Implementação do Compilador C

Aluno - Manoel Augusto de Souza Serafim

Professor - Prof. Dr. Luiz Eduardo Galvão Martins

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

Instituto de Ciência e Tecnologia - UNIFESP
08/09/2024

Índice

1 Introdução

Os compiladores são fundamentais na engenharia de software, atuando como a ponte entre o código fonte escrito em linguagens de alto nível e a linguagem de máquina que os processadores entendem. Eles traduzem o código legível para humanos em um formato que pode ser executado diretamente pelo hardware, garantindo que os programas funcionem de forma eficiente e correta. Esse processo é essencial para o desenvolvimento de software moderno.

Assim como existem diferentes arquiteturas de computadores, há uma variedade de compiladores especializados para cada uma delas. Na indústria de defesa, por exemplo, fornecedores como a Green Hills oferecem compiladores que geram código de máquina altamente otimizado e fortemente acoplados ao código fonte. Em contraste, compiladores de código aberto geralmente focam em maximizar a performance para arquiteturas gerais e minimizar o uso de memória, com menos instruções e laços rolados, em arquiteturas embarcadas. Sendo assim, a escolha de um compilador está fortemente ligada a estrutura de hardware da plataforma onde o código vai ser executado.

Este projeto final da disciplina tem como objetivo desenvolver um compilador para a arquitetura de conjunto de instruções RISCV32IM. O compilador será responsável por traduzir código escrito em uma linguagem de alto nível para o conjunto de instruções que pode ser interpretado por processadores RISCV32.

A estrutura deste relatório está organizada para fornecer uma visão abrangente e detalhada do projeto. O **Capítulo 4** inicia com a apresentação do processador, discutindo o diagrama de blocos do processador, seus componentes principais, o conjunto de instruções que ele suporta e a organização da memória.

No **Capítulo 5**, o foco é a fase de análise do compilador. Este capítulo cobre a modelagem, utilizando diagramas de blocos e diagramas de atividades, além de abordar a análise léxica, sintática e semântica, que são cruciais para a tradução correta do código fonte.

O **Capítulo 6** explora a fase de síntese do compilador, discutindo a modelagem com diagramas apropriados, a geração do código intermediário (incluindo a explicação dos tipos de quádruplas), e a produção do código assembly e executável. Este capítulo também aborda o gerenciamento de memória.

No **Capítulo 7**, são apresentados exemplos práticos para ilustrar o funcionamento do compilador. Este capítulo mostra o código fonte, o código intermediário gerado, o código assembly e o código executável, detalhando a correspondência entre essas etapas para demonstrar a eficácia do compilador.

Finalmente, o **Capítulo 8** oferece uma conclusão, discutindo as dificuldades encontradas durante o desenvolvimento e os principais destaques do projeto.

2 O Processador

Nesta seção, é fornecida uma visão abrangente do processador desenvolvido para o qual o compilador foi projetado. Este processador de 32-bits é baseado na arquitetura RISC-V 32IM, uma implementação do conjunto de instruções RISC-V de 32 bits que compreende a base RV32I e a extensão M para multiplicação e divisão. O softcore que será usado nesse projeto se baseia nessa ISA e é chamado de NUGGET.

2.1 Arquitetura RISC-V 32IM

A arquitetura RISC-V 32IM é uma implementação do conjunto de instruções de 32 bits, que é composto por dois componentes principais:

2.1.1 Base RV32I: A base RV32I define o conjunto básico de instruções para processadores RISC-V de 32 bits. De acordo com o manual da ISA RISC-V, o RV32I inclui um conjunto de instruções reduzido e eficiente que abrange operações aritméticas, lógicas, de controle e de acesso à memória. As instruções RV32I utilizam formatos compactos e consistentes, o que reduz a complexidade de decodificação e execução. O conjunto inclui instruções para operações aritméticas básicas, como adição e subtração, além de operações lógicas como E, OU e XOR. Também oferece suporte ao controle de fluxo, com instruções que permitem saltos condicionais e incondicionais, facilitando a implementação de estruturas de controle mais complexas. Por fim, o RV32I suporta operações de leitura e escrita em memória, com instruções específicas para o acesso a endereços e manipulação de dados.

2.1.2 Extensão M: A extensão M do conjunto de instruções RISC-V adiciona suporte para operações de multiplicação e divisão, que não estão presentes na base RV32I. Essa extensão inclui instruções de multiplicação de inteiros, permitindo a execução eficiente de operações aritméticas complexas diretamente no hardware. Também incorpora instruções de divisão de inteiros, o que melhora o desempenho em algoritmos que requerem tais operações. A extensão M é projetada para oferecer precisão e desempenho otimizados nessas operações, eliminando a necessidade de implementações por software.

2.2 Diagrama de Blocos do NUGGET

A Figura 1 mostra o diagrama de blocos do processador NUGGET. Este diagrama ilustra a organização e interconexão dos principais componentes do processador, incluindo a Unidade de Controle, a Unidade Aritmética e Lógica, a Unidade de Multiplicação e Divisão, os Registradores, a Unidade de Acesso à Memória, a Unidade de Decodificação de Instruções, a Unidade de Controle de Fluxo e a Unidade de Controle de Exceções.

- **UnidadeDeControle**: Gerencia execução e controle das operações.
- **UnidadeAritmeticaELogica**: Realiza operações aritméticas e lógicas.
- **UnidadeMultiplicacaoEDivisao**: Executa operações de multiplicação e divisão.
- **Registradores**: Armazena dados temporários e resultados.
- **UnidadeAcessoAMemoria**: Lida com leitura e escrita de dados na memória.
- **UnidadeDecodificacaoDeInstrucoes**: Traduz instruções para sinais de controle.

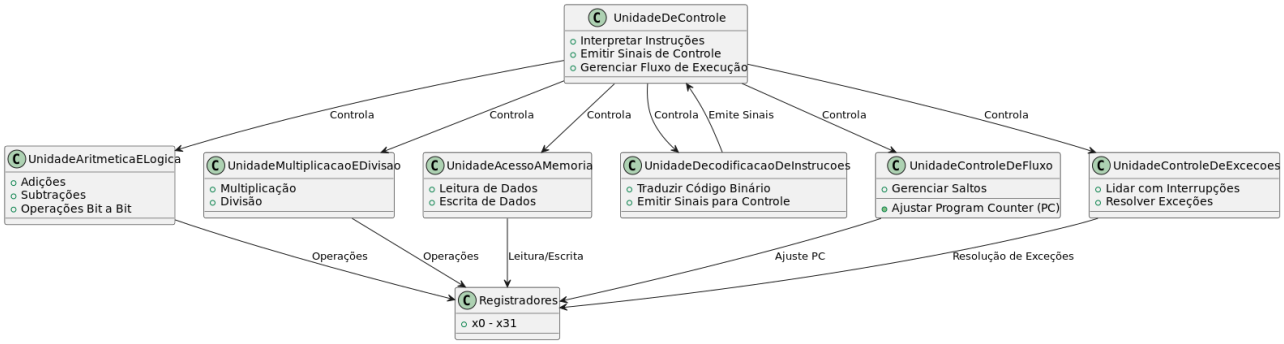


Figure 1: Diagrama PUML do processador

- **UnidadeControleDeFluxo**: Gerencia o fluxo de execução das instruções.
- **UnidadeControleDeExcecoes**: Lida com interrupções e exceções.

2.3 Explicação dos Componentes do NUGGET

O NUGGET é uma implementação específica de um processador RISC-V 32IM, projetada para ser compacta e eficiente, ideal para aplicações embarcadas.

2.3.1 Unidade de Controle: A Unidade de Controle no NUGGET é responsável por interpretar as instruções e emitir os sinais de controle necessários para coordenar as operações das demais unidades do processador. Ela gerencia o fluxo de execução das instruções e controla como os dados são manipulados e transferidos entre as diferentes partes do processador. Esta unidade é projetada para ser compacta e eficiente, considerando o objetivo de manter o núcleo pequeno e de baixo consumo de energia.

2.3.2 Unidade Aritmética e Lógica (ALU): A ALU no NUGGET realiza operações aritméticas e lógicas essenciais, como adições, subtrações e operações bit a bit (AND, OR, XOR). Ela é fundamental para a execução das operações básicas que formam a base dos cálculos realizados pelo processador. A ALU é otimizada para ser compacta e eficiente, mantendo a simplicidade do núcleo.

2.3.3 Unidade de Multiplicação e Divisão: Embora o NUGGET implemente a arquitetura RISC-V com o conjunto de instruções ‘M‘ para multiplicação e divisão, a abordagem específica pode variar. Dependendo dos requisitos de desempenho e área, a multiplicação e a divisão podem ser implementadas por unidades especializadas ou por meio de algoritmos baseados em software.

2.3.4 Registradores: O PicoSoC32V possui um conjunto de 32 registradores de propósito geral, cada um com 32 bits. Estes registradores são utilizados para armazenar dados temporários e resultados intermediários durante a execução das instruções. Além dos registradores gerais, o Processador RISC-V 32IM também inclui registradores especiais. A tabela a seguir mostra uma descrição dos registradores e como serão utilizados pelo compilador:

Registrador	Descrição
x0	zero (zero fixo)
x1	ra (endereço de retorno)
x2	sp (ponteiro de pilha)
x3	gp (ponteiro global)
x4	tp (ponteiro de thread)
x5	t0 (registrador temporário 0)
x6	t1 (registrador temporário 1)
x7	t2 (registrador temporário 2)
x8	s0 / fp (registrador salvo 0 / ponteiro de quadro)
x9	s1 (registrador salvo 1)
x10	a0 (argumento de função 0 / valor de retorno 0)
x11	a1 (argumento de função 1 / valor de retorno 1)
x12	a2 (argumento de função 2)
x13	a3 (argumento de função 3)
x14	a4 (argumento de função 4)
x15	a5 (argumento de função 5)
x16	a6 (argumento de função 6)
x17	a7 (argumento de função 7)
x18	s2 (registrador salvo 2)
x19	s3 (registrador salvo 3)
x20	s4 (registrador salvo 4)
x21	s5 (registrador salvo 5)
x22	s6 (registrador salvo 6)
x23	s7 (registrador salvo 7)
x24	s8 (registrador salvo 8)
x25	s9 (registrador salvo 9)
x26	s10 (registrador salvo 10)
x27	s11 (registrador salvo 11)
x28	t3 (registrador temporário 3)
x29	t4 (registrador temporário 4)
x30	t5 (registrador temporário 5)
x31	t6 (registrador temporário 6)

Table 1: Descrição dos Registradores do Processador RISC-V 32IM

2.3.5 Unidade de Acesso à Memória: A Unidade de Acesso à Memória no NUGGET lida com operações de leitura e escrita de dados na memória. Esta unidade gerencia a comunicação entre o processador e a memória principal, permitindo que o processador busque dados e instruções e armazene resultados. O PicoSoC32V é projetado para ser eficiente em termos de área e consumo de energia, portanto, esta unidade é compacta e direta.

2.3.6 Unidade de Decodificação de Instruções: A Unidade de Decodificação de Instruções no NUGGET traduz o código binário das instruções para sinais que podem ser usados pela Unidade de Controle e outras unidades do processador. Ela interpreta o formato das instruções e garante que os sinais corretos sejam emitidos para a execução das operações.

2.3.7 Unidade de Controle de Fluxo: A Unidade de Controle de Fluxo gerencia as instruções que alteram o fluxo de execução, como saltos e chamadas de sub-rotinas. Ela ajusta o valor do Program Counter (PC) conforme necessário para garantir que o fluxo de execução

continue corretamente.

2.3.8 Unidade de Controle de Exceções: A Unidade de Controle de Exceções lida com interrupções e exceções que podem ocorrer durante a execução do programa. Ela garante que o processador possa responder a condições anômalas e retomar a execução normal após a resolução das exceções.

2.4 Conjunto de Instruções

O conjunto de instruções do processador *RISC-V 32IM* abrange operações aritméticas, lógicas, de controle de fluxo e de acesso à memória. Abaixo estão detalhadas as principais instruções, com exemplos baseados no que o gerador de código assembly ira construir.

2.4.1 Instruções de Acesso à Memória: Essas instruções são usadas para carregar ou armazenar dados na memória.

- `lw` (*load word*): Carrega uma palavra (32 bits) da memória em um registrador.
  - Exemplo: `lw x1, -4(x2)` carrega o valor armazenado no endereço calculado como `-4 + x2` para o registrador `x1`.
- `sw` (*store word*): Armazena uma palavra (32 bits) de um registrador na memória.
  - Exemplo: `sw x1, -4(x2)` armazena o valor do registrador `x1` no endereço calculado como `-4 + x2`.

2.4.2 Instruções Aritméticas: Instruções usadas para realizar operações matemáticas básicas.

- `add`: Soma dois registradores e armazena o resultado no registrador de destino.
  - Exemplo: `add x1, x2, x3` soma `x2` e `x3` e armazena o resultado em `x1`.
- `sub`: Subtrai o valor do segundo registrador do primeiro e armazena o resultado no registrador de destino.
  - Exemplo: `sub x1, x2, x3` subtrai `x3` de `x2` e armazena o resultado em `x1`.
- `mul`: Multiplica dois registradores e armazena o resultado no registrador de destino.
  - Exemplo: `mul x1, x2, x3` multiplica `x2` e `x3` e armazena o resultado em `x1`.
- `div`: Divide dois registradores e armazena o resultado no registrador de destino.
  - Exemplo: `div x1, x2, x3` divide `x2` por `x3` e armazena o quociente em `x1`.
- `addi`: Soma um valor imediato a um registrador.
  - Exemplo: `addi x1, x2, 10` soma 10 ao valor de `x2` e armazena o resultado em `x1`.

2.4.3 Instruções de Fluxos Condicionais: Estas instruções permitem comparações entre valores armazenados nos registradores.

- `beq`: Realiza um salto condicional se dois registradores forem iguais.
  - Exemplo: `beq x1, x2, label` faz o salto para `label` se `x1` e `x2` forem iguais.
- `bne`: Realiza um salto condicional se dois registradores forem diferentes.
  - Exemplo: `bne x1, x2, label` faz o salto para `label` se `x1` e `x2` forem diferentes.
- `ble`, `bge`, `blt`, `bgt`: São usadas para comparação de menor ou igual, maior ou igual, menor e maior, respectivamente, entre dois registradores, realizando um salto condicional.
  - Exemplo: `blt x1, x2, label` salta para `label` se `x1` for menor que `x2`.

2.4.4 Instruções de Controle de Fluxo: Estas instruções são usadas para controlar o fluxo de execução do programa.

- `j`: Realiza um salto incondicional para um rótulo.
  - Exemplo: `j label` faz o salto incondicional para `label`.
- `jal`: Realiza um salto incondicional e armazena o endereço de retorno no registrador `ra`.
  - Exemplo: `jal label` salta para `label` e armazena o endereço de retorno.
- `ret`: Retorna para o endereço armazenado no registrador `ra`.
  - Exemplo: `ret` retorna à execução no endereço armazenado em `ra`.

2.4.5 Instruções Miscelâneas:

- `li`: Carrega um valor imediato em um registrador.
  - Exemplo: `li x1, 10` carrega o valor 10 no registrador `x1`.
- `mv`: Move o valor de um registrador para outro.
  - Exemplo: `mv x1, x2` copia o valor de `x2` para `x1`.
- `nop`: Instrução de não operação, usada para marcação de rótulos.

2.5 Organização da Memória

A organização da memória em processadores RISC-V 32IM, como o NUGGET, segue uma arquitetura von Neumann tradicional, onde a memória RAM, memória ROM e qualquer memória especial, como cache, estão interconectadas e acessíveis pelo processador através de um barramento de dados compartilhado. A memória é dividida em seções distintas:

Memória RAM (Memória de Acesso Aleatório): A RAM é usada para armazenar dados temporários, variáveis, e a pilha (*stack*). Ela oferece acesso rápido, permitindo leitura e escrita, sendo o principal local de armazenamento de dados em tempo de execução.

Memória ROM (Memória Somente de Leitura): A ROM contém o código imutável, como o firmware e rotinas essenciais para a inicialização do sistema. O processador pode apenas ler dados da ROM, que são carregados e executados conforme necessário.

2.5.1 Pilha (Stack) no RISC-V 32IM: A pilha é uma área da memória RAM usada para armazenar dados temporários, endereços de retorno e parâmetros de função durante a execução de um programa. No RISC-V 32IM, a pilha cresce no sentido de endereços mais baixos à medida que novos dados são empurrados (*push*) para ela. O ponteiro de pilha (`sp`) sempre aponta para o topo da pilha, que é o endereço da última posição utilizada.

A pilha é fundamental para o gerenciamento de chamadas de função e contextos de execução, permitindo que o processador armazene o estado de execução corrente e o recupere após a execução de sub-rotinas.

2.5.2 Acesso e Gerenciamento de Memória: O processador RISC-V 32IM acessa a memória através de instruções de leitura e escrita que manipulam endereços de memória. Instruções como `lw` (load word) e `sw` (store word) são usadas para carregar dados da RAM e armazenar resultados.

2.5.3 Implementação de Push e Pop: A implementação das operações *push* e *pop* na pilha em processadores RISC-V 32IM é realizada por meio da manipulação do ponteiro de pilha (stack pointer – `sp`, associado ao registrador `x2`). Quando um valor é empurrado (*push*) na pilha, o ponteiro de pilha é decrementado e o valor é armazenado no novo endereço apontado. No caso do *pop*, o valor é retirado da posição atual da pilha e o ponteiro de pilha é incrementado, restaurando o valor anterior.

Exemplo de uma operação de *push*:

```
addi sp, sp, -4   # Decrementa o ponteiro de pilha
sw t0, 0(sp)      # Armazena o valor de t0 no topo da pilha
```

Exemplo de uma operação de *pop*:

```
lw t0, 0(sp)      # Carrega o valor do topo da pilha para t0
addi sp, sp, 4    # Incrementa o ponteiro de pilha
```

2.5.4 Endianness: O processador RISC-V 32IM utiliza o formato *big-endian* para a organização dos bytes na memória. Isso significa que o byte mais significativo (MSB) de uma palavra é armazenado no endereço de memória mais baixo, enquanto o byte menos significativo (LSB) é armazenado no endereço mais alto. Por exemplo, se armazenarmos o valor `0x12345678` em uma posição de memória, ele será representado como:

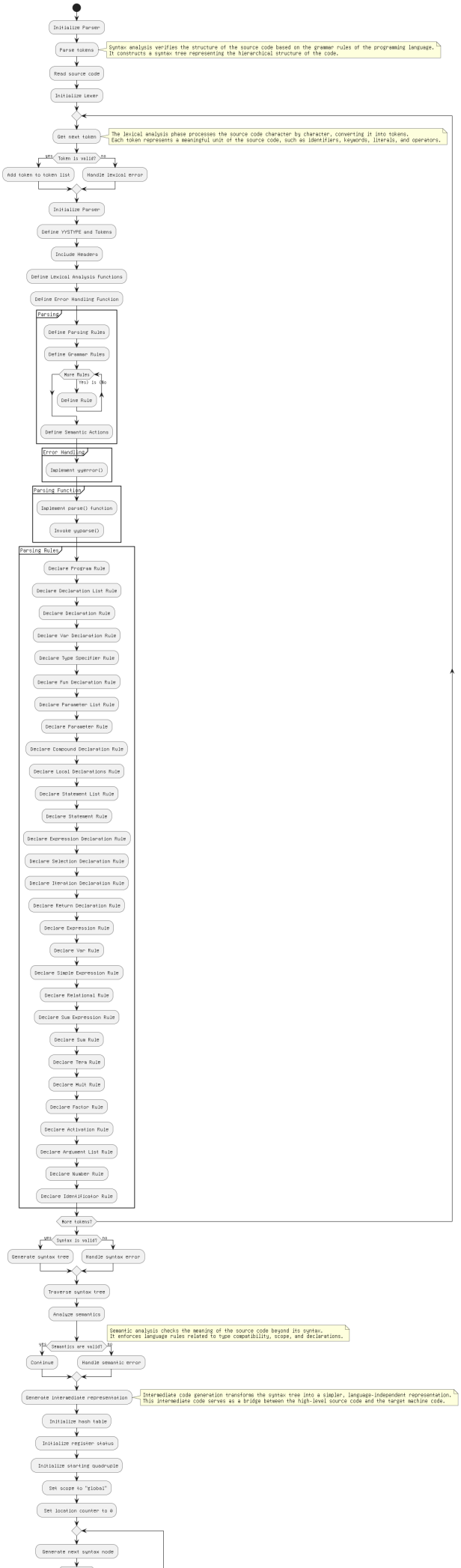
```
Endereço:   +0    +1    +2    +3
Valor:    0x12  0x34  0x56  0x78
```

3 Compilador: Fase de Análise

A fase de análise de um compilador é responsável por decompor e interpretar o código-fonte, transformando-o em uma representação interna que servirá de base para as próximas fases de compilação, como a geração de código intermediário e a otimização. Esta fase divide-se em três subfases principais: análise léxica, análise sintática e análise semântica, cada uma com seu papel específico na compreensão do programa a ser compilado.

3.1 Diagramas da Fase de Análise PUML

Segue, nessa etapa do relatório, as figuras que referenciam os diagramas de atividades e blocos:


Initialize Parser
Parse tokens
Syntax analysis verifies the structure of the source code based on the grammar rules of the programming language. It constructs a syntax tree representing the hierarchical structure of the code.
Read source code
Initialize Lexer
Get next token
The lexical analysis phase processes the source code character by character, converting it into tokens. Each token represents a meaningful unit of the source code, such as identifiers, keywords, literals, and operators.
Token is valid?
yes
no
Add token to token list
Handle lexical error
Initialize Parser
Define YYSTYPE and Tokens
Include Headers
Define Lexical Analysis Functions
Define Error Handling Function
Parsing
Define Parsing Rules
Define Grammar Rules
More Rules
Yes) is (No
Define Rule
Define Semantic Actions
Error Handling
Implement yyerror()
Parsing Function
Implement parse() function
Invoke yyparse()
Parsing Rules
Declare Program Rule
Declare Declaration List Rule
Declare Declaration Rule
Declare Var Declaration Rule
Declare Type Specifier Rule
Declare Fun Declaration Rule
Declare Parameter List Rule
Declare Parameter Rule
Declare Compound Declaration Rule
Declare Local Declarations Rule
Declare Statement List Rule
Declare Statement Rule
Declare Expression Declaration Rule
Declare Selection Declaration Rule
Declare Iteration Declaration Rule
Declare Return Declaration Rule
Declare Expression Rule
Declare Var Rule
Declare Simple Expression Rule
Declare Relational Rule
Declare Sum Expression Rule
Declare Sum Rule
Declare Term Rule
Declare Mult Rule
Declare Factor Rule
Declare Activation Rule
Declare Argument List Rule
Declare Number Rule
Declare Identificator Rule
More tokens?
Syntax is valid?
yes
no
Generate syntax tree
Handle syntax error
Traverse syntax tree
Analyze semantics
Semantic analysis checks the meaning of the source code beyond its syntax. It enforces language rules related to type compatibility, scope, and declarations.
Semantics are valid?
yes
no
Continue
Handle semantic error
Generate intermediate representation
Intermediate code generation transforms the syntax tree into a simpler, language-independent representation. This intermediate code serves as a bridge between the high-level source code and the target machine code.
Initialize hash table
Initialize register status
Initialize starting quadruple
Set scope to "global"
Set location counter to 0
Generate next syntax node

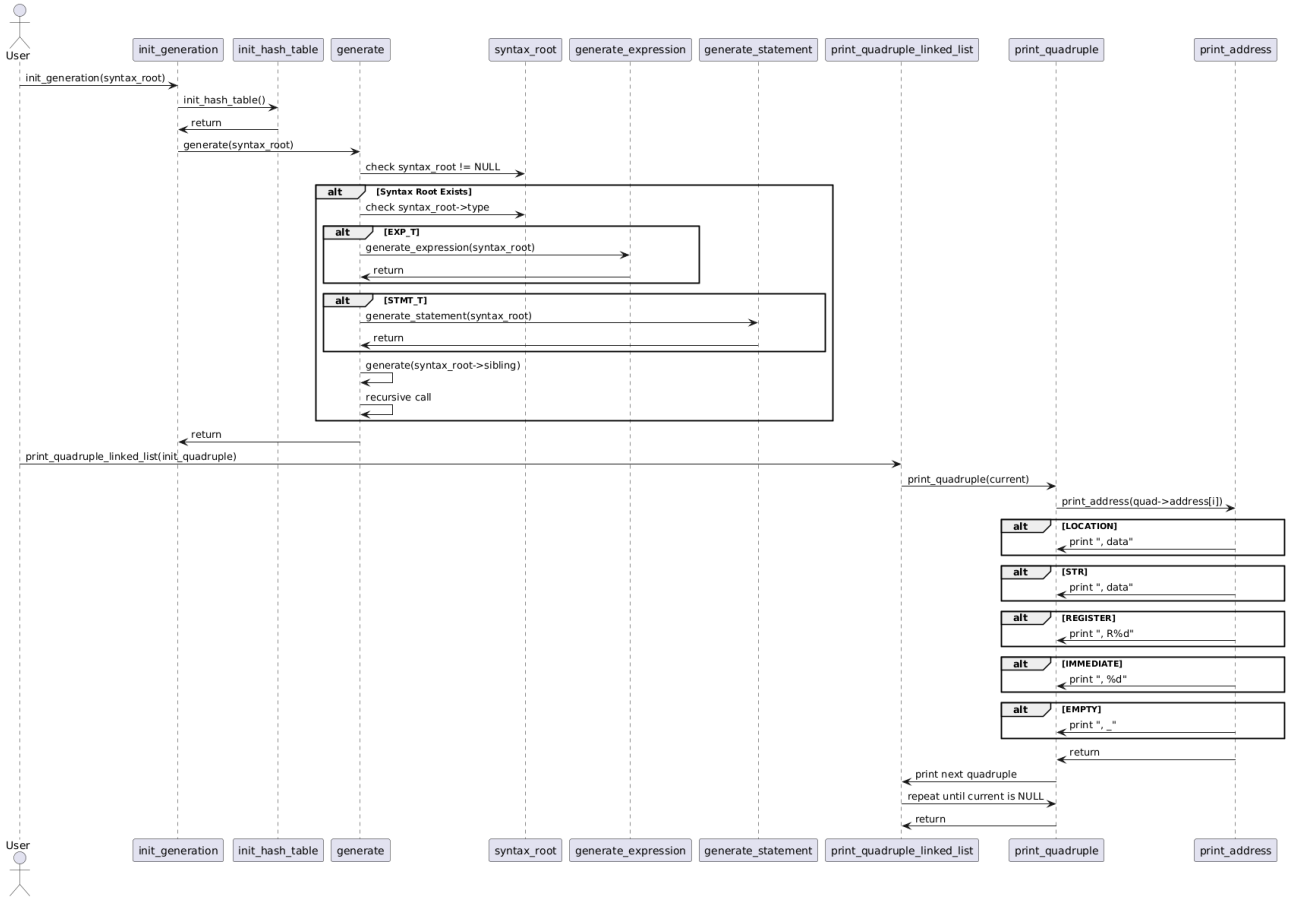


Figure 3: Enter Caption

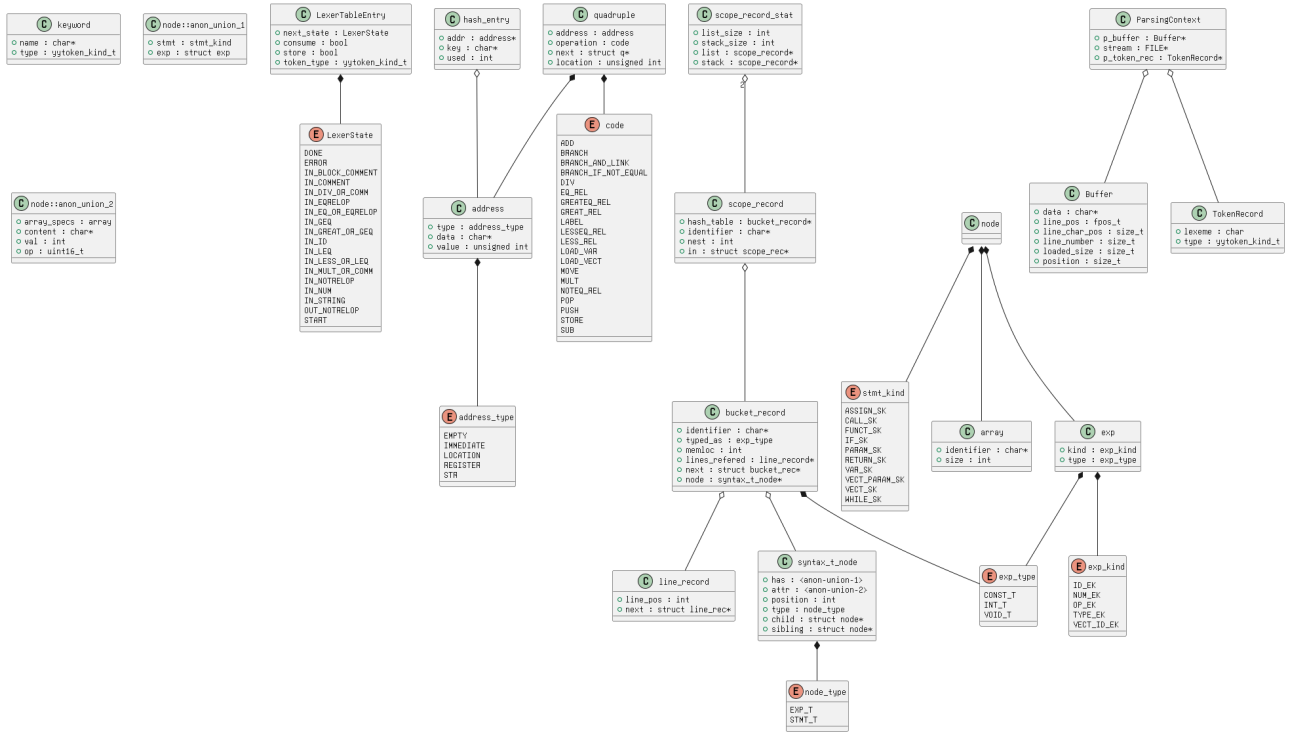


Figure 4: Bloco

3.2 Análise Léxica

A análise léxica é a primeira etapa da fase de análise. Sua função principal é ler o código-fonte como uma sequência de caracteres e agrupá-los em unidades sintaticamente significativas chamadas *tokens*. Exemplos de *tokens* incluem palavras-chave, identificadores, operadores e literais.

Nesta implementação, foi utilizado um *DFA* (*Deterministic Finite Automaton*) para cada possibilidade de estrutura léxica. A partir dessas autômatos, desenvolveu-se uma máquina de estados que permite identificar e classificar os *tokens* conforme as regras gramaticais do programa. Essa abordagem baseada em autômatos finitos garante que a leitura e identificação de *tokens* seja realizada de forma eficiente e precisa. Cada estado do autômato representa um avanço na identificação de um *token* específico, como palavras-chave ou operadores, e ao final, o autômato categoriza a sequência de caracteres adequadamente.

Por exemplo, a leitura de um identificador ou palavra-chave é tratada de forma semelhante, com a máquina de estados lendo uma sequência de caracteres alfabéticos e numéricos até atingir o final do *token*. Dessa maneira, o analisador léxico é capaz de diferenciar um identificador de uma palavra-chave com base na comparação de *tokens* com o conjunto de palavras reservadas.

A tabela `lexerTable`, descrita no código 1, define transições de estados para o lexer com o objetivo de reconhecer diferentes padrões lexicais. Através dessa lista de correlação entre letras, números e símbolos, o lexer é capaz de identificar a tipagem de cada lexema. Por exemplo, a parir do estado inicial, caracteres alfabéticos como `'a'... 'z'` e `'A'... 'Z'` são categorizados como identificadores (`ID`), enquanto números `'0'...'9'` são reconhecidos como tokens numéricos (`NUM`). Operadores como `'+'` e `'-'` são tratados como operadores aritméticos, o que facilita a correta classificação dos tokens durante a análise lexical.

Listing 1: Tabela de Transição para o Lexer

```
LexerTableEntry lexerTable[18][128] = {
    /* Definindo apenas os delimitadores relevantes */
    [START] = {
        /* Delimitadores e regras padr o */
        ['\0'...127] = { ERROR, YYerror, false, false },
        ['a'...'z'] = { IN_ID, ID, true, true },
        ['A'...'Z'] = { IN_ID, ID, true, true },
        ['0'...'9'] = { IN_NUM, NUM, true, true },
        /* Demais regras omitidas por brevidade */
    },
    /* Demais estados omitidos */
};
```

Para a gestão eficiente das palavras reservadas do compilador, como `while`, `if`, e outras, utilizei o ‘gperf‘ para gerar uma tabela de hash. O ‘gperf‘ é uma ferramenta que cria tabelas de hash otimizadas para pesquisa de strings, o que é crucial para identificar rapidamente palavras-chave no código fonte. O código de definição para o ‘gperf‘, apresentado a seguir, especifica uma estrutura de dados para armazenar as palavras reservadas e seus respectivos tipos de token. Isso permite uma rápida e eficiente correspondência de palavras reservadas durante a análise lexical:

Listing 2: Definição de palavras reservadas usando gperf

```
%language=ANSI-C
%readonly-tables
%compare-lengths
%includes
%struct-type
struct keyword { char *name; TokenType type; }
%%
```

```
8  if,    IF,
9  else,  ELSE,
10 int,   INT,
11 void,  VOID,
12 return, RETURN,
13 while, WHILE
```

3.3 Análise Sintática

Após a análise léxica, o compilador passa para a análise sintática. Aqui, a sequência de *tokens* gerada pela análise léxica é organizada de acordo com a gramática do programa, gerando uma árvore sintática, ou *parse tree*. A árvore sintática é uma representação hierárquica da estrutura do programa, onde cada nó corresponde a uma construção sintática, como expressões, comandos de controle de fluxo ou declarações.

Esta fase é essencial para garantir que o código-fonte siga as regras da linguagem de programação. Qualquer erro de sintaxe, como uma expressão malformada ou um bloco de controle sem uma chave de fechamento, é detectado aqui.

A árvore sintática gerada é crucial para o funcionamento do compilador, pois define a estrutura lógica do programa e a ordem de execução dos blocos de código. Essa árvore será posteriormente utilizada nas fases de otimização e geração de código intermediário. Durante a análise sintática, também são identificados blocos de fluxo de execução, como *loops*, condicionais e chamadas de função, que precisam ser organizados adequadamente para refletir a lógica do programa original.

Para implementar a análise sintática, utilizei o *bison*, uma ferramenta que facilita a criação de analisadores sintáticos para linguagens de programação. O *bison* utiliza uma gramática formal para definir a estrutura da linguagem e gerar o código necessário para realizar a análise. A gramática empregada é uma gramática livre de contexto (CFG), que é adequada para descrever a sintaxe de linguagens de programação. O uso do *bison* permite que o compilador interprete corretamente a estrutura do código-fonte, identificando e organizando os blocos de código de acordo com as regras definidas na gramática.

A estrutura de dados utilizada na análise sintática é essencial para representar a hierarquia e os elementos do programa. As definições a seguir são utilizadas para criar e manipular a árvore sintática:

Listing 3: Estruturas de dados que definem o programa

```
1  #ifndef __PARSER_H_
2  #define __PARSER_H_
3
4  #include "libs.h"
5
6  /*--[Sintax Tree Definitions]--*/
7
8  typedef enum {STMT_T, EXP_T} node_type;
9  typedef enum {IF_SK, WHILE_SK, RETURN_SK, ASSIGN_SK/*=2*/, VAR_SK, VECT_SK,
       FUNCT_SK, CALL_SK, PARAM_SK, VECT_PARAM_SK} stmt_kind;
10 typedef enum {OP_EK, ID_EK, NUM_EK, TYPE_EK/*ie. int decl*/, VECT_ID_EK/*type
       vector[size]*/} exp_kind;
11 typedef enum {INT_T, VOID_T, CONST_T} exp_type;
12 struct exp {exp_kind kind; exp_type type;};
13
14 #define MAXCHILDREN 3 // max of three expressions under each stmt
15 typedef struct arr{
16     int size;
17     char* identifier;
```

```
18 } array;
19 /*--[Sintax Tree Structure - used also in semantic analysis]--*/
20 typedef struct node
21 {
22     struct node * child[MAXCHILDREN];
23     struct node * sibling;
24     int position[2]; //line position e char position resp
25     node_type type;
26     union {stmt_kind stmt; struct exp exp; } has; // node has a stmt or an exp
27     union { uint16_t op;/*tok type*/ int val;/*value assign*/ array array_specs;
           char* content;/*content of*/} attr; // used in semantic analysis
28
29 } syntax_t_node;
30
31 syntax_t_node* new_stmt_node(stmt_kind);
32 syntax_t_node* new_exp_node(exp_kind);
33 char* cp_str(char*);
34 void print_syntax_tree(syntax_t_node*);
35 extern syntax_t_node* parse(void);
36
37 #endif
```

As estruturas de dados definidas, como `syntax_t_node`, `stmt_kind`, e `exp_kind`, são fundamentais para a análise sintática e também desempenham um papel crucial na geração de código intermediário. Essas estruturas armazenam informações sobre a hierarquia do programa, os tipos de declarações e expressões, bem como os atributos associados a cada nó da árvore sintática. A definição de `syntax_t_node` permite a criação de nós da árvore que podem representar declarações (`stmt_kind`) ou expressões (`exp_kind`), e o uso de uniões (`union`) permite que cada nó armazene diferentes tipos de informações necessárias para a análise semântica e para a geração do código intermediário.

Na fase de geração de código intermediário, essas estruturas serão utilizadas para traduzir a representação hierárquica do programa em uma forma intermediária que facilita a tradução final para o código de máquina. Portanto, as definições que determinam a funcionalidade, a operacionalidade e a estrutura do programa são essenciais para garantir que o compilador produza um código eficiente e correto.

3.4 Análise Semântica

A análise semântica vem logo após a sintática e visa garantir que o programa tenha um significado correto, de acordo com as regras da linguagem. Enquanto a análise sintática verifica a estrutura, a análise semântica verifica aspectos como tipos de dados e escopo de variáveis. Erros como a tentativa de realizar operações aritméticas com tipos incompatíveis ou o uso de variáveis não declaradas são capturados nesta fase.

Embora a análise semântica seja crucial para o funcionamento correto do programa, na prática, a geração de código intermediário depende, principalmente, da árvore sintática já construída. A árvore de análise sintática fornece informações suficientes para que o compilador possa gerar o código intermediário, que será refinado nas fases subsequentes, como a otimização e a geração de código final.

4 Compilador: Fase de Síntese

4.1 Diagramas de Bloco de Fase de Síntese PUML

Segue nessa sessão os diagramas correspondentes a fase de Síntese:

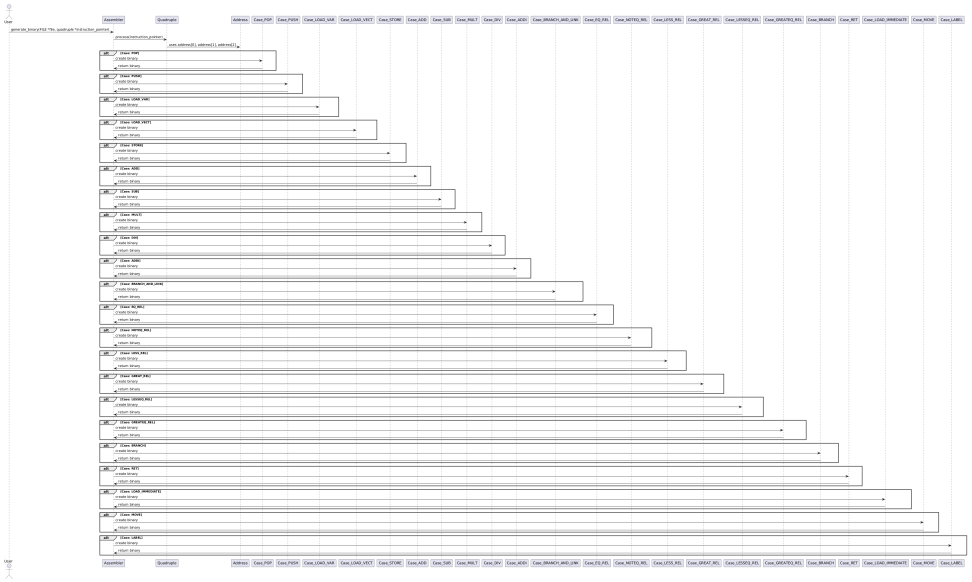


Figure 5: Atividades - Assembler

4.2 Geração do Código Intermediário

A geração do código intermediário é uma etapa crucial no processo de compilação. Durante esta fase, o compilador traduz a árvore sintática em um formato intermediário que pode ser otimizado e eventualmente convertido em código de máquina. Este formato intermediário frequentemente utiliza quádruplas, que são representações abstratas das operações que o código final deve executar. A seguir, estão os tipos de quádruplas definidos e suas respectivas descrições:

- `LOAD_VAR` (256): Carrega uma variável para um registrador. Os parâmetros são `reg_addr` (registrador de destino), `scope` (escopo da variável) e `name` (nome da variável).
- `LOAD_VECT` (257): Carrega um valor de um vetor para um registrador. Os parâmetros são `reg_addr` (registrador de destino), `add_result_sizeaddr` (tamanho e endereço) e `NULL`.
- `BRANCH_IF` (258): Realiza um desvio condicional com base em uma condição. Os parâmetros são `NULL`, `condition` (condição) e `inst_addr` (endereço da instrução de desvio).
- `BRANCH` (259): Realiza um desvio incondicional. Os parâmetros são `NULL`, `NULL` e `inst_addr` (endereço da instrução de desvio).
- `LABEL` (260): Marca um ponto no código com um rótulo. Os parâmetros são `id` (identificador do rótulo), `NULL` e `NULL`.
- `MOVE` (261): Move um valor de um registrador para outro. Os parâmetros são `dest` (registrador de destino), `NULL` e `data` (dados a serem movidos).
- `PUSH` (262): Empurra um valor na pilha. O parâmetro é `reg` (registrador ou dados a serem empurrados para a pilha).
- `POP` (263): Retira um valor da pilha para um registrador. O parâmetro é `dest` (registrador de destino).
- `BRANCH_AND_LINK` (264): Realiza um desvio com link para uma função. O parâmetro é `inst_addr` (endereço da instrução de desvio).

- `STORE` (265): Armazena um valor de um registrador na memória. Os parâmetros são `reg` (registrador a ser armazenado), `scope` (escopo) e `offset_mem` (offset na memória).
- `ADD` (266): Adiciona dois valores e armazena o resultado. Os parâmetros são `receivesresult` (registrador de destino) e os dois valores a serem somados.
- `SUB` (267): Subtrai dois valores e armazena o resultado. Os parâmetros são `dest` (registrador de destino) e os dois valores a serem subtraídos.
- `MULT` (268): Multiplica dois valores e armazena o resultado. Os parâmetros são `dest` (registrador de destino) e os dois valores a serem multiplicados.
- `DIV` (269): Divide dois valores e armazena o resultado. Os parâmetros são `dest` (registrador de destino) e os dois valores a serem divididos.
- `EQ_REL` (270): Realiza uma comparação de igualdade. Os parâmetros são `reg1` e `reg2`, e o endereço relativo para desvio.
- `NOTEQ_REL` (271): Realiza uma comparação de desigualdade. Os parâmetros são `reg1` e `reg2`, e o endereço relativo para desvio.
- `LESSEQ_REL` (272): Realiza uma comparação de menor ou igual. Os parâmetros são `reg1` e `reg2`, e o endereço relativo para desvio.
- `GREATEQ_REL` (273): Realiza uma comparação de maior ou igual. Os parâmetros são `reg1` e `reg2`, e o endereço relativo para desvio.
- `GREAT_REL` (274): Realiza uma comparação de maior. Os parâmetros são `reg1` e `reg2`, e o endereço relativo para desvio.
- `LESS_REL` (275): Realiza uma comparação de menor. Os parâmetros são `reg1` e `reg2`, e o endereço relativo para desvio.
- `RET` (276): Executa uma instrução de retorno.
- `LOAD_IMMEDIATE` (277): Carrega um valor imediato em um registrador. Os parâmetros são `reg` (registrador) e o valor imediato.
- `ADDI` (278): Adiciona um valor imediato a um registrador. Os parâmetros são `dest` (registrador de destino), `src` (registrador fonte) e o valor imediato.
- `GLOB_MAIN` (279): Declara um símbolo global principal. O parâmetro é o nome do símbolo global.

4.3 Geração do Código Assembly

A fase de geração de código Assembly é uma etapa crucial no processo de compilação, onde o código intermediário é traduzido em instruções Assembly específicas da arquitetura do processador. A função ‘assemble‘ realiza essa tradução, gerando um arquivo de código Assembly a partir de uma lista de quádruplas. A seguir, descrevemos o funcionamento desta função com exemplos ilustrativos.

4.3.1 app.s: A função começa abrindo um arquivo para escrita, onde o código Assembly será armazenado. Se ocorrer um erro ao abrir o arquivo, a função imprime uma mensagem de erro e encerra:

```
FILE *file = fopen("app.s", "w");
if (file == NULL) {
    perror("Error opening file");
    return;
}
```

4.3.2 Iteração sobre as Quádruplas: Em seguida, a função itera sobre cada quádrupla na lista. Cada quádrupla representa uma instrução ou operação a ser traduzida em Assembly. Dependendo da operação especificada na quádrupla, a função gera o código Assembly correspondente. A seguir, descrevemos os diferentes tipos de operações e o código Assembly gerado para cada uma delas.

4.3.3 Carregamento de Variáveis e Vetores: Para operações de carregamento, como LOAD_VAR e LOAD_VECT, a função gera instruções para carregar valores da memória para registradores:

```
case LOAD_VAR:
    fprintf(file, "lw x%d, %d(x%d)\n",
        instruction_pointer->address[0].value,
        (instruction_pointer->address[2].value + 1) * -4,
        instruction_pointer->address[1].value);
    break;
```

No exemplo acima, a instrução 'lw' carrega um valor da memória, especificado pelo deslocamento e registrador base, para o registrador destino.

```
case LOAD_VECT:
    fprintf(file, "lw x%d, %d(s0)\n",
        instruction_pointer->address[0].value, 9);
    break;
```

Para LOAD_VECT, um deslocamento fixo é usado para carregar o valor de um vetor.

4.3.4 Desvios Condicionais: Desvios condicionais, como EQ_REL, NOTEQ_REL, LESSEQ_REL, GREATEQ_REL, GREAT_REL, e LESS_REL, a função gera instruções de desvio condicional baseadas em comparações entre registradores.

```
case EQ_REL:
    fprintf(file, "beq x%d, x%d, %s\n",
        instruction_pointer->address[1].value,
        instruction_pointer->address[2].value,
        relative_loc);
    break;
```

A instrução beq realiza um desvio condicional se dois registradores forem iguais, e relative_loc é o endereço para o desvio.

4.3.5 Desvios Incondicionais e Rótulos: Para desvios incondicionais e rótulos, a função gera instruções como 'j' para desvio e marca pontos no código com rótulos:

```
case BRANCH:
    fprintf(file, "j %s\n", instruction_pointer->address[2].data);
    break;
```

```
case LABEL:
    fprintf(file, "%s:\n", instruction_pointer->address[0].data);
    break;
```

Aqui, ‘j‘ realiza um desvio incondicional, e o rótulo é usado para identificar pontos específicos no código Assembly.

4.3.6 Movimentação e Pilha: Para operações de movimentação e manipulação de pilha, como ‘MOVE‘, ‘PUSH‘, e ‘POP‘, o código Assembly é gerado para mover valores entre registradores e gerenciar a pilha:

```
case MOVE:
    fprintf(file, "mv x%d, x%d\n",
        instruction_pointer->address[0].value,
        instruction_pointer->address[1].value);
    break;
```

```
case PUSH:
    fprintf(file, "addi x2, x2, -4\n");
    fprintf(file, "sw x%d, 0(x2)\n", instruction_pointer->address[0].value);
    break;
```

```
case POP:
    fprintf(file, "lw x%d, 0(x2)\n", instruction_pointer->address[0].value);
    fprintf(file, "addi x2, x2, 4\n");
    break;
```

As instruções ‘mv‘, ‘addi‘, e ‘sw‘ são usadas para movimentar valores e ajustar a pilha.

4.3.7 Chamada e Retorno de Funções: Para chamadas de função e retornos, a função gera instruções ‘jal‘ e ‘ret‘:

```
case BRANCH_AND_LINK:
    fprintf(file, "jal %s\n", instruction_pointer->address[2].data);
    break;
```

```
case RET:
    fprintf(file, "ret\n");
    break;
```

A instrução ‘jal‘ realiza uma chamada de função com link, enquanto ‘ret‘ realiza o retorno da função.

4.3.8 Operações Aritméticas: Operações aritméticas como adição, subtração, multiplicação e divisão são geradas da seguinte forma:

```
case ADDI:
    fprintf(file, "addi x%d, x%d, %d\n",
        instruction_pointer->address[0].value,
        instruction_pointer->address[1].value,
        instruction_pointer->address[2].value);
    break;
```

```
case ADD:
    fprintf(file, "add x%d, x%d, x%d\n",
        instruction_pointer->address[0].value,
        instruction_pointer->address[1].value,
        instruction_pointer->address[2].value);
    break;
```

```
case SUB:
    fprintf(file, "sub x%d, x%d, x%d\n",
        instruction_pointer->address[0].value,
        instruction_pointer->address[1].value,
        instruction_pointer->address[2].value);
    break;
```

```
case MULT:
    fprintf(file, "mul x%d, x%d, x%d\n",
        instruction_pointer->address[0].value,
        instruction_pointer->address[1].value,
        instruction_pointer->address[2].value);
    break;
```

```
case DIV:
    fprintf(file, "div x%d, x%d, x%d\n",
        instruction_pointer->address[0].value,
        instruction_pointer->address[1].value,
        instruction_pointer->address[2].value);
    break;
```

Cada operação aritmética é traduzida em uma instrução Assembly correspondente que realiza a operação desejada.

4.3.9 Carregamento de Valores Imediatos e Declaração de Globais: Por fim, a função também gera instruções para carregar valores imediatos e declarar símbolos globais:

```
case LOAD_IMMEDIATE:
    fprintf(file, "li x%d, %d\n",
        instruction_pointer->address[0].value,
        instruction_pointer->address[2].value);
    break;
```

```
case GLOB_MAIN:
    fprintf(file, ".globl %s\n",
        instruction_pointer->address[2].data);
    break;
```

A instrução 'li' carrega um valor imediato em um registrador, enquanto '.globl' declara um símbolo global.

— Esta função completa a tradução do código intermediário em Assembly, pronto para ser assemblado e executado. Esta seção fornece uma visão geral detalhada de como a função 'assemble' converte quádruplas em código Assembly, com exemplos específicos que ilustram o processo para diferentes tipos de operações.

4.4 Geração do Código Executável

A geração do código executável é a fase final onde o código Assembly gerado é convertido em um formato binário que pode ser executado pela máquina. Este processo envolve a vinculação de endereços e a criação de um arquivo executável que pode ser carregado e executado pelo sistema operacional NUGGET_OS.

Nesta seção, vamos explorar a função `generate_binary`, que realiza essa tarefa. Dividiremos a função em partes para explicar sua estrutura e funcionamento.

4.4.1 Estrutura Geral da Função: A função `generate_binary` é responsável por converter uma série de instruções para seu formato binário correspondente e gravar essas instruções em um arquivo. A seguir, vamos detalhar cada parte da função.

```
1 void generate_binary(FILE *file, quadruple *instruction_pointer) {
2     uint32_t binary = 0;
```

Neste trecho inicial, a função recebe um ponteiro para um arquivo `FILE` e um ponteiro para a estrutura `quadruple` que representa a instrução. A variável `binary` é inicializada para armazenar a representação binária da instrução.

A função utiliza uma estrutura de controle `switch` para tratar diferentes tipos de operações, que são especificadas pelo campo `operation` da estrutura `quadruple`. A seguir, explicamos como cada tipo de operação é processado.

4.4.2 Operações de Pilha: Para operações de pilha como `POP` e `PUSH`, a função gera instruções que interagem com a memória para carregar ou armazenar valores e ajustar o ponteiro da pilha.

```
1     switch (instruction_pointer->operation) {
2         case POP:
3             binary = (0x03 << 0) |                              // opcode for LOAD
4                      (instruction_pointer->address[0].value << 7) |  // rd
5                      (0x2 << 12) |                              // funct3 for LW
6                      (2 << 15) | // rs1
7                      (0 << 20);
8             binary = htobe32(binary);
9             fwrite(&binary, sizeof(binary), 1, file);
10            binary = ( 4 << 20) |  // imm
11                     (2 << 15) |  // rs1
12                     (0x0 << 12) |                                    // funct3
                          for ADDI
13                     (2 << 7) |   // rd
14                     (0x13);  // opcode for I-type instructions
15
16            break;
17        case PUSH:
18            binary = ( 4 << 20) |  // imm
19                     (2 << 15) |  // rs1
20                     (0x0 << 12) |                                    // funct3
                          for ADDI
21                     (2 << 7) |   // rd
22                     (0x13);  // opcode for I-type instructions
23            binary = htobe32(binary);
24            fwrite(&binary, sizeof(binary), 1, file);
25
26            binary = (((instruction_pointer->address[2].value + 1) * -4 & 0xFE0)
                  << 25) |  // imm[11:5]
27                     (instruction_pointer->address[0].value << 20) |
                                                 // rs2
28                     (2 << 15) |                         // rs1
29                     (0x2 << 12) |

                          // funct3 for SW
30                     ((0 ) << 7) |    // imm[4:0]
31                     (0x23);  // opcode for STORE
32
33            break;
34    }
```

Para a operação `POP`, a função gera uma instrução para carregar dados da memória e outra para ajustar o ponteiro da pilha. Para a operação `PUSH`, são geradas instruções para armazenar dados na memória e ajustar o ponteiro da pilha.

4.4.3 Operações de Carregamento e Armazenamento: Instruções que envolvem carregamento e armazenamento de variáveis e vetores são processadas para gerar instruções de carregamento (LOAD_VAR, LOAD_VECT).

```
1         case LOAD_VAR:
2             // lw x[rd], offset(x[rs1])
3             binary = (0x03) |                                  // opcode for LOAD
4                      (instruction_pointer->address[0].value << 7) |  // rd
5                      (0x2 << 12) |                              // funct3 for LW
6                      (instruction_pointer->address[1].value << 15) | // rs1
7                      ((instruction_pointer->address[2].value + 1) * -4 << 20);
                           // imm
8
9             break;
10        case LOAD_VECT:
11            // lw x[rd], offset(s0)
12            binary = (0x03) |                                  // opcode for LOAD
13                     (instruction_pointer->address[0].value << 7) |  // rd
14                     (0x2 << 12) |                              // funct3 for LW
15                     (16 << 15) |                               // s0 (x16)
16                     (9 << 20);                                 // imm (9 in this
                           case)
17
18            break;
```

Aqui, a função gera instruções para carregar dados da memória para os registradores. O tipo de instrução depende do formato e do endereço especificado na instrução Assembly.

4.4.4 Operações Aritméticas: As operações aritméticas como ADD, SUB, MULT, e DIV são processadas para gerar instruções de adição, subtração, multiplicação e divisão entre registradores.

```
1         case ADD:
2             // add x[rd], x[rs1], x[rs2]
3             binary = (0x00 << 25) |                                  // funct7
                  for ADD
4                      (instruction_pointer->address[2].value << 20) |  // rs2
5                      (instruction_pointer->address[1].value << 15) |  // rs1
6                      (0x0 << 12) |                                  // funct3
                           for ADD
7                      (instruction_pointer->address[0].value << 7) |   // rd
8                      (0x33);  // opcode for R-type instructions
9
10            break;
11        case SUB:
12            // sub x[rd], x[rs1], x[rs2]
13            binary = (0x20 << 25) |                                  // funct7
                  for SUB
14                     (instruction_pointer->address[2].value << 20) |  // rs2
15                     (instruction_pointer->address[1].value << 15) |  // rs1
16                     (0x0 << 12) |                                  // funct3
                           for SUB
17                     (instruction_pointer->address[0].value << 7) |   // rd
18                     (0x33);  // opcode for R-type
19
20            break;
21        case MULT:
22            // mul x[rd], x[rs1], x[rs2]
23            binary = (0x01 << 25) |                                  // funct7
                  for MUL
```

```
24                      (instruction_pointer->address[2].value << 20) |  // rs2
25                      (instruction_pointer->address[1].value << 15) |  // rs1
26                      (0x0 << 12) |                                     // funct3
                            for MUL
27                      (instruction_pointer->address[0].value << 7) |   // rd
28                      (0x33);  // opcode for R-type
29
30              break;
31          case DIV:
32              // div x[rd], x[rs1], x[rs2]
33              binary = (0x1 << 25) |                                  // funct7 for
                    DIV
34                      (instruction_pointer->address[2].value << 20) |  // rs2
35                      (instruction_pointer->address[1].value << 15) |  // rs1
36                      (0x4 << 12) |                                     // funct3
                            for DIV
37                      (instruction_pointer->address[0].value << 7) |   // rd
38                      (0x33);  // opcode for R-type
39
40              break;
```

Cada operação aritmética é convertida para uma instrução binária de formato R, onde os campos incluem os registradores de origem e destino, além do código de operação.

4.4.5 Operações Imediatas e Controle de Fluxo: Instruções envolvendo valores imediatos e controle de fluxo, como `ADDI`, `BRANCH_AND_LINK`, e instruções de comparação, são processadas para gerar o código binário apropriado.

```
1           case ADDI:
2               // addi x[rd], x[rs1], imm
3               binary = (instruction_pointer->address[2].value << 20) |  // imm
4                         (instruction_pointer->address[1].value << 15) |  // rs1
5                         (0x0 << 12) |                                     // funct3
                              for ADDI
6                         (instruction_pointer->address[0].value << 7) |   // rd
7                         (0x13);  // opcode for I-type instructions
8
9               break;
10          case BRANCH_AND_LINK:
11              // jal x[rd], imm
12              binary = ((instruction_pointer->address[2].value & 0xFFFFF) << 12) |
                      // imm
13                        (0x6F);  // opcode for JAL
14
15              break;
16          case RET:
17              // ret
18              // Equivalent to jalr x0, x1, 0
19              binary = (0x0 << 20) |  // imm = 0
20                        (1 << 15) |   // rs1 = x1 (return address)
21                        (0x0 << 12) |  // funct3
22                        (0 << 7) |     // rd = x0 (discard result)
23                        (0x67);        // opcode for JALR
24
25              break;
```

4.5 Gerenciamento de Memória em RISC-V 32IM

O gerenciamento de memória é crucial para garantir que o compilador aloque e libere corretamente a memória durante a execução do código. Em sistemas baseados na arquitetura RISC-V 32IM, este gerenciamento envolve várias técnicas, incluindo o uso de ponteiros de quadro, registros de retorno e alocação assíncrona de pilha. Vamos explorar como esses conceitos são aplicados e gerenciados, usando o código Assembly para a função `gcd` como exemplo.

4.5.1 Estrutura da Pilha e Registro de Retorno: Em RISC-V, a pilha é usada para armazenar variáveis locais e estados de função, como registros temporários e parâmetros. A função `gcd` ilustra como gerenciar a pilha e os registros durante a execução.

4.5.2 Ponteiro de Quadro (Frame Pointer): O ponteiro de quadro (x8, também conhecido como `fp`) é utilizado para facilitar o acesso às variáveis locais e parâmetros de função. No início da função, o ponteiro de quadro é salvo na pilha, e o novo ponteiro de quadro é configurado para o topo da pilha atual.

4.5.3 Registro de Retorno: O registro de retorno (x1, também conhecido como `ra`) armazena o endereço para o qual a execução deve retornar após a conclusão da função. Antes de chamar outra função ou retornar, o valor de `ra` deve ser preservado e restaurado.

4.5.4 Alocação de Parâmetros e Stack: Durante a execução da função, a memória da pilha é usada para alocar espaço para variáveis temporárias e parâmetros. O gerenciamento assíncrono da pilha é alcançado ajustando o ponteiro da pilha (x2) para alocar e desalocar espaço conforme necessário.

4.6 Exemplo Alocação de memória e retorno Assembly: Função `gcd`

O código a seguir mostra um exemplo de função `gcd` em Assembly que utiliza o gerenciamento de memória e o ponteiro de quadro para calcular o máximo divisor comum. O código ilustra como os parâmetros são salvos e recuperados, como a pilha é gerenciada e como o retorno é tratado.

```
1  gcd:
2      addi x2, x2, -4           # Aloca espa o para x1 na pilha
3      sw x1, 0(x2)              # Salva o valor de x1 na pilha
4      addi x2, x2, -4           # Aloca espa o para x8 na pilha
5      sw x8, 0(x2)              # Salva o valor de x8 na pilha
6      mv x8, x2                 # Atualiza o ponteiro de quadro
7      addi x2, x2, -8           # Aloca espa o para x10 e x11,
8                                # caso mais vars locais estivessem definidas
9                                #, um valor maior seria adicionado
10     sw x10, -4(x8)            # Salva x10 na pilha
11     sw x11, -8(x8)            # Salva x11 na pilha
12
13 [...]
14     mv x2, x8                 # Restaura o ponteiro de quadro
15     lw x8, 0(x2)              # Restaura x8
16     addi x2, x2, 4            # Ajusta o ponteiro da pilha
17     lw x1, 0(x2)              # Restaura x1
18     addi x2, x2, 4            # Ajusta o ponteiro da pilha
19     mv x10, x28               # Restaura x10
20     ret                       # Retorna da fun   o
21
```

```
22 [...]
23     mv x2, x8                # Restaura o ponteiro de frame
24     lw x8, 0(x2)             # Restaura x8
25     addi x2, x2, 4           # Ajusta o ponteiro da pilha
26     lw x1, 0(x2)             # Restaura x1
27     addi x2, x2, 4           # Ajusta o ponteiro da pilha
28     mv x10, x26              # Restaura x10
29     ret                      # Retorna da fun   o
```

O gerenciamento de memória em RISC-V 32IM envolve a configuração e uso de ponteiros de quadro, o gerenciamento de registros de retorno e a alocação de espaço na pilha de forma assíncrona. O código da função `gcd` exemplifica como esses elementos são utilizados para garantir que a memória seja alocada e gerenciada corretamente durante a execução da função. A implementação eficaz desses conceitos contribui para a estabilidade e a eficiência do código compilado, garantindo que os recursos de memória sejam utilizados de maneira adequada.

5 Exemplos

Nesta subseção, serão apresentados três exemplos de uso do compilador para ilustrar o processo de transformação do código fonte em código executável. Cada exemplo incluirá o código fonte, o código intermediário gerado, o código assembly resultante, e o código executável final. Também serão discutidas as correspondências entre o código fonte e o código intermediário, bem como entre o código intermediário e o código assembly.

5.1 Exemplo 1 : GCD

5.1.1 a. Código Fonte: A seguir, mostramos o uso do comilador em um código c- de programa que calcula o mdc segundo o algoritmo de Euclides.

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

5.1.2 b. Código Intermediário: A partir desse, obtemos as seguintes quadruplas

```
1 0::     GLOB_MAIN, _, _, main
2 1::     LABEL, gcd, _, _
3 2::     PUSH, R1, _, _
4 3::     PUSH, R8, _, _
5 4::     MOVE, R8, R2, _
6 5::     ADDI, R2, R2, -8
7 6::     STORE, R10, R8, gcd_u
8 7::     STORE, R11, R8, gcd_v
```

```
8::     LOAD_VAR, R28, R8, gcd_v
9::     LOAD_IMMEDIATE, R27, _, 0
10::    NOTEQ_RELOP, _, R28, R27
11::    BRANCH_IF, _, _, .L19
12::    LOAD_VAR, R28, R8, gcd_u
13::    MOVE, R2, R8, _
14::    POP, R8, _, _
15::    POP, R1, _, _
16::    MOVE, R10, R28, _
17::    RET, _, _, _
18::    BRANCH, _, _, .L37
19::    LABEL, .L19, _, _
20::    LOAD_VAR, R27, R8, gcd_v
21::    MOVE, R10, R27, _
22::    LOAD_VAR, R26, R8, gcd_u
23::    LOAD_VAR, R25, R8, gcd_u
24::    LOAD_VAR, R24, R8, gcd_v
25::    DIV_PRE_ALOP, R22, R25, R24
26::    LOAD_VAR, R25, R8, gcd_v
27::    MULT_PRE_ALOP, R24, R22, R25
28::    MINUS_ALOP, R25, R26, R24
29::    MOVE, R11, R25, _
30::    BRANCH_AND_LINK, _, _, gcd
31::    MOVE, R26, R10, _
32::    MOVE, R2, R8, _
33::    POP, R8, _, _
34::    POP, R1, _, _
35::    MOVE, R10, R26, _
36::    RET, _, _, _
37::    LABEL, .L37, _, _
38::    LABEL, main, _, _
39::    PUSH, R1, _, _
40::    PUSH, R8, _, _
41::    MOVE, R8, R2, _
42::    ADDI, R2, R2, -8
43::    BRANCH_AND_LINK, _, _, input
44::    MOVE, R28, R10, _
45::    STORE, R28, R8, main_x
46::    BRANCH_AND_LINK, _, _, input
47::    MOVE, R28, R10, _
48::    STORE, R28, R8, main_y
49::    LOAD_VAR, R28, R8, main_x
50::    MOVE, R10, R28, _
51::    LOAD_VAR, R27, R8, main_y
52::    MOVE, R11, R27, _
53::    BRANCH_AND_LINK, _, _, gcd
54::    MOVE, R26, R10, _
55::    MOVE, R10, R26, _
56::    BRANCH_AND_LINK, _, _, output
57::    MOVE, R2, R8, _
58::    POP, R8, _, _
59::    POP, R1, _, _
0::     RET, _, _, _
}
```

5.1.3 c. Código Assembly: Essas quadruplas geram o seguinte código em assembly. Note que para que se torne possível que o NUGGET_OS possa se encontrar a função main, a seção global é definida.

```
.globl main
```

```
2
3  # Fun  o gcd (M ximo Divisor Comum)
4  gcd:
5      # Prepara  o da pilha
6      addi x2, x2, -4          # Reserva espa o na pilha para o valor de 'u'
7      sw x1, 0(x2)             # Armazena o valor de 'u' em x1 na pilha
8      addi x2, x2, -4          # Reserva espa o na pilha para o frame pointer
9      sw x8, 0(x2)             # Armazena o frame pointer antigo na pilha
10     mv x8, x2                # Atualiza o frame pointer para o topo atual da
           pilha
11     addi x2, x2, -8          # Reserva espa o para 'v' e vari veis
           tempor rias
12     sw x10, -4(x8)           # Armazena o valor de 'v' em x10 na pilha
13     sw x11, -8(x8)           # Armazena uma vari vel tempor ria na pilha
14
15     # Verifica se 'v'    zero
16     lw x28, -8(x8)           # Carrega 'v' da pilha
17     li x27, 0                # Carrega 0 em x27
18     bne x28, x27, .L19      # Se 'v' != 0, pula para .L19
19
20     # Se 'v' == 0, retorna 'u'
21     lw x28, -4(x8)           # Carrega 'u' da pilha
22     mv x2, x8                # Restaura o frame pointer
23     lw x8, 0(x2)             # Carrega o frame pointer antigo
24     addi x2, x2, 4           # Ajusta o ponteiro da pilha
25     lw x1, 0(x2)             # Carrega o valor de 'u'
26     addi x2, x2, 4           # Ajusta o ponteiro da pilha
27     mv x10, x28              # Define o valor de retorno (x10) como 'u'
28     ret                      # Retorna da fun  o gcd
29
30     j .L37                  # Pular para o r tulo .L37 (fim da fun  o gcd)
31
32 .L19:
33     # Calcula gcd(v, u % v)
34     lw x27, -8(x8)           # Carrega 'v' da pilha para x27
35     mv x10, x27              # Atualiza o valor de retorno (x10) com 'v'
36     lw x26, -4(x8)           # Carrega 'u' da pilha
37     lw x25, -4(x8)           # Carrega 'u' novamente
38     lw x24, -8(x8)           # Carrega 'v' da pilha
39     div x22, x25, x24        # Calcula u / v, resultado em x22
40     lw x25, -8(x8)           # Carrega 'v' da pilha novamente
41     mul x24, x22, x25        # Calcula (u / v) * v, resultado em x24
42     sub x25, x26, x24        # Calcula u % v = u - (u / v) * v, resultado em x25
43     mv x11, x25              # Armazena o resto em x11
44     jal gcd                  # Chama recursivamente a fun  o gcd
45     mv x26, x10              # Atualiza x26 com o valor retornado
46     mv x2, x8                # Restaura o frame pointer
47     lw x8, 0(x2)             # Carrega o frame pointer antigo
48     addi x2, x2, 4           # Ajusta o ponteiro da pilha
49     lw x1, 0(x2)             # Carrega o valor de 'u'
50     addi x2, x2, 4           # Ajusta o ponteiro da pilha
51     mv x10, x26              # Define o valor de retorno (x10) com o resultado
           da chamada recursiva
52     ret                      # Retorna da fun  o gcd
53
54 .L37:
55     # Fun  o main
56     addi x2, x2, -4          # Reserva espa o na pilha para 'x'
57     sw x1, 0(x2)             # Armazena o valor de 'x' em x1 na pilha
58     addi x2, x2, -4          # Reserva espa o na pilha para o frame pointer
```

```
59      sw x8, 0(x2)            # Armazena o frame pointer antigo na pilha
60      mv x8, x2               # Atualiza o frame pointer para o topo atual da
            pilha
61      addi x2, x2, -8         # Reserva espa o para 'x' e 'y'
62      jal input               # Chama a fun   o input() para ler o valor de 'x'
63      mv x28, x10             # Armazena o valor lido de 'x' em x28
64      sw x28, -4(x8)          # Armazena o valor de 'x' na pilha
65      jal input               # Chama a fun   o input() para ler o valor de 'y'
66      mv x28, x10             # Armazena o valor lido de 'y' em x28
67      sw x28, -8(x8)          # Armazena o valor de 'y' na pilha
68      lw x28, -4(x8)          # Carrega o valor de 'x' da pilha
69      mv x10, x28             # Atualiza x10 com o valor de 'x'
70      lw x27, -8(x8)          # Carrega o valor de 'y' da pilha
71      mv x11, x27             # Atualiza x11 com o valor de 'y'
72      jal gcd                 # Chama a fun   o gcd
73      mv x26, x10             # Atualiza x26 com o resultado da chamada de gcd
74      mv x10, x26             # Atualiza o valor de retorno (x10) com o resultado
75      jal output              # Chama a fun   o output() para exibir o resultado
76      mv x2, x8               # Restaura o frame pointer
77      lw x8, 0(x2)            # Carrega o frame pointer antigo
78      addi x2, x2, 4          # Ajusta o ponteiro da pilha
79      lw x1, 0(x2)            # Carrega o valor de 'x'
80      addi x2, x2, 4          # Ajusta o ponteiro da pilha
81      ret                     # Retorna da fun   o main
```

5.2 d. Código Executável

Usando o comando xxd -g 4 -c 4 app.bin , podemos observar o código binário que foi gerado a partir do assembly.

```
1  00000000: 1301c1ff  ....
2  00000004: 23201100  # ..
3  00000008: 1301c1ff  ....
4  0000000c: 23208100  # ..
5  00000010: 13040100  ....
6  00000014: 130181ff  ....
7  00000018: 232ea4fe  #...
8  0000001c: 232cb4fe  #,..
9  00000020: 032e84ff  ....
10 00000024: 930d0000  ....
11 00000028: 6314be03  c...
12 0000002c: 032ec4ff  ....
13 00000030: 13010400  ....
14 00000034: 03240100  .$..
15 00000038: 13014100  ..A.
16 0000003c: 83200100  . ..
17 00000040: 13014100  ..A.
18 00000044: 13050e00  ....
19 00000048: 67800000  g...
20 0000004c: 6f000005  o...
21 00000050: 832d84ff  .-..
22 00000054: 13850d00  ....
23 00000058: 032dc4ff  .-..
24 0000005c: 832cc4ff  .,..
25 00000060: 032c84ff  .,..
26 00000064: 33cb8c03  3...
27 00000068: 832c84ff  .,..
28 0000006c: 330c9b03  3...
29 00000070: b30c8d41  ...A
30 00000074: 93850c00  ....
```

```
00000078: eff09ff8  ....
0000007c: 130d0500  ....
00000080: 13010400  ....
00000084: 03240100  .$..
00000088: 13014100  ..A.
0000008c: 83200100  . ..
00000090: 13014100  ..A.
00000094: 13050d00  ....
00000098: 67800000  g...
0000009c: 1301c1ff  ....
000000a0: 23201100  # ..
000000a4: 1301c1ff  ....
000000a8: 23208100  # ..
000000ac: 13040100  ....
000000b0: 130181ff  ....
000000b4: eff0dff4  ....
000000b8: 130e0500  ....
000000bc: 232ec4ff  #...
000000c0: eff01ff4  ....
000000c4: 130e0500  ....
000000c8: 232cc4ff  #,..
000000cc: 032ec4ff  ....
000000d0: 13050e00  ....
000000d4: 832d84ff  .-..
000000d8: 93850d00  ....
000000dc: eff05ff2  .._.
000000e0: 130d0500  ....
000000e4: 13050d00  ....
000000e8: eff09ff1  ....
000000ec: 13010400  ....
000000f0: 03240100  .$..
000000f4: 13014100  ..A.
000000f8: 83200100  . ..
000000fc: 13014100  ..A.
00000100: 67800000  g...
```

5.3 Exemplo 2 : Fluxo Básico

5.3.1 a. Código Fonte: A seguir, mostramos o uso do compilador em um código c- para um programa que faz uma verificação, atribuição, if , else, loop input e output.

```
1
2  int somar(int a, int b) {
3      return a + b;
4  }
5
6  int main(void) {
7      int x;
8      int y;
9      int resultado;
10
11     /* Atribui  o */
12     x = input();
13     y = input();
14
15     /* Chamada de fun  o */
16     resultado = somar(x, y);
17
18     /* Sele  o (Condicional) */
19     if (resultado > 10) {
```

```
20          /* Repeti  o (La o) */
21          int i;
22          i=0;
23          while( i < resultado ) {
24              /* Impress o de resultado */
25              output(i);
26              i=i+1;
27          }
28      } else {
29          output(resultado);
30      }
31
32      return 0;
33  }
```

5.3.2 b. Código Intermediário: A partir desse, obtemos as seguintes quadruplas

```
1  0::     GLOB_MAIN, _, _, main
2  1::     LABEL, somar, _, _
3  2::     PUSH, R1, _, _
4  3::     PUSH, R8, _, _
5  4::     MOVE, R8, R2, _
6  5::     ADDI, R2, R2, -8
7  6::     STORE, R10, R8, somar_a
8  7::     STORE, R11, R8, somar_b
9  8::     LOAD_VAR, R28, R8, somar_a
10 9::     LOAD_VAR, R27, R8, somar_b
11 10::    PLUS_ALOP, R26, R28, R27
12 11::    MOVE, R2, R8, _
13 12::    POP, R8, _, _
14 13::    POP, R1, _, _
15 14::    MOVE, R10, R26, _
16 15::    RET, _, _, _
17 16::    LABEL, main, _, _
18 17::    PUSH, R1, _, _
19 18::    PUSH, R8, _, _
20 19::    MOVE, R8, R2, _
21 20::    ADDI, R2, R2, -16
22 21::    BRANCH_AND_LINK, _, _, input
23 22::    MOVE, R28, R10, _
24 23::    STORE, R28, R8, main_x
25 24::    BRANCH_AND_LINK, _, _, input
26 25::    MOVE, R28, R10, _
27 26::    STORE, R28, R8, main_y
28 27::    LOAD_VAR, R28, R8, main_x
29 28::    MOVE, R10, R28, _
30 29::    LOAD_VAR, R27, R8, main_y
31 30::    MOVE, R11, R27, _
32 31::    BRANCH_AND_LINK, _, _, somar
33 32::    MOVE, R26, R10, _
34 33::    STORE, R26, R8, main_resultado
35 34::    LOAD_VAR, R26, R8, main_resultado
36 35::    LOAD_IMMEDIATE, R25, _, 10
37 36::    LESSEQ_RELOP, _, R26, R25
38 37::    BRANCH_IF, _, _, .L55
39 38::    LOAD_IMMEDIATE, R26, _, 0
40 39::    STORE, R26, R8, main_i
41 40::    LABEL, .L40, _, _
42 41::    LOAD_VAR, R26, R8, main_i
43 42::    LOAD_VAR, R25, R8, main_resultado
```

```
43::    GREATEQ_RELOP, _, R26, R25
44::    BRANCH_IF, _, _, .L53
45::    LOAD_VAR, R26, R8, main_i
46::    MOVE, R10, R26, _
47::    BRANCH_AND_LINK, _, _, output
48::    LOAD_VAR, R25, R8, main_i
49::    LOAD_IMMEDIATE, R24, _, 1
50::    PLUS_ALOP, R22, R25, R24
51::    STORE, R22, R8, main_i
52::    BRANCH, _, _, .L40
53::    LABEL, .L53, _, _
54::    BRANCH, _, _, .L59
55::    LABEL, .L55, _, _
56::    LOAD_VAR, R25, R8, main_resultado
57::    MOVE, R10, R25, _
58::    BRANCH_AND_LINK, _, _, output
59::    LABEL, .L59, _, _
60::    MOVE, R2, R8, _
61::    POP, R8, _, _
62::    POP, R1, _, _
63::    MOVE, R10, 0, _
0::     RET, _, _, _
```

5.3.3 c. Código Assembly: Essas quadruplas geram o seguinte código em assembly. Note que para que se torne possível que o NUGGET_OS possa se encontrar a função main, a seção global é definida.

```
.globl main
somar:
addi x2, x2, -4
sw x1, 0(x2)
addi x2, x2, -4
sw x8, 0(x2)
mv x8, x2
addi x2, x2, -8
sw x10, -4(x8)
sw x11, -8(x8)
lw x28, -4(x8)
lw x27, -8(x8)
add x26, x28, x27
mv x2, x8
lw x8, 0(x2)
addi x2, x2, 4
lw x1, 0(x2)
addi x2, x2, 4
mv x10, x26
ret
main:
addi x2, x2, -4
sw x1, 0(x2)
addi x2, x2, -4
sw x8, 0(x2)
mv x8, x2
addi x2, x2, -16
jal input
mv x28, x10
sw x28, -4(x8)
jal input
mv x28, x10
sw x28, -8(x8)
```

```
34 lw x28, -4(x8)
35 mv x10, x28
36 lw x27, -8(x8)
37 mv x11, x27
38 jal somar
39 mv x26, x10
40 sw x26, -12(x8)
41 lw x26, -12(x8)
42 li x25, 10
43 ble x26, x25, .L55
44 li x26, 0
45 sw x26, -16(x8)
46 .L40:
47 lw x26, -16(x8)
48 lw x25, -12(x8)
49 bge x26, x25, .L53
50 lw x26, -16(x8)
51 mv x10, x26
52 jal output
53 lw x25, -16(x8)
54 li x24, 1
55 add x22, x25, x24
56 sw x22, -16(x8)
57 j .L40
58 .L53:
59 j .L59
60 .L55:
61 lw x25, -12(x8)
62 mv x10, x25
63 jal output
64 .L59:
65 mv x2, x8
66 lw x8, 0(x2)
67 addi x2, x2, 4
68 lw x1, 0(x2)
69 addi x2, x2, 4
70 mv x10, x0
71 ret
```

5.4 d. Código Executável

Usando o comando xxd -g 4 -c 4 app.bin , podemos observar o código binário que foi gerado a partir do assembly.

```
1  00000000: 1301c1ff  ....
2  00000004: 23201100  # ..
3  00000008: 1301c1ff  ....
4  0000000c: 23208100  # ..
5  00000010: 13040100  ....
6  00000014: 130181ff  ....
7  00000018: 232ea4fe  #...
8  0000001c: 232cb4fe  #,..
9  00000020: 032ec4ff  ....
10 00000024: 832d84ff  .-..
11 00000028: 330dbe01  3...
12 0000002c: 13010400  ....
13 00000030: 03240100  .$..
14 00000034: 13014100  ..A.
15 00000038: 83200100  . ..
16 0000003c: 13014100  ..A.
```

```
00000040: 13050d00  ....
00000044: 67800000  g...
00000048: 1301c1ff  ....
0000004c: 23201100  # ..
00000050: 1301c1ff  ....
00000054: 23208100  # ..
00000058: 13040100  ....
0000005c: 130101ff  ....
00000060: eff01ffa  ....
00000064: 130e0500  ....
00000068: 232ec4ff  #...
0000006c: eff05ff9  .._.
00000070: 130e0500  ....
00000074: 232cc4ff  #,..
00000078: 032ec4ff  ....
0000007c: 13050e00  ....
00000080: 832d84ff  .-..
00000084: 93850d00  ....
00000088: eff09ff7  ....
0000008c: 130d0500  ....
00000090: 232aa4ff  #*..
00000094: 032d44ff  .-D.
00000098: 930ca000  ....
0000009c: 63deac03  c...
000000a0: 130d0000  ....
000000a4: 2328a4ff  #(..
000000a8: 032d04ff  .-..
000000ac: 832c44ff  .,D.
000000b0: 63529d03  cR..
000000b4: 032d04ff  .-..
000000b8: 13050d00  ....
000000bc: eff05ff4  .._.
000000c0: 832c04ff  .,..
000000c4: 130c1000  ....
000000c8: 338b8c01  3...
000000cc: 232864ff  #(d.
000000d0: 6ff09ffd  o...
000000d4: 6f000001  o...
000000d8: 832c44ff  .,D.
000000dc: 13850c00  ....
000000e0: eff01ff2  ....
000000e4: 13010400  ....
000000e8: 03240100  .$..
000000ec: 13014100  ..A.
000000f0: 83200100  . ..
000000f4: 13014100  ..A.
000000f8: 13050000  ....
000000fc: 67800000  g...
```

5.5 Correspondência entre Código Fonte e Código Intermediário

O código fonte em C é traduzido em um código intermediário que reflete a estrutura e as operações do código fonte. O alocamento de variáveis e a execução das operações são representados no código intermediário com instruções de alocação e carregamento de valores, bem como a operação de adição.

5.6 Correspondência entre Código Intermediário e Código Assembly

O código intermediário é convertido em código assembly com operações específicas para a arquitetura RISC-V. O código assembly reflete diretamente as operações descritas no código intermediário, incluindo a alocação de espaço na pilha, carregamento de valores, operações aritméticas e armazenamento dos resultados.