**Tutorial do simulador P3**

Para a versão JavaScript online

https://p3js.goncalomb.com/

As considerações sobre o funcionamento do simulador do processador P3 que de seguida se apresentam, não pretendem ser exaustivas, tão pouco consistir nas melhores abordagens, e não estará livre de conter alguma incorreção.

O processador P3 constitui uma plataforma de aprendizagem da linguagem Assembly, elaborada pelo IST de Lisboa, que acompanha o livro “Arquitetura de Computadores – Dos Sistemas Digitais aos Microprocessadores”: http://algos.inesc-id.pt/arq-comp/

O P3JS é um assemblador e simulador não oficial para o processador P3 elaborado por Gonçalo Baltazar.

Por César Barros, nº1902295 – Dez 2021

O código que aqui se apresenta a título de exemplo, foi elaborado com base nas rotinas expostas na 4ª edição do livro, página 549, subordinado ao tema “Passagem de parâmetros pela pilha”, e tem como propósito o cálculo de um fatorial, fazendo uso de recursividade.

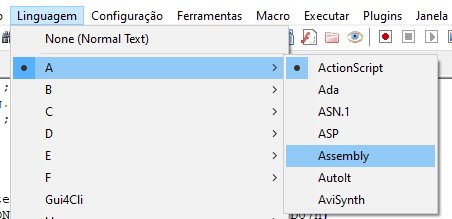
O código pode ser escrito diretamente no simulador, ou num editor de texto à escolha e depois copiado para o separador ASSEMBLER do simulador P3. O ficheiro com o código deve ser gravado com extensão ***.as***

**Programa exemplo no Notepad++**

Uma imagem com texto

Descrição gerada automaticamente

Deve ser selecionada a opção de Linguagem Assembly no Notepad++



**Exemplo do código no simulador** (copiado do processador de texto ou escrito diretamente no simulador)



Depois do código escrito ou copiado para a janela do separador “Assembler” do simulador, o código terá que ser assemblado, para o que se faz uso do botão do lado direito ASSEMBLE, assinalado a vermelho na imagem anterior.

Depois do código assemblado, podemos então ir ao separador “Simulator”, para correr o nosso código e verificar como se comporta o processador P3 durante a sua execução.

Aí encontramos por defeito numa área do lado esquerdo os registos, e respetivos valores, que vão mudando conforme a execução do nosso programa os afete.

São por defeito listados:

* os registos genéricos R0 a R7
* o ponteiro para pilha SP (stack pointer)
* o ponteiro para instrução do programa PC (program counter)
* as flags do registo de estado:
  + E (enable interrupts)
  + Z (zero)
  + C (carry – transporte)
  + N (negativo)
  + O (overflow – excesso)



Uma imagem com texto

Descrição gerada automaticamenteNo separador “Program”, na área “Labels/References” serão listadas as etiquetas e referências usadas no código, associadas aos endereços correspondentes.

Uma imagem com texto

Descrição gerada automaticamente

Na área central, em cima, temos a zona de memória, que poderá ser a de programa (Main Memory Zone - Vermelho), e em baixo a zona de memória destinada à pilha (Stack Memory Zone - Azul).

Acontece que tal só é como exposto, se essas áreas tiverem os intervalos de memória de interesse, ou não servirão o propósito.

Por exemplo, o programa exemplo começa com *ORIG 0000h*. Como temos por defeito mapeado no simulador essa zona no intervalo de endereços 8000:80ff, facilmente se percebe que o nosso programa começando no endereço 0000h, não vai ter as instruções do programa visíveis no simulador.

Já a pilha (stack) no programa exemplo, que vai fazer uso de um intervalo de endereços FDF0h:FDFFh, estará visível apenas se deslocar-mos o cursor da janela todo até abaixo, o que não é prático.

Recomenda-se por isso que, fazendo uso dos botões, ao centro em baixo, EDIT MAIN MEMORY ZONE e EDIT STACK MEMORY ZONE, se definam intervalos de interesse.

Para o programa exemplo apresentado, poderiam utilizar-se os seguintes intervalos de endereços:

MAIN MEMORY ZONE – 0000h:0027h

STACK MEMORY ZONE – fdf0h:fd07h

Nota: A pilha cresce de baixo para cima, pelo que, para lhe conceder mais espaço, o limite a alterar é o inferior.

Na coluna do lado direito (Disassemble zone – verde) aparecem as instruções, à frente dos endereços (sombreado a cinza) de cada uma delas.

**Considerações sobre o espaço de memória que ocupam diferentes tipos de instrução** (ver imagem seguinte)

Uma imagem com texto

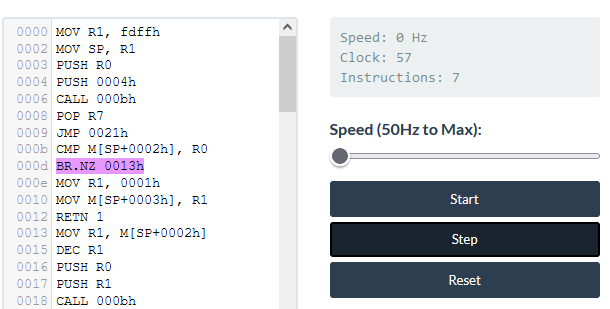
Descrição gerada automaticamente

Observando a 1ª linha, cuja instrução é *MOV R1,fdffh*, portanto, uma instrução de transferência de dados de memória para registo, esta começa no endereço 0000h

A 2ª linha com instrução *MOV SP,R1*, instrução igualmente de deslocamento de registo para registo, começa no endereço 0002h. Isto diz-nos que a instrução anterior ocupou de 0000h e 0001h, portanto 2 palavras de memória, e a instrução da 3ª linha a começar em 0003h, diz-nos que a instrução da 2ª apenas ocupou 1 palavra de memória. Isto diz-nos que o endereçamento imediato usado na 1ª instrução consome o dobro da memória em relação ao endereçamento por registo da 2ª instrução.

Análise similar se pode fazer com as duas instruções PUSH, onde a instrução *PUSH R0* só consome 1 word, e a instrução *PUSH 0004h* consome 2. Por isso se usa o PUSH R0 em vez de um *PUSH 0000h*, cujo resultado seria o mesmo, colocar o valor 0 na pilha, mas endereçado por registo temos uma instrução PUSH menos consumidora. Note-se que o registo R0, contém sempre o valor zero, e não pode ser manipulado de outra forma.

**Para executar a simulação do programa**, temos 2 formas de o fazer. Ou correr o programa todo de uma vez, usando o botão START, ou no modo passo-a-passo, com o botão STEP. Este último modo de execução deve ser o normalmente utilizado para nos permitir observar em detalhe o que ocorre após a execução de cada instrução do programa. O botão RESET, serve para repor a execução do programa no início, em qualquer dos dois modos.



No momento de recolha imagem anterior, pode observar-se no contador acima dos botões que foram executadas 7 instruções, que necessitaram de 57 ciclos de relógio para o efeito.

No entanto, pode ver-se na listagem do código que estamos na 9º linha. Significa isto que, teve que haver um salto entretanto, caso contrário não seria possível terem sido executadas 7 instruções quando estamos na 9ª.

Veja-se que a 5ª instrução faz um *CALL 000bh*, ou seja, salta para o endereço 000bh, ignorando 2 instruções pelo meio (a POP e a JMP)

**Mapa de memória**, na zona de código em cima ao centro (endereços a vermelho), com correspondência aos endereços associados á coluna do lado direito onde se listam as instruções assembly, e ao centro em baixo o mapa de memória (endereços a azul) destinada à pilha (stack)

Uma imagem com mesa

Descrição gerada automaticamente

Exemplos de correspondência entre o endereço de uma instrução e a localização no mapa de memória do programa

Uma imagem com mesa

Descrição gerada automaticamente

**Inserir um ponto de quebra**, clicando à frente do endereço de instrução de programa, coloca um ponto vermelho à frente desta, e ao fazer START o programa corre até chegar à instrução marcada.

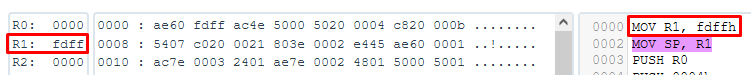
Uma imagem com mesa

Descrição gerada automaticamenteIsso permite-nos por exemplo prosseguir com a execução passo-a-passo do programa desse ponto em diante, sem ter que estar a corrê-lo todo dessa forma exaustiva, se o ponto de interesse se situa após dada instrução. Permite também saber quantos ciclos de relógio ocorreram e quantas instruções foram executadas até chegar a essa instrução marcada, observando o quadro à direita, assinalado a vermelho.

**Execução passo-a-passo do programa**:

1ª instrução – *MOV R1, fdffh*

Coloca em R1 o valor fdffh, como se pode verificar no simulador



2ª instrução – *MOV SP, R1*

Inicializa o ponteiro da pilha SP, com o valor de R1. No caso particular, definimos onde queremos que se situe o início da pilha no mapa de memória.

Uma imagem com mesa

Descrição gerada automaticamente

3ª instrução – *PUSH R0*

Esta instrução é traiçoeira para observar, pois não vamos ver alteração alguma no mapa de memória da pilha, dado que fomos colocar o valor 0 (R0) num endereço fdff (assinalado a azul) apontado pelo ponteiro de pilha SP, que já estava a zero. Mas pode-se observar que o valor de SP foi decrementado e passou de fdff para apontar agora o endereço fdfe (assinalado a vermelho). Note-se que a pilha “cresce” de baixo para cima.

Uma imagem com mesa

Descrição gerada automaticamente

4ª instrução – *PUSH \_N* ⬄*PUSH 0004h*

A azul, da direita para a esquerda, assinala-se a instrução que coloca na pilha o valor 0004h, no endereço de memória apontado na altura pelo ponteiro da pilha fdfe (ver mapa de memória da pilha ao centro). A execução desta instrução fez com que o ponteiro da pilha aponte agora o endereço seguinte fdfd, e como acontece com a execução sequencial de qualquer instrução, o ponteiro de programa PC, ficou a apontar o endereço da instrução seguinte.

Verificar também a evolução anterior de PC nas imagens de execução das 2ª e 3ª instruções. De resto, na lista das instruções do programa, o simulador assinala sempre com sombreado rosa a próxima instrução a executar, ou seja, a instrução que vamos executar após carregar no botão STEP, que está apontada pelo ponteiro de programa PC.

Uma imagem com mesa

Descrição gerada automaticamente

5ª instrução – *CALL fatorial* ⬄ *CALL 000bh*

Esta instrução vai produzir um salto para sub-rotina que se inicia no endereço de código 000b, e não só…

Uma imagem com mesa

Descrição gerada automaticamente

A instrução CALL, que produz o salto para uma sub-rotina, faz com que seja guardado na pilha o endereço da instrução que se seguia à que produziu o salto, para que possa retornar da execução da sub-rotina (RET, RETN) e continuar a executar o programa onde tinha sido interrompido.

Portanto, temos no endereço fdfd da pilha o valor 0008 que corresponde ao endereço da instrução que o programa executaria a seguir se não tivesse ocorrido uma instrução de salto. E em SP fica, como sempre, o endereço seguinte da pilha, no caso, fdfc.

PC, aponta agora para a 1ª instrução da sub-rotina, ou seja, para o endereço para onde o nosso código o mandou, isto é, 000b.

Uma imagem com mesa

Descrição gerada automaticamente

6ª instrução (a ser executada) – *CMP M[SP+0002h], R0*

É feita uma comparação entre R0 (ou seja 0h) e o que está em memória, duas words após o endereço apontado pelo ponteiro de pilha (SP+0002h).

Na figura indica-se para onde aponta SP (a vermelho), e para onde está localizado o dado apontado pelo resultado da operação de soma de SP com 2 unidades (a verde).

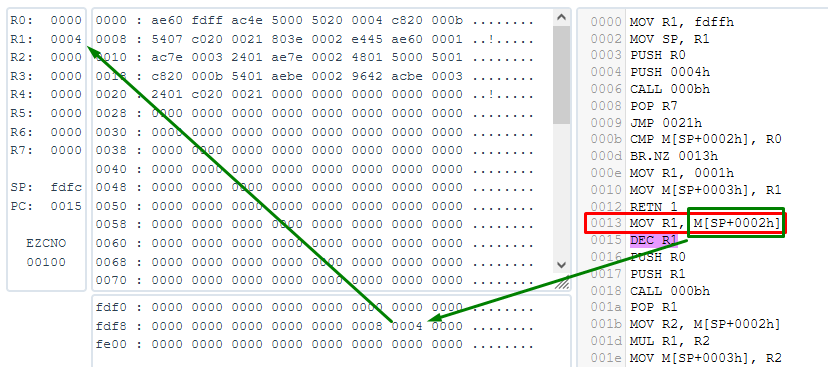
Portanto a instrução em apreço, compara se na zona de memória apontada por SP+2h, está o valor 0. O que lá está é o valor 4h, pelo que a instrução seguinte BR.NZ vai produzir um salto, uma vez que a condição não se vai verificar.

Uma imagem com mesa

Descrição gerada automaticamente

Poucas instruções mais tarde… *MOV R1, M[SP+0002h]*

Esta instrução vai mover para o registo R1 o conteúdo da célula de memória apontado por SP+2. Pelo exposto anteriormente, entende-se que agora se percebe com facilidade o resultado da operação da instrução



A instrução imediatamente a seguir, *DEC R1*, vai decrementar R1 ficando este com o valor 0003.

E as seguintes, *PUSH R0* e *PUSH R1*, vão colocar na pilha em células de memória consecutivas (fdfc e fdfb), os conteúdos de R0 e R1, respetivamente, correspondendo no código a passagem de parâmetros por pilha, do valor de N em R1, e de reserva de espaço em pilha para posteriormente receber o resultado da execução da sub-rotina, recorrendo a R0, que deixando o conteúdo da célula para já a zero, faz com que o ponteiro SP avance e deixe esse espaço disponível para o valor que há-de vir quando se der o retorno da sub-rotina. Depois de tudo isto, SP fica a apontar em resultado da operação, para fdfa.

Uma imagem com mesa

Descrição gerada automaticamente

Instrução – *CALL fatorial* ⬄ *CALL 000bh*

Na continuidade da execução do programa, temos a chamada à sub-rotina (CALL), que tem início no endereço 000bh (a vermelho). Esse salto faz com que seja salvaguardado na pilha o endereço da instrução que se seguiria, isto é, 001ª (a verde).

Uma imagem com mesa

Descrição gerada automaticamente

Umas dezenas de instruções depois, a pilha contém todos os valores que o presente programa gerou e nela guardou, sejam parâmetros de entrada para sub-rotina (a azul) ou espaço reservado para resultado (a verde), ou endereços de retorno (espaços não assinalados entre os anteriores).

Depois de inicialmente ter sido usado para definir o endereço de início da pilha, R1 passou conter o valor de N (N=4 cujo fatorial se pretende calcular) que ao longo da execução do programa foi sendo decrementado de 4 a 0, e é agora falso o resultado do teste BR.NZ pois em M[SP+0002h] onde tem vindo a ser guardado também na pilha “a cada passo” o valor de N, está agora o valor 0000h (retângulo a azul na área da pilha com endereço fdf2).

Uma imagem com texto, mesa

Descrição gerada automaticamente

Portanto a próxima instrução a executar é 000e: *MOV R1, 0001h*

Do ponto anterior em diante, vai sendo devolvido pela sub-rotina, o resultado do cálculo de cada iteração, colocado na pilha. Neste 1º passo da iteração N=1, com resultado 1, que é colocado na pilha no endereço SP+0003h, isto porque SP: fdf0, logo o resultado é colocado em fdf3, cujo espaço tinha vindo a ser reservado (via PUSH de R0)

Uma imagem com mesa

Descrição gerada automaticamente

Numa iteração intermédia…

A azul assinala-se o código da sub-rotina responsável pela recolha de resultados.

O valor contido na presente localização do ponteiro da pilha SP (seta a verde), é colocado em R1 (*POP R1*) como pode ser observado pelo assinalado a vermelho, com o resultado atualizado em R1

Uma imagem com mesa

Descrição gerada automaticamente

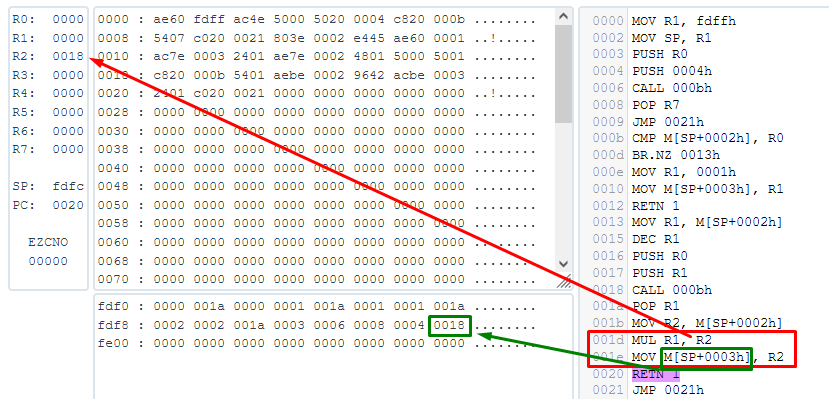
A iteração seguinte, *MOV R2, M[SP+0002h]*, vai buscar à pilha o valor atual de N que foi sendo sucessivamente armazenado na pilha pela sub-rotina anterior responsável para o efeito, e coloca-o em R2

Depois desta instrução e da anterior, temos em R1 o resultado das operações anteriores, e em R2 o valor atual de N. Estamos na fase em que o resultado já foi calculado para N= 1, 2 e 3, com resultado N!=1\*2\*3=6 (em R2), e estamos com N=4 (em R1)

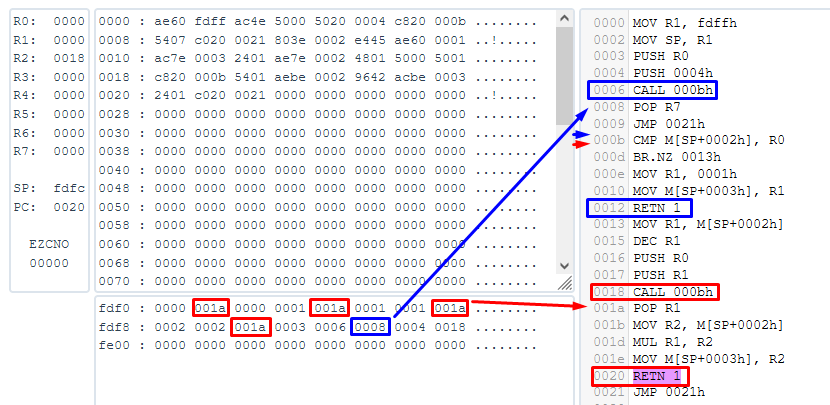
Uma imagem com mesa

Descrição gerada automaticamente

Nas 2 instruções seguintes, primeiro multiplica-se o resultado anterior com o novo valor de N (*MUL R1, R2*), ficando o resultado armazenado em R2, indicado pela seta vermelha. Este resultado, a ser devolvido pela pilha, é copiado de R2 para a posição de memória da pilha SP+0003h, ou seja, fdff (assinalado a verde)



Pelos endereços de instrução seguinte guardados na pilha, de cada vez que ocorre um salto devido a uma instrução CALL, pode perceber-se quantas ocorrências ouve de uma determinada instrução CALL. Como foi guardado uma vez na pilha o endereço de instrução seguinte 0008h, a chamada *CALL 000bh* decorrente da instrução com endereço 0006h, ocorreu 1 vez (assinalado a azul), já a instrução *CALL 000bh*, que por acaso até é idêntica, mas com endereço 0018h (assinalado a vermelho), ocorreu 4 vezes, pois encontramos 4 armazenamentos em pilha de instrução de valor seguinte 001ah. Note-se contudo, que esta observação só é válida por não ter havido alterações posteriores à pilha que tivessem alterado essas zonas de memória.



O resultado final da execução do programa pressupõe que o cálculo final seja apresentado no registo genérico R7, pelo que, esse resultado é passado da sub-rotina, pela pilha, no endereço fdffh, fazendo-se dele o POP para R7

Uma imagem com mesa

Descrição gerada automaticamente

A instrução seguinte, *JMP 0021h*, vai conduzir à finalização da execução do programa, colocando-o em loop infinito.

**Algumas variantes ao exposto:**

**1)** Se em vez de definir N como uma constante (EQU) se definisse como variável (WORD) e por conveniência fosse localizada numa zona de memória, distanciada da zona de memória destinada ao programa, o código poderia ser como se segue.

A partir do endereço 0060h, colocam-se as variáveis, e o programa inicia no 1º endereço de memória 0000h, tal como anteriormente.

Note-se que \_N não é o valor 0004h. \_N aponta agora para uma variável que está no endereço 0060h, por nós definido, que contém o valor 4. Por isso, onde antes se fez uso de \_N enquanto constante na instrução *PUSH \_N*, se o fizéssemos agora, o que ia para pilha não era o valor da variável (0004), mas o seu endereço (0060). Então a forma de colocar na pilha o valor pretendido é chamá-lo pela sua localização em memória *PUSH M[\_N]*

Uma imagem com texto

Descrição gerada automaticamente

O mapa de memória ficaria como se apresenta a seguir, onde a variável N tem o valor 0004h e está localiza em 0060h

Uma imagem com mesa

Descrição gerada automaticamente

Obs: coincidentemente a instrução tem o endereço 0004h, igual ao valor 0004h que a mesma coloca na pilha, apenas porque definimos N=4, mas não tem relação o endereço da instrução com o valor colocado na pilha.

**2)** Com a variável N agora a assumir o valor 5

Calculando: 5! = 120d = 78h e este resultado é apresentado corretamente em R7 (assinalado a vermelho)

Exercício: tentar encontrar no mapa de memória onde está o valor de N

Uma imagem com mesa

Descrição gerada automaticamente

Para o que se pretende neste exemplo chamar a atenção, é que se calhar na área de Stack não estão a ser visualizados todos os endereços de memória relevantes. Foram usados, e a prova disso é que o programa apresentou um resultado correto, mas não estão todos a ser visualizados.

Para resolver o problema, há que aumentar os limites de visualização de endereços da pilha, por exemplo de fdf0 para fde8, para se conseguir ver a informação em falta.

Uma imagem com mesa

Descrição gerada automaticamente

Recorde-se que a pilha cresce de endereços maiores para menores, pelo que é o limite inferior que deve ser alargado. Claro que com o aumento de valores de N, no nosso exemplo, a pilha cresce e novos limites teriam que ser redefinidos, apenas com o propósito de visualização dos endereços de interesse, não porque tal limitasse a execução do programa.

**3) Constantes e endereços** especificados no código, ficam na zona de memória que contém o código, como parte integrante de cada uma das instruções que delas faz uso. Recomenda-se que se veja no *“Manual do Simulador do Processador P3”* do IST, o anexo *A – Formatos das Instruções Assembly*

Assinalam-se alguns exemplos:

A vermelho, contantes como a que define onde começa a pilha (fdffh), ou o valor da contante com etiqueta “\_N” (0004h), ou o valor a atribuir ao registo R1 (0001h)

A azul, endereços de memória que no código estão associados a etiquetas como “fatorial” (000bh) ou “Fim” (0021h)

A verde, valores a associar a um registo para efetuar um endereçamento indexado como M[SP+W], onde W assume no código valores de 0002h ou 0003h

Uma imagem com mesa

Descrição gerada automaticamente

**Notas adicionais:** As variáveis devem ser remetidas para uma zona de memória afastada da zona que o código virá a ocupar, para que possam essas variáveis ocupar o espaço que lhes for destinado, sem interferir, quer com o espaço de código, com a pilha, ou com endereços reservados para portos, interrupções e temporizadores na zona mais alta da memória

**A memória** (64k endereços, com word de 16 bits)

|  |  |  |
| --- | --- | --- |
| 0000h | … | O código deverá começar em 0000h |
|  |  |
|  |  |
| 8000h | … | As variáveis podem vir para aqui (o importante é não terem endereços demasiado baixos que colidam com o código, ou demasiado elevados que possam vir a ser apanhados pelo crescimento da pilha) |
|  |  |
|  | … |  |
| FDFFh | A pilha pode começar em FDFFh e cresce de baixo para cima |
|  | … |  |
| FFF0h | … | Endereços reservados para I/O, temporizadores e interrupções |
|  |
| FFFFh |