CCF 252 - Organização de Computadores I
Trabalho Prático 02 - Montador RISC-V

Mateus Aparecido - 3858, Artur Papa - 3886, Luciano Belo - 3897

26 de novembro de 2020

1 Introdução

O trabalho em questão tem como objetivo abordar os conteúdos trabalhados na disciplina de Organização de Computadores 1, em específico, como visto no módulo 2, a utilização do RISC-V. Um montador [5] é um programa que pega instruções primordiais de uma máquina e as converte em binário, para que o processador do computador possa realizar suas operações básicas.

Vale ressaltar que este trabalho foi feito utilizando a linguagem interpretada e multiparadigma Python [2] juntamente com a utilização de JSON(JavaScript Object Notation) [1] para referenciar os registradores e as instruções, além disso todos os testes foram feitos com o arquivo '.asm' e os outputs em binário.

2 Desenvolvimento

Outrossim, cabe-se dizer que por usarmos a linguagem Python será necessário mostrar como fazer a instalação em um máquina com Ubuntu, como sabemos que o código será testado em um Ubuntu 20, iremos usar a instalação referente a essa distribuição. O Ubuntu 20 e outras versões do Debian linux já vem com o Python3 pré instalado, para ter certeza que a versão da máquina pode receber um update iremos usar o comando apt.

```
$ sudo apt update
$ sudo apt -y upgrade
```

A flag -y irá confirmar que concordamos com qualquer termo a ser pedido, porém em algumas distribuições ainda será requisitado algumas autorizações. Uma vez que o processo estiver acabado poderá ser verificada a versão do Python que está instalada na máquina com o comando:

```
$ python3 -V
```

O output deverá ser similar a isto:

```
Output
Python 3.8.2
```

Para gerenciar pacotes de software para Python, vamos instalar o pip , uma ferramenta que irá instalar e gerenciar pacotes de programação que possamos querer usar em nossos projetos de desenvolvimento:

```
$ sudo apt install -y python3-pip
```

A instalação dos pacotes pode ser feita usando o comando pip3 install + o nome do pacote, como na imagem abaixo:

```
$ pip3 install package_name
```

Existem mais alguns pacotes e ferramentas de desenvolvimento para instalar para garantir que tenhamos uma configuração robusta para nosso ambiente de programação:

```
$ sudo apt install -y build-essential libssl-dev libffi-dev python3-dev
```

Vale lembrar que utilizamos JSON em nosso projeto, assim para fazer a instalação deste formato de dados, basta utilizar o comando abaixo:

```
$   pip install json
```

3 Explicando o código

3.1 RISC-V

A priori, cabe a nós dizer que o RISC-V é um conjunto de instruções criados com base nos princípios RISC, dentre os quais são([4] e [3]):

- Processador com pequeno número de instruções muito simples
- Instruções capazes de serem executadas em único ciclo do caminho de dados
- Operações de registrador para registrador
- Modos de endereçamento simples
- Formato de instrução simples e de tamanho fixo

3.2 Assembler

Analisando a função de um montador, podemos dizer que nada mais é do que a tradução de um código Assembly para binário. Desta forma, é válido ressaltar que a implementação do código nos próximos tópicos tem como objetivo principal a criação de uma versão simplificada do montador RISC-V.

3.3 Criação dos JSONs

Primeiramente, antes de partirmos para a demonstração do código, cabe a nós citar a criação dos JSONs(para as instruções e os registradores), o JSON é um formato que armazena informações estruturadas e é principalmente usado para transferência de dados server/client, é similar ao XML porém mais leve. Apesar do nome, o formato JSON não possui a exclusividade para uso apenas em JavaScript, podendo ser utilizado em qualquer outra linguagem que possui compatibilidade com o mesmo:

Figura 1: Instrução do tipo R

Figura 2: Instrução do tipo I

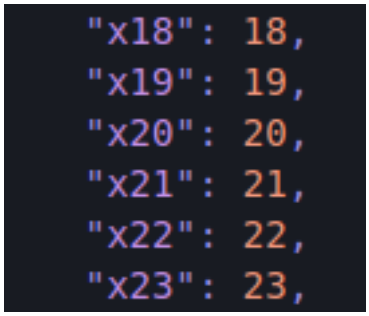

Figura 3: Exemplo de alguns registradores

3.4 Import dos arquivos

Em segundo lugar, devemos constatar que para o funcionamento do tipo de dados JSON é necessário a instalação de sua biblioteca como exposto anteriormente. Ademais, todos os arquivos deste formato estarão no diretório "data", além disso, são armazenados após a leitura em variáveis que serão utilizadas em todo o escopo do programa, como é evidenciado na imagem abaixo:

```python
import json

# ----------------------JSONS--------------------------

'''
    lê o json com as instruções do risc-v
'''
with open('data/instructions.json', 'r') as json_file:
    instructions = json.load(json_file)

'''
    lê o json com os registradores do risc-v
'''
with open('data/registers.json', 'r') as json_file:
    registers = json.load(json_file)
```

Figura 4: Leitura dos JSONs das instruções e dos registradores

3.5 Função Assembler

Em princípio, a função assembler faz a leitura linha a linha do código em Assembly, é separado o comando e os registradores, após essa rotina, a partir de cada tipo da instrução - formato 'R' ou 'I' - a função gera o resultado em binário de acordo com cada especificação.

Ademais, instruções do tipo 'R' por exemplo possui os campos 'funct7', 'rs2', 'rs1', 'rd', 'funct3' e 'opcode', já as instruções do tipo 'I' possui os campos 'immediate', 'rs1', 'funct3', 'rd' e 'opcode'. Vale ressaltar que cada um dos campos citados anteriormente possuem um tamanho determinado de bits e para isso foi necessário a utilização de uma função para a extensão de bits("add_zero").

Entretanto, para o devido funcionamento do código é preciso que os registradores e os campos de cada instrução estejam em binário, para isso, usamos a função 'change_reg_to_bin' que faz o 'casting' das strings para binário.

```python
def assembler(file_line):
    # lê a linha até o  primeiro espaço
    command = file_line[: file_line.find(" ")]
    regs = file_line[file_line.find(" "):].split(",")
    for i in range(len(regs)):
        # retira o espaço e o '\n' de cada registrador
        regs[i] = regs[i].strip()

    binary_result = ""
```

Figura 5: Criação da função assembler

```
if instructions[command]['type'] == 'r':
    f7 = instructions[command]['funct7']
    rs2 = change_reg_to_bin(regs[2])  # terceiro reg
    rs1 = change_reg_to_bin(regs[1])  # segundo reg
    rd = change_reg_to_bin(regs[0])  # primeiro reg
    f3 = instructions[command]['funct3']
    opcode = instructions[command]['opcode']
```

Figura 6: Chamada da função "change _reg _to _bin"

```
    rs2 = add_zero(rs2, 5)
    rs1 = add_zero(rs1, 5)
    rd = add_zero(rd, 5)

    binary_result = f7 + rs2 + rs1 + f3 + rd + opcode

elif instructions[command]['type'] == 'i':
    immediate = regs[2]  # valor imediato
```

Figura 7: Chamada da função "add _zero"

3.6 Suporte a outras bases numéricas

Ainda que não tenha sido pedido como a principal instrução a ser executada pelo trabalho, o grupo optou por fazer a implementação de outras bases numéricas para o projeto. Vale ressaltar que cada base deve estar com sua devida formatação, exemplo: o início em binário será 0b, em octal será 0o e em hexadecimal será 0x. Nas imagens a seguir veremos uma instrução com os números em decimais e em seguida a mesma instrução com outras bases numéricas e suas respectivas saídas:

src > tests > test_3.asm

```
1    andi x2, x1, 16
```

Figura 8: Instrução 'andi' com base decimal

src > tests > test_3.asm

```
1    andi x2, x1, 0b10000
```

Figura 9: Instrução 'andi' em binário

src > tests > test_3.asm

```
1    andi x2, x1, 0x10
```

Figura 10: Instrução 'andi' em hexadecimal

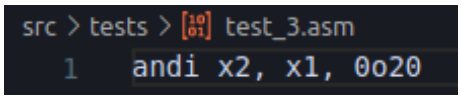

Figura 11: Instrução 'andi' em octal

Tendo dito isto, todas as bases numéricas tiveram a mesma saída assim como esperado, veja a seguir:

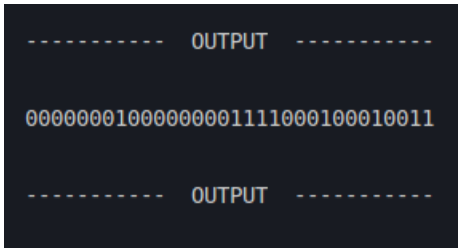

Figura 12: Output das saídas

3.7 Execução e makefile

Visto que já foi explicado as principais funções do código, agora iremos mostrar como executá-lo e como foi feito o makefile. Em síntese o makefile da aplicação possui os campos 'input' com o arquivo de entrada para teste em assembly, 'output' do arquivo de saída em binário e 'assembler' com o arquivo principal do projeto, e executa o comando:

Figura 13: Conteúdo do makefile

Assim, para executar o código basta digitar 'make' no terminal de sua máquina tendo aberto as devidas pastas:

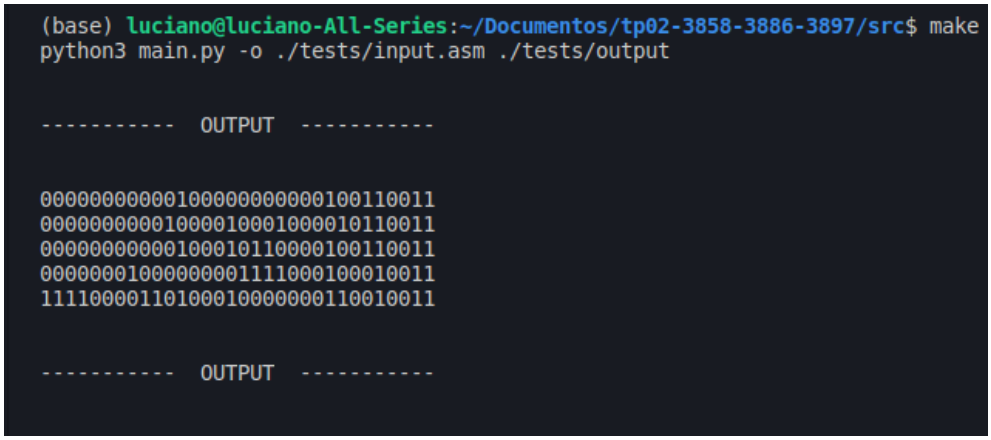

Figura 14: Execução e output do projeto

Observação: Assim como foi pedido na documentação, o código também pode ser executado via linha de comando sem a opção com arquivo de saída:

Figura 15: Execução e output do projeto sem o arquivo de saída

4 Conclusão

Considerando as medidas e análises realizadas neste trabalho, inferimos que o trabalho em questão atingiu os objetivos esperados, haja visto que a função principal do montador foi feita, traduzir a instrução em Assembly para binário. Além disso vale notar que não houve nenhum erro ou bug no projeto, seja nos inputs ou nos outputs.

Por conseguinte podemos ressaltar algumas vantagens do uso da linguagem Python para este projeto, como a simplicidade no desenvolvimento do código, a facilidade para entender o algoritmo e a compatibilidade com o formato JSON. Pode-se dizer também que o uso de JSON foi de grande ajuda no projeto, já que facilitou os imports dos dados e até a implementação do código.

Posto isso, concluimos que o trabalho foi executado conforme o planejado, sendo tratado tudo que foi pedido pelo professor e pelos monitores. Em adição, é válido dizer que apesar das dificuldades na implementação do código, o grupo foi capaz de superar e corrigir quaisquer erros no desenvolvimento do algoritmo. Por fim, verificou-se a assertiva para o objetivo do projeto em construir uma versão simplificada do montador RISC-V.

Referências

[1] About json:https://www.json.org/json-pt.html.

[2] About python:https://www.python.org/.

[3] Princípios risc 2:http://prof.valiante.info/disciplinas/hardware/risc-e-cisc.

[4] Princípios risc:http://www.ifba.edu.br/professores/antoniocarlos/index_arquivos/risccisc pipeline.pdf.

[5] What is assembler:hhttps://searchdatacenter.techtarget.com/definition/assembler.