    //load
MemInst[30] = { 6'd23, 5'd2, 5'd16, 16'd0 }    //load
MemInst[31] = { 6'd23, 5'd2, 5'd17, 16'd1 }    //load
MemInst[32] = { 6'd4, 5'd18, 5'd16, 5'd17, 11'd0 }     //div
MemInst[33] = { 6'd23, 5'd2, 5'd19, 16'd1 }    //load
MemInst[34] = { 6'd3, 5'd20, 5'd18, 5'd19, 11'd0 }     //mul
MemInst[35] = { 6'd2, 5'd21, 5'd15, 5'd20, 11'd0 }     //sub
```

```
41 MemInst[36] = { 6'd16, 5'd21, 5'd7, 16'd0 }      //mov
42 MemInst[37] = { 6'd26, 26'd5 }                 //jal
43 MemInst[38] = { 6'd16, 5'd5, 5'd22, 16'd0 }      //mov
44 MemInst[39] = { 6'd16, 5'd22, 5'd5, 16'd0 }      //mov
45 MemInst[40] = { 6'd1, 5'd2, 5'd2, -16'd2 }     //addi
46 MemInst[41] = { 6'd23, 5'd2, 5'd4, 16'd1 }     //load
47 MemInst[42] = { 6'd16, 5'd2, 5'd1, 16'd0 }      //mov
48 MemInst[43] = { 6'd1, 5'd1, 5'd1, -16'd1 }     //addi
49 MemInst[44] = { 6'd23, 5'd2, 5'd2, 16'd0 }     //load
50 MemInst[45] = { 6'd25, 26'd4 }                  //jr
51 //L1
52 MemInst[46] = { 6'd1, 5'd2, 5'd2, -16'd2 }     //addi
53 MemInst[47] = { 6'd23, 5'd2, 5'd4, 16'd1 }     //load
54 MemInst[48] = { 6'd16, 5'd2, 5'd1, 16'd0 }      //mov
55 MemInst[49] = { 6'd1, 5'd1, 5'd1, -16'd1 }     //addi
56 MemInst[50] = { 6'd23, 5'd2, 5'd2, 16'd0 }     //load
57 MemInst[51] = { 6'd25, 26'd4 }                  //jr
58 //main
59 MemInst[52] = { 6'd1, 5'd1, 5'd1, 16'd1 }     //addi
60 MemInst[53] = { 6'd1, 5'd1, 5'd1, 16'd1 }     //addi
61 MemInst[54] = { 6'd23, 5'd2, 5'd23, 16'd0 }     //load
62 MemInst[55] = { 6'd21, 26'd24 }                  //in
63 MemInst[56] = { 6'd16, 5'd24, 5'd23, 16'd0 }      //mov
64 MemInst[57] = { 6'd24, 5'd2, 5'd23, 16'd0 }    //store
65 MemInst[58] = { 6'd23, 5'd2, 5'd25, 16'd1 }     //load
66 MemInst[59] = { 6'd21, 26'd26 }                  //in
67 MemInst[60] = { 6'd16, 5'd26, 5'd25, 16'd0 }      //mov
68 MemInst[61] = { 6'd24, 5'd2, 5'd25, 16'd1 }    //store
69 MemInst[62] = { 6'd23, 5'd2, 5'd27, 16'd0 }     //load
70 MemInst[63] = { 6'd16, 5'd27, 5'd6, 16'd0 }      //mov
71 MemInst[64] = { 6'd23, 5'd2, 5'd28, 16'd1 }     //load
72 MemInst[65] = { 6'd16, 5'd28, 5'd7, 16'd0 }      //mov
73 MemInst[66] = { 6'd26, 26'd5 }                 //jal
74 MemInst[67] = { 6'd16, 5'd5, 5'd29, 16'd0 }      //mov
75 MemInst[68] = { 6'd16, 5'd29, 5'd6, 16'd0 }      //mov
76 MemInst[69] = { 6'd16, 5'd6, 5'd30, 16'd0 }      //mov
77 MemInst[70] = { 6'd22, 26'd30 }                 //out
78 MemInst[71] = { 6'd20, 26'd72 }                   //j
79 //end
80 MemInst[72] = { 6'd27, 26'd0 }                //halt
```

Listing 5.4 – Código Executável - GCD. Fonte: Autor.

5.2 Sort

5.2.1 Código em C-

```
1  /* programa para ordenacao por selecao de
2     uma matriz com dez elementos. */
3
4  int vet[ 10 ];
5
6  int minloc ( int a[], int low, int high )
7  {       int i; int x; int k;
8          k = low;
9          x = a[low];
10         i = low + 1;
11         while (i < high){
```

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

Listing 5.5 – Código C- Sort, Fonte: Autor.

5.2.2 Código Intermediário

```
1   ..::Intermediate Code for ZAFx32 Processor::..
2
3   (ALLOC, vet, 10, global)
4   (FUNC, minloc, -, 3)
5   (PARAM, a, -, minloc)
6   (PARAM, low, -, minloc)
7   (PARAM, 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, $t3, low, -)
17  (ARR, $t4, a, $t3)
```

```
(ASSIGN, $t2, $t4, -)
(STORE, x, $t2, -)
(LOAD, $t5, i, -)
(LOAD, $t6, low, -)
(IMME, $t7, 1, -)
(ADD, $t8, $t6, $t7)
(ASSIGN, $t5, $t8, -)
(STORE, i, $t5, -)
(LAB, L0, -, -)
(LOAD, $t9, i, -)
(LOAD, $t10, high, -)
(LT, $t11, $t9, $t10)
(IFF, $t11, L1, -)
(LOAD, $t12, i, -)
(ARR, $t13, a, $t12)
(LOAD, $t14, x, -)
(LT, $t15, $t13, $t14)
(IFF, $t15, L2, -)
(LOAD, $t16, x, -)
(LOAD, $t17, i, -)
(ARR, $t18, a, $t17)
(ASSIGN, $t16, $t18, -)
(STORE, x, $t16, -)
(LOAD, $t19, k, -)
(LOAD, $t20, i, -)
(ASSIGN, $t19, $t20, -)
(STORE, k, $t19, -)
(GOTO, L3, -, -)
(LAB, L2, -, -)
(LAB, L3, -, -)
(LOAD, $t21, i, -)
(LOAD, $t0, i, -)
(IMME, $t1, 1, -)
(ADD, $t2, $t0, $t1)
(ASSIGN, $t21, $t2, -)
(STORE, i, $t21, -)
(GOTO, L0, -, -)
(LAB, L1, -, -)
(LOAD, $t3, k, -)
(RET, $t3, -, -)
(END, minloc, -, -)
(FUNC, sort, -, 3)
(PARAM, a, -, sort)
(PARAM, low, -, sort)
(PARAM, high, -, sort)
(ALLOC, i, 1, sort)
(ALLOC, k, 1, sort)
(LOAD, $t4, i, -)
(LOAD, $t5, low, -)
(ASSIGN, $t4, $t5, -)
(STORE, i, $t4, -)
(LAB, L4, -, -)
(LOAD, $t6, i, -)
(LOAD, $t7, high, -)
(IMME, $t8, 1, -)
(SUB, $t9, $t7, $t8)
(LT, $t10, $t6, $t9)
(IFF, $t10, L5, -)
(ALLOC, t, 1, sort)
(LOAD, $t11, k, -)
```

```
78  (LOAD, $t12, a, -)
79  (ARG, $t12, -, -)
80  (LOAD, $t13, i, -)
81  (ARG, $t13, -, -)
82  (LOAD, $t14, high, -)
83  (ARG, $t14, -, -)
84  (CALL, $t15, minloc, 3)
85  (ASSIGN, $t11, $t15, -)
86  (STORE, k, $t11, -)
87  (LOAD, $t16, t, -)
88  (LOAD, $t17, k, -)
89  (ARR, $t18, a, $t17)
90  (ASSIGN, $t16, $t18, -)
91  (STORE, t, $t16, -)
92  (LOAD, $t19, k, -)
93  (ARR, $t20, a, $t19)
94  (LOAD, $t21, i, -)
95  (ARR, $t0, a, $t21)
96  (ASSIGN, $t20, $t0, -)
97  (STORE, a, $t20, $t19)
98  (LOAD, $t1, i, -)
99  (ARR, $t2, a, $t1)
100 (LOAD, $t3, t, -)
101 (ASSIGN, $t2, $t3, -)
102 (STORE, a, $t2, $t1)
103 (LOAD, $t4, i, -)
104 (LOAD, $t5, i, -)
105 (IMME, $t6, 1, -)
106 (ADD, $t7, $t5, $t6)
107 (ASSIGN, $t4, $t7, -)
108 (STORE, i, $t4, -)
109 (GOTO, L4, -, -)
110 (LAB, L5, -, -)
111 (END, sort, -, -)
112 (FUNC, main, -, 1)
113 (ALLOC, i, 1, main)
114 (LOAD, $t8, i, -)
115 (IMME, $t9, 0, -)
116 (ASSIGN, $t8, $t9, -)
117 (STORE, i, $t8, -)
118 (LAB, L6, -, -)
119 (LOAD, $t10, i, -)
120 (IMME, $t11, 10, -)
121 (LT, $t12, $t10, $t11)
122 (IFF, $t12, L7, -)
123 (LOAD, $t13, i, -)
124 (ARR, $t14, vet, $t13)
125 (CALL, $t15, input, 0)
126 (ASSIGN, $t14, $t15, -)
127 (STORE, vet, $t14, $t13)
128 (LOAD, $t16, i, -)
129 (LOAD, $t17, i, -)
130 (IMME, $t18, 1, -)
131 (ADD, $t19, $t17, $t18)
132 (ASSIGN, $t16, $t19, -)
133 (STORE, i, $t16, -)
134 (GOTO, L6, -, -)
135 (LAB, L7, -, -)
136 (LOAD, $t20, vet, -)
137 (ARG, $t20, -, -)
```

```
(IMME, $t21, 0, -)
(ARG, $t21, -, -)
(IMME, $t0, 10, -)
(ARG, $t0, -, -)
(CALL, $t1, sort, 3)
(LOAD, $t2, i, -)
(IMME, $t3, 0, -)
(ASSIGN, $t2, $t3, -)
(STORE, i, $t2, -)
(LAB, L8, -, -)
(LOAD, $t4, i, -)
(IMME, $t5, 10, -)
(LT, $t6, $t4, $t5)
(IFF, $t6, L9, -)
(LOAD, $t7, i, -)
(ARR, $t8, vet, $t7)
(ARG, $t8, -, -)
(CALL, $t9, output, 1)
(LOAD, $t10, i, -)
(LOAD, $t11, i, -)
(IMME, $t12, 1, -)
(ADD, $t13, $t11, $t12)
(ASSIGN, $t10, $t13, -)
(STORE, i, $t10, -)
(GOTO, L8, -, -)
(LAB, L9, -, -)
(END, main, -, -)
(HALT, -, -, -)
```

Listing 5.6 – Código Intermediário - Sort. Fonte: Autor.

5.2.3 Código Assembly

```
..::Assembly Code for ZAFx32 Processor:::.

0:          li sp, -1
1:          li fp, 0
2:          li zero, 0
3:          li gp, 249
4:          j main
.minloc
5:          addi sp, sp, 2
6:          store ra, sp, 0
7:          store fp, sp, -1
8:          mov fp, sp
9:          addi fp, fp, 1
10:         addi sp, sp, 1
11:         store a0, sp, 0
12:         addi sp, sp, 1
13:         store a1, sp, 0
14:         addi sp, sp, 1
15:         store a2, sp, 0
16:         addi sp, sp, 1
17:         addi sp, sp, 1
18:         addi sp, sp, 1
19:         load t0, fp, 5
20:         load t1, fp, 1
21:         mov t0, t1
22:         store t0, fp, 5
```

```
23:         load t2, fp, 4
24:         load t3, fp, 1
25:         load t4, fp, 0
26:         add t3, t4, t3
27:         load t4, t3, 0
28:         mov t2, t4
29:         store t2, fp, 4
30:         load t5, fp, 3
31:         load t6, fp, 1
32:         li t7, 1
33:         add t8, t6, t7
34:         mov t5, t8
35:         store t5, fp, 3
.L0
36:         load t9, fp, 3
37:         load t10, fp, 2
38:         slt t11, t9, t10
39:         beq t11, zero, L1
40:         load t12, fp, 3
41:         load t13, fp, 0
42:         add t12, t13, t12
43:         load t13, t12, 0
44:         load t14, fp, 4
45:         slt t15, t13, t14
46:         beq t15, zero, L2
47:         load t16, fp, 4
48:         load t17, fp, 3
49:         load t18, fp, 0
50:         add t17, t18, t17
51:         load t18, t17, 0
52:         mov t16, t18
53:         store t16, fp, 4
54:         load t19, fp, 5
55:         load t20, fp, 3
56:         mov t19, t20
57:         store t19, fp, 5
58:         j L3
.L2
.L3
59:         load t21, fp, 3
60:         load t0, fp, 3
61:         li t1, 1
62:         add t2, t0, t1
63:         mov t21, t2
64:         store t21, fp, 3
65:         j L0
.L1
66:         load t3, fp, 5
67:         mov ret, t3
68:         addi fp, fp, -2
69:         load ra, fp, 1
70:         mov sp, fp
71:         addi sp, sp, -1
72:         load fp, fp, 0
73:         jr ra
74:         addi fp, fp, -2
75:         load ra, fp, 1
76:         mov sp, fp
77:         addi sp, sp, -1
78:         load fp, fp, 0
```

```
79:         jr ra
.sort
80:         addi sp, sp, 2
81:         store ra, sp, 0
82:         store fp, sp, -1
83:         mov fp, sp
84:         addi fp, fp, 1
85:         addi sp, sp, 1
86:         store a0, sp, 0
87:         addi sp, sp, 1
88:         store a1, sp, 0
89:         addi sp, sp, 1
90:         store a2, sp, 0
91:         addi sp, sp, 1
92:         addi sp, sp, 1
93:         load t4, fp, 3
94:         load t5, fp, 1
95:         mov t4, t5
96:         store t4, fp, 3
.L4
97:         load t6, fp, 3
98:         load t7, fp, 2
99:         li t8, 1
100:        sub t9, t7, t8
101:        slt t10, t6, t9
102:        beq t10, zero, L5
103:        addi sp, sp, 1
104:        load t11, fp, 4
105:        load t12, fp, 0
106:        mov a0, t12
107:        load t13, fp, 3
108:        mov a1, t13
109:        load t14, fp, 2
110:        mov a2, t14
111:        jal minloc
112:        mov t15, ret
113:        mov t11, t15
114:        store t11, fp, 4
115:        load t16, fp, 5
116:        load t17, fp, 4
117:        load t18, fp, 0
118:        add t17, t18, t17
119:        load t18, t17, 0
120:        mov t16, t18
121:        store t16, fp, 5
122:        load t19, fp, 4
123:        load t20, fp, 0
124:        add t19, t20, t19
125:        load t20, t19, 0
126:        load t21, fp, 3
127:        load t0, fp, 0
128:        add t21, t0, t21
129:        load t0, t21, 0
130:        mov t20, t0
131:        store t20, t19, 0
132:        load t1, fp, 3
133:        load t2, fp, 0
134:        add t1, t2, t1
135:        load t2, t1, 0
136:        load t3, fp, 5
```

```
137:        mov t2, t3
138:        store t2, t1, 0
139:        load t4, fp, 3
140:        load t5, fp, 3
141:        li t6, 1
142:        add t7, t5, t6
143:        mov t4, t7
144:        store t4, fp, 3
145:        j L4
.L5
146:        addi fp, fp, -2
147:        load ra, fp, 1
148:        mov sp, fp
149:        addi sp, sp, -1
150:        load fp, fp, 0
151:        jr ra
.main
152:        addi sp, sp, 1
153:        load t8, fp, 0
154:        li t9, 0
155:        mov t8, t9
156:        store t8, fp, 0
.L6
157:        load t10, fp, 0
158:        li t11, 10
159:        slt t12, t10, t11
160:        beq t12, zero, L7
161:        load t13, fp, 0
162:        add t13, gp, t13
163:        addi t13, t13, 0
164:        load t14, t13, 0
165:        in t15
166:        mov t14, t15
167:        store t14, t13, 0
168:        load t16, fp, 0
169:        load t17, fp, 0
170:        li t18, 1
171:        add t19, t17, t18
172:        mov t16, t19
173:        store t16, fp, 0
174:        j L6
.L7
175:        addi t20, gp, 0
176:        mov a0, t20
177:        li t21, 0
178:        mov a1, t21
179:        li t0, 10
180:        mov a2, t0
181:        jal sort
182:        mov t1, ret
183:        load t2, fp, 0
184:        li t3, 0
185:        mov t2, t3
186:        store t2, fp, 0
.L8
187:        load t4, fp, 0
188:        li t5, 10
189:        slt t6, t4, t5
190:        beq t6, zero, L9
191:        load t7, fp, 0
```

```
192:        add t7, gp, t7
193:        addi t7, t7, 0
194:        load t8, t7, 0
195:        mov a0, t8
196:        mov t9, a0
197:        out t9
198:        load t10, fp, 0
199:        load t11, fp, 0
200:        li t12, 1
201:        add t13, t11, t12
202:        mov t10, t13
203:        store t10, fp, 0
204:        j L8
.L9
205:        j end
.end
206:        halt
```

Listing 5.7 – Código Assembly - Sort. Fonte: Autor.

5.2.4 Código Executável

```
..::Binary Code for ZAFx32 Processor::..

MemInst[0] = { 6'd17, 5'd0, 5'd1, 16'd-1 }      //li
MemInst[1] = { 6'd17, 5'd0, 5'd2, 16'd0 }      //li
MemInst[2] = { 6'd17, 5'd0, 5'd0, 16'd0 }      //li
MemInst[3] = { 6'd17, 5'd0, 5'd3, 16'd249 }      //li
MemInst[4] = { 6'd20, 26'd152 }                  //j
//minloc
MemInst[5] = { 6'd1, 5'd1, 5'd1, 16'd2 }    //addi
MemInst[6] = { 6'd24, 5'd1, 5'd4, 16'd0 }   //store
MemInst[7] = { 6'd24, 5'd1, 5'd2, -16'd1 }   //store
MemInst[8] = { 6'd16, 5'd1, 5'd2, 16'd0 }     //mov
MemInst[9] = { 6'd1, 5'd2, 5'd2, 16'd1 }    //addi
MemInst[10] = { 6'd1, 5'd1, 5'd1, 16'd1 }    //addi
MemInst[11] = { 6'd24, 5'd1, 5'd6, 16'd0 }   //store
MemInst[12] = { 6'd1, 5'd1, 5'd1, 16'd1 }    //addi
MemInst[13] = { 6'd24, 5'd1, 5'd7, 16'd0 }   //store
MemInst[14] = { 6'd1, 5'd1, 5'd1, 16'd1 }    //addi
MemInst[15] = { 6'd24, 5'd1, 5'd8, 16'd0 }   //store
MemInst[16] = { 6'd1, 5'd1, 5'd1, 16'd1 }    //addi
MemInst[17] = { 6'd1, 5'd1, 5'd1, 16'd1 }    //addi
MemInst[18] = { 6'd1, 5'd1, 5'd1, 16'd1 }    //addi
MemInst[19] = { 6'd23, 5'd2, 5'd10, 16'd5 }    //load
MemInst[20] = { 6'd23, 5'd2, 5'd11, 16'd1 }    //load
MemInst[21] = { 6'd16, 5'd11, 5'd10, 16'd0 }     //mov
MemInst[22] = { 6'd24, 5'd2, 5'd10, 16'd5 }   //store
MemInst[23] = { 6'd23, 5'd2, 5'd12, 16'd4 }    //load
MemInst[24] = { 6'd23, 5'd2, 5'd13, 16'd1 }    //load
MemInst[25] = { 6'd23, 5'd2, 5'd14, 16'd0 }    //load
MemInst[26] = { 6'd0, 5'd13, 5'd14, 5'd13, 11'd0 }     //add
MemInst[27] = { 6'd23, 5'd13, 5'd14, 16'd0 }    //load
MemInst[28] = { 6'd16, 5'd14, 5'd12, 16'd0 }     //mov
MemInst[29] = { 6'd24, 5'd2, 5'd12, 16'd4 }   //store
MemInst[30] = { 6'd23, 5'd2, 5'd15, 16'd3 }    //load
MemInst[31] = { 6'd23, 5'd2, 5'd16, 16'd1 }    //load
MemInst[32] = { 6'd17, 5'd0, 5'd17, 16'd1 }      //li
MemInst[33] = { 6'd0, 5'd18, 5'd16, 5'd17, 11'd0 }     //add
```

```
38  MemInst[34] = { 6'd16, 5'd18, 5'd15, 16'd0 }     //mov
39  MemInst[35] = { 6'd24, 5'd2, 5'd15, 16'd3 }   //store
40  //L0
41  MemInst[36] = { 6'd23, 5'd2, 5'd19, 16'd3 }    //load
42  MemInst[37] = { 6'd23, 5'd2, 5'd20, 16'd2 }    //load
43  MemInst[38] = { 6'd10, 5'd21, 5'd19, 5'd20, 11'd0 }     //slt
44  MemInst[39] = { 6'd18, 5'd21, 5'd0, 16'd66 }     //beq
45  MemInst[40] = { 6'd23, 5'd2, 5'd22, 16'd3 }    //load
46  MemInst[41] = { 6'd23, 5'd2, 5'd23, 16'd0 }    //load
47  MemInst[42] = { 6'd0, 5'd22, 5'd23, 5'd22, 11'd0 }     //add
48  MemInst[43] = { 6'd23, 5'd22, 5'd23, 16'd0 }    //load
49  MemInst[44] = { 6'd23, 5'd2, 5'd24, 16'd4 }    //load
50  MemInst[45] = { 6'd10, 5'd25, 5'd23, 5'd24, 11'd0 }     //slt
51  MemInst[46] = { 6'd18, 5'd25, 5'd0, 16'd59 }     //beq
52  MemInst[47] = { 6'd23, 5'd2, 5'd26, 16'd4 }    //load
53  MemInst[48] = { 6'd23, 5'd2, 5'd27, 16'd3 }    //load
54  MemInst[49] = { 6'd23, 5'd2, 5'd28, 16'd0 }    //load
55  MemInst[50] = { 6'd0, 5'd27, 5'd28, 5'd27, 11'd0 }     //add
56  MemInst[51] = { 6'd23, 5'd27, 5'd28, 16'd0 }    //load
57  MemInst[52] = { 6'd16, 5'd28, 5'd26, 16'd0 }     //mov
58  MemInst[53] = { 6'd24, 5'd2, 5'd26, 16'd4 }   //store
59  MemInst[54] = { 6'd23, 5'd2, 5'd29, 16'd5 }    //load
60  MemInst[55] = { 6'd23, 5'd2, 5'd30, 16'd3 }    //load
61  MemInst[56] = { 6'd16, 5'd30, 5'd29, 16'd0 }     //mov
62  MemInst[57] = { 6'd24, 5'd2, 5'd29, 16'd5 }   //store
63  MemInst[58] = { 6'd20, 26'd59 }                 //j
64  //L2
65  //L3
66  MemInst[59] = { 6'd23, 5'd2, 5'd31, 16'd3 }    //load
67  MemInst[60] = { 6'd23, 5'd2, 5'd10, 16'd3 }    //load
68  MemInst[61] = { 6'd17, 5'd0, 5'd11, 16'd1 }      //li
69  MemInst[62] = { 6'd0, 5'd12, 5'd10, 5'd11, 11'd0 }     //add
70  MemInst[63] = { 6'd16, 5'd12, 5'd31, 16'd0 }     //mov
71  MemInst[64] = { 6'd24, 5'd2, 5'd31, 16'd3 }   //store
72  MemInst[65] = { 6'd20, 26'd36 }                 //j
73  //L1
74  MemInst[66] = { 6'd23, 5'd2, 5'd13, 16'd5 }    //load
75  MemInst[67] = { 6'd16, 5'd13, 5'd5, 16'd0 }     //mov
76  MemInst[68] = { 6'd1, 5'd2, 5'd2, -16'd2 }    //addi
77  MemInst[69] = { 6'd23, 5'd2, 5'd4, 16'd1 }    //load
78  MemInst[70] = { 6'd16, 5'd2, 5'd1, 16'd0 }     //mov
79  MemInst[71] = { 6'd1, 5'd1, 5'd1, -16'd1 }    //addi
80  MemInst[72] = { 6'd23, 5'd2, 5'd2, 16'd0 }    //load
81  MemInst[73] = { 6'd25, 26'd4 }                //jr
82  MemInst[74] = { 6'd1, 5'd2, 5'd2, -16'd2 }    //addi
83  MemInst[75] = { 6'd23, 5'd2, 5'd4, 16'd1 }    //load
84  MemInst[76] = { 6'd16, 5'd2, 5'd1, 16'd0 }     //mov
85  MemInst[77] = { 6'd1, 5'd1, 5'd1, -16'd1 }    //addi
86  MemInst[78] = { 6'd23, 5'd2, 5'd2, 16'd0 }    //load
87  MemInst[79] = { 6'd25, 26'd4 }                //jr
88  //sort
89  MemInst[80] = { 6'd1, 5'd1, 5'd1, 16'd2 }    //addi
90  MemInst[81] = { 6'd24, 5'd1, 5'd4, 16'd0 }   //store
91  MemInst[82] = { 6'd24, 5'd1, 5'd2, -16'd1 }   //store
92  MemInst[83] = { 6'd16, 5'd1, 5'd2, 16'd0 }     //mov
93  MemInst[84] = { 6'd1, 5'd2, 5'd2, 16'd1 }    //addi
94  MemInst[85] = { 6'd1, 5'd1, 5'd1, 16'd1 }    //addi
95  MemInst[86] = { 6'd24, 5'd1, 5'd6, 16'd0 }   //store
96  MemInst[87] = { 6'd1, 5'd1, 5'd1, 16'd1 }    //addi
97  MemInst[88] = { 6'd24, 5'd1, 5'd7, 16'd0 }   //store
```

```
98  MemInst[89] = { 6'd1, 5'd1, 5'd1, 16'd1 }    //addi
99  MemInst[90] = { 6'd24, 5'd1, 5'd8, 16'd0 }   //store
100 MemInst[91] = { 6'd1, 5'd1, 5'd1, 16'd1 }    //addi
101 MemInst[92] = { 6'd1, 5'd1, 5'd1, 16'd1 }    //addi
102 MemInst[93] = { 6'd23, 5'd2, 5'd14, 16'd3 }    //load
103 MemInst[94] = { 6'd23, 5'd2, 5'd15, 16'd1 }    //load
104 MemInst[95] = { 6'd16, 5'd15, 5'd14, 16'd0 }     //mov
105 MemInst[96] = { 6'd24, 5'd2, 5'd14, 16'd3 }   //store
106 //L4
107 MemInst[97] = { 6'd23, 5'd2, 5'd16, 16'd3 }    //load
108 MemInst[98] = { 6'd23, 5'd2, 5'd17, 16'd2 }    //load
109 MemInst[99] = { 6'd17, 5'd0, 5'd18, 16'd1 }      //li
110 MemInst[100] = { 6'd2, 5'd19, 5'd17, 5'd18, 11'd0 }     //sub
111 MemInst[101] = { 6'd10, 5'd20, 5'd16, 5'd19, 11'd0 }     //slt
112 MemInst[102] = { 6'd18, 5'd20, 5'd0, 16'd146 }     //beq
113 MemInst[103] = { 6'd1, 5'd1, 5'd1, 16'd1 }    //addi
114 MemInst[104] = { 6'd23, 5'd2, 5'd21, 16'd4 }    //load
115 MemInst[105] = { 6'd23, 5'd2, 5'd22, 16'd0 }    //load
116 MemInst[106] = { 6'd16, 5'd22, 5'd6, 16'd0 }     //mov
117 MemInst[107] = { 6'd23, 5'd2, 5'd23, 16'd3 }    //load
118 MemInst[108] = { 6'd16, 5'd23, 5'd7, 16'd0 }     //mov
119 MemInst[109] = { 6'd23, 5'd2, 5'd24, 16'd2 }    //load
120 MemInst[110] = { 6'd16, 5'd24, 5'd8, 16'd0 }     //mov
121 MemInst[111] = { 6'd26, 26'd5 }               //jal
122 MemInst[112] = { 6'd16, 5'd5, 5'd25, 16'd0 }     //mov
123 MemInst[113] = { 6'd16, 5'd25, 5'd21, 16'd0 }     //mov
124 MemInst[114] = { 6'd24, 5'd2, 5'd21, 16'd4 }   //store
125 MemInst[115] = { 6'd23, 5'd2, 5'd26, 16'd5 }    //load
126 MemInst[116] = { 6'd23, 5'd2, 5'd27, 16'd4 }    //load
127 MemInst[117] = { 6'd23, 5'd2, 5'd28, 16'd0 }    //load
128 MemInst[118] = { 6'd0, 5'd27, 5'd28, 5'd27, 11'd0 }     //add
129 MemInst[119] = { 6'd23, 5'd27, 5'd28, 16'd0 }    //load
130 MemInst[120] = { 6'd16, 5'd28, 5'd26, 16'd0 }     //mov
131 MemInst[121] = { 6'd24, 5'd2, 5'd26, 16'd5 }   //store
132 MemInst[122] = { 6'd23, 5'd2, 5'd29, 16'd4 }    //load
133 MemInst[123] = { 6'd23, 5'd2, 5'd30, 16'd0 }    //load
134 MemInst[124] = { 6'd0, 5'd29, 5'd30, 5'd29, 11'd0 }     //add
135 MemInst[125] = { 6'd23, 5'd29, 5'd30, 16'd0 }    //load
136 MemInst[126] = { 6'd23, 5'd2, 5'd31, 16'd3 }    //load
137 MemInst[127] = { 6'd23, 5'd2, 5'd10, 16'd0 }    //load
138 MemInst[128] = { 6'd0, 5'd31, 5'd10, 5'd31, 11'd0 }     //add
139 MemInst[129] = { 6'd23, 5'd31, 5'd10, 16'd0 }    //load
140 MemInst[130] = { 6'd16, 5'd10, 5'd30, 16'd0 }     //mov
141 MemInst[131] = { 6'd24, 5'd29, 5'd30, 16'd0 }   //store
142 MemInst[132] = { 6'd23, 5'd2, 5'd11, 16'd3 }    //load
143 MemInst[133] = { 6'd23, 5'd2, 5'd12, 16'd0 }    //load
144 MemInst[134] = { 6'd0, 5'd11, 5'd12, 5'd11, 11'd0 }     //add
145 MemInst[135] = { 6'd23, 5'd11, 5'd12, 16'd0 }    //load
146 MemInst[136] = { 6'd23, 5'd2, 5'd13, 16'd5 }    //load
147 MemInst[137] = { 6'd16, 5'd13, 5'd12, 16'd0 }     //mov
148 MemInst[138] = { 6'd24, 5'd11, 5'd12, 16'd0 }   //store
149 MemInst[139] = { 6'd23, 5'd2, 5'd14, 16'd3 }    //load
150 MemInst[140] = { 6'd23, 5'd2, 5'd15, 16'd3 }    //load
151 MemInst[141] = { 6'd17, 5'd0, 5'd16, 16'd1 }      //li
152 MemInst[142] = { 6'd0, 5'd17, 5'd15, 5'd16, 11'd0 }     //add
153 MemInst[143] = { 6'd16, 5'd17, 5'd14, 16'd0 }     //mov
154 MemInst[144] = { 6'd24, 5'd2, 5'd14, 16'd3 }   //store
155 MemInst[145] = { 6'd20, 26'd97 }                 //j
156 //L5
157 MemInst[146] = { 6'd1, 5'd2, 5'd2, -16'd2 }    //addi
```

```
158 MemInst[147] = { 6'd23, 5'd2, 5'd4, 16'd1 }    //load
159 MemInst[148] = { 6'd16, 5'd2, 5'd1, 16'd0 }     //mov
160 MemInst[149] = { 6'd1, 5'd1, 5'd1, -16'd1 }    //addi
161 MemInst[150] = { 6'd23, 5'd2, 5'd2, 16'd0 }    //load
162 MemInst[151] = { 6'd25, 26'd4 }                //jr
163 //main
164 MemInst[152] = { 6'd1, 5'd1, 5'd1, 16'd1 }    //addi
165 MemInst[153] = { 6'd23, 5'd2, 5'd18, 16'd0 }    //load
166 MemInst[154] = { 6'd17, 5'd0, 5'd19, 16'd0 }      //li
167 MemInst[155] = { 6'd16, 5'd19, 5'd18, 16'd0 }     //mov
168 MemInst[156] = { 6'd24, 5'd2, 5'd18, 16'd0 }   //store
169 //L6
170 MemInst[157] = { 6'd23, 5'd2, 5'd20, 16'd0 }    //load
171 MemInst[158] = { 6'd17, 5'd0, 5'd21, 16'd10 }      //li
172 MemInst[159] = { 6'd10, 5'd22, 5'd20, 5'd21, 11'd0 }     //slt
173 MemInst[160] = { 6'd18, 5'd22, 5'd0, 16'd175 }     //beq
174 MemInst[161] = { 6'd23, 5'd2, 5'd23, 16'd0 }    //load
175 MemInst[162] = { 6'd0, 5'd23, 5'd3, 5'd23, 11'd0 }     //add
176 MemInst[163] = { 6'd1, 5'd23, 5'd23, 16'd0 }    //addi
177 MemInst[164] = { 6'd23, 5'd23, 5'd24, 16'd0 }    //load
178 MemInst[165] = { 6'd21, 26'd25 }                //in
179 MemInst[166] = { 6'd16, 5'd25, 5'd24, 16'd0 }     //mov
180 MemInst[167] = { 6'd24, 5'd23, 5'd24, 16'd0 }   //store
181 MemInst[168] = { 6'd23, 5'd2, 5'd26, 16'd0 }    //load
182 MemInst[169] = { 6'd23, 5'd2, 5'd27, 16'd0 }    //load
183 MemInst[170] = { 6'd17, 5'd0, 5'd28, 16'd1 }      //li
184 MemInst[171] = { 6'd0, 5'd29, 5'd27, 5'd28, 11'd0 }     //add
185 MemInst[172] = { 6'd16, 5'd29, 5'd26, 16'd0 }     //mov
186 MemInst[173] = { 6'd24, 5'd2, 5'd26, 16'd0 }   //store
187 MemInst[174] = { 6'd20, 26'd157 }                 //j
188 //L7
189 MemInst[175] = { 6'd1, 5'd3, 5'd30, 16'd0 }    //addi
190 MemInst[176] = { 6'd16, 5'd30, 5'd6, 16'd0 }     //mov
191 MemInst[177] = { 6'd17, 5'd0, 5'd31, 16'd0 }      //li
192 MemInst[178] = { 6'd16, 5'd31, 5'd7, 16'd0 }     //mov
193 MemInst[179] = { 6'd17, 5'd0, 5'd10, 16'd10 }      //li
194 MemInst[180] = { 6'd16, 5'd10, 5'd8, 16'd0 }     //mov
195 MemInst[181] = { 6'd26, 26'd80 }               //jal
196 MemInst[182] = { 6'd16, 5'd5, 5'd11, 16'd0 }     //mov
197 MemInst[183] = { 6'd23, 5'd2, 5'd12, 16'd0 }    //load
198 MemInst[184] = { 6'd17, 5'd0, 5'd13, 16'd0 }      //li
199 MemInst[185] = { 6'd16, 5'd13, 5'd12, 16'd0 }     //mov
200 MemInst[186] = { 6'd24, 5'd2, 5'd12, 16'd0 }   //store
201 //L8
202 MemInst[187] = { 6'd23, 5'd2, 5'd14, 16'd0 }    //load
203 MemInst[188] = { 6'd17, 5'd0, 5'd15, 16'd10 }      //li
204 MemInst[189] = { 6'd10, 5'd16, 5'd14, 5'd15, 11'd0 }     //slt
205 MemInst[190] = { 6'd18, 5'd16, 5'd0, 16'd205 }     //beq
206 MemInst[191] = { 6'd23, 5'd2, 5'd17, 16'd0 }    //load
207 MemInst[192] = { 6'd0, 5'd17, 5'd3, 5'd17, 11'd0 }     //add
208 MemInst[193] = { 6'd1, 5'd17, 5'd17, 16'd0 }    //addi
209 MemInst[194] = { 6'd23, 5'd17, 5'd18, 16'd0 }    //load
210 MemInst[195] = { 6'd16, 5'd18, 5'd6, 16'd0 }     //mov
211 MemInst[196] = { 6'd16, 5'd6, 5'd19, 16'd0 }     //mov
212 MemInst[197] = { 6'd22, 26'd19 }               //out
213 MemInst[198] = { 6'd23, 5'd2, 5'd20, 16'd0 }    //load
214 MemInst[199] = { 6'd23, 5'd2, 5'd21, 16'd0 }    //load
215 MemInst[200] = { 6'd17, 5'd0, 5'd22, 16'd1 }      //li
216 MemInst[201] = { 6'd0, 5'd23, 5'd21, 5'd22, 11'd0 }     //add
217 MemInst[202] = { 6'd16, 5'd23, 5'd20, 16'd0 }     //mov
```

```
MemInst[203] = { 6'd24, 5'd2, 5'd20, 16'd0 }   //store
MemInst[204] = { 6'd20, 26'd187 }                 //j
//L9
MemInst[205] = { 6'd20, 26'd206 }                 //j
//end
MemInst[206] = { 6'd27, 26'd0 }              //halt
```

Listing 5.8 – Código Executável - Sort. Fonte: Autor.

5.3 Fatorial

5.3.1 Código em C-

```
/* Fatorial recursivo */

int fatorial(int n){

    if(n <= 1){
        return(1);
    }
    else{
        return(n * fatorial(n - 1));
    }
}

void main(void){
    int n;
    n = input();
    output(fatorial(n));
}
```

Listing 5.9 – Código C- Fatorial, Fonte: Autor.

5.3.2 Código Intermediário

```
..::Intermediate Code for ZAFx32 Processor::..

(FUNC, fatorial, -, 1)
(PARAM, n, -, fatorial)
(LOAD, $t0, n, -)
(IMME, $t1, 1, -)
(LET, $t2, $t0, $t1)
(IFF, $t2, L0, -)
(IMME, $t3, 1, -)
(RET, $t3, -, -)
(GOTO, L1, -, -)
(LAB, L0, -, -)
(LOAD, $t4, n, -)
(LOAD, $t5, n, -)
(IMME, $t6, 1, -)
(SUB, $t7, $t5, $t6)
(ARG, $t7, -, -)
(CALL, $t8, fatorial, 1)
(MUL, $t9, $t4, $t8)
(RET, $t9, -, -)
(LAB, L1, -, -)
(END, fatorial, -, -)
```

```
(FUNC, main, -, 1)
(ALLOC, n, 1, main)
(LOAD, $t10, n, -)
(CALL, $t11, input, 0)
(ASSIGN, $t10, $t11, -)
(STORE, n, $t10, -)
(LOAD, $t12, n, -)
(ARG, $t12, -, -)
(CALL, $t13, fatorial, 1)
(ARG, $t13, -, -)
(CALL, $t14, output, 1)
(END, main, -, -)
(HALT, -, -, -)
```

Listing 5.10 – Código Intermediário - Fatorial. Fonte: Autor.

5.3.3 Código Assembly

```
..::Assembly Code for ZAFx32 Processor::..

0:          li sp, -1
1:          li fp, 0
2:          li zero, 0
3:          li gp, 249
4:          j main
.fatorial
5:          addi sp, sp, 2
6:          store ra, sp, 0
7:          store fp, sp, -1
8:          mov fp, sp
9:          addi fp, fp, 1
10:         addi sp, sp, 1
11:         store a0, sp, 0
12:         load t0, fp, 0
13:         li t1, 1
14:         slet t2, t0, t1
15:         beq t2, zero, L0
16:         li t3, 1
17:         mov ret, t3
18:         addi fp, fp, -2
19:         load ra, fp, 1
20:         mov sp, fp
21:         addi sp, sp, -1
22:         load fp, fp, 0
23:         jr ra
24:         j L1
.L0
25:         load t4, fp, 0
26:         load t5, fp, 0
27:         li t6, 1
28:         sub t7, t5, t6
29:         mov a0, t7
30:         jal fatorial
31:         mov t8, ret
32:         mul t9, t4, t8
33:         mov ret, t9
34:         addi fp, fp, -2
35:         load ra, fp, 1
36:         mov sp, fp
```

```
37:         addi sp, sp, -1
38:         load fp, fp, 0
39:         jr ra
.L1
40:         addi fp, fp, -2
41:         load ra, fp, 1
42:         mov sp, fp
43:         addi sp, sp, -1
44:         load fp, fp, 0
45:         jr ra
.main
46:         addi sp, sp, 1
47:         load t10, fp, 0
48:         in t11
49:         mov t10, t11
50:         store t10, fp, 0
51:         load t12, fp, 0
52:         mov a0, t12
53:         jal fatorial
54:         mov t13, ret
55:         mov a0, t13
56:         mov t14, a0
57:         out t14
58:         j end
.end
59:         halt
```

Listing 5.11 – Código Assembly - Fatorial. Fonte: Autor.

5.3.4 Código Executável

```
..::Binary Code for ZAFx32 Processor::..

MemInst[0] = { 6'd17, 5'd0, 5'd1, 16'd-1 }      //li
MemInst[1] = { 6'd17, 5'd0, 5'd2, 16'd0 }      //li
MemInst[2] = { 6'd17, 5'd0, 5'd0, 16'd0 }      //li
MemInst[3] = { 6'd17, 5'd0, 5'd3, 16'd249 }      //li
MemInst[4] = { 6'd20, 26'd46 }                  //j
//fatorial
MemInst[5] = { 6'd1, 5'd1, 5'd1, 16'd2 }    //addi
MemInst[6] = { 6'd24, 5'd1, 5'd4, 16'd0 }   //store
MemInst[7] = { 6'd24, 5'd1, 5'd2, -16'd1 }   //store
MemInst[8] = { 6'd16, 5'd1, 5'd2, 16'd0 }     //mov
MemInst[9] = { 6'd1, 5'd2, 5'd2, 16'd1 }    //addi
MemInst[10] = { 6'd1, 5'd1, 5'd1, 16'd1 }    //addi
MemInst[11] = { 6'd24, 5'd1, 5'd6, 16'd0 }   //store
MemInst[12] = { 6'd23, 5'd2, 5'd10, 16'd0 }    //load
MemInst[13] = { 6'd17, 5'd0, 5'd11, 16'd1 }      //li
MemInst[14] = { 6'd12, 5'd12, 5'd10, 5'd11, 11'd0 }    //slet
MemInst[15] = { 6'd18, 5'd12, 5'd0, 16'd25 }     //beq
MemInst[16] = { 6'd17, 5'd0, 5'd13, 16'd1 }      //li
MemInst[17] = { 6'd16, 5'd13, 5'd5, 16'd0 }     //mov
MemInst[18] = { 6'd1, 5'd2, 5'd2, -16'd2 }    //addi
MemInst[19] = { 6'd23, 5'd2, 5'd4, 16'd1 }    //load
MemInst[20] = { 6'd16, 5'd2, 5'd1, 16'd0 }     //mov
MemInst[21] = { 6'd1, 5'd1, 5'd1, -16'd1 }    //addi
MemInst[22] = { 6'd23, 5'd2, 5'd2, 16'd0 }    //load
MemInst[23] = { 6'd25, 26'd4 }                 //jr
MemInst[24] = { 6'd20, 26'd40 }                  //j
```

```
//L0
MemInst[25] = { 6'd23, 5'd2, 5'd14, 16'd0 }    //load
MemInst[26] = { 6'd23, 5'd2, 5'd15, 16'd0 }    //load
MemInst[27] = { 6'd17, 5'd0, 5'd16, 16'd1 }      //li
MemInst[28] = { 6'd2, 5'd17, 5'd15, 5'd16, 11'd0 }     //sub
MemInst[29] = { 6'd16, 5'd17, 5'd6, 16'd0 }     //mov
MemInst[30] = { 6'd26, 26'd5 }               //jal
MemInst[31] = { 6'd16, 5'd5, 5'd18, 16'd0 }     //mov
MemInst[32] = { 6'd3, 5'd19, 5'd14, 5'd18, 11'd0 }     //mul
MemInst[33] = { 6'd16, 5'd19, 5'd5, 16'd0 }     //mov
MemInst[34] = { 6'd1, 5'd2, 5'd2, -16'd2 }    //addi
MemInst[35] = { 6'd23, 5'd2, 5'd4, 16'd1 }    //load
MemInst[36] = { 6'd16, 5'd2, 5'd1, 16'd0 }     //mov
MemInst[37] = { 6'd1, 5'd1, 5'd1, -16'd1 }    //addi
MemInst[38] = { 6'd23, 5'd2, 5'd2, 16'd0 }    //load
MemInst[39] = { 6'd25, 26'd4 }                //jr
//L1
MemInst[40] = { 6'd1, 5'd2, 5'd2, -16'd2 }    //addi
MemInst[41] = { 6'd23, 5'd2, 5'd4, 16'd1 }    //load
MemInst[42] = { 6'd16, 5'd2, 5'd1, 16'd0 }     //mov
MemInst[43] = { 6'd1, 5'd1, 5'd1, -16'd1 }    //addi
MemInst[44] = { 6'd23, 5'd2, 5'd2, 16'd0 }    //load
MemInst[45] = { 6'd25, 26'd4 }                //jr
//main
MemInst[46] = { 6'd1, 5'd1, 5'd1, 16'd1 }    //addi
MemInst[47] = { 6'd23, 5'd2, 5'd20, 16'd0 }    //load
MemInst[48] = { 6'd21, 26'd21 }                //in
MemInst[49] = { 6'd16, 5'd21, 5'd20, 16'd0 }     //mov
MemInst[50] = { 6'd24, 5'd2, 5'd20, 16'd0 }   //store
MemInst[51] = { 6'd23, 5'd2, 5'd22, 16'd0 }    //load
MemInst[52] = { 6'd16, 5'd22, 5'd6, 16'd0 }     //mov
MemInst[53] = { 6'd26, 26'd5 }               //jal
MemInst[54] = { 6'd16, 5'd5, 5'd23, 16'd0 }     //mov
MemInst[55] = { 6'd16, 5'd23, 5'd6, 16'd0 }     //mov
MemInst[56] = { 6'd16, 5'd6, 5'd24, 16'd0 }     //mov
MemInst[57] = { 6'd22, 26'd24 }               //out
MemInst[58] = { 6'd20, 26'd59 }                 //j
//end
MemInst[59] = { 6'd27, 26'd0 }              //halt
```

Listing 5.12 – Código Executável - Fatorial. Fonte: Autor.

5.4 Fibonacci

5.4.1 Código em C-

```
/* Calculo do valor de finonacci */

int fibo(int n){
    int c;
    int next;
    int first;
    int second;

    c = 0;

    while(c <= n){
        if(c <= 1){
```

```
13              next = c;
14          }
15          else{
16              next = first + second;
17              first = second;
18              second = next;
19          }
20          c = c + 1;
21      }
22      return next;
23  }
24
25  void main(void){
26      int n;
27
28      n = input();
29      output(fibo(n));
30  }
```

Listing 5.13 – Código C- Fibonacci, Fonte: Autor.

5.4.2 Código Intermediário

```
1   ..::Intermediate Code for ZAFx32 Processor::..
2
3   (FUNC, fibo, -, 1)
4   (PARAM, n, -, fibo)
5   (ALLOC, c, 1, fibo)
6   (ALLOC, next, 1, fibo)
7   (ALLOC, first, 1, fibo)
8   (ALLOC, second, 1, fibo)
9   (LOAD, $t0, c, -)
10  (IMME, $t1, 0, -)
11  (ASSIGN, $t0, $t1, -)
12  (STORE, c, $t0, -)
13  (LAB, L0, -, -)
14  (LOAD, $t2, c, -)
15  (LOAD, $t3, n, -)
16  (LET, $t4, $t2, $t3)
17  (IFF, $t4, L1, -)
18  (LOAD, $t5, c, -)
19  (IMME, $t6, 1, -)
20  (LET, $t7, $t5, $t6)
21  (IFF, $t7, L2, -)
22  (LOAD, $t8, next, -)
23  (LOAD, $t9, c, -)
24  (ASSIGN, $t8, $t9, -)
25  (STORE, next, $t8, -)
26  (GOTO, L3, -, -)
27  (LAB, L2, -, -)
28  (LOAD, $t10, next, -)
29  (LOAD, $t11, first, -)
30  (LOAD, $t12, second, -)
31  (ADD, $t13, $t11, $t12)
32  (ASSIGN, $t10, $t13, -)
33  (STORE, next, $t10, -)
34  (LOAD, $t14, first, -)
35  (LOAD, $t15, second, -)
36  (ASSIGN, $t14, $t15, -)
```

```
(STORE, first, $t14, -)
(LOAD, $t16, second, -)
(LOAD, $t17, next, -)
(ASSIGN, $t16, $t17, -)
(STORE, second, $t16, -)
(LAB, L3, -, -)
(LOAD, $t18, c, -)
(LOAD, $t19, c, -)
(IMME, $t20, 1, -)
(ADD, $t21, $t19, $t20)
(ASSIGN, $t18, $t21, -)
(STORE, c, $t18, -)
(GOTO, L0, -, -)
(LAB, L1, -, -)
(LOAD, $t0, next, -)
(RET, $t0, -, -)
(END, fibo, -, -)
(FUNC, main, -, 1)
(ALLOC, n, 1, main)
(LOAD, $t1, n, -)
(CALL, $t2, input, 0)
(ASSIGN, $t1, $t2, -)
(STORE, n, $t1, -)
(LOAD, $t3, n, -)
(ARG, $t3, -, -)
(CALL, $t4, fibo, 1)
(ARG, $t4, -, -)
(CALL, $t5, output, 1)
(END, main, -, -)
(HALT, -, -, -)
```

Listing 5.14 – Código Intermediário - Fibonacci. Fonte: Autor.

5.4.3 Código Assembly

```
..::Assembly Code for ZAFx32 Processor::..

0:          li sp, -1
1:          li fp, 0
2:          li zero, 0
3:          li gp, 249
4:          j main
.fibo
5:          addi sp, sp, 2
6:          store ra, sp, 0
7:          store fp, sp, -1
8:          mov fp, sp
9:          addi fp, fp, 1
10:         addi sp, sp, 1
11:         store a0, sp, 0
12:         addi sp, sp, 1
13:         addi sp, sp, 1
14:         addi sp, sp, 1
15:         addi sp, sp, 1
16:         load t0, fp, 1
17:         li t1, 0
18:         mov t0, t1
19:         store t0, fp, 1
.L0
```

```
20:         load t2, fp, 1
21:         load t3, fp, 0
22:         slet t4, t2, t3
23:         beq t4, zero, L1
24:         load t5, fp, 1
25:         li t6, 1
26:         slet t7, t5, t6
27:         beq t7, zero, L2
28:         load t8, fp, 2
29:         load t9, fp, 1
30:         mov t8, t9
31:         store t8, fp, 2
32:         j L3
.L2
33:         load t10, fp, 2
34:         load t11, fp, 3
35:         load t12, fp, 4
36:         add t13, t11, t12
37:         mov t10, t13
38:         store t10, fp, 2
39:         load t14, fp, 3
40:         load t15, fp, 4
41:         mov t14, t15
42:         store t14, fp, 3
43:         load t16, fp, 4
44:         load t17, fp, 2
45:         mov t16, t17
46:         store t16, fp, 4
.L3
47:         load t18, fp, 1
48:         load t19, fp, 1
49:         li t20, 1
50:         add t21, t19, t20
51:         mov t18, t21
52:         store t18, fp, 1
53:         j L0
.L1
54:         load t0, fp, 2
55:         mov ret, t0
56:         addi fp, fp, -2
57:         load ra, fp, 1
58:         mov sp, fp
59:         addi sp, sp, -1
60:         load fp, fp, 0
61:         jr ra
62:         addi fp, fp, -2
63:         load ra, fp, 1
64:         mov sp, fp
65:         addi sp, sp, -1
66:         load fp, fp, 0
67:         jr ra
.main
68:         addi sp, sp, 1
69:         load t1, fp, 0
70:         in t2
71:         mov t1, t2
72:         store t1, fp, 0
73:         load t3, fp, 0
74:         mov a0, t3
75:         jal fibo
```

```
76:         mov t4, ret
77:         mov a0, t4
78:         mov t5, a0
79:         out t5
80:         j end
.end
81:         halt
```

Listing 5.15 – Código Assembly - Fibonacci. Fonte: Autor.

5.4.4 Código Executável

```
..::Binary Code for ZAFx32 Processor::..

MemInst[0] = { 6'd17, 5'd0, 5'd1, 16'd-1 }      //li
MemInst[1] = { 6'd17, 5'd0, 5'd2, 16'd0 }       //li
MemInst[2] = { 6'd17, 5'd0, 5'd0, 16'd0 }       //li
MemInst[3] = { 6'd17, 5'd0, 5'd3, 16'd249 }      //li
MemInst[4] = { 6'd20, 26'd68 }                  //j
//fibo
MemInst[5] = { 6'd1, 5'd1, 5'd1, 16'd2 }    //addi
MemInst[6] = { 6'd24, 5'd1, 5'd4, 16'd0 }   //store
MemInst[7] = { 6'd24, 5'd1, 5'd2, -16'd1 }   //store
MemInst[8] = { 6'd16, 5'd1, 5'd2, 16'd0 }     //mov
MemInst[9] = { 6'd1, 5'd2, 5'd2, 16'd1 }    //addi
MemInst[10] = { 6'd1, 5'd1, 5'd1, 16'd1 }    //addi
MemInst[11] = { 6'd24, 5'd1, 5'd6, 16'd0 }   //store
MemInst[12] = { 6'd1, 5'd1, 5'd1, 16'd1 }    //addi
MemInst[13] = { 6'd1, 5'd1, 5'd1, 16'd1 }    //addi
MemInst[14] = { 6'd1, 5'd1, 5'd1, 16'd1 }    //addi
MemInst[15] = { 6'd1, 5'd1, 5'd1, 16'd1 }    //addi
MemInst[16] = { 6'd23, 5'd2, 5'd10, 16'd1 }    //load
MemInst[17] = { 6'd17, 5'd0, 5'd11, 16'd0 }      //li
MemInst[18] = { 6'd16, 5'd11, 5'd10, 16'd0 }     //mov
MemInst[19] = { 6'd24, 5'd2, 5'd10, 16'd1 }   //store
//L0
MemInst[20] = { 6'd23, 5'd2, 5'd12, 16'd1 }    //load
MemInst[21] = { 6'd23, 5'd2, 5'd13, 16'd0 }    //load
MemInst[22] = { 6'd12, 5'd14, 5'd12, 5'd13, 11'd0 }    //slet
MemInst[23] = { 6'd18, 5'd14, 5'd0, 16'd54 }     //beq
MemInst[24] = { 6'd23, 5'd2, 5'd15, 16'd1 }    //load
MemInst[25] = { 6'd17, 5'd0, 5'd16, 16'd1 }      //li
MemInst[26] = { 6'd12, 5'd17, 5'd15, 5'd16, 11'd0 }    //slet
MemInst[27] = { 6'd18, 5'd17, 5'd0, 16'd33 }     //beq
MemInst[28] = { 6'd23, 5'd2, 5'd18, 16'd2 }    //load
MemInst[29] = { 6'd23, 5'd2, 5'd19, 16'd1 }    //load
MemInst[30] = { 6'd16, 5'd19, 5'd18, 16'd0 }     //mov
MemInst[31] = { 6'd24, 5'd2, 5'd18, 16'd2 }   //store
MemInst[32] = { 6'd20, 26'd47 }                 //j
//L2
MemInst[33] = { 6'd23, 5'd2, 5'd20, 16'd2 }    //load
MemInst[34] = { 6'd23, 5'd2, 5'd21, 16'd3 }    //load
MemInst[35] = { 6'd23, 5'd2, 5'd22, 16'd4 }    //load
MemInst[36] = { 6'd0, 5'd23, 5'd21, 5'd22, 11'd0 }     //add
MemInst[37] = { 6'd16, 5'd23, 5'd20, 16'd0 }     //mov
MemInst[38] = { 6'd24, 5'd2, 5'd20, 16'd2 }   //store
MemInst[39] = { 6'd23, 5'd2, 5'd24, 16'd3 }    //load
MemInst[40] = { 6'd23, 5'd2, 5'd25, 16'd4 }    //load
MemInst[41] = { 6'd16, 5'd25, 5'd24, 16'd0 }     //mov
```

```
48 MemInst[42] = { 6'd24, 5'd2, 5'd24, 16'd3 }   //store
49 MemInst[43] = { 6'd23, 5'd2, 5'd26, 16'd4 }    //load
50 MemInst[44] = { 6'd23, 5'd2, 5'd27, 16'd2 }    //load
51 MemInst[45] = { 6'd16, 5'd27, 5'd26, 16'd0 }     //mov
52 MemInst[46] = { 6'd24, 5'd2, 5'd26, 16'd4 }   //store
53 //L3
54 MemInst[47] = { 6'd23, 5'd2, 5'd28, 16'd1 }    //load
55 MemInst[48] = { 6'd23, 5'd2, 5'd29, 16'd1 }    //load
56 MemInst[49] = { 6'd17, 5'd0, 5'd30, 16'd1 }      //li
57 MemInst[50] = { 6'd0, 5'd31, 5'd29, 5'd30, 11'd0 }     //add
58 MemInst[51] = { 6'd16, 5'd31, 5'd28, 16'd0 }     //mov
59 MemInst[52] = { 6'd24, 5'd2, 5'd28, 16'd1 }   //store
60 MemInst[53] = { 6'd20, 26'd20 }                  //j
61 //L1
62 MemInst[54] = { 6'd23, 5'd2, 5'd10, 16'd2 }    //load
63 MemInst[55] = { 6'd16, 5'd10, 5'd5, 16'd0 }     //mov
64 MemInst[56] = { 6'd1, 5'd2, 5'd2, -16'd2 }    //addi
65 MemInst[57] = { 6'd23, 5'd2, 5'd4, 16'd1 }    //load
66 MemInst[58] = { 6'd16, 5'd2, 5'd1, 16'd0 }     //mov
67 MemInst[59] = { 6'd1, 5'd1, 5'd1, -16'd1 }    //addi
68 MemInst[60] = { 6'd23, 5'd2, 5'd2, 16'd0 }    //load
69 MemInst[61] = { 6'd25, 26'd4 }                 //jr
70 MemInst[62] = { 6'd1, 5'd2, 5'd2, -16'd2 }    //addi
71 MemInst[63] = { 6'd23, 5'd2, 5'd4, 16'd1 }    //load
72 MemInst[64] = { 6'd16, 5'd2, 5'd1, 16'd0 }     //mov
73 MemInst[65] = { 6'd1, 5'd1, 5'd1, -16'd1 }    //addi
74 MemInst[66] = { 6'd23, 5'd2, 5'd2, 16'd0 }    //load
75 MemInst[67] = { 6'd25, 26'd4 }                 //jr
76 //main
77 MemInst[68] = { 6'd1, 5'd1, 5'd1, 16'd1 }    //addi
78 MemInst[69] = { 6'd23, 5'd2, 5'd11, 16'd0 }    //load
79 MemInst[70] = { 6'd21, 26'd12 }                //in
80 MemInst[71] = { 6'd16, 5'd12, 5'd11, 16'd0 }     //mov
81 MemInst[72] = { 6'd24, 5'd2, 5'd11, 16'd0 }   //store
82 MemInst[73] = { 6'd23, 5'd2, 5'd13, 16'd0 }    //load
83 MemInst[74] = { 6'd16, 5'd13, 5'd6, 16'd0 }     //mov
84 MemInst[75] = { 6'd26, 26'd5 }                //jal
85 MemInst[76] = { 6'd16, 5'd5, 5'd14, 16'd0 }     //mov
86 MemInst[77] = { 6'd16, 5'd14, 5'd6, 16'd0 }     //mov
87 MemInst[78] = { 6'd16, 5'd6, 5'd15, 16'd0 }     //mov
88 MemInst[79] = { 6'd22, 26'd15 }               //out
89 MemInst[80] = { 6'd20, 26'd81 }                  //j
90 //end
91 MemInst[81] = { 6'd27, 26'd0 }              //halt
```

Listing 5.16 – Código Executável - Fibonacci. Fonte: Autor.

6 Conclusão

O projeto desenvolvido visou as implementações de todas as unidades especificadas para as fases de análise e síntese do compilador C- proposto em (1), seguindo com rigor todas as definições estabelecidas para os tokens reconhecidos e para a gramática adotada.

Os principais desafios apresentados pelo projeto foram, na geração do código Intermediário onde o processo de se percorrer a árvore sintática e a partir das informações presentes conseguir de forma correta gerar as quadruplas correspondentes junto dos valores corretos utilizados de variáveis temporárias; na geração do código Assembly onde o mapeamento e gerenciamento de memória tiveram que ser pensados com muita cautela para todos os casos buscando atender a demanda do código fonte de maneira correta com suporte a recursão e utilização de vetores passados por referência para funções.

O projeto e as dificuldades foram de grande proveito para o aprendizado do discente visto que ajudaram a fixação de muitos conceitos relacionados ao processo de compilação (tanto na fase de análise quanto de síntese). Um outro ponto positivo foi o aprendizado de novas técnicas de programação envolvendo linguagem C e Python.

Futuras melhorias podem ser realizadas na adição de novas funcionalidades suportadas pela linguagem (que poderão ser vantajosas para as próximas disciplinas relacionadas aos Laboratórios de Sistemas Computacionais da grade curricular de Engenharia de Computação da Universidade Federal de São Paulo) e pela otimização do código Assembly gerado, possibilitando assim realizar tarefas utilizando menos linhas de código.

Referências

1 LOUDEN, K. C. *COMPILADORES – PRINCÍPIOS E PRÁTICAS*. [S.l.]: Pioneira Thomson Learning, 2004. Citado 5 vezes nas páginas 2, 7, 22, 29 e 65.

2 AHO MONICA S. LAM, R. S. J. D. U. A. V. *Compiladores: Princípios, Técnicas e Ferramentas*. [S.l.]: Pearson, 2008. Citado na página 7.