Relátorio Trabalho Prático 1 Montador RISC-V

Gabriel Bonfim - 4252[1], **Marcos Biscotto - 4236**[1],
Gabriel Pádua - 4705[1], **Patrick Oliveira - 4217** [1]

[1] Universidade Federal de Viçosa (UFV) - Campus Florestal
Florestal – MG – Brasil

gabriel.bonfim@ufv.br, marcos.biscotto@ufv.br,

gabriel.padua@ufv.br, patrick.araujo@ufv.br

Abstract.

In this report, the construction of a RISC-V assembler capable of translating assembly commands into machine language will be exemplified and explained. For this, the high-level language C was used. Throughout the report, it will be possible to understand the different tools and functions used in order to carry out such a proposal.

Resumo.

Neste relatório será exemplificado e explicado a construção de um montador RISC-V capaz de traduzir comandos assembly para linguagem de máquina. Para tal, foi utilizada a linguagem de alto nível C. Ao longo do relatório, será possível entender as diferentes ferramentas e funções utilizadas, a fim de realizar tal proposta.

1. Informações Gerais

Para a realização do trabalho prático, foi utilizada a linguagem C. Pois sua tipagem mais próxima da linguagem de máquina e sua execução mais rápida facilitam a execução dos códigos. O intuito principal do código é a montagem de valores binários através da leitura de comandos em RISC-V. Ou seja, um montador capaz de reconhecer as funções passadas e retornar seu devido código binário.

2. Apresentação do problema

O conjunto de instruções RISC-V, é utilizado na projeção e venda de processadores. Suas intruções possuem uma vasta gama de usos e suas instruções básicas possuem tamanho fixo de 32 bits.

R-type Instructions	funct7	rs2	rs1	funct3	rd	opcode	Example
add (add)	0000000	00011	00010	000	00001	0110011	add x1, x2, x3
sub (sub)	0100000	00011	00010	000	00001	0110011	sub x1, x2, x3
I-type Instructions	**immediate**		**rs1**	**funct3**	**rd**	**opcode**	**Example**
addi (add immediate)	001111101000		00010	000	00001	0010011	addi x1, x2, 1000
ld (load doubleword)	001111101000		00010	011	00001	0000011	ld x1, 1000 (x2)
S-type Instructions	**immed -iate**	**rs2**	**rs1**	**funct3**	**immed -iate**	**opcode**	**Example**
sd (store doubleword)	0011111	00001	00010	011	01000	0100011	sd x1, 1000(x2)

Figure 1. Instruções RISC-V

Cada instrução, é composta por seu opcode, seus respectivos registradores e functs que estão dispostos em diferentes combinações de posição baseado no tipo de instrução que está sendo passada. O montador deve ser capaz de reconhecer qual tipo de instrução está sendo passada e logo após aplicar a combinação correta a fim de gerar uma sequência binária válida que possa ser interpretada pelo processador.

3. Funcionamento do código

O diretório do trabalho prático presente no Github possui três arquivos, um .c que possui os códigos e funções, um arquivo de input e um de output. O Código ao ser executado, lê linha por linha cada um dos comandos e os inprime linha a linha no arquivo de output em ordem.

4. Função Main

A função main recebe o arquivo de entrada, e enquanto houverem linhas à serem lidas, a função montarBinario é chamada para escrever o arquivo de saída.

```c
int main(){
    FILE *in_ptr, *out_ptr;
    char str[50];
    char * bin = (char *) malloc(33);
    char inputfile[50];
    printf("Insira o nome do arquivo: ");
    scanf("%s", inputfile);
    in_ptr = fopen(inputfile, "a+");
    out_ptr = fopen("output.rsm", "w");

    if (NULL == in_ptr) {
        printf("input file can't be opened \n");
    }
    if (NULL == out_ptr) {
        printf("output file can't be opened \n");
    }
    while (fgets(str, 50, in_ptr) != NULL) {
        montarBinario(str, &bin);
        fprintf(out_ptr, "%s\n", bin);
    }
    fclose(in_ptr);
    fclose(out_ptr);
    return 0;
}
```

Figure 2. Função Main

5. Função TipoI

A função tipo I recebe cada um dos parâmetros que compõem uma string completa de um comando RISC-V convertido para linguagem de máquina e o concatena na ordem necessária.

```c
void tipoI(char* f3, char* p1, char* p2, char* p3, char ** bin) {
    char result[33];
    strcat(result, p3);
    strcat(result, p2);
    strcat(result, f3);
    strcat(result, p1);
    strcat(result, "0010011");
    strcpy(*bin, result);
}
```

Figure 3. Função TipoI

6. Função: MontadorBinario

Esta é a principal função do código, através dela e de if's cadenciados para cada comando assembly, pode-se converter de maneira correta cada um dos casos. Cada if define se a variavel "cmd" que possui a primeira parte da string do input corresponde a um determinado comando. Assim, aplicando da maneira necessária o opcode e func de tal comando. Além disso, cada registrador é convertido para binário através da função intToBin.

```
void montadorBinario(char* cmd, char* p1, char* p2, char* p3, char** bin) {
    int p1n, p2n, p3n;
    p1n = (int) strtol(p1, (char **)NULL, 10);
    p2n = (int) strtol(p2, (char **)NULL, 10);
    p3n = (int) strtol(p3, (char **)NULL, 10);
```

Figure 4. Função: MontadorBinário

Dentro da função MontadorBinario, cada if retorna um comando em binário

```
if (!strcmp(cmd, "add")) {
    tipoR("0000000", "000", intToBin(p1n, 5), intToBin(p2n, 5), intToBin(p3n, 5), bin);
    return;
}
```

Figure 5. leitor da função add

```
if (!strcmp(cmd, "sub")) {
    tipoR("0100000", "000", intToBin(p1n, 5), intToBin(p2n, 5), intToBin(p3n, 5), bin);
    return;
}
```

Figure 6. leitor da função Sub

```
if (!strcmp(cmd, "and")) {
    tipoR("0000000", "111", intToBin(p1n, 5), intToBin(p2n, 5), intToBin(p3n, 5), bin);
    return;
}
```

Figure 7. leitor da função And

```
if (!strcmp(cmd, "or")) {
    tipoR("0000000", "110", intToBin(p1n, 5), intToBin(p2n, 5), intToBin(p3n, 5), bin);
    return;
}
```

Figure 8. leitor da função Or

```
if (!strcmp(cmd, "xor")) {
    tipoR("0000000", "100", intToBin(p1n, 5), intToBin(p2n, 5), intToBin(p3n, 5), bin);
    return;
}
```

Figure 9. leitor da função Xor

```
if (!strcmp(cmd, "sll")) {
    tipoR("0000000", "001", intToBin(p1n, 5), intToBin(p2n, 5), intToBin(p3n, 5), bin);
    return;
}
```

Figure 10. leitor da função Sll

```
if (!strcmp(cmd, "srl")) {
    tipoR("0000000", "101", intToBin(p1n, 5), intToBin(p2n, 5), intToBin(p3n, 5), bin);
    return;
}
```

Figure 11. leitor da função Srl

```
if (!strcmp(cmd, "addi")) {
    tipoI("000", intToBin(p1n, 5), intToBin(p2n, 5), intToBin(p3n, 12), bin);
    return;
}
```

Figure 12. leitor da função Addi

```
if (!strcmp(cmd, "addi")) {
    tipoI("000", intToBin(p1n, 5), intToBin(p2n, 5), intToBin(p3n, 12), bin);
    return;
}
```

Figure 13. leitor da função Andi

```
if (!strcmp(cmd, "ori")) {
    tipoI("110", intToBin(p1n, 5), intToBin(p2n, 5), intToBin(p3n, 12), bin);
    return;
}
```

Figure 14. leitor da função Ori

7. Função formatRegs

A função formatRegs vai receber cada linha do arquivo de entrada e vai retirar os “x” e as vírgulas no final de cada registrador mas manterá o X no operador no caso do XOR.

```
char* formatRegs(char * str) {
    if (str[strlen(str)-1] == ',')
    {
        str[strlen(str)-1] = '\0';
    }
    if (str[0] == 'x')
    {
        memmove(str, str+1, strlen(str));
    }
    return str;
}
```

Figure 15. Função FormatRegs

8. Função intToBin

A função intToBin percorrerá os valores inteiros e retornará o binário correspondente com o número de bits adequado.

```
const char * intToBin(int num, int n) {
    int x, y, z;
    x = num;
    char* result = (char*) malloc(n+1);
    for(z = n-1; z >= 0; z--) {
        y = x >> z;
        if (z == n-1)
        {
            if(y & 1) {
                strcpy(result, "1");
            } else {
                strcpy(result, "0");
            }
        } else {
            if(y & 1) {
                strcat(result, "1");
            } else {
                strcat(result, "0");
            }
        }
    }
    return result;
}
```

Figure 16. Função intToBin

9. Função: montarBinario

A função montarBinário vai pegar cada segmento de uma linha do arquivo de entrada e salvará em uma variável correspondente. Por exemplo, o operador será salvo na variável “cmd”, os registradores de destino, registrador 1 e registrador 2 (já sem o x e a vírgula) serão salvos nas variáveis “p1”, “p2” e “p3” respectivamente.

```
void montarBinario(char * str, char **bin) {
    char* cmds;
    char *cmd, *p1, *p2, *p3;
    int i = 0;
    cmds = strtok(str, " ");
    while (cmds != NULL) {
        if (i == 0) {
            cmd = cmds;
        }
        if (i == 1) {
            p1 = formatRegs(cmds);
        }
        if (i == 2) {
            p2 = formatRegs(cmds);
        }
        if (i == 3) {
            p3 = formatRegs(cmds);
        }
        i += 1;
        cmds = strtok (NULL, " ");
    }
    montadorBinario(cmd, p1, p2, p3, bin);
}
```

Figure 17. Função MontarBinario

10. Função TipoR

A função tipo R recebe cada um dos parâmetros que compõem uma string completa de um comando RISC-V convertido para linguagem de máquina e o concatena na ordem necessária.

```
void tipoR(char* f7, char* f3, char* p1, char* p2, char* p3, char ** bin) {
    char result[33];
    strcat(result, f7);
    strcat(result, p3);
    strcat(result, p2);
    strcat(result, f3);
    strcat(result, p1);
    strcat(result, "0110011");
    strcpy(*bin, result);
}
```

Figure 18. Função TipoR

11. Conclusão

Após começarmos a construir um montador do zero, a partir de buscas e pesquisas para melhorar nosso código, conseguimos ter uma maior noção de como funciona um assembler e o trabalho interno exercido por ele para fazer com que os programas escritos em linguagem de alto nível possam ser convertidas e lidas pelo computador. Desta forma, expandindo nosso conhecimento sobre como é gerada a linguagem de máquina no baixo nível.

12. Referências

- Patterson, D.A.; Hennessy J.L. Computer Organization and Design: RISC-V Edition, 2a Edic ao, Editora Morgan Kaufman, 2021.