Due: Monday, July 27th. Written 4pm, Programs 11:59pm.
File names: mystrstr.asm, strstr2.c, mult.mps, authors.txt

Written (30 points)

Written from Chapter 11 (24 points): 1; 2 b, d, f, g; 3 b, c, g, h; 4 b, d; 5, 6, 8, 11, 13.

MIPS Written: (6 points) What are the machine codes (in hexadecimal) for the instructions in the following MIPS code fragment?

```
loop: addu $4, $8, $11
lw  $5, 16($sp)
beq $8, 0, loop
nop
addi $7, $9, -2
j  loop
```

Programs

1) (25 points) Write an Intel assembly string searching function for the nasm assembler.
Filename: mystrstr.asm

We will test your function by calling it from within the strstr1.c program available in ~ssdavis/50/p5. The function will receive two parameters: 1) The address of an array of char; and 2) The address of an array of char. You should look at the assembly code of strstr1.c, using `gcc -masm=intel -S` to determine the placement and data length of the parameters on the stack. Your function must be register neutral. Your code must be efficient, and use registers instead of local variables. There should be no local variables. You will find that sorting1.c and sort.asm from a previous quarter will provide you some guidance. You will find sorting1.c, sort.asm, Makefile1, and strstr1.c in ~ssdavis/50/p5. Here is the algorithm you should implement. A copy of it is in strstr1.c, but unused.

```c
char* strstr(char *string, char *substring)
{
    int i;

    if (*substring == 0)
        return string;

    for(; *string != 0; string++)
    {
        if (*string != *substring)
            continue;

        i = 1;

        while (1)
        {
            if (substring[i] == 0)
                return string;

            if (string[i] != substring[i])
                break;

            i++;
        } // while
    } // for

    return (char*) 0;
}
```
Lessons learned:

1. Since you are working with chars, you will need to compare them in one byte versions of the registers, e.g. al or cl.
2. In gdb, use `set disassembly-flavor intel`, or in .gdbinit in your home directory have
   
   ```
   set disassembly-flavor intel
   set disassemble-next-line auto
   ```
3. In gdb, use `info registers, steapi (or si)` and `x/s $eax` to view information.
4. Work little by little, by writing some code, compiling, and then running. At the beginning, you will just want the program to handle the first strstr call, and comment out the balance of the code in main(). Use the standard push ebp; mov ebp, esp; to setup the frame pointer. Get your pushes and pops working properly before bothering writing your searching code.
5. 

   ```
   [ssdavis@lect1 private]$ strstr1.out
   People respond to people who respond.
   people who respond.
   OK
   People respond to people who respond.
   [ssdavis@lect1 private]$ strstr2.out
   People respond to people who respond.
   people who respond.
   OK
   People respond to people who respond.
   [ssdavis@lect1 private]$ 
   ```

2) (10 points) Filename: strstr2.c Imbed your assembly code from #1 in the mystrstr() function of strstr2.c. You will want to read http://www.reversing.be/article.php?story=20051203194931893. You can avoid enclosing every line in “asm(..)” by placing asm( above your assembly program, and placing ); below your program. You will still have to include each line in double quotes though. Comments must be prefaced with a # instead of a semi-colon. Get rid of the standard push ebp; move ebp, esp at the top of your program. Those two lines are automatically done by the function call. Once you have imbedded your code you can use Makefile2 to compile your code. You will find strstr2.c, and Makefile2 in ~ssdavis/50/p5 too.

   ```
   [ssdavis@lect1 private]$ strstr2.out
   People respond to people who respond.
   people who respond.
   OK
   People respond to people who respond.
   [ssdavis@lect1 private]$
   ```

3) (25 points) Filename: mult.mps

   Write a MIPS program that multiplies the values in an array of ten unsigned integers by the values contained in the in another array of ten unsigned integers, and print the results to the screen. The two arrays will start at 0xA00 and 0xA30. To print an integer, place it in $a0, set $v0 to 1, and make a syscall. To exit, set $v0 to 10, and make a syscall.

   You may assume that all of the integers are less than 0xFFFF so that the result will fit in one integer. You may not use the MULT, nor MULTU opcodes in your program. You must multiply by shifting and adding as in the example below. Your multiplication routine should be a subroutine called from your main program. You should use the appropriate registers, e.g. parameter passing registers, based on their role in the program. Note that hexadecimal numbers are prefaced with “0x” and not “$” in MIPS. Please do not start this until Saturday so that I can ensure that CUSP and ASIDE work properly.
In C++, you would implement:

```cpp
unsigned int multiply(unsigned int x, unsigned int y) {
    unsigned int reg = 0;
    while (y != 0) {
        if (y & 1)
            reg += x;
        x <<= 1;
        y >>= 1;
    } // while y not zero
    return reg;
} // multiply()

int main() {
    unsigned int a[10], b[10];
    for (int i = 0; i < 10; i++)
    {
        unsigned int result = multiply(a[i], b[i]);
        cout << result << endl;
    } // for i
    exit(0);
} // main()
```