CS341 Bob Wilson

MP2: i386 Assembly Language functions, ports, and timing

Due: At start of Class 12

The purpose of this assignment is gain some familiarity with writing C callable functions that do things that can't be done in C such as accessing I/O ports. Copy all files from ~bobw/cs341/mp2 to your mp2 subdirectory of cs341. Use the provided makefile for all builds except where instructed otherwise.

0.1 Follow the instructions in mp2warmup.doc to get oriented. In file tiny.script, leave the script of your run of these actions (steps 1-4, but all you need to do for .1 is "cat tiny.s" to show your file.)

0.2 Build sum10.lnx by "make A=sum10". Learn about remote gdb by making your own version of sum10.script following the steps in the provided sum10.script.

1. Write a C callable assembler function that counts the number of characters in a string. The function should work for any string and character. The address of the string and character to be counted are passed as arguments according to the C function prototype:

   ```
   int count (char *string, char c);
   ```

   Since there are arguments, your assembly function should use a stack frame. It should return the count to the calling C program in %eax. Use 32-bit quantities for all data. Chars have only 8 significant bits but they are stored in memory (and on the stack) as 32 bits in "little endian" format. After moving a 32 bit char value from memory to a register, the char value is available in the 8 lsb's, e.g. %al.

   You are given a C calling program that calls count to count the number of 'a's in "aabcabcabc" and prints the result. The C code is in countc.c. Put your assembly code in count.s.

   In count.script, provide a script showing a run with a Tutor breakpoint set where the count is incremented, showing the count (in a register) each time the breakpoint is hit, just before the increment is made. In the script, show how you determined where to set the breakpoint, i.e., how you determined where the increment instruction is located in memory.

Checkpoint: 0.1, 0.2, and 1. should be done by one week before the due date, leaving at least one full week for the more advanced work below!

2. You are given a C calling program (printbinc.c). Call your file printbin.s. The C caller for this program provides a char to the assembly function printbin. It asks the user for a hex number between 0 and 0xff, converts the input from an ASCII string to a char value, passes it to printbin as an argument, and displays the returned string which should be the ASCII characters for the binary value, e.g. for entry of the hex value 3d, you will get the printout: "The binary format for 3d is 0011 1101" You can see the function prototype for the printbin function in the calling C code.

   The function printbin should be C callable using a stack frame and call an assembly language subprogram "donibble" that's not required to be C callable. Avoiding use of stack frames is one way assembly code can be more efficient than C compiler generated code. The function printbin needs to declare space in the .data section for storage of a string to be returned and return the address of that location in the %eax. While processing the bits of the input argument, keep a pointer to the string in an available register. Printbin and donibble can store an ascii character 0x20, 0x30, or 0x31 in the string indirectly via that register and then increment the pointer in that register until the entire return string has been filled in.

The donibble function handles one half of the char value producing the four ASCII characters (0/1) for the bits in one hex digit. Printbin should call donibble twice once with each nibble to be processed in half of an available register, e.g. the %al. Donibble should scan the 4 bits of the register and move an appropriate ascii code into the string for each bit. Printbin adds the space between the two nibbles.

In file printbin.script, show a run of printbin on the SAPC to display 0xab, and then a run with a breakpoint set at the helper function, to prove that it is called exactly twice. You can use Tutor or remote gdb here as you wish.

3. Programming a UART directly in assembler. Please note that we consider this the old-fashioned way to do this programming! The current way is to use assembler only where it is required.

   Rewrite $pcex/echo.c as a C/assembler program as follows. The C driver does the same initial logic determining whether or not the console is a COM port and if so which one, and if not, just return. In the new version, it then goes on to ask the user for the "escape character" that stops the echo loop. This can be CONTROL-A but doesn't have to be. The user types the actual character, not its ASCII code or whatever, to specify the escape character.

   Then the C driver calls the assembler echo function:

   ```
   void echo(int comport, unsigned char esc_char)
   ```

   The assembler code does the loop with in and out instructions to do the actual I/O, testing for the special esc_char and returning when it is seen coming from the user. Call your files echo.s and echoc.c. In file echo.script, show a run with "abc" followed by the esc_char, a second run with "abcde<CR>xy" followed by the esc_char, and a third run with "abcde<CR><LF>xy", then esc_char. If your keyboard doesn't have a key marked linefeed, use <control-J>. Since we haven't asked for any special treatment for <CR>, the "xy" of the second test should overwrite the "ab", but the <LF> should prevent this in the third test.

4. Write a C-callable assembler function called pollcount.s that has entry point pollcount_putc which can be called from pollcountc.c as:

   ```
   int pollcount_putc(unsigned char ch)
   ```

   It takes a char, counts how many tests of COM2 THRE are done in the polling loop and then outputs it to COM2. It returns the count as the function return value. Note: Use symbolic constants to avoid hard coded numbers here!

   If you want, you can omit the stack frame from your assembly code. If you do this, you can't get the argument using the %ebp since it will be pointing to the wrong stack frame. Get the argument using an offset indirect addressing mode on the %esp itself. Figure out the required value for the offset since it will be different than if there were a stack frame on the stack.

The provided C driver (pollcountc.c) does the following:

a. Executes a do-nothing for loop that lasts between 4 and 5 seconds on the SAPCs (the first ten SAPCs all use the same Pentium processors so it shouldn't matter which one you use). This allows the FIFO in the COM2 port to become empty for the subsequent test.

b. Without any other output, goes into a loop calling pollcount_putc for 'A', 'B', 'C', 'D', and 'E' and putting each returned count in an array of int.

c. Prints out the five counts with printf.

Make a script of your run in pollcount.script. Think about the results--can you see the effect of the "double-buffering" described in the lecture notes?

Turn in a hard copy of all script files. Leave working versions of the following files in your mp2 project directory. The grader or I may run them to test them. We may also recompile some.

Every mp2 directory should have the following files:

```
     tiny.script and sum10.script
     count.s, countc.c, and count.script
     printbin.s, printbinc.c, and printbin.script
     echo.s, echoc.c, and echo.script
     pollcount.s, pollcountc.c, and pollcount.script
```

In the event that you are unable to correctly complete this assignment by the due date, do not remove the work you were able to accomplish - partial credit is always better than none.