Benton Guess
CSCE-451
Feb 12, 2020

Question 1

Looking a the context of the function, we can see that it has at least 3 arguments.

Question 2

These were all compiled with `gcc -m32 ...` with gcc 9.1

Addition

For all three of the addition operators, `a++`, `a = a + 1`, `a += 1`, they compile to the same instructions. However, amidst this we get a `nop` instruction which is unrelated to the question at hand, but this was interesting nonetheless.

C code

```c
void func() {
   int a = 0;
   int b = 0;
   int c = 0;

   a++;
   b += 1;
   c = c + 1;

   return;
}
```

x86

```
   push ebp
   mov, ebp, esp
   sub ebp, 16
   mov dword_ptr [ebp - 4], 0
   mov dword_ptr [ebp - 8], 0
   mov dword_ptr [ebp - 12], 0
   add dword_ptr [ebp - 4], 1
   add dword_ptr [ebp - 8], 1
   add dword_ptr [ebp - 12], 1
   nop
   leave
   ret
```

Conditionals

Trying both the traditional `if {...} else {...}` statement and the tertiary operator, we get the same compiled code. Keeping the code as simple as possible, we can see that both statements take slightly different approaches to a conditional. The normal statement never makes use of a register, writing all of its instructions directly to the stack. The tertiary operator, on the other hand, stores its results in the eax register, then moves the eax register into the stack variable after the statement is done. The normal statement requires less operations, but it is hard to tell which produces more efficient code since they are so similar.

C code

```c
int func(int num) {

   int a;
   int b;

   if (num == 1) {
      a = 10;
   } else {
      a = 100;
   }

   b = (num == 1) ? 10 : 100;

   return 0;
}
```

x86

```
   func:
   .LFB5:
         push   ebp
         mov    ebp, esp
         sub    esp, 16
         cmp    DWORD PTR [ebp+8], 1
         jne    .L2
         mov    DWORD PTR [ebp-4], 10
         jmp    .L3
   .L2:
         mov    DWORD PTR [ebp-4], 100
   .L3:
         cmp    DWORD PTR [ebp+8], 1
         jne    .L4
         mov    eax, 10
         jmp    .L5
   .L4:
         mov    eax, 100
   .L5:
         mov    DWORD PTR [ebp-8], eax
         mov    eax, 0
         leave
         ret
```

Loops

Trying all three types of loops, the for, while, and do-while, gave significantly different results in terms of control flow. These were tested by creating a simple loop that increments a stack variable. First off, the do-while loop appears to be the most efficient in terms of operations and jumps. Since the condition check and the jump occur at the same point in the code, they can be chained into one operation. This means that at most the do-while loop will only perform at most `num - 1` jumps. The for loop seems to be the next most efficient, where we have both adds occurring in sequence then a quick comparison. However, since the code must start with a condition check, the for loop must do `num` jumps. The while loop plays out the strangest, with the start occuring like the for loop. However, we do not see the counter get incremented in the body. This makes since, since this while loop has the counter incremented in the condition. However, we see some compiler trickery as it uses `lea edx, [1 + eax]` to increment our counter. Since the counter is incremented after the comparison, we get a slightly more complicated conditional comparison, however the overall control flow is identical to the for loop.

C code

```c
int func(int num) {
   int a = 0;
   int b = 0;
   int c = 0;

   for (int i = 0; i < num; i++) a++;

   int j = 0;
   while (j++ < num)
      b++;

   int k = 0;
   do {
      c++;
   } while (++k < num);

   return 0;
}
```

x86

```
func:
.LFB5:
      push   ebp
      mov    ebp, esp
      sub    esp, 32
      mov    DWORD PTR [ebp-4], 0
      mov    DWORD PTR [ebp-8], 0
      mov    DWORD PTR [ebp-12], 0
.LBB2:
      mov    DWORD PTR [ebp-16], 0
      jmp    .L2
.L3:
      add    DWORD PTR [ebp-4], 1
      add    DWORD PTR [ebp-16], 1
.L2:
      mov    eax, DWORD PTR [ebp-16]
```

```
        cmp     eax, DWORD PTR [ebp+8]
        jl      .L3
.LBE2:
        mov     DWORD PTR [ebp-20], 0
        jmp     .L4
.L5:
        add     DWORD PTR [ebp-8], 1
.L4:
        mov     eax, DWORD PTR [ebp-20]
        lea     edx, [eax+1]
        mov     DWORD PTR [ebp-20], edx
        cmp     DWORD PTR [ebp+8], eax
        jg      .L5
        mov     DWORD PTR [ebp-24], 0
.L6:
        add     DWORD PTR [ebp-12], 1
        add     DWORD PTR [ebp-24], 1
        mov     eax, DWORD PTR [ebp-24]
        cmp     eax, DWORD PTR [ebp+8]
        jl      .L6
        mov     eax, 0
        leave
        ret
```

Question 3

The value returned should be equal to the function's first argument minus its second argument.

Question 4

Function `_Z3sixiii` has 3 arguments, and function `_Z4fiveiii` has 3 arguments.

Question 5

First, the "six" function calls the "five" function passing its 3 arguments to the function in the order of 3, 1, 2. The "five" functions 3 arguments, a, b, and c, are put through the following equation. In terms of "five", "six" arguments 1, 2, 3 are b, c, and a respectively.

$$n_1 = (a - c) + b$$

$$n_2 = \begin{cases} 0 & n_1 \geq 0 \\ -1 & n_1 < 0 \end{cases}$$

$$n_3 = \left\lfloor \frac{(n_1 * 1431655766)}{2^{32}} \right\rfloor - n_2$$

else

n_3 is returned by the "five" function and the subsequently returned by the "six" function.

Question 6

Note this is done on 8 bit signed/unsigned integers.

Arg 1	Arg 2	CF	PF	AF	ZF	SF	OF
0x10	0x00	0	0	0	0	0	0
0x80	0x10	0	0	0	0	0	1
0x80	0x00	0	0	0	0	1	0
0x10	0x20	0	0	1	0	0	0
0x80	0x10	0	0	1	0	0	1
0x90	0x10	0	0	1	0	1	0
0x30	0x00	0	1	0	0	0	0
0x80	0x20	0	1	0	0	0	1
0x81	0x00	0	1	0	0	1	0
0x00	0x00	0	1	0	1	0	0
0x10	0x10	0	1	1	0	0	0
0x80	0x20	0	1	1	0	0	1
0x90	0x20	0	1	1	0	1	0
0x00	0x90	1	0	0	0	0	0
0x00	0x20	1	0	0	0	1	0
0x00	0x80	1	0	0	0	1	1
0x00	0x81	1	0	1	0	0	0
0x00	0x20	1	0	1	0	1	0
0x10	0x81	1	0	1	0	1	1
0x00	0xa0	1	1	0	0	0	0
0x00	0x10	1	1	0	0	1	0
0x10	0x80	1	1	0	0	1	1
0x00	0x82	1	1	1	0	0	0
0x00	0x10	1	1	1	0	1	0
0x10	0x82	1	1	1	0	1	1

Question 7

The code here is simply a static array being passed to a function, then summed based on the lower 16 bits of each array element.cThe final value of `EAX` will be 26862.

Question 8

This function just does a bunch of operations on two parameters. The final value of `EAX`, the return value, will be 9000.

Fibonacci Code for Graduate Students

Since I am in the honors section, I was not completely sure If I needed to do this one, so I did it just to be safe. The instructions to compile is as follows.

- `nasm -f elf32 -g -F dwarf fib.s -o fib.o`

- `g++ -m32 -o fib fib.o -lc`

The binary runs as a CLA with an option of 0 or 1 arguments. Supplying no arguments to the binary will simply compute the 13th Fibonacci number. If a number is supplied as the first command line argument, it will calculate based on that degree. Note that the result will only be calculated with 0 as the 0th Fibonacci number and 1 as the 1st, and from there on the formula is as it is normally calculated.

```
; nasm -f elf32 -g -F dwarf fib.s -o fib.o
; g++ -m32 -o fib fib.o -lc
global main

section .data
   result_text: db "The value of a %d degree fibonacci is %u",0x0a,0x0
   failed_text: db "Arguments: [fibonacci degeree (defaults to 13)]",0x0a,0x0

section .text
   extern printf
   extern atoi

; fib(degree : i32)
; returns the result into degree
fib:
   push ebp
   mov ebp, esp
   sub esp, 16
   push dword [ebp + 8]
   pop dword [esp + 12] ; target
   mov dword [esp + 8], 0 ; n-2 value
   mov dword [esp + 4], 1 ; n-1 value
   mov dword [esp], 1 ; counter


   ; ebp is useless from this point until return
   ; deal with 0 and 1
   mov ebp, [esp+12]
   cmp ebp, 0
   je fib_zero_cond
   cmp ebp, 1
   je fib_done

   ; deal with [2,47]
   fib_loop:
   mov ebp, [esp + 8]
   add ebp, [esp + 4]
   push ebp
   mov ebp, [esp + 8]
   mov [esp + 12], ebp
   pop ebp
   mov [esp + 4], ebp

   inc dword [esp]
   mov ebp, [esp]
   cmp ebp, [esp + 12]
```

```
    jl fib_loop
    jmp fib_done


    fib_zero_cond:
    mov dword [esp+4], 0

    fib_done:
    push dword [esp + 4]
    pop ebp
    add esp, 16
    mov [esp + 8], ebp
    pop ebp
    ret

; Registers are used in main for operate functions like printf and atoi, as
    well as return the proper code
; However, only ebp and esp are used in fib
main:
    mov ebp, esp
    sub esp, 4

    ; Ensures that we have the right number of arguments
    cmp dword [ebp + 4], 2
    jg failed_code
    cmp dword [ebp + 4], 1
    jne has_argument
    mov dword [ebp - 4], 13
    jmp run_fib


    has_argument:
    ; Parse command line args using atoi
    ; get type of op
    mov edx, [ebp + 8]
    mov eax, [edx + 4]
    push eax
    call atoi
    add esp, 4
    mov [ebp - 4], eax

    run_fib:
    push dword [ebp - 4]
    call fib
    mov ebx, [esp]
    add esp, 4

    push ebx
    push dword [ebp - 4]
    push result_text
    call printf
    add esp, 12
    jmp Main_Done
```

```
failed_code:
push failed_text
call printf
add esp, 8
mov eax, 1
ret



Main_Done:
add esp, 4
mov eax, 0
ret
```