Construção de Assembler em C
Documentação

Caíque G. Leite[1], **Tharsos Gabriel C. Fernandes**[2]

[1]Instituto de Ciências Exatas e Aplicadas (ICEA) – Universidade Federal de Ouro Preto (UFOP)
Caixa Postal 24 – 35.931-008 – João Monlevade – MG – Brazil

Abstract. *This document aims to show and explain the entire assembler code for the RISC-V instruction set architecture (ISA), written in the C language. The main objective of the project is to show the fundamental processes involved in converting readable code to machine language. The methodology used was an assembler that reads the file twice, the first time storing the addresses of the labels and the second time actually translating the code. The program accepts a source file in asm format and writes an output file in the same format. The implementation covers the types R, I, S and B, provided as a tool to consolidate the learning received in the Fundamentals of Computer Organization and Architecture classes.*

Resumo. *O presente documento visa mostrar e explicar todo o código do montador da arquitetura de conjunto de instruções (ISA) RISC-V, feito na linguagem C. O objetivo principal do projeto é mostrar os processos fundamentais da passagem do código legível para a linguagem de máquina. A metodologia empregada foi a de um montador que lê o arquivo duas vezes, a primeira armazenando os endereços dos rótulos e a segunda traduzindo, de fato, o código. O programa aceita um arquivo fonte no formato asm e escreve um arquivo de saída no mesmo formato. A implementação abrange os tipos R, I, S e B, servindo como ferramenta para fixar os aprendizados recebidos nas aulas de Fundamentos de Organização e Arquitetura de Computadores.*

1. Introdução

O código está dividido em 3 principais blocos, além do que chamará todas as funções para dar origem ao executável, sendo eles o bloco das funções auxiliares que conterá instruções menores, como buscas e conversões, esse bloco consiste na camada mais simples do código, uma vez que é composto por structs, variáveis globais e funções que utilizarão, majoritariamente, as bibliotecas do projeto, o bloco da primeira passagem, onde o código irá ler todo o arquivo enquanto registra o endereço de cada rótulo, a fim de usar depois, e por fim o bloco da segunda passagem, que vai ler novamente a entrada e passar o código da linguagem de baixo nível para a linguagem de máquina.

Ao longo do documento haverão explicações detalhadas da lógica empregada, incluindo a explicação de algumas funções das bibliotecas, e imagens do código para ilustrar. Ao final será mostrado um exemplo, através de capturas de tela, da execução e depois haverá uma checagem, provando a eficácia do código.

2. Desenvolvimento

2.1. Bloco das funções auxiliares

A princípio temos o bloco das funções auxiliares e, como foi falado anteriormente, nele é possível ver constantes, mais especificamente a constante do tamanho máximo da linha, que utilizaremos para delimitar o tamanho do vetor de buffer que irá copiar a linha do arquivo para análise, a constante do tamanho maximo do rotulo, que vai definir o tamanho máximo do nome do rótulo que será armazenado em um vetor posteriormente e, por fim, a constante do número máximo de rótulos. Adiante temos a estrutura de um rotulo, que consiste em uma string armazenando seu nome e um inteiro que armazena seu endereço no arquivo. Por fim, nessa parte anterior às funções, é possível ver um vetor que armazenará os rotulos e um inteiro que contará quantos rótulos existem, além de um inteiro que tem o propósito de ser utilizado como cursor, armazenando o endereço da instrução atual.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>

#define TAMANHO_MAX_LINHA 256  // Tamanho máximo de uma linha
#define TAMANHO_MAX_ROTULO 50    // Define o tamanho máximo do nome do rotulo
#define MAX_ROTULOS 100        // Define o máximo de rótulos que vão ser usados

//-----------------------------------------------------------------------------

// Estrutura do rotulo
typedef struct {
    char nome[TAMANHO_MAX_ROTULO]; // Armazena o nome do rótulo (LOOP, END, ETC.).
    int endereco;                  // Armazena o endereço em bytes onde esse rótulo ta
} Rotulo;

//-----------------------------------------------------------------------------

Rotulo tabelaDeRotulos[MAX_ROTULOS]; // Lista de todos os rotulos
int contadorDeRotulos = 0;             // Quantos rotulos tem
int enderecoInstrucaoAtual = 0;      // Tipo um cursor

//-----------------------------------------------------------------------------
```

Figure 1. Imagem das variáveis, constantes e tipos de dados

As duas primeiras funções auxiliares são a de ”adicionarRotulo” e a de ”obterEnderecoRotulo”. A de adicionar rótulo recebe como parâmetro uma string, que é o nome do rótulo, e um inteiro que é o endereço do rótulo, seu funcionamento é simples, ela executará um teste em uma variável global para ver se ainda existe espaço para inserir um novo rótulo ao vetor de rótulos, caso positivo, é utilizada a função ”strcpy” para copiar o nome do rótulo para a próxima posição livre do vetor, depois faz o mesmo com o endereço, mas sem utilizar ”strcpy”, por se tratar de um inteiro. A função de obter endereço do rótulo recebe o nome do rótulo como parâmetro e percorre todo o vetor de rótulos, comparando os nomes, utilizando ”strcmp”, se for igual, a função retornará o endereço do rótulo, caso contrário, a iteração continua, se, ao final, não houver nenhum rótulo com esse nome, será retornado -1.

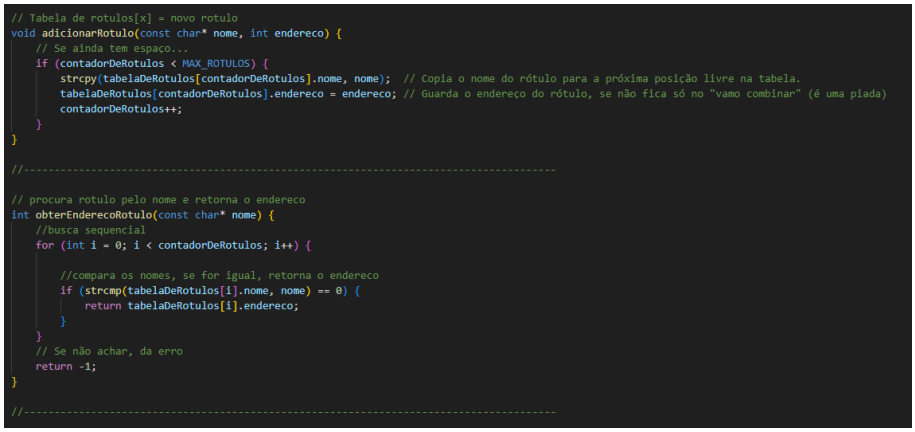


Figure 2. Funções de adicionar rótulo e obter endereço

Por fim, vêm as funções de conversão ”registrador_para_numero” e ”inteiro_para_string_binaria”. A função que passa um registrador para um inteiro vai, de forma simples, recebe a string contendo o registrador como parâmentro e testa se a string não é nula e começa com ”x”, se for positivo em ambos os casos, utiliza-se do comando ”atoi”, da biblioteca stdlib.h, se o caso for negativo, retorna -1. A função de passar de inteiro para string binária recebe um valor inteiro e uma string de saída como parâmetro, no início é colocado na posição 32 da string o delimitador ”\0”, para garantir que sua extensão será de 32 bits, após isso é iniciado o processo de conversão com um laço for começando em 31 e caminhando até 0, dentro do laço é testado se o resto do valor atual é igual a 1, caso seja, é armazenado 1 na posição ”i” da string, caso contrário é arnazenado ”0”, ao final da iteração o número é dividido por 2, por armazenar diretamente na string com o uso de um ponteiro, essa função não apresenta retorno.

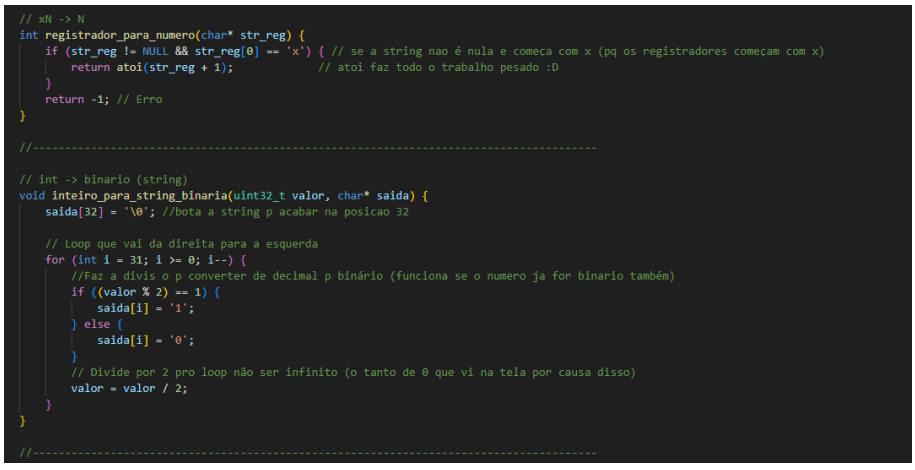
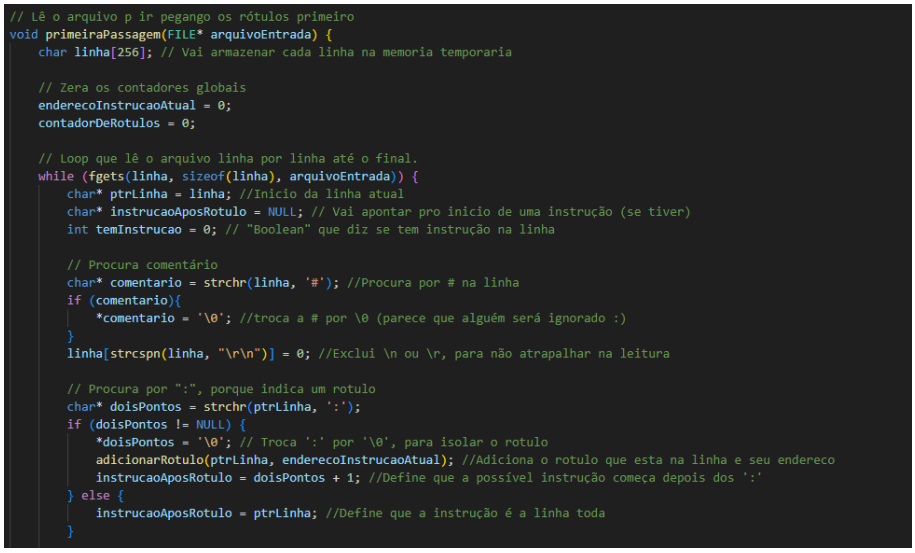


Figure 3. Funções de conversão

2.2. Primeira Passagem

O principal objetivo da primeira passagem é identificar quais são os rótulos utilizados e em qual local eles se encontram, para que o beq, por exemplo, seja compilado corretamente quando usado. A função recebe apenas o arquivo como parâmetro, a princípio ela cria uma

string que será a linha de cada instrução em assembly e depois zera os contadores globais, seguindo o código vemos o loop principal, que itera sobre cada linha do arquivo até atingir o final, utilizando-se do ”fgets” para isso. No início da iteração são atribuídos valores para as variáveis criadas e a seguir há uma parte que verifica se existe algum comentário na linha usando um ponteiro do tipo char e atribuindo o endereço do caractere ”#” utilizando ”strchr”, caso seja encontrado um comentário, o código fará com que o caractere ”#” seja trocado por ”\0”, fazendo com que as próximas funções ignorem a partir do comentário. Utilizando a mesma técnica, a ideia é procurar por ”:”, que é o caractere que ”marca” cada rótulo, caso seja encontrado, troca os dois pontos por ”\0” e adiciona o rótulo e o seu endereço no vetor de rótulos e define que a próxima instrução estará logo após o rótulo, caso não encontre nenhum rótulo, define que a próxima instrução já será a linha.



```c
// Lê o arquivo p ir pegango os rótulos primeiro
void primeiraPassagem(FILE* arquivoEntrada) {
    char linha[256]; // Vai armazenar cada linha na memoria temporaria

    // Zera os contadores globais
    enderecoInstrucaoAtual = 0;
    contadorDeRotulos = 0;

    // Loop que lê o arquivo linha por linha até o final.
    while (fgets(linha, sizeof(linha), arquivoEntrada)) {
        char* ptrLinha = linha; //Inicio da linha atual
        char* instrucaoAposRotulo = NULL; // Vai apontar pro inicio de uma instrução (se tiver)
        int temInstrucao = 0; // "Boolean" que diz se tem instrução na linha

        // Procura comentário
        char* comentario = strchr(linha, '#'); //Procura por # na linha
        if (comentario){
            *comentario = '\0'; //troca a # por \0 (parece que alguém será ignorado :)
        }
        linha[strcspn(linha, "\r\n")] = 0; //Exclui \n ou \r, para não atrapalhar na leitura

        // Procura por ":", porque indica um rotulo
        char* doisPontos = strchr(ptrLinha, ':');
        if (doisPontos != NULL) {
            *doisPontos = '\0'; // Troca ':' por '\0', para isolar o rotulo
            adicionarRotulo(ptrLinha, enderecoInstrucaoAtual); //Adiciona o rotulo que esta na linha e seu endereco
            instrucaoAposRotulo = doisPontos + 1; //Define que a possível instrução começa depois dos ':'
        } else {
            instrucaoAposRotulo = ptrLinha; //Define que a instrução é a linha toda
        }
```

Figure 4. Primeira parte da primeira passagem

Para finalizar, é criada uma string como buffer temporário para ver se há alguma palavra após a instrução, utilizando sscanf e testando se a instrução após o rótulo não é nula, caso seja positivo, a variável ”temInstrucao” recebe 1, mostrando que é verdadeira, e a seguir, caso ”temInstrução” seja verdadeira, o endereço da instrução atual é incrmentado (em bytes), finalizando a função da primeira passagem.

```
        // Olha se tem algo além de espaços na instrução
        char temp[10]; // Buffer temporário para ver se tem uma palavra.

        // 'sscanf' tenta ler uma palavra da string. Se conseguir, retorna 1.
        if (instrucaoAposRotulo != NULL && sscanf(instrucaoAposRotulo, " %s", temp) == 1) {
            temInstrucao = 1; // Marca que esta linha tem uma instrução.
        }

        // Se a linha tinha uma instrução
        if (temInstrucao) {
            enderecoInstrucaoAtual += 4; //Incrementa o endereço em 4 bytes
        }
    }
}

//-----------------------------------------------------------------------------
```

Figure 5. Segunda parte da primeira passagem

2.3. Segunda Passagem

A segunda passagem se inicia de maneira similar à primeira, com o diferencial de ser a função que tem como objetivo traduzir o código de assembly para a linguagem de máquina por meio de strings, a princípio ela recebe o arquivo de entrada e o arquivo de saída como parâmetros, depois inicia e declara variáveis, como a string que será usada para ler a linha, a string que armazenará o valor binário e o inteiro que indica o endereço. Após essa parte inicial, se inicia um loop que, através do ”fgets”, vai ler todo o arquivo linha a linha e no princípio do loop é possível notar que foram utilizados códigos muito parecidos à primeira passagem, que é a string que armazena a posição do comentário para depois substituí-lo por ”\0”, para que as demais funções ignorem os comentários e a string que identifica onde estão os dois pontos do rótulo e atribui ao endereço da instrução a posição do sinal incrementado por 1, para que os rótulos não causem erro na leitura, uma vez que já temos eles registrados.

```
// Vai ler o arquivo e passar ele para binário
void segundaPassagem(FILE* arquivoEntrada, FILE* arquivoSaida) {
    char linha[256]; // Linha atual
    char str_binaria[33]; // String para armazenar a representação binária da instrução
    enderecoInstrucaoAtual = 0; // Zera o contador do "cursor"

    // Loop que lê o arquivo de entrada linha por linha
    while (fgets(linha, sizeof(linha), arquivoEntrada)) {
        char* comentario = strchr(linha, '#'); //Procura um '#' na linha

        // Se o ponteiro não for null...
        if (comentario){
            *comentario = '\0'; // Troca '#' por '\0', o que faz as funções pularem o comentário
        }
        linha[strcspn(linha, "\r\n")] = 0; //A linha na posição da quebra de linha troca o n(\n) ou o r(\r) por 0(\0)

        char* ptrLinha = linha; //Ponteiro -> inicio da linha
        char* doisPontos = strchr(ptrLinha, ':'); // procura um rotulo
        //Se tem rotulo
        if (doisPontos){
            ptrLinha = doisPontos + 1; // manda o cursor pra depois dele
        }
```

Figure 6. Primeira parte da segunda passagem

A partir de agora inicia-se a parte mais importante do código que é a conversão e gravação em si, é criada uma string que vai armazenar, a princípio, somente o comando

presente na linha, para isso foi utilizada a função ”strtok”, que faz exatamente o que é necessário para a tarefa, a partir disso o código continuará somente se a string não estiver nula. Seguido essa parte, é também declarado uma variável do tipo 32 bits que vai guardar a instrução binária. O código a seguir fará uma série de comparações com a string que armazena o comando em assembly para descobrir qual comando é e tratá-lo da forma correta, uma vez encontrado, começa o processo de ”segmentação” do comando utilizando ”strtok” novamente, mas dessa vez passando ”NULL” como parâmetro para conseguir atribuir cada parte a uma string declarada no momento para armazenar um valor, a string ”op_rs1” armazenará o rs1 em string, por exemplo, os registradores receberão seus valores em inteiro por meio das funções auxiliares e o imediato, quando houver, receberá o valor inteiro também ao utilizar ”atoi”.

```
char* mnemonico = strtok(ptrLinha, " \t,()"); //Quebra a linha usando \t e () como delimitadores
if (!mnemonico) {continue;} // Se não tiver instrução, continua para a próxima iteração

//Variavel do tipo 32bits (é cada coisa que a gente aprende nesse trabalho kkkkkkk)
uint32_t instrucaoBinaria = 0;

// Verifica qual a instrução e codifica ela
if (strcmp(mnemonico, "lb") == 0) { //Se a instrução for lb...
    // Formato esperado: lb rd, imm(rs1)
    char* op_rd = strtok(NULL, " \t,()"); // Pega o rd
    char* op_imm = strtok(NULL, " \t,()"); // Pega o imediato
    char* op_rs1 = strtok(NULL, " \t,()"); // Pega o rs1
    int rd = registrador_para_numero(op_rd); // xN -> N
    int rs1 = registrador_para_numero(op_rs1); // mesma coisa
    int imm = atoi(op_imm); // imm string -> int
```

Figure 7. Segunda parte da segunda passagem

Agora com todas as partes devidamente separadas é necessário reorganizá-las e colocá-las nas posições corretas, para isso será muito utilizado os operadores ”¡¡” e ”¿¿”, que equivalem, de certa forma, ao ”sll” e ”srl” do RISC-V, além do ”ou lógico”, porque não é possível adicionar os números corretamente utilizando adição, seguindo a ordem padrão de cada tipo, são colocados todos os códigos em seu lugar, variando de acordo com o comando em assembly. Por fim, ao final de todos os testes o valor do tipo 32bits é passado para uma string através das funções auxiliares e essa string é impressa no arquivo de saída ou no terminal, utilizando ”\0” na função para imprimir 1 instrução por linha.

```
instrucaoBinaria |= 0b0000011;      //Opcoc
instrucaoBinaria |= (rd << 7);
instrucaoBinaria |= (0b000 << 12);
instrucaoBinaria |= (rs1 << 15);
instrucaoBinaria |= ((imm & 0xFFF) << 20);
```

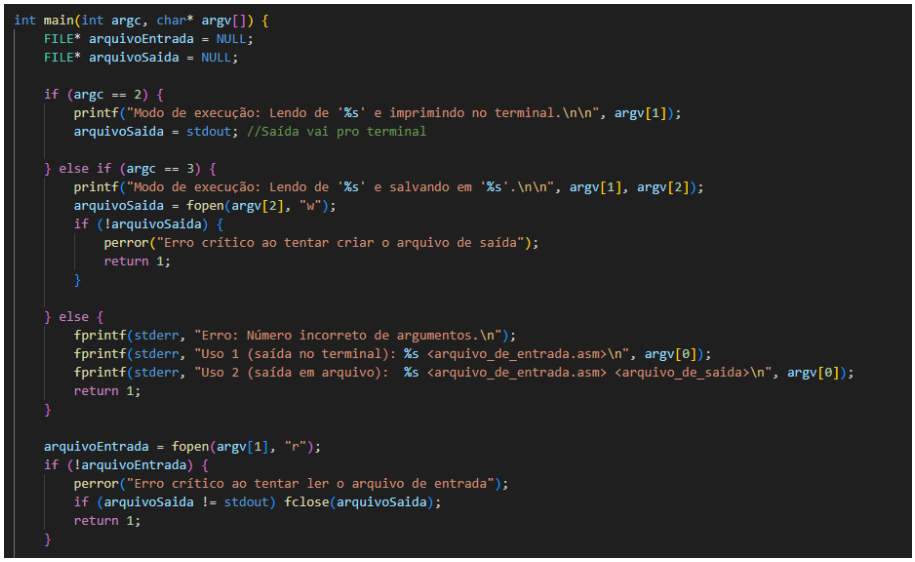
Figure 8. Terceira parte da segunda passagem

3. Execução do código

Para executar o código é necessário compilar antes utilizando o comando respectivo em cada sistema operacional, após essa fase o usuário pode escolher entre passar apenas o arquivo de entrada, para que apareça no terminal o código traduzido para binário ou passar o arquivo de entrada e de saída como parâmetro, para que o programa salve no arquivo de saída o código em linguagem de máquina.

3.1. O método main

O método main se inicia declarando os ponteiros para o arquivo de entrada e o arquivo de saída, é importante afirmar que sempre se deve passar o arquivo de entrada como primeiro parâmetro, logo em seguida ele testa quantos argumentos foram passados ao executar o programa, se foram 2, ou seja, se foi passado somente o arquivo de entrada, ele redireciona a saída para o terminal, caso tenha sido 3, a execução ocorre normalmente com o arquivo de saída sendo preenchido. Após as verificações de segurança é aberto o arquivo de entrada e testado se ele foi aberto corretamente.



```c
int main(int argc, char* argv[]) {
    FILE* arquivoEntrada = NULL;
    FILE* arquivoSaida = NULL;

    if (argc == 2) {
        printf("Modo de execução: Lendo de '%s' e imprimindo no terminal.\n\n", argv[1]);
        arquivoSaida = stdout; //Saída vai pro terminal

    } else if (argc == 3) {
        printf("Modo de execução: Lendo de '%s' e salvando em '%s'.\n\n", argv[1], argv[2]);
        arquivoSaida = fopen(argv[2], "w");
        if (!arquivoSaida) {
            perror("Erro crítico ao tentar criar o arquivo de saída");
            return 1;
        }

    } else {
        fprintf(stderr, "Erro: Número incorreto de argumentos.\n");
        fprintf(stderr, "Uso 1 (saída no terminal): %s <arquivo_de_entrada.asm>\n", argv[0]);
        fprintf(stderr, "Uso 2 (saída em arquivo):  %s <arquivo_de_entrada.asm> <arquivo_de_saida>\n", argv[0]);
        return 1;
    }

    arquivoEntrada = fopen(argv[1], "r");
    if (!arquivoEntrada) {
        perror("Erro crítico ao tentar ler o arquivo de entrada");
        if (arquivoSaida != stdout) fclose(arquivoSaida);
        return 1;
    }
```

Figure 9. Primeira parte do método main

Seguindo a frente são utilizados os métodos da primeira passagem, ”rewind” que volta o cursor ao início do arquivo, e depois é feita a chamada da segunda passagem. Ao finalizar o arquivo de entrada é fechado e o arquivo de saída é fechado somente se ele não é o terminal.

```c
    primeiraPassagem(arquivoEntrada);
    rewind(arquivoEntrada);

    segundaPassagem(arquivoEntrada, arquivoSaida);

    fclose(arquivoEntrada);
    if (arquivoSaida != stdout) {
        fclose(arquivoSaida);
        printf("\nMontagem concluída. Saída salva em: %s\n", argv[2]);
    } else {
        printf("\nMontagem concluída.\n");
    }
    return 0;
}
```

Figure 10. Segunda parte do método main

3.2. A entrada e a saída

Para o arquivo de entrada, criamos um exemplo básico com todos os comandos implementados, como é possível ver na imagem 11, logo a seguir. Como saída o arquivo já estará preenchido, como se o programa já tivesse sido executado, para fins de demonstração, como é possível ver na figura 12. O terminal também foi preenchido quando passado somente o arquivo de entrada como parâmetro, como é possível constatar na imagem 13.

entrada.asm

```
1    ori  x5, x0, 1
2    ori  x6, x0, 2
3    beq  x5, x6, FIM
4    sub  x6, x6, x5
5    FIM: and  x5, x5, x0
```

Figure 11. Arquivo ”entrada.asm”

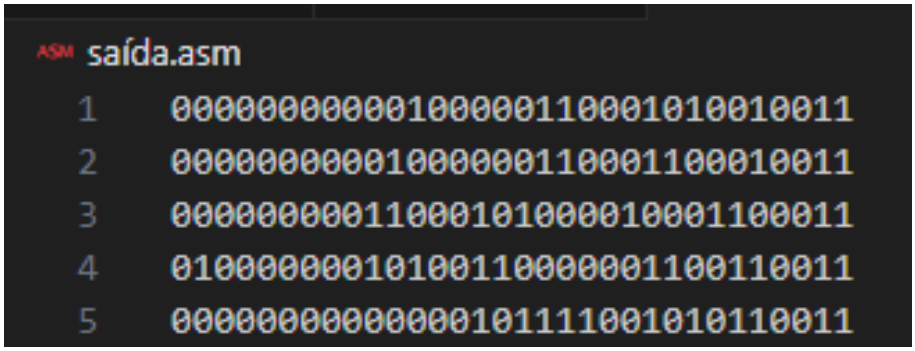


Figure 12. Arquivo ”saída.asm”

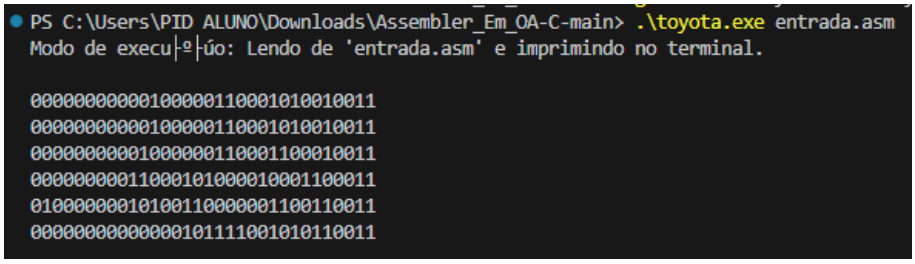


Figure 13. Terminal preenchido

3.3. Conferência da saída

A conferência do código será realizada linha a linha utilizando a ferramenta ”rvcodec.js” disponível gratuitamente. As instrução em binário devem ser idênticas às instruções mostradas na imagem 11, com uma sutil diferença em instruções com rótulos, como ”beq”, pois ao trazer da linguagem de máquina para a linguagem de baixo nível, virá o endereço do rótulo, não a palavra utilizada. Seguirão as imagens das instruções em ordem mostrando a acurácia e funcionabilidade do programa em montar o código:

Figure 14. ”ori x5, x0, 1”

Figure 15. ”ori x6, x0, 2”

Figure 16. ”beq x5, x6, FIM”

Figure 17. ”sub x6, x6, x5”

Figure 18. ”FIM: and x5, x5, x0”

4. Conclusão

Como é possível notar através da seção de execução do código, a entrada, em assembly, foi convertida com sucesso em binário, mostrando que, apesar de desafiador, foi possível empregar os conceitos ensinados nas aulas de Fundamentos de Organização e Arquitetura de Computadores para montar uma versão reduzida de um montador, como os utilizados atualmente e que grande parte das pessoas sequer sabem que existem, nos ajudando a não só ver melhor como funcionam as nossas máquinas, mas também a entender o quanto a história dos computadores é impressionante e como diferentes arquiteturas, por mais que parecidas, podem ser completamente diferentes.