Júlia Wotzasek Pereira

Implementação do Compilador C-

São José dos Campos - Brasil

Outubro de 2020

Júlia Wotzasek Pereira

Implementação do Compilador C-

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

Docente: Prof. Dr. Luiz Eduardo Galvão Martins

Universidade Federal de São Paulo - UNIFESP

Instituto de Ciência e Tecnologia - Campus São José dos Campos

São José dos Campos - Brasil

Outubro de 2020

Sumário

1 INTRODUÇÃO . . . 1

2 O PROCESSADOR . . . 3

2.1 Requisitos do Processador . . . 3

2.1.1 Parte I - Estrutura Principal . . . 3

2.1.2 Parte II - Estruturas de Pilha e de *Buffer* . . . 5

2.1.3 Parte III - Recursividade, Conjunto de Instruções e Troca de Contexto . . . 6

2.1.4 Parte IV - *Hardware* . . . 7

2.2 Componentes do Processador . . . 7

2.3 Conjunto de Instruções . . . 10

2.4 Organização da Memória . . . 12

3 COMPILADOR: FASE DE ANÁLISE . . . 15

3.1 Diagrama de Blocos . . . 15

3.2 Diagrama de Atividades . . . 16

3.3 Análise Léxica . . . 18

3.3.1 Palavras Reservadas . . . 18

3.3.2 Símbolos Especiais . . . 18

3.3.3 Marcadores de Número e de Identificador de Variável . . . 18

3.3.4 Espaço em Branco . . . 18

3.3.5 Comentários . . . 18

3.4 Análise Sintática . . . 19

3.4.1 Regras Gramaticais . . . 19

3.5 Análise Semântica . . . 20

3.5.1 Erros relacionados à declarações de funções e variáveis . . . 20

3.5.2 Erros relacionados à indexação . . . 20

3.5.3 Erros relacionados à tipos . . . 20

3.5.4 Erros relacionados à função *main* . . . 21

3.6 Resultados Gerados pela Fase de Análise . . . 21

4 COMPILADOR: FASE DE SÍNTESE . . . 23

4.1 Geração de Código Intermediário . . . 23

4.1.1 Modelagem . . . 23

4.1.1.1 Diagrama de Blocos . . . 23

4.1.1.2 Diagramas de Atividades . . . 24

4.1.1.2.1 Geração de quádruplas para nós do tipo *While* . . . 25

4.1.1.2.2 Geração de quádruplas para nós do tipo *if* . . . 26
4.1.1.2.3 Geração de quádruplas para nós do tipo definição de função . . . 28
4.1.1.2.4 Geração de quádruplas para nós do tipo simétrico . . . 29
4.1.1.2.5 Geração de quádruplas para nós do tipo chamada de função . . . 30
4.1.1.2.6 Geração de quádruplas para nós do tipo chamada de função outros nós . . . 31
4.1.2 Quádruplas Utilizadas . . . 32
4.2 Geração do Código *Assembly* . . . **33**
4.2.1 Modelagem . . . 33
4.2.1.1 Diagrama de Blocos . . . 33
4.2.1.2 Diagramas de Atividades . . . 34
4.2.2 Funções de Geração de Código *Assembly* Específicas . . . 36
4.2.2.1 Geração de *Assembly* para Quádruplas de Declaração e Chamada de Funções . . . 36
4.2.2.1.1 Quádrupla *FUN* . . . 36
4.2.2.1.2 Quádrupla *ARG* . . . 36
4.2.2.1.3 Quádrupla *CALL* . . . 37
4.2.2.1.4 Quádrupla *PARAM* . . . 37
4.2.2.1.5 Quádrupla *RET* . . . 38
4.2.2.1.6 Quádrupla *END* . . . 38
4.2.2.2 Geração de *Assembly* para Quádruplas de Leitura/Escrita e Declaração de Variáveis 38
4.2.2.2.1 Quádrupla *ALLOC* . . . 38
4.2.2.2.2 Quádrupla *LOAD* . . . 39
4.2.2.2.3 Quádrupla *STORE* . . . 39
4.2.2.2.4 Quádrupla *ASSIGN* . . . 39
4.2.2.3 Geração de *Assembly* para Quádruplas de Controle . . . 40
4.2.2.3.1 Quádrupla *HALT* . . . 40
4.2.2.3.2 Quádrupla *LABEL* . . . 40
4.2.2.3.3 Quádrupla *GOTO* . . . 40
4.2.2.3.4 Quádrupla *IFF* . . . 40
4.2.2.4 Geração de *Assembly* para Quádruplas Lógico-Aritméticas . . . 41
4.2.2.4.1 Quádruplas *ADD*, *SUB*, *MULT*, *DIV* . . . 41
4.2.2.4.2 Quádrupla *GT* . . . 41
4.2.2.4.3 Quádrupla *LT* . . . 42
4.2.2.4.4 Quádrupla *GEQ* . . . 43
4.2.2.4.5 Quádrupla *LEQ* . . . 44
4.2.2.4.6 Quádrupla *EQ* . . . 45
4.2.2.4.7 Quádrupla *NEQ* . . . 45
4.3 Geração do Código Executável (Binário) . . . **47**
4.3.1 Modelagem . . . 47
4.3.1.1 Diagrama de Blocos . . . 47

4.3.1.2 Diagrama de Atividades . . . 48
4.4 Gerenciamento de Memória . . . 50

5 EXEMPLOS . . . 51
5.1 Exemplo 1 - Fatorial . . . 51
5.1.1 Código Fonte . . . 51
5.1.2 Código Intermediário . . . 51
5.1.3 Código *Assembly* . . . 52
5.1.4 Código executável . . . 54
5.1.5 Relação entre os códigos . . . 55
5.2 Exemplo 2 - Sort . . . 58
5.2.1 Código Fonte . . . 58
5.2.2 Código Intermediário . . . 58
5.2.3 Códigos *Assembly* e Executável . . . 61
5.3 Exemplo 3 - Gcd . . . 70
5.3.1 Código Fonte . . . 70
5.3.2 Código Intermediário . . . 71
5.3.3 Códigos *Assembly* e Executável . . . 72

6 CONCLUSÃO . . . 75

REFERÊNCIAS . . . 77

APÊNDICES 79

APÊNDICE A – QUÁDRUPLAS PARA DECLARAÇÃO E CHAMADA DE FUNÇÕES . . . 81
A.1 Definição de Funções . . . 81
A.1.1 Aplicação . . . 81
A.1.2 Formato . . . 81
A.1.3 Parâmetros . . . 81
A.1.4 Exemplo . . . 81
A.2 Argumento de Funções . . . 82
A.2.1 Aplicação . . . 82
A.2.2 Formato . . . 82
A.2.3 Parâmetros . . . 82
A.2.4 Exemplo . . . 82
A.3 Chamada de Funções . . . 83
A.3.1 Aplicação . . . 83

A.3.2 Formato . . . 83
A.3.3 Parâmetros . . . 83
A.3.4 Exemplo . . . 83
A.4 Parâmetros de Função . . . 84
A.4.1 Aplicação . . . 84
A.4.2 Formato . . . 84
A.4.3 Parâmetros . . . 84
A.4.4 Exemplo . . . 84
A.5 Retorno de Funções . . . 85
A.5.1 Aplicação . . . 85
A.5.2 Formato . . . 85
A.5.3 Parâmetros . . . 85
A.5.4 Exemplo . . . 85
A.6 Finalização de Funções . . . 86
A.6.1 Aplicação . . . 86
A.6.2 Formato . . . 86
A.6.3 Parâmetros . . . 86
A.6.4 Exemplo . . . 86

APÊNDICE B – QUÁDRUPLAS PARA LEITURA/ESCRITA E DECLARAÇÃO DE VARIÁVEIS . . . 87
B.1 Declaração de Variáveis . . . 87
B.1.1 Aplicação . . . 87
B.1.2 Formato . . . 87
B.1.3 Parâmetros . . . 87
B.1.4 Exemplo . . . 87
B.2 Leitura de variáveis da Memória . . . 88
B.2.1 Aplicação . . . 88
B.2.2 Formato . . . 88
B.2.3 Parâmetros . . . 88
B.2.4 Exemplo . . . 88
B.3 Escrita de variáveis na Memória . . . 90
B.3.1 Aplicação . . . 90
B.3.2 Formato . . . 90
B.3.3 Parâmetros . . . 90
B.3.4 Exemplo . . . 90
B.4 Atribuição . . . 92
B.4.1 Aplicação . . . 92
B.4.2 Formato . . . 92
B.4.3 Parâmetros . . . 92

B.4.4 Exemplo 92

APÊNDICE C – QUÁDRUPLAS DE CONTROLE 93
C.1 Finalização de Programas 93
C.1.1 Aplicação 93
C.1.2 Formato 93
C.1.3 Parâmetros 93
C.2 Rótulos de Controle - *Labels* 94
C.2.1 Aplicação 94
C.2.2 Formato 94
C.2.3 Parâmetros 94
C.2.4 Exemplo 94
C.3 Desvio de Fluxo 95
C.3.1 Aplicação 95
C.3.2 Formato 95
C.3.3 Parâmetros 95
C.3.4 Exemplo 95
C.4 Desvio Condicional - *If False* 96
C.4.1 Aplicação 96
C.4.2 Formato 96
C.4.3 Parâmetros 96
C.4.4 Exemplo 96

APÊNDICE D – QUÁDRUPLAS LÓGICO-ARITMÉTICAS 99
D.1 Adição 99
D.1.1 Aplicação 99
D.1.2 Formato 99
D.1.3 Parâmetros 99
D.1.4 Exemplo 99
D.2 Subtração 100
D.2.1 Aplicação 100
D.2.2 Formato 100
D.2.3 Parâmetros 100
D.2.4 Exemplo 100
D.3 Multiplicação 101
D.3.1 Aplicação 101
D.3.2 Formato 101
D.3.3 Parâmetros 101
D.3.4 Exemplo 101
D.4 Divisão 102

D.4.1 Aplicação 102
D.4.2 Formato 102
D.4.3 Parâmetros 102
D.4.4 Exemplo 102
D.5 Maior que 103
D.5.1 Aplicação 103
D.5.2 Formato 103
D.5.3 Parâmetros 103
D.5.4 Exemplo 103
D.6 Menor que 104
D.6.1 Aplicação 104
D.6.2 Formato 104
D.6.3 Parâmetros 104
D.6.4 Exemplo 104
D.7 Maior ou igual a 105
D.7.1 Aplicação 105
D.7.2 Formato 105
D.7.3 Parâmetros 105
D.7.4 Exemplo 105
D.8 Menor ou igual a 106
D.8.1 Aplicação 106
D.8.2 Formato 106
D.8.3 Parâmetros 106
D.8.4 Exemplo 106
D.9 É igual a 107
D.9.1 Aplicação 107
D.9.2 Formato 107
D.9.3 Parâmetros 107
D.9.4 Exemplo 107
D.10 É diferente de 108
D.10.1 Aplicação 108
D.10.2 Formato 108
D.10.3 Parâmetros 108
D.10.4 Exemplo 108

1 Introdução

Segundo Aho (1), um compilador é um programa que lê um código em uma linguagem - *linguagem fonte (source language)* - e o traduz para outra linguagem - *linguagem objetivo (target language)* -, preservando o funcionamento do programa. Essa tradução se dá muitas vezes de uma linguagem de programação, que é mais próxima da linguagem humana, para uma linguagem de máquina (binária) que é possível de ser executada pelas máquinas.

Aho (1) ainda destaca que a importância dos compiladores se baseia no fato de que o mundo depende de linguagens de programação para conseguir descrever todo o *software* hoje utilizado, precisando, portanto, dos compiladores para traduzir esses códigos para códigos executáveis pelas máquinas.

Como estudante de Engenharia de Computação se faz então de primordial importância entender os conceitos básicos de Compiladores e saber implementá-los dado o seu papel fundamental na computação vigente. Neste contexto, o presente projeto visa o desenvolvimento de um compilador da linguagem C- (uma versão simplificada da linguagem C) para a linguagem de máquina do processador desenvolvido na matéria de Arquitetura de Computadores do curso, cujo nome é *AsterixForth* (2), um processador de pilha. Originalmente, o objetivo incluiria os testes em *FPGA (Field Programmable Gate Array)* mas, devido ao regime especial de atividades, o objetivo se limita em ir até a geração do código binário sem a realização dos testes em *hardware*.

Para explicar o desenvolvimento do projeto, as seguintes seções se organizam da seguinte maneira: no Capítulo 2 falaremos sobre o processador *AsterixForth*, destacando sua estrutura, seu conjunto de instruções e sua organização de memória. Seguiremos no Capítulo 3 falando da fase de análise do compilador, isto é, da parte responsável por analisar o código fonte em termos léxicos, sintáticos e semânticos. No Capítulo 4 falaremos da fase de síntese do compilador, onde transformamos as estruturas intermediárias geradas na fase de análise para gerar o código escrito na linguagem objetivo. No capítulo 5 aplicaremos o compilador em três exemplos de código e, por fim, no capítulo 6 iremos analisar as dificuldades encontradas e destacar alguns pontos importantes.

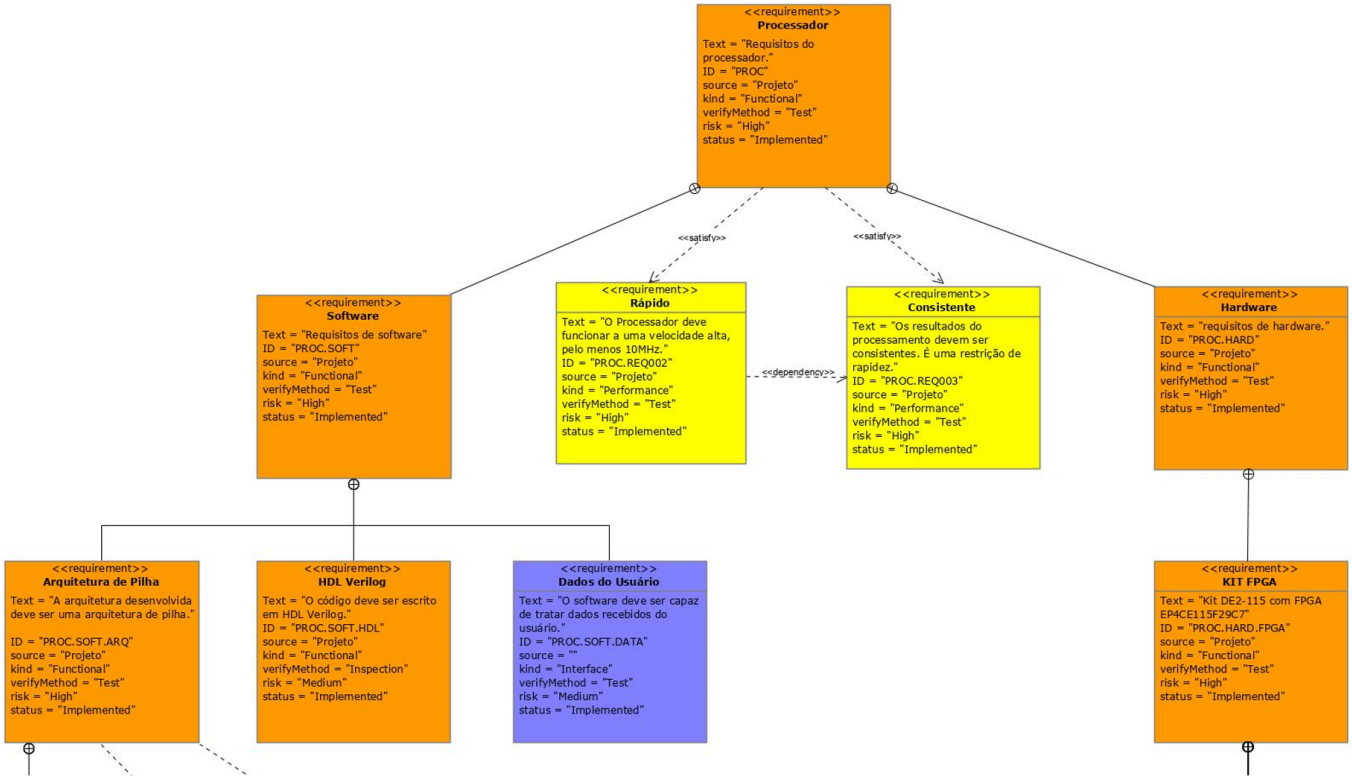
2 O Processador

2.1 Requisitos do Processador

Conforme pode ser visto na Figura 2, a estrutura de requisitos do processador *AsterixForth* é bastante extensa. Para melhor entendimento da correlação de suas partes, optamos por apresentar a Figura 2 com a designação gráfica de quatro partes que explicaremos. A **Parte I** falará sobre a estrutura básica do processador, em seus requisitos principais e em um nível mais alto; a **Parte II** falará sobre as estruturas de pilha e *buffer*; a **Parte III** engloba recursividade, as características do conjunto de instruções e questões como troca de contexto e facilidade de uso; por fim a **Parte IV** traz uma discussão mais aprofundada sobre o *hardware* necessário ao processador. São denotados em laranja os requisitos funcionais, em roxo os requisitos de interface e em amarelo os requisitos de performance.

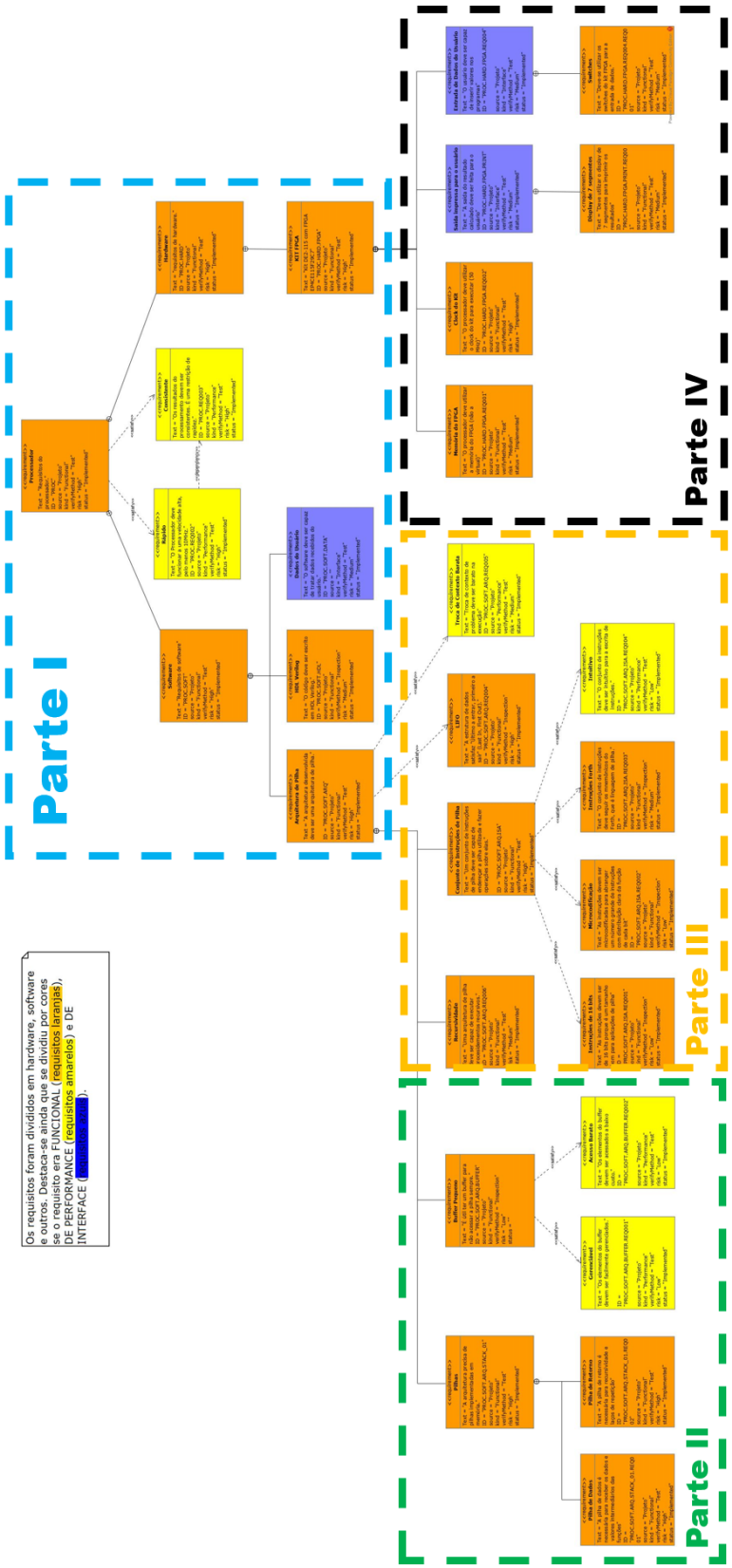
2.1.1 Parte I - Estrutura Principal

Figura 1 – Diagrama de Blocos do Processador *AsterixForth* - Parte I - Estrutura Principal



Fonte: Autoria Própria

Figura 2 – Diagrama de Blocos do Processador *AsterixForth*



Fonte: Autoria Própria

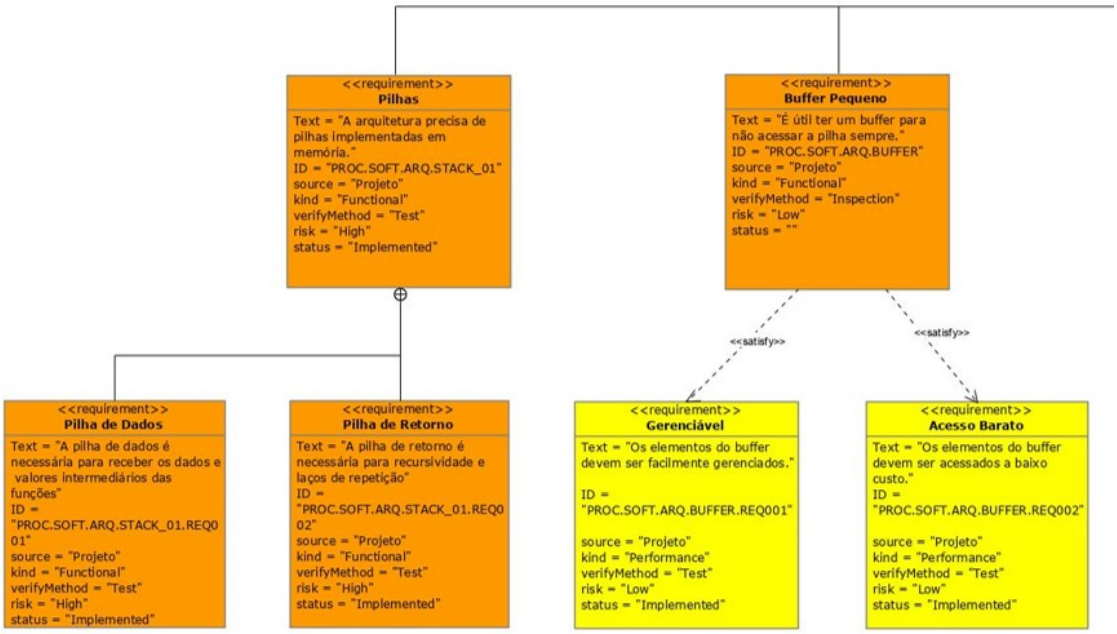
Na Figura 1 observamos que o diagrama possui requisitos funcionais principais tanto de *software* quanto de *hardware*. Dos requisitos funcionais de *software*, percebemos a necessidade de ser uma arquitetura de pilha, que foi uma determinação da autora quanto ao tipo de arquitetura utilizada, além da necessidade de que fosse descrito em *Verilog*, requisito este proveniente da disciplina de Arquitetura de Computadores onde foi desenvolvido. Dos requisitos funcionais de *hardware*, o principal requisito era que fosse mapeado para um *kit FPGA DE-115* da *Altera*.

Dos requisitos de interface, temos apenas a necessidade de que seja possível ao usuário inserir dados no processador e ter os seus dados devidamente tratados.

Dos requisitos de performance da Parte I temos que o processador deve ser capaz de executar em uma velocidade alta, porém sem perder a consistência da execução, dois requisitos bastante inter relacionados, como podemos ver em 1.

2.1.2 Parte II - Estruturas de Pilha e de *Buffer*

Figura 3 – Diagrama de Blocos do Processador *AsterixForth* - Parte II - Estruturas de Pilha e de *Buffer*



Fonte: Autoria Própria

Na Figura 3, temos os requisitos necessários de ser desenvolvidos como parte do que compõe o requisito de ser uma arquitetura de pilha apresentado em 2.1.1.

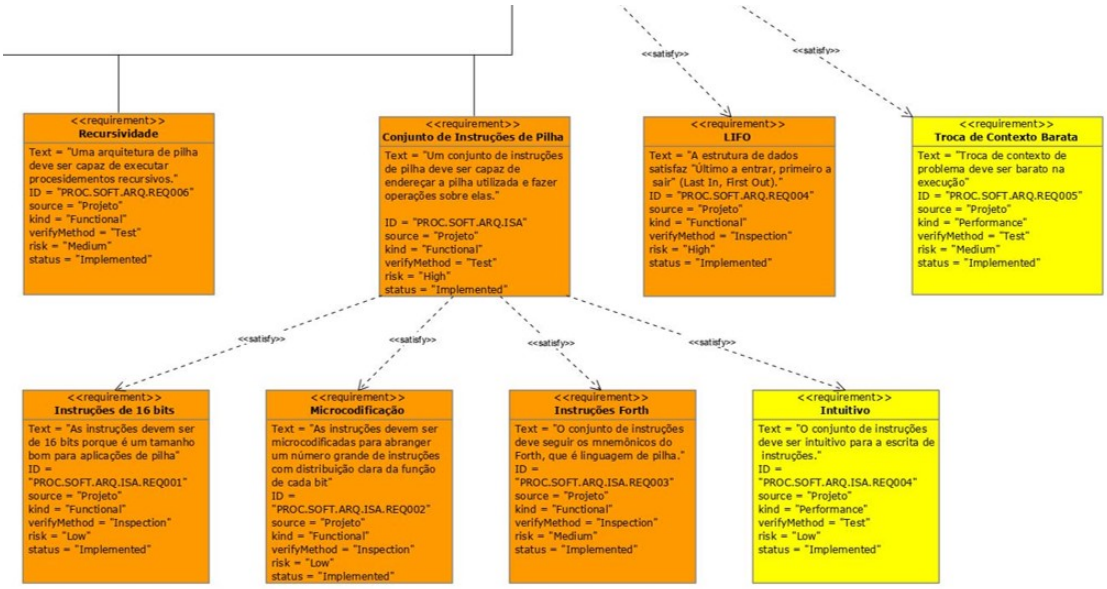
O primeiro conjunto de requisitos funcionais se referem à estrutura de pilha em si. A necessidade é por pilhas armazenadas como memória, dividas em **pilha de dados**

e **pilha de retorno**. A primeira é responsável por cálculos e valores intermediários de funções, enquanto a segunda é responsável por permitir a recursividade e a chamada de funções de forma simplificada.

O segundo conjunto de requisitos se refere às características do *buffer*. Optou-se por um *buffer* pequeno, o que é necessário para cumprir os requisitos de performance de ser mais gerenciável e ter acesso barato.

2.1.3 Parte III - Recursividade, Conjunto de Instruções e Troca de Contexto

Figura 4 – Diagrama de Blocos do Processador *AsterixForth* - Parte III - Recursividade, Conjunto de Instruções e Troca de Contexto



Fonte: Autoria Própria

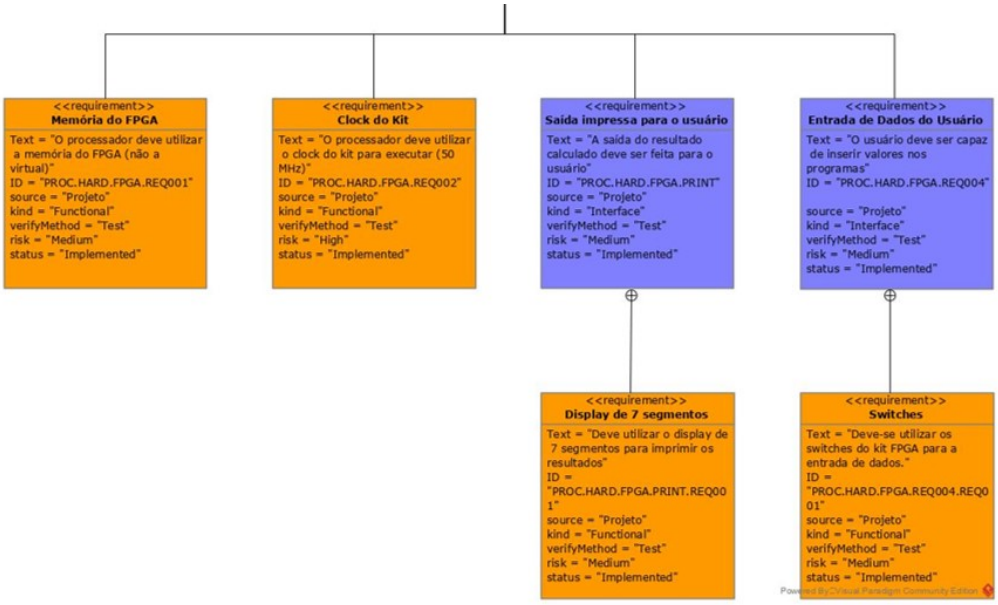
Algumas questões importantes são abordadas nos requisitos da parte III na Figura 4. O primeiro deles é que a arquitetura de pilha precisa ser capaz de executar facilmente processos recursivos.

O conjunto de instruções tem também um papel muito importante na arquitetura, e precisa ser composto por instruções de 16 *bits*, ser micro-codificada e com mnemônicos que correspondam as instruções da linguagem *Forth*. O conjunto de instruções precisa também cumprir o requisito funcional de ser intuito, para permitir uma escrita rápida de seus códigos.

Um requisito inerente à qualquer estrutura de pilha é a regra de acesso ***LIFO (Last In, First Out)***, isto é, o último elemento a ser inserido da pilha é o elemento que deve ser retirado quando se acessar a pilha. Um último requisito de performance muito importante que aparece é a necessidade de a troca de contexto ser muito barata na arquitetura, permitindo troca rápida durante a execução.

2.1.4 Parte IV - *Hardware*

Figura 5 – Diagrama de Blocos do Processador *AsterixForth* - Parte III - *Hardware*



Fonte: Autoria Própria

O último conjunto de requisitos do processador se refere aos requisitos de *hardware* utilizados no *kit FPGA*. De requisitos funcionais, necessitamos da memória do *kit FPGA*, do relógio do *kit* (*clock*), dos *displays* de 7 segmentos e dos *switches*. Estes dois últimos servem aos propósitos de interface de entregar o resultado do processamento para o usuário e de receber a entrada do usuário, respectivamente.

2.2 Componentes do Processador

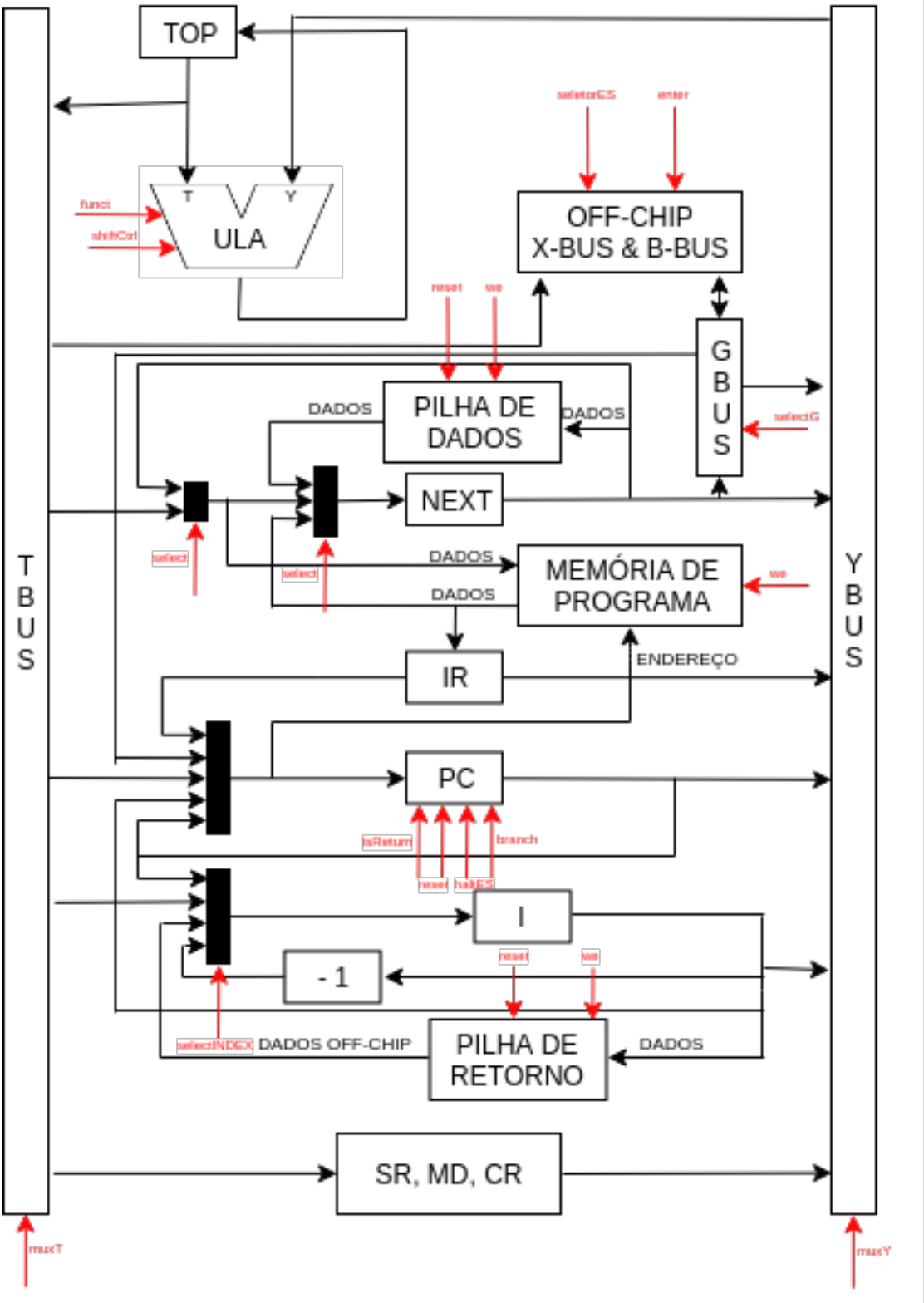
Uma explicação detalhada do processador pode ser encontrada em (2). A figura 6 traz o *datapath* da arquitetura, e faremos a apresentação dos principais componentes da arquitetura a seguindo para manter a discussão sobre o processador reduzida, visto que a arquitetura do *AsterixForth* possui 28 componentes.

A arquitetura do *AsterixForth* é baseada na arquitetura do *NOVIX NC4016*, que é uma arquitetura de pilha, e desta herdou muito do *datapath*. Foi desenvolvida em *HDL Verilog*.

Começando pela **ULA** (Unidade Lógico-Aritmética), esta é responsável por fazer as operações aritméticas, lógicas e de *shift*. Ela tem por entrada sempre o topo da pilha (entrada T) e a entrada Y, que pode ser proveniente da entrada do usuário, do segundo elemento da pilha, do endereço, do *PC* (*Program Counter*), da pilha de retorno ou dos

registradores auxiliares. Observe ainda que o retorno sempre é colocado sobre o registrador **TOP**, que é o topo da pilha.

Figura 6 – *Datapath* da arquitetura



Fonte: Autoria Própria em (2)

O **T bus** é apenas um barramento que redireciona o conteúdo do registrador TOP para o resto da arquitetura. É interessante notar que, como umas das possibilidades de alimentação do registrador TOP é pela pilha de retorno, acaba que, quando se troca para a execução sobre a pilha de retorno, o registrador passa a ser o topo desta pilha. Como o T bus pode ser consumido pela pilha de retorno, observe que a pilha de dados não é afetada quando trocamos para a pilha de retorno.

O registrador **NEXT** representa o segundo elemento da pilha de dados quando o contexto está na pilha de dados e representa o primeiro elemento da pilha de dados quando o contexto está na pilha de retorno. Esse registrador pode ser alimentado pela pilha de dados, pelo barramento T e pela memória de programa. Esse registrador é bem importante porque ele facilita o acesso ao segundo elemento da pilha, que é bastante utilizado. Isso compactua com a ideia de se manter um *buffer* pequeno, como descrito no diagrama de requisitos.

A **pilha de dados** e a *pilha de retorno* foram desenvolvidas sobre a memória do *FPGA*, adaptando-se um *template* de memória de leitura e escrita com dois *clocks*. A adaptação foi feita no sentido de garantir a obediência e regra de execução LIFO das pilhas.

A **memória de programa**, o **registrador IR** e o **registrador PC** conversam bastante, pois se referem basicamente ao controle da execução do programa. Na memória de programa, temos uma memória com as instruções binárias salvas, que já considera a possibilidade de se ler e de se escrever nessa memória. A instrução lida é o número de instrução é armazenado no *PC*, enquanto a instrução é armazenada no registrador IR. É interessante destacar que a memória de programa tem acesso à ULA pelo uso do barramento Y e pode receber dados do barramento T, sendo assim a forma que utiliza para realizar as contas necessárias aos desvios condicionais e incondicionais da arquitetura.

A **pilha de retorno** tem uma forte relação com os registradores **INDEX (I)** e **-1**. O registrador INDEX possui esse nome porque ele pode ser utilizado junto com o registrador decrementador -1 para executar laços *for* com o teste do laço ocorrendo sem a necessidade de se acessar a ULA, economizando 2 instruções por teste. É uma abordagem bastante útil quando se trabalha com uma linguagem de pilha como *Forth*.

A entrada e a saída de dados é feita pelos componentes **G BUS** e **OFF-CHIP X-BUS E B-BUS**. O G BUS apenas recebe os dados de NEXT e escreve em Y BUS, servindo como caminho para os dados que vão para impressão nos *displays de* 7 *segmentos* e que vem da configuração dos *switches*. Essa é uma parte importante por se tratar da interface com o usuário, e tentou cumprir com a usabilidade necessária, fazendo com que os dados só fossem lidos da entrada do usuário quando da cofirmação por meio da pressão sobre o botão de *enter* (*push button*).

Todos esses componentes foram integrados em um módulo único chamado **AsterixForth**, responsável por garantir as ligações e os barramentos e o envio dos sinais de controle. É interessante ver que estruturas como o barramento T, que é apenas um barramento sem nenhum tratamento, não precisariam ter sido criadas, porém o foram para coincidir ao máximo com a arquitetura de inspiração e principalmente para prover modularidade de código.

Os sinais de controle, por fim, são determinados pela **Unidade de Controle**. Esta recebe a instrução corrente e a decompõe para enviar os sinais de controle ao resto do processador. Como veremos a seguir, o conjunto de instruções da arquitetura é micro-codificado, de modo que os *bits* das instruções e suas combinações refletem várias possibilidades e várias combinações de instruções, podendo-se, por exemplo, executar uma instrução na ULA com a pilha de dados ao mesmo tempo que a pilha de retorno está executando a saída de uma função.

2.3 Conjunto de Instruções

Como dissemos anteriormente, o conjunto de instruções criado é micro-codificado, isto é, uma instrução pode carregar várias instruções nela, com a informação de cada *bit* variando de acordo com o prefixo da instrução. Por exemplo, toda instrução que começa com zero carrega um imediato na pilha, é uma instrução de **PUSH**. Instruções que começam com 10, mas não são seguidas por 00, são as instruções de desvio, que carregam em seus outros *bits* qual é o *offset* que deve ser adicionado na instrução vigente em complemento de 2.

Instruções iniciadas em 1000 são as instruções da ULA e merecem uma atenção especial. Os 5 *bits* que seguem o prefixo 1000 são responsáveis por determinar a instrução e a origem do segundo parâmetro da operação (campos $UUUYY$ na Figura 10). O campo C define se o valor do primeiro elemento da pilha deve ser copiador para o valor do segundo elemento da pilha (que é a instrução de **DUP** da Figura 11). O campo R indica se deve ser feito o retorno da função ou não (instrução de **EXIT** na Figura 10). O campo S indica qual pilha (*stack*) está ativa no momento (isso permite a troca de pilha nas instruções $R >$ e $> R$ da Figura 11). Os campos ED ainda não foram utilizados. Por fim, os campos SS são para instruções de *shift* nos dados. Uma instrução muito útil que é possível apenas pelo fato de se ter dois *clocks* executando (um para escrita e outro para leitura) é a instrução de $SWAP$ (Figura 11). Esta instrução permite a troca do primeiro elemento da pilha com o segundo e é amplamente utilizada.

As instruções de controle são apenas para encerrar a execução do programa. As instruções de entrada e saída correspondem ao uso da interface do usuário para receber os dados dos *switches* e imprimir os dados nos *displays de* 7 segmentos.

As instruções de acesso à memória são responsáveis por ler o conteúdo do endereço de memória escrito em T e escrever o conteúdo do segundo elemento da pilha no endereço descrito em T. Vale salientar que estas instruções não foram implementadas ainda no processador *AsterixForth*.

Figura 7 – Instruções com Imediato

Nome	Formato	Descrição
Instruções com Imediato		
PUSH	**0**XXX_XXXX_XXXX_XXXX	Coloca um imediato no topo da pilha

Fonte: Autoria própria em (2)

Figura 8 – Instruções de Desvio

Nome	Formato	Descrição
Instruções de Desvio		
IF	**1001**_TAAA_AAAA_AAAA	Desvia se T == 0
CALL	**1010**_TAAA_AAAA_AAAA	Desvio incondicional
LOOP	**1011**_TAAA_AAAA_AAAA	*Loop* sobre o registrador de índice

Fonte: Autoria própria em (2)

Figura 9 – Instruções de controle

Nome	Formato	Descrição
Instruções de Controle		
NOP	**1000_000**Y_Y**000_0000**	Nada
HALT	**1110_000**Y_Y**000_0000**	Entra em Loop Infinito

Fonte: Autoria própria em (2)

Figura 10 – Instruções da unidade lógico-aritmética

Instruções da Unidade Lógico-Aritmética		
AND	**1000_001**Y_YCRS_EDSS	Faz *bit-a-bit* a operação lógica AND
OR	**1000_011**Y_YCRS_EDSS	Faz *bit-a-bit* a operação lógica OR
XOR	**1000_101**Y_YCRS_EDSS	Faz *bit-a-bit* a operação lógica XOR
+	**1000_100**Y_YCRS_EDSS	Faz a operação de soma dos operandos
-	**1000_010**Y_YCRS_EDSS	Faz a operação T - Y
--	**1000_110**Y_YCRS_EDSS	Faz a operação Y - T
SHIFT LR	**1000**_UUUY_YCRS_ED**01**	Faz a operação de *shift* lógico para direita
SHIFT AL	**1000**_UUUY_YCRS_ED**10**	Faz a operação de *shift* aritimético para esquerda
SHIFT AR	**1000**_UUUY_YCRS_ED**11**	Faz a operação de shift aritmético para direita
/	**1000_0001_0**CRS_E**1**SS	Faz a operação de divisão usando T e o reg MD
*	**1000_0001_0**CRS_E**0**SS	Faz a operação de multiplicação usando T e o reg MD
SQRT	**1000_0001_1**CRS_EDSS	Faz a operação de raiz quadrada inteira (piso) de T com o reg SR

Fonte: Autoria própria em (2)

Figura 11 – Instruções de manipulação da pilha

Nome	Formato	Descrição
Instruções de Manipulação da Pilha		
DUP	**1000**_UUU**0**_**01**RS_EDSS	Copia o valor de T para N
OVER	**1000_1110_00**RS_EDSS	Copia o valor de N para T
EXIT	**1000**_UUUY_YC**1**S_EDSS	Retorna para a função anterior
DOUBLE	**1000**_UUUY_YCRS_**1**DSS	Faz com que a pilha interprete o dado como double (32-bits)
R>	**1000**_UUUY_Y**1**R**0**_**00**SS	Faz passar da pilha de retorno para a pilha de dados
>R	**1000**_UUUY_Y**1**R**1**_**00**SS	Faz passar da pilha de dados para a pilha de retorno
DROP	**1110_1110_00**R**0_0001**	Diminui o tamanho da pilha
SWAP	**1000_1110_01**RS_EDSS	Troca o topo da pilha com o segundo elemento da pilha

Fonte: Autoria própria em (2)

Figura 12 – Instruções de acesso à memória

Nome	Formato	Descrição
Instruções de Acesso à Memória		
@	**1110**_UUUY_YCRO_OOOO	Faz a leitura do conteúdo do endereço de memória em T
!	**1111**_UUUY_YCRO_OOOO	Faz a escrita de N no endereço indicado por T

Fonte: Autoria própria em (2)

Figura 13 – Instruções de entrada e saída

Nome	Formato	Descrição
Instruções de Entrada e Saída		
INPUT	**1100_0000_0000_0000**	Lê a entrada do usuário (*switches*) e armazena na pilha
OUTPUT	**1101_0000_0000_0000**	Imprime o topo da pilha no *display*

Fonte: Autoria própria em (2)

2.4 Organização da Memória

A organização da memória na arquitetura *AsterixForth* é bastante segmentada.

Primeiro, temos a **memória de instruções**, que traz em si o código binário para que seja executado. Ela pode ter seu conteúdo não só lido como também pode ser alterado, prevendo uma necessidade possível do futuro sistema operacional que vai ser futuramente desenvolvido.

A **pilha de dados** e a **pilha de retorno** são também unidades de memória da arquitetura, e são amplamente utilizadas para as contas executadas e para as chamadas de função, recursividade, dentre outros.

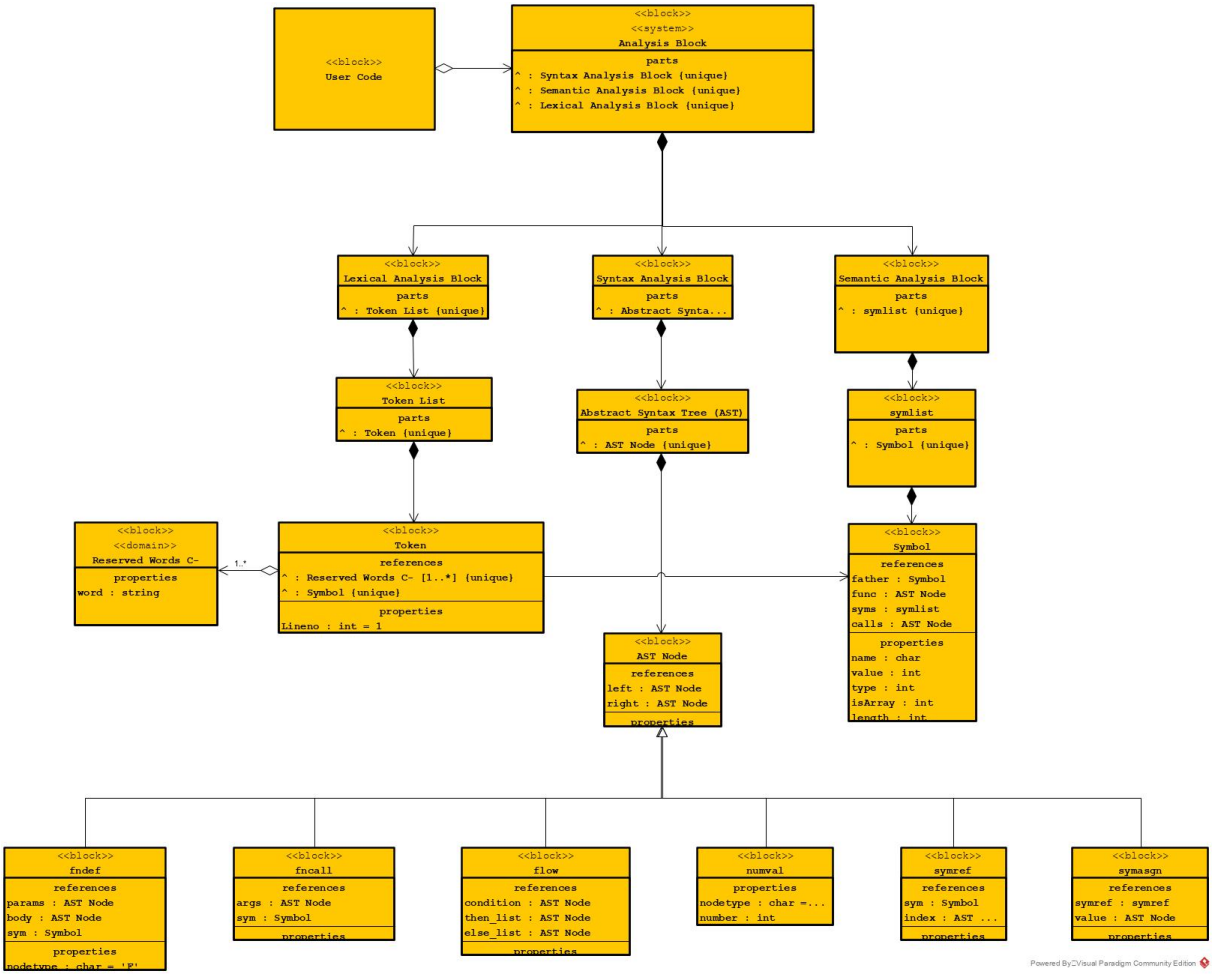
Em uma linguagem como *Forth*, que é uma linguagem para ser executada em uma arquitetura de pilha, o uso extensivo de sub-rotinas é aplicado, e a necessidade de uma memória de dados externa é pouco necessária, não tendo sido, portanto, implementada *a priori*. Porém, se previra a possível necessidade desta para casos de códigos não optimizados, de modo que as instruções necessárias foram devidamente definidas e a estrutura já estabelecida como previra a arquitetura fonte *NOVIX NC4016*, com essa memória sendo acessada também pelo o **G BUS**, como é feito com a entrada e saída do usuário. Em outras palavras, a memória de dados extra é tratada como uma entrada/saída externa do processador. Destacamos que o "não optimizado"pode decorrer, por exemplo, da não existência de um módulo de optimização no compilador combinado com o mapeamento de uma linguagem não feita para pilhas, como é o caso do $C-$.

3 Compilador: Fase de Análise

3.1 Diagrama de Blocos

A figura 14 traz a modelagem em diagrama de blocos referente ao módulo de análise do compilador. A fase de análise do compilador é dividida em três partes: **análise léxica**, **análise sintática** e **análise semântica**. Iremos explicar cada uma delas melhor nos próximas subseções, mas já podemos notar aqui as estruturas necessárias à cada umas dessas fases.

Figura 14 – Diagrama de Blocos do Módulo de Análise do Compilador *Panoramix*



Fonte: Autoria própria

A fase de análise léxica tem por estrutura principal o ***Token***, que pode ser resumido como um trecho de texto (*lexema*) que carrega significado no código. Destacamos a relação dos *Tokens* com as ***Palavras Reservadas***, que são palavras com significados específicos

na linguagem e não podem, por exemplo, ser o nome de uma variável (ex.: *while*). Outro ponto a ser destacado é a relação com a estrutura de símbolo. Essa não é abordagem tradicional, mas foi escolhida por facilitar o formato de retorno do *Token* já como uma estrutura de símbolo para as fases conseguintes.

A fase de análise sintática tem por estrutura principal uma **AST** (*Abstract Syntax Tree*). Esta estrutura é composta por nós **AST Node**, que é uma estrutura que engloba 8 tipos de estrutura: **fndef** (para nós do tipo de definição de função), **fncall** (para chamadas de função), **flow** (para nós de fluxo como $if - then - else$ e $while$, sendo o único nó que pode ter três filhos), **numval** (para valores numéricos imediatos), **symref** (para variáveis) e **symasgn** (para atribuições com =). Os demais nós são alocados simplesmente como nós AST básicos.

A fase de análise semântica também se baseia na estrutura de ***Símbolo***. Ela é constituída por uma tabela com os símbolos contendo as informações necessárias à analise semântica do código.

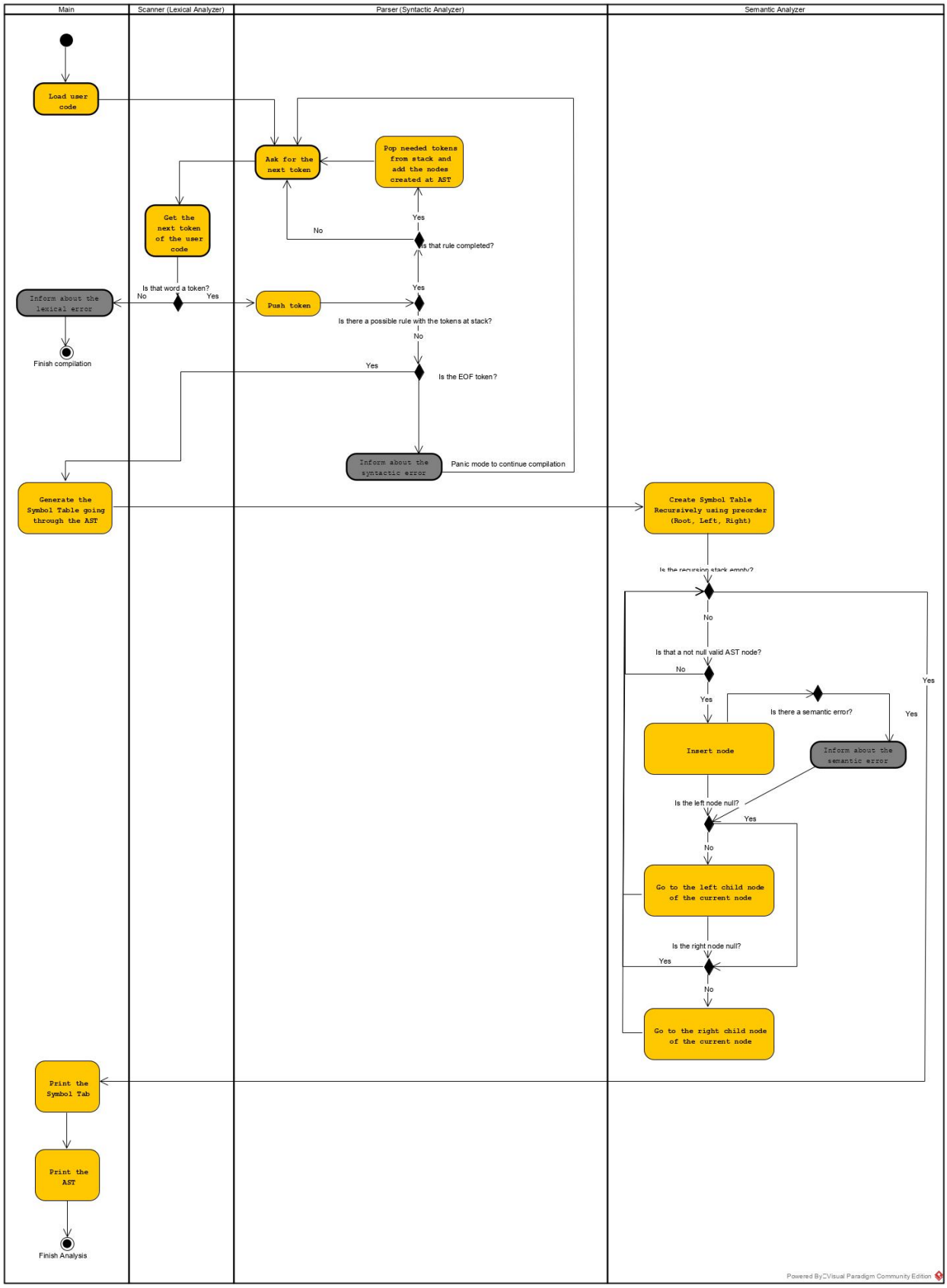
3.2 Diagrama de Atividades

Na Figura 15, trazemos o diagrama de atividades da fase de análise do compilador. Ele é composto por 4 raias: a raia **main**, a raia **scanner** (referente ao módulo de análise léxica), a raia **parser** (referente ao módulo de análise sintática) e a raia **análise semântica**. Antes da descrição mais aprofundada do diagrama, podemos já observar que há certa divisão entre os blocos, tendo uma relação bastante forte entre os módulos de análise léxica e sintática, seguindo o fluxo do código depois inteiramente para o módulo de análise semântica.

De fato isto ocorre porque a condição para que módulo de análise sintática possa reduzir as regras da gramática é que este possua *tokens* para analisar, e quem os retorna é o módulo de análise léxica, sendo portanto consultado em todo ciclo no começo da Figura 15. Em todo ciclo, ao se requisitar o *token*, verifica-se se este é de fato um *token* válido. Caso não o seja, o erro léxico é reportado e a compilação encerrada. Caso seja válido, este é empilhado na módulo de análise sintática, onde se vai verificar se o conteúdo da pilha corresponde a alguma regra ou não, reportando o erro sintático correspondente e continuando a execução. Quando uma regra é completada, esta é desempilhada e os nós respectivos são inseridos na árvore sintática abstrata (***AST***).

Quando por fim o símbolo de final do arquivo é encontrado, o fluxo de execução passa ao módulo de análise semântica, onde começa a criar a tabela de símbolos percorrendo a AST em pré-ordem e inserindo os símbolos criados na tabela. Em caso de erros semânticos, estes são reportados sem, entretanto, encerrar a execução.

Figura 15 – Diagrama de Atividades do Módulo de Análise do Compilador *Panoramix*



Fonte: Autoria Própria

Encerra-se essa parte do compilador com a impressão da tabela de símbolos e da *AST* em arquivos.

3.3 Análise Léxica

Como já foi brevemente dito, a análise léxica é responsável por fazer a geração de *tokens* para o módulo de análise sintática. Isto é, ele deve verificar se a ortografia das palavras fazem sentido com a linguagem-fonte determinada que, no nosso caso, é *C-*. O módulo de análise léxica foi escrito com *Flex*.

O projeto base para as definições e regras da linguagem *C-* podem ser encontradas em (3).

3.3.1 Palavras Reservadas

As palavras reservadas, que devem ser escritas em caixa baixa, são:

else, **if**, **int**, **return**, **void** e **while**.

3.3.2 Símbolos Especiais

Os símbolos especiais são:

+, -, *, /, <, <=, >, >=, ==, !=, =, ;, ,, (), [], {}, /* e */.

3.3.3 Marcadores de Número e de Identificador de Variável

As expressões regulares que definem os marcadores de números e de variáveis (estas sensíveis à caixa alta) são:

$$ID = letra\ letra*$$
$$NUM = digito\ digito*$$
$$letra = a|\ldots|z|A|\ldots|Z$$
$$digito = 0|\ldots|9$$

3.3.4 Espaço em Branco

É composto por mudanças de linha e tabulações, servindo para separar *IDs*, *NUMs* e palavras-chave.

3.3.5 Comentários

Os comentários são cercados por /* e */ e tudo escrito entre esses delimitadores é ignorado na analise.

3.4 Análise Sintática

A análise sintática é responsável por ver se a sequência de *tokens* faz sentido de acordo com as regras da linguagem-fonte determinada. Escrevemos o módulo com *Bison* e contruímos a *AST* e a tabela de símbolos com um código em *C*, em um arquivo chamado *AST.c* e com os *headers* em *AST.h*.

As regras da linguagem *C*- podem ser encontrada em (3).

3.4.1 Regras Gramaticais

$program \rightarrow decl_list$

$decl_list \rightarrow decl_list\ decl \mid decl$

$decl \rightarrow var_decl \mid fun_decl$

$var_decl \rightarrow e_type\ ID\ ; \mid e_type\ ID\ [\ NUM\ ];$

$e_type \rightarrow int \mid void$

$fun_decl \rightarrow e_type\ ID\ (\ params\ )\ com_decl$

$params \rightarrow param_list \mid void$

$param_list \rightarrow param_list\ ,\ param \mid param$

$param \rightarrow e_type\ ID \mid e_type\ ID\ [\ ]$

$com_decl \rightarrow \{\ loc_decl\ stmt_list\ \} \mid \{loc_decl\} \mid \{stmt_list\} \mid \{\ \}$

$loc_decl \rightarrow loc_decl\ var_decl \mid var_decl$

$stmt_list \rightarrow stmt_list\ stmt \mid stmt$

$stmt \rightarrow exp_decl \mid com_decl \mid sel_decl \mid it_decl \mid ret_decl$

$exp_decl \rightarrow exp\ ; \mid ;$

$sel_decl \rightarrow if\ (\ exp\ )\ stmt \mid if\ (\ exp\ )\ stmt\ else\ stmt$

$it_decl \rightarrow while\ (\ exp\ )\ stmt$

$ret_decl \rightarrow return\ ; \mid return\ exp\ ;$

$exp \rightarrow var = exp \mid simple_exp$

$var \rightarrow ID \mid ID\ [\ exp\ ]$

$simple_exp \rightarrow sum_exp\ rel\ sum_exp \mid sum_exp$

$rel \rightarrow <= \mid < \mid > \mid >= \mid == \mid !=$

$sum_exp \rightarrow sum_exp\ sum\ term \mid term$

$sum \rightarrow + \mid -$

$term \rightarrow term\ mult\ factor \mid factor$

$mult \rightarrow * \mid /$

$factor \rightarrow (\ exp\ ) \mid var \mid activation \mid NUM$

$activation \rightarrow ID\ (\ arg_list\ ) \mid ID\ (\ )$

$arg_list \rightarrow arg_list\ ,\ exp \mid exp$

3.5 Análise Semântica

A análise semântica é responsável por localizar erros de significado das expressões escritas. Sua análise é feita através da lista de símbolos criada a partir da *AST*. A abordagem escolhida foi por já montar a tabela de símbolos enquanto se fazia a análise sintática, economizando o ato de percorrer a árvore novamente.

Vários erros são passíveis de serem reportados pelo compilador, e podemos dividí-los em 4 classes:

3.5.1 Erros relacionados à declarações de funções e variáveis

1. Redeclaração de funções como variáveis;
2. Declaração de uma variável como *void*;
3. Redeclaração de variável;
4. Redeclaração de função;
5. Declaração implícita de função;
6. variáveis declaradas, mas não utilizadas;
7. variáveis não declaradas.

3.5.2 Erros relacionados à indexação

1. Utilização de variáveis indexadas sem utilizar índice;
2. Utilização de índice em variáveis inteiras.

3.5.3 Erros relacionados à tipos

1. Retorno de funções *void*;
2. Falta de retorno em funções *int*;

3. Atribuição de resultado de uma função *void* a uma variável.

3.5.4 Erros relacionados à função *main*

1. A função *main* deve ser a última função a ser delcarada;
2. a função *main* deve ser *void*;
3. A função *main* não deve possuir parâmetros;
4. A função *void main()* não foi declarada.

3.6 Resultados Gerados pela Fase de Análise

Como resultado do módulo de análise, temos alguns arquivos importantes gerados que servem para entender os erros escritos no código e as estruturas geradas.

O primeiro arquivo gerado é o arquivo de **análise**, que é o arquivo que irá armazenar os erros léxicos, sintáticos e semânticos gerados.

O segundo arquivo gerado é o arquivo da **tabela de símbolos**, que irá armazenar uma tabela com dados do nome do identificador (**identifier**), o tipo (**type**), o escopo (**scope**), a linha em que o identificador foi declarado (**decl_line**) e o tipo de retorno (**return**) caso seja função.

O último arquivo proveniente da fase de análise é a impressão da *AST*. A árvore é o recurso necessário para a geração do código intermediário na fase de síntese.

4 Compilador: Fase de Síntese

4.1 Geração de Código Intermediário

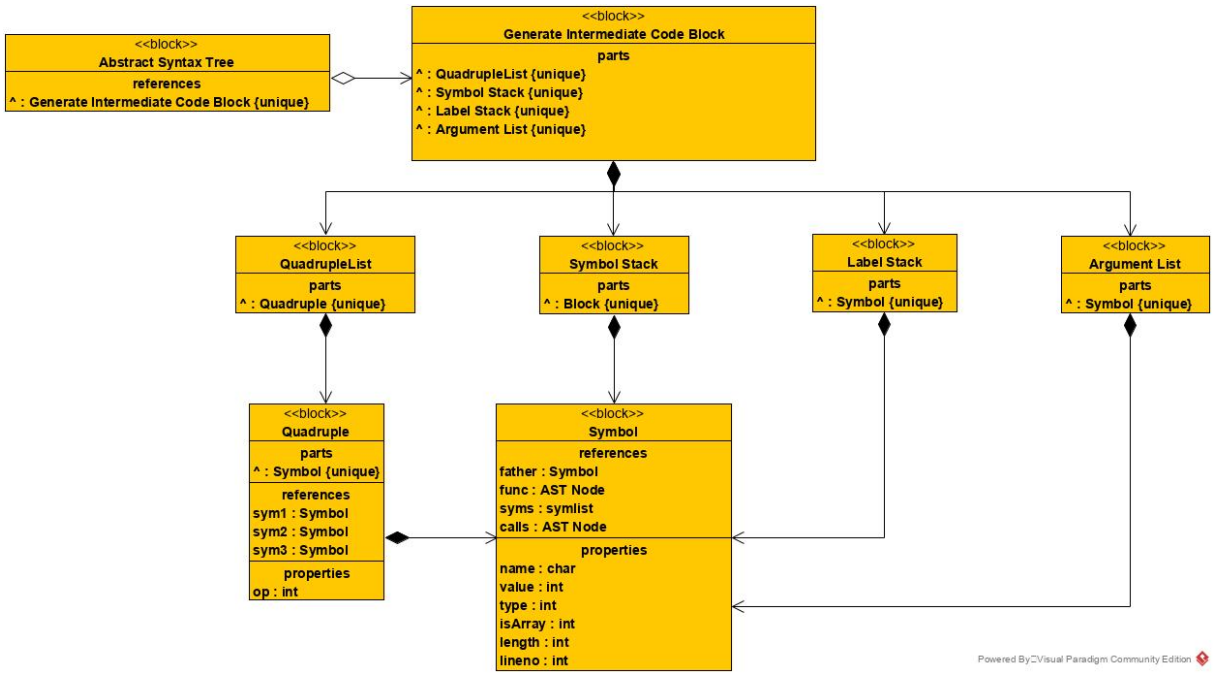
O módulo de geração do código intermediário é responsável por transformar a *AST* em uma lista de quádruplas, isto é, expressões que contém a operação e os até três símbolos envolvidos. Este módulo foi desenvolvido em Linguagem *C*, também no arquivo *AST.c*, aproveitando-se assim a árvore gerada ali.

4.1.1 Modelagem

4.1.1.1 Diagrama de Blocos

Podemos ver na Figura 16 o diagrama de blocos referente à geração do código intermediário do compilador *Panoramix*. Nele podemos observar a necessidade da *AST*, como já havíamos dito, como entrada do módulo. O módulo requer quatro estruturas principais, sendo elas: **Lista de Quádruplas**, **Pilha de Símbolos**, **Pilha de Rótulos** e **Pilha de Argumentos**.

Figura 16 – Diagrama de Blocos do Módulo de Geração do Código Intermediário do Compilador *Panoramix*



Fonte: Autoria própria

A **Lista de Quádruplas** é necessária para armazenas as quádruplas que servirão

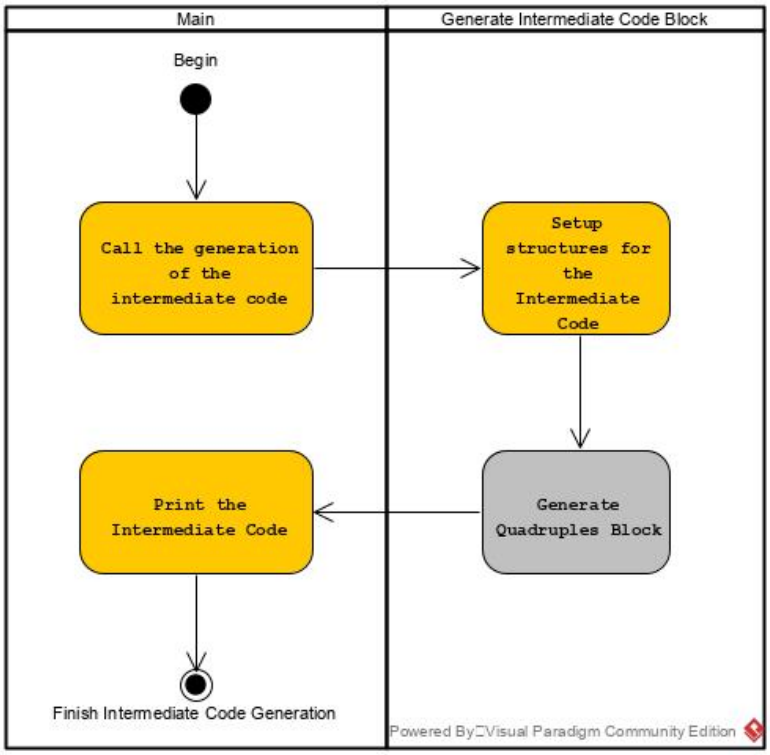
de entrada para a geração do código *assembly*. Ela é composta por **quádruplas**, que são estruturas que armazenam a operação e até três símbolos envolvidos naquela operação. Por armazenar referências a símbolos, a estrutura de quádrupla requer a definição de símbolo, que é proveniente da estrutura de símbolo da fase de análise.

As **pilhas** são necessárias para conseguir substituir as variáveis e os nós da árvore para as quádruplas substituindo corretamente os rótulos (*labels*) necessários para a geração das quádruplas.

4.1.1.2 Diagramas de Atividades

Para a geração do código intermediário, dependendo do tipo de nó encontrado, não só a quádrupla como até o percurso na árvore a partir dela podem variar. Desta maneira, podemos ver na Figura 17 o que seria a estrutura geral do módulo de geração do código intermediário, que seria a chamada feita na função *main* para a geração do código intermediário, seguida pela inicialização das estruturas necessárias ao código intermediário explicadas no diagrama de blocos da Figura 16 e da geração das quádruplas e então a impressão do código intermediário em um arquivo. Podemos, entretanto, dividir os tipos de nós em conjuntos de geração de quádruplas semelhantes, e vamos passar por casa um desses tipos explicando a execução para esses tipos do bloco *Generate Quadruple Block*.

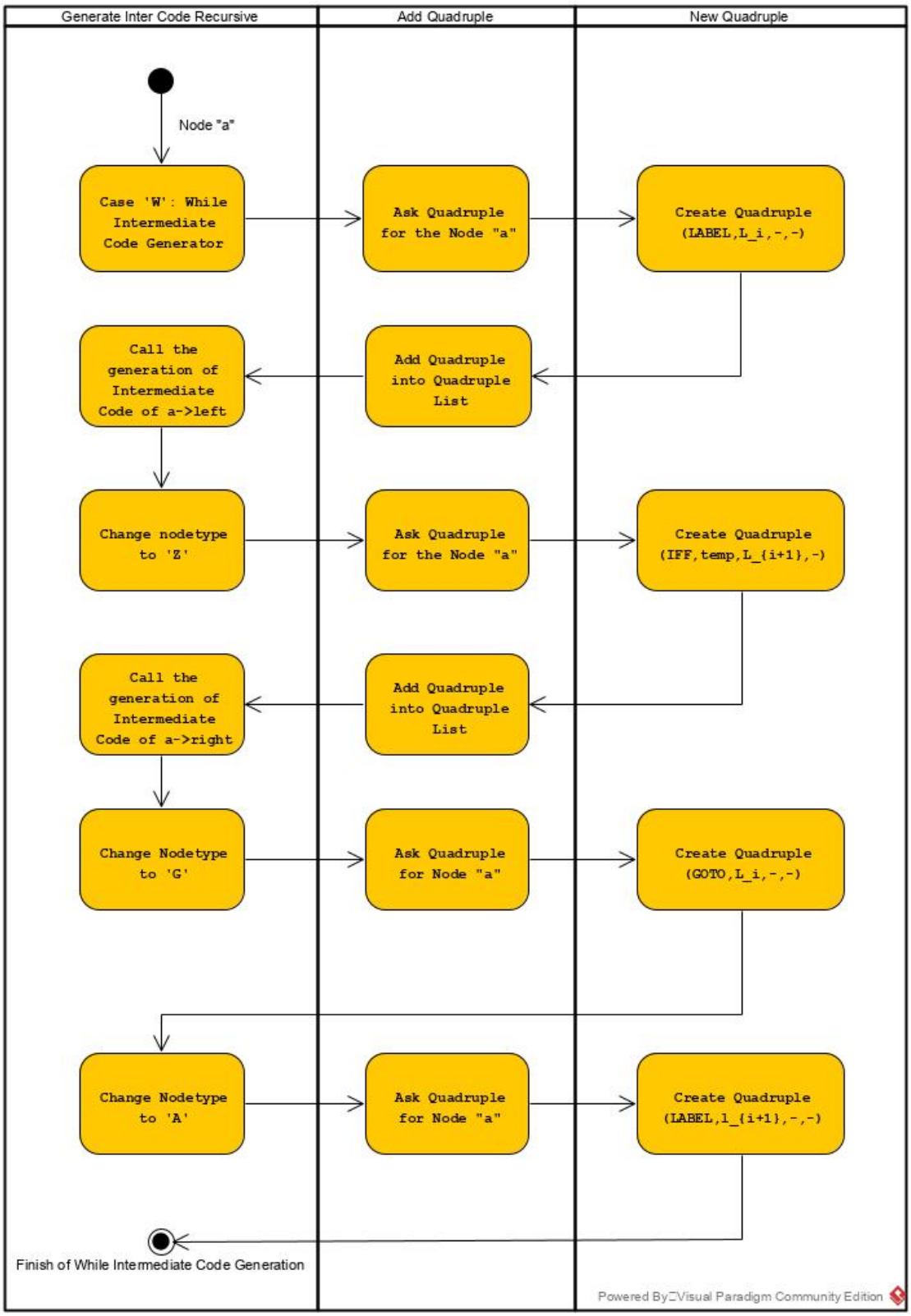
Figura 17 – Diagrama de Atividades Principal do Módulo de Geração do Código Intermediário do Compilador *Panoramix*



Fonte: Autoria própria

4.1.1.2.1 Geração de quádruplas para nós do tipo *While*

Figura 18 – Diagrama de Atividades para Nós do Tipo *While*



Fonte: Autoria própria

Na Figura 18 podemos observar o diagrama de atividades referente aos nós do tipo *While*. Ao percorrer a árvore, quando encontra um nó do tipo W, se solicita a adição de quádrupla para o módulo **Add Quadruple**, que por sua vez, solicita ao **New Quadruple** a criação da quádrupla $(LABEL, L_I, -, -)$, que é para aonde o código deve retornar a cada finalização do laço.

Tendo, então, adicionado a quádrupla à lista de quádruplas, se solicita a criação de quádruplas para a subárvore da esquerda. Esta subárvore é responsável por definir o teste de parada do laço *While*.

Quando se retorna da execução da subárvore esquerda, se deve, então, solicitar a criação de uma quádrupla $(IFF, temp, L_{i+1}, -)$ que é gerada quando trocamos o tipo do nó para E.

Uma vez terminado o teste do *While*, devemos solicitar a geração do corpo do *While*, e fazemos isso invocando a geração do código intermediário da subárvore da direita.

Quando retornamos da recursão da subárvore da direita, trocamos o tipo do nó para G e, então, criamos uma quádrupla do tipo $(GOTO, L_i, -, -)$. Esta quádrupla faz com que a execução retorne para o começo do *While*.

Por fim, trocamos o tipo do nó para A, e geramos uma quádrupla do tipo $(LABEL, L_{i+1}, -, -)$, que coincide com o *Label* para o qual o teste da quádrupla *IFF* desvia.

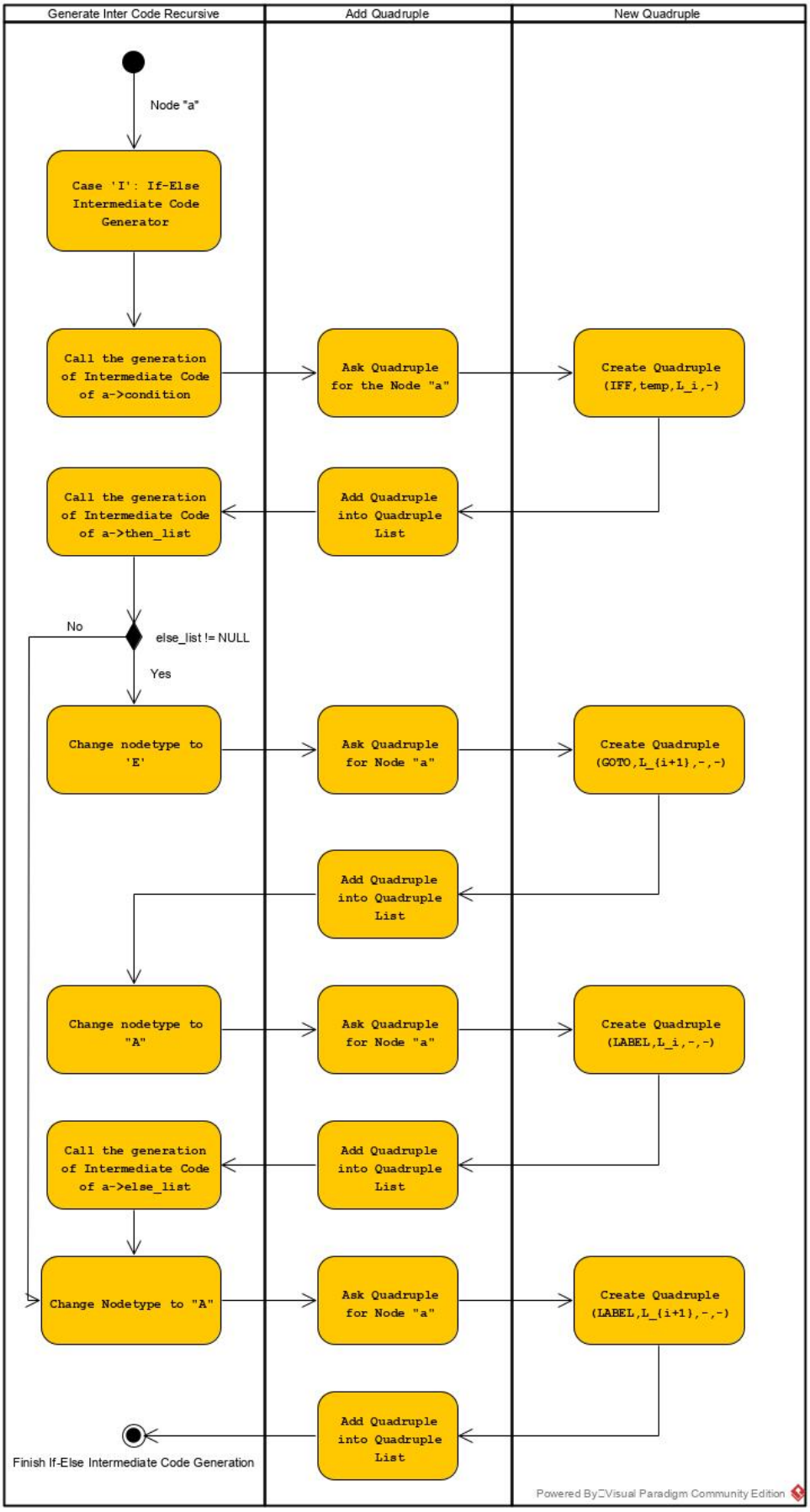
4.1.1.2.2 Geração de quádruplas para nós do tipo *if*

Nós do tipo *if* são bastante interessantes, pois utilizam o fato de que nós do tipo *flow* podem possuir até 3 filhos em caso de uma cláusula *else*. Um nó do tipo *if* inicia chamando a geração do código intermediário para a sub árvore *condition*. Ao retornar da execução recursiva desta se pode, portanto, criar uma quádrupla do tipo $(IFF, temp, L_i, -)$ que terá por variável *temp* o resultado empilhado da sub árvore *condition*.

A seguir sem inicia a geração do código intermediário para a sub árvore *then_list*. Neste ponto de execução há 2 possibilidades: caso exista um filho em else, trocamos o tipo do nó para E É geramos assim uma quádrupla do tipo $(GOTO, L_{i+1}, -, -)$. A seguir trocamos o tipo do nó para A é geramos assim uma quádrupla do tipo $(LABEL, L_i, -, -)$. Observe que isso faz sentido com a ideia de que, se temos uma cláusula *else* e a condição do *if* for falsa, então a execução deve desviar para o *else*.

Se solicita então a geração do código intermediário para a sub árvore *else*. O fluxo coincide com o que teríamos caso não tivéssemos a cláusula *else*. Trocamos o tipo do nó para A, Criamos uma quádrupla do tipo $(LABEL, L_{i+1}, -, -)$ e encerramos a geração de quádruplas no caso de nós do tipo *if*.

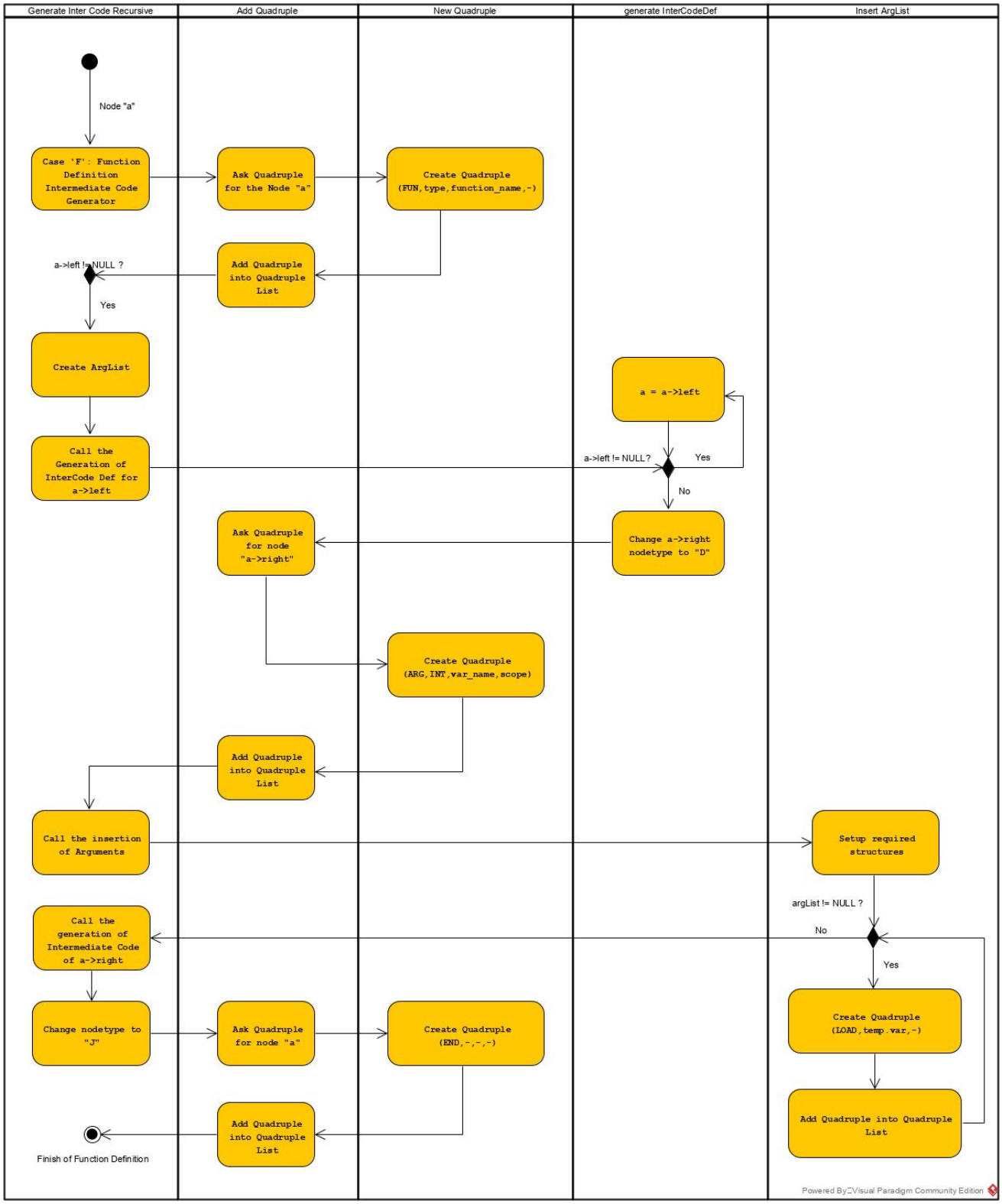
Figura 19 – Diagrama de Atividades para Nós do *if*



Fonte: Autoria própria

4.1.1.2.3 Geração de quádruplas para nós do tipo definição de função

Figura 20 – Diagrama de Atividades para Nós do tipo *Função*



Fonte: Autoria própria

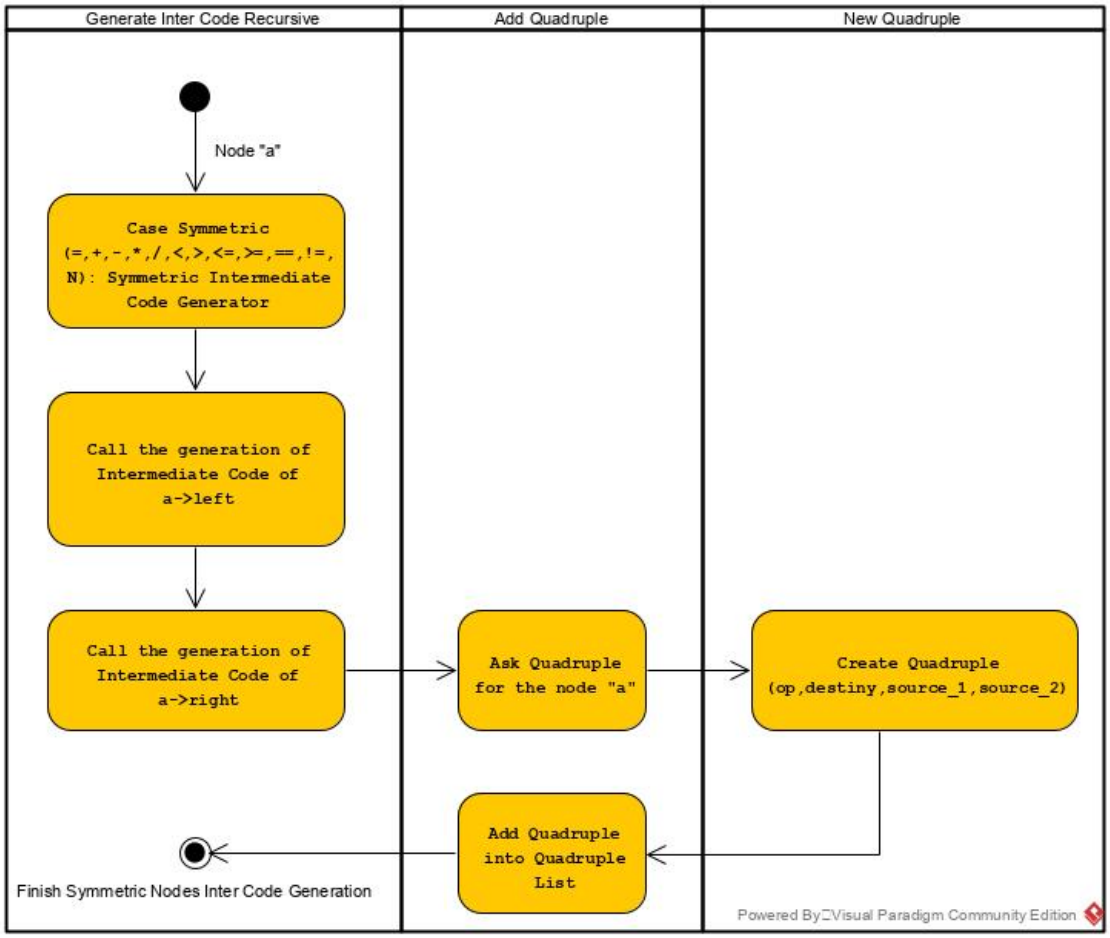
Quando encontramos um nó do tipo definição de função, iniciamos inserindo uma quádrupla do tipo $(FUN, type, function_name, -)$. Caso a função possua argumentos, então sua sub árvore esquerda deve ser diferente de NULL. Criamos , neste caso, uma lista de argumentos. Invocamos o método de definição da lista de argumentos para sub árvore esquerda. Dentro do método, enquanto a sub árvore esquerda é diferente de NULL, atribuímos ao nó suas sub árvore esquerda. Trocamos então o tipo para D, para que possamos inserir quádruplas do tipo $(ARG, INT, var_name, scope)$.

Em seguida chamamos a inserção dos argumentos. Para isso devemos inicializar as estruturas necessárias, e, enquanto a lista de argumentos for diferente de NULL, devemos inserir quádruplas do tipo $(LOAD, temp, var, -)$.

Por fim, trocamos o tipo do nó para J, para adicionarmos uma quádrupla de encerramento da definição da função: $(END, -, -, -)$.

4.1.1.2.4 Geração de quádruplas para nós do tipo simétrico

Figura 21 – Diagrama de Atividades para Nós do Tipo Simétrico



Fonte: Autoria própria

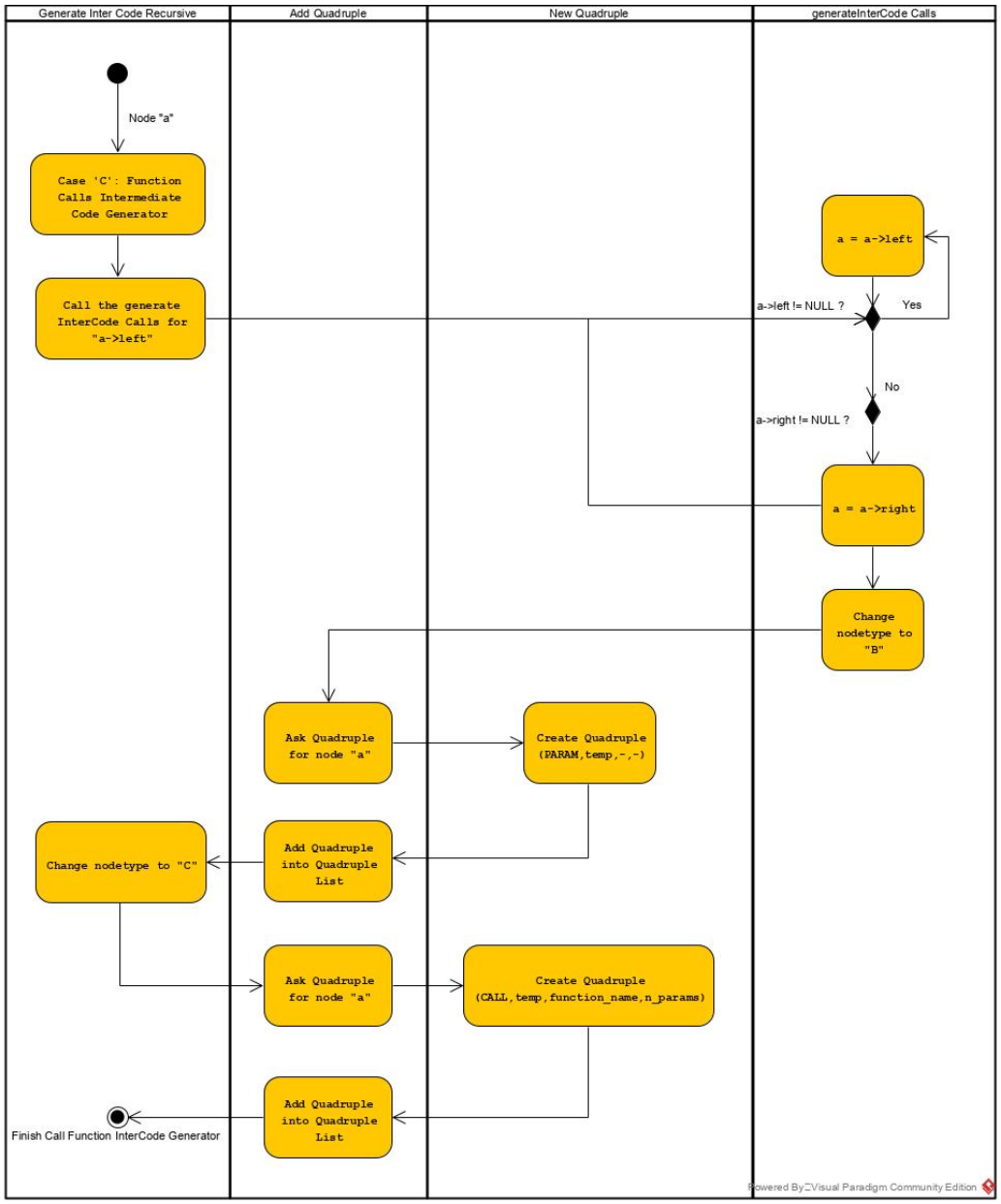
No caso de nós do tipo simétrico, sempre executamos o percurso em ordem, isto é, chamamos a geração do código intermediário para a subárvore esquerda, chamamos a geração do código intermediário para a sub árvore direita e, por fim, Criamos uma quádrupla do tipo $(op, destiny, source_1, source_2)$ e adicionamos a lista de quádruplas.

Definimos como nós do tipo simétrico os seguintes:

+, -, *, /, <, <=, >, >=, ==, !=, =.

4.1.1.2.5 Geração de quádruplas para nós do tipo chamada de função

Figura 22 – Diagrama de Atividades para Nós do Tipo Chamada de Função

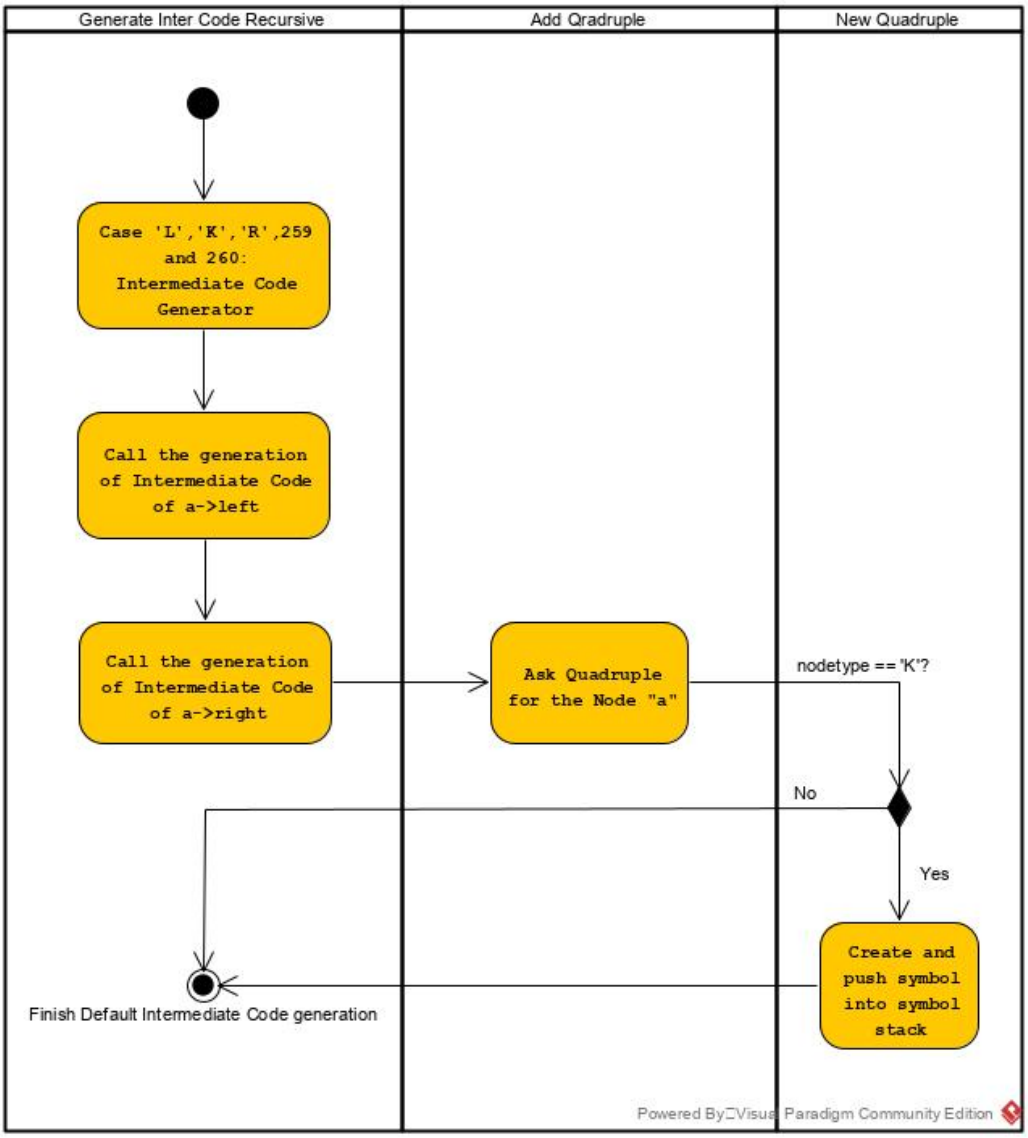


Fonte: Autoria própria

No caso de nós do tipo chamada de função, iniciamos percorrendo os argumentos, questão nas sub árvores direitas, trocamos o tipo do nó para B e assim inserimos uma quádrupla do tipo $(PARAM, temp, -, -)$. Após isso, trocamos o tipo do nó para C em inserimos então uma quádrupla do tipo $(CALL, temp, function_name, params)$, encerrando a geração do código intermediário no caso de chamada de função.

4.1.1.2.6 Geração de quádruplas para nós do tipo chamada de função outros nós

Figura 23 – Diagrama de Atividades para Outros Nós



Fonte: Autoria própria

Para o resto dos nós, isto é, nós do tipo lista (L), constante (K), retorno (R), tipo inteiro (259) e tipo void (260), a geração do código intermediário é bastante simples. Executa se o percurso em ordem, chamando a execução sob a sub árvore esquerda , depois

sobre a sub árvore direita e, se for um nó do tipo K, cria um símbolo e o coloca na pilha de símbolos.

4.1.2 Quádruplas Utilizadas

Temos quatro classes de quádruplas: quádruplas para declaração e chamada de funções, Quádruplas para leitura/escrita e declaração de variáveis, quádruplas de controle é quádruplos lógico-aritméticas. A explicação detalhada destas podem ser encontrada nos Apêndices. Lá explicamos para cada quádrupla a sua aplicação, o seu formato, os parâmetros e um exemplo.

As quádruplas para declaração e chamada de funções são:

- **(FUN, type, function_name, -)**
- **(ARG, INT, var, scope)**
- **(CALL, temp, function_name, n_params)**
- **(PARAM, temp, -, -)**
- **(END, function_name, -, -)**

As quádruplas para leitura/escrita e declaração de variáveis são:

- **(ALLOC, var_name, scope, [size])**
- **(LOAD, temp, var_name, [index])**
- **(LOAD, temp, var_name, [index])**
- **(STORE, temp, var_name, [index])**
- **(ASSIGN, temp_D, temp_S, -)**

As quádruplas de controle são:

- **(LABEL, label_name, -, -)**
- **(GOTO, label_name, -, -)**
- **(IFF, temp_test, label_branch, -)**

As quádruplas lógico-aritméticas são:

- **(ADD, temp_D, temp_S1, temp_S2)**

- **(SUB, temp_D, temp_S1, temp_S2)**
- **(MULT, temp_D, temp_S1, temp_S2)**
- **(DIV, temp_D, temp_S1, temp_S2)**
- **(GT, temp_D, temp_S1, temp_S2)**
- **(LT, temp_D, temp_S1, temp_S2)**
- **(GEQ, temp_D, temp_S1, temp_S2)**
- **(LEQ, temp_D, temp_S1, temp_S2)**
- **(EQ, temp_D, temp_S1, temp_S2)**
- **(NEQ, temp_D, temp_S1, temp_S2)**

O resultado proveniente deste módulo é o arquivo de código intermediário, que será utilizado como entrada para o módulo de geração de código *Assembly*.

4.2 Geração do Código *Assembly*

O módulo de geração de código *assembly* é responsável por traduzir as quádruplas geradas no código intermediário em *assembly*. Até agora, mesmo no código intermediário, todas as decisões foram voltadas a linguagem fonte, isto é, a linguagem *C-*. Agora, no código *assembly*, desenvolvemos um *assembly* adequado à arquitetura de pilha *AsterixForth*. Neste sentido, as instruções em assembly terão no máximo um argumento passado, por se tratar de uma arquitetura de pilha.

4.2.1 Modelagem

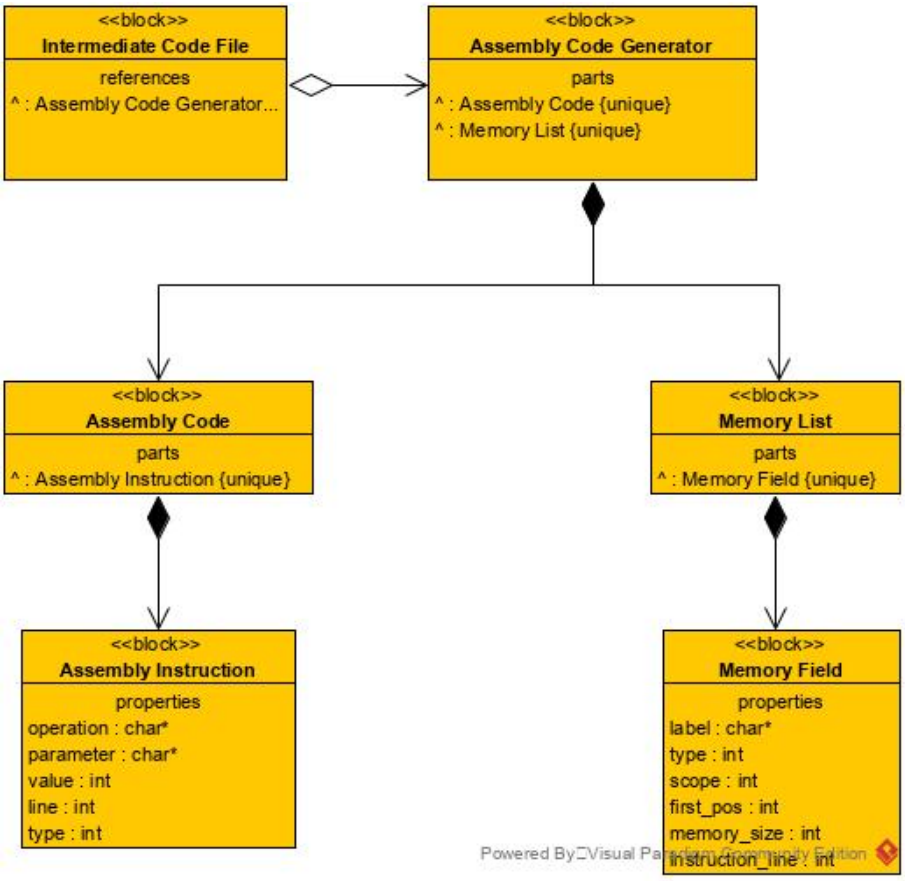
4.2.1.1 Diagrama de Blocos

O módulo de geração do código *assembly* é composto por duas estruturas principais: o **código *assembly*** e a **lista de memória**. O código *assembly* é composto por **instruções *assembly***, que armazenam a operação a ser executada, o parâmetro caso o tenha, o valor caso o tenha, a linha do código *assembly* e o tipo de instrução (se é de controle ou não).

A lista de memória é útil apenas para a geração do código binário, porém precisamos construí-la aqui. Ela é composta por **campos de memória**, que por sua vez armazenam o *label* (o que pode ser uma variável, uma função ou um rótulo de desvio e controle de execução), o tipo (que se aplica a variáveis e a funções), o escopo (novamente para variáveis e funções), a *first_pos*, que é a primeira posição daquela variável ou função na

memória, o *memory_size*, que armazena quantas posições conseguintes e, finalmente, a instruction_line, que é a instrução para qual aquele *label* desvio aponta.

Figura 24 – Diagrama de Blocos do Módulo de Geração do Código Intermediário*Assembly* do Compilador *Panoramix*



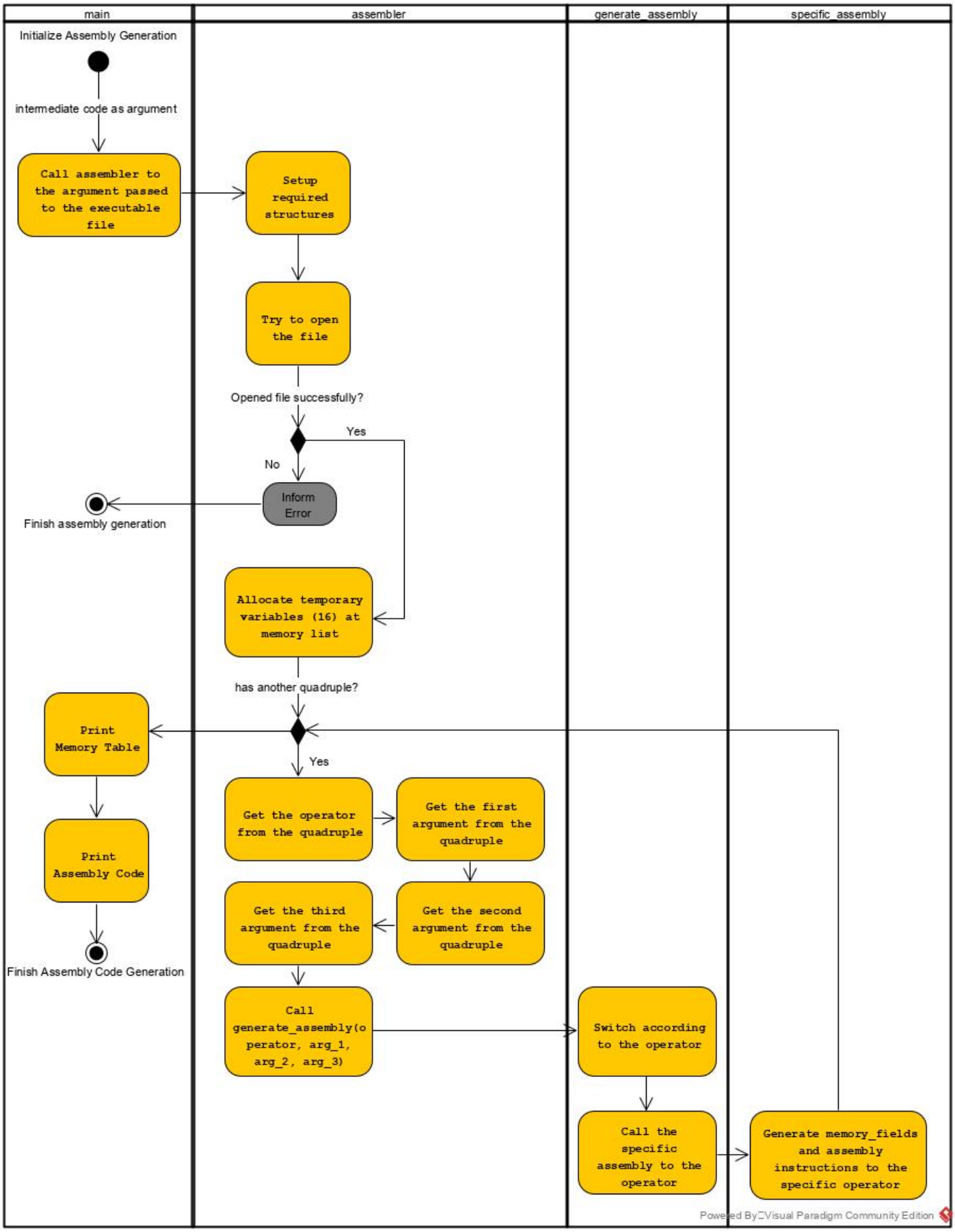
Fonte: Autoria própria

4.2.1.2 Diagramas de Atividades

Para a geração do código *assembly*, iniciamos chamando a execução do *assembler* para o argumento passado para o executável na linha de comando, isto é, pegamos o argumento em *argv*. Dentro da função *assembler*, inicializamos as estruturas necessárias, e tentamos abrir o arquivo passado. Caso o arquivo não abra corretamente, informamos o erro e encerramos a geração do código *assembly*. Caso abra corretamente, alocamos as 16 variáveis temporárias na lista de memória.

Iniciamos então um ciclo que irá executar enquanto houver quádruplas a serem lidas no arquivo. Para cada quádrupla irá se separar o *operator* e cada um dos 3 argumentos da quádrupla. Esta separação é feita por meio da função *strtok*, que serve para *tokenizar* a *string*. Com os argumentos e o operador separados, chamamos a função *generate_assembly*.

Figura 25 – Diagrama de Atividades para Nós do *if*



Fonte: Autoria própria

A função *generate assembly* é um *switch case* de acordo com operador. Em cada caso se chama a função específica de geração do código *assembly*, onde se irá gerar campos de memória e instruções *assembly* de acordo com o caso.

Quando não há mais quádruplas, a execução retorna para a função *main*, onde se irá solicitar a impressão da tabela de memórias e do código *assembly*, encerrando a geração de código *assembly*.

4.2.2 Funções de Geração de Código *Assembly* Específicas

Como dissemos na seção 4.2.1.2, para cada operador, temos uma função específica de geração do código *assembly*. Por isso, iremos agora passar por essas funções analisando o seu pseudo código e a visualização na pilha quando necessário. Por uma questão de facilitar a leitura é a comparação com as quádruplas, iremos seguir a mesma ordem com a qual as quádruplas foram apresentadas no apêndice.

4.2.2.1 Geração de *Assembly* para Quádruplas de Declaração e Chamada de Funções

4.2.2.1.1 Quádrupla *FUN*

Para quádruplas do tipo **FUN**, criamos um campo na memória armazenando nome da função, o tipo do seu retorno, a posição onde sua memória se inicia sem tamanho e a linha na qual aquela função se inicia, pois precisaremos desta informação quando esta função for chamada.

Algoritmo 1 *assembly_FUN(char* arg_1, char* arg_2)*

```
1: int type
2: se arg_1 = INT então  type = 1
3: senão type = 0
4: fim se
5: adiciona_memoria(arg_2, type, next_free_memory_pos, 0, assembly_code_line)
```

4.2.2.1.2 Quádrupla *ARG*

Em quádruplas do tipo **ARG**, já temos os argumentos na pilha é precisamos colocá-los na memória nas posições adequadas a função. Desta forma, empilhamos o endereço daquela variável é mandamos escrever o conteúdo da segunda posição da pilha no endereço definido na primeira posição.

Algoritmo 2 *assembly_ARG(char* arg_2)*

```
1: adiciona_assembly("PUSH",'&' + arg_2)
2: adiciona_assembly("!")
```

4.2.2.1.3 Quádrupla *CALL*

No caso de quádruplas do tipo **CALL**, começamos verificando se a função é a função *input*, pois neste caso precisamos fazer o tratamento especial de chamar *input read* e *input Load*, como solicita a arquitetura para instruções desse tipo. Em seguida se deve adicionar o endereço de memória onde este valor será salvo e mandar que escreva.

Caso a função seja a função *output*, precisamos chamar a função *output send* e *output print*.

Caso contrário, adicionamos o *assembly* de *Call* para essa função. Se a função tiver retorno do tipo *int*, adicionamos na pilha o endereço onde esse resultado deve ser escrito é mandamos escrever nele.

Algoritmo 3 *assembly_CALL(char* arg_1, char* arg_2)*

```
1: se arg_2 = input então
2: adiciona_assembly("INPUT READ")
3: adiciona_assembly("INPUT LOAD")
4: adiciona_assembly("PUSH",'&' + arg_1)
5: adiciona_assembly("!")
6: fim se
7: se arg_2 = output então
8: adiciona_assembly("OUTPUT SEND")
9: adiciona_assembly("OUTPUT PRINT")
10: senão
11:     adiciona_assembly("CALL",arg_2)
12:     se tipo(arg_2) = 1 então                    ▷ função do tipo INT
13:         adiciona_assembly("PUSH", '&' + arg_1)
14:         adiciona_assembly("!")
15:     fim se
16: fim se
```

4.2.2.1.4 Quádrupla *PARAM*

No caso de quádruplas do tipo parâmetro, se for um dígito, apenas adicionamos na pilha. Caso contrário, adicionamos no endereço na pilha é trazemos o seu valor para a pilha.

Algoritmo 4 *assembly_PARAM(char* arg_1)*

```
1: se arg_1[0] é digito então
2: adiciona_assembly("PUSH",arg_1)
3: senão
4:     adiciona_assembly("PUSH",'&' + arg_1)
5:     adiciona_assembly("@")
6: fim se
```

4.2.2.1.5 Quádrupla *RET*

No caso de quádruplas do tipo **ret**, se for um dígito, apenas adicionamos na pilha. Caso contrário, adicionamos no endereço na pilha é trazemos o seu valor para a pilha.

Algoritmo 5 *assembly_RET(char* arg_1)*

```
1: se arg_1[0] é digito então
2: adiciona_assembly("PUSH",arg_1)
3: senão
4:     adiciona_assembly("PUSH",'&' + arg_1)
5:     adiciona_assembly("@")
6: fim se
```

4.2.2.1.6 Quádrupla *END*

No caso de quádruplas do tipo **END**, se a função não for a função *main*, devemos inserir uma instrução de **exit** no assembly.

Algoritmo 6 *assembly_END(char* arg_1)*

```
1: se arg_1 != main então
2: adiciona_assembly("EXIT")
3: fim se
```

4.2.2.2 Geração de *Assembly* para Quádruplas de Leitura/Escrita e Declaração de Variáveis

4.2.2.2.1 Quádrupla *ALLOC*

No caso de quádruplas do tipo **ALLOC**, apenas devemos alocar a variável na memória. Os testes feitos se resumem a definir se a variável é do tipo *int* ou se é vetor e, se for vetor, qual o tamanho a ser alocado. Observa-se também o escopo da variável, podendo ser este global ou local.

Algoritmo 7 *assembly_ALLOC(char* arg_1, char* arg_2, char* arg_3)*

```
1: se arg_3 esta vazio então  type = 1                    ▷ var do tipo INT
2: size = 1
3: senão type = 2                                         ▷ var do tipo INT[ ]
4: size = arg_3 * 2
5: fim se
6: se arg_3 = vet_pointer então  size = 0
7: fim se
8: se arg_2 = GLOBAL então scope = 1
9: senão scope = 0
10: fim se
11: adiciona_memoria(arg_1, type, scope, next_free_memory_pos, size, -1);
```

4.2.2.2.2 Quádrupla *LOAD*

Para quádruplas do tipo **load**, se o argumento 3 for um endereço, então devemos trazer da memória o conteúdo deste endereço, adicionar a pilha o endereço base do vetor e somar os 2 para trazer o conteúdo da memória. Por fim adicionamos o endereço de destino e escrevemos na memória. Caso a variável seja do tipo inteiro, basta adicionarmos o endereço na pilha ler o conteúdo escrever no endereço de destino.

Algoritmo 8 *assembly_LOAD(char* arg_1, char* arg_2, char* arg_3)*

1: **se** arg_3[0] = '$' **então**
2: adiciona_assembly("PUSH",'&' + arg_3)
3: adiciona_assembly("@")
4: adiciona_assembly("PUSH",'&' + arg_2)
5: adiciona_assembly("+")
6: adiciona_assembly("@")
7: adiciona_assembly("PUSH",'&' + arg_1)
8: adiciona_assembly("!")
9: **senão**
10: adiciona_assembly("PUSH",'&' + arg_2)
11: adiciona_assembly("@")
12: adiciona_assembly("PUSH",'&' + arg_1)
13: adiciona_assembly("!")
14: **fim se**

4.2.2.2.3 Quádrupla *STORE*

Para quádruplas do tipo **store**, precisamos apenas trazer o conteúdo da memória escrevê-lo no endereço de destino.

Algoritmo 9 *assembly_STORE(char* arg_1, char* arg_2)*

1: adiciona_assembly("PUSH",'&' + arg_1)
2: adiciona_assembly("@")
3: adiciona_assembly("PUSH",'&' + arg_2)
4: adiciona_assembly("!")

4.2.2.2.4 Quádrupla *ASSIGN*

Para quádruplas do tipo **assign**, temos um comportamento muito parecido com as quádruplas *store*, excetuando-se o fato de que aparecem em ordem invertida os argumentos.

Algoritmo 10 *assembly_ASSIGN(char* arg_1, char* arg_2)*

1: adiciona_assembly("PUSH",'&' + arg_2)
2: adiciona_assembly("@")
3: adiciona_assembly("PUSH",'&' + arg_1)
4: adiciona_assembly("!")

4.2.2.3 Geração de *Assembly* para Quádruplas de Controle

4.2.2.3.1 Quádrupla *HALT*

Quádruplas do tipo **halt** são bem simples e simplesmente requerem a criação da instrução *halt* do *assembly*.

Algoritmo 11 *assembly_HALT()*

```
1: adiciona_assembly("HALT")
```

4.2.2.3.2 Quádrupla *LABEL*

Quádruplas do tipo **label** apenas escrevem na memória, salvando o rótulo do argumento 1 e a linha do *assembly* a qual aquele *label* se refere.

Algoritmo 12 *assembly_LABEL(char* arg_1)*

```
1: adiciona_memoria(arg_1, -1, 0, -1, 0 assembly_code_line)
```

4.2.2.3.3 Quádrupla *GOTO*

Quádruplas do tipo **goto** apenas requsitam que haja o desvio para aquele *label*. Note que é tratado como uma instrução *CALL*, pois é ela que faz desvios.

Algoritmo 13 *assembly_GOTO(char* arg_1)*

```
1: adiciona_assembly("CALL",arg_1)
```

4.2.2.3.4 Quádrupla *IFF*

Em quádruplas do tipo **iff** devemos inserir a variável ou valor de teste na pilha e então inserir a instrução *assembly* de *if*. Vale destacar que a condição de *VERDADEIRO* se dá no caso da variável no topo da pilha ter valor 0 e que qualquer outro valor é considerado como *FALSO*.

Algoritmo 14 *assembly_IFF(char* arg_1, char* arg_2)*

```
1: se arg_1[0] é digito então
2: adiciona_assembly("PUSH",arg_1)
3: senão
4:     adiciona_assembly("PUSH",'&' + arg_1)
5:     adiciona_assembly("@")
6: fim se
7: adiciona_assembly("IF",arg_2)
```

4.2.2.4 Geração de *Assembly* para Quádruplas Lógico-Aritméticas

4.2.2.4.1 Quádruplas *ADD*, *SUB*, *MULT*, *DIV*

Para quádrupla do tipo aritmético, o comportamento é idêntico para as operações de adição, subtração, multiplicação e divisão. Basta que adicionemos os valores na pilha e executemos a operação apropriada.

Algoritmo 15 *assembly_OP(char* arg_1, char* arg_2, char* arg_3)*

```
1: se arg_2[0] é digito então
2: adiciona_assembly("PUSH",arg_2)
3: senão
4:     adiciona_assembly("PUSH",'&' + arg_2)
5:     adiciona_assembly("@")
6: fim se
7: se arg_3[0] é digito então
8: adiciona_assembly("PUSH",arg_3)
9: senão
10:    adiciona_assembly("PUSH",'&' + arg_3)
11:    adiciona_assembly("@")
12: fim se
13: adiciona_assembly("OP")
14: adiciona_assembly("PUSH",arg_1)
15: adiciona_assembly("!")
```

4.2.2.4.2 Quádrupla *GT*

Algoritmo 16 *assembly_GT(char* arg_1, char* arg_2, char* arg_3)*

```
1: se arg_2[0] é digito então
2: adiciona_assembly("PUSH",arg_2)
3: senão
4:     adiciona_assembly("PUSH",'&' + arg_2)
5:     adiciona_assembly("@")
6: fim se
7: se arg_3[0] é digito então
8: adiciona_assembly("PUSH",arg_3)
9: senão
10:    adiciona_assembly("PUSH",'&' + arg_3)
11:    adiciona_assembly("@")
12: fim se
13: adiciona_assembly("LEQ")
14: adiciona_assembly("PUSH",arg_1)
15: adiciona_assembly("!")
```

Para quádruplas do tipo **GT**, iniciamos também inserindo os dois argumentos na pilha, mas precisamos proceder com a troca da ordem dos elementos para então testar se um é menor do que o outro.

Por exemplo: se queremos testar se $a > b$, perguntamos se $a \leq b$. Se o resultado for falso, significa que $a > b$, como queremos. Podemos fazer isso porque o teste *LEQ* retorna 0 se for falso.

4.2.2.4.3 Quádrupla *LT*

Algoritmo 17 *assembly_LT(char* arg_1, char* arg_2, char* arg_3)*

```
se arg_2[0] é digito então
adiciona_assembly("PUSH",arg_2)
senão
    adiciona_assembly("PUSH",'&' + arg_2)
    adiciona_assembly("@")
fim se
se arg_3[0] é digito então
adiciona_assembly("PUSH",arg_3)
senão
    adiciona_assembly("PUSH",'&' + arg_3)
    adiciona_assembly("@")
fim se
adiciona_assembly("-")
adiciona_assembly("DUP")
adiciona_assembly("PUSH",0)
adiciona_assembly("LEQ")
adiciona_assembly("SWAP")
adiciona_assembly("-")
adiciona_assembly("PUSH",0)
adiciona_assembly("SWAP")
adiciona_assembly("LEQ")
adiciona_assembly("- -")
adiciona_assembly("PUSH",1)
adiciona_assembly("-")
adiciona_assembly("PUSH",arg_1)
adiciona_assembly("!")
```

A quádrupla **LT** é bastante difícil de ser convertida em *assembly*, precisando de vários passos. Queremos uma expressão que $f(a, b) = 0 \Leftrightarrow a < b$. Vamos mostrar que a expressão $1 - ((0 \leq a - b) - (a - b \leq 0))$ cumpre essa função por meio de uma tabela verdade na Tabela 1:

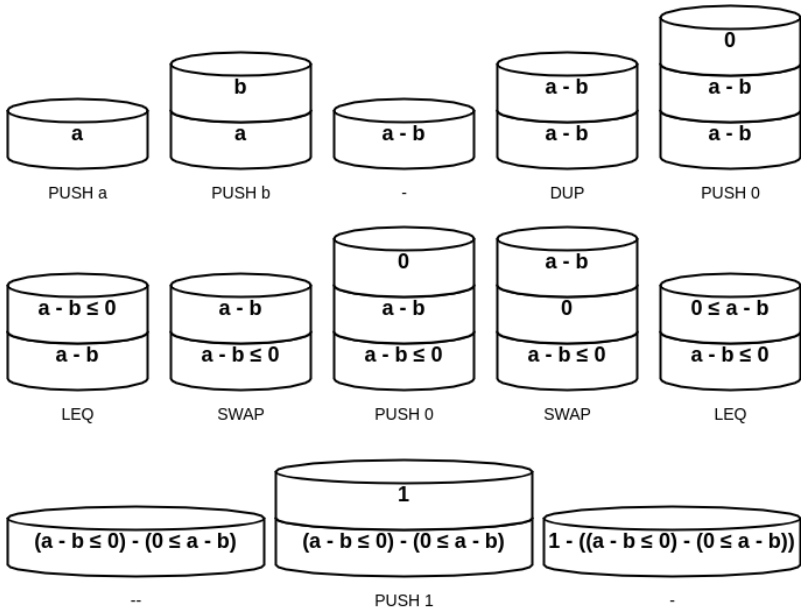
Fonte: Autoria Própria

$(a - b \leq 0)$	$(0 \leq a - b)$	$(a - b \leq 0)$ **-** $(0 \leq a - b)$	**1 -** $((a - b \leq 0)$ **-** $(0 \leq a - b))$
1	0	1	0
1	1	0	1
0	0	0	1
0	1	-1	2

Tabela 1 – Tabela-verdade para a quádrupla do tipo LT

De fato, $a < b$ se, e somente se, $a - b \leq 0$ e $0 \nleq a - b$.

Figura 26 – Geração do *Assembly* para uma Quádrupla do tipo LT Representada em Pilha



Fonte: Autoria própria

4.2.2.4.4 Quádrupla *GEQ*

Para quádruplas do tipo **GEQ**, trocamos a ordem dos elementos inseridos e inserimos uma instrução de *LEQ* e invertemos o *bit* do seu resultado.

Por exemplo: Queremos saber se $a \geq b$, perguntamos então se $b \leq a$.

Algoritmo 18 *assembly_GEQ(char* arg_1, char* arg_2, char* arg_3)*

1: **se** arg_2[0] é digito **então**
2: adiciona_assembly("PUSH",arg_2)
3: **senão**
4: adiciona_assembly("PUSH",'&' + arg_2)
5: adiciona_assembly("@")
6: **fim se**
7: **se** arg_3[0] é digito **então**
8: adiciona_assembly("PUSH",arg_3)
9: **senão**
10: adiciona_assembly("PUSH",'&' + arg_3)
11: adiciona_assembly("@")
12: **fim se**
13: adiciona_assembly("SWAP")
14: adiciona_assembly("LEQ")
15: adiciona_assembly("PUSH",1)
16: adiciona_assembly("-")
17: adiciona_assembly("PUSH",'&' + arg_1)
18: adiciona_assembly("!")

4.2.2.4.5 Quádrupla *LEQ*

Como a operação *leq* pertence ao conjunto de instruções do processador, sua sequência de instruções em *assembly* segue a estrutura das quádruplas aritméticas. A única diferença é que, como retorna 1 caso *VERDADEIRO*, precisamos adicionar 1 a pilha e subtrair o resultado para inverter o *bit*.

Algoritmo 19 *assembly_LEQ(char* arg_1, char* arg_2, char* arg_3)*

1: **se** arg_2[0] é digito **então**
2: adiciona_assembly("PUSH",arg_2)
3: **senão**
4: adiciona_assembly("PUSH",'&' + arg_2)
5: adiciona_assembly("@")
6: **fim se**
7: **se** arg_3[0] é digito **então**
8: adiciona_assembly("PUSH",arg_3)
9: **senão**
10: adiciona_assembly("PUSH",'&' + arg_3)
11: adiciona_assembly("@")
12: **fim se**
13: adiciona_assembly("LEQ")
14: adiciona_assembly("PUSH",1)
15: adiciona_assembly("-")
16: adiciona_assembly("PUSH",'&' + arg_1)
17: adiciona_assembly("!")

4.2.2.4.6 Quádrupla *EQ*

Para quádruplas do tipo **eq** precisamos utilizar um conjunto de outras instruções da arquitetura para contornar o fato de que não temos esta instrução diretamente. Para isso, fazemos o teste utilizando a instrução *XOR*, pois esta retorna zero sempre que os *bits* são iguais, que é exatamente o que queremos.

Algoritmo 20 *assembly_EQ(char* arg_1, char* arg_2, char* arg_3)*

```
1: se arg_2[0] é digito então
2: adiciona_assembly("PUSH",arg_2)
3: senão
4:     adiciona_assembly("PUSH",'&' + arg_2)
5:     adiciona_assembly("@")
6: fim se
7: se arg_3[0] é digito então
8: adiciona_assembly("PUSH",arg_3)
9: senão
10:    adiciona_assembly("PUSH",'&' + arg_3)
11:    adiciona_assembly("@")
12: fim se
13: adiciona_assembly("XOR")
14: adiciona_assembly("PUSH",'&' + arg_1)
15: adiciona_assembly("!")
```

4.2.2.4.7 Quádrupla *NEQ*

A quádrupla **NEQ** é também bastante difícil de ser convertida em *assembly*, precisando de vários passos. Queremos uma expressão que $f(a,b) = 0 \Leftrightarrow a \neq b$. Vamos mostrar que a expressão $1 - ((0 \leq a - b) \veebar (a - b \leq 0))$ cumpre essa função por meio de uma tabela verdade na Tabela 2:

Fonte: Autoria Própria

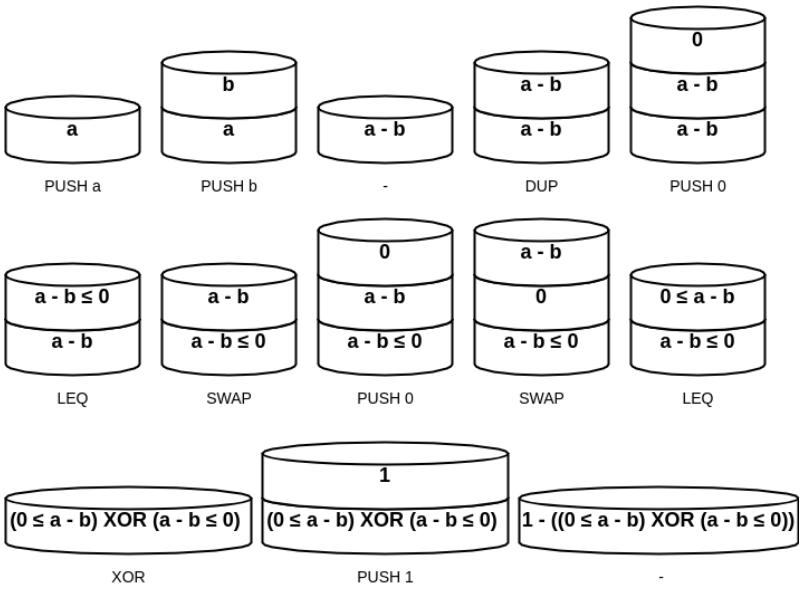
$(a - b \leq 0)$	$(0 \leq a - b)$	$(a - b \leq 0) \veebar (0 \leq a - b)$	**1 -** $((a - b \leq 0) \veebar (0 \leq a - b))$
1	0	1	0
1	1	0	1
0	0	0	1
0	1	1	0

Tabela 2 – Tabela-verdade para a quádrupla do tipo *NEQ*

Algoritmo 21 *assembly_NEQ(char* arg_1, char* arg_2, char* arg_3)*

```
se arg_2[0] é digito então
adiciona_assembly("PUSH",arg_2)
senão
    adiciona_assembly("PUSH",'&' + arg_2)
    adiciona_assembly("@")
fim se
se arg_3[0] é digito então
adiciona_assembly("PUSH",arg_3)
senão
    adiciona_assembly("PUSH",'&' + arg_3)
    adiciona_assembly("@")
fim se
adiciona_assembly("-")
adiciona_assembly("DUP")
adiciona_assembly("PUSH",0)
adiciona_assembly("LEQ")
adiciona_assembly("SWAP")
adiciona_assembly("PUSH",0)
adiciona_assembly("SWAP")
adiciona_assembly("LEQ")
adiciona_assembly("XOR")
adiciona_assembly("PUSH",1)
adiciona_assembly("-")
adiciona_assembly("PUSH",arg_1)
adiciona_assembly("!")
```

Figura 27 – Geração do *Assembly* para uma Quádrupla do tipo *NEQ* Representada em Pilha



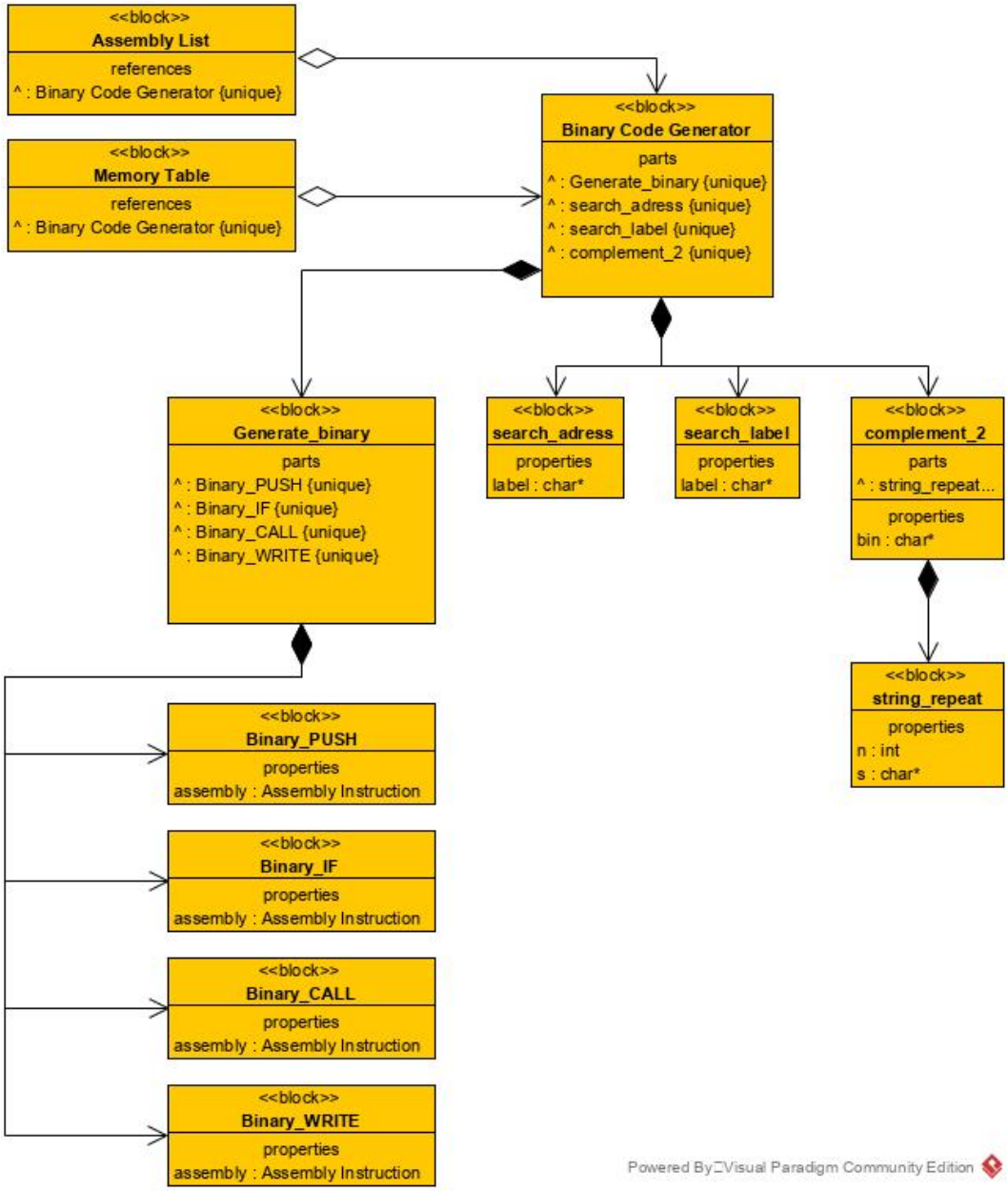
Fonte: Autoria própria

4.3 Geração do Código Executável (Binário)

4.3.1 Modelagem

4.3.1.1 Diagrama de Blocos

Figura 28 – Diagrama de Blocos do Módulo de Geração do Código Binário do Compilador *Panoramix*



Fonte: Autoria própria

A geração do código binário é responsável por traduzir o código *assembly* para o código binário, que é por sua vez entendido pelo processador *AsterixForth*. Este módulo requer o código assembly gerado e a tabela de memórias para definir os endereços e os desvios

necessários. Como podemos ver na figura 28, a geração do código *assembly* possui 4 métodos principais: **generate_binary**, **search_address**, **search_label** e **complement_2**.

O método **generate_binary** é bastante simples, visto que a maioria das instruções acaba sendo direta, sem a necessidade de definir os *bits* específicos. Porém, no caso das instruções *PUSH*, *IF*, *CALL* e *WRITE* são necessários outros tratamentos por questões de valor de endereço e imediato. Neste sentido surgem as funções auxiliares **search_adress**, **search_label** e **complement_2**.

A função de complemento de 2 merece atenção especial pois ela é necessária para que os desvios relativos possam ser feitos tanto para avançar quanto para retroceder nas instruções definidas. Observa-se o fato de que, sendo uma arquitetura de apenas 16 *bits*, se trabalhássemos com desvios absolutos teríamos um alcance muito menor do que com relação ao desvio relativo. Além disso, pelo princípio da localidade, é pouco provável que tentemos desvios para longe, sendo então o desvio relativo bastante prático.

4.3.1.2 Diagrama de Atividades

O diagrama de atividades na figura 31 consiste em uma sequência de *if-else* para gerar a quádrupla específica para cada instrução. Em algumas quádruplas precisamos de atenção, como é o caso das instruções *assembly* de **função** que não são impressas na lista *assembly* e que servem para acelerar a busca na tabela por definir qual o escopo.

No caso de instruções do tipo ***push***, precisamos averiguar se estamos empilhando um valor ou o endereço, é colocar este valor em binário em uma palavra de 16 *bits*. É interessante destacar que a função para converter o número para binário foi converter o valor para a *string* e usar o conversor *itoa* com base 2.

No caso da instruções do tipo ***if***, colocamos os 5 primeiros *bits* que a definem e buscamos a linha para a qual deve desviar buscando-a com a função *search_label*. Calculando a diferença da linha corrente com essa nova linha obtida, se o valor for menor do que zero, precisamos então calcular o seu complemento de 2 para então poder completar a palavra da instrução.

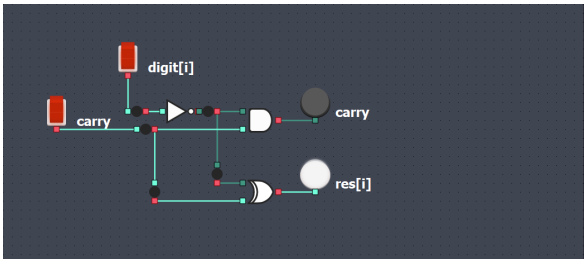


Figura 29 – Representação de um *bit* complemento de 2

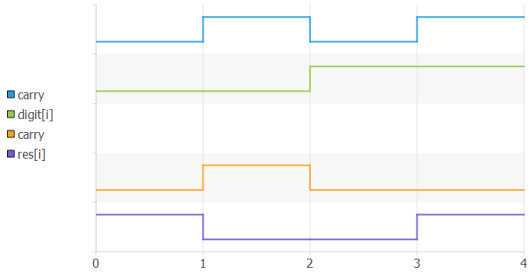
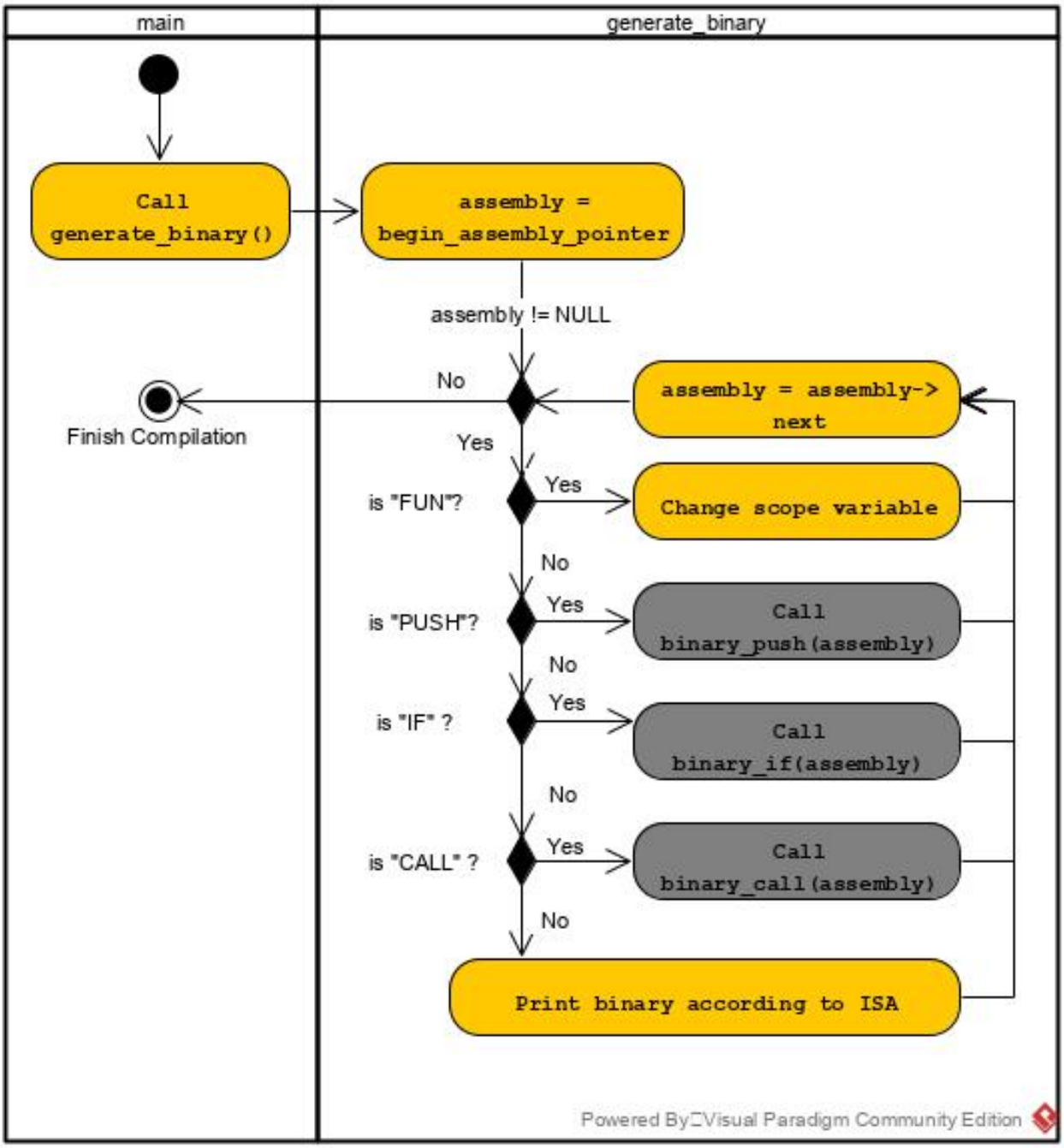


Figura 30 – *Waveform* - complemento de 2

A instrução de complemento de 2 é bastante interessante justamente devido a sua simplicidade pois ela é estruturada da mesma forma aprendida em circuitos digitais, como podemos ver na representação em 29 e comprovar pela forma de onda em 30, feitas com o *Software Wired Panda* (4).

Instruções do tipo CALL tem comportamento idêntico às instruções if, excetuando-se um bit do prefixo.

Figura 31 – Diagrama de Atividades para a Geração do Código Binário do Compilador *Panoramix*



Fonte: Autoria própria

4.4 Gerenciamento de Memória

Como apresentado na Figura 24, a tabela de memória final é propriamente estrutura durante a geração do código *assembly*.

Como por exemplo para o código:

```
void main(void){
        int x;
        x = 1;
        while(x < 10){
                x = x + 1;
        }
}
```

Algoritmo 4.1 – Exemplo de Gerenciamento de Memória

Obtemos a tabela:

```
|-------------------------------------------------------------------|
|                            Memory Table                           |
|-------------------------------------------------------------------|
|    label    |  type  |  scope   | first_pos  |  size  | inst_line |
|-------------------------------------------------------------------|
|         $t0 |  INT   | GLOBAL   |          0 |      2 |         - |
|         $t1 |  INT   | GLOBAL   |          2 |      2 |         - |
|         $t2 |  INT   | GLOBAL   |          4 |      2 |         - |
|         $t3 |  INT   | GLOBAL   |          6 |      2 |         - |
|         $t4 |  INT   | GLOBAL   |          8 |      2 |         - |
|         $t5 |  INT   | GLOBAL   |         10 |      2 |         - |
|         $t6 |  INT   | GLOBAL   |         12 |      2 |         - |
|         $t7 |  INT   | GLOBAL   |         14 |      2 |         - |
|         $t8 |  INT   | GLOBAL   |         16 |      2 |         - |
|         $t9 |  INT   | GLOBAL   |         18 |      2 |         - |
|        $t10 |  INT   | GLOBAL   |         20 |      2 |         - |
|        $t11 |  INT   | GLOBAL   |         22 |      2 |         - |
|        $t12 |  INT   | GLOBAL   |         24 |      2 |         - |
|        $t13 |  INT   | GLOBAL   |         26 |      2 |         - |
|        $t14 |  INT   | GLOBAL   |         28 |      2 |         - |
|        $t15 |  INT   | GLOBAL   |         30 |      2 |         - |
|-------------------------------------------------------------------|
|        main |  VOID  | FUNCTION |         32 |      - |         0 |
|           x |  INT   | LOCAL    |         32 |      2 |         - |
|          L1 |   -    | LOCAL    |          - |      - |        13 |
|          L2 |   -    | LOCAL    |          - |      - |        59 |
|-------------------------------------------------------------------|
```

A tabela carrega, em termos da memória propriamente dita: os ***labels***, que são os rótulos das variáveis; o ***scope***, que diz se o acesso é local ou global; a ***first_pos***, que a primeira posição ocupada da memória para aquela variável; e ***size***, que é a quantidade de *bytes* ocupados.

5 Exemplos

5.1 Exemplo 1 - Fatorial

O primeiro exemplo que trabalharemos é o código do **fatorial** de um número passado pelo usuário. É um código bastante interessante porque aborda tópicos como entrada do usuário, chamadas de função, desvios e recursividade em poucas linhas de código.

5.1.1 Código Fonte

Neste código temos na função *main* a leitura da entrada do usuário com a função *input* e a impressão do cálculo do fatorial do número passado.

A função de fatorial (*fact*) tem por caso base números menores que zero, retornando 1 nestes casos. Caso seja maior do que 0, ele retorna $n \times fact(n-1)$, o que vai gerar o produto desejado $fact(n) = n \times (n-1) \times \ldots 2 \times 1$.

```
int fact(int n){
   if(n <= 0){
      return 1;
   }
   return n * fact(n - 1);
}

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

Algoritmo 5.1 – Código Fonte do Código do Fatorial fact.txt

5.1.2 Código Intermediário

O código intermediário para este código gera 27 quádruplas. Vamos apresentar por função o código gerado.

No Algoritmo 5.2, referente à função *fact*, podemos observar nas linhas 1 e 14 as quádruplas de início e fim da função *fact*. A quádrupla da linha 2 indica que o parâmetro n deve ser carregado em $\$t1$. As quádrupla 3 e 4 se encarregam de testar se o parâmetro n passado é menor do que 0. Se o for, retorna 1 na quádrupla 5. Caso contrário, desvia para a linha 6 e prepara a chamada recursiva da função. Para isso, carrega n em $\$t3$ e $n-1$ em $\$t5$ para poder chamar a função para $n-1$ e multiplicar o resultado obtido por n, retornando na linha 13.

```
1  (FUN,INT,fact,-)
2  (LOAD,$t1,n,-)
3  (LEQ,$t2,$t1,0)
4  (IFF,$t2,L1,-)
5  (RET,1,-,-)
6  (LABEL,L1,-,-)
7  (LOAD,$t3,n,-)
8  (LOAD,$t4,n,-)
9  (SUB,$t5,$t4,1)
10 (PARAM,$t5,-,-)
11 (CALL,$t6,fact,1)
12 (MULT,$t7,$t3,$t6)
13 (RET,$t7,-,-)
14 (END,fact,-,-)
```

Algoritmo 5.2 – Código Intermediário fact.txt - função fact

No algoritmo 5.3, temos na linha 1 e na linha 12 as demarcações de início e fim dá função *main*. nas linhas 2 a 4, temos a leitura da entrada do usuário na variável n, para que possamos colocá-la de parâmetro para a função *fact*. Depois disso, passamos este resultado como parâmetro para a função de *output*. A linha 13 indica o final do programa.

```
1  (FUN,VOID,main,-)
2  (ALLOC,n,main,-)
3  (LOAD,$t8,n,-)
4  (CALL,$t9,input,0)
5  (ASSIGN,$t8,$t9,-)
6  (STORE,$t8,n,-)
7  (LOAD,$t10,n,-)
8  (PARAM,$t10,-,-)
9  (CALL,$t11,fact,1)
10 (PARAM,$t11,-,-)
11 (CALL,$t12,output,1)
12 (END,main,-,-)
13 (HALT,-,-,-)
```

Algoritmo 5.3 – Código Intermediário fact.txt - função main

5.1.3 Código *Assembly*

Os algoritmos 5.6 e 5.7 são referentes à função *fact*.

No algoritmo 5.6 começamos com a inserção na pilha do endereço de n para que possamos então ler o seu conteúdo e colocar na pilha com a instrução @. Colocamos na pilha então o endereço de $t1$ e escrevemos o valor de n lá. Trazemos novamente o valor de n para a pilha para que possamos compará-lo com o valor de 0. Com o resultado em mãos, podemos fazer o teste de desvio em *IF* para definir se apenas colocaremos o valor 1 na pilha ou se iremos prosseguir com a chamada recursiva da função na linha 17. Caso prossigamos, então precisamos trazer n novamente da memória para poder colocá-lo na pilha, e depois, subtraindo 1 dele, possamos fazer a chamada recursiva. Por fim, isto é, na volta da recursão, multiplicamos o resultado obtido com n e retornamos a execução para a função que a chamou, de acordo com o endereço de retorno na pilha de retornos.

```
PUSH &n
@
PUSH &$t1
!
PUSH &$t1
@
PUSH 0
LEQ
PUSH 1
-
PUSH &$t2
!
PUSH &$t2
@
IF L1
PUSH 1
EXIT
PUSH &n
@
PUSH &$t3
!
PUSH &n
@
```

Algoritmo 5.4 – Assembly-função fact-1

```
PUSH &$t4
!
PUSH &$t4
@
PUSH 1
-
PUSH &$t5
!
PUSH &$t5
@
CALL fact
PUSH &$t6
!
PUSH &$t3
@
PUSH &$t6
@
*
PUSH &$t7
!
PUSH &$t7
@
EXIT
```

Algoritmo 5.5 – Assembly-função fact-2

A função *main* inicia colocando o endereço de n na pilha, lendo o seu conteúdo e escrevendo na memória $\$t8$. Note que esses passos não eram necessários, pois ainda não temos nada em n, porém esse é o padrão não otimizado de utilização de dados.

Depois disso ele executa as duas funções necessárias para executar a leitura da entrada do usuário para então colocá-la em $\$t9$. Atribui então este valor para n para poder passá-lo de parâmetro e chamar a função *fact*. Por fim, o resultado é mandando para impressão no *display* de 7 segmentos com as funções de *output* e encerra com *halt*.

```
PUSH &n
@
PUSH &$t8
!
INPUT READ
INPUT LOAD
PUSH &$t9
!
PUSH &$t9
@
PUSH &$t8
!
PUSH &$t8
@
PUSH &n
```

Algoritmo 5.6 – Assembly-função main-1

```
!
PUSH &n
@
PUSH &$t10
!
PUSH &$t10
@
CALL fact
PUSH &$t11
!
PUSH &$t11
@
OUTPUT SEND
OUTPUT PRINT
HALT
```

Algoritmo 5.7 – Assembly-função main-2

5.1.4 Código executável

Para o código executável, nos Algoritmos 5.8 e 5.9 temos o código referente à função *fact*. Colocamos as colunas lado a lado pois o binário apresentado corresponde ao *assembly* da mesma linha. Alguns exemplos interessantes são o *push* de endereço, que de fato coincidem com os endereços na tabela e o tratamento para fazer o *call* novamente, e o produto com o resultado após a chamada recursiva para fazer a multiplicação.

```
1  PUSH &n
2  @
3  PUSH &$t1
4  !
5  PUSH &$t1
6  @
7  PUSH 0
8  LEQ
9  PUSH 1
10 -
11 PUSH &$t2
12 !
13 PUSH &$t2
14 @
15 IF L1
16 PUSH 1
17 EXIT
18 PUSH &n
19 @
20 PUSH &$t3
21 !
22 PUSH &n
23 @
24 PUSH &$t4
25 !
26 PUSH &$t4
27 @
28 PUSH 1
29 -
30 PUSH &$t5
31 !
32 PUSH &$t5
33 @
34 CALL fact
35 PUSH &$t6
36 !
37 PUSH &$t3
38 @
39 PUSH &$t6
40 @
41 *
42 PUSH &$t7
43 !
44 PUSH &$t7
45 @
46 EXIT
```

Algoritmo 5.8 – Assembly-função fact

```
1  0000000000100000
2  1110100000000000
3  0000000000000010
4  1111000000000000
5  0000000000000010
6  1110100000000000
7  0000000000000000
8  1000000110000000
9  0000000000000000
10 1000010000000000
11 0000000000000100
12 1111000000000000
13 0000000000000100
14 1110100000000000
15 1001100000000011
16 0000000000000001
17 1000000000100000
18 0000000000100000
19 1110100000000000
20 0000000000000110
21 1111000000000000
22 0000000000100000
23 1110100000000000
24 0000000000001000
25 1111000000000000
26 0000000000001000
27 1110100000000000
28 0000000000000001
29 1000010000000000
30 0000000000001010
31 1111000000000000
32 0000000000001010
33 1110100000000000
34 1010011111011111
35 0000000000001100
36 1111000000000000
37 0000000000000110
38 1110100000000000
39 0000000000001100
40 1110100000000000
41 1000000100000000
42 0000000000001110
43 1111000000000000
44 0000000000001110
45 1110100000000000
46 1000000000100000
```

Algoritmo 5.9 – Executável-função fact

Nos algoritmos 5.36 e 5.37 temos o código referente à função *main*. O ponto mais interessante é com relação à leitura da entrada do usuário e a impressão do resultado com as funções de *output*.

```
PUSH &n
@
PUSH &$t8
!
INPUT READ
INPUT LOAD
PUSH &$t9
!
PUSH &$t9
@
PUSH &$t8
!
PUSH &$t8
@
PUSH &n
!
PUSH &n
@
PUSH &$t10
!
PUSH &$t10
@
CALL fact
PUSH &$t11
!
PUSH &$t11
@
OUTPUT SEND
OUTPUT PRINT
HALT
```

Algoritmo 5.10 – Assembly-função main

```
0000000000100000
1110100000000000
0000000000010000
1111000000000000
1100000000000001
1100000000000000
0000000000010010
1111000000000000
0000000000010010
1110100000000000
0000000000010000
1111000000000000
0000000000010000
1110100000000000
0000000000100000
1111000000000000
0000000000100000
1110100000000000
0000000000010100
1111000000000000
0000000000010100
1110100000000000
1010011110111011
0000000000010110
1111000000000000
0000000000010110
1110100000000000
0000000000000000
1101000000000000
1110000000000000
```

Algoritmo 5.11 – Executável-função main

5.1.5 Relação entre os códigos

Quádrupla	**Assembly**	**Executável**
(FUN,INT,fact,-)		
(LOAD,$t1,n,-)	PUSH &n	0000000000100000
	@	1110100000000000
	PUSH &$t1	0000000000000010
	!	1111000000000000
(LEQ,$t2$,t1,0)	PUSH &$t1	0000000000000010
	@	1110100000000000
	PUSH 0	0000000000000000
	LEQ	1000000110000000
	PUSH 1	0000000000000000

	-	1000010000000000
	PUSH &$t2	0000000000000100
	!	1111000000000000
(IFF,$t2,L1,-)	PUSH &$t2	0000000000000100
	@	1110100000000000
	IF L1	1001100000000011
	PUSH 1	0000000000000001
(RET,1,-,-)	EXIT	1000000000100000
(LABEL,L1,-,-)		
(LOAD,$t3,n,-)	PUSH &n	0000000000100000
	@	1110100000000000
	PUSH &$t3	0000000000000110
	!	1111000000000000
(LOAD,$t4,n,-)	PUSH &n	0000000000100000
	@	1110100000000000
	PUSH &$t4	0000000000001000
	!	1111000000000000
(SUB,$t5$,t4,1)	PUSH &$t4	0000000000001000
	@	1110100000000000
	PUSH 1	0000000000000001
	-	1000010000000000
	PUSH &$t5	0000000000001010
	!	1111000000000000
(PARAM,$t5,-,-)	PUSH &$t5	0000000000001010
	@	1110100000000000
(CALL,$t6,fact,1)	CALL fact	1010011111011111
	PUSH &$t6	0000000000001100
	!	1111000000000000
(MULT,$t7$,t3,$t6)	PUSH &$t3	0000000000000110
	@	1110100000000000
	PUSH &$t6	0000000000001100
	@	1110100000000000
	*	1000000100000000
	PUSH &$t7	0000000000001110
	!	1111000000000000
(RET,$t7,-,-)	PUSH &$t7	0000000000001110
	@	1110100000000000

	EXIT	1000000000100000
(END,fact,-,-)	EXIT	1000000000100000
(FUN,VOID,main,-)		
(ALLOC,n,main,-)		
(LOAD,$t8,n,-)	PUSH &n	0000000000100000
	@	1110100000000000
	PUSH &$t8	0000000000010000
	!	1111000000000000
(CALL,$t9,input,0)	INPUT READ	1100000000000001
	INPUT LOAD	1100000000000000
	PUSH &$t9	0000000000010010
	!	1111000000000000
(ASSIGN,*t*8,t9,-)	PUSH &$t9	0000000000010010
	@	1110100000000000
	PUSH &$t8	0000000000010000
	!	1111000000000000
(STORE,$t8,n,-)	PUSH &$t8	0000000000010000
	@	1110100000000000
	PUSH &n	0000000000100000
	!	1111000000000000
(LOAD,$t10,n,-)	PUSH &n	0000000000100000
	@	1110100000000000
	PUSH &$t10	0000000000010100
	!	1111000000000000
(PARAM,$t10,-,-)	PUSH &$t10	0000000000010100
	@	1110100000000000
(CALL,$t11,fact,1)	CALL fact	1010011110111011
	PUSH &$t11	0000000000010110
	!	1111000000000000
(PARAM,$t11,-,-)	PUSH &$t11	0000000000010110
	@	1110100000000000
(CALL,$t12,output,1)	OUTPUT SEND	0000000000000000
	OUTPUT PRINT	1101000000000000
(END,main,-,-)		
(HALT,-,-,-)	HALT	1110000000000000

Tabela 3 – Relação entre os códigos intermediário, *assembly* e executável para o código *fact.txt*

5.2 Exemplo 2 - Sort

5.2.1 Código Fonte

```
/* programa para ordenacao por selecao de
   uma matriz com dez elementos. */

int vet[ 10 ];

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

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

void main(void)
{
        int i;
        i = 0;
        while (i < 10){
                vet[i] = input();
                i = i + 1;
        }
        sort(vet,0,10);
        i = 0;
        while (i < 10){
                output(vet[i]);
                i = i + 1;
        }
}
```

5.2.2 Código Intermediário

```
(ALLOC,vet,GLOBAL,10)
(FUN,INT,minloc,-)
(ARG,INT,a,minloc)
(ARG,INT,low,minloc)
```

```
(ARG,INT,high,minloc)
(LOAD,$t1,a,-)
(LOAD,$t2,low,-)
(LOAD,$t3,high,-)
(ALLOC,i,minloc,-)
(ALLOC,x,minloc,-)
(ALLOC,k,minloc,-)
(LOAD,$t4,k,-)
(LOAD,$t5,low,-)
(ASSIGN,$t4,$t5,-)
(STORE,$t4,k,-)
(LOAD,$t6,x,-)
(LOAD,$t7,low,-)
(LOAD,$t8,a,$t7)
(ASSIGN,$t6,$t8,-)
(STORE,$t6,x,-)
(LOAD,$t9,i,-)
(LOAD,$t10,low,-)
(ADD,$t11,$t10,1)
(ASSIGN,$t9,$t11,-)
(STORE,$t9,i,-)
(LABEL,L1,-,-)
(LOAD,$t12,i,-)
(LOAD,$t13,high,-)
(LT,$t14,$t12,$t13)
(IFF,$t14,L2,-)
(LOAD,$t15,i,-)
(LOAD,$t0,a,$t15)
(LOAD,$t1,x,-)
(LT,$t2,$t0,$t1)
(IFF,$t2,L3,-)
(LOAD,$t3,x,-)
(LOAD,$t4,i,-)
(LOAD,$t5,a,$t4)
(ASSIGN,$t3,$t5,-)
(STORE,$t3,x,-)
(LOAD,$t6,k,-)
(LOAD,$t7,i,-)
(ASSIGN,$t6,$t7,-)
(STORE,$t6,k,-)
(LABEL,L3,-,-)
(LOAD,$t8,i,-)
(LOAD,$t9,i,-)
(ADD,$t10,$t9,1)
(ASSIGN,$t8,$t10,-)
(STORE,$t8,i,-)
(GOTO,L1,-,-)
(LABEL,L2,-,-)
(RET,$t3,-,-)
(LOAD,$t11,k,-)
(END,minloc,-,-)
(FUN,VOID,sort,-)
(ALLOC,a,sort,)
(ALLOC,low,sort,-)
(ALLOC,high,sort,-)
(ALLOC,i,sort,-)
(ALLOC,k,sort,-)
(LOAD,$t12,i,-)
(LOAD,$t13,low,-)
(ASSIGN,$t12,$t13,-)
```

```
(STORE,$t12,i,-)
(LABEL,L4,-,-)
(LOAD,$t14,i,-)
(LOAD,$t15,high,-)
(SUB,$t0,$t15,1)
(LT,$t1,$t14,$t0)
(IFF,$t1,L5,-)
(ALLOC,t,sort,-)
(LOAD,$t2,k,-)
(LOAD,$t3,a,$t2)
(PARAM,$t3,-,-)
(LOAD,$t4,i,-)
(PARAM,$t4,-,-)
(LOAD,$t5,high,-)
(PARAM,$t5,-,-)
(CALL,$t6,minloc,3)
(ASSIGN,$t11,$t6,-)
(STORE,$t11,k,-)
(LOAD,$t7,t,-)
(LOAD,$t8,k,-)
(LOAD,$t9,a,$t8)
(ASSIGN,$t7,$t9,-)
(STORE,$t7,t,-)
(LOAD,$t10,k,-)
(LOAD,$t11,a,$t10)
(LOAD,$t12,i,-)
(LOAD,$t13,a,$t12)
(ASSIGN,$t11,$t13,-)
(STORE,$t11,a,$t10)
(LOAD,$t14,i,-)
(LOAD,$t15,a,$t14)
(LOAD,$t0,t,-)
(ASSIGN,$t15,$t0,-)
(STORE,$t15,a,$t14)
(LOAD,$t1,i,-)
(LOAD,$t2,i,-)
(ADD,$t3,$t2,1)
(ASSIGN,$t1,$t3,-)
(STORE,$t1,i,-)
(GOTO,L4,-,-)
(LABEL,L5,-,-)
(END,sort,-,-)
(FUN,VOID,main,-)
(ALLOC,i,main,-)
(LOAD,$t4,i,-)
(ASSIGN,$t4,0,-)
(STORE,$t4,i,-)
(LABEL,L6,-,-)
(LOAD,$t5,i,-)
(LT,$t6,$t5,10)
(IFF,$t6,L7,-)
(LOAD,$t7,i,-)
(LOAD,$t8,vet,$t7)
(CALL,$t9,input,0)
(ASSIGN,$t8,$t9,-)
(STORE,$t8,vet,$t7)
(LOAD,$t10,i,-)
(LOAD,$t11,i,-)
(ADD,$t12,$t11,1)
(ASSIGN,$t10,$t12,-)
```

```
(STORE,$t10,i,-)
(GOTO,L6,-,-)
(LABEL,L7,-,-)
(LOAD,$t13,vet,$t2)
(PARAM,$t13,-,-)
(PARAM,0,-,-)
(PARAM,10,-,-)
(CALL,$t14,sort,3)
(LOAD,$t15,i,-)
(ASSIGN,$t15,0,-)
(STORE,$t15,i,-)
(LABEL,L8,-,-)
(LOAD,$t0,i,-)
(LT,$t1,$t0,10)
(IFF,$t1,L9,-)
(LOAD,$t2,i,-)
(LOAD,$t3,vet,$t2)
(PARAM,$t3,-,-)
(CALL,$t4,output,1)
(LOAD,$t5,i,-)
(LOAD,$t6,i,-)
(ADD,$t7,$t6,1)
(ASSIGN,$t5,$t7,-)
(STORE,$t5,i,-)
(GOTO,L8,-,-)
(LABEL,L9,-,-)
(END,main,-,-)
(HALT,-,-,-)
```

5.2.3 Códigos *Assembly* e Executável

```
PUSH &high
!
PUSH &low
!
PUSH &high
@
PUSH &$t1
!
PUSH &low
@
PUSH &$t2
!
PUSH &k
@
PUSH &$t3
!
PUSH &low
@
PUSH &$t4
!
PUSH &$t4
@
PUSH &$t3
!
PUSH &$t3
```

Algoritmo 5.12 – Assembly

```
0000000001010000
1111000000000000
0000000001001110
1111000000000000
0000000001010000
1110100000000000
0000000000000010
1111000000000000
0000000001001110
1110100000000000
0000000000000100
1111000000000000
0000000001001100
1110100000000000
0000000000000110
1111000000000000
0000000001001110
1110100000000000
0000000000001000
1111000000000000
0000000000001000
1110100000000000
0000000000000110
1111000000000000
0000000000000110
```

Algoritmo 5.13 – Executável

```
26 @
27 PUSH &k
28 !
29 PUSH &x
30 @
31 PUSH &$t5
32 !
33 PUSH &low
34 @
35 PUSH &$t6
36 !
37 PUSH &$t6
38 @
39 PUSH &a
40 +
41 @
42 PUSH &$t7
43 !
44 PUSH &$t7
45 @
46 PUSH &$t5
47 !
48 PUSH &$t5
49 @
50 PUSH &x
51 !
52 PUSH &i
53 @
54 PUSH &$t8
55 !
56 PUSH &low
57 @
58 PUSH &$t9
59 !
60 PUSH &$t9
61 @
62 PUSH 1
63 +
64 PUSH &$t10
65 !
66 PUSH &$t10
67 @
68 PUSH &$t8
69 !
70 PUSH &$t8
71 @
72 PUSH &i
73 !
74 PUSH &i
75 @
76 PUSH &$t11
77 !
78 PUSH &high
79 @
80 PUSH &$t12
81 !
```

Algoritmo 5.14 – Assembly

```
26 1110100000000000
27 0000000001001100
28 1111000000000000
29 0000000001001010
30 1110100000000000
31 0000000000001010
32 1111000000000000
33 0000000001001110
34 1110100000000000
35 0000000000001100
36 1111000000000000
37 0000000000001100
38 1110100000000000
39 0000000001001110
40 1000100000000000
41 1110100000000000
42 0000000000001110
43 1111000000000000
44 0000000000001110
45 1110100000000000
46 0000000000001010
47 1111000000000000
48 0000000000001010
49 1110100000000000
50 0000000001001010
51 1111000000000000
52 0000000001001000
53 1110100000000000
54 0000000000010000
55 1111000000000000
56 0000000001001110
57 1110100000000000
58 0000000000010010
59 1111000000000000
60 0000000000010010
61 1110100000000000
62 0000000000000001
63 1000100000000000
64 0000000000010100
65 1111000000000000
66 0000000000010100
67 1110100000000000
68 0000000000010000
69 1111000000000000
70 0000000000010000
71 1110100000000000
72 0000000001001000
73 1111000000000000
74 0000000001001000
75 1110100000000000
76 0000000000010110
77 1111000000000000
78 0000000001010000
79 1110100000000000
80 0000000000011000
81 1111000000000000
```

Algoritmo 5.15 – Executável

```
82  PUSH &$t11
83  @
84  PUSH &$t12
85  @
86  -
87  DUP
88  PUSH 0
89  LEQ
90  SWAP
91  PUSH 0
92  SWAP
93  LEQ
94  --
95  PUSH 1
96  -
97  PUSH &$t13
98  !
99  PUSH &$t13
100 @
101 IF L2
102 PUSH &i
103 @
104 PUSH &$t14
105 !
106 PUSH &$t14
107 @
108 PUSH &a
109 +
110 @
111 PUSH &$t15
112 !
113 PUSH &x
114 @
115 PUSH &$t0
116 !
117 PUSH &$t15
118 @
119 PUSH &$t0
120 @
121 -
122 DUP
123 PUSH 0
124 LEQ
125 SWAP
126 PUSH 0
127 SWAP
128 LEQ
129 --
130 PUSH 1
131 -
132 PUSH &$t1
133 !
134 PUSH &$t1
135 @
136 IF L3
137 PUSH &x
138 @
139 PUSH &$t2
140 !
141 PUSH &i
142 @
```

Algoritmo 5.16 – Assembly

```
82  0000000000010110
83  1110100000000000
84  0000000000011000
85  1110100000000000
86  1000010000000000
87  1000000001000000
88  0000000000000000
89  1000000110000000
90  1000111001000000
91  0000000000000000
92  1000111001000000
93  1000000110000000
94  1000110000000000
95  0000000000000001
96  1000010000000000
97  0000000000011010
98  1111000000000000
99  0000000000011010
100 1110100000000000
101 1001100001100010
102 0000000001001000
103 1110100000000000
104 0000000000011100
105 1111000000000000
106 0000000000011100
107 1110100000000000
108 0000000001001110
109 1000100000000000
110 1110100000000000
111 0000000000011110
112 1111000000000000
113 0000000001001010
114 1110100000000000
115 0000000000000000
116 1111000000000000
117 0000000000011110
118 1110100000000000
119 0000000000000000
120 1110100000000000
121 1000010000000000
122 1000000001000000
123 0000000000000000
124 1000000110000000
125 1000111001000000
126 0000000000000000
127 1000111001000000
128 1000000110000000
129 1000110000000000
130 0000000000000001
131 1000010000000000
132 0000000000000010
133 1111000000000000
134 0000000000000010
135 1110100000000000
136 1001100000101000
137 0000000001001010
138 1110100000000000
139 0000000000000100
140 1111000000000000
141 0000000001001000
142 1110100000000000
```

Algoritmo 5.17 – Executável

```
143 PUSH &$t3
144 !
145 PUSH &$t3
146 @
147 PUSH &a
148 +
149 @
150 PUSH &$t4
151 !
152 PUSH &$t4
153 @
154 PUSH &$t2
155 !
156 PUSH &$t2
157 @
158 PUSH &x
159 !
160 PUSH &k
161 @
162 PUSH &$t5
163 !
164 PUSH &i
165 @
166 PUSH &$t6
167 !
168 PUSH &$t6
169 @
170 PUSH &$t5
171 !
172 PUSH &$t5
173 @
174 PUSH &k
175 !
176 PUSH &i
177 @
178 PUSH &$t7
179 !
180 PUSH &i
181 @
182 PUSH &$t8
183 !
184 PUSH &$t8
185 @
186 PUSH 1
187 +
188 PUSH &$t9
189 !
190 PUSH &$t9
191 @
192 PUSH &$t7
193 !
194 PUSH &$t7
195 @
196 PUSH &i
197 !
198 CALL L1
199 PUSH &k
200 @
201 PUSH &$t10
202 !
203 PUSH &$t10
```

Algoritmo 5.18 – Assembly

```
143 0000000000000110
144 1111000000000000
145 0000000000000110
146 1110100000000000
147 0000000001001110
148 1000100000000000
149 1110100000000000
150 0000000000001000
151 1111000000000000
152 0000000000001000
153 1110100000000000
154 0000000000000100
155 1111000000000000
156 0000000000000100
157 1110100000000000
158 0000000001001010
159 1111000000000000
160 0000000001001100
161 1110100000000000
162 0000000000001010
163 1111000000000000
164 0000000001001000
165 1110100000000000
166 0000000000001100
167 1111000000000000
168 0000000000001100
169 1110100000000000
170 0000000000001010
171 1111000000000000
172 0000000000001010
173 1110100000000000
174 0000000001001100
175 1111000000000000
176 0000000001001000
177 1110100000000000
178 0000000000001110
179 1111000000000000
180 0000000001001000
181 1110100000000000
182 0000000000010000
183 1111000000000000
184 0000000000010000
185 1110100000000000
186 0000000000000001
187 1000100000000000
188 0000000000010010
189 1111000000000000
190 0000000000010010
191 1110100000000000
192 0000000000001110
193 1111000000000000
194 0000000000001110
195 1110100000000000
196 0000000001001000
197 1111000000000000
198 1010011110000100
199 0000000001001100
200 1110100000000000
201 0000000000010100
202 1111000000000000
203 0000000000010100
```

Algoritmo 5.19 – Executável

```
204 @
205 EXIT
206 EXIT
207 PUSH &i
208 @
209 PUSH &$t11
210 !
211 PUSH &low
212 @
213 PUSH &$t12
214 !
215 PUSH &$t12
216 @
217 PUSH &$t11
218 !
219 PUSH &$t11
220 @
221 PUSH &i
222 !
223 PUSH &i
224 @
225 PUSH &$t13
226 !
227 PUSH &high
228 @
229 PUSH &$t14
230 !
231 PUSH &$t14
232 @
233 PUSH 1
234 -
235 PUSH &$t15
236 !
237 PUSH &$t13
238 @
239 PUSH &$t15
240 @
241 -
242 DUP
243 PUSH 0
244 LEQ
245 SWAP
246 PUSH 0
247 SWAP
248 LEQ
249 --
250 PUSH 1
251 -
252 PUSH &$t0
253 !
254 PUSH &$t0
255 @
256 IF L5
257 PUSH &k
258 @
259 PUSH &$t1
260 !
261 PUSH &$t1
262 @
263 PUSH &a
264 +
```

Algoritmo 5.20 – Assembly

```
204 1110100000000000
205 1000000000100000
206 1000000000100000
207 0000000001001000
208 1110100000000000
209 0000000000010110
210 1111000000000000
211 0000000001001110
212 1110100000000000
213 0000000000011000
214 1111000000000000
215 0000000000011000
216 1110100000000000
217 0000000000010110
218 1111000000000000
219 0000000000010110
220 1110100000000000
221 0000000001001000
222 1111000000000000
223 0000000001001000
224 1110100000000000
225 0000000000011010
226 1111000000000000
227 0000000001010000
228 1110100000000000
229 0000000000011100
230 1111000000000000
231 0000000000011100
232 1110100000000000
233 0000000000000001
234 1000010000000000
235 0000000000011110
236 1111000000000000
237 0000000000011010
238 1110100000000000
239 0000000000011110
240 1110100000000000
241 1000010000000000
242 1000000001000000
243 0000000000000000
244 1000000110000000
245 1000111001000000
246 0000000000000000
247 1000111001000000
248 1000000110000000
249 1000110000000000
250 0000000000000001
251 1000010000000000
252 0000000000000000
253 1111000000000000
254 0000000000000000
255 1110100000000000
256 1001100010001000
257 0000000001001100
258 1110100000000000
259 0000000000000010
260 1111000000000000
261 0000000000000010
262 1110100000000000
263 0000000001001110
264 1000100000000000
```

Algoritmo 5.21 – Executável

```
265 @
266 PUSH &$t2
267 !
268 PUSH &$t2
269 @
270 PUSH &i
271 @
272 PUSH &$t3
273 !
274 PUSH &$t3
275 @
276 PUSH &high
277 @
278 PUSH &$t4
279 !
280 PUSH &$t4
281 @
282 CALL minloc
283 PUSH &$t5
284 !
285 PUSH &$t5
286 @
287 PUSH &$t2
288 !
289 PUSH &$t2
290 @
291 PUSH &low
292 !
293 PUSH &t
294 @
295 PUSH &$t6
296 !
297 PUSH &k
298 @
299 PUSH &$t7
300 !
301 PUSH &$t7
302 @
303 PUSH &a
304 +
305 @
306 PUSH &$t8
307 !
308 PUSH &$t8
309 @
310 PUSH &$t6
311 !
312 PUSH &$t6
313 @
314 PUSH &t
315 !
316 PUSH &k
317 @
318 PUSH &$t9
319 !
320 PUSH &$t9
321 @
322 PUSH &a
323 +
324 @
325 PUSH &$t10
```

Algoritmo 5.22 – Assembly

```
265 1110100000000000
266 0000000000000100
267 1111000000000000
268 0000000000000100
269 1110100000000000
270 0000000001001000
271 1110100000000000
272 0000000000000110
273 1111000000000000
274 0000000000000110
275 1110100000000000
276 0000000001010000
277 1110100000000000
278 0000000000001000
279 1111000000000000
280 0000000000001000
281 1110100000000000
282 1010011011100111
283 0000000000001010
284 1111000000000000
285 0000000000001010
286 1110100000000000
287 0000000000000100
288 1111000000000000
289 0000000000000100
290 1110100000000000
291 0000000001001110
292 1111000000000000
293 0000000001010110
294 1110100000000000
295 0000000000001100
296 1111000000000000
297 0000000001001100
298 1110100000000000
299 0000000000001110
300 1111000000000000
301 0000000000001110
302 1110100000000000
303 0000000001001110
304 1000100000000000
305 1110100000000000
306 0000000000010000
307 1111000000000000
308 0000000000010000
309 1110100000000000
310 0000000000001100
311 1111000000000000
312 0000000000001100
313 1110100000000000
314 0000000001010110
315 1111000000000000
316 0000000001001100
317 1110100000000000
318 0000000000010010
319 1111000000000000
320 0000000000010010
321 1110100000000000
322 0000000001001110
323 1000100000000000
324 1110100000000000
325 0000000000010100
```

Algoritmo 5.23 – Executável

```
!
PUSH &i
@
PUSH &$t11
!
PUSH &$t11
@
PUSH &a
+
@
PUSH &$t12
!
PUSH &$t12
@
PUSH &$t10
!
PUSH &$t10
@
PUSH &a
!
PUSH &i
@
PUSH &$t13
!
PUSH &$t13
@
PUSH &a
+
@
PUSH &$t14
!
PUSH &t
@
PUSH &$t15
!
PUSH &$t15
@
PUSH &$t14
!
PUSH &$t14
@
PUSH &a
!
PUSH &i
@
PUSH &$t0
!
PUSH &i
@
PUSH &$t1
!
PUSH &$t1
@
PUSH 1
+
PUSH &$t2
!
PUSH &$t2
@
PUSH &$t0
!
```

Algoritmo 5.24 – Assembly

```
1111000000000000
0000000001001000
1110100000000000
0000000000010110
1111000000000000
0000000000010110
1110100000000000
0000000001001110
1000100000000000
1110100000000000
0000000000011000
1111000000000000
0000000000011000
1110100000000000
0000000000010100
1111000000000000
0000000000010100
1110100000000000
0000000001001110
1111000000000000
0000000001001000
1110100000000000
0000000000011010
1111000000000000
0000000000011010
1110100000000000
0000000001001110
1000100000000000
1110100000000000
0000000000011100
1111000000000000
0000000001010110
1110100000000000
0000000000011110
1111000000000000
0000000000011110
1110100000000000
0000000000011100
1111000000000000
0000000000011100
1110100000000000
0000000001001110
1111000000000000
0000000001001000
1110100000000000
0000000000000000
1111000000000000
0000000001001000
1110100000000000
0000000000000010
1111000000000000
0000000000000010
1110100000000000
0000000000000001
1000100000000000
0000000000000100
1111000000000000
0000000000000100
1110100000000000
0000000000000000
1111000000000000
```

Algoritmo 5.25 – Executável

```
387 PUSH &$t0
388 @
389 PUSH &i
390 !
391 CALL L4
392 EXIT
393 PUSH &i
394 @
395 PUSH &$t3
396 !
397 PUSH &0
398 @
399 PUSH &$t3
400 !
401 PUSH &$t3
402 @
403 PUSH &i
404 !
405 PUSH &i
406 @
407 PUSH &$t4
408 !
409 PUSH &$t4
410 @
411 PUSH 10
412 -
413 DUP
414 PUSH 0
415 LEQ
416 SWAP
417 PUSH 0
418 SWAP
419 LEQ
420 --
421 PUSH 1
422 -
423 PUSH &$t5
424 !
425 PUSH &$t5
426 @
427 IF L7
428 PUSH &i
429 @
430 PUSH &$t6
431 !
432 PUSH &$t6
433 @
434 PUSH &vet
435 +
436 @
437 PUSH &$t7
438 !
439 INPUT READ
440 INPUT LOAD
441 PUSH &$t8
442 !
443 PUSH &$t8
444 @
445 PUSH &$t7
446 !
447 PUSH &$t7
```

Algoritmo 5.26 – Assembly

```
387 0000000000000000
388 1110100000000000
389 0000000001001000
390 1111000000000000
391 1010011101011000
392 1000000000100000
393 0000000001001000
394 1110100000000000
395 0000000000000110
396 1111000000000000
397 0000000000000000
398 1110100000000000
399 0000000000000110
400 1111000000000000
401 0000000000000110
402 1110100000000000
403 0000000001001000
404 1111000000000000
405 0000000001001000
406 1110100000000000
407 0000000000001000
408 1111000000000000
409 0000000000001000
410 1110100000000000
411 0000000000001010
412 1000010000000000
413 1000000001000000
414 0000000000000000
415 1000000110000000
416 1000111001000000
417 0000000000000000
418 1000111001000000
419 1000000110000000
420 1000110000000000
421 0000000000000001
422 1000010000000000
423 0000000000001010
424 1111000000000000
425 0000000000001010
426 1110100000000000
427 1001100000101111
428 0000000001001000
429 1110100000000000
430 0000000000001100
431 1111000000000000
432 0000000000001100
433 1110100000000000
434 0000000000100000
435 1000100000000000
436 1110100000000000
437 0000000000001110
438 1111000000000000
439 1100000000000001
440 1100000000000000
441 0000000000010000
442 1111000000000000
443 0000000000010000
444 1110100000000000
445 0000000000001110
446 1111000000000000
447 0000000000001110
```

Algoritmo 5.27 – Executável

```
PUSH &vet
!
PUSH &i
@
PUSH &$t9
!
PUSH &i
@
PUSH &$t10
!
PUSH &$t10
@
PUSH 1
+
PUSH &$t11
!
PUSH &$t11
@
PUSH &$t9
!
PUSH &$t9
@
PUSH &i
!
CALL L6
PUSH &$t1
@
PUSH &vet
+
@
PUSH &$t12
!
PUSH &$t12
@
PUSH 0
PUSH 10
CALL sort
PUSH &i
@
PUSH &$t14
!
PUSH &0
@
PUSH &$t14
!
PUSH &$t14
@
PUSH &i
!
PUSH &i
@
PUSH &$t15
!
PUSH &$t15
@
PUSH 10
-
DUP
PUSH 0
LEQ
SWAP
```

Algoritmo 5.28 – Assembly

```
1110100000000000
0000000000100000
1111000000000000
0000000001001000
1110100000000000
0000000000010010
1111000000000000
0000000001001000
1110100000000000
0000000000010100
1111000000000000
0000000000010100
1110100000000000
0000000000000001
1000100000000000
0000000000010110
1111000000000000
0000000000010110
1110100000000000
0000000000010010
1111000000000000
0000000000010010
1110100000000000
0000000001001000
1111000000000000
1010011110111100
0000000000000010
1110100000000000
0000000000100000
1000100000000000
1110100000000000
0000000000011000
1111000000000000
0000000000011000
1110100000000000
0000000000000000
0000000000001010
1010011011101010
0000000001001000
1110100000000000
0000000000011100
1111000000000000
0000000000000000
1110100000000000
0000000000011100
1111000000000000
0000000000011100
1110100000000000
0000000001001000
1111000000000000
0000000001001000
1110100000000000
0000000000011110
1111000000000000
0000000000011110
1110100000000000
0000000000001010
1000010000000000
1000000001000000
0000000000000000
1000000110000000
```

Algoritmo 5.29 – Executável

```
@
PUSH 0
SWAP
LEQ
--
PUSH 1
-
PUSH &$t0
!
PUSH &$t0
@
IF L9
PUSH &i
@
PUSH &$t1
!
PUSH &$t1
@
PUSH &vet
+
@
PUSH &$t2
!
PUSH &$t2
@
OUTPUT SEND
OUTPUT PRINT
PUSH &i
@
PUSH &$t4
!
PUSH &i
@
PUSH &$t5
!
PUSH &$t5
@
PUSH 1
+
PUSH &$t6
!
PUSH &$t6
@
PUSH &$t4
!
PUSH &$t4
@
PUSH &i
!
CALL L8
HALT
```

Algoritmo 5.30 – Assembly

```
1000111001000000
0000000000000000
1000111001000000
1000000110000000
1000110000000000
0000000000000001
1000010000000000
0000000000000000
1111000000000000
0000000000000000
1110100000000000
1001100000100111
0000000001001000
1110100000000000
0000000000000010
1111000000000000
0000000000000010
1110100000000000
0000000000100000
1000100000000000
1110100000000000
0000000000000100
1111000000000000
0000000000000100
1110100000000000
0000000000000000
1101000000000000
0000000001001000
1110100000000000
0000000000001000
1111000000000000
0000000001001000
1110100000000000
0000000000001010
1111000000000000
0000000000001010
1110100000000000
0000000000000001
1000100000000000
0000000000001100
1111000000000000
0000000000001100
1110100000000000
0000000000001000
1111000000000000
0000000000001000
1110100000000000
0000000001001000
1111000000000000
1010011111000100
1110000000000000
```

Algoritmo 5.31 – Executável

5.3 Exemplo 3 - Gcd

5.3.1 Código Fonte

```
int gcd(int u, int v)
{
        if (v==0) return u;
        else return gcd(v, u-u/v*v);
}
void main(void)
{
        int x; int y;
        x = input();
        y = input();
        output(gcd(x,y));
}
```

5.3.2 Código Intermediário

```
(FUN,INT,gcd,-)
(ARG,INT,v,gcd)
(LOAD,$t1,v,-)
(LOAD,$t2,v,-)
(EQ,$t3,$t2,0)
(IFF,$t3,L1,-)
(LOAD,$t4,u,-)
(RET,$t4,-,-)
(GOTO,L2,-,-)
(LABEL,L1,-,-)
(LOAD,$t5,v,-)
(PARAM,$t5,-,-)
(LOAD,$t6,u,-)
(LOAD,$t7,u,-)
(LOAD,$t8,v,-)
(DIV,$t9,$t7,$t8)
(LOAD,$t10,v,-)
(MULT,$t11,$t9,$t10)
(SUB,$t12,$t6,$t11)
(PARAM,$t12,-,-)
(CALL,$t13,gcd,2)
(RET,$t13,-,-)
(LABEL,L2,-,-)
(END,gcd,-,-)
(FUN,VOID,main,-)
(ALLOC,x,main,-)
(ALLOC,y,main,-)
(LOAD,$t14,x,-)
(CALL,$t15,input,0)
(ASSIGN,$t14,$t15,-)
(STORE,$t14,x,-)
(LOAD,$t0,y,-)
(CALL,$t1,input,0)
(ASSIGN,$t0,$t1,-)
(STORE,$t0,y,-)
(LOAD,$t2,x,-)
(PARAM,$t2,-,-)
(LOAD,$t3,y,-)
(PARAM,$t3,-,-)
(CALL,$t4,gcd,2)
(PARAM,$t4,-,-)
(CALL,$t5,output,1)
(END,main,-,-)
(HALT,-,-,-)
```

5.3.3 Códigos *Assembly* e Executável

```
PUSH &v
!
PUSH &v
@
PUSH &$t1
!
PUSH &v
@
PUSH &$t2
!
PUSH &$t2
@
PUSH 0
XOR
PUSH &$t3
!
PUSH &$t3
@
IF L1
PUSH &u
@
PUSH &$t4
!
PUSH &$t4
@
EXIT
CALL L2
PUSH &v
@
PUSH &$t5
!
PUSH &$t5
@
PUSH &u
@
PUSH &$t6
!
PUSH &u
@
PUSH &$t7
!
PUSH &v
@
PUSH &$t8
!
PUSH &$t7
@
PUSH &$t8
@
/
PUSH &$t9
!
PUSH &v
@
PUSH &$t10
```

Algoritmo 5.32 – Assembly

```
0000000000000000
1111000000000000
0000000000000000
1110100000000000
0000000000000010
1111000000000000
0000000000000000
1110100000000000
0000000000000100
1111000000000000
0000000000000100
1110100000000000
0000000000000000
1000101000000000
0000000000000110
1111000000000000
0000000000000110
1110100000000000
1001100000001001
0000000000000000
1110100000000000
0000000000001000
1111000000000000
0000000000001000
1110100000000000
1000000000100000
1010000000110100
0000000000000000
1110100000000000
0000000000001010
1111000000000000
0000000000001010
1110100000000000
0000000000000000
1110100000000000
0000000000001100
1111000000000000
0000000000000000
1110100000000000
0000000000001110
1111000000000000
0000000000000000
1110100000000000
0000000000010000
1111000000000000
0000000000001110
1110100000000000
0000000000010000
1110100000000000
1000000100000100
0000000000010010
1111000000000000
0000000000000000
1110100000000000
0000000000010100
```

Algoritmo 5.33 – Executável

```
!
PUSH &$t9
@
PUSH &$t10
@
*
PUSH &$t11
!
PUSH &$t6
@
PUSH &$t11
@
-
PUSH &$t12
!
PUSH &$t12
@
CALL gcd
PUSH &$t13
!
PUSH &$t13
@
EXIT
EXIT
PUSH &x
@
PUSH &$t14
!
INPUT READ
INPUT LOAD
PUSH &$t15
!
PUSH &$t15
@
PUSH &$t14
!
PUSH &$t14
@
PUSH &x
!
PUSH &y
@
PUSH &$t0
!
INPUT READ
INPUT LOAD
PUSH &$t1
!
PUSH &$t1
@
PUSH &$t0
!
PUSH &$t0
@
PUSH &y
!
PUSH &x
@
PUSH &$t2
!
PUSH &$t2
```

Algoritmo 5.34 – Assembly

```
0000000000010010
1110100000000000
0000000000010100
1110100000000000
1000000100000000
0000000000010110
1111000000000000
0000000000001100
1110100000000000
0000000000010110
1110100000000000
1000010000000000
0000000000011000
1111000000000000
0000000000011000
1110100000000000
1010011110111000
0000000000011010
1111000000000000
0000000000011010
1110100000000000
1000000000100000
1000000000100000
0000000000100000
1110100000000000
0000000000011100
1111000000000000
1100000000000001
1100000000000000
0000000000011110
1111000000000000
0000000000011110
1110100000000000
0000000000011100
1111000000000000
0000000000011100
1110100000000000
0000000000100000
1111000000000000
0000000000100010
1110100000000000
0000000000000000
1111000000000000
1100000000000001
1100000000000000
0000000000000010
1111000000000000
0000000000000010
1110100000000000
0000000000000000
1111000000000000
0000000000000000
1110100000000000
0000000000100010
1111000000000000
0000000000100000
1110100000000000
0000000000000100
1111000000000000
0000000000000100
1111000000000000
```

Algoritmo 5.35 – Executável

```
@
PUSH &y
@
PUSH &$t3
!
PUSH &$t3
@
CALL gcd
PUSH &$t4
!
PUSH &$t4
@
OUTPUT SEND
OUTPUT PRINT
HALT
```

Algoritmo 5.36 – Assembly

```
1110100000000000
0000000000100010
1110100000000000
0000000000000110
1111000000000000
0000000000000110
1110100000000000
1010011110000101
0000000000001000
1111000000000000
0000000000001000
1110100000000000
0000000000000000
1101000000000000
1110000000000000
```

Algoritmo 5.37 – Executável

6 Conclusão

O presente projeto tem por proposta desenvolver um compilador da Linguagem *C-* para a Linguagem do processador *AsterixForth* desenvolvido pela autora na matéria de laboratório de Sistemas Computacionais: Arquitetura de Computadores. Para esse relatório final, esperava-se que se implementasse em Flex o módulo de análise léxica, em Bison o módulo de análise sintática e em C os módulos de análise semântica e de síntese.

Conseguimos desenvolver o compilador como um todo, aprofundando nossos conhecimentos acerca do assunto. A decisão por implementar desde a árvore sintática abstrata até o código binário apenas com o entendimento base sobre o assunto e sem se basear no *tiny* foi bastante desafiador, mas foi também bastante enriquecedor, a medida que cada passo do processo requeria um estudo aprofundado dos detalhes do conteúdo. A implementação da recursividade foi um passo extremamente natural proveniente da arquitetura *AsterixForth* pois, sendo uma arquitetura de pilha, trocas de contexto e recursividade são de baixíssimo custo. Neste sentido o exemplo *fact* vem mostrar essa recursividade utilizada.

Algumas dificuldades devem ser avultadas, principalmente referentes à parte de síntese. Por se tratar de uma arquitetura de pilha, o *AsterixForth* era mais adaptado a uma linguagem de pilha como *Forth*, que foi a linguagem que originou seu *Assembly* originalmente. Sendo assim, compilar *C-* para *AsterixForth* foi extremamente desafiador, pois se trata de uma linguagem muito mais adaptada para um processador *MIPS*, por exemplo. Desta forma, muito se gastou em desenhar e redesenhar caminhos para que as instruções de quádruplas muito mais próximas ao *MIPS* se encaminhassem para um *Assembly Forth*. Neste sentido, é possível melhorar o compilador bastante com a inserção de um módulo de otimização para minimizar essa certa incompatibilidade linguística de *C-* e o *AsterixForth*.

Foi um projeto longo - mais de um ano de execução - e extremamente complexo, mas muito gratificante e necessário para poder dar prosseguimento ao laboratório de Sistemas Operacionais, utilizando o presente projeto como compilador.

Referências

1 AHO, A. V. *Compilers: principles, techniques and tools*. [S.l.]: Pearson, 2007. Citado na página 1.

2 PEREIRA, J. W. *AsterixForth - Desenvolvimento e descrição de uma Arquitetura de Pilha*. 2019. Disponível em: <https://github.com/JuliaWPereira/asterixForth/blob/master/AsterixForth.pdf>. Citado 5 vezes nas páginas 1, 7, 8, 11 e 12.

3 LOUDEN, K. C.; SILVA, F. S. C. *Compiladores-Princípios e Práticas*. [S.l.]: Cengage Learning Editores, 2004. Citado 2 vezes nas páginas 18 e 19.

4 TEAM, W. *WiRed Panda - Learn about logic circuits and simulate them in an easy and friendly way*. Disponível em: <https://wiredpanda.org/>. Citado na página 49.

Apêndices

APÊNDICE A – Quádruplas para Declaração e Chamada de Funções

A.1 Definição de Funções

A.1.1 Aplicação

Definição de Função. Indica que as próximas quádruplas serão referentes ao corpo da função. A função acaba na declaração da quádrupla **END** (ver seção A.6).

A.1.2 Formato

(FUN, type, function_name, -)

A.1.3 Parâmetros

- **type:** refere-se ao tipo de retorno da função. Se a função tem retorno inteiro, deve ser colocado **INT** e se não tem retorno deve ser colocado **VOID**.
- **function_name:** É o nome dado a função.

A.1.4 Exemplo

Considere o seguinte código em C-:

```
/* Codigo em C- */
int print_Hello_World()
{
    /* corpo da funcao */
}
```

Obtemos as seguintes quádruplas:

(FUN, INT, print_Hello_World, -)
(END, print_Hello_World, -, -)

A.2 Argumento de Funções

A.2.1 Aplicação

Argumento de função. Deve aparecer sempre que uma função é declarada indicando os seus argumentos (ver seção A.1). Após serem declarados, os argumentos devem ser carregados (ver seção B.2).

A.2.2 Formato

(ARG, INT, var, scope)

A.2.3 Parâmetros

- **var:** É o nome da variável de argumento.
- **scope:** É o nome da função na qual a variável está sendo declarada como argumento.

A.2.4 Exemplo

Considere o seguinte código em C-:

```
/* Codigo em C- */
int sort(int a[], int low, int high)
{
    /* corpo do codigo */
}
```

Obtemos as seguintes quádruplas:

(FUN, INT, sort, -)
(ARG, INT, a, sort)
(ARG, INT, low, sort)
(ARG, INT, high, sort)
(LOAD, $t1, a, -)
(LOAD, $t2, low, -)
(LOAD, $t3, high, -)
(END, sort, -, -)

A.3 Chamada de Funções

A.3.1 Aplicação

Chamada de função. Indica qual função deve ser chamada, em qual variável ela deve ser alocada e quantos parâmetros esta possui. Os parâmetros são passados como **PARAM** (ver seção A.4).

A.3.2 Formato

(CALL, temp, function_name, n_params)

A.3.3 Parâmetros

- **temp:** é o registrador temporário que recebe o retorno da função.
- **function_name:** é o nome dado a função chamada.
- **n_params:** é o número de parâmetros que devem ser passados para a função.

A.3.4 Exemplo

Considere o seguinte código em C-:

```
1   /* Codigo em C- */
2   i = sort(a, b)
```

Obtemos as seguintes quádruplas:

(LOAD, $t1, i, -)
(LOAD, $t2, a, -)
(PARAM, $t2, -, -)
(LOAD, $t3, b, -)
(PARAM, $t3, -, -)
(CALL, $t4, sort, 3)
(ASSIGN, $t1, $t4, -)
(STORE, i, $t1, -, -)

A.4 Parâmetros de Função

A.4.1 Aplicação

Parâmetro de função. Cada vez que uma variável é declarada como parâmetro, ela é empilhada para ser usada quando a função por chamada. Além disso, conforme os parâmetros são adicionados, um contador é incrementado, para garantir que a chamada de função saiba quantos parâmetros ela possui (ver seção A.3).

A.4.2 Formato

(PARAM, temp, -, -)

A.4.3 Parâmetros

- **temp:** é o registrador temporário que deve ser empilhado como parâmetro de função.

A.4.4 Exemplo

Considere o seguinte código em C-:

```
/* Codigo em C- */
i = sort(a, b)
```

Obtemos as seguintes quádruplas:

(LOAD, $t1, i, -)
(LOAD, $t2, a, -)
(PARAM, $t2, -, -)
(LOAD, $t3, b, -)
(PARAM, $t3, -, -)
(CALL, $t4, sort, 3)
(ASSIGN, $t1, $t4, -)
(STORE, i, $t1, -, -)

A.5 Retorno de Funções

A.5.1 Aplicação

Retorno de função. Deve aparecer sempre que a função tiver retorno do tipo inteiro em sua assinatura. (ver seção A.1)

A.5.2 Formato

(RET, temp, -, -)

A.5.3 Parâmetros

- **temp:** É o nome do registrador temporário que possui o valor da variável a ser retornada.

A.5.4 Exemplo

Considere o seguinte código em C-:

```
/* Codigo em C- */
int sort(int a[], int low, int high)
{
    /* corpo do codigo */
    return low;
}
```

Obtemos as seguintes quádruplas:

(FUN, INT, sort, -)
(ARG, INT, a, sort)
(ARG, INT, low, sort)
(ARG, INT, high, sort)
(LOAD, $t1, a, -)
(LOAD, $t2, low, -)
(LOAD, $t3, high, -)
(RET, $t2, -, -)
(END, sort, -, -)

A.6 Finalização de Funções

A.6.1 Aplicação

Finalização de função. Indica que o corpo da função está encerrado. Deve ser precedido por um indicador de início de função (ver seção A.1).

A.6.2 Formato

(END, function_name, -, -)

A.6.3 Parâmetros

- **function_name:** É o nome dado a função.

A.6.4 Exemplo

Considere o seguinte código em C-:

```
/* Codigo em C- */
int print_Hello_World()
{
    /* corpo da funcao */
}
```

Obtemos as seguintes quádruplas:

(FUN, INT, print_Hello_World, -)
(END, print_Hello_World, -, -)

APÊNDICE B – Quádruplas para Leitura/Escrita e Declaração de Variáveis

B.1 Declaração de Variáveis

B.1.1 Aplicação

Alocação de variáveis. Destina a aquele rótulo de variável o espaço de memória necessário. Serve para declaração de variáveis simples ou vetores. Se for vetor, precisa passar o parâmetro de tamanho na declaração.

B.1.2 Formato

(ALLOC, var_name, scope, [size])

B.1.3 Parâmetros

- **var_name:** refere-se ao rótulo dado à variável.
- **scope:** é o escopo de declaração da variável. Se esta for declarada fora de qualquer função, seu escopo deve ser indicado como "GLOBAL".
- **size:** é o tamanho, caso seja vetor. Se for variável simples, não é passado. **(opcional)**

B.1.4 Exemplo

Considere o seguinte código em C-:

```
/* Codigo em C- */
int vet[10];
int sort()
{
    int w;
    /* corpo da funcao */
}
```

Obtemos as seguintes quádruplas:

(ALLOC,vet,GLOBAL,10)
(FUN, INT, sort, -)
(ALLOC,w,sort,-)
(END, sort, -, -)

B.2 Leitura de variáveis da Memória

B.2.1 Aplicação

Leitura de variável da memória. Deve ser utilizada sempre que a variável é utilizada no código e quando a variável é parâmetro da função. Para variáveis indexadas, pode-se carregar ou o vetor ou apenas um campo.

B.2.2 Formato

(LOAD, temp, var_name, [index])

B.2.3 Parâmetros

- **temp:** É o registrador temporário que recebe a leitura feita da memória.

- **var_name:** É o nome da variável a ser carregada.

- **index:** É a posição a ser carregada da variável indexada em **var_name**. **(opcional)**

B.2.4 Exemplo

Considere o seguinte código em C-:

```
/* Codigo em C- */
int sort(int a[], int low)
{
    int w;
    w = low;
    /* corpo da funcao */
}
```

Obtemos as seguintes quádruplas:

(FUN,INT,sort,-)
(ARG,INT,a,sort)
(ARG,INT,low,sort)
(LOAD,$t1,a,-)
(LOAD,$t2,low,-)
(ALLOC,w,sort,-)
(LOAD,$t3,w,-)
(LOAD,$t4,low,-)
(ASSIGN,$t5,$t6,-)
(STORE,$t5,i,-)
(END,sort,-,-)

B.3 Escrita de variáveis na Memória

B.3.1 Aplicação

Escrita de variável na memória. Deve ser utilizada sempre que a variável recebe um novo valor (ver seção B.4). Para que haja escrita na memória, é preciso ter havido o carregamento da variável em algum registrador temporário (ver seção B.2).

B.3.2 Formato

(STORE, temp, var_name, [index])

B.3.3 Parâmetros

- **temp:** É o registrador temporário que possui o conteúdo a ser escrito na memória.

- **var_name:** É o nome da variável que receberá o valor do registrador temporário.

- **index:** É a posição a ser escrita da variável indexada em **var_name**. **(opcional)**

B.3.4 Exemplo

Considere o seguinte código em C-:

```
/* Codigo em C- */
int sort(int a[], int low)
{
    int w;
    w = low;
    /* corpo da funcao */
}
```

Obtemos as seguintes quádruplas:

(FUN,INT,sort,-)
(ARG,INT,a,sort)
(ARG,INT,low,sort)
(LOAD,$t1,a,-)
(LOAD,$t2,low,-)
(ALLOC,w,sort,-)
(LOAD,$t3,w,-)
(LOAD,$t4,low,-)
(ASSIGN,$t5,$t6,-)
(STORE,$t5,i,-)
(END,sort,-,-)

B.4 Atribuição

B.4.1 Aplicação

Atribuição. Faz o valor do segundo parâmetro ser atribuído ao primeiro. Sempre é acompanhado de **STORE** (ver seção B.3).

B.4.2 Formato

(ASSIGN, temp_D, temp_S, -)

B.4.3 Parâmetros

- **temp_D:** é o registrador temporário de destino (D - Destiny) associado a uma variável.
- **temp_S:** é o primeiro registrador temporário de base (S - Source). É o registrador de origem do valor.

B.4.4 Exemplo

Considere o seguinte código em C-:

```
/* Codigo em C- */
i = m;
```

Obtemos as seguintes quádruplas:

(LOAD, $t1, i, -)
(LOAD, $t2, m, -)
(ASSIGN, $t1, $t2, -)
(STORE, $t1, i, -)

APÊNDICE C – Quádruplas de Controle

C.1 Finalização de Programas

C.1.1 Aplicação

Finalização de programa. Deve aparecer em todo código gerado para encerrar a execução de instruções.

C.1.2 Formato

(HALT, -, -, -)

C.1.3 Parâmetros

Não Possui parâmetros.

C.2 Rótulos de Controle - *Labels*

C.2.1 Aplicação

Labels de controle servem para que saltos possam ser feitos na execução do código. Assim, eles aparecem em para execução de laços *while* e para *if*. Para o salto para uma *label*, utiliza-se a função **GOTO** (ver seção C.3).

C.2.2 Formato

(LABEL, label_name, -, -)

C.2.3 Parâmetros

- **label_name:** é o nome do *label* de referência para aquele pedaço de código.

C.2.4 Exemplo

Considere o seguinte código em C-:

```
/* Codigo em C- */
if(i >= 0)
{
    output(i)
}
else
{
    output(i)
}
```

Obtemos as seguintes quádruplas:

(LOAD, $t1, i, -)
(GEQ, $t2, $t1, 0)
(IFF, $t2, L1, 0)
(LOAD, $t3, i, -)
(PARAM, $t3, -, -)
(CALL, $t4, output, 1)
(GOTO, L2, -, -)
(LABEL, L1, -, -)
(LOAD, $t5, i, -)
(PARAM, $t5, -, -)
(CALL, $t6, output, 1)
(LABEL, L2, -, -)

C.3 Desvio de Fluxo

C.3.1 Aplicação

Para desviar o fluxo de execução, utiliza-se a quádrupla **GOTO**. Para que ocorra os desvios, precisa haver os rótulos de partes do código (ver seção C.2).

C.3.2 Formato

(GOTO, label_name, -, -)

C.3.3 Parâmetros

- **label_name:** é o nome do *label* de referência para o pedaço de código que deve passar a ser executado.

C.3.4 Exemplo

Considere o seguinte código em C-:

```
/* Codigo em C- */
if(i >= 0)
{
    output(i)
}
else
{
    output(i)
}
```

Obtemos as seguintes quádruplas:

(LOAD, $t1, i, -)
(GEQ, $t2, $t1, 0)
(IFF, $t2, L1, 0)
(LOAD, $t3, i, -)
(PARAM, $t3, -, -)
(CALL, $t4, output, 1)
(GOTO, L2, -, -)
(LABEL, L1, -, -)
(LOAD, $t5, i, -)
(PARAM, $t5, -, -)
(CALL, $t6, output, 1)
(LABEL, L2, -, -)

C.4 Desvio Condicional - *If False*

C.4.1 Aplicação

Desvio Condicional. Sempre vem precedido de uma condição. Se a condição for falsa, desvia para o rótulo passado; caso a condição seja verdadeira, continua o fluxo corrente de instruções. (ver seções C.2 e C.3)

C.4.2 Formato

(IFF, temp_test, label_branch, -)

C.4.3 Parâmetros

- **temp_test:** É o nome do registrador temporário que possui o teste que foi feito para averiguar se a condição é verdadeira ou falsa.

- **label_branch:** É o nome do rótulo para o qual deve haver desvio caso a condição seja falsa.

C.4.4 Exemplo

Considere o seguinte código em C-:

```
/* Codigo em C- */
if(i >= 0)
{
    output(i)
}
else
{
    output(i)
}
```

Obtemos as seguintes quádruplas:

(LOAD, $t1, i, -)
(GEQ, $t2, $t1, 0)
(IFF, $t2, L1, 0)
(LOAD, $t3, i, -)
(PARAM, $t3, -, -)
(CALL, $t4, output, 1)
(GOTO, L2, -, -)
(LABEL, L1, -, -)
(LOAD, $t5, i, -)
(PARAM, $t5, -, -)
(CALL, $t6, output, 1)
(LABEL, L2, -, -)

APÊNDICE D – Quádruplas Lógico-Aritméticas

D.1 Adição

D.1.1 Aplicação

Soma. Soma os valores contidos nos dois registradores passados de parâmetros e atribui o valor calculado a um terceiro registrador.

D.1.2 Formato

(ADD, temp_D, temp_S1, temp_S2)

D.1.3 Parâmetros

- **temp_D:** é o registrador temporário de destino (D - Destiny). Recebe o valor da soma.
- **temp_S1:** é o primeiro registrador temporário de base (S - Source). É a primeira parcela da soma.
- **temp_S2:** é o segundo registrador temporário de base (S- Source). É a segunda parcela da soma.

D.1.4 Exemplo

Considere o seguinte código em C-:

```
/* Codigo em C- */
i = m + 1;
```

Obtemos as seguintes quádruplas:

(LOAD, $t1, i, -)
(LOAD, $t2, m, -)
(ADD, $t3, m, 1)
(ASSIGN, $t1, $t3, -)
(STORE, $t1, i, -)

D.2 Subtração

D.2.1 Aplicação

Subtração. Subtrai os valores contidos nos dois registradores passados de parâmetros e atribui o valor calculado a um terceiro registrador.

D.2.2 Formato

(SUB, temp_D, temp_S1, temp_S2)

D.2.3 Parâmetros

- **temp_D:** é o registrador temporário de destino (D - Destiny). Recebe o valor da subtração.
- **temp_S1:** é o primeiro registrador temporário de base (S - Source). É a primeira parcela da subtração.
- **temp_S2:** é o segundo registrador temporário de base (S- Source). É a segunda parcela da subtração.

D.2.4 Exemplo

Considere o seguinte código em C-:

```
/* Codigo em C- */
i = m - 1;
```

Obtemos as seguintes quádruplas:

(LOAD, $t1, i, -)
(LOAD, $t2, m, -)
(SUB, $t3, m, 1)
(ASSIGN, $t1, $t3, -)
(STORE, $t1, i, -)

D.3 Multiplicação

D.3.1 Aplicação

Multiplicação. Multiplica os valores contidos nos dois registradores passados de parâmetros e atribui o valor calculado a um terceiro registrador.

D.3.2 Formato

(MULT, temp_D, temp_S1, temp_S2)

D.3.3 Parâmetros

- **temp_D:** é o registrador temporário de destino (D - Destiny). Recebe o valor da multiplicação.
- **temp_S1:** é o primeiro registrador temporário de base (S - Source). É o primeiro fator da multiplicação.
- **temp_S2:** é o segundo registrador temporário de base (S- Source). É o segundo fator da multiplicação.

D.3.4 Exemplo

Considere o seguinte código em C-:

```
/* Codigo em C- */
i = m * 2;
```

Obtemos as seguintes quádruplas:

(LOAD, $t1, i, -)
(LOAD, $t2, m, -)
(MULT, $t3, m, 2)
(ASSIGN, $t1, $t3, -)
(STORE, $t1, i, -)

D.4 Divisão

D.4.1 Aplicação

Divisão. divide os valores contidos nos dois registradores passados de parâmetros e atribui o valor calculado a um terceiro registrador.

D.4.2 Formato

(DIV, temp_D, temp_S1, temp_S2)

D.4.3 Parâmetros

- **temp_D:** é o registrador temporário de destino (D - Destiny). Recebe o valor da divisão.
- **temp_S1:** é o primeiro registrador temporário de base (S - Source). É o dividendo.
- **temp_S2:** é o segundo registrador temporário de base (S- Source). É o divisor.

D.4.4 Exemplo

Considere o seguinte código em C-:

```
/* Codigo em C- */
i = m / 2;
```

Obtemos as seguintes quádruplas:

(LOAD, $t1, i, -)
(LOAD, $t2, m, -)
(DIV, $t3, m, 2)
(ASSIGN, $t1, $t3, -)
(STORE, $t1, i, -)

D.5 Maior que

D.5.1 Aplicação

Maior que. É a comparação se o primeiro valor passado é maior do que o segundo. Retorna 1 se verdadeiro e zero se falso.

D.5.2 Formato

(GT, temp_D, temp_S1, temp_S2)

D.5.3 Parâmetros

- **temp_D:** é o registrador temporário de destino (D - Destiny). Recebe o valor booleano da comparação.
- **temp_S1:** é o primeiro registrador temporário de base (S - Source). É o primeiro fator da comparação.
- **temp_S2:** é o segundo registrador temporário de base (S- Source). É o segundo fator da comparação.

D.5.4 Exemplo

Considere o seguinte código em C-:

```
/* Codigo em C- */
i = m > 0;
```

Obtemos as seguintes quádruplas:

(LOAD, $t1, i, -)
(LOAD, $t2, m, -)
(GT, $t3, m, 0)
(ASSIGN, $t1, $t3, -)
(STORE, $t1, i, -)

D.6 Menor que

D.6.1 Aplicação

Menor que. É a comparação se o primeiro valor passado é menor do que o segundo. Retorna 1 se verdadeiro e zero se falso.

D.6.2 Formato

(LT, temp_D, temp_S1, temp_S2)

D.6.3 Parâmetros

- **temp_D:** é o registrador temporário de destino (D - Destiny). Recebe o valor booleano da comparação.
- **temp_S1:** é o primeiro registrador temporário de base (S - Source). É o primeiro fator da comparação.
- **temp_S2:** é o segundo registrador temporário de base (S- Source). É o segundo fator da comparação.

D.6.4 Exemplo

Considere o seguinte código em C-:

```
/* Codigo em C- */
i = m < 0;
```

Obtemos as seguintes quádruplas:

(LOAD, $t1, i, -)
(LOAD, $t2, m, -)
(LT, $t3, m, 0)
(ASSIGN, $t1, $t3, -)
(STORE, $t1, i, -)

D.7 Maior ou igual a

D.7.1 Aplicação

Maior ou igual a. É a comparação se o primeiro valor passado é maior ou igual ao segundo. Retorna 1 se verdadeiro e zero se falso.

D.7.2 Formato

(GEQ, temp_D, temp_S1, temp_S2)

D.7.3 Parâmetros

- **temp_D:** é o registrador temporário de destino (D - Destiny). Recebe o valor booleano da comparação.
- **temp_S1:** é o primeiro registrador temporário de base (S - Source). É o primeiro fator da comparação.
- **temp_S2:** é o segundo registrador temporário de base (S- Source). É o segundo fator da comparação.

D.7.4 Exemplo

Considere o seguinte código em C-:

```
/* Codigo em C- */
i = m >= 0;
```

Obtemos as seguintes quádruplas:

(LOAD, $t1, i, -)
(LOAD, $t2, m, -)
(GEQ, $t3, m, 0)
(ASSIGN, $t1, $t3, -)
(STORE, $t1, i, -)

D.8 Menor ou igual a

D.8.1 Aplicação

Menor ou igual a. É a comparação se o primeiro valor passado é menor ou igual ao segundo. Retorna 1 se verdadeiro e zero se falso.

D.8.2 Formato

(LEQ, temp_D, temp_S1, temp_S2)

D.8.3 Parâmetros

- **temp_D:** é o registrador temporário de destino (D - Destiny). Recebe o valor booleano da comparação.
- **temp_S1:** é o primeiro registrador temporário de base (S - Source). É o primeiro fator da comparação.
- **temp_S2:** é o segundo registrador temporário de base (S- Source). É o segundo fator da comparação.

D.8.4 Exemplo

Considere o seguinte código em C-:

```
/* Codigo em C- */
i = m <= 0;
```

Obtemos as seguintes quádruplas:

(LOAD, $t1, i, -)
(LOAD, $t2, m, -)
(LEQ, $t3, m, 0)
(ASSIGN, $t1, $t3, -)
(STORE, $t1, i, -)

D.9 É igual a

D.9.1 Aplicação

É igual a. É a comparação se o primeiro valor passado é igual ao segundo. Retorna 1 se verdadeiro e zero se falso.

D.9.2 Formato

(EQ, temp_D, temp_S1, temp_S2)

D.9.3 Parâmetros

- **temp_D:** é o registrador temporário de destino (D - Destiny). Recebe o valor booleano da comparação.
- **temp_S1:** é o primeiro registrador temporário de base (S - Source). É o primeiro fator da comparação.
- **temp_S2:** é o segundo registrador temporário de base (S- Source). É o segundo fator da comparação.

D.9.4 Exemplo

Considere o seguinte código em C-:

```
/* Codigo em C- */
i = m == 0;
```

Obtemos as seguintes quádruplas:

(LOAD, $t1, i, -)
(LOAD, $t2, m, -)
(EQ, $t3, m, 0)
(ASSIGN, $t1, $t3, -)
(STORE, $t1, i, -)

D.10 É diferente de

D.10.1 Aplicação

É diferente de. É a comparação se o primeiro valor passado é diferente do segundo. Retorna 1 se verdadeiro e zero se falso.

D.10.2 Formato

(NEQ, temp_D, temp_S1, temp_S2)

D.10.3 Parâmetros

- **temp_D:** é o registrador temporário de destino (D - Destiny). Recebe o valor booleano da comparação.
- **temp_S1:** é o primeiro registrador temporário de base (S - Source). É o primeiro fator da comparação.
- **temp_S2:** é o segundo registrador temporário de base (S- Source). É o segundo fator da comparação.

D.10.4 Exemplo

Considere o seguinte código em C-:

```
/* Codigo em C- */
i = m != 0;
```

Obtemos as seguintes quádruplas:

(LOAD, $t1, i, -)
(LOAD, $t2, m, -)
(NEQ, $t3, m, 0)
(ASSIGN, $t1, $t3, -)
(STORE, $t1, i, -)