Trabalho final de Sistemas de Programação

Lucas Yuji Harada

May 11, 2020

Contents

1 Objetivo

Para a primeira etapa do projeto, o objetivo é desenvolver três peças essencials a qualquer computador:

- Loader
- CPU (Von Neumann Machine)
- Assembler

2 Desenvolvimento

2.1 CPU

Baseado em princípios de test-driven-development (TDD), eu decidi primeiro criar os testes de cada função a ser executada pela CPU (fetch, decode, add, load, etc). Como o enunciado não especificava, fiz algumas considerações sobre o tamanho de cada um dos components:

- memória: 12 bits (4096 endereços)
- contador de instruções (PC): 12 bits
- acumulador: 8 bits

O motivo para a escolha da memória com 12 bits é devido ao tamanho do operando. A única operação de OS call implementada até o momento é a de argumento 0, que realiza a saída do programa.

2.2 Loader

O loader foi escrito em assembly, e possui 59 bytes. GOAL_ONE e GOAL_TWO são as variáveis que guardam o MSB e LSB do endereço do programa a ser carregado. Ao final do procedimento de load, o loader automaticamente move o Program Counter para o endereço inicial do programa carregado.

```
@   0
START
            GD  0
            MM GOAL_ONE
            +   NINETY
            MM  FIRST_BYTE
            GD  0
            MM GOAL_TWO
            MM  SECOND_BYTE
            GD  0
            MM  LENGTH
REPEAT
            GD  0
            JP  FIRST_BYTE
RETURN
            LD  SECOND_BYTE
            +   ONE
```

```
            MM  SECOND_BYTE
            JZ  CARRY
CHECK_IF_DONE
            LD  LENGTH
            -   ONE
            MM  LENGTH
            JZ  FINISH
            JP  REPEAT
CARRY
            LD  FIRST_BYTE
            +   ONE
            MM  FIRST_BYTE
            +   ONE
            JP  CHECK_IF_DONE

FIRST_BYTE   K  00 ;test
SECOND_BYTE  K  00
            JP RETURN
LENGTH       K  00
ONE          K  01
NINETY       K  /90
FINISH
GOAL_ONE     K 0
GOAL_TWO     K 0
#  START
```

2.3 Assembler

Assemblers são programas que convertem mnemônicos assembly (ASM) para código de máquina. Podem ser de um ou dois passos. Assemblers de dois passos primeiro precisam gerar uma tabela de símbolos para depois conseguir traduzir à código de máquina, enquanto que assembler de um passo fazem o processo de coletar referências ainda não resolvidas, símbolos, e a montagem tudo ao mesmo tempo.

Nesse projeto eu decidi criar um assembler de dois passos por simplicidade. Para a primeira entrega, o assembler não trata de casos em que existem mais de uma psudo instrução de start (@). Outra restrição é que, quando usar símbolos (como START/CARRY no exemplo do loader), é necessário que elas estejam em linhas separadas de comandos.

3 Uso

Requisitos mínimos:

- python 3.8+
- Linux

Opcional:

- git

3.1 Instalação

3.1.1 TODO Double check the instalation procedure, maybe add a windows step-by-step as well? or pyinstaller

1. Dentro de um terminal emulator, usar o comando:

   `git clone https://github.com/Adarah/2020-PCS3216-9345853.git --shallow`

   - Caso não tenha o git instalado, pode baixar o código fonte manualmente do site da disciplina: `https://sites.google.com/view/2020-pcs3216-9345853`

2. Entrar na pasta que foi baixada: `cd 2020-PCS3216-9345853`
3. Executar `pip install .`

3.2 Testes

Usando a biblioteca pytest e hypothesis, cada um das minhas funções recebem ao menos 200 inputs aleatórios, o que ajuda a garantir que o código funciona como esperado. Para iniciar uma sessão de testes, basta executar o comando `pytest` na raiz do diretório onde o código foi baixado.

Um programa exemplo que calcula os 12 primeiros dígitos da sequência fibonnaci foi adicionado como teste para verficar o funcionamento do assembler.

3.3 Uso normal

O script `src/assembler.py` aceita um argumento opcional com a flag `-f`, que é o caminho até o arquivo que se deseja montar. Caso nenhum argumento

seja fornecido, ele monta o arquivo `src/data/fibonacci.asm` por default. Para realizar a montagem, use o comando seguindo esse padrão:

```
python3 src/assembler.py -f my_custom_assembly.asm
```

O novo binário será salvo em `src/data/program.bin`. Em máquinas Linux, caso deseje verificar o conteúdo binário gerado pelo montador, use o comando `hexdump -C src/data/program.bin`

Com o programa já montado, agora rode o script `python3 src/cpu.py`, e a execução irá iniciar automaticamente. Os outputs do programa serão guardados no arquivo `src/data/output.txt`