

Montador RISC-V

Grupo 13 - CCF252 - Organização de Computadores I

Tariky Rodrigues Campos - 5758

Vitor Afonso de Souza Carvalho - 5906

Universidade Federal de Viçosa – Campus Florestal

Professor: José Augusto Miranda Nacif

Abril de 2025

Sumário

1	Introdução	3
1.1	Formatos UJ e Instruções Pseudo	3
2	Objetivo	3
3	Tecnologias Utilizadas	4
3.1	Linguagem de Programação	4
3.2	Compilação	4
4	Execução do Programa	4
4.1	Modo Terminal	4
4.2	Modo Arquivo de Saída	4
5	Instruções Implementadas	5
6	Figuras	5
6.1	Função de Impressão e Estrutura de Instruções R	6
6.2	Função de Identificação de Instrução	6
6.3	Função de Leitura do Arquivo e Identificação de Tipos	7
6.4	Arquivo de entrada - entrada.asm	8
7	Considerações sobre o Opcode do Tipo SB	8
8	Conclusão	9
	Referências	9

1 Introdução

Para que um computador execute comandos escritos em linguagens de alto nível, é necessário convertê-los para linguagem de máquina, que o processador entende diretamente. Esse processo passa pela compilação para Assembly e posterior conversão por um montador (assembler) em instruções binárias.

Essas instruções seguem formatos fixos e contêm códigos de operação (opcodes) e operandos que definem o comportamento da CPU. Este projeto foca na criação de um montador simplificado para a arquitetura RISC-V, amplamente utilizada por sua modularidade e simplicidade.

A implementação do montador reforça conceitos como manipulação de bits, strings, estrutura de dados e organização lógica de sistemas, conectando teoria e prática na transformação de código de alto nível em instruções executáveis.

1.1 Formatos UJ e Instruções Pseudo

Além dos formatos clássicos como R, I, S e SB, o projeto também aborda o formato UJ, utilizado por instruções de salto incondicional de longo alcance, como a `jal` (jump and link). Esse tipo de instrução exige a manipulação de um imediato de 20 bits, dividido em campos específicos para codificação binária, como demonstrado na função `CabecarioUJ`.

Outro ponto importante são as instruções pseudo, que embora não existam diretamente no conjunto de instruções RISC-V, são interpretadas pelo montador e convertidas para uma ou mais instruções reais. No projeto, pseudo-instruções como `mv` (movimentação de registradores) e `li` (carregamento de valor imediato) são tratadas pela função `CabecarioPseudo`, sendo traduzidas para instruções do tipo I, como `addi`.

Esse tratamento facilita a escrita de programas em Assembly, ao mesmo tempo que evidencia o papel do montador na tradução de instruções mais legíveis para binários executáveis. A inclusão do suporte aos tipos UJ e às pseudo-instruções amplia a funcionalidade do montador e aproxima sua lógica de montadores reais.

2 Objetivo

O propósito central deste trabalho foi desenvolver, de forma prática, um montador capaz de converter instruções escritas na linguagem Assembly da arquitetura RISC-V para seu correspondente em linguagem de máquina — ou seja, instruções codificadas em binário que possam ser diretamente interpretadas pela CPU.

Através da implementação do montador, buscamos consolidar o entendimento sobre a estrutura das instruções RISC-V, abordando aspectos fundamentais como o reconhecimento de opcodes, a definição dos campos de registradores e a manipulação de valores imediatos. O projeto exigiu que cada instrução fosse corretamente identificada, interpretada e traduzida conforme sua estrutura específica, de acordo com os formatos definidos na arquitetura (como R, I, S e SB).

Durante o desenvolvimento, foi necessário aplicar conhecimentos adquiridos em disciplinas relacionadas à arquitetura de computadores, lógica de programação, manipulação de arquivos e operações com bits. Além disso, o trabalho proporcionou uma oportunidade de aprofundamento em Python, que foi utilizado como linguagem de implementação devido à sua clareza sintática e suporte a manipulações de string e estruturas de dados.

Também foi um objetivo compreender e tratar os desafios práticos envolvidos no processo de montagem, como o posicionamento correto dos campos binários nas instruções e a representação precisa de números negativos utilizando complemento de dois. Com isso, o projeto não apenas reforçou os conceitos teóricos da arquitetura RISC-V, como também demonstrou sua aplicabilidade prática por meio do desenvolvimento de uma ferramenta funcional.

3 Tecnologias Utilizadas

3.1 Linguagem de Programação

A linguagem utilizada foi **C**, escolhida por sua proximidade com o baixo nível e por permitir maior controle sobre manipulação de bits, facilitando a implementação das instruções em formato binário.

3.2 Compilação

Para compilar o código, é necessário ter um compilador C instalado. No Linux, o compilador pode ser instalado com o seguinte comando:

```
sudo apt install gcc
```

4 Execução do Programa

O montador pode ser executado de duas formas:

4.1 Modo Terminal

Para Windows, insira no terminal passo a passo:

```
mingw32-make compile
```

```
mingw32-make run
```

```
digite o nome do arquivo.asm
```

```
ou
```

```
gcc codigo/main.c codigo/EntradaArquivo/Entrada.c codigo/
```

```
ConverterInstrucoes/Converte_Instrucoes.c codigo/
```

```
ConverterInstrucoes/EstruturaInstrucoes/Formato_Instrucoes.c
```

```
-o programa
```

```
./programa.exe
```

```
digite o nome do arquivo.asm
```

Para o linux, digite apenas:

```
make run
```

4.2 Modo Arquivo de Saída

Faça o mesmo processo de compilação e, em seguida, insira o nome da entrada.asm.

Neste modo, a saída binária é salva no arquivo especificado (pasta `output`) após o parâmetro `-o`.

5 Instruções Implementadas

Todas as instruções foram devidamente implementadas no montador, conforme especificado no enunciado do trabalho. As instruções contemplam diferentes formatos (R, I, S e SB), permitindo testar a versatilidade do montador e sua capacidade de interpretar corretamente diversos padrões da arquitetura RISC-V.

A seguir, apresentamos as instruções implementadas pelo grupo 13:

- **lb**: Load byte
- **sb**: Store byte
- **add**: Soma entre registradores
- **and**: Operação lógica AND
- **ori**: Operação OR com imediato
- **sll**: Deslocamento lógico à esquerda
- **bne**: Desvio condicional se diferente

Cada uma dessas instruções foi testada individualmente, assegurando que os campos binários fossem corretamente organizados e os resultados gerados estivessem de acordo com o formato exigido pela arquitetura.

6 Figuras

A partir da Seção 6, serão apresentadas algumas figuras que ilustram o funcionamento do montador RISC-V. Elas mostram exemplos de entrada, a saída esperada em binário e como o programa responde na prática. O objetivo é facilitar a visualização do processo e reforçar o entendimento do que foi implementado.

Abaixo, temos a função responsável por converter um número inteiro em sua representação binária com quantidade de bits definida, armazenando o resultado em uma string de caracteres:

```
void TransformarBinario(int Numero, int QuantidadeBits, char* Transformado){
    unsigned int valor = (unsigned int)Numero;
    for(int i = QuantidadeBits-1; i >= 0; i--) {
        int r = valor >> i;
        if(r & 1) {
            Transformado[QuantidadeBits - 1 - i] = '1';
        }
        else{
            Transformado[QuantidadeBits - 1 - i] = '0';
        }
    }
    Transformado[QuantidadeBits] = '\0';
}
```

Figura 1: Função TransformarBinario: conversão de número inteiro para binário com quantidade de bits definida.

6.1 Função de Impressão e Estrutura de Instruções R

A função a seguir é responsável por imprimir os campos de uma instrução do tipo R na tela e também gravá-los em um arquivo de saída. Essa função utiliza a estrutura TipoR, que representa os campos binários de uma instrução do formato R na arquitetura RISC-V.

```
void ImprimirR(TipoR* instrucao, FILE *outputFile){
    printf("%s", instrucao->funct7);
    printf("%s", instrucao->rs2);
    printf("%s", instrucao->rs1);
    printf("%s", instrucao->funct3);
    printf("%s", instrucao->rd);
    printf("%s\n", instrucao->opcode);
    fprintf(outputFile,"%s", instrucao->funct7);
    fprintf(outputFile,"%s", instrucao->rs2);
    fprintf(outputFile,"%s", instrucao->rs1);
    fprintf(outputFile,"%s", instrucao->funct3);
    fprintf(outputFile,"%s", instrucao->rd);
    fprintf(outputFile,"%s\n", instrucao->opcode);
}
```

Figura 2: Função ImprimirR que imprime e grava os campos binários de uma instrução do tipo R.

A seguir, temos a estrutura TipoR, que contém os campos necessários para compor uma instrução do tipo R, incluindo registradores, códigos de função e opcode.

```
typedef struct{
    char funct7[8];
    char rs2[6];
    char rs1[6];
    char funct3[4];
    char rd[6];
    char opcode[8];
}TipoR;

> typedef struct{ ...
}TipoI;

> typedef struct{ ...
}TipoS;

> typedef struct { ...
} TipoSB;

> typedef struct{ ...
}TipoUJ;
```

Figura 3: Declaração da estrutura TipoR, utilizada para armazenar os campos binários de uma instrução do tipo R.

6.2 Função de Identificação de Instrução

A função a seguir é responsável por identificar o tipo da instrução fornecida:

```

char IdentificarTipo (char tipo[6]){
    if (strcmp(tipo, "add")==0
    || strcmp(tipo, "sub")==0 || strcmp(tipo, "sll")==0
    || strcmp(tipo, "xor")==0 || strcmp(tipo, "srl")==0
    || strcmp(tipo, "sra")==0 || strcmp(tipo, "or")==0
    || strcmp(tipo, "and")==0 || strcmp(tipo, "lrd")==0
    || strcmp(tipo, "sc.d")==0){
        return 'R';
    }

    else if(strcmp(tipo, "lb")==0
    || strcmp(tipo, "lh")==0 || strcmp(tipo, "lw")==0
    || strcmp(tipo, "ld")==0 || strcmp(tipo, "lbu")==0
    || strcmp(tipo, "lhu")==0 || strcmp(tipo, "lwu")==0
    || strcmp(tipo, "addi")==0 || strcmp(tipo, "slli")==0
    || strcmp(tipo, "xori")==0 || strcmp(tipo, "srli")==0
    || strcmp(tipo, "srai")==0 || strcmp(tipo, "ori")==0
    || strcmp(tipo, "andi")==0 || strcmp(tipo, "jalr")==0){
        return 'I';
    }

    else if(strcmp(tipo, "sb")==0 || strcmp(tipo, "sh")==0
    || strcmp(tipo, "sw")==0 || strcmp(tipo, "sd")==0){
        return 'S';
    }

    else if(strcmp(tipo, "beq")==0 || strcmp(tipo, "bne")==0
    || strcmp(tipo, "blt")==0 || strcmp(tipo, "bge")==0
    || strcmp(tipo, "bltu")==0 || strcmp(tipo, "bgeu")==0){
        return 'B';
    }

    else if(strcmp(tipo, "mv")==0 || strcmp(tipo, "li")==0
    || strcmp(tipo, "j")==0 || strcmp(tipo, "la")==0){
        return 'P';
    }

    else if(strcmp(tipo, "jal")==0){
        return 'J';
    }
}

```

Figura 4: Função que identifica qual tipo de instrução foi fornecida.

6.3 Função de Leitura do Arquivo e Identificação de Tipos

A função abaixo identifica o tipo de instrução e realiza a leitura do arquivo .asm com as instruções fornecidas:

```

void LerEscrever(char f[], char s[]){
    FILE *file = fopen(f, "r");
    FILE *outputFile = fopen(s, "w");

    if(file && outputFile){
        char linha[50];
        while (fgets(linha, sizeof(linha), file)){
            char tipo[12];
            sscanf(linha, "%s", tipo);
            TipoR instrucao; TipoI instrucao; TipoS instrucao; TipoSB instrucao; TipoUJ instrucao;
            char r1[8], r2[8], r3[8];
            int re1, re2, re3;
            char j = IdentificarTipo(tipo);
            switch (j) {
                case 'R':
                    sscanf(linha, "%s %[^,], %[^,], %s", tipo, r1, r2, r3);
                    re1 = ConverterNumero(r1);
                    re2 = ConverterNumero(r2);
                    re3 = ConverterNumero(r3);
                    CabecarioR(&instrucao, tipo, re1, re2, re3, outputFile);
                    break;

                case 'I':

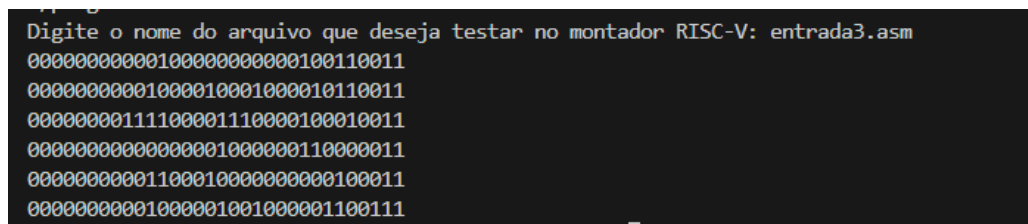
```

Figura 5: Função que identifica o tipo da instrução e realiza a leitura do arquivo .asm.

6.4 Arquivo de entrada - entrada.asm

```
add x2, x0, x1
sll x1, x2, x2
ori x2, x1, 15
lb x3, 0(x1)
sb x3, 0(x2)
bne x1, x2, label
```

Saída esperada - binário



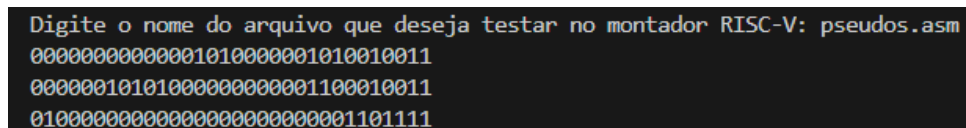
```
Digite o nome do arquivo que deseja testar no montador RISC-V: entrada3.asm
000000000000100000000000100110011
000000000001000010001000010110011
000000000111100001110000100010011
000000000000000001000000110000011
000000000001100010000000000100011
00000000001000001001000001100111
```

Figura 6: Resultado obtido da instrução acima.

Arquivo de entrada - pseudos.asm

```
mv x5, x10
li x6, 42
j 1024
```

Saída esperada - binário



```
Digite o nome do arquivo que deseja testar no montador RISC-V: pseudos.asm
00000000000001010000001010010011
0000001010100000000001100010011
010000000000000000000001101111
```

Figura 7: Resultado obtido da instrução acima.

7 Considerações sobre o Opcode do Tipo SB

Durante a implementação do formato de instrução do tipo SB, surgiram dúvidas quanto ao seu opcode. De acordo com os slides disponibilizados em sala de aula, o opcode correspondente era 1100111. No entanto, ao consultarmos outras fontes, como o site *rvcodec.js* e a especificação oficial da ISA RISC-V, encontramos o opcode 1100011 para esse tipo.

A Tabela abaixo apresenta os campos `Opcode` e `Funct3` das instruções do tipo SB, conforme disponibilizado nos slides da disciplina. Nela, observa-se que todas as instruções utilizam o opcode 1100111, o que motivou sua adoção neste projeto, mesmo diante de divergências com outras fontes.

Format	Instruction	Opcode	Funct3	Funct6/7
SB-type	beq	1100111	000	n.a.
	bne	1100111	001	n.a.
	blt	1100111	100	n.a.
	bge	1100111	101	n.a.
	bltu	1100111	110	n.a.
	bgeu	1100111	111	n.a.

Figura 8: Tabela de instruções SB-type com respectivos opcodes e funct3 conforme material da disciplina.

Apesar dessa divergência, optamos por utilizar o valor apresentado nos slides, registrando essa diferença apenas para fins de documentação e referência. Essa decisão visa manter a coerência com o material didático utilizado durante o curso, mesmo que isso represente uma variação em relação à especificação oficial.

8 Conclusão

Este projeto proporcionou uma compreensão prática da arquitetura RISC-V, especialmente no que diz respeito à estrutura das instruções e sua conversão para binário. A organização modular do código, com funções específicas para cada instrução, facilitou tanto o desenvolvimento quanto a manutenção do montador.

Além do aprendizado técnico, o trabalho também contribuiu para o aprimoramento de habilidades como análise, organização e resolução de problemas. O projeto foi concluído com sucesso, atendendo aos requisitos propostos e oferecendo uma base sólida para futuros aprimoramentos.

Referências

- <https://luplab.gitlab.io/rvcodecjs/>
- https://drive.google.com/file/d/1r_4eAPUh7zSyHgEoIDfc4fG4LWTPiGJb/view
- <https://github.com/oc-ufv/tp01-5758-5906>