Jairo Scherrer Júnior

Software Simulador de Microprocessadores 8085

Trabalho final apresentado aos coordenadores do curso de PMC 581 – Projeto Mecânico II do Departamento de Engenharia Mecânica da Escola Politécnica da Universidade de São Paulo

9,2 (nove e dois)

São Paulo
1998

Jairo Scherrer Júnior

Software Simulador de Microprocessadores 8085

Trabalho final apresentado aos coordenadores do curso de PMC 581 – Projeto Mecânico II do Departamento de Engenharia Mecânica da Escola Politécnica da Universidade de São Paulo

Área: Engenharia Mecânica
Habilitação: Automação e Sistemas

Orientador: Jun Okamoto Jr.

N° USP 743170

São Paulo
1998

Agradecimentos

- A meus pais, por todo o amor e dedicação que sempre demonstraram por mim, apoiando-me nos momentos difíceis e estando presente nos momentos alegres da minha vida.

- Ao meu orientador, por seu entusiasmo e motivação que sempre contribuíram para que este trabalho fosse feito da forma mais agradável possível.

- A minha família, que sempre esteve a meu lado e que possibilitou que eu tivesse ao meu alcance todos os recursos necessários a conclusão deste curso.

- Aos meus amigos, pelos momentos que passamos juntos e pelo aprendizado mútuo que continua existindo em nossas vidas. Decidi não citar nomes porque posso cometer a injustiça de esquecer alguém, já que são tantas as pessoas que participaram e participam da minha vida.

Índice

Objetivos:

O objetivo deste trabalho é implementar um software simulador de microprocessadores 8085 para ambiente Windows 95. O simulador deverá permitir ao usuário acompanhar o estado dos registradores, flags, memória e IO a cada passo de execução de um programa escrito em padrão LST. Pretende-se ainda que o software seja colocado na Internet com a finalidade de alcançar um número maior de usuários.

Introdução:

O presente projeto visa implementar em ambiente Windows 95 um simulador de microprocessadores 8085. Um simulador é um programa que permite verificar o funcionamento que o microprocessador terá ao se executar um programa em assembler antes mesmo de que este programa esteja gravado em uma EPROM. Com isto, pode-se testar se o algoritmo desenvolvido está realmente satisfazendo o objetivo à que se propõe.

Com o simulador, pode-se achar os erros que o programa em assembler possui de maneira mais rápida e fácil. É possível monitorar as ações que acontecem internamente, passo a passo, verificando-se cada registrador e cada flag da CPU. Sem o simulador, a localização de erros em programas é certamente mais difícil.

O processo convencional para teste de programas em microprocessadores segue o seguinte padrão: Implementa-se o código em assembler; grava-se o mesmo em uma EPROM e testa-se o sistema como um todo. Caso o sistema funcione a todos os testes, então assume-se que o programa esteja correto. Caso o sistema não funcione da maneira adequada, localizar o lugar aonde o erro aconteceu torna-se uma tarefa complicada, já que é impossível monitorar diretamente o funcionamento da CPU.

O programador tem então o papel de tentar identificar o erro através dos sintomas apresentados durante o teste, ou ainda simular o código mentalmente até achar o lugar em que houve o erro. Dependendo do tamanho do código e da complexidade do mesmo, tal trabalho pode exigir muito tempo e paciência.

Após ser identificado o erro, é necessário regravar a EPROM e fazer um novo teste. Se novamente ocorrer qualquer falha no comportamento do sistema, repete-se todo o procedimento, desde a identificação do erro, regravação da EPROM até a execução de um novo teste. Para cada erro encontrado, é necessária uma nova gravação; e portanto, o tempo gasto em se eliminar os erros é cada vez maior. Em um simulador, caso seja necessária alguma correção na código, isto pode ser feito a qualquer momento, bastando para isso, editar o código diretamente no computador.

Outra característica de um simulador é a possibilidade de realizar vários testes em um tempo menor do que implementando diretamente o código na EPROM. Logo pode-se fazer um número de testes maior e portanto assegurar uma confiabilidade melhor ao funcionamento do sistema.

O simulador é implementado totalmente em inglês tendo em vista o objetivo de disponibilizá-lo a usuários de outros países por meio da Internet. No Anexo, encontra-se o help implementado para o programa, onde pode-se verificar este fato.

Estrutura interna de um microprocessador 8085:

Para que se conheça o estado de um microprocessador é preciso que se conheça o conteúdo de seus registradores e flags. O 8085 possui o seu conjunto próprio de registradores e flags, que diferem de microprocessadores de outras famílias. Por esta razão será detalhado qual o conjunto de registradores e flags compõe um microprocessador 8085. A partir da figura 1, mostrada a seguir, pode-se verificar quais

são os componentes internos de um 8085 e qual o sentido do fluxo de informações entre os mesmos:

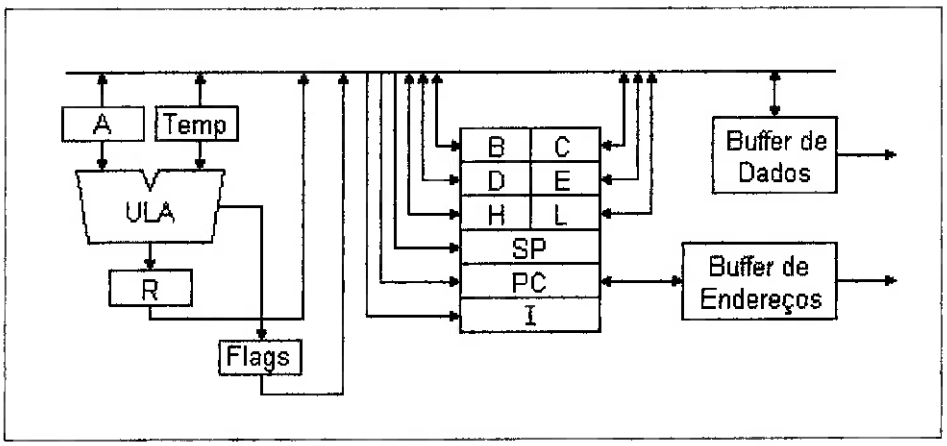


Figura 1 – Estrutura interna de um microprocessador 8085

O principal componente é a unidade lógica aritmética, ou simplesmente ULA. É este o componente que é responsável pela execução das instruções do programa e também por todo processamento de dados. A ULA recebe um sinal de controle que indica qual a instrução está sendo executada e a partir desse sinal, coordena todo o processamento de informações. Os dados necessários para se completar a instrução são adquiridos através dos registradores A e Temp. Após a execução do comando, o resultado é colocado no barramento através do uso do registrador R.

O segundo tipo de componente são os registradores. Registradores têm a função de armazenar dados a fim de que os mesmos ou sejam processados pela ULA ou então colocados em algum lugar específico da memória. Alguns tipos, como PC, SP, e I tem funções especiais que destinam-se a auxiliar a execução de um programa.

Alguns dos registradores não são totalmente acessíveis para o usuário. Por exemplo, o registrador I que guarda as instruções só é acessível para a escrita, não o sendo para a leitura. Os registradores Temp e R são de uso interno do

microprocessador e têm a função de realizar armazenamento de dados temporários, sendo totalmente inacessíveis ao programador.

O principal registrador é o acumulador, também chamado de registrador A. Isto porque ele é o único dos registradores que pode ser diretamente acessado pela unidade lógica programável e também pelo programa em execução. Logo, qualquer instrução que seja executada utiliza o mesmo para colocar os dados que necessitam de processamento na ULA.

Além do acumulador, o 8085 possui dois pares de registradores que são de uso geral para armazenamento de dados. Estes registradores são denominados par BC e par DE. Existe ainda um outro par disponível para utilização que é o par HL, porém sua ênfase é de se guardar endereços de memória, já que existem instruções próprias no assembler do 8085 que fazem endereçamento do tipo indireto utilizando o mesmo.

O par SP (stack pointer) é responsável por apontar a localização da pilha de dados do 8085. Esta pilha de dados, implementada em uma região da memória, é utilizada para se armazenar os conteúdos dos registradores e os endereços de memória para retorno de subrotinas. Como o número de registradores é relativamente pequeno, existem várias situações em que comandos alteram totalmente o conteúdo dos mesmos. Logo um procedimento que é constantemente executado é o de se guardar o conteúdo dos registradores na pilha, executar a instrução e em seguida recuperar o conteúdo da pilha. Todo este procedimento é feito utilizando-se o par SP como indicador da posição da pilha.

O par PC (program counter) é utilizado para indicar qual a posição na memória que contém a próxima instrução a ser executada. O programa é armazenado em uma região especifica da memória, definida pelo programador. Logo, para se executar o mesmo, deve-se inicializar o par PC com o endereço da primeira instrução. A cada

execução de um novo comando, o par PC é incrementado atingindo assim o próximo comando da lista.

O último registrador do 8085 dedica-se somente a armazenar dados sobre o controle de interrupções e estados da entrada e da saída serial.

As flags são elementos que indicam condições que resultaram de determinadas operações. No caso do 8085 existem cinco flags que são descritas a seguir:

- Flag Zero (Z): Esta flag indica se o resultado de uma operação é nulo ou não. Caso seja nulo, a flag assume valor unitário. Caso seja diferente de zero, a flag assume valor zero.
- Flag Sign (S): Esta flag indica se o bit mais significativo do resultado é igual a um. Se for observada esta condição, o flag assume valor unitário. Caso contrário, a flag assume valor zero.
- Flag Parity (P): Esta flag indica se a paridade do resultado é par. Paridade é uma propriedade numericamente igual ao resto da divisão da soma dos bits de um byte por dois. Se a paridade do resultado for par, a flag assume valor unitário. Caso a paridade seja impar, a flag assume valor zero.
- Flag Carry (CY): Esta flag indica se o resultado de uma operação de adição resultou em um número maior do que FFh ou se houve a necessidade de um "borrow" no caso de operações de subtração e comparação. Caso tenha ocorrido uma destas situações, a flag assume valor unitário. Caso não se observe a ocorrência das mesmas, a flag assume valor zero.
- Flag Auxiliary Carry (AC): Indica se a soma dos terceiros bits dos dados de entrada resultaram em um "carry" para o quarto bit do resultado. A flag assume valor unitário se for observado tal situação e valor zero em caso contrário.

O 8085 possui ainda duas estruturas que são o buffer de dados e buffer de endereços. Tais estruturas tem a finalidade única de servir de interface com os componentes externos que estão conectados ao microprocessador.

Memória e I/O associados ao microprocessador 8085:

O 8085 possui 16 pinos que servem para endereçamento de memória. Logo, a faixa de endereços de memória começa em 0000h e termina em FFFFh. Para o I/O, é possível se endereçar somente endereços entre 00h e FFh.

Interrupções do microprocessador 8085:

Interrupções são desvios na execução do programa em decorrência de um evento externo. Geralmente, seu uso mais comum é o de se processar algum evento que está ocorrendo em um determinado periférico.

O 8085 possui 5 interrupções:

- Trap – Não mascarável - Endereço de resposta: 0024h;
- RST 7.5 – mascarável – Endereço de resposta: 003Ch;
- RST 6.5 – mascarável – Endereço de resposta: 0034h;
- RST 5.5 – mascarável – Endereço de resposta: 002Ch;
- INTR – Endereço de resposta é fornecido à CPU.

A interrupção TRAP é a que tem a maior prioridade, e portanto não é mascarável nem é afetada pelo flip-flop de habilitação de interrupções. Seu uso em sistemas controlados pelo 8085 é geralmente o de atender a situações críticas, tais como queda de voltagem, acionamento de modo de emergência,...

As interrupções RST 7.5, RST 6.5, RST 5.5 são mascaráveis e afetadas pelo flip-flop de habilitação de interrupções. Portanto, para se utilizar qualquer uma das mesmas, deve-se modificar o registrador de máscaras de interrupção através da instrução SIM e habilitar as interrupções através da instrução EI. A RST 7.5 difere das demais na medida em que seu acionamento é do tipo sensível a borda enquanto que o das outras é do tipo sensível ao nível. Se uma interrupção 7.5 estiver mascarada e for requisitada, ao se desmascarar a interrupção ocorre o atendimento à mesma. Tal fato não acontece com as interrupções 6.5 e 5.5. Se o usuário não quiser que sejam atendidas interrupções 7.5 requisitadas anteriormente, o mesmo deve ressetar o flip-flop da 7.5 utilizando a instrução SIM antes de desmascará-la.

A interrupção do tipo INTR não é mascarável, mas é afetada pelo flip-flop de habilitação de interrupções. Caso as interrupções estejam habilitadas e o 8085 receba uma interrupção desde tipo, muda-se o tipo de operação do 8085. O periférico fornece qual é a instrução que o mesmo deve seguir, normalmente um CALL ou RST. Depois da mudança de PC, o 8085 volta a seu ciclo de execução normal. Por causa desta peculiaridade de dependência de um periférico externo, este tipo de interrupção não é utilizada no simulador.

Conjunto de instruções do 8085:

Vista a parte do hardware interno do 8085, deve-se verificar como é a estrutura do software do 8085. É importante observar como é a sintaxe da linguagem de programação de um 8085, já que o simulador deverá ser capaz de transformar cada comando da linguagem, no conjunto de ações que derivam do mesmo.

A linguagem de programação do 8085 é composta por 256 instruções. Essas instruções correspondem a números hexadecimais que variam entre 00h e FFh. Essas

instruções podem ser divididas em cinco grandes grupos conforme o tipo de função que desempenham. Estes grupos são citados e explicados abaixo:

- **Grupo de movimentação de dados:** este grupo possui instruções que comandam a transferência de dados entre dois registradores, ou entre registradores e memórias. Fazem parte deste grupo as seguintes instruções:

1. MOV – transfere dados entre registradores ou entre registradores e endereços de memória.
2. MVI – transfere o byte seguinte do programa diretamente para um registrador ou para um conteúdo de memória.
3. LXI - transfere os próximos dois bytes do programa diretamente para um par de registradores.
4. LDA – O conteúdo do endereço de memória definido pelos dois próximos bytes do programa é transferido para o acumulador.
5. STA – O conteúdo do acumulador é transferido para a posição de memória indicada pelos dois próximos bytes do programa.
6. LHLD – O conteúdo do endereço de memória definido pelos dois próximos bytes do programa é transferido para a posição de memória definida pelos registradores HL.
7. SHLD - O conteúdo do endereço de memória definido pelos registradores HL é transferido para a posição de memória indicada pelos dois próximos bytes do programa.
8. LDAX - O conteúdo do endereço de memória definido por um determinado par de registradores é transferido para o acumulador.
9. STAX - O conteúdo do acumulador é transferido para a posição de memória indicada por um determinado par de registradores.

10. XCHG – O conteúdo dos registradores HL são transferidos para o par DE. O par DE recebe o conteúdo prévio do par HL.

- **Grupo de operações aritméticas:** Conjunto de instruções que realizam operações aritméticas, tais como adição, subtração. É composto pelas seguintes instruções:

1. ADD – O conteúdo de um determinado registrador ou de uma posição de memória é somado ao conteúdo do acumulador.
2. ADI – O próximo byte do programa é somado ao conteúdo do registrador.
3. ADC – Realiza a mesma operação que o ADD, só que inclui-se na soma um bit que é proveniente da flag Carry.
4. ACI – Similar à instrução ADI, porém com a inclusão do bit da flag Carry.
5. SUB – O conteúdo do acumulador é subtraído do conteúdo de um registrador ou de uma posição de memória.
6. SUI - O conteúdo do acumulador é subtraído diretamente do próximo byte do programa.
7. SBB – Realiza a subtração da mesma maneira que o comando SUB, porém com a inclusão do bit da flag Carry na operação.
8. SBI - Similar à instrução SUI, porém com a inclusão do bit da flag Carry.
9. INR – Incrementa o conteúdo de um registrador ou de uma posição de memória.
10. DCR – Decrementa o conteúdo de um registrador ou de uma posição de memória.
11. INX – Incrementa o conteúdo de um par de registradores.
12. DCX – Decrementa o conteúdo de um par de registradores.
13. DAD – Soma o conteúdo de um par de registradores com o conteúdo do par HL.

14. DAA – Adapta o conteúdo do registrador para dois números decimais no padrão BCD.

- **Grupo de lógica:** Este grupo contém instruções de elementos de lógica booleana utilizados na comparação entre dois bytes. Possui também algumas instruções de manipulação de bytes. Os seguintes comando integram este grupo:

1. ANA – Realiza uma operação de AND entre o conteúdo do acumulador e o conteúdo de um registrador ou posição de memória especificados.
2. ANI – Realiza a operação de AND diretamente entre o conteúdo do acumulador e o próximo byte do programa.
3. XRA – Realiza uma operação de XOR entre o conteúdo do acumulador e o conteúdo de um registrador ou posição de memória especificados.
4. XRI - Realiza a operação de XOR diretamente entre o conteúdo do acumulador e o próximo byte do programa.
5. ORA – Realiza uma operação de OR entre o conteúdo do acumulador e o conteúdo de um registrador ou posição de memória especificados.
6. ORI - Realiza a operação de OR diretamente entre o conteúdo do acumulador e o próximo byte do programa.
7. CMP – Realiza a comparação entre o conteúdo do acumulador com um registrador ou uma posição de memória. Pode-se através das flags obter informações se os bytes são iguais e se não são, qual deles é maior.
8. CPI – Realiza a mesma operação que o comando CMP, porém diretamente entre o acumulador e o próximo byte do programa.
9. RLC - Rotaciona os bits do acumulador para a esquerda. O bit menos significativo recebe o valor do bit mais significativo.
10. RRC - Rotaciona os bits do acumulador para a direita. O bit mais significativo recebe o valor do bit menos significativo.

11. RAL - Rotaciona os bits do acumulador para a esquerda. O bit menos significativo recebe o valor da flag Carry
12. RAR - Rotaciona os bits do acumulador para a direita. O bit mais significativo recebe o valor da flag Carry.
13. CMA – Realiza uma operação de NOT em cada bit do acumulador.
14. CMC – Realiza uma operação de NOT na flag Carry.
15. STC – Impõe o valor unitário para a flag Carry.

- **Grupo de desvios de execução do programa:** São instruções que alteram a seqüência de execução normal do programa. É composto pelas seguintes instruções:

1. JMP – O programa passa a ser executado a partir do endereço especificado pelos dois próximos bytes do programa.
2. Jcondition – O programa passa a ser executado a partir do endereço especificado pelos dois próximos bytes dependendo de determinadas condições das flags.
3. CALL – Chamada de subrotina. O programa passa a ser executado a partir do endereço especificado pelos dois próximos bytes do programa e retorna para o ponto de chamada na próxima instrução RET.
4. Ccondition – Chamada de subrotina condicional. Executa o mesmo procedimento do comando call dependendo do estado de determinadas flags.
5. RET – Retorno de subrotina. Retorna à instrução seguinte à instrução CALL que realizou a chamada à subrotina.
6. Rcondition – Retorno de subrotina condicional. Retorna da subrotina dependendo do estado de determinada flag.
7. RST – Move a execução de programas para posições de reset do microprocessador.

8. PCHL – Transfere o conteúdo do par BC diretamente para o par PC.

- **Grupo de controle de pilha, I/O e do microprocessador:** São instruções únicas de controle de movimentação da pilha, da entrada e saída e da configuração do microprocessador. Fazem parte deste grupo as seguintes instruções:

1. PUSH – Coloca o conteúdo de um determinado par de registradores na pilha.
2. POP – Recupera da pilha o conteúdo de um determinado par de registrador.
3. PUSH PSW – Coloca o conteúdo do acumulador e o estados das flags na pilha.
4. POP PSW – Recupera da pilha o conteúdo do acumulador e o estado das flags.
5. XTHL – O par HL recebe os bytes que estão no topo da pilha e os bytes que estavam no par HL são colocados no novo topo.
6. SPHL – O conteúdo do par HL é movido para o topo da pilha.
7. IN – O acumulador recebe o byte que se encontra em um determinado endereço de I/O
8. OUT – O conteúdo do acumulador é transferido para um determinado endereço de I/O
9. EI – Habilita as interrupções
10. DI – Desabilita as interrupções.
11. HLT – Pára a execução do programa.
12. NOP – Não realiza nenhuma modificação no microprocessador.
13. RIM – Leitura da máscara de interrupções.
14. SIM – Escrita da máscara de interrupções.

O simulador deve reconhecer todas estas instruções e realizar todas as modificações intrínsecas a cada instrução.

Escolha da ferramenta de programação:

Para-se implementar o software simulador, escolheu-se como ferramenta de programação o software Borland C++ Builder. Este software foi escolhido pelas seguintes razões:

- Maior familiaridade do programador com as linguagens C, C++ em relação a outros tipos de linguagem.
- Linguagem orientada a objeto: O C++ possui todas as vantagens de uma linguagem orientada a objeto, ou seja, pode-se encapsular dados em rotinas em único bloco e tratá-los assim de maneira única. Outra vantagem é que as classes implementadas podem ser utilizadas em outro softwares que porventura necessitem de uma estrutura similar.
- Facilidade de implementação da interface: Normalmente a linguagem C++, para em programação em ambiente Windows, é bastante complexa no que diz respeito à implementação da interface. Isto não ocorre com o Builder, já que a ênfase do mesmo é propiciar um ambiente de fácil implementação de interface.

Estrutura do programa:

O programa tem a seguinte estrutura, mostrada na figura 2:

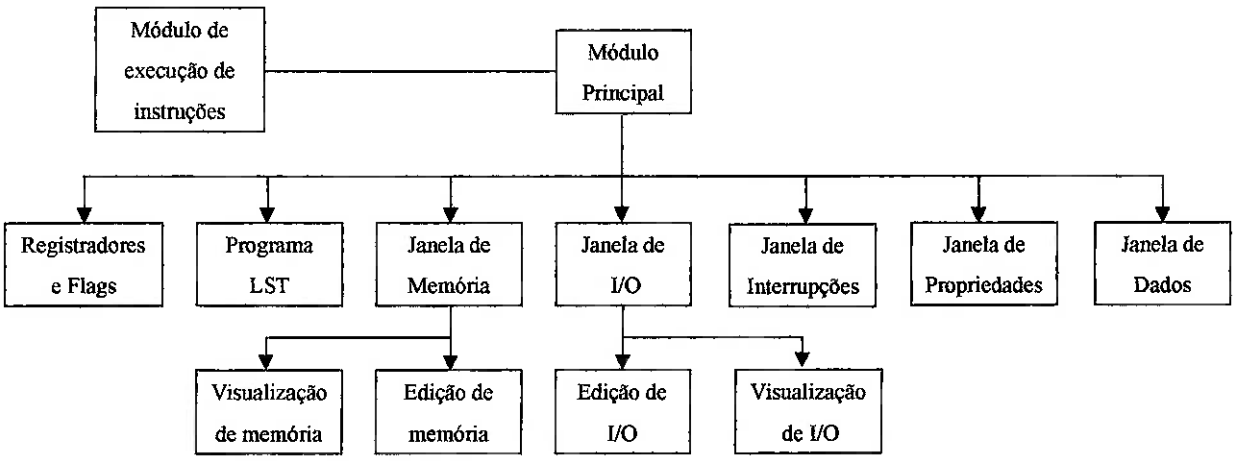


Figura 2 – Estrutura de módulos do programa

O programa simulador é implementado utilizando-se o padrão MDI (Multiple Document Interface). O módulo principal, que é implementado como sendo uma janela do tipo MDI Parent, corresponde à parte de gerenciamento do programa. É no mesmo que são implementados os menus e comandos do programa e também as rotinas de simulação.

Cada módulo correspondente a uma estrutura do 8085 (Registradores e flags, Programa LST, Memória, I/O, Interrupções) é desenvolvido como uma janela do tipo MDI Child. Dessa forma pode-se obter um controle do trânsito de dados mais organizado. Cada divisão controla uma parte da estrutura de dados e respectivas rotinas que se limitam exclusivamente à edição e visualização dos mesmos. A parte de simulação, que deve ter acesso a todas as informações, comunica-se com todas as estruturas de dados e por isso é implementada numa hierarquia superior às estruturas.

Os demais módulos são módulos de apoio, não sendo necessário desenvolvê-los como janelas do tipo MDI Child. Grande parte deles são janelas do tipo dialog já que o que se requer normalmente deles é a entrada de uma informação de uso imediato.

Comandos de simulação:

O software simulador tem basicamente quatro tipos de simulação. A seguir são descritos quais são esses tipos e em que modo os mesmos diferem entre si:

1. Simulação em modo Run:

Nesse modo de simulação é feita a execução das instruções sem que haja a atualização dos registradores, flags, memória e I/O. O programa em assembler é executado de forma continua, até que seja encontrado um breakpoint ou o usuário aperte o botão de pause ou stop. Os registradores, flags, memórias e I/O somente são atualizados quando a simulação é interrompida.

Este modo de simulação é indicado para se atingir rapidamente um ponto em que se deseja ter um maior detalhamento. Para que seja possível este procedimento, o usuário deve definir um breakpoint um pouco antes do trecho em questão. O programa é então simulado em modo Run até o que seja atingido o breakpoint. O usuário pode então mudar para um outro modo de simulação que possibilite ao mesmo um maior controle das mudanças que são efetuadas pelas instruções.

2. Simulação em modo Animate:

Nesse modo de simulação é feita uma animação contínua do funcionamento de um programa em assembler do 8085. O mecanismo de simulação tem a seguinte forma: O simulador executa uma instrução do programa. Ocorre então uma pausa de tamanho

determinado pelo usuário. Ao término da pausa, a instrução seguinte é executada . A simulação prossegue deste modo até que o usuário interrompa a execução através de um comando de parada ou até que seja encontrado um breakpoint.

A velocidade da simulação neste modo é definida pelo usuário resultando na definição do tamanho das pausas. Uma pausa maior implica em uma velocidade menor e vice-versa. Esta velocidade deve ser escolhida pelo usuário de maneira a possibilitar o acompanhamento de todas as mudanças que ocorrem com a execução das instruções em assembler.

3. Simulação em modo Step Over:

Neste modo de simulação, há a necessidade de que usuário dê um sinal de comando a fim de que cada nova instrução seja executada. Dessa forma, o período de pausa entre a execução de duas instruções consecutivas depende exclusivamente do comando do usuário. Pode-se, então, verificar todas as mudanças ocorridas nos registradores e flags, na memória e no I/O e só autorizar o próximo passo quando tudo for verificado.

Na simulação em modo Animate, corre-se o risco de um passo acontecer antes que todas as mudanças sejam verificadas. O modo Step Over é então mais indicado, portanto, quando se deseja ter uma visão mais detalhada do funcionamento do programa. Em contrapartida, o modo Animate é adequado quando o desejado é uma visão mais ampla e global da simulação.

4. Simulação em modo Step Into:

Este modo de simulação é similar ao modo anterior, porém existe a diferença de que no outro modo como é simulado o funcionamento de subrotinas chamadas pela

instrução CALL (ou CALLs condicionais). No caso de ocorrer uma subrotina em modo de simulação Step Into, a simulação passará a acontecer internamente à subrotina.

Esse modo de simulação é ainda mais detalhado que o modo Step Over e deve ser utilizado quando não se pode garantir que as subrotinas estão operando de maneira satisfatória. No caso de se ter confiabilidade no funcionamento das subrotinas, o modo Step Over torna-se mais interessante para se executar a simulação.

Além dos comandos acima, existem dois comandos de parada de simulação. Os modos de simulação Run e Step Over podem levar a situações em que ocorram loops infinitos. Por exemplo, se o modo Run for chamado sem que haja um breakpoint na listagem ou se o modo Step Over for chamado e a sub-rotina que estiver sendo executada não contiver uma instrução RET, a execução das instruções continuará indefinidamente já que não há nenhum elemento de parada.

Outro motivo para a inserção de comandos de parada é o de se interromper a animação do modo Animate. Deve existir algum modo se desabilitar a animação caso o usuário decida mudar o modo de simulação. Foram definidos então os seguintes comandos de parada:

- Pause:

Interrompe a animação ou a execução contínua. Os registradores são atualizados até a execução da última instrução e PC estará indicando o endereço da próxima instrução.

- Stop:

Interrompe a animação ou a execução contínua. Difere do Pause por alterar PC de forma a reiniciar a simulação na primeira instrução do programa.

Comandos genéricos:

O software dispõe de um menu principal e uma toolbar que permitem ao usuário executar alguns comandos. Os comandos relativos à simulação foram listados no item anterior. Neste tópico são listados comandos de apoio do simulador:

1. Comandos de Arquivos:

Neste item de menu localizam-se os comandos que são responsáveis por abrir, imprimir e fechar um programa de 8085.

O comando Open permite a abertura de um novo arquivo. Ao se acionar o mesmo, uma caixa de diálogo é aberta, e então o usuário pode escolher qual o arquivo a ser aberto. Caso já exista algum arquivo aberto, antes de se exibir a caixa de diálogo, mostra-se um mensagem para verificar se o usuário deseja fechar o programa atual. Tal mensagem é necessária, pois só é possível a simulação de um programa a cada instante.

O comando Print possibilita ao usuário imprimir desde dados de simulação, conteúdo de registradores, flags, memória e I/O até a Listagem do programa em assembler.

O comando Print Setup permite a alteração da impressora e também a alterações nas propriedades de impressão (tamanho do papel, disposição da impressão).

O comando Close permite o fechamento de um programa de 8085. Após ser realizada a execução, o usuário pode fechar o programa e assim executar uma nova simulação com outro programa.

O comando Exit permite ao usuário deixar o software de simulação. É a mesma ação que resulta de se pressionar o comando de fechar janela no canto superior direito.

Antes porém de acontecer o fechamento, o usuário é deve confirmar que deseja sair do programa. Confirmando-se a intenção, o software é fechado. Caso contrário, o software continua disponível.

2. Comandos de visualização de janelas:

As janelas de memória, I/O e interrupções podem ser habilitadas ou não para serem visíveis. Para se mudar o estado da janela quanto à visibilidade, basta acionar o nome da janela no menu View. Um sinal de check indica se a janela está visível ou não.

As janelas de Registradores e flags e do programa LST são sempre visíveis se há algum arquivo aberto. Tal circunstância deve-se ao fato de que a janela de registradores e flags representa a estrutura interna do 8085 e que o programa é essencial à simulação, não fazendo sentido a possibilidade de deixar essas janelas não visíveis.

Como o programa possui cinco janelas que podem ser dispostas conforme a utilidade para o usuário, foram implementados alguns comandos que são relacionadas ao posicionamento e dimensionamento das janelas. O comando cascade posiciona as janelas de forma cascateada, uma por sobre a outra, mas com um offset de posição que permite qualquer uma das janelas seja tornada ativa. O comando tile dispõe as janelas lado a lado da mesma maneira em que as janelas foram inicializadas. Esse comando também redimensiona as janelas de programa, memória e IO no caso de um redimensionamento da janela MDI Parent. Finalmente, o comando Arrange Icons alinha e reorganiza as janelas minimizadas.

Interface do programa:

Neste tópico são apresentadas todos os elementos que constituem o simulador. Todas as telas e janelas que são utilizadas são mostradas e em cada uma é feito um comentário sobre o papel que a mesma desempenha dentro do software e eventuais detalhes significativos em sua forma de construção.

Ao ser executado o programa, uma tela de apresentação do mesmo é exibida por cerca de quatro segundos. Essa tela pode ser vista abaixo na figura 3:

Figura 3 – Tela de apresentação

Durante os quatro segundos, a janela principal do programa é carregada e exibida atrás da tela de apresentação. Esta janela principal é a que representa o módulo principal do programa e que, por sua vez, contém as janelas filhas responsáveis pelo gerenciamento de partes diferentes da estrutura de dados. Quando a tela de apresentação deixar de ser mostrada, o programa estará pronto para receber algum comando do usuário. A configuração da tela do mesmo, nesta situação, está mostrada na figura 4:

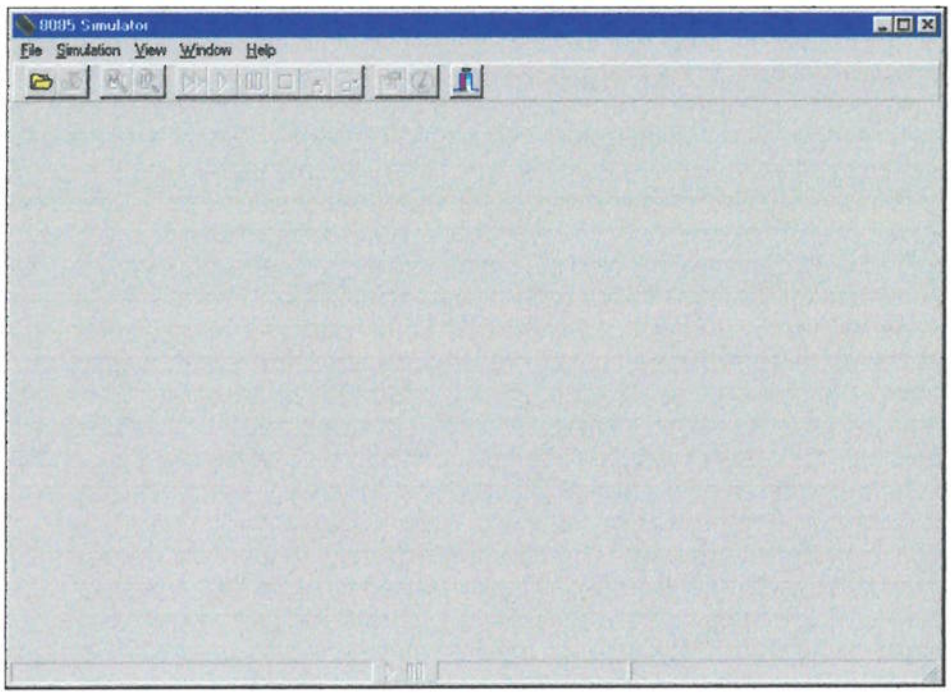


Figura 4 – Janela principal do programa

Neste instante, somente é permitido ao usuário a execução de dois comandos, pois ainda não existe nenhum programa de 8085 aberto. Estes dois comandos são: abrir um arquivo ou sair do programa. Um detalhe importante é que não é possível a criação de programas assembler dentro do simulador. Os programas LST devem ter sido compilados previamente em outro programa ou deve-se utilizar um editor externo para escrevê-los diretamente.

Quando o usuário abrir algum arquivo, são exibidas as quatro janelas do tipo MDI Child. Estas janelas são posicionadas inicialmente numa posição default, porém a localização das mesmas pode ser alterada a qualquer instante pelo usuário. A quinta janela do tipo MDI Child somente é mostrada se o usuário a requisitar. Trata-se da janela de interrupções, e para exibi-la basta escolher o item de menu Window|Interrupt.

Vários comandos são habilitados após a abertura de um arquivo válido. Uma característica do simulador é que ao mesmo tempo em que ocorre a abertura do programa, executa-se uma rotina que verifica se o programa está de acordo com o

padrão LST exigido. Caso o mesmo seja considerado inválido, uma mensagem é exibida explicando o motivo do erro e os comandos de simulação são todos desabilitados.

A tela principal, após a abertura de um arquivo, pode ser vista na figura 5:

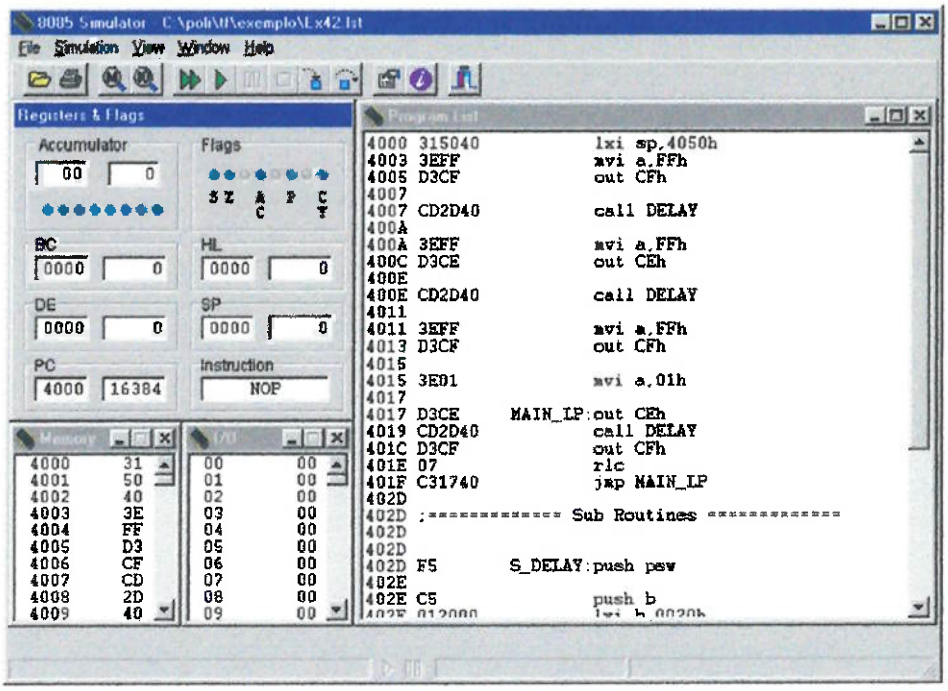


Figura 5 – Tela do programa com um arquivo aberto

As janelas que foram abertas são explicadas individualmente a seguir:

1. Janela dos registradores:

Possui o nome de Registers & Flags. Essa janela é responsável pelo gerenciamento e monitoração do estado dos registradores e das flags do 8085. A mesma possui 8 divisões: 6 representam registradores e pares de registradores (Acumulador, BC, DE, HL, SP, PC), uma representa as flags (Flags) e a última representa uma instrução de assembler (Instruction). Na figura 6, mostra-se um detalhe desta janela:

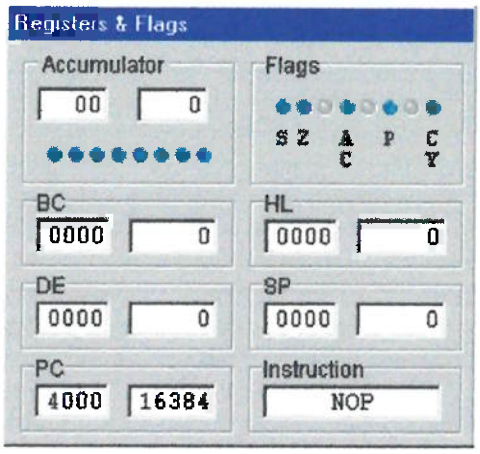


Figura 6 – Registradores e Flags

De maneira geral, cada registrador ou par de registradores possui dois campos. No primeiro, é mostrado o conteúdo do mesmo em hexadecimal e no segundo em decimal. Estes campos são alterados conforme a execução do programa, ou podem ser alterados durante a execução do mesmo pelo usuário.

O acumulador, por se tratar do principal e mais utilizado registrador, possui um outro tipo de visualização e edição do seu conteúdo. São oito leds azuis alinhados. Cada um desses leds representa um bit do acumulador. Deste modo, pode-se visualizar o conteúdo do mesmo verificando-se quais leds estão acesos e quais estão apagados da mesma forma que a representação de número binário. Cada um desses leds pode ser apagado ou aceso pelo usuário, ao se clicar em cima do mesmo.

As flags estão dispostas da mesma maneira em que as mesmas são implementadas em um 8085, ou seja através de um byte de flags. Nesta configuração, a flag Sign (S) está no bit 7, a Zero (Z) no bit 6, a Auxiliary Carry (AC) no bit 4, a Parity (P) no bit 2 e a Carry (CY) no bit 0. Os bits 1,3,5 não são utilizados. As flags foram dispostas nesta configuração para que quando ocorra uma instrução do tipo PUSH PSW, o usuário saiba qual byte é colocados na pilha juntamente com o conteúdo do acumulador. Caso fosse utilizada uma configuração mais simples, como por exemplo,

mostrar as cinco flags ligadas ou desligadas, perderia-se essa informação. Da mesma forma que no acumulador, cada uma das flags pode ser alterada pelo usuário, bastando para isso clicar em cima do led da mesma.

O campo Instruction é responsável por mostrar ao usuário qual é a instrução que está sendo apontada pelo Program Counter (PC) no programa que está sendo simulado. Isto ajuda a simulação na medida que o usuário pode prestar atenção somente na janela de registradores, se o mesmo assim desejar, não tendo que verificar em que lugar do programa a simulação se encontra. Este campo, ao contrário dos demais, não pode ser alterado pelo usuário.

2. Janela do programa :

Nesta janela, é mostrado o programa que está sendo simulado. Na figura 7, encontra-se mostrada a mesma com um programa de exemplo. O programa apresenta a estrutura bem definida com relação à localização dos endereços, instruções hexadecimais e comandos em assembler. Essa estrutura é denominada LST e será descrita no próximo tópico.

Pode-se verificar na janela que a linha que será executada no próximo passo é a que está escrita em letras azuis. Conforme o andamento da simulação do programa, esta linha se movimenta indicando em cada instante a próxima instrução a ser executada. As linhas que estão escritas em vermelho representam breakpoints da simulação. Para defini-los, o usuário deve clicar duas vezes em cima da linha desejada. Para retirá-los, executa-se o mesmo procedimento. É importante observar que os breakpoints são utilizados somente no modo de simulação Run, Animate e Step Over (dentro de subrotinas) já que nos modos Step Into e Step Over (fora de subrotinas) a simulação só é executada com o comando do usuário.

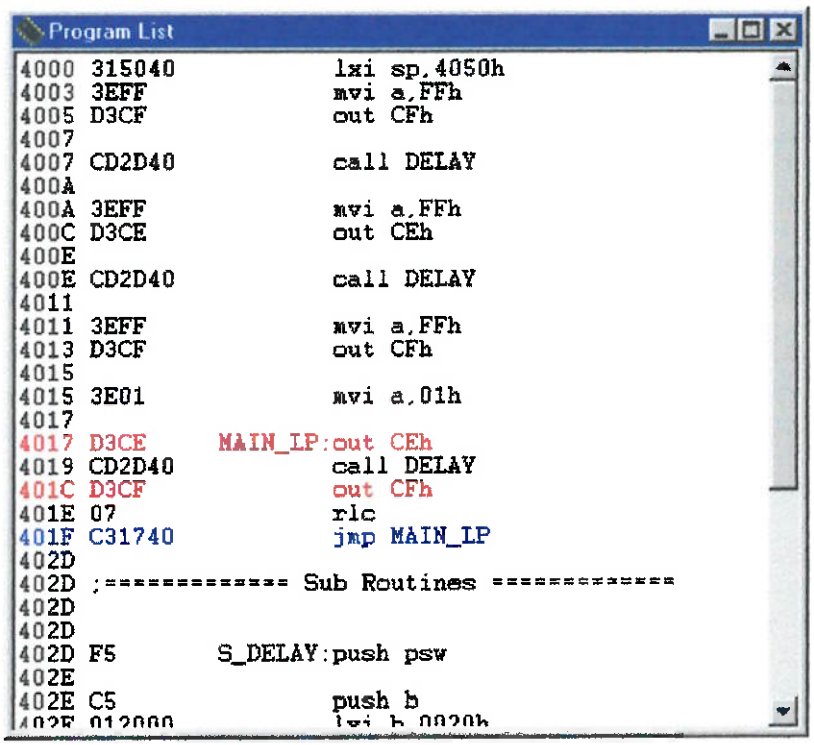


Figura 7 – Janela do programa a ser simulado

3. Janela da memória:

Esta janela apresenta o conteúdo da memória que eventualmente está associada ao 8085. Por ser uma lista relativamente grande (65536 itens), a cada momento somente é mostrada uma partição da memória equivalente a 512 itens ou 0,5 Kbytes. Ao se abrir um programa LST, a janela de memória mostrará automaticamente as primeiras posições que o programa ocupa. A figura 8 apresenta esta janela para os endereços iniciais de um programa que ocupa a posição 4000h:

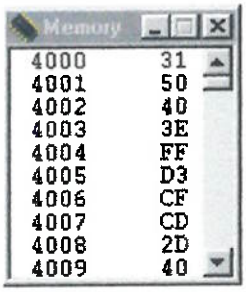


Figura 8 – Janela da memória

No caso de o usuário desejar visualizar um endereço que não se encontra na partição que está sendo mostrada, o mesmo deve acionar o comando de visualização de memória presente no menu e na toolbar. Basta entrar então o primeiro endereço da nova partição e assim visualizar o endereço de seu interesse. A figura 9 apresenta a janela que permite ao usuário modificar o intervalo de visualização de memória:

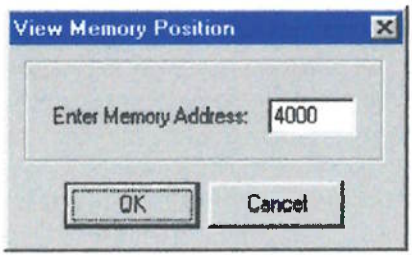


Figura 9 – Janela de mudança na visualização da memória

Para se editar o conteúdo de cada endereço de memória, basta clicar-se em cima do mesmo. Nesta situação, uma janela de edição de conteúdo de memória é aberta e pode-se entrar o novo conteúdo pelos mesmos métodos utilizados para se entrar o conteúdo do acumulador: valor hexadecimal, valor decimal ou bit a bit por meio de leds. Essa alteração é permitida tanto no estado em que não esteja ocorrendo simulação quanto na situação em que a mesma está acontecendo. Só não é permitido alterar os endereços que contém o programa em simulação, já que a alteração de programas durante a execução não é uma prática aconselhável. Um exemplo desta janela encontra-se mostrado na figura 10:

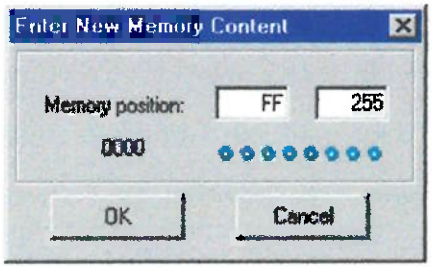


Figura 10 – Janela de edição do conteúdo de memória

4. Janela de IO:

Da mesma forma que a memória, existe a janela que é responsável pela visualização e edição do conteúdo de memória, a janela de IO é responsável pela monitoração e manipulação do conteúdo dos endereços de IO. Por se tratar de um lista bem menor de endereços (256 itens) , a listagem de IO pode ser exibida de forma integral, não ocorrendo o problema que existe com a listagem de endereços de memória. Entretanto, existe uma janela de visualização de I/O que funciona de forma semelhante a da memória. O usuário pode optar entre utilizar este recurso ou utilizar simplesmente a barra de scroll da janela. Para se editar o conteúdo de um endereço em particular, procede-se da mesma forma em relação à memória. Basta clicar-se em cima do endereço e uma janela de edição idêntica é aberta permitindo ao usuário entrar novos valores. A figura 11 apresenta a janela de endereços de IO, para os endereços iniciais com todos os conteúdos iguais a 00h:

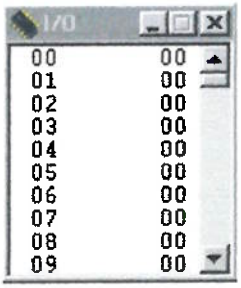


Figura 11 - Janela de IO

5. Janela de Interrupções:

Como foi dito anteriormente, esta janela não é mostrada ao se abrir um arquivo. Para exibi-la o usuário deve utilizar o comando Window|Interrupts. Esta janela é responsável pela simulação de interrupções e também pela entrada e saída serial do 8085. A figura 12 apresenta o conteúdo desta janela:

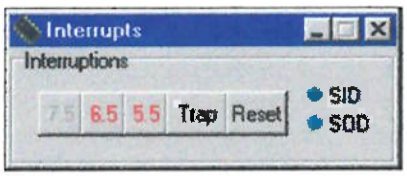


Figura 12 - Janela de Interrupções

A entrada serial e a saída serial são representadas por leds, da mesma forma como se fosses bits do acumulador. Para modificar a entrada, basta clicar em cima da mesma. A saída serial não é alterável diretamente, mas somente através da simulação de uma instrução SIM.

Os três primeiros botões representam as interrupções RST 7.5, RST 6.5 e RST 5.5. Essas interrupções são mascaráveis e afetadas pelo flip-flop de habilitação de interrupções. Assim sendo, para representar qual o estado em que as mesmas se encontram adotou-se um código de cores com o seguinte significado:

- Vermelho: Interrupção mascarada;
- Cinza: Interrupção desmascarada e não habilitada;
- Preto: Interrupção desmascarada e habilitada.

O próximo botão corresponde à interrupção TRAP. Este botão está sempre habilitado, uma vez que essa interrupção não é afetada por máscaras nem pelo flip-flop de habilitação.

Ao se clicar em uma interrupção habilitada, o endereço atual contido no par PC é disponibilizado na pilha, PC recebe um novo endereço que varia conforme a

interrupção acionada e a linha em processamento será atualizada para este novo endereço.

O quinto botão não representa uma interrupção, mas sim uma comando de RESET IN do 8085. Ao ser acionado, PC será alterado para o endereço 0000h e algumas variáveis do simulador serão inicializadas.

6. Janela de propriedades da simulação:

O usuário pode definir algumas das propriedades de simulação. A figura 13 mostra a janela exibida ao ser utilizado o comando Simulation|Properties:

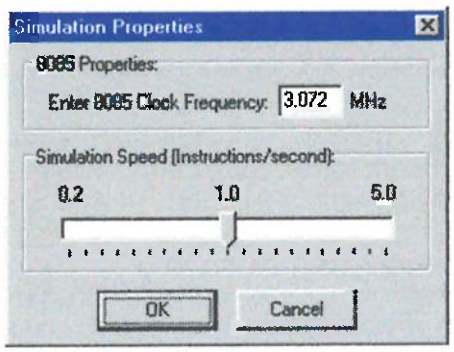


Figura 13 - Janela de Propriedades

Ao acessar a janela de propriedades, o usuário pode entrar qual o clock do seu 8085. Esse valor de clock é utilizado com a finalidade de se executar cálculos de tempo de simulação, conforme será visto adiante.

Outro procedimento que é possível de ser executado é o de se selecionar a velocidade de animação do modo Animate. A velocidade pode ser definida entre 0,2 a 5 instruções por segundo. Para alterar a velocidade basta que a trackbar seja posicionada na velocidade escolhida.

7. Janela de dados da simulação:

Uma ferramenta auxiliar que o simulador possui é a de fornecer alguns dados relevantes à simulação. A figura 14 mostra a janela responsável pela exibição desses dados:

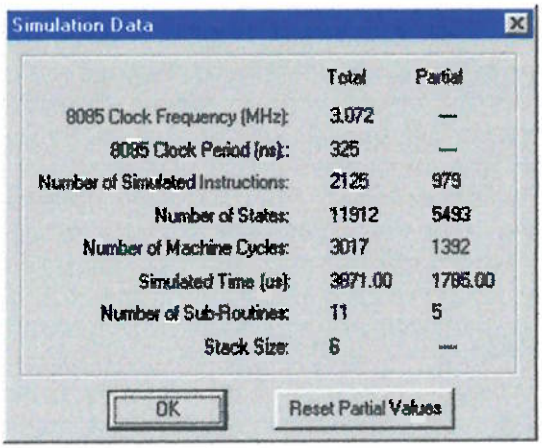


Figura 14 - Janela de Dados de Simulação

O primeiro dado é a freqüência do clock que foi fornecido na janela de propriedades. Caso não seja editado esse valor, assume-se um default de 3,072 MHz. A segunda informação é o período do clock do 8085. A seguir são relacionados o número de instruções simuladas, o número de estados simulados, o número de ciclos de máquina executados, o tempo de simulação, o número de sub-rotinas chamadas e o tamanho da pilha.

Os dados são divididos em duas colunas. A primeira se refere a valores totais, ou seja, desde a primeira instrução executada. A segunda se refere a valores parciais que podem ser anulados a qualquer instante da simulação. Utilizando valores parciais, pode-se obter valores que se referem a um intervalo específico do programa. Tal ferramenta é útil principalmente para se obter o delay de tempo na execução de um trecho do programa, o que é bastante comum ao se programar microprocessadores.

8. Janela de impressão:

Esta janela é responsável por mostrar quais itens podem ser impressos, deixando o usuário escolher um conjunto entre os mesmos:

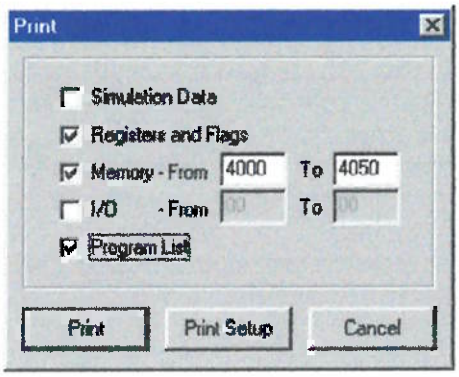


Figura 15 - Janela de Dados de Simulação

Pode-se imprimir os dados de simulação, o estados dos registradores e das flags, um intervalo de memória, um intervalo de I/O e também a listagem do programa LST. Se for escolhida a impressão de memória ou I/O, o usuário deve fornecer os limites dos intervalos nos campos indicados.

Formato do arquivo LST:

Para que a simulação seja possível, o programa a ser simulado deve estar escrito num padrão próprio denominado LST. Esse padrão tem o seguinte formato:

- Cada linha começa com quatro caracteres hexadecimais que representam a posição de memória em que se encontra a instrução.
- Do quinto ao vigésimo caracter, deve-se escrever os opcodes e os bytes auxiliares que venham a existir. É aconselhável que se deixe um espaço em branco entre o endereço de memória e o opcode.
- Do vigésimo caracter em diante, pode-se incluir qualquer texto que auxilie a simulação (tais como número da linha, instrução em assembler, comentários) já que a mesma é feita utilizando-se exclusivamente os opcodes.

Existem compiladores que já escrevem a partir de uma listagem em assembler, uma listagem LST neste formato. Neste caso, basta utilizar o arquivo que foi gerado. Pode-se também escrever o programa LST em um editor de textos (txt) comum.

Um exemplo de um programa escrito no formato LST pode ser visto abaixo:

```
2000  31 C2 20    01          LXI SP, 20C2H
2003  3E 08       02          MVI A, 08H
2005  30          03          SIM
2006  CD E7 02    04    LOOP: CALL RDKBD
2009  CD 6E 03    05          CALL UPDDT
200C  C3 06 20    06          JMP LOOP
```

Variáveis globais utilizadas:

Neste tópico relaciona-se a estrutura de dados globais utilizadas na implementação do simulador. O conhecimento dessas estruturas é de fundamental importância para o entendimento dos algoritmos adotados para as rotinas básicas de simulação.

Program: Lista ligada que armazena as posições de memória em que estão as instruções do programa LST. Program refere-se somente as posições dos opcodes e não dos operandos. Essa variável é alterada toda vez que um novo programa é aberto e é utilizada para se determinar o próximo endereço de memória que contém uma instrução válida.

Program2: Lista ligada que armazena as posições de memória em que está situado o programa a ser simulado (opcodes + operandos). Program2 é alterada toda vez que um novo programa é aberto e é utilizada para que não seja possível alterar os endereços de memória que contém o programa.

ValidProgram: Variável booleana que indica se o arquivo aberto possui pelo menos uma linha de acordo com o formato LST. Esta variável é utilizada para se habilitar ou desabilitar alguns dos comandos do simulador.

Continuous: Variável booleana que indica se o simulador está em modo de execução contínua (Run ou Step over).

Debugging: Variável booleana que indica a simulação atual já foi iniciada. Essa variável é utilizada para se definir qual ação será tomada em caso de um novo comando de simulação. Se a simulação não foi iniciada, apenas indica-se a primeira linha. Caso contrário, executa-se a instrução atual e indica-se a próxima linha.

Running: Variável booleana que indica se o simulador está em modo de animação (Animate).

StopCtn: Variável booleana que comanda o término da execução contínua ou animação. Quando os comandos Stop e Pause são acionados, a variável StopCtn recebe valor true. O mesmo ocorre se ocorre um aviso ao usuário durante a simulação.

Finish: Variável booleana que comanda o término da execução contínua com saída imediata. Existem situações em que é exigida a saída imediata das rotinas RUN ou STEP OVER.

Irq: Variável booleana que indica o estado do flip-flop de habilitação de interrupções.

Irq7: Variável booleana que indica o estado do mascara da interrupção 7.5.

Irq6: Variável booleana que indica o estado do mascara da interrupção 6.5.

Irq5: Variável booleana que indica o estado do mascara da interrupção 5.5.

Irq7pend: Variável booleana que indica pendência da interrupção 7.5.

Irq6pend: Variável booleana que indica pendência da interrupção 6.5.

Irq5pend: Variável booleana que indica pendência da interrupção 5.5.

IrqReq: Variável booleana que indica que a simulação está atendendo uma rotina de interrupção.

Sod: Variável booleana que indica o estado da saída serial.

Sid: Variável booleana que indica o estado da entrada serial.

Hidden: Variável inteira que indica o nível de sub-rotinas durante a simulação Step Over. O funcionamento da mesma segue o seguinte procedimento: No inicio de um comando Step Over, Hidden recebe zero. Para cada chamada de sub-rotina é acrescentada uma unidade e para cada retorno de sub-rotina é subtraída uma unidade. Quando Hidden for zero novamente, o procedimento de step over é finalizado.

PresentLine: Variável inteira que indica a linha em que está ocorrendo o processamento das instruções.

LastLine: Variável inteira que indica a última linha que foi indicada em azul na janela de programas. Nem sempre essa linha é a PresentLine, já que em simulações do tipo Run e Step Over não ocorre indicação da linha, mas ocorre o processamento das mesmas.

Instruction: Variável inteira que representa o valor da instrução da linha em processamento. É a partir dessa variável que se define quais procedimentos são tomados em um case com 256 opções.

Byte1: Variável inteira que representa o valor do primeiro operando em instruções que requerem tal termo.

Byte2: Variável inteira que representa o valor do segundo operando em instruções que requerem tal termo.

Period: Variável inteira que armazena o período do clock do 8085 em nanossegundos.

States[2]: Variável inteira que guarda o número de estados executados. Na posição 0 é armazenado o número de estados totais e na posição 1 o número de estados até a zeragem dos valores parciais. Desta forma, o número de estados parcial é dado por States[0]-States[1].

Cycles[2]: Variável inteira que armazena o número de ciclos de máquina simulados. O funcionamento é semelhante ao da variável States.

Executed[2]: Variável inteira que armazena o número de instruções simuladas. O funcionamento é semelhante ao da variável States.

Calls[2]: Variável inteira que armazena o número de sub-rotinas chamadas. O funcionamento é semelhante ao da variável States.

StackSize: Variável inteira que armazena o tamanho da pilha.

FirstLine: Variável inteira que guarda o índice da primeira linha que aparece na janela de programa.

DeltaLine: Variável inteira que guarda o número total de linhas que aparece na janela de programa. Essa variável depende do tamanho da tela do usuário e da definição escolhida. A mesma é calculada sempre que a janela de programa é aberta e a função principal é de se verificar se a linha de índice PresentLine está sendo mostrada na tela dado que a primeira linha é a de índice FirstLine. Se não é necessário realizar um scroll no texto.

Restart: Variável inteira que armazena a primeira posição de memória que o programa LST ocupa. Seu uso é o de substituir o valor de PC quando o usuário clica em Stop.

MemPosition: Variável inteira que guarda qual é o primeiro endereço de memória que se encontra disponível na janela, ou seja, o primeiro endereço da atual partição de memória. Utiliza-se essa variável por não ser possível exibir os 65535 endereços de memória no ListBox da janela de memória.

ScreenFactor; Variável real que é responsável por corrigir a resolução da tela quando o tamanho da fonte é diferente do padrão. Se o usuário está utilizando a configuração de fonte grande, as janelas dos registradores e das interrupções necessitam ser multiplicadas por este fator de escala para que sejam mostradas de forma correta.

Clock: Variável real que armazena o valor da freqüência do clock do 8085 em MHz.

Memoria: Lista ligada que armazena o conteúdo dos endereços de memória. Esta lista é implementada como uma classe e possui diversas funções implementadas de forma a facilitar a inserção, remoção e localização de endereços de memória. Como é inviável implementar-se uma lista ligada com 65535 adotou-se a seguinte solução:

Somente são inseridos elementos na lista que tenha valor diferente de 00h. Assim sendo, se é feita uma busca e o endereço não está presente na lista, retorna-se o valor 00h. Se o endereço está presente na lista, retorna-se o conteúdo que está associado ao mesmo. Tal solução é válida, pois dificilmente o simulador será utilizado com programas que ocupem muito espaço da memória.

IO: Vetor que armazena os conteúdos de entrada e saída do 8085. Esse vetor também é implementado através de uma classe própria que possui funções específicas. Como o número de endereços de I/O é bem menor que o número de endereços de memória, optou-se por utilizar um vetor ao invés de uma alocação dinâmica de elementos de lista ligada.

Algoritmos utilizados:

A seguir são descritos os algoritmos implementados para a solução dos procedimentos mais relevantes. Apenas os pontos mais importantes são relacionados, para se verificar os detalhes de implementação deve-se consultar a listagem fonte.

1. Step Into:

O Step Into tem dois tipos de procedimento. Se a simulação não começou ainda ou se a simulação já começou, mas o usuário alterou o conteúdo de PC, o primeiro Step Into deve apenas mostrar a próxima linha do programa sem executar nenhuma instrução. Se a simulação já começou e não houve nenhuma alteração em PC, o Step Into deve executar a instrução da linha que está indicada e indicar qual é a próxima linha. O algoritmo utilizado tem o seguinte padrão:

Se simulação não começou

Se PC está apontando para um endereço existente

Ache linha do programa correspondente a PC

Se PC não está apontando para um endereço existente

Se usuário modificou PC

Avise usuário sobre continuação no próximo endereço válido

Ache o próximo endereço existente

Ache linha do programa correspondente a PC

Defina simulação como já iniciada

Mostre linha pintada de azul na janela de programas

Retorne

Se simulação já começou

Se usuário modificou PC

Pinte linha atual em preto na janela de programas

Se PC não indica endereço é valido

Avise usuário sobre continuação no próximo endereço válido

Ache o próximo endereço existente

Ache linha do programa correspondente a PC

Mostre linha pintada de azul na janela de programas

Retorne

Execute instrução da linha

Pinte linha atual em preto na janela de programas

Atualize registradores e flags

Se PC não indica endereço é valido

Avise usuário que simulação continuará no próximo endereço válido

Ache o próximo endereço existente

Ache linha do programa correspondente a PC

Mostre linha pintada de azul na janela de programas

2. Step Over:

O Step Over é semelhante ao Step Into. Os mesmos cuidados são válidos: Só executa-se um instrução se a simulação já estiver iniciada e se PC não foi alterado. Caso contrário apenas a próxima linha é indicada. No Step Over deve-se verificar se

alguma instrução do tipo CALL foi executada. Em caso positivo, inicia-se a execução das instruções internas a sub-rotina de forma escondida até que o respectivo RET seja encontrado. O algoritmo adotado é o seguinte:

Zera nível de sub-rotinas
Se simulação não começou
    Se PC está apontando para um endereço existente
        Ache linha do programa correspondente a PC
    Se PC não está apontando para um endereço existente
        Se usuário modificou PC
            Avise usuário sobre continuação no próximo endereço válido
        Ache o próximo endereço existente
        Ache linha do programa correspondente a PC
    Defina simulação como já iniciada
    Mostre linha pintada de azul na janela de programas
    Retorne
Se simulação já começou
    Se usuário modificou PC
        Pinte linha atual em preto na janela de programas
        Se PC não indica endereço é valido
            Avise usuário sobre continuação no próximo endereço válido
            Ache o próximo endereço existente
        Ache linha do programa correspondente a PC
        Mostre linha pintada de azul na janela de programas
        Retorne
    Execute instrução da linha
    Se instrução é uma chamada de sub-rotina
        Incremente o nível de sub-rotinas
    Enquanto nível de sub-rotinas for maior que zero
        Se PC não indica endereço é valido
            Avise usuário sobre continuação no próximo endereço válido
            Termine execução de sub-rotina
        Ache linha do programa correspondente a PC
        Se a linha não é breakpoint

Execute instrução da linha
Se instrução é uma chamada de sub-rotina
Incremente o nível de sub-rotinas
Se instrução é um retorno de sub-rotina
Decremente o nível de sub-rotinas
Se a linha é breakpoint
Nível de sub-rotinas é zerado
Pinte linha atual em preto na janela de programas
Atualize registradores e flags
Se PC não indica endereço é valido
Avise usuário sobre continuação no próximo endereço válido
Ache o próximo endereço existente
Ache linha do programa correspondente a PC
Mostre linha pintada de azul na janela de programas

3. Animate:

O que a rotina de animação faz é habilitar um timer que toda vez que é zerado chama a rotina Step Into. O único ponto importante é que deve-se desabilitar o timer caso a linha seja um breakpoint. O algoritmo implementado é o seguinte:

Defina animação como iniciada
Chame a rotina de Step Into
Se a primeira linha é um breakpoint
Defina animação como terminada
Se a primeira linha não é um breakpoint
Habilite timer

Além disso, na rotina de Step Into é acrescentado a seguinte trecho no final. Este trecho desabilita o timer se a linha atual é um breakpoint.

Se está em modo de animação
Se linha é breakpoint

Desabilite timer

Defina animação como terminada

4. Run:

A rotina Run deve executar as instruções de forma continua. Se a simulação não foi iniciada, mostra-se a primeira linha do programa e a partir disso começa a execução. Caso a simulação já tenha começado, parte-se diretamente para a rotina de execução. Ao final, registradores, flags, memória e I/O devem ser atualizados, e a linha azul deverá indicar qual será a próxima instrução a ser executada. O algoritmo tem a seguinte estrutura:

Se simulação não começou

Se PC está apontando para um endereço existente

Ache linha do programa correspondente a PC

Se PC não está apontando para um endereço existente

Se usuário modificou PC

Avise usuário sobre continuação no próximo endereço válido

Ache o próximo endereço existente

Ache linha do programa correspondente a PC

Defina simulação como já iniciada

Mostre linha pintada de azul na janela de programas

Se a primeira linha é um breakpoint

retorne

Se simulação já começou

Se usuário modificou PC

Pinte linha atual em preto na janela de programas

Se PC não indica endereço é valido

Avise usuário sobre continuação no próximo endereço válido

Ache o próximo endereço existente

Ache linha do programa correspondente a PC

Mostre linha pintada de azul na janela de programas

Defina modo de simulação como execução contínua

Defina valor da flag de saída como false
Defina valor da flag de saída imediata como false
Execute instrução da linha
Enquanto a flag de saída for false
Se PC está apontando para um endereço existente
Ache linha do programa correspondente a PC
Se PC não está apontando para um endereço existente
Avise usuário sobre continuação no próximo endereço válido
Ache o próximo endereço existente
Ache linha do programa correspondente a PC
Mude valor de flag de saída para true
Verifique se usuário acionou algum comando
Se comando exige saída imediata
Volte valor da flag de saída imediata para false
Retorne
Se a linha atual não é breakpoint e se a flag de saída for false
Execute instrução da linha
Caso a linha seja breakpoint ou a flag de saída seja true
Termine execução continua
Pinte de preto a última linha indicada em azul
Ache linha do programa correspondente a PC
Atualize registradores, flags, memória e I/O
Pinte linha atual de azul

5. Busca da próxima linha:

A rotina de busca à próxima linha é uma das mais utilizadas em toda a simulação. Para cada instrução simulada, um ciclo de busca posterior é necessário. O algoritmo adotado tem as seguintes características:

1. A rotina de busca recebe uma linha inicial em que deve iniciar o processo de busca. Essa linha normalmente é a linha atual. Tal medida visa economizar tempos de busca já que na maioria das situações a próxima instrução a ser simulada está localizada na linha posterior à linha atual.

2. A busca percorre da linha inicial até o fim do programa. Caso não seja encontrada a linha, a busca recomeça na primeira linha do programa e segue até a linha atual. Deve-se realizar a busca também em trechos anteriores à linha atual já que podem ocorrer instruções do tipo RST, CALL, JMP e interrupções que forcem o programa a continuar em um PC anterior ao PC atual.

Um detalhe importante da função de busca é que sempre se tem a certeza de que a rotina achará alguma linha, já que sempre que a mesma é chamada é feito uma checagem na variável Program que indica quais endereços existem no programa.

O algoritmo implementado tem a seguinte estrutura:

```
Cursor recebe índice da linha inicial de busca
Flag de saída recebe valor falso
Enquanto cursor for menor que o número total de linhas e flag de saída for falsa
    Separe linha apontada por cursor
    Separe os quatro primeiros caracteres
    Se os caracteres são hexadecimais
        Busque próximo byte
        Se achou byte válido
            Opcode = byte
            Busque próximo byte
            Se achou byte válido
                Byte auxiliar 1 = byte
                Busque próximo byte
                Se achou byte válido
                    Byte auxiliar 2 = byte
            Saia e retorne o índice da linha apontada por cursor
    Se cursor chegou na última linha
        Cursor recebe índice da primeira linha
        Flag de saída recebe valor verdadeiro
    Incremente cursor
```

6. Busca do próximo byte:

Na rotina de busca a próxima linha, há sempre trechos onde busca-se possíveis bytes válidos que indiquem opcodes ou operandos. A função que faz este papel foi implementada e seu algoritmo encontra-se descrito abaixo. A função recebe como valor a posição da linha em que deve iniciar a busca e retorna a posição que foi encontrado o byte ou se não existe byte válido. Por exemplo: se a função está procurando o opcode, passa-se a mesma a posição 5 da linha já que as quatro primeiras posições são ocupadas pelo endereço. Se foi achado, a próxima busca será na posição 5+Valor de retorno da primeira busca.

São examinados sempre cinco posições após a posição inicial. Se o byte não foi encontrado até a quinta posição, então retorna-se indicando que naquela posição não existe byte válido.

Cursor recebe posição inicial de verificação
Contador = 0
Enquanto contador for menor ou igual a cinco
    Separe o caracter de posição (Cursor + Contador)
    Se caracter é hexadecimal
        Separe os caracteres de posições (Cursor + Contador) e (Cursor + Contador +1)
        Se os caracteres são hexadecimais
            Retorne (Contador)
        Se não são hexadecimais
            Incremente Contador
    Se caracter não é hexadecimal
        Incremente Contador
Se chegou até aqui retorne indicando ausência de byte válido

Testes Realizados:

Foram implementados vários programas de testes que visavam verificar o funcionamento específico de várias situações que podem ocorrer durante o uso do simulador. A seguir serão listados os testes executados:

1. Teste das instruções:

Todas as instruções foram simuladas pelo menos uma vez. Foram escritos vários programas LST que continham todas as instruções. Foi tomado um cuidado especial com as instruções de desvios condicionais. A seguir são listados alguns deste programas.

1 – Teste das instruções 00h a 0Fh:

```
2000 00        loop:   nop
2001 013245            lxi b,4532h
2004 02                stax b
2005 03                inx b
2006 0610              mvi b,10h
2008 04                inr b
2009 05                dcr b
200A 3E55              mvi a,55h
200C 07                rlc
200D 07                rlc
200E 09                dad b
200F 0A                ldax b
2010 0B                dcx b
2011 0C                inr c
2012 0D                dcr c
2013 0EFF              mvi c,FFh
2015 0F                rrc
2016 C30020            jmp loa
```

2 – Teste das instruções 10h a 1Fh:

```
2000 00        loop:   nop
2001 113245            lxi d,4532h
2004 12                stax d
2005 13                inx d
2006 1610              mvi d,10h
2008 14                inr d
2009 15                dcr d
200A 3E55              mvi a,55h
```

```
200C 17              ral
200D 17              ral
200E 19              dad d
200F 1A              ldax d
2010 1B              dcx d
2011 1C              inr e
2012 1D              dcr e
2013 1EFF            mvi e,FFh
2015 1F              rar
2016 C30020          jmp loop
```

3 – Teste das instruções 20h a 2Fh:

```
2000 00       loop:  nop
2001 213245          lxi h,4532h
2004 223545          shld 4535
2007 23              inx h
2008 2610            mvi h,10h
200A 24              inr h
200B 25              dcr h
200C 3E55            mvi a,55h
200E 27              daa
200F 27              daa
2010 29              dad h
2011 2A4045          lhld 4540
2014 2B              dcx h
2015 2C              inr l
2016 2D              dcr l
2017 2EFF            mvi l,FFh
2019 2F              cma
201A C30020          jmp loop
```

3 – Teste das instruções 20h a 2Fh:

```
2000 00       loop:  nop
2001 213245          lxi h,4532h
2004 223545          shld 4535
2007 23              inx h
2008 2610            mvi h,10h
200A 24              inr h
200B 25              dcr h
200C 3E55            mvi a,55h
200E 27              daa
200F 27              daa
2010 29              dad h
2011 2A4045          lhld 4540
2014 2B              dcx h
2015 2C              inr l
2016 2D              dcr l
2017 2EFF            mvi l,FFh
2019 2F              cma
201A C30020          jmp loop
```

4 – Teste das instruções 30h a 3Fh:

```
2000 00       loop:  nop
2001 313245          lxi sp,4532h
2004 323545          sta 4535
2007 33              inx sp
2008 3610            mvi m,10h
200A 34              inr m
200B 35              dcr m
```

```
200C 3E55           mvi a,55h
200E 37             stc
200F 37             stc
2010 39             dad sp
2011 3A4045         lda 4540
2014 3B             dcx sp
2015 3C             inr a
2016 3D             dcr a
2017 3EFF           mvi a,FFh
2019 3F             cmc
201A C30020         jmp loop
```

Da instrução 40h até a instrução BFh, encontra-se o grupo formado pelas seguintes instruções: MOV, ADD, ADC, SUB, SBB, ANA, XRA, ORA, CMP. Estas instruções foram testadas, porém com menos rigor já que cada instrução se diferencia de outra apenas pelos registradores envolvidos. Por este motivo, testou-se 3 ou 4 instruções de cada tipo.

Da instrução C0h até a instrução FFh, predominam as instruções de desvio de execução. Foram testados separadamente JUMPs, CALLs e RETs. Os programas de teste estão listados abaixo:

5 – Teste das instruções de JUMP:

```
2000 00        main:  nop
2001 C20020           jnz main
2004 CA0020           jz  main
2007 D20020           jnc main
200A DA0020           jc  main
200D E20020           jpo main
2010 EA0020           jpe main
2013 F20020           jp  main
2016 FA0020           jm  main
2019 C30020           jmp main
```

6 – Teste das instruções de CALL:

```
2000 00        main:  nop
2001 C42020           cnz loop
2004 CC2020           cz  loop
2007 D42020           cnc loop
200A DC2020           cc  loop
200D E42020           cpo loop
2010 EC2020           cpe loop
2013 F42020           cp  loop
2016 FC2020           cm  loop
2019 CD2020           call loop
201C C30020           jmp main
2020 00        loop:  nop
2021 C9               ret
```

7 – Teste das instruções de RET:

```
2000 00        main:  nop
2001 CD1020           call loop
2004 C30020           jmp main
2010 00        loop:  nop
2011 C0               rnz
2012 C8               rz
2013 D0               rnc
2014 D8               rc
2015 E0               rpo
2016 E8               rpe
2017 F0               rp
2018 F8               rm
2019 C9               ret
```

Foram ainda simuladas as instruções relacionadas às interrupções. Foi verificado o funcionamento das máscaras, dos procedimentos de interrupção, das interrupções pendentes e a utilização da saída e da entrada serial. O programa utilizado encontra-se abaixo:

8 – Teste das instruções de interrupção:

```
0024 00           trap:  nop
0025 FB                  ei
0026 C9                  ret
003C 00           rst7.5:nop
003D FB                  ei
003E C9                  ret
2000 314020       main:  lxi sp,2040h
2003 3E0B                mvi a,0Bh
2005 30                  sim
2006 FB                  ei
2007 DB00                in 00h
2009 00           loop:  nop
200A 20                  rim
200B C30920              jmp loop
```

Durante os testes foram corrigidas algumas instruções (cerca de oito) que não tinham o funcionamento esperado. Após a correção, a simulação das instruções apresentaram respostas satisfatórias. É importante observar que apesar de serem testadas todas as instruções, não se pode garantir que as mesmas não apresentem erros em determinadas situações. Isto porque é impossível realizar testes que simulem todas as condições possíveis de acontecer.

2. Teste dos quatro métodos de simulação.

Outro ponto do simulador que foi extensamente verificado é se os quatro métodos de simulação (Run, Animate, Step Over, Step Into) apresentavam o funcionamento esperado. Programas mais elaborados foram utilizados para testes e para cada programa simulado testou-se sempre todos os quatro métodos de simulação. Estes testes foram os que geraram um maior número de correções no programa fonte do simulador.

3. Teste dos Warnings.

Foram testadas as situações que geram warnings, ou seja, fora do funcionamento normal do simulador. Essas situações são:

- Abertura de um programa fora do padrão LST;
- Abertura de um programa que possua dois conteúdos diferentes para o mesmo endereço de memória;
- PC indicar um endereço que não possua instrução do programa;
- Tentativa de alteração de um endereço de memória ocupado pelo programa;
- Entrada de dados inválidos: nem hexadecimais, nem decimais ou fora da faixa aceitável.

Todas estas situações foram testadas e o comportamento obtido foi o esperado.

4. Teste da simulação de sub-rotinas.

Um dos testes realizados relaciona-se a execução de sub-rotinas, especialmente em casos de simulação do tipo Step Over. Tentou-se simular algumas situações que podem acontecer durante a execução de um programa qualquer. Estas situações são as seguintes:

- Teste de uma única sub-rotina com todos os quatro modos de simulação
- Teste de simulação Step Over em sub-rotinas que não apresentem a instrução de retorno
- Teste de vários níveis de sub-rotinas, ou seja, sub-rotinas que chamem internamente outras sub-rotinas
- Teste de interrupções durante execução de subrotina.

Estes teste apresentaram resultados satisfatórios, apresentando warnings quando necessário e também permitindo a parada da simulação nos casos em que eram previstos loops infinitos.

5. Teste dos breakpoints.

O último quesito testado foi o funcionamento dos breakpoints. Deve-se assegurar que os mesmo interrompam a simulação em todas as situações possíveis. Simulou-se o uso de breakpoints nas seguintes situações:

- Dentro de subrotinas – Uso de Step Over;
- Com o modo de simulação Run;
- Com o modo de simulação Animate.

Em particular, no modo de simulação Animate foi corrigido um trecho de programa que causava a seguinte falha: A parada não ocorria se o breakpoint estivesse na primeira linha do programa. Tal falha devia-se ao fato de que a busca da primeira instrução é feito de forma separada das outras buscas (ver algoritmo). Corrigindo-se este ponto, obteve-se a performance esperada.

Observações Finais:

O software foi desenvolvido de acordo com o cronograma proposto e todas as características que eram previstas foram implementadas. Outras implementações, que não tinham sido previstas originalmente, foram realizadas. Um dos exemplo é a simulação das interrupções, último item a ser incorporado.

O simulador foi desenvolvido com o intuito de se constituir uma ferramenta didática aos alunos da disciplina PMC-492. Todavia, um objetivo secundário é o de se possibilitar a utilização a outros usuários. Pretende-se disponibilizá-lo a outras pessoas através da colocação do mesmo na Internet e esta é a razão básica do desenvolvimento do mesmo em inglês.

Conforme foi visto no item anterior, vários testes foram feitos para se assegurar o funcionamento correto da simulação. Entretanto, é de se esperar que ocorram situações imprevistas que levem a comportamentos acidentais do programas. Pode-se esperar, portanto que algumas correções e revisões sejam necessárias em um primeiro momento.

Bibliografia:

[1] Holdsworth, B. *Microprocessor Engineering*. Buttsworth & Co. Ltda, 1987

[2] Intel *MCS-80/85™ family user's manual*. Intel Corporation, 1980

[3] Okamoto Jr, J. e Motohashi, C.T. *Apostila de PMC 492 – Laboratório de microprocessadores*. Departamento de engenharia mecânica – Escola Politécnica – USP, 1995

Anexo – Help do Programa

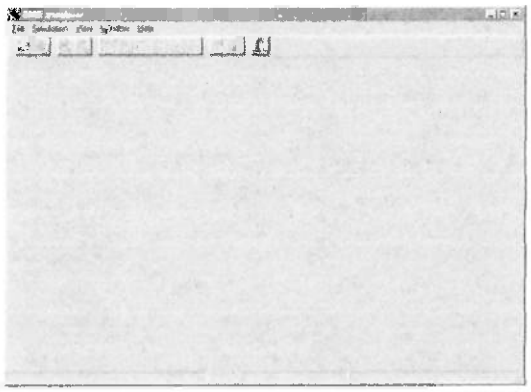
Overview

8085 simulator is a software created with the purpose of helping users to find errors in 8085 assembler programs and to verify if the program is executing what it was designed for. Using 8085 simulator, you can visualize all changes that happen in the registers, flags, memory and I/O when an instruction is executed.

There are four kinds of simulation: Run, Animate, Step Over and Step Into. Depending on what the user wants to verify, one kind of simulation is more appropriated than the other ones. Generally, Run and Animate should be used to visualize the execution of a block of instruction, while Step Over and Step Into are more suitable to inspect the execution of a single instruction each time.

Starting 8085 simulator

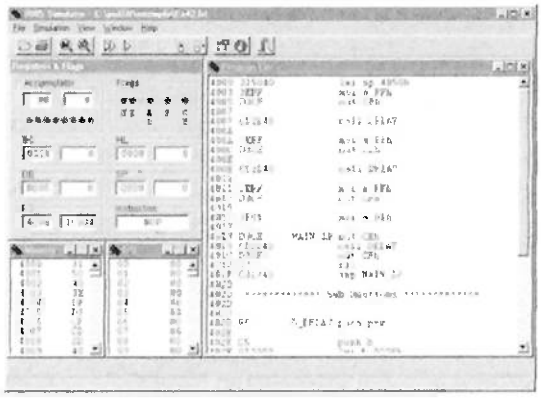
When you start 8085 simulator up, the following window is shown:



All Commands are disabled, except Open and Exit. This is one of the main characteristics of the software. You can only simulate programs that were previously compiled in an 8085 compiler or that were written in an external edit program. It is not possible to write the assembler program using 8085 simulator.

Presentation

After you have opened an 8085 LST program, the following window is shown:



Four new child windows are displayed:

- Registers and Flags;
- Program List;
- Memory;
- I/O;

Another child window that controls interrupts simulation is available. To open this window, click on Window|Interrupt.

Some of the commands of the toolbar and main menu are also enabled. You can start a simulation, change one of the simulation's properties or view pieces of information about the simulation.

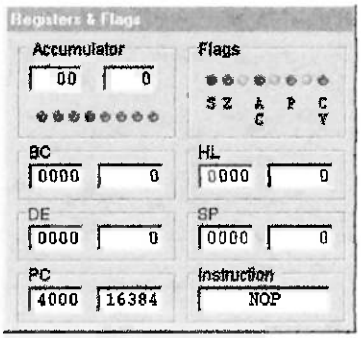
Registers & Flags Window

The Registers & Flags window enables you to:

- View the present values that are stored in the registers;
- View the present state of each flag;

- Change any of registers' content;
- Change the state of any flag;
- View the current instruction that PC is pointing.

The Registers & Flags window has the following appearance:



As it is shown, the window has eight fields. Each one responsible for one of the followings components:

- Accumulator;
- BC;
- DE;
- HL;
- SP;
- PC;
- Flags;
- Instruction.

Accumulator

In this field, you can view and edit the content of the accumulator. The accumulator is the most important register of an 8085 because all arithmetic, logical and I/O operations are made through it.

In the left edit, its content is displayed in hexadecimal format while in the right edit, its content is displayed in decimal format. You can also view or edit its content using the blue leds that represents each bit of the accumulator.

BC

In this field, you can view and edit the content of BC, which is a general-purpose register pair. In the left edit, its content is displayed in hexadecimal format while in the right edit, its content is displayed in decimal format.

DE

In this field, you can view and edit the content of DE, which is a general-purpose register pair. In the left edit, its content is displayed in hexadecimal format while in the right edit, its content is displayed in decimal format.

HL

In this field, you can view and edit the content of HL, which is a register pair used to point a specific memory position. In the left edit, its content is displayed in hexadecimal format while in the right edit, its content is displayed in decimal format.

SP

In this field, you can view and edit the content of SP, which is a register pair used to point the memory address that is currently the stack top. In the left edit, its content is displayed in hexadecimal format while in the right edit, its content is displayed in decimal format.

PC

In this field, you can view and edit the content of PC, which is a register pair used to point the memory address that has the current instruction in execution. In the left edit, its content is displayed in hexadecimal format while in the right edit, its content is displayed in decimal format.

Flags

In this field, you can view and change the status of each 8085 flag. If the correspondent blue led is on, than the flag is set. Otherwise, the flag is clear. You can change the status of a flag by clicking in its led.

Instruction

In this field, you can view the current instruction pointed by PC. You cannot change this instruction.

Program Window

In the program window, you can view the 8085 assembler program and visualize which instruction is being executed. Also, you can set or remove breakpoints in order to help your simulation.

The first important observation about the program window is that your 8085 program must be written in a specific LST Format. Otherwise, you will just be able to view the assembler program and not able to do any simulation. See more information about LST format on the next topic.

One element that you will have to be acquainted with is the execution point. The execution point indicates the next executable line in your source code that will be

executed when you use Simulation|Step Into or Simulation|Step Over. It is indicated by the blue line of code, as seen in the picture below.

If you use Simulation|Animate, then the execution point indicates which line is being executed. If you use Simulation|Run, then the execution point will not be refreshed every instruction execution. Instead, it will only be refreshed after the simulation encounters a breakpoint or you press Simulation|Pause or Simulation|Stop.

Breakpoints are indicated by red lines of code. To set a breakpoint, double click the desired line. To remove a breakpoint, do the same procedure. It toggles between breakpoint line and non-breakpoint line.

LST Format

Your 8085 program must use this LST format in order to do the simulation. Below, you can see the rules of this format. Some 8085 compilers already generate LST programs in this format, so that you do not have to worry about it.

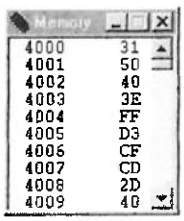
- Each line begins with four hexadecimal characters representing the memory address of the instruction.
- From the fifth character to the twentieth character, the opcodes and auxiliary bytes of instructions must be included. It is advisable to leave a blank space between the memory address and the opcode.
- From the twentieth character to the end of the line, you can include anything (instruction in assembler, comments, ...) since the simulation is made using exclusively the opcodes.

An example of this LST format is shown below:

```
============ mov test ============
2000 3ECD          loop:   mvi a,CDh
2002 47                    mov b,a
2003 3EAB                  mvi a,ABh
2005 4F                    mov c,a
2006 57                    mov d,a
2007 5F                    mov e,a
2008 58                    mov e,b
2009 53                    mov d,e
200A 3E20                  mvi a,20h
200C 67                    mov h,a
200D 3E30                  mvi a,30h
200F 6F                    mov l,a
2010 3EEF                  mvi a,EFh
2012 77                    mov m,a
2013 46                    mov b,m
2014 5E                    mov e,m
2015 C30020                jmp loop
```

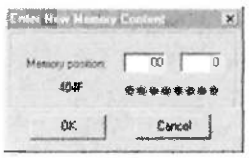
Memory Window

Use the memory window to view and edit memory contents. A list box containing memory addresses and its content composes the memory window.



As long as an 8085 memory can have 65535 addresses, it is impossible to show all addresses in the same list box. Therefore, the memory window always displays 512 addresses at the same time. You can decide which interval is being displayed using the menu command View|Memory Position or the correspondent button.

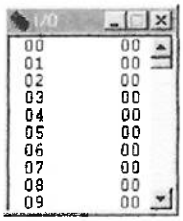
Each memory address can be modified when you double click it. After you double click a memory address, the following memory edit dialog is shown:



The procedure to change its value is similar to the accumulator's one. You can modify its content in hexadecimal format on the left edit or in decimal format on the right edit. You can also modify it using the blue leds that represents its bits.

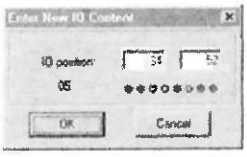
I/O Window

Use the I/O window to view and edit I/O contents. A ListBox containing I/O addresses and its content composes the I/O window.



An 8085 microprocessor can have 256 I/O addresses to the maximum. Therefore, it is possible to show all I/O addresses in the same list box. You can visualize any I/O address either using the scroll bar or using View|IO Position command. You can also use its correspondent button.

Each I/O address can be modified when you double click it. After you double click an I/O address, the following I/O edit dialog is shown:



The procedure to change its value is similar to the accumulator's one. You can modify its content in hexadecimal format on the left edit or in decimal format on the right edit. You can also modify it using the blue leds that represents its bits.

Interrupts

8085 simulator can also simulate 8085 interrupts. To open interrupts window, click on Window|Interrupt. The following window will be shown:

There are 4 buttons that represent interrupts and one button that symbolizes a RESET IN command. There are also two leds that represent the Serial Input Data (SID) and Serial Output Data (SOD) latches.

The interrupts are:

- Trap: non-maskable interrupt. Address of service interrupt routine: 24H
- RST 7.5: maskable interrupt. Address of service interrupt routine: 3CH
- RST 6.5: maskable interrupt. Address of service interrupt routine: 34H
- RST 5.5: maskable interrupt. Address of service interrupt routine: 2CH

Trap is always enabled since is non-maskable and is not affected by the interrupt-enabled flip-flop. RST 7.5, RST 6.5 and RST 5.5 are not always enabled. If their buttons are written in red letters, they are masked and will not work. To change the interrupt mask status, execute a SIM instruction. If their buttons are written in gray letters, they are unmasked, but disabled. To enable interrupts, execute EI instruction. If their buttons are written in black letters, interrupts are unmasked and enabled. Therefore, they are fully functional.

You can change the status of the Serial Input Data latch by clicking on its led. However, you cannot change the status of Serial Output Data latch directly. You have to execute a SIM instruction to change it.

Run

Choose Simulation|Run to execute instructions without showing the changes in registers, flags, memory and I/O. All changes will be update only after the simulation encounters a breakpoint or you press Simulation|Stop or Simulation|Pause.

Run is very helpful to execute large amount of instructions that you were previously tested and that precede a sequence of instructions you want to simulate. You just have to set a breakpoint in the beginning of the sequence and use Run. The execution point will indicate the desired line and the state of registers, flags, memory and I/O will be a result of all former instructions. Now, you can continue the simulation using either Simulation|Step Into or Simulation|Step Over to test the sequence of instructions.

Alternative ways to perform this command are:

- Choose the Run button on the toolbar.
- Press Shift+F5 keys

Animate

Use Simulation|Animate to start an animation of the execution of an 8085 program. You can define the speed of the animation in the Properties dialog. Instructions are performed one after another. Each execution is considered a step into command. Therefore, Animate will enter in every function in your 8085 program.

You can stop an animation using either Simulation|Stop or Simulation|Pause. A breakpoint will also pause an animation session.

Animate is useful to simulate 8085 programs as a whole, while Step Into and Step Over are more adequate to simulate instruction by instruction. Using animate all changes are updated after each instruction, consequently you can have a clear idea of the evolution of registers and flags states during the execution of the program.

Alternative ways to perform this command are:

- Choose the Animate button on the toolbar.

- Press F5 key

Step Into

Choose Simulation|Step Into to execute a program one line at a time, tracing into functions and following the execution of each line. The Step Into command executes the current program statement (written in blue) and advances the execution point to the next statement.

If you issue the Step Into command when the execution point is located on a function call, the debugger traces into the function, positioning the execution point on the function's first statement. In addition to tracing into procedures, you can step over them, executing each procedure as a single unit. Use Simulation|Step Over to execute procedures as a single unit.

By default, when you initiate a simulation with Simulation|Step Into, 8085 simulator moves the execution point to the first instruction pointed by PC.

Alternative ways to perform this command are:

- Choose the Step Into button on the toolbar.
- Press F7 key

Step Over

Choose Simulation|Step Over to execute a program one line at a time, stepping over functions while executing them as a single unit. The Step Over command executes the current program statement (written in blue) and advances the execution point to the next statement.

If you issue the Step Over command when the execution point is located on a function call, the debugger runs that function at full speed, then positions the execution point on the statement that follows the function call. In addition to stepping over procedures, you can also step into functions. Use Simulation|Step Into on a function call to execute program inside it.

If you issue Step Over when the execution point is located on a function call, and the function does not have a RET statement, simulation will start an infinite loop. You can finish this infinite loop pressing either Pause or Stop buttons.

By default, when you initiate a simulation with Simulation|Step Over, 8085 simulator moves the execution point to the first instruction pointed by PC.

Alternative ways to perform this command are:

- Choose the Step Over button on the toolbar.
- Press F8 key

Breakpoints

Breakpoints pause program execution during a debugging session at source code or address locations that you specify. You can set breakpoints before potential problem areas, then run your program at full speed. Your program pauses when it encounters a breakpoint, and the Program window displays the line or address location containing the breakpoint. You can then view the state of your program, or to step over or step into your code one line at a time.

When you simulate your program, it will stop whenever the simulator reaches the location in your program where the breakpoint is set, but before it executes the line or machine instruction. At this point, you can perform any other debugging actions.

If you set a breakpoint on a line in your source code, the line that contains the breakpoint will be displayed in the Program window in red characters.

You ought to set a breakpoint on an executable line of code or machine instruction. If you set breakpoints comment lines, blank lines, declarations, or other non-executable lines of code, a valid breakpoint will be created in the next executable lines of code.

You can set breakpoints before you begin simulating or while your program is simulating using the Program window. To set a breakpoint on a line of source code, double click the line in the Program Window.

When you no longer need to examine the code at a breakpoint location, you can delete the breakpoint from the simulation. You can delete breakpoints using the Program window. To delete a single breakpoint, double click the breakpoint line in the Program Window.

Halting Simulation

When you are using Simulation|Run, Simulation|Animate or Simulation|Step Over, you can halt the simulation using two different methods: Pause and Stop. You will have to halt a simulation, for example, when the 8085 program results in an infinite loop.

Pause – Use Simulation|Pause to cease the simulation. The indication point will indicate the next executable line in your source code that would be executed. From now on, you can continue the simulation using anyone of the four methods of simulation.

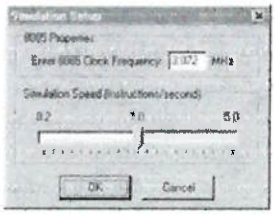
Stop – Use Simulation|Stop to cease the simulation and change PC to the first memory address of the 8085 program. Stop is very useful if you want to restart a simulation from its beginning. You can restart the simulation using anyone of the four methods of simulation.

Simulation Properties

Use Simulation|Properties to set any relevant property of the simulation.

You can set the 8085 clock frequency in order to calculate times involved in the simulation. One important observation is that if the simulation is running or the animation is happening, this field is disabled. You must pause or stop your simulation before proceeding.

You can also set the speed of the animation. This speed can be chosen from a range of 0.2 instructions per second to 5.0 instructions per second. To set a speed just move the trackbar to the desired position.



Simulation Data

Use Simulation|Data to view any relevant information of the simulation.

The data available are:

- 8085 clock frequency;
- 8085 clock period;
- number of simulated instructions;
- number of states;
- number of machine cycles;
- simulated time
- number of called sub-routines
- stack size

Some information topics have a partial field. This is helpful to obtain data exclusively of one particular section of the 8085 program. To reset partial values just press the correspondent button. Total values are considered from the beginning of the simulation.