

Ícaro Travain Darwich da Rocha
156307

Implementação do Processador C-

São José dos Campos - Brasil

Julho de 2025

Ícaro Travain Darwich da Rocha
156307

Implementação do Processador C-

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

Docente: Prof. Dr. Luiz Eduardo Galvão Martins
Universidade Federal de São Paulo - UNIFESP
Instituto de Ciência e Tecnologia - Campus São José dos Campos

São José dos Campos - Brasil
Julho de 2025

Lista de ilustrações

Figura 1 – Datapath das instruções de processamento de dados	9
Figura 2 – Datapath das instruções de transferência de dados	10
Figura 3 – Datapath das instruções de desvio	11
Figura 4 – Datapath das instrução OUT	12
Figura 5 – Datapath das instrução IN	13
Figura 6 – Banco de registradores	14
Figura 7 – ULA	14
Figura 8 – Memória de dados (RAM)	16
Figura 9 – Memória de instruções (ROM)	17
Figura 10 – Unidade de instrução	17
Figura 11 – Diagrama de blocos - Fase de Análise	25
Figura 12 – Diagrama de atividades - Analise semântica	25
Figura 13 – Diagrama de pacotes do compilador	26
Figura 14 – Diagrama de Blocos - Fase de síntese	37
Figura 15 – Diagrama de Atividades - Percorrer árvore	38
Figura 16 – Diagrama de atividades - Gerador de código Intermediário	38

Lista de tabelas

Tabela 1 – Formato do sinal de controle	18
Tabela 2 – Operações do sinal SelClock	19
Tabela 3 – Instruções de Processamento de Dados	20
Tabela 4 – Formato da instrução LDR	21
Tabela 5 – Formato da instrução STR	21
Tabela 6 – Instruções de Transferência de Dados	21
Tabela 7 – Formato das instruções B	21
Tabela 8 – Formato das instruções BI	21
Tabela 9 – Instruções de desvio	22
Tabela 10 – Formato das outras instruções	22
Tabela 11 – Outras Instruções	22
Tabela 12 – Tokens e Expressões Regulares da Linguagem C-	27
Tabela 13 – Instruções do Código Intermediário (TAC) e suas Funções	40
Tabela 14 – Correspondência entre as etapas de compilação: Programa básico.	55
Tabela 15 – Tabela de Correspondência para o Exemplo de Ordenação	61
Tabela 16 – Correspondência entre Código C- e Código Intermediário para o algoritmo de Ordenação.	81

Sumário

1	INTRODUÇÃO	7
2	O PROCESSADOR	8
2.1	Caminho de Dados	8
2.1.1	DataPath - Instruções de processamento de dados	8
2.1.2	DataPath - Instruções de transferência de dados	9
2.1.3	DataPath - Instruções de desvio	10
2.1.4	DataPath - Outras Instruções	11
2.2	Componentes	13
2.2.1	Banco de registradores	13
2.2.2	Unidade Lógica e Aritmética (ULA)	14
2.2.3	Program Counter (PC)	14
2.2.4	Módulo de entrada	15
2.2.5	Módulo de saída	15
2.2.6	Multiplexadores	15
2.2.7	Memória de dados (RAM)	15
2.2.8	Memória de Instruções (ROM)	16
2.2.9	Unidade de instrução	17
2.2.10	Unidade de controle	18
2.2.11	Módulo de divisão do clock	18
2.3	Conjunto de Instruções	19
2.3.1	Instruções de Processamento de dados	20
2.3.2	Instruções de Transferência de Dados	20
2.3.3	Instruções de Desvio	21
2.3.4	Outras Instruções	22
2.4	Organização da Memória	22
3	COMPILADOR: FASE DE ANÁLISE	24
3.1	Modelagem	24
3.2	Análise Léxica	26
3.2.1	Implementação do Analisador Léxico	26
3.2.2	Implementação em Código	27
3.3	Análise Sintática	28
3.3.1	Implementação do Analisador Sintático	29
3.3.2	Estrutura do Arquivo de Gramática	29
3.3.2.1	Seção de Declarações e Configurações	29

3.3.2.2	Seção de Regras da Gramática	31
3.3.2.2.1	Regra Inicial.	31
3.3.2.2.2	Regras de Declaração.	31
3.3.2.2.3	Regras de Comando.	32
3.3.2.2.4	Regras de Expressão.	32
3.3.2.3	Seção de Código Adicional	33
3.4	Análise Semântica	33
3.4.1	Implementação do Analisador Semântico	34
3.4.2	Implementação em Código	34
4	COMPILADOR: FASE DE SÍNTESE	37
4.1	Modelagem	37
4.2	Geração do Código Intermediário	38
4.2.1	Implementação da Geração de Código Intermediário	39
4.2.2	Implementação em Código	40
4.3	Geração do Código Assembly	43
4.3.1	Implementação do Gerador de Código Assembly	43
4.3.2	Implementação em Código	43
4.4	Geração do Código Executável (Montagem)	46
4.4.1	Implementação do Montador	46
4.4.2	Implementação em Código	46
4.5	Gerenciamento de Memória e Chamada de Funções	48
4.5.1	O Registro de Ativação (Stack Frame)	48
4.5.1.0.1	Código do Chamador.	49
4.5.1.0.2	Prólogo da Função.	49
4.5.1.0.3	Epílogo da Função.	49
4.5.2	Passagem de Parâmetros e Vetores	50
4.5.2.0.1	Passagem de Vetores.	50
4.5.3	Suporte à Recursividade	51
5	EXEMPLOS DE USO	52
5.1	Exemplo 1: Programa básico	52
5.1.1	Código-Fonte	52
5.1.2	Código Intermediário Gerado	52
5.1.3	Código Assembly Gerado	53
5.1.4	Correspondência entre as Etapas	54
5.2	Exemplo 1: GCD	54
5.2.1	Código-Fonte	54
5.2.2	Código Intermediário Gerado	56
5.2.3	Código Assembly Gerado	57

5.2.4	Código Executável (Binário)	59
5.2.5	Correspondência entre códigos	61
5.3	Exemplo 3: Sort	66
5.3.1	Código-Fonte	66
5.3.2	Código Intermediário Gerado	68
5.3.3	Código Assembly Gerado	70
5.3.4	Código Executável (Binário)	76
5.3.5	Correspondência entre as Etapas	80
6	CONCLUSÃO	85
	REFERÊNCIAS	86

1 Introdução

No início da computação, o desenvolvimento de sistemas complexos era dificultado pela escrita direta em **código de máquina**, o que resultava em problemas de difícil programação, falta de portabilidade, baixa produtividade na codificação, manutenção complexa e uma curva de aprendizado íngreme. ([LOUDEN, 1997](#)).

Para solucionar esses desafios, foram desenvolvidos os compiladores, programas de computador cuja funcionalidade é traduzir código de uma linguagem para outra. Geralmente, eles convertem linguagens de alto nível, como C, C++ ou C-, para linguagens de baixo nível, como **código de máquina** ou **binário**. ([LOUDEN, 1997](#))

Dada a relevância dos compiladores, o objetivo central deste projeto, desenvolvido na disciplina de Laboratório de Compiladores, é a criação e implementação de um compilador para a linguagem C-. O desenvolvimento abrange todas as fases de um compilador: a fase de análise, que inclui os analisadores léxico, sintático e semântico, responsável por validar a estrutura e a coerência do código-fonte e construir uma representação intermediária (a Árvore de Sintaxe Abstrata – AST); e a fase de síntese, cujo objetivo é gerar o código intermediário em formato de quádruplas, traduzi-lo para a linguagem de montagem (*Assembly*) e, finalmente, produzir o código executável final em **binário**.

Este compilador foi projetado com o objetivo de gerar um código executável para um processador com arquitetura ARM, desenvolvido na disciplina de Arquitetura e Organização de Computadores, o qual utiliza uma arquitetura RISC (Reduced Instruction Set Computer) do tipo Harvard, com um conjunto de instruções de 32 bits. A geração de código assembly deste projeto foi, projetada para ser compatível com o conjunto de instruções desse processador. Essa integração entre o projeto de hardware (o processador) e o de software (o compilador) foi realizada com intuito educacional de compreender como os sistemas computacionais funcionam.

O Capítulo 2 detalha a arquitetura do processador ARM para o qual o compilador foi desenvolvido, incluindo seus componentes, conjunto de instruções e caminho de dados. O Capítulo 3 aborda a fase de análise do compilador, descrevendo as etapas léxica, sintática e semântica. O Capítulo 4 explora a fase de síntese, com foco na geração do código intermediário, do código Assembly e do executável final. O Capítulo 5 apresenta exemplos práticos de compilação, ilustrando a tradução de códigos-fonte em C- até o seu formato executável e a correspondência entre as etapas. Por fim, o Capítulo 6 sumariza os resultados, as dificuldades encontradas e os destaques do projeto.

2 O Processador

Essa seção detalha a arquitetura do processador ARM, o sistema alvo para o qual o compilador foi projetado. Será apresentado seu diagrama de blocos e a funcionalidade de seus componentes. Além disso, serão descritos o conjunto de instruções que o processador executa e a organização de sua memória, que são fundamentais para a geração de código.

2.1 Caminho de Dados

Para garantir que as instruções sejam interpretadas corretamente pelo processador, é fundamental dispor de estruturas capazes de decifrá-las de maneira adequada. Para essa implementação, foi empregado o datapath representado na figura ???. Nesse esquema, destacam-se o Contador de Programa (PC), a memória de instruções, a Unidade de Instrução, a Unidade de Controle, o Banco de Registradores, a Unidade Lógica e Aritmética (ULA), a memória de dados, diversos multiplexadores que direcionam os dados pelo processador, o módulo de entrada, o módulo de saída e o módulo divisor do clock.

A unidade de instrução desempenha um papel intermediário crucial ao referenciar adequadamente os bits dos registradores para o banco de registradores. A estrutura é necessária devido aos campos das instruções, os quais não possuem uma estrutura fixa e podem variar conforme o tipo da instrução.

As cores na figura ?? representam os tipos de informação em cada trecho do caminho de dados, os seus significados estão descritos no canto inferior direito da imagem. Já os retângulos cinza são as saídas ou entradas que serão conectadas ao FPGA.

2.1.1 *DataPath - Instruções de processamento de dados*

Para que uma instrução do tipo de processamento de dados seja realizada tem-se os seguintes passos identificados na figura 1.

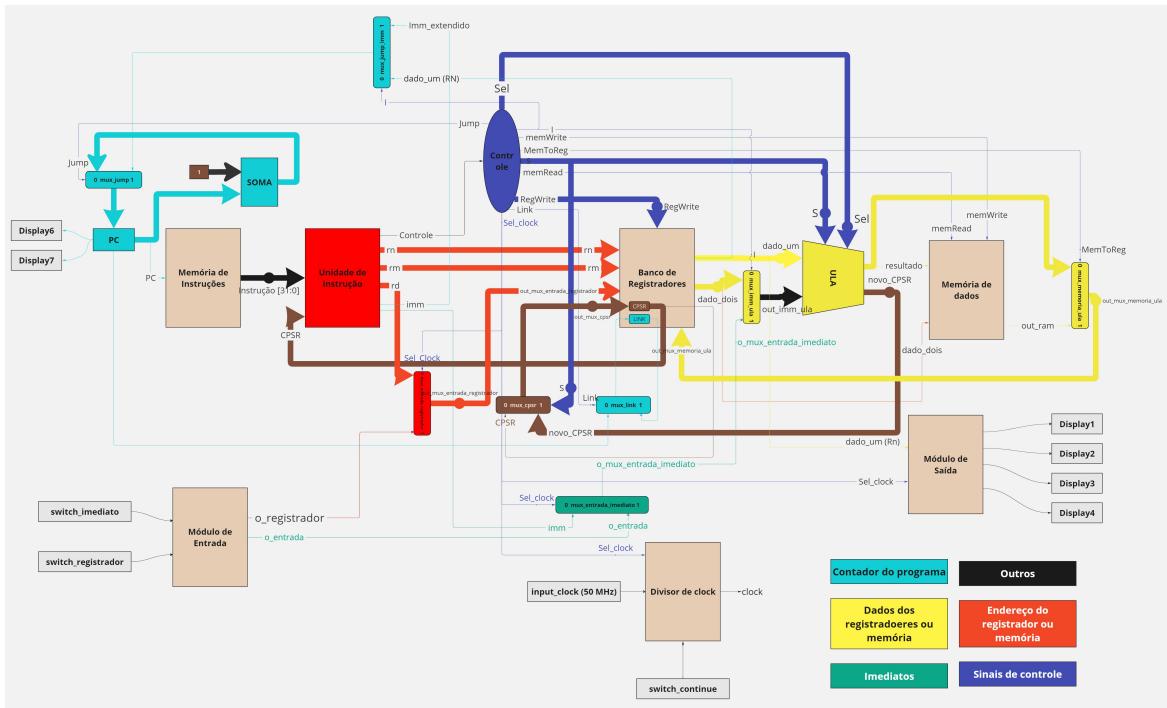
Do lado esquerdo observa-se um circuito sequencial, o qual sempre soma 1 ao PC para ser realizada a próxima instrução.

Do lado direito é visto uma instrução SUBSEQ sendo executada, ela sai da memória de instruções passa para a unidade de instrução na qual os bits que representam os bits dos endereços dos registradores são devidamente direcionados para os destinos. Os dados saem do banco de registradores são processados na ULA e depois escritos em algum registrador do banco, pois o *RegWrite* está ativo.

Como o campo cond está setado como EQ, a unidade de instrução só vai permitir que a instrução seja executada caso os valores do CPSR esteja com a mesma codificação.

Por fim, a unidade de controle manda a informação S para a ULA reescrever o dado presente no CPSR.

Figura 1 – Datapath das instruções de processamento de dados



Fonte: O Autor (2024)

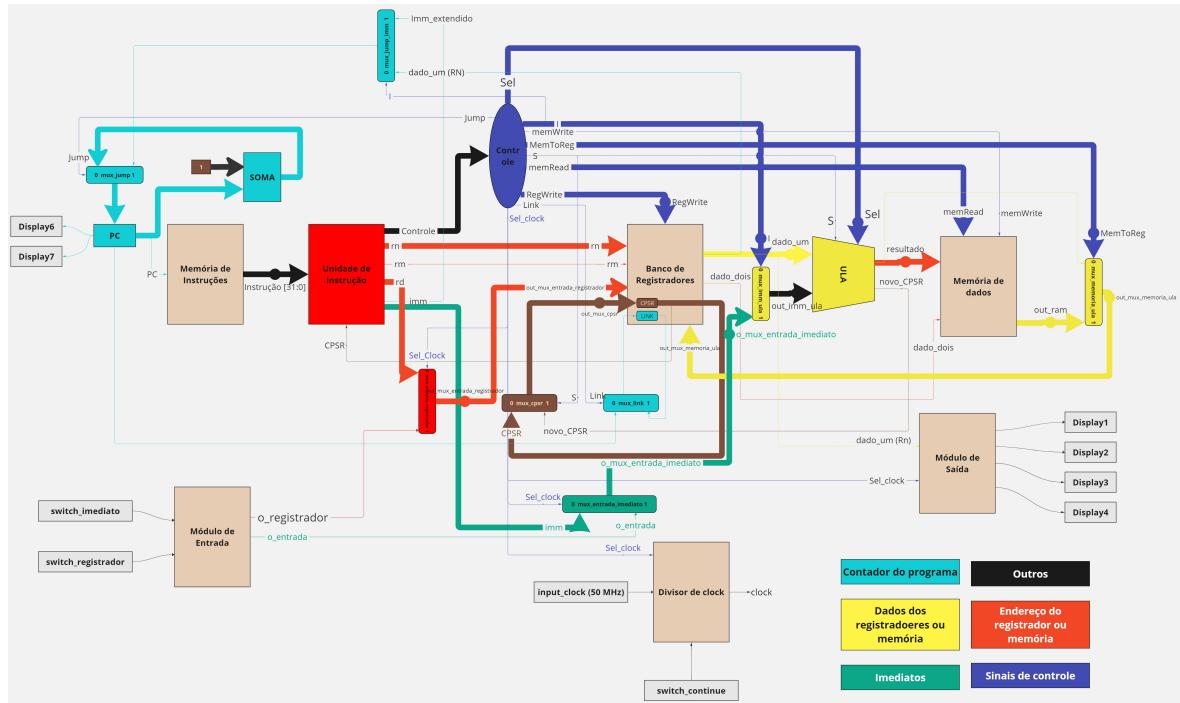
2.1.2 DataPath - Instruções de transferência de dados

A instrução LDREQ inicia-se com a memória de instrução enviando a instrução para a unidade de instrução que por sua vez distribui os sinais pelo processador caso esteja em conformidade com o CPSR. Rn e Ed são enviados para o banco de registradores, porém o valor do imediato é enviado diretamente para a ULA.

A soma entre o imediato e Rn é necessária para calcular o endereço da memória RAM e assim é enviada para a porta de endereço da memória principal. Com a ativação do sinal MemRead, torna-se possível a leitura do dado, que é então transferido para o registrador Rd.

A unidade de controle emite os comandos MemToReg para que a instrução lida na memória seja encaminhada para o registrador de destino, o sinal RegWrite, para permitir a escrita no registrador de destino, o sinal sel indica a operação de soma na ULA e o bit I indica a presença do imediato na instrução. Todo esse caminho de dados pode ser verificado na Figura 2.

Figura 2 – Datapath das instruções de transferência de dados



Fonte: O Autor (2024)

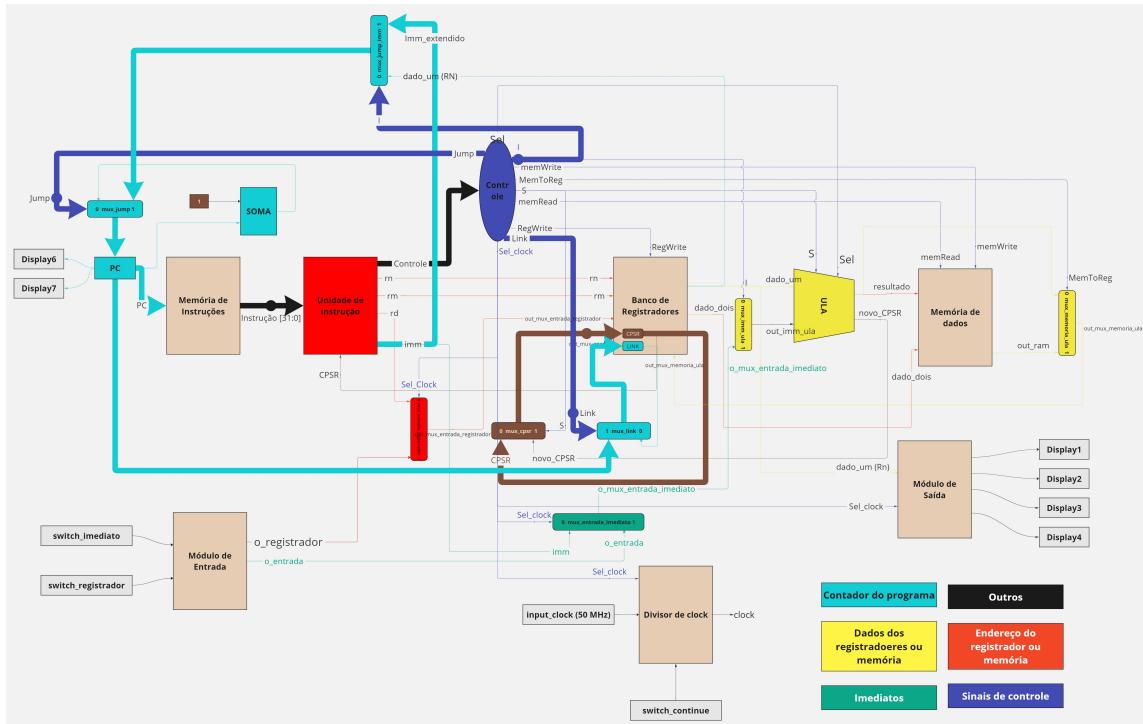
2.1.3 DataPath - Instruções de desvio

Diferente das anteriores, uma instrução de desvio impede a soma habitual de 1 ao PC.

No exemplo da figura 3 está sendo ilustrada a execução de uma instrução BIL, nesta instrução estão setadas as flags I e L, elas representam que um imediato está sendo utilizado e que o registrador especial link vai ser carregado com o valor atual do PC.

A unidade de controle manda os comandos de Jump e I. O comando Jump altera o multiplexador próximo ao PC, impedindo que ele seja incrementado de um. O I sinaliza para que o valor do imediato saia direto da unidade de instrução e vá para o PC.

Figura 3 – Datapath das instruções de desvio



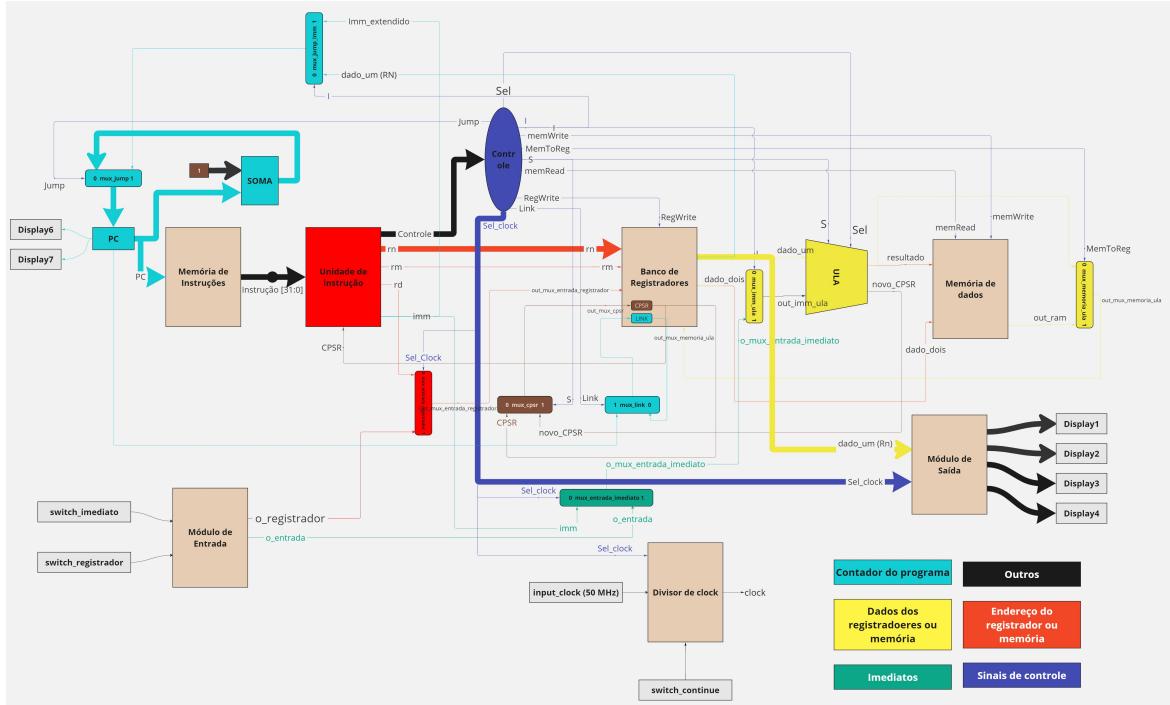
Fonte: O Autor (2024)

2.1.4 DataPath - Outras Instruções

As demais instruções possuem caminhos de dados bem distintos, por isso serão apresentadas as instruções de saída e entrada de dados.

A instrução de saída está ilustrada na figura 4, nela o pc é incrementado de maneira similar as outras instruções anteriores. A unidade de controle envia o sinal do registrador especial 29 para o módulo de saída, o qual atualiza os displays do FPGA caso o sinal de controle indique que seja uma instrução de saída.

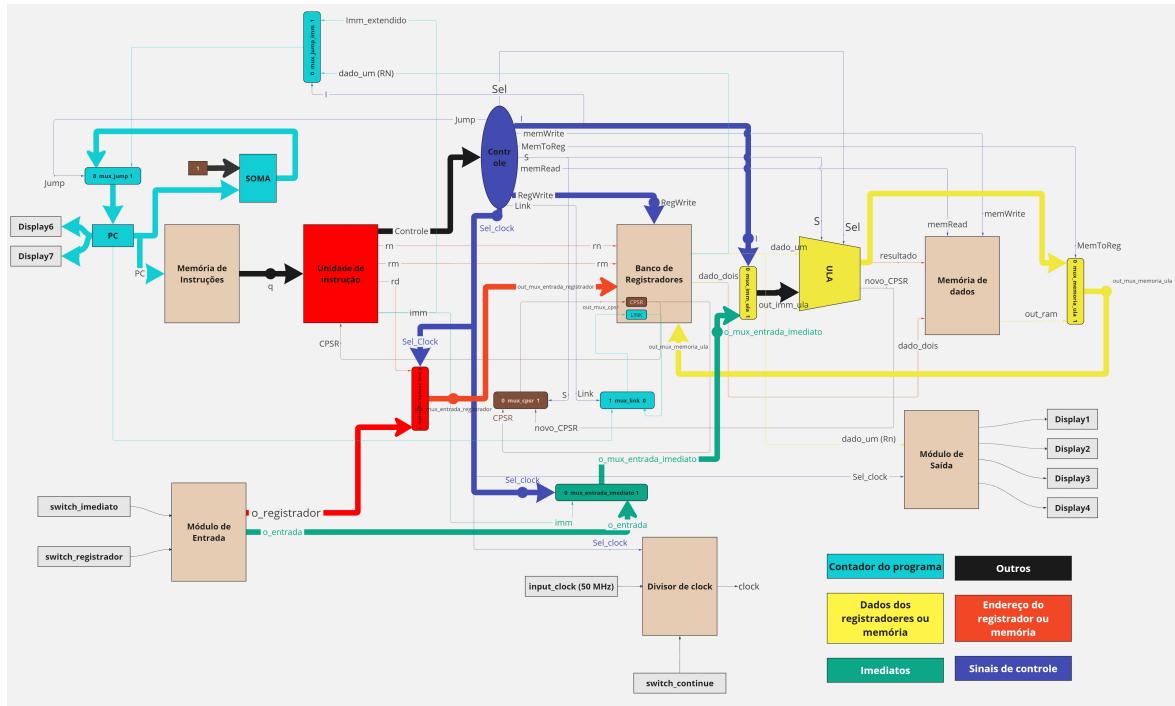
Figura 4 – Datapath das instrução OUT



Fonte: O Autor (2024)

Na instrução de entrada, o módulo de entrada recebe o valor do dado a ser salvo e o endereço de um registrador em que o dado será salvo. Para que esses sinais entrem no caminho de dados, a unidade de controle envia o sinal "sel clock" para os multiplexadores de entrada. O sinal segue um caminho similar a uma instrução de processamento de dados de soma e então é armazenada no registrador destino. Todo esse percurso pode ser visto na Figura 5.

Figura 5 – Datapath das instrução IN



Fonte: O Autor (2024)

2.2 Componentes

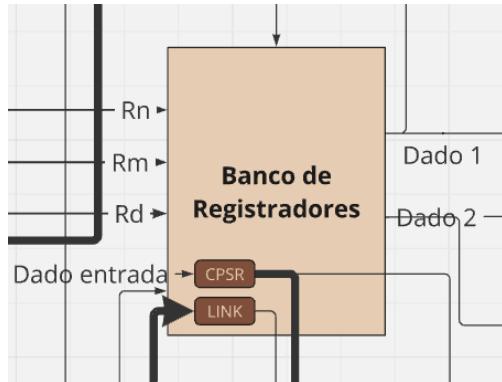
O processador é composto por um conjunto de unidades funcionais interconectadas descritos em verilog que trabalham em conjunto para buscar, decodificar e executar instruções. Cada componente possui uma responsabilidade específica, e a coordenação entre eles é o que permite o processamento de dados. As seções a seguir detalham a função de cada um desses módulos.

2.2.1 Banco de registradores

O banco de registradores consiste em um conjunto de 32 registradores de propósito geral. Esta implementação em Verilog utiliza um vetor, conforme o código abaixo. Cada registrador armazena 32 bits. As entradas para o banco de registradores incluem rn, rm, rd, a entrada de dados, o CPSR, o link e o *RegWrite*. As saídas são os dados 1 e 2, correspondentes aos valores armazenados em rn e rm, respectivamente. Todas as entradas e saídas estão representadas na Figura 6.

Além dos 32 registradores gerais, existem também os registradores especiais CPSR e link, cada um com suas próprias entradas e saídas. Para que o dado de entrada seja armazenado no registrador rd, o bit de controle chamado *RegWrite* deve estar ativo.

Figura 6 – Banco de registradores

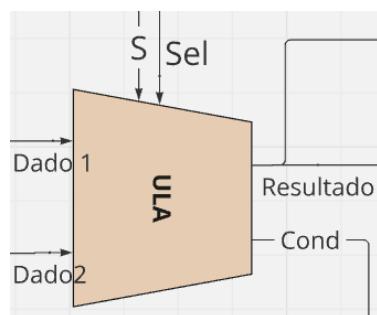


Fonte: O Autor (2024)

2.2.2 Unidade Lógica e Aritmética (ULA)

A Unidade Lógica e Aritmética (ULA) é responsável pela execução da maior parte das operações do processador, incluindo soma, subtração, multiplicação, divisão e operações lógicas. As entradas e saídas de dados da ULA são fixadas em 32 bits. Os sinais de controle são compostos por 4 bits para a seleção da operação e 1 bit para a ativação do envio de dados ao CPSR. Todas as entradas e saídas podem ser observados na figura 7. As operações executadas pela ULA podem ser examinadas na tabela ??.

Figura 7 – ULA



Fonte: O Autor (2024)

2.2.3 Program Counter (PC)

O PC (Program Counter) é responsável por indicar qual instrução deve ser executada. Ele é controlado por um clock que determina o ritmo das mudanças. Além disso, o PC possui uma entrada para receber a nova linha de instruções a ser lida e uma saída, que é configurada pela função *assign*. Além da função principal, o pc também atualiza os displays 7 e 6 com o valor do pc atual.

2.2.4 Módulo de entrada

Como visto na figura ??, o módulo de entrada possui as entradas `switchImediato` e `switchRegistrador`, estas duas entradas são conectadas diretamente a interruptores do Kit FPGA, `switchImediato` recebe um dado, enquanto o `switchRegistrador` informa em qual registrador o dado será salvo, estas duas entradas são passadas diretamente para as saídas `oEntrada` e `oRegistrador`, respectivamente. A implementação deste unidade está logo abaixo.

Para que os valores da entrada sejam lidos apenas quando uma instrução IN é inserida, o processador faz uso de dois multiplexadores, um que recebe o endereço do registrador e o outro o dado a ser salvo.

2.2.5 Módulo de saída

O módulo de saída é a unidade que faz a integração entre o processador e os displays do Kit FPGA. Ele possui duas entradas, o `dadoUm` que vem do registrador 29 e um sinal de controle `selClock` que vem da unidade de controle, este sinal de controle gerencia a atualização dos displays. As saídas são os displays 1, 2, 3 e 4 do Kit FPGA, podendo mostrar números até 9999.

A implementação do módulo de saída foi realizada em duas partes, na primeira o número vindo do registrador 29 é separado em unidade, dezena, centena e milhar, estes valores são salvados em variáveis que posteriormente são convertidos em sinais para os displays de sete segmentos no Módulo bcd.

2.2.6 Multiplexadores

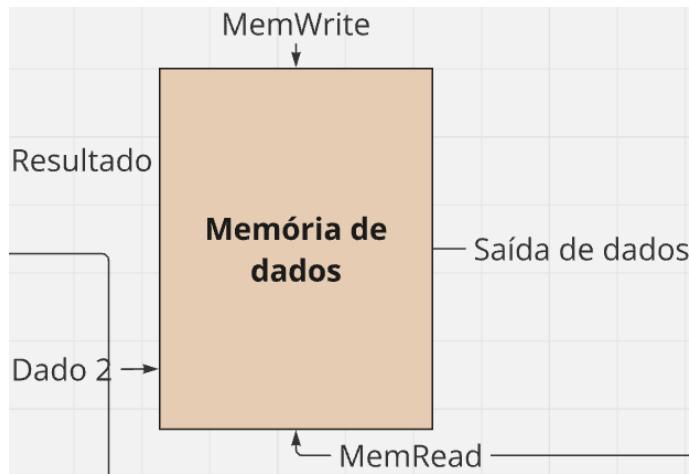
Os multiplexadores são componentes cruciais para a correta direção do fluxo de informações no caminho de dados. Eles atuam como seletores que, com base em um sinal de controle proveniente da Unidade de Controle, escolhem qual de suas várias entradas de dados será encaminhada para sua única saída. No processador, eles são utilizados, por exemplo, para decidir se o próximo valor do PC será o endereço incrementado ou um endereço de desvio, e para selecionar se o dado a ser escrito em um registrador virá da ULA ou da memória de dados.

2.2.7 Memória de dados (RAM)

A memória de dados, também conhecida como *random access memory* (RAM), é a memória principal do sistema. Ao contrário dos registradores, que são limitados a 32 unidades, a RAM possui um tamanho significativamente maior, com 1024 posições, cada uma com 32 bits. Entre as portas de entrada, estão o resultado da ULA e o dado 2,

que pode ser um valor imediato ou um dado armazenado em um registrador, esse dado é utilizado para armazenamento na memória. Os bits de controle incluem memWrite, que é ativado para gravar dados na memória, e memRead, que é usado para ler dados da memória. A saída de dados é fornecida por uma porta de saída de 32 bits. Todas as portas estão ilustradas na figura 8.

Figura 8 – Memória de dados (RAM).



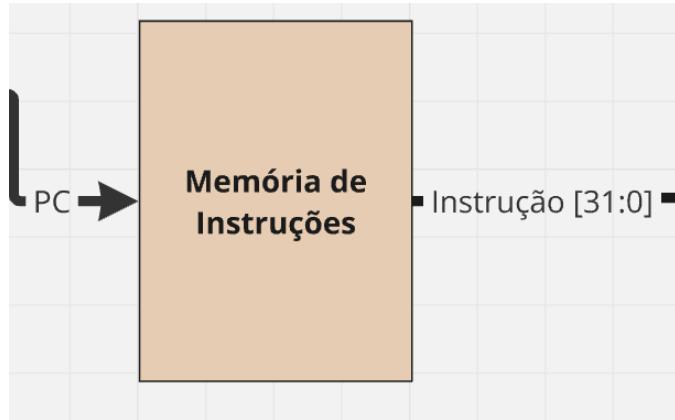
Fonte: O Autor (2024)

Para a implementação da memória de dados, foi utilizado um template fornecido pelo *Quartus Prime*. Foram ajustados apenas os parâmetros: DATA-WIDTH, configurado para 32, que define o tamanho dos espaços de memória, e ADDR-WIDTH, configurado para 10, que define a memória com 1024 posições. O template declara as entradas e saídas, define a variável RAM e implementa um bloco always para conectar o resultado acessado com a saída. Além disso, inclui uma estrutura condicional para implementar as funcionalidades de memRead e memWrite.

2.2.8 Memória de Instruções (ROM)

A memória de instruções (ROM) é a memória onde são armazenadas todas as instruções que serão lidas pelo processador, ela possui uma ligação direta com o PC, e uma saída que sai a instrução de 32 bits. Essa estrutura pode ser visualizada na figura 9.

Figura 9 – Memória de instruções (ROM).



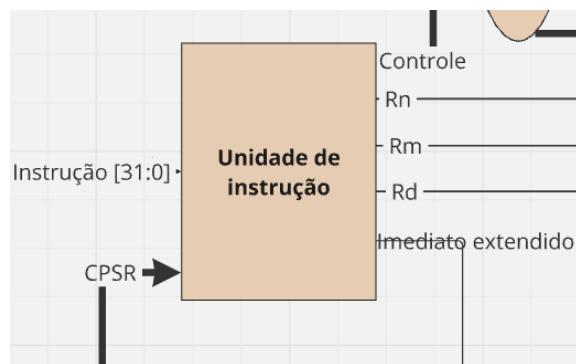
Fonte: O Autor (2024)

2.2.9 Unidade de instrução

Esta unidade foi projetada para lidar com os diferentes tipos de instrução e suas variações de formato. Além disso, ela implementa a funcionalidade do registrador especial CPSR. Caso o campo cond da instrução não corresponda à informação contida no CPSR, a instrução a ser executada é a NOP, o que significa que nenhuma ação será realizada. Além dessas funcionalidades, a unidade também gera uma string de controle formatada, que facilita o trabalho da unidade de controle. Essa string inclui todas as informações necessárias para ativar os multiplexadores e os bits de sinal para as demais unidades.

A unidade possui uma entrada para a instrução completa e outra para o valor atual do CPSR. Como saída, fornece os endereços de todos os registradores e o imediato, já estendido para 32 bits. Toda essa configuração pode ser visualizada na Figura 10.

Figura 10 – Unidade de instrução.



Fonte: O Autor (2024)

Para implementar diversas funcionalidades, foram necessárias várias instruções condicionais if e case. A primeira condição avalia se o campo cond corresponde ao CPSR,

caso contrário, todas as saídas são configuradas para uma instrução NOP.

Para lidar com os diferentes formatos de instrução, foi utilizada uma estrutura case, onde cada tipo de instrução é tratado individualmente. Por fim, a saída de controle também é gerida de acordo com o tipo de instrução.

2.2.10 Unidade de controle

A unidade de controle é responsável por receber o sinal de controle da unidade de instrução e enviar os sinais apropriados para todos os multiplexadores, bem como para o funcionamento adequado de cada unidade do processador. Os sinais de controle são: Jump, MemWrite, MemRead, MemToReg, S, Link, RegWrite, I e selclock.

O sinal de controle é uma string de 11 bits que possui uma formatação especial ilustrada na tabela 1. Esta formatação é realizada pela unidade de instrução e ela facilita o processamento do sinal na unidade de controle. Caso um campo não esteja presente na instrução, ele será 0.

Tabela 1 – Formato do sinal de controle

[10:9]	[8]	[7]	[6]	[5]	[4]	[3:0]
Sel	I	S	L/S	L	U	Opcode

Fonte: O Autor (2024)

2.2.11 Módulo de divisão do clock

A unidade divisoria de clock reduz a frequência do clock para que as instruções possam ser visualizadas no Kit FPGA.

A primeira das entradas é a selClock, que seleciona o modo de operação da unidade de acordo com a instrução. Instruções normais são executadas em uma frequência mais alta, enquanto instruções de entrada e saída são executadas mais lentamente. A instrução finish trava o clock e sinaliza para o processador que o programa foi finalizado. A implementação dessa unidade pode ser vista abaixo.

O clock do FPGA é inserido pela entrada *CLOCK50* e, a cada quantidade estipulada de ciclos, o sinal é invertido. Isso permite gerar um sinal de clock com a frequência desejada para o processador. Com a frequência do clock do FPGA em 50 MHz, para selecionar uma frequência de 1 MHz, o sinal deve ser invertido a cada 25 pulsos. Para as instruções IN e OUT, o período escolhido foi de 5 segundos, então o sinal é invertido a cada 250 milhões de pulsos.

No código, a estrutura condicional da linha 40 impede que o clock continue a operar caso o sinal continue esteja em baixa. A instrução FINISH, codificada no sinal selClock,

também atua como uma condição de parada do clock. Sua implementação pode ser vista na linha 34, onde o contador recebe continuamente o valor 0.

O sinal Jump determina se a próxima instrução será obtida a partir do endereço PC+1 ou de um valor arbitrário, especificado pelo código da instrução. O sinal MemWrite é direcionado para a Memória de Dados (RAM) e indica se o dado na entrada deve ser escrito, enquanto o sinal MemRead determina se a memória pode ser lida.

O sinal S define se uma instrução de processamento de dados pode modificar o valor do CPSR (Status do Programa), direcionando o resultado da condição (cond) da ULA para o registrador CPSR. Já o sinal Link realiza uma função similar ao sinal S, mas direciona o valor atual do PC para um registrador especial chamado Link.

O sinal RegWrite indica se o dado calculado pela ULA pode ser armazenado no Banco de Registradores. O sinal MemToReg seleciona qual informação será enviada para o Banco de Registradores, podendo ser o resultado calculado pela ULA ou o dado lido da Memória de Dados (RAM).

O sinal I indica se a instrução contém um valor imediato, direcionando-o diretamente para a ULA em instruções de processamento de dados ou para o PC em instruções de desvio.

Além dos sinais de controle, a unidade de controle também envia uma string chamada "sel", que codifica a operação a ser executada pela ULA, e o sinal selclock, que informa o tipo de instrução em execução para a Unidade de Divisão do Clock e algumas outras unidades. Os possíveis valores para o selclock podem ser vistos na tabela 2.

Tabela 2 – Operações do sinal SelClock

Instrução	SelClock
Normal	00
OUT	01
IN	10
Finish	11

Fonte: O Autor (2024)

2.3 Conjunto de Instruções

Na implementação do processador ARM serão utilizados quatro tipos de instruções: Instruções de processamento de dados, instruções de transferência de dados, instruções de desvio e outras instruções. Deste modo, são necessários 2 bits de distinção.

2.3.1 Instruções de Processamento de dados

O primeiro tipo de instrução corresponde ao processamento de dados e é identificado pela decodificação dos bits 00. O formato dessas instruções pode ser visto na Tabela 3.

Essa categoria de instrução é caracterizada pela presença de um Opcode que define o tipo de operação, um bit "I" para indicar a presença de um valor imediato, um bit "S" para determinar se o CPSR será alterado, dois campos para registradores e, por fim, o último campo pode ser um registrador ou um valor imediato, dependendo do bit "I". As possíveis instruções estão descritas na tabela 3, as quais dependem do OPcode, do campo I e do S. Qualquer instrução de processamento de dados pode conter um imediato.

Tabela 3 – Instruções de Processamento de Dados

Opcode	I	S	Nome	Função
0001	0	0	AND	$rd \leftarrow rn \& rm$
0001	1	0	ANDI	$rd \leftarrow rn \& Imm$
0011	0	0	SUB	$rd \leftarrow rn - rm$
0011	1	0	SUBI	$rd \leftarrow rn - Imm$
0101	0	0	ADD	$rd \leftarrow rn + rm$
0101	1	0	ADDI	$rd \leftarrow rn + Imm$
1010	0	1	CMP	$CPSR \leftarrow rn - rm$
1100	0	0	ORR	$rd \leftarrow rn OR Rm$
1100	1	0	ORRI	$rd \leftarrow rn OR Imm$
1101	0	0	MOV	$rd \leftarrow Rm$
1101	1	0	MOVI	$rd \leftarrow Imm$
1110	0	0	MUL	$rd \leftarrow rn * rm$
1111	0	0	UDIV	$rd \leftarrow rn / rm$

Fonte: O Autor (2024).

2.3.2 Instruções de Transferência de Dados

As instruções de transferência de dados são empregadas para carregar ou copiar dados da memória. Elas são identificadas pela codificação dos bits 01 no campo "Tipo de instrução" e apresentam três bits de codificação.

O bit I indica se a instrução contém um valor imediato. O bit U seleciona a direção do deslocamento no modo de endereçamento direto por deslocamento, optando por posições menores ou maiores na memória. Por fim, quando o bit L/S é igual a 1, indica que a instrução é de carregamento (*LDR*), caso contrário, quando o bit L é igual a 0, indica uma instrução de armazenamento (*STR*).

O formato das instruções de load e store pode ser visto nas Tabelas 4 e 5, respectivamente. Nas instruções LDR, o campo [23] é 1, indicando uma instrução LDR. Nesse caso, o campo [17:13] apresenta o registrador no qual os dados serão carregados. Em contraste, nas instruções STR, o mesmo campo [17:13] representa o registrador de onde os dados

serão retirados. Essas operações estão melhor representadas na Tabela 6, na qual a base representa o endereço da memória RAM.

Tabela 4 – Formato da instrução LDR

[31:28]	[27:26]	[25]	[24]	[23]	[22:18]	[17:13]	[12:0]
Cond	Tipo Instrução	I	U	1	Rn	Rd	Imm

Fonte: O Autor (2024)

Tabela 5 – Formato da instrução STR

[31:28]	[27:26]	[25]	[24]	[23]	[22:18]	[17:13]	[12:0]
Cond	Tipo Instrução	I	U	0	Rn	Rm	Imm

Fonte: O Autor (2024)

Tabela 6 – Instruções de Transferência de Dados

I	U	L	Nome	Função
1	0	1	LDR	$Rd \leftarrow Base[Rn + Imm]$
1	0	0	STR	$Base[Rn + Imm] \leftarrow Rm$

Fonte: O Autor (2024)

2.3.3 Instruções de Desvio

As instruções de desvio, codificadas como 10, incluem o campo I para indicar a presença de um valor imediato, junto com o bit L, que sinaliza a necessidade de atualizar o registrador "Link". Se o bit I no campo [25] for 1, o processador utilizará o formato da Tabela 7. Caso contrário, será usado o formato da Tabela 8.

Essas instruções alteram o valor do contador de programa (PC), o que resulta em uma mudança na posição atual da memória de instruções. As instruções implementadas estão mostradas na Tabela 9, na qual também pode ser observada a alteração no registrador Link.

Tabela 7 – Formato das instruções B

[31:28]	[27:26]	[25]	[24]	[23:0]
Cond	Tipo Instrução	1	L	Imm

Fonte: O Autor (2024)

Tabela 8 – Formato das instruções BI

[31:28]	[27:26]	[25]	[24]	[23:19]	[18:0]
Cond	Tipo Instrução	0	L	Rn	xxx

Fonte: O Autor (2024)

Tabela 9 – Instruções de desvio

I	L	Nome	Função
1	0	BI	PC ← Imm
1	1	BIL	PC ← Imm and L ← PC
0	0	B	PC ← Rn
0	1	BL	PC ← Rn and L ← PC

Fonte: O Autor (2024)

2.3.4 Outras Instruções

Outras instruções de uso geral são codificadas como 11. Elas possuem apenas um campo de OPcode, conforme ilustrado na tabela 10.

Dentre as instruções de uso geral, está a instrução NOP, que tem a finalidade de pular um ciclo de clock sem alterar o processador. A instrução IN indica que um dado será inserido no módulo de entrada. A instrução OUT envia o dado de um registrador específico para o módulo de saída. Por fim, a instrução FINISH encerra o programa. Todas essas instruções e seus respectivos opcodes podem ser vistos na tabela 11.

Tabela 10 – Formato das outras instruções

[31:28]	[27:26]	[25:23]	[22:0]
Cond	Tipo Instrução	Opcode	xxx

Fonte: O Autor (2024)

Tabela 11 – Outras Instruções

Opcode	Nome
000	NOP
001	IN
010	OUT
011	FINISH

Fonte: O Autor (2024)

2.4 Organização da Memória

O projeto do processador é fundamentado no modelo de arquitetura Harvard, o que implica uma separação física entre o armazenamento de instruções e o de dados. Essa escolha de design resulta em dois sistemas de memória independentes e com barramentos distintos: uma memória de instruções (ROM) e uma memória de dados (RAM).

A memória de instruções, com capacidade para 256 palavras de 32 bits, é responsável por armazenar o código do programa. O seu endereçamento é realizado pelo Contador

de Programa (PC), que dita a sequência de instruções a serem buscadas e executadas e também pelas instruções de branch.

Em paralelo, a memória de dados oferece um espaço de 1024 posições para o armazenamento de variáveis e outras informações manipuladas durante a execução. O acesso a esta memória é realizado por meio das instruções de transferência de dados do processador, LDR (Load) e STR (Store). O endereço efetivo para essas operações é calculado pela Unidade Lógica e Aritmética (ULA), que soma o conteúdo de um registrador base com um valor de deslocamento imediato, permitindo um acesso flexível aos dados na pilha ou em outras regiões da memória.

3 Compilador: Fase de Análise

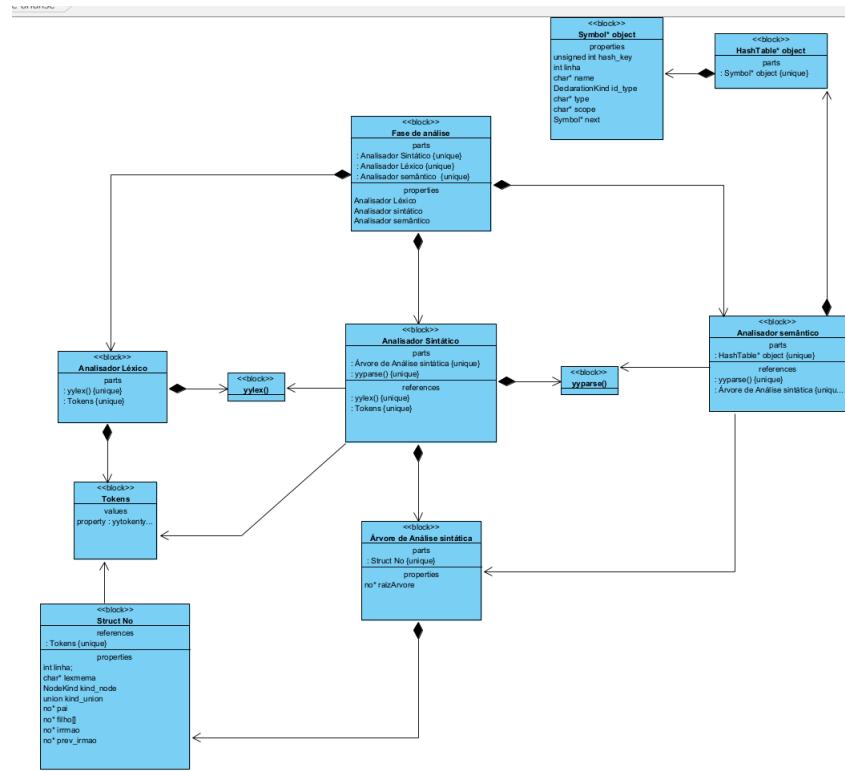
Este capítulo aborda a fase de análise do compilador. A principal responsabilidade desta fase é decompor o código-fonte para verificar sua validade estrutural, gramatical e de significado, construindo uma representação intermediária que será utilizada pela fase de síntese.

Inicialmente, serão apresentados os modelos em SysML que ilustram o fluxo de trabalho e a interação entre os componentes desta etapa. Em seguida, serão detalhadas as três sub-fases que a compõem: a **Análise Léxica**, responsável por converter o fluxo de caracteres do código-fonte em uma sequência de *tokens*; a **Análise Sintática**, que verifica se esses *tokens* seguem as regras gramaticais da linguagem C- e constrói a Árvore de Sintaxe Abstrata (AST); e, por fim, a **Análise Semântica**, que utiliza a AST e a Tabela de Símbolos para validar a coerência e o significado do código.

3.1 Modelagem

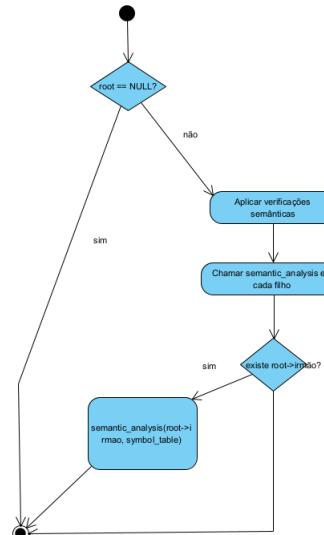
Abaixo está ilustrado o diagrama de blocos na figura 11 e os diagramas de atividade podem ser vistos nas figuras 12 e 13.

Figura 11 – Diagrama de blocos - Fase de Análise



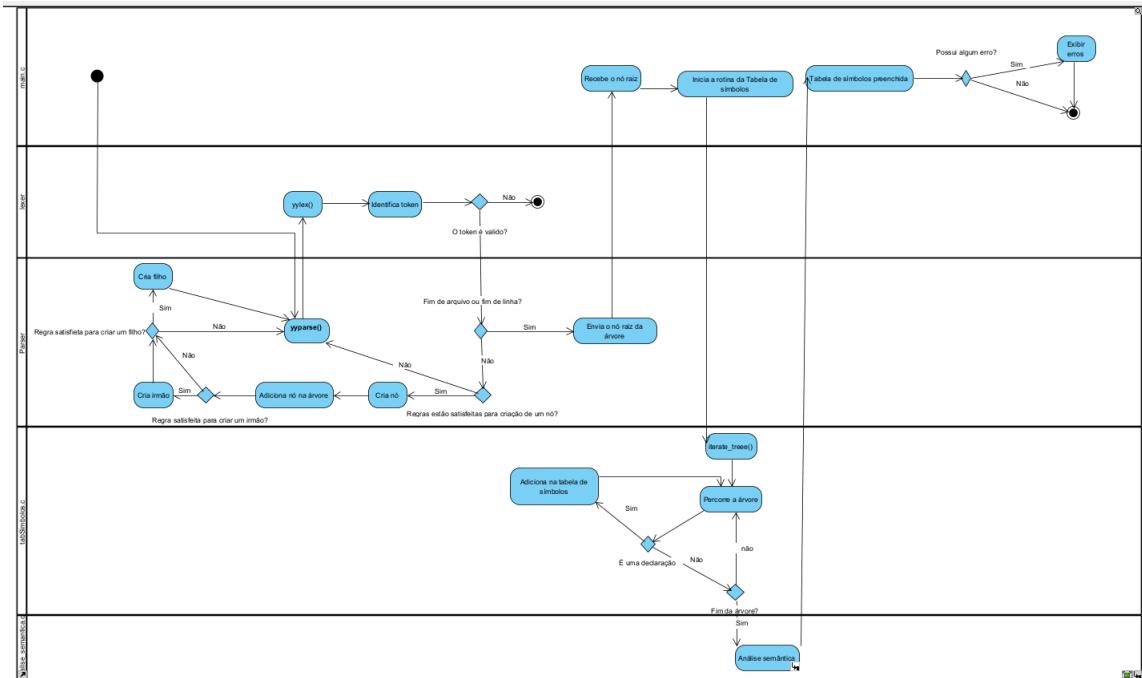
Fonte: O Autor (2025)

Figura 12 – Diagrama de atividades - Analise semântica



Fonte: O Autor (2025)

Figura 13 – Diagrama de pacotes do compilador



Fonte: O Autor (2025)

3.2 Análise Léxica

A análise léxica, também conhecida como *scanner*, é a primeira fase do processo de compilação. Sua principal função é ler o código-fonte como uma sequência de caracteres e convertê-lo em uma série de componentes significativos, denominados *tokens*. Cada *token* representa uma unidade lexical da linguagem, como palavras-chave, identificadores, constantes numéricas, operadores e símbolos. Além de gerar os *tokens*, o analisador léxico é responsável por ignorar elementos que não possuem significado para a compilação, como espaços em branco, tabulações, novas linhas e comentários.

3.2.1 Implementação do Analisador Léxico

No desenvolvimento deste projeto, a implementação do analisador léxico foi realizada com o auxílio da ferramenta **Flex (Fast Lexical Analyzer Generator)**. O Flex é um gerador de analisadores léxicos que, a partir de um arquivo de entrada contendo um conjunto de regras baseadas em expressões regulares, gera automaticamente um programa em linguagem C (o arquivo `lex.yy.c`) capaz de realizar a análise e o reconhecimento dos *tokens* da linguagem C-.

As regras que definem cada *token* foram especificadas no arquivo `src/lexical_analyser.l`. A tabela a seguir, extraída da documentação do projeto, detalha os principais *tokens*

reconhecidos pelo compilador e suas respectivas expressões regulares.

Tabela 12 – Tokens e Expressões Regulares da Linguagem C-

Token	Expressão Regular
Identificador	[A-Za-z]+([A-Za-z])*
Número	[1-9]+[0-9]*
Palavras Reservadas	if else while int void return
Comentários	/.../ ou //...
Símbolos	{ } () [] ; . ,
Operadores	+ - / * == != > < >= <= =

Quando o analisador léxico identifica uma sequência de caracteres que corresponde a uma dessas regras, ele retorna o *token* correspondente para a fase seguinte, a análise sintática. Por exemplo, ao encontrar a sequência `while`, ele a classifica como a palavra-chave `WHILE`, e ao encontrar `var1`, a classifica como um ID (Identificador). Essa etapa é fundamental para simplificar a tarefa do analisador sintático, que passa a operar sobre uma sequência estruturada de *tokens* em vez de um fluxo bruto de caracteres.

3.2.2 Implementação em Código

O arquivo de definição para o Flex (.l) é estruturado em três seções, separadas por `%%`. A primeira seção contém definições e inclusões de código C. A segunda seção, a principal, contém os pares de expressão regular e a ação em C a ser executada quando a expressão é reconhecida. A terceira seção contém código C adicional.

A seguir, é apresentado um trecho representativo da implementação em `src/lexical_analyser.l`, demonstrando como as palavras-chave, identificadores e operadores são reconhecidos.

```

1 %{  

2 #include "../globals.h"  

3 #include "../analise_sintatica.tab.h"  

4 #include <string.h>  

5  

6 char *id_lexema;  

7 %}  

8  

9 %option yylineno  

10  

11 %%  

12  

13 "if"           { return IF; }  

14 "else"         { return ELSE; }  

15 "while"        { return WHILE; }  

16 "int"          { return INT; }  

17 "void"         { return VOID; }
```

```

18 "return"           { return RETURN; }
19
20 [a-zA-Z_][a-zA-Z0-9_]* { id_lexema = strdup(yytext); return ID; }
21 [0-9]+             { yyval.ival = atoi(yytext); return NUM; }
22
23 "+"                { return PLUS; }
24 "-"                { return MINUS; }
25 "*"               { return TIMES; }
26 "/"                { return OVER; }
27 "=="               { return EQ; }
28 "!="               { return NEQ; }
29 "<"               { return LT; }
30 "<="               { return LTE; }
31 ">"               { return GT; }
32 ">="               { return GTE; }
33 "="                { return ASSIGN; }
34
35 [ \t\r\n]+          { /* Ignora espacos em branco */ }
36 "/*".* { /* Ignora comentarios de linha */ }
37 "/*([^\n]*|\n+[^*/])* */ { /* Ignora comentarios de bloco */ }
38
39 .                  { fprintf(stderr, "Erro lexico na linha %d:
40         caractere invalido '%s'\n", yylineno, yytext); }
41 %%
42
43 int yywrap(void) {
44     return 1;
45 }
```

Listing 3.1 – Trecho do arquivo de definição do Flex (`lexical_analyser.l`)

No código acima (Listagem 3.1), cada vez que uma expressão regular é casada com o texto de entrada (armazenado na variável `yytext`), a ação correspondente em C é executada. Para palavras-chave, a ação é retornar o *token* apropriado, como `ID`. Para identificadores, o lexema é duplicado e armazenado na variável global `id_lexema` antes de retornar o *token* `ID`, permitindo que o analisador sintático tenha acesso ao nome do identificador.

3.3 Análise Sintática

A análise sintática, também conhecida como *parsing*, é a segunda fase da compilação. Sua função é receber a sequência de *tokens* gerada pelo analisador léxico e verificar se essa sequência obedece à estrutura gramatical da linguagem de programação. O analisador sintático valida se os *tokens* estão organizados de acordo com as regras formais da gramática

e, como resultado principal, constrói uma estrutura de dados hierárquica chamada de Árvore de Sintaxe Abstrata (AST - *Abstract Syntax Tree*). A AST representa a estrutura sintática do código-fonte e será a base para as etapas subsequentes de análise semântica e geração de código.

3.3.1 Implementação do Analisador Sintático

Para este projeto, o analisador sintático foi implementado utilizando a ferramenta **Bison**, um gerador de analisadores sintáticos para a linguagem C. O Bison opera a partir de um arquivo de entrada (`src/analise_sintatica.y`) que contém a gramática formal da linguagem C-, escrita em um formato semelhante à Notação de Backus-Naur (BNF). A partir dessas regras gramaticais, o Bison gera um parser em C (o arquivo `analise_sintatica.tab.c`) que implementa um autômato de pilha para reconhecer a linguagem.

Durante o processo de análise, à medida que o parser reconhece as regras da gramática, ele executa ações semânticas associadas a cada regra. No contexto deste compilador, essas ações consistem principalmente na construção da AST. Para isso, são invocadas funções auxiliares, como `create_node()` e `add_filho()`, que estão definidas no arquivo `src/arvore.c`. Cada nó criado na árvore armazena informações sobre o elemento sintático que representa, como o tipo do nó (declaração, expressão, etc.), seu lexema e o número da linha no código-fonte.

3.3.2 Estrutura do Arquivo de Gramática

O arquivo `src/analise_sintatica.y` é dividido em três partes principais, separadas por `%%`.

3.3.2.1 Seção de Declarações e Configurações

A primeira seção do arquivo é dedicada às configurações do Bison e ao código C que será inserido diretamente no topo do arquivo gerado. Este trecho inclui:

- **Inclusões C (%{ ... %}):** Importação das bibliotecas C padrão e dos arquivos de cabeçalho do projeto, como `globals.h`, que define a estrutura dos nós da AST e outras estruturas de dados globais.
- **Declaração de Tokens (%token):** Lista todos os *tokens* (símbolos terminais) que o analisador léxico pode fornecer, como `IF`, `WHILE`, `ID`, `NUM`, e operadores como `PLUS` e `ASSIGN`.

- **União de Tipos (%union):** Define uma união de tipos de dados para associar valores semânticos aos *tokens* e às regras. No projeto, o principal tipo é um ponteiro para um nó da AST (`struct no *node`).
- **Associação de Tipos (%type):** Vincula um tipo da `%union` a um símbolo não-terminal (uma regra da gramática). Por exemplo, `%type <node>` comando especifica que o valor semântico associado à regra ‘comando’ é um ponteiro para um nó da árvore.

```

1 %{
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include "../globals.h"
6
7 extern int yylex();
8 extern int yylineno;
9 extern char *yytext;
10 No *raizArvore; // Variavel global para a raiz da AST
11 %}
12
13 /* Declaracao dos tokens vindos do Flex */
14 %token IF ELSE WHILE INT VOID RETURN
15 %token ID NUM
16 %token PLUS MINUS TIMES OVER EQ NEQ LT LTE GT GTE ASSIGN
17
18 /* Definicao dos tipos de nos para a AST */
19 %union {
20     struct no *node;
21     int ival;
22     char *sval;
23 }
24
25 /* Associacao de tipos aos nao-terminalis */
26 %type <node> declaracao_lista declaracao declaracao_variavel
27 %type <node> tipo_especificador declaracao_funcao
28 %type <node> comando_composto declaracao_local
29 %type <node> comando_lista comando comando_selecao comando_iteracao
30 %type <node> expressao expressao_simples
31
32 %%
33 /* ... Regras da Gramatica ... */

```

Listing 3.2 – Seção de declarações do arquivo `analise_sintatica.y`

3.3.2.2 Seção de Regras da Gramática

Esta é a seção central do arquivo, onde a gramática da linguagem C- é formalmente definida através de regras de produção. Cada regra especifica como um símbolo não-terminal pode ser derivado de uma sequência de outros símbolos (terminais ou não-terminais).

3.3.2.2.1 Regra Inicial.

A análise começa pela regra `programa`, que é definida como uma `declaracao_lista`. Quando esta regra é completamente reconhecida, a ação associada (`{ raizArvore = $1; }`) atribui o nó da árvore resultante (`$1`) à variável global `raizArvore`, que se torna a raiz de toda a AST.

```
34 /* Regra inicial da gramatica */
35 programa: declaracao_lista
36 {
37     raizArvore = $1;
38     $$ = $1;
39 }
```

3.3.2.2.2 Regras de Declaração.

As regras de declaração lidam com a definição de variáveis, vetores e funções. A regra para `declaracao_variavel`, por exemplo, trata tanto de variáveis simples (`int x;`) quanto de vetores (`int v[10];`). A ação semântica correspondente cria um nó do tipo `var_k` ou `array_k` e o associa ao seu tipo.

```
40 declaracao_variavel: tipo_especificador ID ';' 
41 {
42     $$ = create_node(yylineno, $2, declaration_k,
43     var_k);
44     add_filho($$, $1); // $1 é o tipo (int/void)
45 }
46 | tipo_especificador ID '[' NUM ']' ' ';
47 {
48     $$ = create_node(yylineno, $2, declaration_k,
49     array_k);
50     No* tamanho = create_node(yylineno, $4,
51     expression_k, constant_k);
52     add_filho($$, tamanho);
53     add_filho($$, $1);
54 }
```

3.3.2.2.3 Regras de Comando.

Estas regras definem as estruturas de controle e os comandos da linguagem. A regra para `comando_selecao` (comando `if`), por exemplo, possui duas alternativas: uma para o `if` simples e outra para o `if-else`. As ações semânticas constroem um nó `if_k` e adicionam a expressão condicional e os blocos de comando como seus filhos. As variáveis `$$`, `$1`, `$2`, etc., referem-se, respectivamente, ao valor da regra e aos valores dos componentes na ordem em que aparecem.

```

52 comando_selecao: IF '(' expressao ')', comando
53 {
54     $$ = create_node(yylineno, "if", statement_k, if_k);
55     add_filho($$, $3);
56     add_filho($$, $5);
57 }
58 | IF '(' expressao ')', comando ELSE comando
59 {
60     $$ = create_node(yylineno, "if", statement_k, if_k);
61     add_filho($$, $3); // expressao condicional
62     add_filho($$, $5); // bloco 'then'
63     add_filho($$, $7); // bloco 'else'
64 };

```

3.3.2.2.4 Regras de Expressão.

As regras de expressão definem a estrutura de operações matemáticas, lógicas e de atribuição. Para garantir a precedência e a associatividade corretas dos operadores, as regras são escritas em múltiplos níveis (ex: `expressao`, `expressao_simples`, `soma_expressao`, `termo`, `fator`).

```

65 expressao: expressao_simples
66     | ID ASSIGN expressao
67     {
68         $$ = create_node(yylineno, "=", expression_k, assign_k);
69         No* id_node = create_node(yylineno, $1, expression_k, id_k)
70         ;
71         add_filho($$, id_node);
72         add_filho($$, $3);
73     };
74 soma_expressao: termo
75     | soma_expressao PLUS termo
76     {
77         $$ = create_node(yylineno, "+", expression_k, op_k);
78         add_filho($$, $1);
79         add_filho($$, $3);

```

```

80         }
81     | soma_expressao MINUS termo
82     {
83         $$ = create_node(yylineno, "-", expression_k, op_k);
84         add_filho($$, $1);
85         add_filho($$, $3);
86     };

```

3.3.2.3 Seção de Código Adicional

A terceira e última parte do arquivo contém código C auxiliar. A função mais importante nesta seção é a `yyerror(char const *s)`, que é chamada automaticamente pelo Bison quando um erro de sintaxe é detectado. A implementação no projeto reporta uma mensagem de erro indicando a linha onde a falha ocorreu, auxiliando na depuração do código-fonte.

```

87 %% /* Fim das regras da gramatica */
88
89 int yyerror(char const *s) {
90     fprintf(stderr, "Erro sintatico na linha %d: %s\n", yylineno, s);
91     return 1;
92 }

```

3.4 Análise Semântica

A análise semântica é a terceira fase do *front-end* e atua como uma ponte entre a análise sintática e a geração de código. Após a construção da Árvore de Sintaxe Abstrata (AST), o analisador semântico percorre essa árvore para verificar a consistência e o significado do programa, garantindo que ele esteja em conformidade com as regras semânticas da linguagem. Diferente da análise sintática, que apenas valida a estrutura, a análise semântica verifica se as construções sintaticamente válidas fazem sentido no contexto do programa.

As principais responsabilidades desta fase incluem:

- **Verificação de Tipos:** Garantir que os tipos dos operandos em uma expressão sejam compatíveis com o operador. Por exemplo, impedir a atribuição do resultado de uma função do tipo `void`.
- **Verificação de Declarações:** Assegurar que todas as variáveis, vetores e funções sejam declarados antes de seu uso e verificar se não há múltiplas declarações de um mesmo identificador em um mesmo escopo.

- **Coleta de Informações:** Popular a Tabela de Símbolos com informações detalhadas sobre cada identificador, como seu tipo, escopo (global ou local a uma função) e, posteriormente, seu deslocamento na memória.

Para realizar essas tarefas, o analisador semântico depende fundamentalmente da AST, que fornece a estrutura do programa, e da Tabela de Símbolos, que armazena o contexto dos identificadores.

3.4.1 Implementação do Analisador Semântico

No compilador desenvolvido, a análise semântica é implementada por um conjunto de funções que operam sobre a AST e a Tabela de Símbolos. Conforme detalhado em `src/tabSimbolos.c`, a construção da Tabela de Símbolos é o primeiro passo. Uma função recursiva percorre a AST e, para cada nó de declaração, insere um novo símbolo na tabela hash, registrando seu nome, tipo, escopo e tamanho.

Com a Tabela de Símbolos preenchida, é chamada a função `semantic_analysis()`, cuja implementação se encontra em `src/analise_semantica.c`. Esta função percorre a AST novamente, aplicando um conjunto de regras de verificação em cada nó. A comunicação com a Tabela de Símbolos é feita através de funções como `find_symbol()` e `count_symbol()` para validar o uso e a declaração de identificadores.

3.4.2 Implementação em Código

A lógica da análise semântica está centralizada no arquivo `src/analise_semantica.c`. A função principal, `semantic_analysis`, implementa uma travessia recursiva na AST. A cada nó visitado, ela verifica se aquele nó viola alguma regra semântica.

A seguir, a Listagem 3.3 ilustra a implementação de algumas das principais verificações semânticas.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include "../globals.h"
5
6 // Mantem o controle se a função main foi declarada
7 int main_declared = 0;
8
9 void semantic_analysis(No* root, HashTable* symbol_table) {
10     if (root == NULL) return;
11
12     // Regra: Variável declarada com o tipo inválido 'void'
13     if (strcmp(root->lexema, "void") == 0 && root->filho[0] != NULL &&
        root->kind_union.decl == var_k) {

```

```
14     fprintf(stderr, "Erro Semantico: Variavel '%s' declarada com  
15         tipo invalido 'void' na linha %d.\n", root->filho[0]->lexmema, root->  
16         linha);  
17     }  
18  
19 // Regra: Chamada de funcao nao declarada  
20 if (root->kind_union.expr == ativ_k) {  
21     Symbol* func = find_symbol(symbol_table, root->lexmema, "GLOBAL"  
22 );  
23     if (!func && strcmp(root->lexmema, "input") != 0 && strcmp(root  
24 ->lexmema, "output") != 0) {  
25         fprintf(stderr, "Erro Semantico: Funcao '%s' chamada sem  
26         declaracao na linha %d.\n", root->lexmema, root->linha);  
27     }  
28 }  
29  
30 // Regra: Atribuicao para variavel nao declarada  
31 char* scope = get_scope(root); // Funcao auxiliar para obter o  
32 escopo atual  
33 if (root->kind_union.expr == assign_k) {  
34     Symbol* var = find_symbol(symbol_table, root->filho[0]->lexmema,  
35     scope);  
36     if (!var) {  
37         Symbol* varglobal = find_symbol(symbol_table, root->filho  
38 [0]->lexmema, "GLOBAL");  
39         if (!varglobal){  
40             fprintf(stderr, "Erro: Variavel '%s' atribuida antes da  
41             declaracao na linha %d.\n", root->filho[0]->lexmema, root->linha);  
42         }  
43     }  
44 }  
45  
46 // Analisa recursivamente os nos filhos e irmaos  
47 for (int i = 0; i < NUMMAXFILHOS; i++) {  
48     semantic_analysis(root->filho[i], symbol_table);  
49 }  
50 semantic_analysis(root->irmao, symbol_table);  
51 free(scope);  
52 }  
53  
54 void check_main_function() {
```

```
51     if (!main_declared) {
52         fprintf(stderr, "Erro Semantico: Nenhuma declaracao da funcao "
53             "main' .\n");
54 }
```

Listing 3.3 – Trecho da implementação da Análise Semântica em `analise_semantica.c`

No código acima, a função verifica diversas condições. Por exemplo, para um nó de ativação de função (`ativ_k`), ela consulta a tabela de símbolos com `find_symbol()` no escopo "GLOBAL" para verificar se a função foi declarada. De forma semelhante, para uma atribuição (`assign_k`), ela busca pela variável no escopo local e, se não encontrar, no escopo global. Caso a variável não seja encontrada em nenhum dos dois, um erro semântico é reportado. Ao final de toda a análise, a função `check_main_function()` é chamada para garantir a presença de um ponto de entrada para o programa.

4 Compilador: Fase de Síntese

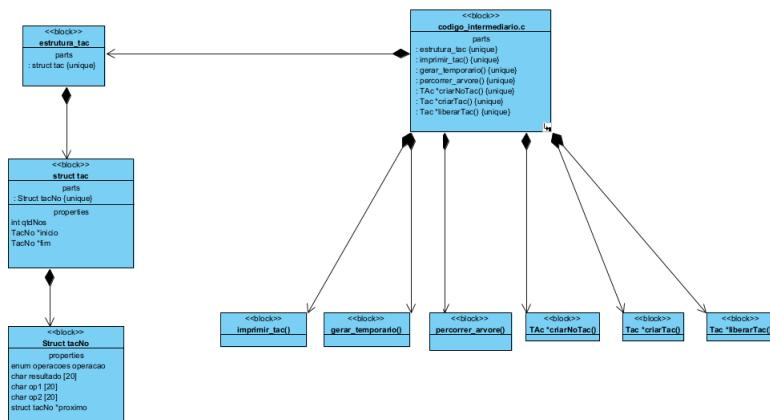
A fase de síntese, também conhecida como *back-end*, é a etapa final do processo de compilação, responsável por traduzir a representação intermediária do programa, gerada pela fase de análise, em um código executável para a arquitetura alvo. Esta fase é crucial, pois lida diretamente com as particularidades do processador, como seu conjunto de instruções e modelo de memória.

Este capítulo detalha as etapas da fase de síntese deste compilador. Primeiramente, será abordada a geração do código intermediário, um formato abstrato que facilita a otimização e a tradução para diferentes arquiteturas. Em seguida, será descrita a geração do código Assembly, o processo de tradução do código intermediário para a linguagem de montagem específica do processador ARM alvo. Finalmente, serão discutidos o gerenciamento de memória e a geração do código executável final.

4.1 Modelagem

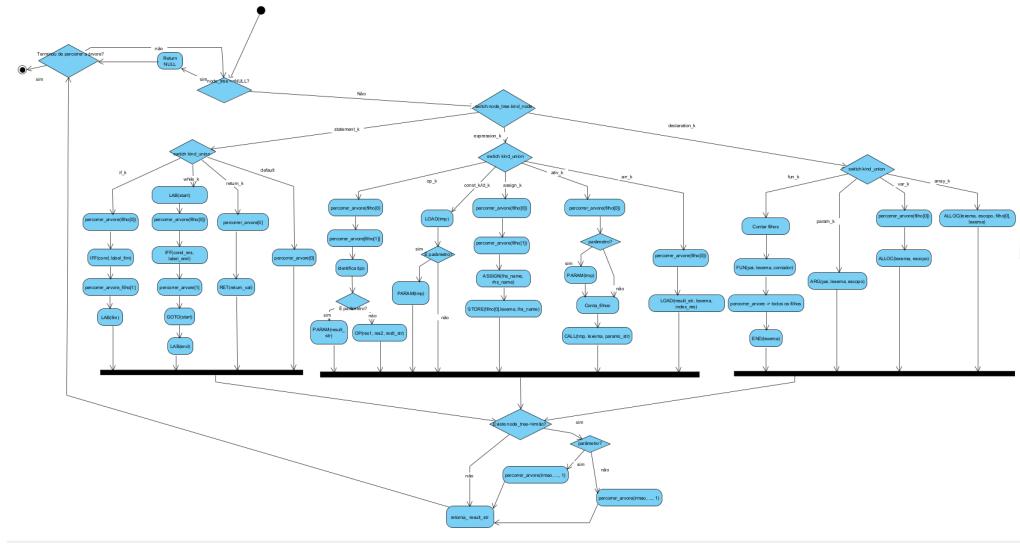
Abaixo está ilustrado o diagrama de blocos da análise sintática na figura ?? e os diagramas de atividade podem ser vistos nas figuras 15 e 16.

Figura 14 – Diagrama de Blocos - Fase de síntese



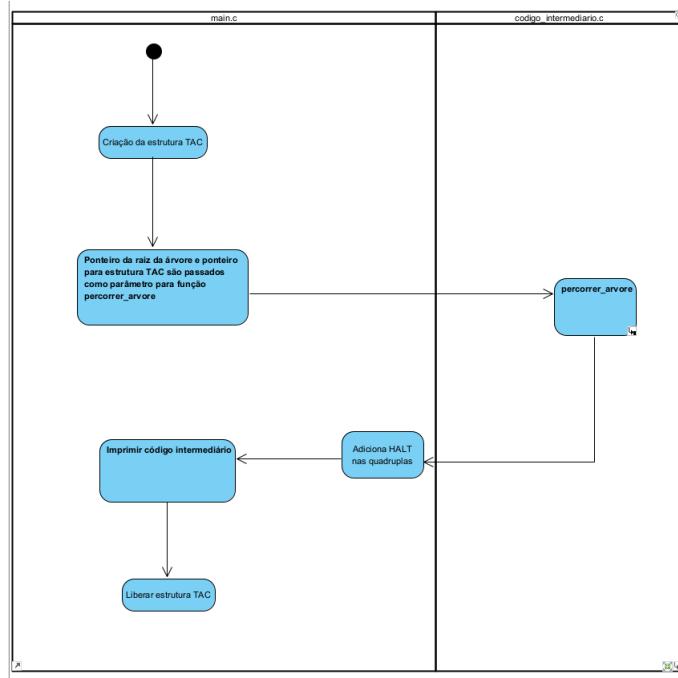
Fonte: O Autor (2025)

Figura 15 – Diagrama de Atividades - Percorrer árvore



Fonte: O Autor (2025)

Figura 16 – Diagrama de atividades - Gerador de código Intermediário



Fonte: O Autor (2025)

4.2 Geração do Código Intermediário

A geração de código intermediário é o primeiro passo da fase de síntese. Em vez de traduzir a Árvore de Sintaxe Abstrata (AST) diretamente para a linguagem de montagem, o compilador primeiro a converte para uma representação intermediária. Essa abordagem

desacopla o *front-end* (análise) do *back-end* (síntese), permitindo que o mesmo código intermediário possa, teoricamente, ser traduzido para múltiplos processadores diferentes com a reescrita apenas da etapa final de geração de código. Além disso, essa representação mais estruturada e próxima da máquina simplifica a aplicação de otimizações de código.

4.2.1 Implementação da Geração de Código Intermediário

Neste projeto, o código intermediário é gerado na forma de um Código de Três Endereços (TAC - *Three-Address Code*), especificamente utilizando uma lista de quádruplas. Cada quádrupla representa uma instrução atômica e é composta por quatro campos: uma operação, dois operandos e um resultado.

A geração do código intermediário é realizada pela função recursiva `percorrer_arvore()`, localizada em `src/codigo_intermediario.c`. Esta função atravessa a AST (pós-ordem) e, para cada nó, gera uma ou mais quádruplas que representam a semântica daquele nó. As quádruplas geradas são armazenadas em uma lista duplamente encadeada, cuja estrutura é definida em `globals.h`.

As operações possíveis para as quádruplas estão listadas na tabela 13

Tabela 13 – Instruções do Código Intermediário (TAC) e suas Funções

Instrução TAC	Função
FUN	Define o início, nome e aridade de uma função.
ARG	Declara um parâmetro formal de uma função.
LOAD	Carrega um valor (de variável, constante ou vetor) para um registrador temporário.
EQUAL	Testa a igualdade (==) entre dois operandos.
GREATER	Testa se o primeiro operando é maior que o segundo (>).
LESS	Testa se o primeiro operando é menor que o segundo (<).
IFF	Desvia o fluxo de execução para um rótulo se uma condição for falsa.
RET	Define o valor de retorno de uma função e prepara o fim de sua execução.
GOTO	Desvia incondicionalmente o fluxo de execução para um rótulo.
LAB	Define um rótulo que serve como destino para instruções de salto.
PARAM	Empilha um parâmetro na preparação para uma chamada de função.
DIV	Realiza a operação de divisão entre dois operandos.
MUL	Realiza a operação de multiplicação entre dois operandos.
SUB	Realiza a operação de subtração entre dois operandos.
CALL	Executa uma chamada de função e armazena seu valor de retorno.
END	Marca o fim do escopo e do corpo de uma função.
STORE	Armazena um valor de um registrador temporário em uma variável ou vetor.
HALT	Termina a execução do programa.
SUM	Realiza a operação de soma entre dois operandos.
ALLOC	Aloca espaço na pilha de execução para uma variável ou vetor.
ASSIGN	Realiza uma cópia direta de um valor entre registradores temporários.

4.2.2 Implementação em Código

A função `percorrer_arvore()` utiliza uma estrutura `switch` para tratar cada tipo de nó da AST (`statement_k`, `expression_k`, `declaration_k`). Para cada caso, ela gera o código intermediário correspondente.

A seguir, a Listagem 4.1 demonstra como as expressões de operação e o comando `if` são traduzidos em quádruplas.

```

1 // Função para gerar um registrador temporário
2 char* gerar_temporario() {
3     char* temp_name = malloc(12);
4     static int temp_count = 1;
5     if (temp_name) {
6         sprintf(temp_name, "t%d", temp_count++);
7     }
8     return temp_name;
9 }
```

```
10
11 // Funcao para gerar um rotulo (label)
12 char* gerar_label() {
13     static int label_count = 0;
14     char* label_name = malloc(12);
15     if (label_name) {
16         sprintf(label_name, "L%d", label_count++);
17     }
18     return label_name;
19 }
20
21 char *percorrer_arvore(No *node_tree, Tac **tac_list_ptr, ...) {
22     if (node_tree == NULL) return NULL;
23     char *result_str = NULL;
24
25     switch (node_tree->kind_node) {
26         case statement_k:
27             switch (node_tree->kind_union.stmt) {
28                 case if_k: {
29                     char *label_else = gerar_label();
30                     char *label_end = gerar_label();
31
32                     // Processa a expressao condicional
33                     char *cond_res = percorrer_arvore(node_tree->filho
34 [0], ...);
35
36                     // (IFF, cond_res, label_else, "") -> se cond for
37                     // falso, salta para else
38                     *tac_list_ptr = criarNotac(*tac_list_ptr, IFF,
39                     cond_res, label_else, "");
40
41                     // Gera codigo para o bloco 'then'
42                     percorrer_arvore(node_tree->filho[1], ...);
43                     // (GOTO, label_end, "", "") -> Salta para o fim do
44                     if
45                         *tac_list_ptr = criarNotac(*tac_list_ptr, GOTO,
46                     label_end, "", "");
47
48                     // (LAB, label_else, "", "") -> Define o rotulo do
49                     else
50                         *tac_list_ptr = criarNotac(*tac_list_ptr, LAB,
51                     label_else, "", "");
52
53                     if (node_tree->filho[2]) { // Se existir um bloco '
54                     else'
55                         percorrer_arvore(node_tree->filho[2], ...);
56                 }
57             }
```

```
49
50           // (LAB, label_end, "", "") -> Define o rotulo do
fim do if
51           *tac_list_ptr = criarNoTac(*tac_list_ptr, LAB,
label_end, "", "");
52           free(label_else);
53           free(label_end);
54           break;
55       }
56   }
57   break;
58
59
60 case expression_k:
61     switch (node_tree->kind_union.expr) {
62         case op_k: {
63             // Processa recursivamente os operandos esquerdo e
direito
64             char *res1 = percorrer_arvore(node_tree->filho[0],
...);
65             char *res2 = percorrer_arvore(node_tree->filho[1],
...);
66
67             result_str = gerar_temporario(); // Cria um novo
temporario (ex: t3)
68
69             enum operacoes op; // Mapeia o lexema para a
operacao da quadrupla
70             if (strcmp(node_tree->lexmema, "+") == 0) op = SUM;
71             else if (strcmp(node_tree->lexmema, "*") == 0) op =
MUL;
72             // ... etc ...
73
74             // (op, t3, res1, res2) -> ex: (SUM, t3, t1, t2)
75             *tac_list_ptr = criarNoTac(*tac_list_ptr, op,
result_str, res1, res2);
76
77             free(res1);
78             free(res2);
79             break;
80         }
81     }
82     break;
83 }
84 // ...
85 return result_str;
```

86 }

Listing 4.1 – Trecho da Geração de Código Intermediário em `codigo_intermediario.c`

4.3 Geração do Código Assembly

A geração de código Assembly é a penúltima etapa do processo de compilação, e a sua principal função é traduzir o código intermediário, que é abstrato e independente da máquina, para a linguagem de montagem (*Assembly*) específica do processador alvo. Esta fase é altamente dependente da arquitetura do processador, pois deve mapear as operações genéricas do código intermediário para o conjunto de instruções, modos de endereçamento e convenções de registradores do hardware de destino. O resultado desta etapa é um arquivo de texto contendo o código Assembly, que está um passo mais próximo do código executável final.

4.3.1 Implementação do Gerador de Código Assembly

No compilador desenvolvido, a geração de código Assembly é realizada por um conjunto de funções definidas no arquivo `src/gerador_assembly.c`. A função principal, `gerar_codigo_final()`, itera sobre a lista duplamente encadeada de quádruplas (TAC) e, para cada instrução intermediária, invoca a função auxiliar `traduzir_tac_para_assembly()` para gerar o código de montagem correspondente.

O processo de tradução mapeia cada operação do código intermediário para uma ou mais instruções da arquitetura ARM alvo. Por exemplo, uma quádrupla (SUM, t2, t0, t1) é convertida diretamente para a instrução ADD R2, R0, R1. Operações mais complexas, como o acesso a variáveis na pilha de execução, exigem uma sequência de instruções Assembly, geralmente envolvendo o *Frame Pointer* (FP) e o *Stack Pointer* (SP) para calcular os endereços corretos e utilizar instruções como LDR (Load) e STR (Store).

A implementação também gerencia o layout da memória, emitindo diretivas de dados (como `.data` e `.word`) para variáveis globais e diretivas de texto (`.text`) para o código das funções, garantindo que o montador final possa alocar o espaço necessário corretamente.

4.3.2 Implementação em Código

A função central `traduzir_tac_para_assembly()` utiliza uma grande estrutura `switch` para tratar cada tipo de operação do código intermediário. Dentro de cada caso, a lógica para gerar o código Assembly apropriado é executada, consultando a tabela de símbolos quando necessário para obter informações como o deslocamento (*offset*) de uma variável.

A seguir, a Listagem 4.2 ilustra a tradução de algumas das principais operações do código intermediário para a linguagem de montagem.

```

1 // Funcao que traduz uma instrucao TAC para uma ou mais instrucoes
2 // Assembly
3 void traduzir_tac_para_assembly(FILE *arquivoSaida, TacNo *tac,
4 HashTable *tabela_simbolos) {
5     // Mapeia registradores temporarios (tX) para registradores fisicos
6     // (RX)
7     char *reg_res = get_reg(tac->resultado);
8     char *reg_op1 = get_reg(tac->op1);
9     char *reg_op2 = get_reg(tac->op2);
10
11    fprintf(arquivoSaida, ";      TAC: (%s, %s, %s, %s)\n", op_nomes[tac->
12 operacao], ...);
13
14    switch (tac->operacao) {
15        case SUM:
16            // Ex: (SUM, t1, t2, t3) -> ADD R1, R2, R3
17            fprintf(arquivoSaida, "      ADD %s, %s, %s\n", reg_res,
18 reg_op1, reg_op2);
19            break;
20        case SUB:
21            // Ex: (SUB, t1, t2, t3) -> SUB R1, R2, R3
22            fprintf(arquivoSaida, "      SUB %s, %s, %s\n", reg_res,
23 reg_op1, reg_op2);
24            break;
25        case LAB:
26            // Ex: (LAB, L1, , ) -> L1:
27            fprintf(arquivoSaida, "%s:\n", tac->resultado);
28            break;
29        case GOTO:
30            // Ex: (GOTO, L1, , ) -> B L1
31            fprintf(arquivoSaida, "      B %s\n", tac->resultado);
32            break;
33        case STORE: {
34             Symbol* simbolo = find_symbol(tabela_simbolos, tac->op1,
35 escopo_atual.escopo);
36             if (simbolo == NULL) { // Procura no escopo global se nao
37                 achar_no_local
38                     simbolo = find_symbol(tabela_simbolos, tac->op1, "GLOBAL
39 ");
40             }
41             if (simbolo != NULL) {
42                 if (simbolo->id_type == var_k && strcmp(simbolo->scope,
43 "GLOBAL") != 0) {
44                     // Armazena em variavel local: STR t2, [FP, #offset]
45                     fprintf(arquivoSaida, "      ; Armazenando em variavel
46 ");
47                 }
48             }
49         }
50     }
51 }

```

```

    local '%s\n", tac->op1);
36         fprintf(arquivoSaida, "      STR %s, [FP, #%d]\n",
reg_res, simbolo->offset);
37     }
38     // ... outros casos (globais, arrays)
39 }
40 break;
41 }
42 case LOAD: {
43     if (isdigit(tac->op2[0])) { // Carrega um valor numérico
imediato
44         // Ex: (LOAD, t1, 123, ) -> MOVI R1, #123
45         fprintf(arquivoSaida, "      MOVI %s, #%"s"\n", reg_op1, tac
->op2);
46     } else { // Carrega o valor de uma variável
47         Symbol* simbolo = find_symbol(tabela_simbolos, tac->op2,
escopo_atual.escopo);
48         // ... (lógica para obter o offset e usar LDR)
49         fprintf(arquivoSaida, "      ; Acessando variável local '%
s'\n", tac->op2);
50         fprintf(arquivoSaida, "      LDR %s, [FP, #%d]\n", reg_op1
, simbolo->offset);
51     }
52     break;
53 }
54 // ... outros casos ...
55 default:
56     fprintf(arquivoSaida, "      ; AVISO: Operação TAC não
reconhecida.\n");
57     break;
58 }
59 // ...
60 }
```

Listing 4.2 – Trecho da Geração de Código Assembly em gerador_assembly.c

Na Listagem 4.2, pode-se observar a tradução direta de operações aritméticas como SUM para ADD. Para operações de memória como STORE, a lógica é mais complexa: a função primeiro busca o símbolo na tabela para encontrar seu deslocamento (*offset*) em relação ao *Frame Pointer* (FP) e, então, gera a instrução STR apropriada para armazenar o valor de um registrador temporário em uma posição na pilha. De forma similar, a instrução LOAD do código intermediário é traduzida para LDR quando carrega de uma variável ou para MOVI quando carrega um valor numérico imediato.

4.4 Geração do Código Executável (Montagem)

A etapa final do processo de compilação é a geração do código executável. No contexto deste projeto, esta fase consiste na **montagem** (*assembling*) do código Assembly. O montador é um programa tradutor que converte as instruções mnemônicas da linguagem de montagem, que são legíveis por humanos (como ADD, LDR, B), em seu formato binário correspondente, ou seja, em código de máquina que pode ser diretamente carregado e executado pelo processador alvo. O resultado é um arquivo contendo a representação binária do programa.

4.4.1 Implementação do Montador

Para realizar a montagem, foi desenvolvido um montador customizado em Python, o script `Assembler/assembler.py`. Este montador foi projetado especificamente para o conjunto de instruções da arquitetura ARM desenvolvida, cujos formatos binários estão detalhados no arquivo `Instrucoes.json`. O script implementa um processo de montagem em duas passagens (*two-pass assembler*) para lidar eficientemente com referências a rótulos (*labels*) antes de sua declaração.

O processo de montagem ocorre da seguinte forma:

1. **Primeira Passagem:** O montador lê o arquivo `.asm` de entrada uma primeira vez com o único propósito de construir uma tabela de símbolos. Nessa tabela, ele armazena cada rótulo encontrado no código (ex: `main:`, `L1:`) e o associa ao endereço da instrução correspondente. Essa etapa é fundamental para resolver os endereços de desvio (saltos), pois garante que, no momento da tradução, o endereço de cada rótulo já seja conhecido.
2. **Segunda Passagem:** O montador lê o arquivo `.asm` uma segunda vez. Agora, com a tabela de símbolos completa, ele traduz cada instrução Assembly para sua representação binária de 32 bits. Para isso, ele analisa a instrução, identifica seus operandos (registradores, valores imediatos, rótulos) e os insere nos campos corretos da palavra de instrução, de acordo com o formato definido para aquela operação.

O passo final do processo de compilação, conforme demonstrado no arquivo `script.sh`, é a execução deste montador, que lê o código Assembly gerado e produz o arquivo binário final.

4.4.2 Implementação em Código

A lógica central do montador está na função `translate_instruction()`, que recebe uma instrução Assembly já analisada e a converte em uma string binária de 32 bits.

A função utiliza uma série de condicionais para identificar o mnemônico da instrução e, em seguida, monta a palavra binária de acordo com o formato especificado para a arquitetura do processador.

A seguir, a Listagem 4.3 demonstra como as instruções de processamento de dados (como ADD) e as instruções de desvio (como B) são traduzidas.

```

1 class Assembler:
2     def __init__(self):
3         self.special_registers = {
4             "Rret": 24, "Rad": 25, "SP": 26, "FP": 27, "Rin": 28,
5             "Rout": 29, "CPSR": 30, "Rlink": 31
6         }
7         self.symbol_table = {}
8
9     def _get_operand_binary(self, operand, bits=5):
10        # Converte um operando (registrador ou imediato) para binário
11        # ... (lógica para converter registradores e imediatos)
12
13    # Primeira passagem para construir a tabela de símbolos (
14    symbol_table)
15    def first_pass(self, lines):
16        address = 0
17        for line in lines:
18            parsed = self.parse_line(line)
19            if not parsed:
20                continue
21            if parsed['type'] == 'label':
22                self.symbol_table[parsed['name']] = address
23            elif parsed['type'] == 'instruction':
24                address += 1
25
26    # Segunda passagem para traduzir
27    def translate_instruction(self, parsed_line, current_address):
28        mnemonic = parsed_line['mnemonic']
29        operands = parsed_line['operands']
30
31        if mnemonic == "ADD":
32            # Formato: Cond[4] 00[2] 0[1] Opcode[4] S[1] Rn[5] Rd[5] Rm
33            [5] X[5]
34            final_binary = [
35                '1110', '00', '0', '0101', '0',
36                self._get_operand_binary(operands[1]), # rn
37                self._get_operand_binary(operands[0]), # rd
38                self._get_operand_binary(operands[2]), # rm
39                '00000'
40            ]
41        elif mnemonic == "B" or mnemonic == "BL":

```

```

40         target_label = operands[0]
41         # Usa a tabela de simbolos para obter o endereço do rotulo
42         target_address = self.symbol_table[target_label]
43
44         link_bit = '1' if mnemonic == 'BL' else '0'
45         # Formato: Cond[4] 10[2] 1[1] L[1] Imm[24]
46         final_binary = [
47             '1110', '10', '1', link_bit,
48             self._get_operand_binary(target_address, 24)
49         ]
50         # ... (outras instrucoes)
51
52     return "".join(final_binary)

```

Listing 4.3 – Trecho do processo de tradução no montador assembler.py

Na Listagem 4.3, para uma instrução ADD, o script simplesmente preenche os campos do formato binário com os códigos correspondentes à operação e os números dos registradores. Para uma instrução de desvio como B, o processo é mais elaborado: o montador utiliza o nome do rótulo para consultar seu endereço na `symbol_table` (preenchida na primeira passagem) e, em seguida, insere esse endereço no campo imediato de 24 bits da instrução. Este mecanismo de duas passagens é o que permite a resolução de saltos para qualquer ponto do programa.

4.5 Gerenciamento de Memória e Chamada de Funções

A tradução de chamadas de função para a linguagem de montagem exige um mecanismo robusto para gerenciar o estado da execução, os parâmetros, as variáveis locais e o endereço de retorno. Este compilador implementa esse gerenciamento através de uma estrutura de dados fundamental na memória: a **pilha de execução** (*call stack*). Cada vez que uma função é chamada, um novo bloco de memória, conhecido como **Registro de Ativação** ou **Stack Frame**, é alocado no topo da pilha.

4.5.1 O Registro de Ativação (Stack Frame)

O Registro de Ativação contém todas as informações necessárias para a execução de uma única chamada de função. A estrutura é gerenciada por registradores especiais do processador, conforme definido na arquitetura:

- **Stack Pointer (SP):** Aponta sempre para o topo livre da pilha. É usado para alocar espaço.
- **Frame Pointer (FP):** Aponta para a base do registro de ativação da função atual. Ele serve como uma referência estável para acessar parâmetros e variáveis locais.

- **Registrador de Link (Rlink):** Utilizado pela instrução BL para salvar o endereço de retorno.
- **Registrador de Retorno (Rret):** Utilizado para armazenar o valor de retorno de uma função.

O ciclo de vida de um registro de ativação é controlado pelo código gerado para a chamada, o prólogo e o epílogo da função.

4.5.1.0.1 Código do Chamador.

Antes de chamar uma função, a rotina que faz a chamada (*caller*) é responsável por preparar a pilha. A instrução PARAM do código intermediário é traduzida para as seguintes instruções Assembly, que empurram cada parâmetro para a pilha:

1. STR <reg_param>, [SP, #0] - Armazena o valor do parâmetro no topo da pilha.
2. ADDI SP, SP, #1 - Incrementa o ponteiro da pilha para alocar espaço.

Após empilhar todos os parâmetros, o chamador configura a chamada em si, que, conforme visto no código do gerador, salva o FP atual, atualiza o FP para o novo frame e finalmente executa a chamada com BL.

4.5.1.0.2 Prólogo da Função.

No início de cada função (o *callee*), o código Assembly gerado executa um prólogo para finalizar a configuração do seu registro de ativação. A instrução FUN do TAC gera o seguinte código:

1. ADDI Rlink, Rlink, #1 - Incrementa o endereço de retorno salvo pelo BL.
2. STR Rlink, [SP, #0] - Salva o endereço de retorno na pilha.
3. ADDI SP, SP, #1 - Aloca espaço para o endereço de retorno salvo.

Além disso, a instrução ALLOC é usada para alocar espaço para as variáveis locais da função, simplesmente incrementando o SP com ADDI SP, SP, #<tamanho>.

4.5.1.0.3 Epílogo da Função.

Ao final da função, antes de retornar, um epílogo é executado para restaurar o estado da função chamadora. A instrução RET do código intermediário dispara a geração do seguinte código:

1. `MOV Rret, <reg_retorno>` - Armazena o valor de retorno no registrador `Rret`.
2. `LDR Rlink, [FP, #1]` - Restaura o endereço de retorno da pilha para o `Rlink`.
3. `MOV SP, FP` - Libera todo o espaço alocado para o frame atual, retornando o `SP` para a base.
4. `LDR FP, [FP, #0]` - Restaura o Frame Pointer da função chamadora.
5. `B Rlink` - Salta de volta para o código do chamador.

4.5.2 Passagem de Parâmetros e Vetores

Conforme descrito, a passagem de parâmetros é feita através da pilha. Dentro da função, esses parâmetros são acessados através de deslocamentos negativos em relação ao `FP`, enquanto as variáveis locais são acessadas com deslocamentos positivos.

4.5.2.0.1 Passagem de Vetores.

Na linguagem C-, vetores são passados por **referência**. O compilador gera código que coloca na pilha o **endereço base** do vetor (o endereço de seu primeiro elemento), em vez de uma cópia de todos os seus elementos. A lógica de acesso a um elemento de um vetor passado como parâmetro (`param_array_k`) é implementada da seguinte forma em `src/gerador_assembly.c`:

1. `MOV Rad, FP` - Usa o `FP` como base.
2. `SUBI Rad, Rad, #<offset_param>` - Encontra a posição do parâmetro na pilha (que contém o endereço do vetor original).
3. `LDR Rad, [Rad, #0]` - Carrega o endereço base do vetor para o registrador `Rad`.
4. `ADD Rad, Rad, <reg_indice>` - Soma o endereço base com o registrador que contém o índice desejado.
5. `LDR <reg_destino>, [Rad, #0]` - Carrega o valor final do elemento do vetor no registrador de destino.

Esta abordagem é eficiente, pois evita a cópia de grandes volumes de dados.

4.5.3 Suporte à Recursividade

A utilização de uma pilha com registros de ativação bem definidos é o que permite, de forma natural, a implementação de **funções recursivas**. Quando uma função chama a si mesma, um novo registro de ativação é empilhado sobre o anterior. Cada instância da chamada recursiva possui seu próprio conjunto isolado de parâmetros, variáveis locais e endereço de retorno. O prólogo e o epílogo garantem que, ao final de cada chamada, o estado da chamada anterior seja perfeitamente restaurado, permitindo que a computação continue do ponto onde parou. Sem essa estrutura de pilha, a recursividade seria impossível de implementar, pois cada chamada sobrescreveria os dados da anterior.

5 Exemplos de Uso

Neste capítulo, serão apresentados exemplos práticos de compilação para demonstrar a funcionalidade do compilador desenvolvido. Para cada exemplo, serão exibidos o código-fonte original em C-, o código intermediário em formato de quádruplas, o código Assembly gerado para o processador ARM alvo e, quando aplicável, o código executável binário final. Adicionalmente, será detalhada a correspondência entre as diferentes etapas da tradução para ilustrar o processo de compilação.

5.1 Exemplo 1: Programa básico

O primeiro exemplo é um programa fundamental que testa a capacidade do compilador em lidar com entrada e saída de dados, declaração de variáveis locais e, principalmente, a geração de código para uma estrutura de controle de fluxo condicional `if-else`. Este caso de uso serve como base para validar a correta tradução das operações de comparação e desvio.

5.1.1 Código-Fonte

A seguir, o Listagem 5.1 apresenta o código-fonte em C-. O programa lê dois números inteiros e exibe o maior entre eles.

```

1 void main() {
2     int x;
3     int y;
4     x = input();
5     y = input();
6     if(x > y){
7         output(x);
8     }
9     else {
10        output(y);
11    }
12 }
```

Listing 5.1 – Código-fonte com estrutura condicional (`very_basic.c`)

5.1.2 Código Intermediário Gerado

```

1 (1) (FUN, void, main, 5)
2 (2) (ALLOC, x, main, )
3 (3) (ALLOC, y, main, )
```

```

4 (4) (CALL, t1, input, 0)
5 (5) (STORE, x, , t1)
6 (6) (CALL, t2, input, 0)
7 (7) (STORE, y, , t2)
8 (8) (LOAD, t3, x, )
9 (9) (LOAD, t4, y, )
10 (10) (GREATER, t5, t3, t4)
11 (11) (IFF, t5, L0, )
12 (12) (LOAD, t6, x, )
13 (13) (PARAM, t6, var, )
14 (14) (CALL, t7, output, 1)
15 (15) (GOTO, L1, , )
16 (16) (LAB, L0, , )
17 (17) (LOAD, t8, y, )
18 (18) (PARAM, t8, var, )
19 (19) (CALL, t9, output, 1)
20 (20) (LAB, L1, , )
21 (21) (END, main, , )
22 (22) (HALT, , , )
23 ; Fim da Lista TAC

```

Listing 5.2 – Código intermediário para very_basic.c

5.1.3 Código Assembly Gerado

```

1 .data
2
3 .text
4     NOP
5     MOVI SP, #0
6     MOVI FP, #0
7     B main
8 .main
9     MOV FP, SP
10    ADDI SP, SP, #1
11    ADDI SP, SP, #1
12    IN
13    MOV R1, Rin
14    ; Armazenando em variavel local 'x'
15    STR R1, [FP, #0]
16    IN
17    MOV R2, Rin
18    ; Armazenando em variavel local 'y'
19    STR R2, [FP, #1]
20    ; Acessando variavel local 'x'
21    LDR R3, [FP, #0]
22    ; Acessando variavel local 'y'

```

```

23   LDR R4, [FP, #1]
24   CMP R3, R4
25   BLE L0
26   ; Acessando variavel local 'x'
27   LDR R6, [FP, #0]
28   MOV Rout, R6
29   OUT
30   B L1
31 L0:
32   ; Acessando variavel local 'y'
33   LDR R8, [FP, #1]
34   MOV Rout, R8
35   OUT
36 L1:
37   FINISH
38
39 ; Fim do programa

```

Listing 5.3 – Código Assembly gerado para `very_basic.c`

5.1.4 Correspondência entre as Etapas

A correspondência entre as etapas das compilação pode ser vista na tabela 14.

5.2 Exemplo 1: GCD

O segundo exemplo é a implementação do algoritmo de Euclides para o cálculo do MDC, uma função inherentemente recursiva. Este exemplo é ideal para validar a correta implementação do gerenciamento da pilha de chamadas, incluindo a declaração de variáveis globais, a passagem de parâmetros e o tratamento de retornos de função.

5.2.1 Código-Fonte

A seguir, o Listagem 5.4 apresenta o código-fonte original em C- que foi utilizado como entrada para o compilador.

```

1 int teste;
2 int vetor[5];
3 int gcd (int u, int v)
4 {
5     if (v == 0) return u ;
6     else return gcd(v,u-u/v*v);
7 }
8
9 void main (void)

```

Tabela 14 – Correspondência entre as etapas de compilação: Programa básico.

Código C-	Código Intermediário (TAC)	Código Assembly Gerado
void main() {	(FUN, 5, void, main)	B main .main MOV FP, SP
int x;	(ALLOC, , x, main)	ADDI SP, SP, #1
int y;	(ALLOC, , y, main)	ADDI SP, SP, #1
x = input();	(CALL, 0, t1, input) (STORE, t1, x,)	IN MOV R1, Rin ; Armazenando em variavel local 'x' STR R1, [FP, #0]
y = input();	(CALL, 0, t2, input) (STORE, t2, y,)	IN MOV R2, Rin ; Armazenando em variavel local 'y' STR R2, [FP, #1]
if(x >y)	(LOAD, , t3, x) (LOAD, , t4, y) (GREATER, t4, t5, t3) (IFF, , t5, L0)	; Acessando variavel local 'x' LDR R3, [FP, #0] ; Acessando variavel local 'y' LDR R4, [FP, #1] CMP R3, R4 BLE L0
{ output(x); }	(LOAD, , t6, x) (PARAM, , t6, var) (CALL, 1, t7, output) (GOTO, , L1,)	; Acessando variavel local 'x' LDR R6, [FP, #0] MOV Rout, R6 OUT B L1
else { output(y); }	(LAB, , L0,) (LOAD, , t8, y) (PARAM, , t8, var) (CALL, 1, t9, output)	L0: ; Acessando variavel local 'y' LDR R8, [FP, #1] MOV Rout, R8 OUT
} (fim do if)	(LAB, , L1,)	L1:
} (fim da main)	(END, , main,) (HALT, , ,)	FINISH

```

10 { int x; int y;
11
12     x = input(); y=input();
13     output(gcd(x,y));
14 }
```

Listing 5.4 – Código-fonte para o cálculo do MDC (gcd.c)

5.2.2 Código Intermediário Gerado

Após as fases de análise, o compilador gera o seguinte Código de Três Endereços (TAC). Note como a chamada recursiva, a expressão aritmética e as declarações globais são decompostas em operações atômicas.

```

1 (1) (ALLOC, teste, global, )
2 (2) (ALLOC, vetor, global, 5)
3 (3) (FUN, int, gcd, 2)
4 (4) (ARG, int, u, gcd)
5 (5) (ARG, int, v, gcd)
6 (6) (LOAD, t1, v, )
7 (7) (LOAD, t2, 0, )
8 (8) (EQUAL, t3, t1, t2)
9 (9) (IFF, t3, L0, )
10 (10) (LOAD, t4, u, )
11 (11) (RET, t4, , )
12 (12) (GOTO, L1, , )
13 (13) (LAB, L0, , )
14 (14) (LOAD, t5, v, )
15 (15) (PARAM, t5, param, )
16 (16) (LOAD, t6, u, )
17 (17) (LOAD, t7, u, )
18 (18) (LOAD, t8, v, )
19 (19) (DIV, t9, t7, t8)
20 (20) (LOAD, t10, v, )
21 (21) (MUL, t11, t9, t10)
22 (22) (SUB, t12, t6, t11)
23 (23) (PARAM, t12, , )
24 (24) (CALL, t13, gcd, 2)
25 (25) (RET, t13, , )
26 (26) (LAB, L1, , )
27 (27) (END, gcd, , )
28 (28) (FUN, void, main, 1)
29 (29) (ALLOC, x, main, )
30 (30) (ALLOC, y, main, )
31 (31) (CALL, t14, input, 0)
32 (32) (STORE, x, , t14)
33 (33) (CALL, t15, input, 0)
```

```

34 (34) (STORE, y, , t15)
35 (35) (LOAD, t16, x, )
36 (36) (PARAM, t16, var, )
37 (37) (LOAD, t17, y, )
38 (38) (PARAM, t17, var, )
39 (39) (CALL, t18, gcd, 2)
40 (40) (CALL, t19, output, 1)
41 (41) (END, main, , )
42 (42) (HALT, , , )
43 ; Fim da Lista TAC

```

Listing 5.5 – Código intermediário para gcd.c

5.2.3 Código Assembly Gerado

Finalmente, o código intermediário é traduzido para a linguagem de montagem específica do processador ARM alvo. O Listagem 5.6 mostra o resultado final, detalhando o gerenciamento da pilha, o acesso a variáveis globais e locais, e a lógica de desvio.

```

1 ; Arquivo Assembly gerado pelo compilador C-
2
3 .data
4     teste: .word 0
5     vetor: .space 5
6
7 .text
8     NOP
9     MOVI SP, #0
10    MOVI FP, #0
11    ADDI SP, SP, #1
12    ADDI SP, SP, #5
13    B main
14 .gcd
15    ADDI Rlink, Rlink, #1
16    STR Rlink [SP #0]
17    ADDI SP, SP, #1
18    ; Acessando param'v'
19    MOV Rad, FP
20    SUBI Rad, Rad, #1
21    LDR R1, [Rad, #0]
22    MOVI R2, #0
23    CMP R1, R2
24    BNE LO
25    ; Acessando param'u'
26    MOV Rad, FP
27    SUBI Rad, Rad, #2
28    LDR R4, [Rad, #0]

```

```
29     MOV Rret, R4
30     LDR Rlink [FP #1]
31     MOV SP, FP
32     LDR FP [FP #0]
33     B Rlink
34     B L1
35 L0:
36     ; Acessando param'v'
37     MOV Rad, FP
38     SUBI Rad, Rad, #1
39     LDR R5, [Rad, #0]
40     STR R5 [SP, #0]
41     ADDI SP, SP, #1
42     ; Acessando param'u'
43     MOV Rad, FP
44     SUBI Rad, Rad, #2
45     LDR R6, [Rad, #0]
46     ; Acessando param'u'
47     MOV Rad, FP
48     SUBI Rad, Rad, #2
49     LDR R7, [Rad, #0]
50     ; Acessando param'v'
51     MOV Rad, FP
52     SUBI Rad, Rad, #1
53     LDR R8, [Rad, #0]
54     UDIV R9, R7, R8
55     ; Acessando param'v'
56     MOV Rad, FP
57     SUBI Rad, Rad, #1
58     LDR R10, [Rad, #0]
59     MUL R11, R9, R10
60     SUB R12, R6, R11
61     STR R12 [SP, #0]
62     ADDI SP, SP, #1
63     STR FP [SP, #0]
64     MOV FP, SP
65     ADDI SP, SP, #1
66     BL gcd
67     MOV R13, Rret
68     MOV Rret, R13
69     LDR Rlink [FP #1]
70     MOV SP, FP
71     LDR FP [FP #0]
72     B Rlink
73 L1:
74 .main
75     MOV FP, SP
```

```

76    ADDI SP, SP, #1
77    ADDI SP, SP, #1
78    IN
79    MOV R14, Rin
80    ; Armazenando em variavel local 'x'
81    STR R14, [FP, #0]
82    IN
83    MOV R15, Rin
84    ; Armazenando em variavel local 'y'
85    STR R15, [FP, #1]
86    ; Acessando variavel local 'x'
87    LDR R16, [FP, #0]
88    STR R16 [SP, #0]
89    ADDI SP, SP, #1
90    ; Acessando variavel local 'y'
91    LDR R17, [FP, #1]
92    STR R17 [SP, #0]
93    ADDI SP, SP, #1
94    STR FP [SP, #0]
95    MOV FP, SP
96    ADDI SP, SP, #1
97    BL gcd
98    MOV R18, Rret
99    MOV Rout, R18
100   OUT
101   FINISH
102
103 ; Fim do programa

```

Listing 5.6 – Código Assembly gerado para gcd.c

5.2.4 Código Executável (Binário)

Após a montagem do código Assembly, o resultado final é um arquivo de texto contendo a representação binária de 32 bits para cada instrução. Este é o código de máquina que pode ser carregado diretamente na memória de instruções do processador para execução.

```

1 11101100000000000000000000000000
2 11100011101000000110100000000000
3 11100011101000000110110000000000
4 11100010101011010110100000000001
5 11100010101011010110100000000101
6 1110101000000000000000000000000011000
7 1110001010101111111100000000000001
8 11100110011010111110000000000000000
9 111000101010110101000000000000001

```

```
10 111000111010110111100100000000000
11 11100010011011001110010000000001
12 11100110111001000010000000000000
13 111000111010000000001000000000000
14 11100001010100001000000001000000
15 000110100000000000000000000000011000
16 111000111010110111100100000000000
17 11100010011011001110010000000010
18 111001101110010010000000000000000
19 11100011101000100110000000000000
20 111001101110111110000000000001
21 111000111010110111101000000000000
22 111001101110111101100000000000000
23 11101000111100000000000000000000
24 1110101000000000000000000000000000000011000
25 111000111010110111100100000000000
26 1110001001101100111001000000000001
27 1110011011100100101000000000000000
28 111001100110100010100000000000000
29 1110001010101101011010000000000001
30 111000111010110111100100000000000
31 111000100110110011100100000000010
32 111001101110010011000000000000000
33 11100011101011011110010000000000
34 11100010011011001110010000000010
35 11100110111001001100000000000000
36 111000111010110111100100000000000
37 111000100110110011100100000000001
38 1110011011100101000000000000000000
39 11100001111000111010010100000000
40 111000111010110111100100000000000
41 111000100110110011100100000000001
42 111001101110010101000000000000000
43 11100001110001001010110101000000
44 1110000001100011001100010110000
45 111001100110100110000000000000000
46 111000101010110101101000000000001
47 111001100110101101100000000000000
48 111000111010110101101100000000000
49 111000101010110101101000000000001
50 11101011000000000000000000000000110
51 111000111010110000110100000000000
52 111000111010011011100000000000000
53 111001101110111111000000000000001
54 1110001110101101111010000000000000
55 1110011011101111011000000000000000
56 111010001111000000000000000000000000000000
```

```

57 111000111010110101101100000000000
58 111000101010110101101000000000001
59 111000101010110101101000000000001
60 11101100100000000000000000000000000
61 1110001110101110001110000000000000
62 11100110011011011100000000000000000
63 111011001000000000000000000000000000
64 111000111010111000111100000000000
65 1110011001101101110000000000000001
66 1110011011101110000000000000000000
67 1110011001101010000000000000000000
68 111000101010110101101000000000001
69 11100110111011100010000000000001
70 11100110011010100010000000000000
71 111000101010110101101000000000001
72 111001100110101101100000000000000
73 111000111010110101101100000000000
74 111000101010110101101000000000001
75 1110101100000000000000000000000000010
76 111000111010110001001000000000000
77 111000111010100101110100000000000
78 111011010000000000000000000000000000
79 111011011000000000000000000000000000000

```

Listing 5.7 – Código binário gerado para gcd.c

5.2.5 Correspondência entre códigos

A tabela 15 mostra a relação direta entre o código fonte e o código intermediário gerado pelo compilador.

Tabela 15 – Tabela de Correspondência para o Exemplo de Ordenação

Código C-	Código Intermediário (TAC)	Código Assembly Gerado
1 <code>int vet[5];</code>	1 (ALLOC , <code>vet</code> , <code>global</code> , 5)	.data 2 <code>vet:</code> .space 5 3 .text 4 ADDI SP, SP, #5

Continua na próxima página

Tabela 15 (continuação)

Código C-	Código Intermediário (TAC)	Código Assembly G gerado
<pre> 1 int minloc(...) { 2 int i; int x; 3 int k; </pre>	<pre> 1 (FUN, int, minloc, 3) 2 ... 3 (ALLOC, i, minloc,) 4 (ALLOC, x, minloc,) 5 (ALLOC, k, minloc,) </pre>	<pre> 1 .minloc 2 ; Prologo 3 ADDI Rlink, Rlink, 4 #1 5 STR Rlink [SP, #0] 6 ADDI SP, SP, #1 7 ; Alocacao de locais 8 ADDI SP, SP, #1 9 ADDI SP, SP, #1 10 ADDI SP, SP, #1 </pre>
<pre> 1 k = low; </pre>	<pre> 1 (LOAD, t1, low,) 2 (STORE, k, , t1) </pre>	<pre> 1 ; Acessa param 'low' 2 MOV Rad, FP 3 SUBI Rad, Rad, #2 4 LDR R1, [Rad, #0] 5 ; Armazena em 'k' 6 STR R1, [FP, #4] </pre>
<pre> 1 x = a[low]; </pre>	<pre> 1 (LOAD, t2, low,) 2 (LOAD, t3, a, t2) 3 (STORE, x, , t3) </pre>	<pre> 1 ; Acessa param 'low' 2 MOV Rad, FP 3 SUBI Rad, Rad, #2 4 LDR R2, [Rad, #0] 5 ; Acessa array 'a' 6 MOV Rad, FP 7 SUBI Rad, Rad, #3 8 LDR Rad, [Rad, #0] 9 ADD Rad, Rad, R2 10 LDR R3, [Rad, #0] 11 ; Armazena em 'x' 12 STR R3, [FP, #3] </pre>
<pre> 1 while (i < high) </pre>	<pre> 1 (LAB, L0, ,) 2 (LOAD, t7, i,) 3 (LOAD, t8, high,) 4 (LESS, t9, t7, t8) 5 (IFF, t9, L1,) </pre>	<pre> 1 L0: 2 LDR R7, [FP, #2] 3 MOV Rad, FP 4 SUBI Rad, Rad, #1 5 LDR R8, [Rad, #0] 6 CMP R7, R8 7 BGE L1 </pre>

Continua na próxima página

Tabela 15 (continuação)

Código C-	Código Intermediário (TAC)	Código Assembly G gerado
<pre> 1 t = a[k]; 2 a[k] = a[i]; 3 a[i] = t; </pre>	<pre> 1 (LOAD, t8, k,) 2 (LOAD, t9, a, t8) 3 (STORE, t, , t9) 4 (LOAD, t10, i,) 5 (LOAD, t11, a, t10) 6 (LOAD, t12, k,) 7 (STORE, a, t12, t11) 8 (LOAD, t13, t,) 9 (LOAD, t14, i,) 10 (STORE, a, t14, t13) </pre>	<pre> 1 ; t = a[k] 2 LDR R8, [FP, #3] 3 ... (acessa a[k]) 4 LDR R9, [Rad, #0] 5 STR R9, [FP, #4] 6 ; a[k] = a[i] 7 ... (acessa a[i]) 8 LDR R11, [Rad, #0] 9 ... (acessa a[k]) 10 STR R11, [Rad, #0] 11 ; a[i] = t 12 LDR R13, [FP, #4] 13 ... (acessa a[i]) 14 STR R13, [Rad, #0] </pre>
<pre> 1 } // Fim da main </pre>	<pre> 1 (LAB, L9, ,) 2 (END, main, ,) 3 (HALT, , ,) </pre>	<pre> 1 L9: 2 FINISH </pre>

O código abaixo mostra a correspondência entre a geração do código intermediário e código assembly para o algoritmo do gcd.c.

```

1 .data
2     teste: .word 0
3     vetor: .space 5
4
5 .text
6     NOP
7     MOVI SP, #0
8     MOVI FP, #0
9 ;     TAC: (ALLOC, , teste, global)
10    ADDI SP, SP, #1
11 ;     TAC: (ALLOC, 5, vetor, global)
12    ADDI SP, SP, #5
13 ;     TAC: (FUN, 2, int, gcd)
14     B main
15 .gcd
16     ADDI Rlink, Rlink, #1
17     STR Rlink [SP #0]
18     ADDI SP, SP, #1
19 ;     TAC: (ARG, gcd, int, u)
20 ;     TAC: (ARG, gcd, int, v)
21 ;     TAC: (LOAD, , t1, v)

```

```
22 ; Acessando param'v
23 MOV Rad, FP
24 SUBI Rad, Rad, #1
25 LDR R1, [Rad, #0]
26 ; TAC: (LOAD, , t2, 0)
27 MOVI R2, #0
28 ; TAC: (EQUAL, t2, t3, t1)
29 CMP R1, R2
30 BNE L0
31 ; TAC: (IFF, , t3, L0)
32 ; TAC: (LOAD, , t4, u)
33 ; Acessando param'u
34 MOV Rad, FP
35 SUBI Rad, Rad, #2
36 LDR R4, [Rad, #0]
37 ; TAC: (RET, , t4, )
38 MOV Rret, R4
39 LDR Rlink [FP #1]
40 MOV SP, FP
41 LDR FP [FP #0]
42 B Rlink
43 ; TAC: (GOTO, , L1, )
44 B L1
45 ; TAC: (LAB, , L0, )
46 L0:
47 ; TAC: (LOAD, , t5, v)
48 ; Acessando param'v
49 MOV Rad, FP
50 SUBI Rad, Rad, #1
51 LDR R5, [Rad, #0]
52 ; TAC: (PARAM, , t5, param)
53 STR R5 [SP, #0]
54 ADDI SP, SP, #1
55 ; TAC: (LOAD, , t6, u)
56 ; Acessando param'u
57 MOV Rad, FP
58 SUBI Rad, Rad, #2
59 LDR R6, [Rad, #0]
60 ; TAC: (LOAD, , t7, u)
61 ; Acessando param'u
62 MOV Rad, FP
63 SUBI Rad, Rad, #2
64 LDR R7, [Rad, #0]
65 ; TAC: (LOAD, , t8, v)
66 ; Acessando param'v
67 MOV Rad, FP
68 SUBI Rad, Rad, #1
```

```
69     LDR R8, [Rad, #0]
70 ; TAC: (DIV, t8, t9, t7)
71     UDIV R9, R7, R8
72 ; TAC: (LOAD, , t10, v)
73 ; Acessando param'v'
74     MOV Rad, FP
75     SUBI Rad, Rad, #1
76     LDR R10, [Rad, #0]
77 ; TAC: (MUL, t10, t11, t9)
78     MUL R11, R9, R10
79 ; TAC: (SUB, t11, t12, t6)
80     SUB R12, R6, R11
81 ; TAC: (PARAM, , t12, )
82     STR R12 [SP, #0]
83     ADDI SP, SP, #1
84 ; TAC: (CALL, 2, t13, gcd)
85     STR FP [SP, #0]
86     MOV FP, SP
87     ADDI SP, SP, #1
88     BL gcd
89     MOV R13, Rret
90 ; TAC: (RET, , t13, )
91     MOV Rret, R13
92     LDR Rlink [FP #1]
93     MOV SP, FP
94     LDR FP [FP #0]
95     B Rlink
96 ; TAC: (LAB, , L1, )
97 L1:
98 ; TAC: (END, , gcd, )
99 ; TAC: (FUN, 1, void, main)
100 .main
101     MOV FP, SP
102 ; TAC: (ALLOC, , x, main)
103     ADDI SP, SP, #1
104 ; TAC: (ALLOC, , y, main)
105     ADDI SP, SP, #1
106 ; TAC: (CALL, 0, t14, input)
107     IN
108     MOV R14, Rin
109 ; TAC: (STORE, t14, x, )
110 ; Armazenando em variavel local 'x'
111     STR R14, [FP, #0]
112 ; TAC: (CALL, 0, t15, input)
113     IN
114     MOV R15, Rin
115 ; TAC: (STORE, t15, y, )
```

```

116      ; Armazenando em variavel local 'y'
117      STR R15, [FP, #1]
118 ; TAC: (LOAD, , t16, x)
119      ; Acessando variavel local 'x'
120      LDR R16, [FP, #0]
121 ; TAC: (PARAM, , t16, var)
122      STR R16 [SP, #0]
123      ADDI SP, SP, #1
124 ; TAC: (LOAD, , t17, y)
125      ; Acessando variavel local 'y'
126      LDR R17, [FP, #1]
127 ; TAC: (PARAM, , t17, var)
128      STR R17 [SP, #0]
129      ADDI SP, SP, #1
130 ; TAC: (CALL, 2, t18, gcd)
131      STR FP [SP, #0]
132      MOV FP, SP
133      ADDI SP, SP, #1
134      BL gcd
135      MOV R18, Rret
136 ; TAC: (CALL, 1, t19, output)
137      MOV Rout, R18
138      OUT
139 ; TAC: (END, , main, )
140 ; TAC: (HALT, , , )
141      FINISH

```

Listing 5.8 – Correspondência do código fonte intermediário para assembly GCD.c

5.3 Exemplo 3: Sort

O terceiro exemplo demonstra a capacidade do compilador em lidar com estruturas de dados mais complexas, especificamente vetores (arrays), além de múltiplas chamadas de função aninhadas e laços de repetição. Foi utilizado um algoritmo de ordenação por seleção (*Selection Sort*), que encontra o menor elemento do vetor e o posiciona no início, repetindo o processo para o restante do vetor. Este caso de uso é ideal para validar o acesso à memória e a passagem de vetores como parâmetro.

5.3.1 Código-Fonte

A Listagem 5.9 apresenta o código-fonte em C- para o algoritmo de ordenação.

```

1 int vet[5];
2
3 int minloc( int a[], int low, int high )
4 {

```

```
5  int i;
6  int x;
7  int k;
8  k = low;
9  x = a[low];
10 i = low + 1;
11 while (i < high){
12     if (a[i] < x){
13         x = a[i];
14         k = i;
15     }
16     i = i + 1;
17 }
18 return k;
19 }
20
21 void sort( int a[], int low, int high)
22 { int i; int k;
23 i = low;
24 while (i < high-1){
25     int t;
26     k = minloc(a,i,high);
27     t = a[k];
28     a[k] = a[i];
29     a[i] = t;
30     i = i + 1;
31 }
32 }
33
34 void main(void)
35 {
36     int i;
37     i = 0;
38     while (i < 5){
39         vet[i] = input();
40         i = i + 1;
41     }
42     sort(vet,0,5);
43     i = 0;
44     while (i < 5){
45         output(vet[i]);
46         i = i + 1;
47     }
48 }
```

Listing 5.9 – Código-fonte para o algoritmo de ordenação (`sort.c`)

5.3.2 Código Intermediário Gerado

A tradução para o código intermediário reflete a decomposição dos laços `while` em saltos condicionais e rótulos (`LAB`, `GOTO`, `IFF`). O acesso aos elementos do vetor, como `a[k]`, é convertido em instruções `LOAD` e `STORE` que utilizam um endereço base e um registrador de deslocamento.

```

1 (1) (ALLOC, vet, global, 5)
2 (2) (FUN, int, minloc, 3)
3 (3) (ARG, int, low, minloc)
4 (4) (ARG, int, high, minloc)
5 (5) (ALLOC, i, minloc, )
6 (6) (ALLOC, x, minloc, )
7 (7) (ALLOC, k, minloc, )
8 (8) (LOAD, t1, low, )
9 (9) (STORE, k, , t1)
10 (10) (LOAD, t2, low, )
11 (11) (LOAD, t3, a, t2)
12 (12) (STORE, x, , t3)
13 (13) (LOAD, t4, low, )
14 (14) (LOAD, t5, 1, )
15 (15) (SUM, t6, t4, t5)
16 (16) (STORE, i, , t6)
17 (17) (LAB, L0, , )
18 (18) (LOAD, t7, i, )
19 (19) (LOAD, t8, high, )
20 (20) (LESS, t9, t7, t8)
21 (21) (IFF, t9, L1, )
22 (22) (LOAD, t10, i, )
23 (23) (LOAD, t11, a, t10)
24 (24) (LOAD, t12, x, )
25 (25) (LESS, t13, t11, t12)
26 (26) (IFF, t13, L2, )
27 (27) (LOAD, t14, i, )
28 (28) (LOAD, t15, a, t14)
29 (29) (STORE, x, , t15)
30 (30) (LOAD, t16, i, )
31 (31) (STORE, k, , t16)
32 (32) (GOTO, L3, , )
33 (33) (LAB, L2, , )
34 (34) (LAB, L3, , )
35 (35) (LOAD, t17, i, )
36 (36) (LOAD, t18, 1, )
37 (37) (SUM, t19, t17, t18)
38 (38) (STORE, i, , t19)
39 (39) (GOTO, L0, , )
40 (40) (LAB, L1, , )
41 (41) (LOAD, t20, k, )

```

```
42 (42) (RET, t20, , )
43 (43) (END, minloc, , )
44 (44) (FUN, void, sort, 3)
45 (45) (ARG, int, low, sort)
46 (46) (ARG, int, high, sort)
47 (47) (ALLOC, i, sort, )
48 (48) (ALLOC, k, sort, )
49 (49) (LOAD, t21, low, )
50 (50) (STORE, i, , t21)
51 (51) (LAB, L4, , )
52 (52) (LOAD, t22, i, )
53 (53) (LOAD, t23, high, )
54 (54) (LOAD, t1, 1, )
55 (55) (SUB, t2, t23, t1)
56 (56) (LESS, t3, t22, t2)
57 (57) (IFF, t3, L5, )
58 (58) (ALLOC, t, sort, )
59 (59) (LOAD, t4, a, )
60 (60) (PARAM, t4, unknown, )
61 (61) (LOAD, t5, i, )
62 (62) (PARAM, t5, var, )
63 (63) (LOAD, t6, high, )
64 (64) (PARAM, t6, param, )
65 (65) (CALL, t7, minloc, 3)
66 (66) (STORE, k, , t7)
67 (67) (LOAD, t8, k, )
68 (68) (LOAD, t9, a, t8)
69 (69) (STORE, t, , t9)
70 (70) (LOAD, t10, i, )
71 (71) (LOAD, t11, a, t10)
72 (72) (LOAD, t12, k, )
73 (73) (STORE, a, t12, t11)
74 (74) (LOAD, t13, t, )
75 (75) (LOAD, t14, i, )
76 (76) (STORE, a, t14, t13)
77 (77) (LOAD, t15, i, )
78 (78) (LOAD, t16, 1, )
79 (79) (SUM, t17, t15, t16)
80 (80) (STORE, i, , t17)
81 (81) (GOTO, L4, , )
82 (82) (LAB, L5, , )
83 (83) (END, sort, , )
84 (84) (FUN, void, main, 1)
85 (85) (ALLOC, i, main, )
86 (86) (LOAD, t18, 0, )
87 (87) (STORE, i, , t18)
88 (88) (LAB, L6, , )
```

```

89 (89) (LOAD, t19, i, )
90 (90) (LOAD, t20, 5, )
91 (91) (LESS, t21, t19, t20)
92 (92) (IFF, t21, L7, )
93 (93) (CALL, t22, input, 0)
94 (94) (LOAD, t23, i, )
95 (95) (STORE, vet, t23, t22)
96 (96) (LOAD, t1, i, )
97 (97) (LOAD, t2, 1, )
98 (98) (SUM, t3, t1, t2)
99 (99) (STORE, i, , t3)
100 (100) (GOTO, L6, , )
101 (101) (LAB, L7, , )
102 (102) (LOAD, t4, vet, )
103 (103) (PARAM, t4, array, )
104 (104) (LOAD, t5, 0, )
105 (105) (PARAM, t5, var, )
106 (106) (LOAD, t6, 5, )
107 (107) (PARAM, t6, var, )
108 (108) (CALL, t7, sort, 3)
109 (109) (LOAD, t8, 0, )
110 (110) (STORE, i, , t8)
111 (111) (LAB, L8, , )
112 (112) (LOAD, t9, i, )
113 (113) (LOAD, t10, 5, )
114 (114) (LESS, t11, t9, t10)
115 (115) (IFF, t11, L9, )
116 (116) (LOAD, t12, i, )
117 (117) (LOAD, t13, vet, t12)
118 (118) (PARAM, t13, var, )
119 (119) (CALL, t14, output, 1)
120 (120) (LOAD, t15, i, )
121 (121) (LOAD, t16, 1, )
122 (122) (SUM, t17, t15, t16)
123 (123) (STORE, i, , t17)
124 (124) (GOTO, L8, , )
125 (125) (LAB, L9, , )
126 (126) (END, main, , )
127 (127) (HALT, , , )
128 ; Fim da Lista TAC

```

Listing 5.10 – Código intermediário (parcial) para `sort.c`

5.3.3 Código Assembly Gerado

No código Assembly, a passagem do vetor `vet` para a função `sort` é feita por referência, ou seja, o endereço base do vetor é que é empilhado. O acesso aos elementos,

como `a[k]`, é traduzido para uma sequência de instruções que calcula o endereço do elemento (`endereço_base + índice`) e então usa LDR ou STR para acessá-lo.

```
1 ; Arquivo Assembly gerado pelo compilador C-
2
3 .data
4     vet: .space 5
5
6 .text
7     NOP
8     MOVI SP, #0
9     MOVI FP, #0
10    ADDI SP, SP, #5
11    B main
12 .minloc
13    ADDI Rlink, Rlink, #1
14    STR Rlink [SP #0]
15    ADDI SP, SP, #1
16    ADDI SP, SP, #1
17    ADDI SP, SP, #1
18    ADDI SP, SP, #1
19    ; Acessando param'low'
20    MOV Rad, FP
21    SUBI Rad, Rad, #2
22    LDR R1, [Rad, #0]
23    ; Armazenando em variavel local 'k'
24    STR R1, [FP, #4]
25    ; Acessando param'low'
26    MOV Rad, FP
27    SUBI Rad, Rad, #2
28    LDR R2, [Rad, #0]
29    ; Acessando param array: 'a'
30    MOV Rad, FP
31    SUBI Rad, Rad, #3
32    LDR Rad, [Rad, #0]
33    ADD Rad, Rad, R2
34    LDR R3, [Rad, #0]
35    ; Armazenando em variavel local 'x'
36    STR R3, [FP, #3]
37    ; Acessando param'low'
38    MOV Rad, FP
39    SUBI Rad, Rad, #2
40    LDR R4, [Rad, #0]
41    MOVI R5, #1
42    ADD R6, R5, R4
43    ; Armazenando em variavel local 'i'
44    STR R6, [FP, #2]
45 L0:
```

```
46 ; Acessando variavel local 'i'
47 LDR R7, [FP, #2]
48 ; Acessando param'high'
49 MOV Rad, FP
50 SUBI Rad, Rad, #1
51 LDR R8, [Rad, #0]
52 CMP R7, R8
53 BGE L1
54 ; Acessando variavel local 'i'
55 LDR R10, [FP, #2]
56 ; Acessando param array: 'a'
57 MOV Rad, FP
58 SUBI Rad, Rad, #3
59 LDR Rad, [Rad, #0]
60 ADD Rad, Rad, R10
61 LDR R11, [Rad, #0]
62 ; Acessando variavel local 'x'
63 LDR R12, [FP, #3]
64 CMP R11, R12
65 BGE L2
66 ; Acessando variavel local 'i'
67 LDR R14, [FP, #2]
68 ; Acessando param array: 'a'
69 MOV Rad, FP
70 SUBI Rad, Rad, #3
71 LDR Rad, [Rad, #0]
72 ADD Rad, Rad, R14
73 LDR R15, [Rad, #0]
74 ; Armazenando em variavel local 'x'
75 STR R15, [FP, #3]
76 ; Acessando variavel local 'i'
77 LDR R16, [FP, #2]
78 ; Armazenando em variavel local 'k'
79 STR R16, [FP, #4]
80 B L3
81 L2:
82 L3:
83 ; Acessando variavel local 'i'
84 LDR R17, [FP, #2]
85 MOVI R18, #1
86 ADD R19, R18, R17
87 ; Armazenando em variavel local 'i'
88 STR R19, [FP, #2]
89 B L0
90 L1:
91 ; Acessando variavel local 'k'
92 LDR R20, [FP, #4]
```

```
93     MOV Rret, R20
94     LDR Rlink [FP #1]
95     MOV SP, FP
96     LDR FP [FP #0]
97     B Rlink
98 .sort
99     ADDI Rlink, Rlink, #1
100    STR Rlink [SP #0]
101    ADDI SP, SP, #1
102    ADDI SP, SP, #1
103    ADDI SP, SP, #1
104    ; Acessando param'low'
105    MOV Rad, FP
106    SUBI Rad, Rad, #2
107    LDR R21, [Rad, #0]
108    ; Armazenando em variavel local 'i'
109    STR R21, [FP, #2]
110 L4:
111    ; Acessando variavel local 'i'
112    LDR R22, [FP, #2]
113    ; Acessando param'high'
114    MOV Rad, FP
115    SUBI Rad, Rad, #1
116    LDR R23, [Rad, #0]
117    MOVI R1, #1
118    SUB R2, R23, R1
119    CMP R22, R2
120    BGE L5
121    ADDI SP, SP, #1
122    MOV Rad, FP
123    SUBI Rad, Rad, #3
124    MOV R4, Rad
125    STR R4 [SP, #0]
126    ADDI SP, SP, #1
127    ; Acessando variavel local 'i'
128    LDR R5, [FP, #2]
129    STR R5 [SP, #0]
130    ADDI SP, SP, #1
131    ; Acessando param'high'
132    MOV Rad, FP
133    SUBI Rad, Rad, #1
134    LDR R6, [Rad, #0]
135    STR R6 [SP, #0]
136    ADDI SP, SP, #1
137    STR FP [SP, #0]
138    MOV FP, SP
139    ADDI SP, SP, #1
```

```
140    BL minloc
141    MOV R7, Rret
142    ; Armazenando em variavel local 'k'
143    STR R7, [FP, #3]
144    ; Acessando variavel local 'k'
145    LDR R8, [FP, #3]
146    ; Acessando param array: 'a'
147    MOV Rad, FP
148    SUBI Rad, Rad, #3
149    LDR Rad, [Rad, #0]
150    ADD Rad, Rad, R8
151    LDR R9, [Rad, #0]
152    ; Armazenando em variavel local 't'
153    STR R9, [FP, #4]
154    ; Acessando variavel local 'i'
155    LDR R10, [FP, #2]
156    ; Acessando param array: 'a'
157    MOV Rad, FP
158    SUBI Rad, Rad, #3
159    LDR Rad, [Rad, #0]
160    ADD Rad, Rad, R10
161    LDR R11, [Rad, #0]
162    ; Acessando variavel local 'k'
163    LDR R12, [FP, #3]
164    ; Armazenando em param array 'a'
165    MOV Rad, FP
166    SUBI Rad, Rad, #3
167    LDR Rad, [Rad, #0]
168    ADD Rad, Rad, R12
169    STR R11, [Rad, #0]
170    ; Acessando variavel local 't'
171    LDR R13, [FP, #4]
172    ; Acessando variavel local 'i'
173    LDR R14, [FP, #2]
174    ; Armazenando em param array 'a'
175    MOV Rad, FP
176    SUBI Rad, Rad, #3
177    LDR Rad, [Rad, #0]
178    ADD Rad, Rad, R14
179    STR R13, [Rad, #0]
180    ; Acessando variavel local 'i'
181    LDR R15, [FP, #2]
182    MOVI R16, #1
183    ADD R17, R16, R15
184    ; Armazenando em variavel local 'i'
185    STR R17, [FP, #2]
186    B L4
```

```
187 L5 :
188     LDR Rlink [FP #1]
189     MOV SP, FP
190     LDR FP [FP #0]
191     B Rlink
192 .main
193     MOV FP, SP
194     ADDI SP, SP, #1
195     MOVI R18, #0
196     ; Armazenando em variavel local 'i'
197     STR R18, [FP, #0]

198 L6 :
199     ; Acessando variavel local 'i'
200     LDR R19, [FP, #0]
201     MOVI R20, #5
202     CMP R19, R20
203     BGE L7
204     IN
205     MOV R22, Rin
206     ; Acessando variavel local 'i'
207     LDR R23, [FP, #0]
208     ; Armazenando em array global 'vet'
209     MOVI Rad, #0
210     ADD Rad, Rad, R23
211     STR R22, [Rad, #0]
212     ; Acessando variavel local 'i'
213     LDR R1, [FP, #0]
214     MOVI R2, #1
215     ADD R3, R2, R1
216     ; Armazenando em variavel local 'i'
217     STR R3, [FP, #0]
218     B L6

219 L7 :
220
221     MOVI Rad, #0
222     MOV R4, Rad
223     STR R4 [SP, #0]
224     ADDI SP, SP, #1
225     MOVI R5, #0
226     STR R5 [SP, #0]
227     ADDI SP, SP, #1
228     MOVI R6, #5
229     STR R6 [SP, #0]
230     ADDI SP, SP, #1
231     STR FP [SP, #0]
232     MOV FP, SP
233     ADDI SP, SP, #1
```

```

234    BL sort
235    MOV R7, Rret
236    MOVI R8, #0
237    ; Armazenando em variavel local 'i'
238    STR R8, [FP, #0]
239 L8:
240    ; Acessando variavel local 'i'
241    LDR R9, [FP, #0]
242    MOVI R10, #5
243    CMP R9, R10
244    BGE L9
245    ; Acessando variavel local 'i'
246    LDR R12, [FP, #0]
247    ; Acessando array global 'vet'
248    MOVI Rad, #0
249    ADD Rad, Rad, R12
250    LDR R13, [Rad, #0]
251    MOV Rout, R13
252    OUT
253    ; Acessando variavel local 'i'
254    LDR R15, [FP, #0]
255    MOVI R16, #1
256    ADD R17, R16, R15
257    ; Armazenando em variavel local 'i'
258    STR R17, [FP, #0]
259    B L8
260 L9:
261    FINISH
262
263 ; Fim do programa

```

Listing 5.11 – Código Assembly (parcial) para a função `sort`

5.3.4 Código Executável (Binário)

A seguir, o código binário final gerado pelo montador para o programa `sort.c`.

```

1 %
2 11101100000000000000000000000000000000
3 11100011101000001101000000000000
4 1110001110100000110110000000000
5 11100010101011010110100000000101
6 1110101000000000000000000010001010
7 1110001010101111111100000000001
8 1110011001101011111000000000000
9 111000101010110101101000000000001
10 111000101010110101101000000000001
11 111000101010110101101000000000001

```



```
59 11100000101010010100111000100000  
60 111001100110111001100000000000010  
61 1110101000000000000000000011110  
62 11100110111011101000000000000100  
63 111000111010100110000000000000  
64 11100110111011111000000000000001  
65 111000111010110111101000000000000  
66 11100110111011110110000000000000  
67 111010001111000000000000000000000  
68 111000101011111111100000000000001  
69 11100110011010111110000000000000  
70 111000101010110101101000000000001  
71 111000101010110101101000000000001  
72 111000101010110101101000000000001  
73 111000111010110111100100000000000  
74 1110001001101100111001000000000010  
75 11100110111001101010000000000000  
76 1110011001101110101000000000000010  
77 11100110111011101100000000000010  
78 111000111010110111100100000000000  
79 111000100110110011100100000000001  
80 11100110111001101110000000000000  
81 1110001110100000000001000000000001  
82 1110000001101011100100000100000  
83 11100001010110110000000001000000  
84 1010101000000000000000000000000001  
85 11100010101101011010000000000001  
86 111000111010110111100100000000000  
87 1110001001101100111001000000000011  
88 11100011101011001001000000000000  
89 111001100110100010000000000000000  
90 11100010101101011010000000000001  
91 111001101110110010100000000000010  
92 111001100110100010100000000000000  
93 11100010101101011010000000000001  
94 1110001110101101111001000000000000  
95 111000100110110011100100000000001  
96 111001101110010011000000000000000  
97 1110011001101000110000000000000000  
98 1110001010101101110100000000000001  
99 111001100110101101100000000000000  
100 1110001110101101011011000000000000  
101 1110001010110101101000000000000001  
102 1110101100000000000000000000000000001  
103 11100011101011000001110000000000000  
104 11100110011011001110000000000000011  
105 11100110111011010000000000000000011
```

```
106 11100011101011011100100000000000
107 11100010011011001110010000000011
108 11100110111001110010000000000000
109 11100000101011001110010100000000
110 11100110111001010010000000000000
111 11100110011011010010000000000000
112 11100110111011010100000000000000
113 11100011101011011100100000000000
114 111000100110110011100100000000000
115 11100110111001110010000000000000
116 11100000101011001110010101000000
117 11100110111001010110000000000000
118 111001101110110110000000000000000
119 11100011101011011100100000000000
120 111000100110110011100100000000000
121 111001101110011100100000000000000
122 11100000101011001110010110000000
123 111001100110010110000000000000000
124 11100110111011011000000000000000
125 111001101110110111000000000000000
126 111000111010110111001000000000000
127 111000100110110011100100000000000
128 1110011011100111001000000000000000
129 11100000101011001110010111000000
130 111001100110010110100000000000000
131 11100110111011011100000000000000
132 1110001110100000010000000000000000
133 11100000101010000100010111100000
134 111001100110111000100000000000000
135 1110101000000000000000000000000000
136 11100110111011111000000000000000
137 111000111010110111010000000000000
138 111001101110111101100000000000000
139 1110100011110000000000000000000000
140 111000111010110110110000000000000
141 111000101010110101101000000000000
142 111000111010000001001000000000000
143 11100110011011001000000000000000
144 111001101110111001100000000000000
145 111000111010000001010000000000000
146 11100001010110011000001010000000
147 1010101000000000000000000000000000
148 11101100100000000000000000000000000
149 111000111010111001011000000000000
150 111001101110111011100000000000000
151 111000111010000001100100000000000
152 11100000101011001110011011100000
```

```

153 11100110011001101100000000000000
154 11100110110110000100000000000000
155 11100011101000000001000000000001
156 11100000101000010000110000100000
157 11100110011011000110000000000000
158 111010100000000000000000000010001110
159 1110001110100000011001000000000000
160 11100011101011001001000000000000
161 111001100110100010000000000000000
162 111000101010110101101000000000001
163 111000111010000000010100000000000
164 111001100110100010100000000000000
165 1110001010101101011010000000000001
166 111000111010000000011000000000101
167 111001100110100011000000000000000
168 1110001010101101011010000000000001
169 111001100110101101100000000000000
170 1110001110101101011011000000000000
171 111000101010110101101000000000001
172 1110101100000000000000000000001000010
173 111000111010110000011000000000000
174 1110001110100000001000000000000000
175 1110011001101101000000000000000000
176 111001101110110100100000000000000
177 111000111010000000101000000000101
178 11100001010101001000000101000000
179 101010100000000000000000000010111101
180 1110011011101101100000000000000000
181 111000111010000001100100000000000
182 11100000101011001110010110000000
183 111001101110010110100000000000000
184 111000111010011011101000000000000
185 111011010000000000000000000000000000
186 1110011011101101111000000000000000
187 11100011101000000100000000000000001
188 11100000101010000100010111100000
189 111001100110111000100000000000000
190 1110101000000000000000000000000000000000000
191 1110110110000000000000000000000000000000000000

```

Listing 5.12 – Código binário para `sort.c`

5.3.5 Correspondência entre as Etapas

Na tabela 16 pode ser visto a conversão entre o código fonte e o código intermediário do algoritmo `sort.c`.

Tabela 16 – Correspondência entre Código C- e Código Intermediário para o algoritmo de Ordenação.

Código C-	Código Intermediário (TAC)
1 <code>int vet[5];</code>	1 (ALLOC , vet, global, 5)
1 <code>int minloc(int a[],</code> 2 <code>int low,</code> 3 <code>int high) {</code> 4 <code>int i; int x; int k;</code>	1 (FUN , int, minloc, 3) 2 (ARG , int, low, minloc) 3 (ARG , int, high, minloc) 4 (ALLOC , i, minloc,) 5 (ALLOC , x, minloc,) 6 (ALLOC , k, minloc,)
1 <code>k = low;</code>	1 (LOAD , t1, low,) 2 (STORE , k, , t1)
1 <code>x = a[low];</code>	1 (LOAD , t2, low,) 2 (LOAD , t3, a, t2) 3 (STORE , x, , t3)
1 <code>i = low + 1;</code>	1 (LOAD , t4, low,) 2 (LOAD , t5, 1,) 3 (SUM , t6, t4, t5) 4 (STORE , i, , t6)
1 <code>while (i < high) {</code>	1 (LAB , L0, ,) 2 (LOAD , t7, i,) 3 (LOAD , t8, high,) 4 (LESS , t9, t7, t8) 5 (IFF , t9, L1,)
1 <code>if (a[i] < x) {</code> 2 <code>x = a[i];</code> 3 <code>k = i;</code> 4 <code>}</code>	1 (LOAD , t10, i,) 2 (LOAD , t11, a, t10) 3 (LOAD , t12, x,) 4 (LESS , t13, t11, t12) 5 (IFF , t13, L2,) 6 (LOAD , t14, i,) 7 (LOAD , t15, a, t14) 8 (STORE , x, , t15) 9 (LOAD , t16, i,) 10 (STORE , k, , t16)

Continua na próxima página

Tabela 16 (continuação)

Código C-	Código Intermediário (TAC)
<pre> 1 i = i + 1; 2 }</pre>	<pre> 1 (GOTO, L3, ,) 2 (LAB, L2, ,) 3 (LAB, L3, ,) 4 (LOAD, t17, i,) 5 (LOAD, t18, 1,) 6 (SUM, t19, t17, t18) 7 (STORE, i, , t19) 8 (GOTO, L0, ,)</pre>
<pre> 1 return k;</pre>	<pre> 1 (LAB, L1, ,) 2 (LOAD, t20, k,) 3 (RET, t20, ,) 4 (END, minloc, ,)</pre>
<pre> 1 void sort(int a[], 2 int low, 3 int high) { 4 int i; int k; 5 i = low;</pre>	<pre> 1 (FUN, void, sort, 3) 2 (ARG, int, low, sort) 3 (ARG, int, high, sort) 4 (ALLOC, i, sort,) 5 (ALLOC, k, sort,) 6 (LOAD, t21, low,) 7 (STORE, i, , t21)</pre>
<pre> 1 while (i < high - 1) {</pre>	<pre> 1 (LAB, L4, ,) 2 (LOAD, t22, i,) 3 (LOAD, t23, high,) 4 (LOAD, t1, 1,) 5 (SUB, t2, t23, t1) 6 (LESS, t3, t22, t2) 7 (IFF, t3, L5,)</pre>
<pre> 1 int t; 2 k = minloc(a,i,high);</pre>	<pre> 1 (ALLOC, t, sort,) 2 (LOAD, t4, a,) 3 (PARAM, t4, unknown,) 4 (LOAD, t5, i,) 5 (PARAM, t5, var,) 6 (LOAD, t6, high,) 7 (PARAM, t6, param,) 8 (CALL, t7, minloc, 3) 9 (STORE, k, , t7)</pre>

Continua na próxima página

Tabela 16 (continuação)

Código C-	Código Intermediário (TAC)
<pre> 1 t = a[k]; 2 a[k] = a[i]; 3 a[i] = t; </pre>	<pre> 1 (LOAD, t8, k,) 2 (LOAD, t9, a, t8) 3 (STORE, t, , t9) 4 (LOAD, t10, i,) 5 (LOAD, t11, a, t10) 6 (LOAD, t12, k,) 7 (STORE, a, t12, t11) 8 (LOAD, t13, t,) 9 (LOAD, t14, i,) 10 (STORE, a, t14, t13) </pre>
<pre> 1 i = i + 1; 2 } </pre>	<pre> 1 (LOAD, t15, i,) 2 (LOAD, t16, 1,) 3 (SUM, t17, t15, t16) 4 (STORE, i, , t17) 5 (GOTO, L4, ,) 6 (LAB, L5, ,) 7 (END, sort, ,) </pre>
<pre> 1 void main(void) { 2 int i; 3 i = 0; </pre>	<pre> 1 (FUN, void, main, 1) 2 (ALLOC, i, main,) 3 (LOAD, t18, 0,) 4 (STORE, i, , t18) </pre>
<pre> 1 while (i < 5) { 2 vet[i] = input(); 3 i = i + 1; 4 } </pre>	<pre> 1 (LAB, L6, ,) 2 (LOAD, t19, i,) 3 (LOAD, t20, 5,) 4 (LESS, t21, t19, t20) 5 (IFF, t21, L7,) 6 (CALL, t22, input, 0) 7 (LOAD, t23, i,) 8 (STORE, vet, t23, t22) 9 (LOAD, t1, i,) 10 (LOAD, t2, 1,) 11 (SUM, t3, t1, t2) 12 (STORE, i, , t3) 13 (GOTO, L6, ,) </pre>

Continua na próxima página

Tabela 16 (continuação)

Código C-	Código Intermediário (TAC)
sort(vet,0,5);	1 (LAB , L7, ,) 2 (LOAD , t4, vet,) 3 (PARAM , t4, array,) 4 (LOAD , t5, 0,) 5 (PARAM , t5, var,) 6 (LOAD , t6, 5,) 7 (PARAM , t6, var,) 8 (CALL , t7, sort, 3)
i = 0; while (i < 5) { output(vet[i]); i = i + 1; }	1 (LOAD , t8, 0,) 2 (STORE , i, , t8) 3 (LAB , L8, ,) 4 (LOAD , t9, i,) 5 (LOAD , t10, 5,) 6 (LESS , t11, t9, t10) 7 (IFF , t11, L9,) 8 (LOAD , t12, i,) 9 (LOAD , t13, vet, t12) 10 (PARAM , t13, var,) 11 (CALL , t14, output, 1) 12 (LOAD , t15, i,) 13 (LOAD , t16, 1,) 14 (SUM , t17, t15, t16) 15 (STORE , i, , t17) 16 (GOTO , L8, ,)
// Fim da main	1 (LAB , L9, ,) 2 (END , main, ,) 3 (HALT , , ,)

6 Conclusão

Este trabalho apresentou o projeto e a implementação de um compilador completo para a linguagem C-, desde a análise do código-fonte até a geração de código executável para uma arquitetura de processador ARM customizada. O desenvolvimento abrangeu todas as fases canônicas de um compilador, resultando em uma ferramenta funcional capaz de traduzir algoritmos complexos. A conclusão deste projeto não apenas solidificou os conceitos teóricos da disciplina, mas também proporcionou uma experiência prática valiosa na construção de uma ferramenta de software fundamental.

Durante o desenvolvimento, a implementação correta do registro de ativação (*stack frame*) emergiu como um dos maiores desafios, exigindo um planejamento cuidadoso para garantir que o prólogo e o epílogo de cada função gerenciassem o estado do chamador de forma consistente, especialmente em chamadas aninhadas e recursivas. Da mesma forma, a tradução do acesso a elementos de vetor para a linguagem de montagem provou ser uma tarefa complexa, que demandou a geração de sequências precisas de instruções para a aritmética de ponteiros.

Apesar das dificuldades, o projeto alcançou seus objetivos, e seu maior destaque é, sem dúvida, a entrega de um fluxo de compilação completo e funcional. A capacidade do compilador de lidar com recursividade, demonstrada no exemplo do MDC, e com a passagem de vetores por referência, validou a solidez do modelo de gerenciamento de memória implementado. Além disso, a sinergia com o projeto de hardware, ao desenvolver o compilador para um processador projetado na disciplina de Arquitetura e Organização de Computadores, representa um diferencial importante, demonstrando uma compreensão integrada do ciclo de desenvolvimento de sistemas computacionais.

Referências

LOUDEN, K. C. *Compiler Construction: Principles and Practice*. 1^a. ed. [S.l.]: Cengage Learning, 1997. ISBN 978-0534939724. Citado na página [7](#).