| Palavra-chave | Tamanho | Significado |
| --- | --- | --- |
| BYTE PTR | 1 byte | unsigned char / char |
| WORD PTR | 2 bytes | short (16 bits) |
| DWORD PTR | 4 bytes | int / long (em 32 bits) |
| QWORD PTR | 8 bytes | long long / ponteiro em 64 bits |

1. \_somaVetor:

2. push ebp

3. mov ebp, esp

4. sub esp, 16

5. mov DWORD PTR [ebp-8], 0

6. mov DWORD PTR [ebp-4], 0

7. jmp L2

\_somaVetor:

RESUMO \_somaVetor:

Ambas variáveis locais  são armazenadas na na pilha da seguinte forma:  Na linha 2 salva o stack frame atual da pilha e na linha 3 define um novo stack frame que será utilizado pela função, fazendo o registrador de base( ebp) e de topo da pilha( esp) apontarem para o mesmo endereço de memória, a partir disso na linha  4 subtrai 16 bytes movendo o registrador de topo de pilha, alocando espaço para as variáveis locais.

A variável “acum = 0” corresponde a linha 5 onde estamos colocando zero em 4 bytes reservados para serem utilizados como a variável, a mesma ideia é utilizada na linha 6 onde é colocado 0 em 4 bytes a partir da posição ebp-4 que equivale a “j = 0;”  no main.c.

1. \_somaVetor = Rótulo que marca o incio da função int somaVetor(int vetor[], int elementos).

2 .push ebp = empilha o valor de ebp antes de ser alterado entao o esp recebe +4 bytes, logo os endereços anteriores a partir de ebp precisam de + 8 bytes de deslocamento

3. mov ebp, esp = criação do stack frame -> Stack frame = o "quadro da pilha" reservado para uma função.

esp = registrador do topo da pilha (Stack pointer).

ebp = registrador de base da pilha (Base pointer).

Faz ebp apontar para o valor atual de esp (topo da pilha nesse momento).

A partir daqui, todas as referências a variáveis e parâmetros vão ser feitas em relação a ebp.

É isso que te permite acessar os valores passados pelas linhas 44 45 e 46 do programa.

[ebp+8] → primeiro parâmetro (vetor)

[ebp+12] → segundo parâmetro (elementos)

[ebp-4], [ebp-8] etc. → variáveis locais

4. sub esp, 16 = O desloca 16 bytes o registrador de topo da pilha para reservar espaço para as variáveis

5. mov DWORD PTR [ebp-8], 0 = DWORD é abreviação para double word, uma palavra de 4 bytes que corresponde exatamente ao valor de um int em C(4 bytes ou 32 bits) sendo uma WORD uma palavra de 2 bytes, então [ebp-8],0 estamos “zerando” a variável local na posição [ebp-8] que corresponde ao acum=0 dentro da chamada da função \_somaVetor, posteriormente esse valor será passado e retornado pelo registrador EAX

6. mov DWORD PTR [ebp-4], 0 =

A mesma ideia anterior porém com deslocamento 4 bytes acima [ebp-4], corresponde a j=0 no main.c.

7. jmp L2 =

Realiza um salto incondicional para a linha com rótulo L2, que corresponde ao for (j = 0; j < elementos; j++).

8. L3:

9. mov eax, DWORD PTR [ebp-4] = eax recebe o valor de j novamente

10. lea edx, [0+eax\*4] = eax contém j logo equivale a eax\*4 que é o deslocamento em bytes de j em cada iteração do for, porque cada int tem 4 bytes

11. mov eax, DWORD PTR [ebp+8] = [ebp+8] contem o endereço de números[](o mesmo que [esp+20] )

12. add eax, edx = adiciona o deslocamento de j pra pegar o o endereço do elemento na posição eax+j ou números[j]

13. mov eax, DWORD PTR [eax] = move o valor naquela posicao

14. add DWORD PTR [ebp-8], eax = adiciona no acumulador

15. add DWORD PTR [ebp-4], 1 = incrementa j em 1

16. L2:

17. mov eax, DWORD PTR [ebp-4] = passa o j para eax

18. cmp eax, DWORD PTR [ebp+12] = compara com [ebp+12] que é ELEMENTOS passado como parametro

19. jl L3 = se for menor repete o loop de soma do vetor

20. mov eax, DWORD PTR [ebp-8] = eax recebe o total do acumulador

21. leave

22. ret

23. \_main:

24. push ebp

25. mov ebp, esp

26. and esp, -16

27. sub esp, 48

28. call \_\_\_main

29. mov DWORD PTR [esp.], 0 = zera o o topo da pilha

30. call \_time = Essa função time() retorna o tempo atual em segundos desde a época (1970). O retorno vem em EAX

31. mov DWORD PTR [esp], eax = move o retorno da \_time para o topo da pilha anteriormente zerado

32. call \_srand = instrução call \_srand desvia a execução para a função srand, passando como argumento o valor retornado por time(), que está armazenado em [esp].

VARIAVEL LOCAL i

33. mov DWORD PTR [esp+44], 0 = utiliza DWORD PTR pra colocar 0 nos bytes a partir do endereço [esp+44], usado para referenciar a variável local i,

34. jmp L6 = salto incondicional para L6 que corresponde ao for em main.c

35. L7: = marca o inicio do for no main.c

36. call \_rand = retorna um numero pseudoaleatorio 4 bytes em eax

37. mov edx, eax = guarda o valor aleatorio

38. mov eax, DWORD PTR [esp+44] = carrega em eax o valor de i

VETOR NUMEROS[i] = rand();

39. mov DWORD PTR [esp+20+eax\*4], edx = aqui ocorre a atribuição dos valores no vetor como em números[i], sendo esp+20 -> posição do vetor, e eax\*4 sendo eax o valor atual de i e movendo o valor aleatório em edx para a posição no endereço do vetor,

40. add DWORD PTR [esp+44], 1 = incrementa o i

41. L6:

42. cmp DWORD PTR [esp+44], 4 = [esp+44] é a variável i na pilha portanto compara com 4 como se fosse [valor] <= 4 que é a quantidade de iterações definido pela variável do tamanho do vetor em main.c *#define TAM\_VETOR 5*  que é o numero de elementos

43. jle L7 = jump if LESS OR EQUAL

44. mov DWORD PTR [esp+4], 5 = empilha o numero de ELEMENTOS logo antes da chamada de \_somaVetor, por convenção em arquitetura x86 os parâmetros são passados via pilha da seguinte forma

[esp] → último argumento (posição do primeiro parâmetro!)

[esp+4] → segundo argumento

[esp+8] → retorno anterior (caso exista mais argumentos)

…

45. lea eax, [esp+20] = [esp+20] = LEA (load effective adress) carrega o endereço calculado a partir de esp+20 que corresponde ao vetor números[] para o registrador eax

46. mov DWORD PTR [esp], eax = coloca o endereço de números[] como primeiro parâmetro da pilha

47. call \_somaVetor = chama a função soma vetor (empilha +4 bytes do endereço de retorno no esp)

48. mov DWORD PTR [esp+40], eax = a variável soma é DWORD PTR [esp+40] pega o valor retornado pela \_somaVetor

49. mov eax, DWORD PTR [esp+40]

50. mov DWORD PTR [esp+4], eax

51. mov DWORD PTR [esp], OFFSET FLAT:LC0

52. call \_printf

53. mov eax, 0

54. leave

55. ret