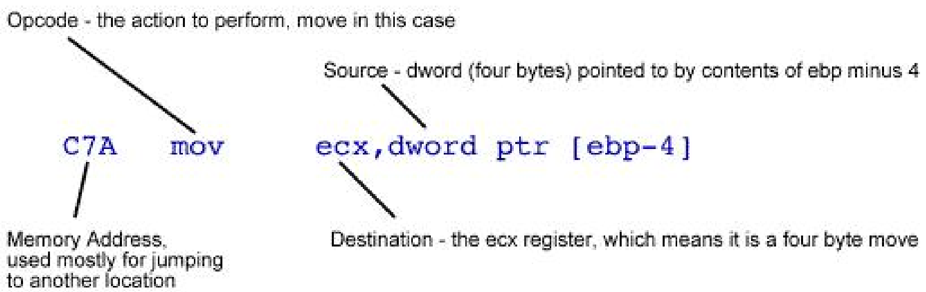
Assembly Review

In case you haven't looked at assembly before, and to orient you with the basics of this assembler's syntax, the following graphic shows one instruction in annotated detail. The element to the left, the memory address, would normally be eight hex digits for a 32 bit processor, but I've shortened it for readability. The Opcode mnemonics are pretty straightforward; mov= move, jmp= jump. The ones you might need hints on are lea = load effective address (compute, but don't dereference the address), j? = conditional jump (je = jump equals, jg = jump greater, etc.), and cmp = compare. It is also very important to note that the destination is the first operand, on the left, and the source is the second, on the right -- things don't make much sense if you get that wrong. For more detail on the Intel instruction set, see Reference for Intel x86 Assembly.



LabVIEW and C Generated Code
The output of the disassembler is on the left. The right column contains annotations that will hopefully make the code more understandable. The blue stripe to the left of the code highlights the loop body, where most of the execution will take place. I've trimmed the addresses to three digits to make it a bit easier to read.

LabVIEW	Annotations
1DD mov dword ptr [ebp+2D8h],0	zero out the loop counter
1E7 mov esi,dword ptr [ebp+2E0h]	
1ED mov eax,esi	
1EF cmp eax,0	
1F4 je 1FE	check for empty array (encoded by NULL pointer)
1FA mov esi,dword ptr [esi]	and determine how many times the loop will iterate
1FC mov eax,dword ptr [esi]	
1FE mov ecx,eax	
200 mov dword ptr [ebp+2DCh],ecx	
206 mov esi,dword ptr [ebp+2E0h]	
20C lea edi,[ebp+2ECh]	Create temporary
212 cmp esi,0	pointer,
215 je 235	stride,
21B mov esi,dword ptr [esi]	count
21D push eax	block for each autoindexing input array
21E mov eax,dword ptr [esi]	and initialize it
220 mov dword ptr [edi+4],eax	
223 pop eax	
224 mov dword ptr [edi+8],4	
22B add esi,4	
22E mov dword ptr [edi],esi	
230 jmp 249	
235 mov dword ptr [edi],4	

23B mov dword ptr [edi+4],0	
242 mov dword ptr [edi+8],4	
249 lea esi,[ebp+2ECh]	
24F mov eax,dword ptr [esi]	Adjust pointers for initial increment,
251 sub eax,dword ptr [esi+8]	and check for empty array and possibly skip loop
254 mov dword ptr [esi],eax	
256 lea edi,[ebp+2D4h]	
25C mov dword ptr [edi],eax	
25E cmp dword ptr [ebp+2DCh],0	
265 jg 270	
26B jmp 2DD	
270 lea esi,[ebp+2ECh]	
276 mov eax,dword ptr [esi]	Advance pointers used to access array elements
278 add eax,dword ptr [esi+8]	
27B mov dword ptr [esi],eax	
27D lea edi,[ebp+2D4h]	
283 mov dword ptr [edi],eax	
285 mov eax,dword ptr [ebp+2D4h]	
28B mov eax,dword ptr [eax]	check if array element is zero
28D cmp eax,0	
292 je 29D	
298 jmp 2AE	
29D mov eax,dword ptr [ebp+2D0h]	
2A3 add eax,1	Add one to value in shift register
2A8 mov dword ptr [ebp+2D0h],eax	
2AE mov eax,dword ptr [ebp+68h]	Check timer
2B1 cmp dword ptr [eax+18h],0	process UI events
2B5 je 3AC	(for Abort button, window moving, debugging, etc.)
2BB mov eax,dword ptr [ebp+2DCh]	
2C1 mov ecx,dword ptr [ebp+2D8h]	
2C7 add ecx,1	Increment loop counter
2CA cmp ecx,eax	and test for last iteration
2CC jge 2DD	
2D2 mov dword ptr [ebp+2D8h],ecx	
2D8 jmp 270	*** END of LOOP

C	Annotations
C7A mov ecx,dword ptr [ebp-4]	store number of array elements into local
C7D mov edx,dword ptr [ecx]	
C7F add edx,4	
C82 mov dword ptr [ebp-8],edx	
C85 cmp dword ptr [ebp-4],0	if(arrayBlock && (arraySize= **(int32**)arrayBlock))
C89 je ccc	
C8B mov eax,dword ptr [ebp-4]	
C8E mov ecx,dword ptr [eax]	
C90 mov edx,dword ptr [ecx]	
C92 mov dword ptr [ebp-14h],edx	
C95 cmp dword ptr [ebp-14h],0	
C99 je cc	
C9B mov dword ptr [ebp-10h],0	zero out loop count
CA2 jmp cad	ptetest loop before beginning
CA4 mov eax,dword ptr [ebp-10h]	for(i= 0; i < arraySize; i++) *** Beginning of LOOP
CA7 add eax,1	
CAA mov dword ptr [ebp-10h],eax	
CAD mov ecx,dword ptr [ebp-10h]	
CB0 cmp ecx,dword ptr [ebp-14h]	
CB3 jge ccc	
CB5 mov edx,dword ptr [ebp-10h]	if(array[i] == 0)
CB8 mov eax,dword ptr [ebp-8]	
CBB cmp dword ptr [eax+edx*4],0	
CBF jne cca	
CC1 mov ecx,dword ptr [ebp-0Ch]	count ++;
CC4 add ecx,1	
CC7 mov dword ptr [ebp-0Ch],ecx	
CCA jmp ca4	*** END of LOOP

```
CCC  mov    edx,dword ptr [ebp-0Ch]          printf("Located %ld zeroes in array", count);
CCF  push   edx
CD0  push   offset string db8                push pointer to format string
CD5  push   0
CD7  push   offset rcsid b74
CDC  movsx  eax,word ptr bcc
CE3  add    eax,0Dh
CE6  push   eax
CE7  push   offset _fileName_ (01f5eb50)
CEC  lea    ecx,[ebp-34h]
CEF  call   @ILT+106745
```

Figure 5. Disassembled Code from LabVIEW and C Compilers

Summary

I know that was a lot of information, but that is why higher level languages were invented -- to encode the operations in a form more suitable to humans than CPUs. For those that are familiar with Intel assembly, you may notice that there are several areas where the LabVIEW generated code could be improved, and rest assured, work is already underway to improve register usage. Even at the current state, the body of the loop is just under twice the size of the C code, and that is roughly what is expected.

The LabVIEW environment trades some efficiency of the computer for development power of the end user. As an example, the loop above has code in it that tests a timer and periodically checks with the UI to let you abort the VI or move and interact with the window in the LabVIEW environment. Similarly, the LabVIEW array is a dynamically sized array. This means that before entering the loop certain things need to be verified. Is there an array? Is it empty? If there are multiple input arrays and an input to the N of the loop, which is smallest? These considerations add a bit to the execution of the VI, but they don't actually affect the inside of the loop, so in the larger scope as the loop count increases, these conveniences don't cost much. For reference, the typical instruction above will execute in about one or two nanoseconds assuming it is in the cache.

Details

The LabVIEW VI above was compiled as a subroutine, which removes the generated code for debugging. As a regular priority VI with debugging the code has an additional 37 instructions in the body of the for loop.

The C code was compiled in standard mode with symbols -- without optimization turned on. This was for ease in annotation and understanding. With optimization, the code of the loop body shrank to six instructions but was much more difficult to annotate because the optimizer rearranges the instructions.