Compilador e CPU do SAP 2

Por Samuel Lobato

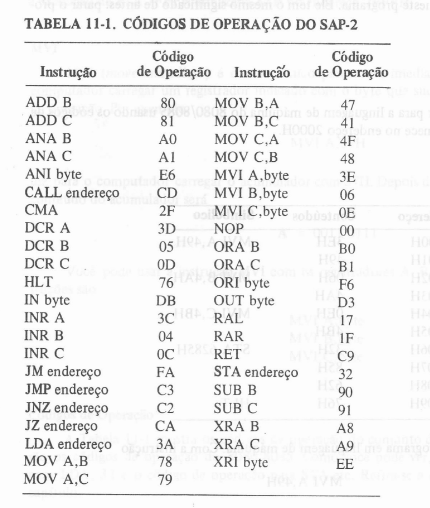
Índice:

1. AST (Assembler SAP 2)
2. CST (CPU SAP 2)
3. ACST (AST + CST)
4. Os Scripts
   1. Config.txt
   2. Input.txt
   3. Assembly.txt
   4. Hexadecimal.txt
5. Ponteiros
6. Recomendações

**Versão Beta**1-AST

O Compilador

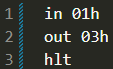
O programa AST traduz os mnemônicos em números em hexadecimal se baseando na tabela abaixo, extraída do livro contido na bibliografia:



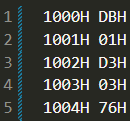
Existem duas formas de executar AST pelo terminal, usando dois ou um arquivo .txt. Vale ressaltar que como o programa é em Java, é preciso um interpretador do programa compilado de .java para .class, desse modo, o primeiro argumento é o interpretador ‘java’:

Exemplo: 

O segundo argumento é o programa AST, o terceiro é o arquivo que AST coletará o código assembly, o quarto é o arquivo que AST colocará o código hexadecimal tendo como base o quinto argumento, que é a posição inicial de impressão na memória RAM do SAP2. Como exemplo, caso file1.txt fosse:



O arquivo file2.txt assumiria, após executar o comando em questão como no exemplo abaixo, caso não exista, seu arquivo será criado automaticmamente, diferentemente de file1.txt que precisa existir:



Por outro lado, é possível executar AST usando somente um arquivo, como no exemplo abaixo:

Exemplo: 

Supondo file1.txt sendo o mesmo do exemplo anterior, AST coletará o código assembly de file1.txt e depois colocará o código hexadecimal no próprio file1.txt. O quarto argumento agora assume a função do quinto argumento do exemplo anterior, a de informar a posição na memória. Desse modo, após executar o comando em questão, file1.txt assumiria exatamente o que está contido no file2.txt do exemplo anterior.

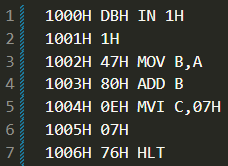
2-CST

A CPU

O programa CST baseia-se nos mesmos mnemônicos da tabela que se encontra no início do tópico anterior, fazendo as funções especificadas no livro contido na bibliografia, para executá-lo basta colocar como argumento o arquivo .txt em que será lido os hexadecimais:

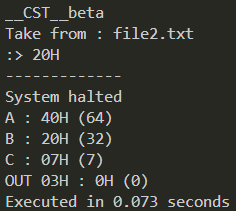
Exemplo: 

Supondo file2.txt sendo:



**showMnemonics** e **upperAllLetters \_when\_showMnemonics** no alto (Explicado no cap. 4)

Após a execução do comando acima e com Input.txt recebendo o número ‘20h’ produzirá a saída:



Na saída, depois de cada acumulador, sempre será o valor em hexadecimal e depois o valor correspondente em decimal entre parênteses, o mesmo ocorre na saída OUT.

3-ACST

AST + CST = ACST

O programa ACST faz nada mais nada menos que executar ACS e CST podendo até compilá-los antes de executar como pelas configurações que serão explicadas no próximo capítulo. Para executar ACST existem duas formas, como aquelas apresentadas no primeiro capítulo para AST:



O arquivo file1.txt contém o código assembly, os quais são compilados por AST e colocados em file2.txt com o início da memória em 1000H, já a forma mais simples coloca tudo em file1.txt:



4-Os Scripts

O projeto em questão foi feito em 3 programas na linguagem Java (ACST, AST e CST) os três capítulos anteriores trataram de cada um deles. Necessita-se também de outros arquivos para o funcionamento dos programas, sendo eles:

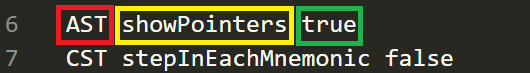
1. Config.txt
2. Input.txt
3. Assembly.txt
4. Hexadecimal.txt

Vale ressaltar que todos os 4 acima possuem a extensão ‘.txt’, explicados nos 4 tópicos abaixo:

**4.1-Config.txt**

Arquivo para configurações dos programas

As configurações dos programas foram divididas para que quando lidas, um programa qualquer entenda que tal configuração pertence ou não ao programa em questão. Por exemplo, na execução de AST, tendo em vista parte do arquivo Config.txt abaixo:



Vermelho: Bloco de Identificação de Programa (BIP);

Amarelo: Bloco de Identificação de Configuração (BIC);

Verde: Bloco de Identificação de Valor (BIV);

O programa lerá a linha 6, pois contém no primeiro o BIP correspondente ao nome do programa, já na linha 7, como o BIP não corresponde ao programa, ele pulará para a próxima e assim por diante. Quando BIP é correto, o programa verifica se BIC existe no programa em questão, caso falso, um erro é produzido. Caso contrário é passado para BIV onde os dois únicos valores permitidos são ‘true’ e ‘false’, para identificar se tal configuração vai ou não ser ativa. Com base nisso, abaixo consta a lista das configurações, todas ativam caso BIV é ‘true’:

**ACST**

* **compileACST**: Compila ACST;
* **compileAST**: Compila AST;
* **compileCST**: Compila CST;

**AST**

* **showMnemonics**: Mostra os mnemônicos relacionados aos códigos em assembly no arquivo hexadecimal;
* **upperAllLetters\_when\_showMnemonics**: Escreve os mnemônicos em letra maiúscula;
* **showPointers\_when\_showMnemonics**: Mostra os ponteiros invés dos endereços;

**CST**

* **stepInEachMnemonic**: Espera em cada execução de um mnemônico por 0.1 segundos, mostra os valores dos acumuladores A, B e C e a instrução base, por exemplo: *MVI A,2H* se torna *MVI A,byte;*
* **clearTerminal\_when\_stepInEachMnemonic**: Imprime um espaço equivalente a 40 ‘\n’’s;
* **show\_when\_useOutput**: Mostra o output quando usado;
* **delay\_when\_showAnd\_useOutput**: Espera um tempo de 1 segundo ao usar algum output;
* **showAllOutputs\_when\_useOutput**: Mostra todos os outputs;
* **noCut\_when\_stackOverflowHardware**: O SAP 2 utiliza até 8 bits para uma dada área da memória ou valor dos registradores, isso implica que CST vai cortar uma informação (String) que passa de 8 bits. Desse modo, quando a configuração em questão está no alto, a informação não tem limite de tamanho.

**4.2Input.txt**

Arquivo para receber inputs do teclado

Com o intuito do de usar o mnemônico *IN byte,* foi criado o arquivo Input.txt, pois quando se executa o programa CST separadamente, a função para ler o buffer lê o teclado a partir do terminal, o qual, por motivos óbvios, é o mesmo de onde CST foi rodado. Entretanto quando se usa ACST para executar os programas, cria-se um ProcessBuilder no próprio Java, criando um terminal à parte, sendo este inacessível para impressão dos caracteres do teclado, sendo necessário um local exterior para ler os dados, o Input.txt.

Quando o mnemônico *IN byte* for executado, CST procura em Input.txt algo para ler, que necessariamente é um byte, podendo ser em decimal ou hexadecimal, sendo necessário um sútil diferenciador (‘H’ ou ‘h’):

 Lê se 14 em decimal;

 Lê se 14h em hexadecimal; (20)

 Lê se 14H em hexadecimal; (20)

A entrada de Input.txt é **extremamente sensível** a qualquer tipo de entrada estranha, desse modo, recomenda-se o uso de somente uma linha do arquivo em questão, pois é somente ela que será lida. O programa CST espera o usuário inserir algo em Input.txt, ou seja, fica lendo-o até que haja algo.

Além disso, a entrada se baseia em bits binários, por conta do primeiro bit, o bit de sinal, calculada a partir de uma **função somática** para os decimais e uma **função conversiva** para os hexadecimais, por exemplo:

**Decimal 14:** A **função somática** identifica que o número é positivo, logo soma o binário ‘0001’ (1) ao número ‘0000’ (0), até que o valor em binário seja igual ao valor em decimal 14:

00 + 01 = 01; 01 + 01 = 0010; ... ; 0000 1101 + 01 = 0000 1110 (14)

Caso negativo, soma-se ‘1001’ ao número ‘0000’ até que o valor em binário seja igual ao valor inserido. Vale notar que a quantidade de bits vai sempre em múltiplos de 4, para facilitar os cálculos:

 Lê se –8 em decimal = 1000 (bit de sinal) 1000 (número)

**Hexadecimal 14H:** A **função conversiva** varre cada algarismo identificando seu valor em binário, e depois os concatena:

1: 0001; 4: 0100; 1h + 4h = 0001b + 0100b = 0001 0100 (20)

Desse modo, caso o usuário insira um valor, que em binário coincidentemente traduza em um número que comece com ‘1’ (bit de sinal negativo), o número será então negativo, apenas na função conversiva, já que na somática ocorre uma soma para identificar o número binário:

 Lê se 81h em hexadecimal = 1000 0001 (-1)

 Lê se 81 em decimal = 0101 0001 (81)

Para contornar essa situação, recomenda-se colocar sempre o algarismo ‘0’ para números positivos e ‘8’ para negativos, na frente de um número, sempre que este for hexadecimal, pois ‘8’ e ‘0’ traduzem em:

 Lê se 081h em hexadecimal = 0000 1000 0001 (129)

 Lê se 881h em hexadecimal = 1000 1000 0001 (-129)

Outros exemplos:

 Lê se A4H em hexadecimal = 1010 0100 (-36)

 Lê se 0A4H em hexadecimal = 0000 1010 0100 (164)

 Lê se –40 em decimal = 1001 1000 (-40)

**4.3-Assembly.txt**

Recebe os mnemônicos

Em suma, o arquivo que recebe os mnemônicos é usado somente por AST, é nele que o programador insere os mnemônicos, os quais são interpretados por AST, passando para o arquivo hexadecimal.

O programa AST lê, linha por linha, todo o arquivo assembly identificando as instruções em mnemônicos e transformando-as as código hexadecimal, pulando a linha quando percebe um **comentário de linha**, **à linha** ou **de bloco**.

Notação:

* Linhas interpretadas: verde
* Linhas ignoradas: vermelho
* Até onde foi interpretado: amarelo

**Comentário de Linha**

Para usar um comentário de linha basta colocar os caracteres ‘--’ numa linha, ele toda será ignorada:

**Comentário à Linha**

Para usar um comentário à linha basta colocar o caractere ‘;’ numa linha, tudo que vem após ele vai ser ignorado, até a próxima linha:

**Comentário de Bloco**

Para usar um comentário de bloco basta usar ‘#’ para abrir o bloco e o mesmo caractere para fechar o bloco, tudo que estiver dentro dele será ignorado, inclusive a linha de sua declaração. AST passa a ler as interpretar as linhas quando o bloco é fechado, ademais no próprio fechamento do bloco, a linha é ignorada, como abaixo:

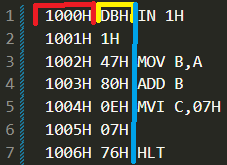
Ambos três códigos assembly geram o mesmo código hexadecimal, inclusive caso se use um comentário da linha, se showMnemonics estiver no alto, a parte da linha que está comentada não aparecerá no código assembly.

Além dos mnemônicos e dos comentários, o código assembly também pode conter uma forma melhor de lidar com endereços, que será tratada no próximo capítulo.

**4.4-Hexadecimal.txt**

Arquivo que somente recebe códigos em hexadecimal

O programa CST é responsável por ler o arquivo hexadecimal e executar as instruções, como o exemplo do capítulo 2:



O bloco vermelho possui o endereço de memória, o amarelo o valor que está em tal endereço, depois da barra azul, CST não lê nada, as instruções que lá estão servem só para ajudar o leitor a saber o que está contido em tal posição da memória.

**5-Ponteiros**

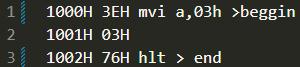
Endereçamento simples

O endereçamento da memória pode ficar bem complicado, especialmente quando se usa qualquer uma das instruções que mudam o valor do **contador de programa.** Uma forma de contornar isso é usando uma forma de coletar e colocar o endereço dependendo de algo controlado pelo programador.

Os ponteiros, em AST, é uma classe que armazena seu nome e o endereço. O processo de interpretação dos ponteiros é feito da seguinte forma, AST primeiramente coleta o endereço dos ponteiros numa primeira iteração, e depois, no processo de compilação, o ponteiro a ser utilizado é substituído pelo endereço, podendo desse modo se usar o ponteiro antes ou depois de sua atribuição. Os exemplos abaixo vão exemplificar isso:

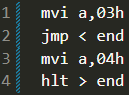


Na primeira linha, o endereço de memória da instrução MVI A*03H* é armazenado no ponteiro cujo nome é ‘beggin’. Já na última o ponteiro ‘end’ recebe o endereço de memória da instrução HLT. Vale considerar que o ponteiro armazena o endereço do começo da instrução, desse modo, considerando que o programa foi armazenado em 1000h:

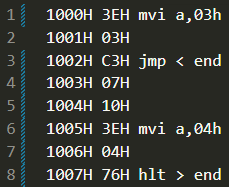
 beggin : 1000H; end : 1002H

**showMnemonics** e **showPointers** no alto

O sinal ‘>’ é usado para inserir o endereço de memória no ponteiro onde foi chamado. Já o sinal ‘<’ insere o endereço de memória do ponteiro em certa posição de memória, como um endereço de memória possui 2 bytes, e a memória possui somente 1, o endereço é dividido em duas partes, da mesma forma que o SAP 2 trata os endereços:



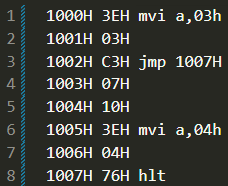
O acumulador recebe 03h, após isso o contador de programa sofre é saltado para o endereço de ‘end’, que equivale ao endereço de *HLT*, resultando em:



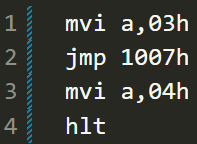
Outrora se compilar novamente usando:

upperAllLetters\_when\_showMnemonics false; showMnemonics true; showPointers false

Resultará em:



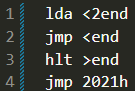
No final da execução A: 03h. Desse modo, vale ressaltar que usar endereços nominalmente, ou seja, com o seu valor propriamente escrito não muda nada na execução nem no arquivo compilado:



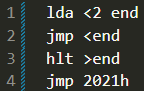
Produzindo:

Percebe-se que o código compilado não muda (CST não lê nada depois da informação da memória (barra azul)). A única diferença é que desse modo a leitura fica mais complicada, por parte do programador, pois não se sabe o que será lido na memória depois da execução da linha 2 do código assembly, \_top\_ somente depois de compilá-lo.

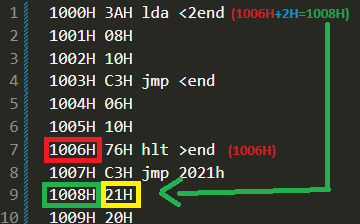
Um agente facilitador complementar aos ponteiros, são os **add-pointers**. Em suma, imediatamente após o emprego dos sinais ‘>’ e ‘<’ adiciona-se um valor de 0 a 9 (incluindo), esse valor será somado ao endereço de memória do ponteiro. Invasivo? Sim, mas genial, com isso pode-se mudar um pouco a posição de memória para acessar outros dados:



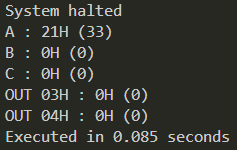
Vale constar que, quando se usa add-pointers, não é necessário dar um espaço entre o valor e o nome do ponteiro, mas fica mais legível:



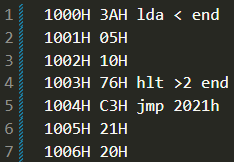
Na primeira linha o acumulador é carregado com o valor que se encontra no endereço do ponteiro ‘end’ mais duas unidades:



O ponteiro ‘end’ é carregado, na linha 7, com o endereço 1006H. Ao ser utilizado na linha 1, o valor do endereço é acrescido de 2 unidades, acessando desse modo o endereço 1008H, o qual possuí o valor 21H:



Além de usar add-pointers no uso do ponteiro, pode ser usado na atribuição:



Já na atribuição do ponteiro, o endereço é somado 2 unidades, fazendo com que, ao ser utilizado, sempre se refira a memória 1005H (1003H + 2H).

6-Recomendações

Para os apressados e perdidos

Todos os programas aqui citados estarão, até o presente momento, no meu GitHub **SGL314** no repositório **OCS\_2024**.

Versão do Java:

**java version "21.0.1" 2023-10-17 LTS**

GitHub:

**https://github.com/SGL314**

**https://github.com/SGL314/OCS\_2024**

Bibliografia

Digital Computer Electronics de Albert Paul Malvino

Aulas de OCS lecionadas por Humberto Nubuyoshi Honda

Belo Horizonte, Samuel Gonçalves Lobato 07/06/2024