Name: ___________________
NetID: __________@wisc.edu

Homework 6

CS/ECE 252 Section-2 (MWF 11:00)

Assigned on October 23rd
Due on Friday, November 6th by the beginning of class (11 AM)
The question will instruct you on whether to submit via hard copy or via the browser based infrastructure at https://discovering.cs.wisc.edu/homework/homework.html
Neat and legible handwriting is preferred, especially for your name and NetID.

1. a) Write the AVR assembly code equivalent to the following Python code snippet. **~~(3)~~(0)**
   Annotate your code with comments for full credit.
   Use the browser-based simulator to write and submit.

```
i = 23
k = 0
while(i > 12):
   j = i - 2
   while(j < 30):
      j = j + 10
      k = k + 1
   i = i - 3
```

   Hint: there is no addi AVR instruction for "j = j + 10". Therefore, you may have to use subi with negative 10 instead.

   b) What is the value of i, j, and k after the program halts? **~~(1)~~(0)**
   Submit via hard copy.

2. Write an AVR assembly program that outputs the indices of 'c' in "computer sciences". Use an assembler directive to load the string into memory and then loop through the memory loaded. It should print 0, 10, and 14. For simplicity, when using assembler directives to initialize RAM or using arrays of <= 256 in length, you can assume that all numbers are stored in the same HI memory pointer. In other words, the LO memory pointer will not overflow when incrementing it to loop through the array. This is because only X was taught in class (and not -X, X+, Y, -Y, Y+, Z, -Z, and Z+). **(3)**

   Annotate your code with comments for full credit.
   Use the browser-based simulator to write and submit.

3. In the following piece of code, what should the ***rjmp*** instruction's immediate values be, in case we want to jump to the label **(1.5)**
   a. L1
   b. L2
   c. L3

```
L1:
ldi r16,1
rjmp L1/L2/L3
ldi r19,1
L2:
ldi r17,1
L3:
ldi r18,1
```

   Note that label names are not a part of ISA, and they are not present in the program memory.

   Submit via hard copy.

   a. L1 =
   b. L2 =
   c. L3 =

4. The stack pointer has been set up to point to 65535, and registers r16 through r23 contains values to push. The top set of tables are the structures before the code is executed. Fill out the bottom set of tables after the code is executed. You need to fill out every highlighted cell. **(0)**

Register File			
r0 = 0	**r1 =** 0	**r2 =** 0	**r3 =** 0
r4 = 0	**r5 =** 0	**r6 =** 0	**r7 =** 0
...			
r16 = 86	**r17 =** 101	**r18 =** 114	**r19 =** 121
r20 = 69	**r21 =** 97	**r22 =** 115	**r23 =** 121

Stack Pointer
65535

RAM
RAM[65531] = 0
RAM[65532] = 0
RAM[65533] = 0
RAM[65534] = 0
RAM[65535] = 0

```
push r16   ; stack manipulation line 1, push 86
push r17   ; stack manipulation line 2, push 101
push r18   ; stack manipulation line 3, push 114
pop r0     ; stack manipulation line 4, pop to r0
push r19   ; stack manipulation line 5, push 121
push r20   ; stack manipulation line 6, push 69
pop r1     ; stack manipulation line 7, pop to r1
pop r2     ; stack manipulation line 8, pop to r2
push r21   ; stack manipulation line 9, push 97
push r22   ; stack manipulation line 10, push 115
pop r3     ; stack manipulation line 11, pop to r3
pop r4     ; stack manipulation line 12, pop to r4
push r23   ; stack manipulation line 13, push 121
pop r5     ; stack manipulation line 14, pop to r5
pop r6     ; stack manipulation line 15, pop to r6
pop r7     ; stack manipulation line 16, pop to r7
```

Register File			
r0 =	r1 =	r2 =	r3 =
r4 =	r5 =	r6 =	r7 =
...			
r16 =	r17 =	r18 =	r19 =
r20 =	r21 =	r22 =	r23 =

Stack Pointer

RAM
RAM[65531] =
RAM[65532] =
RAM[65533] =
RAM[65534] =
RAM[65535] =

Submit via hard copy.

5. The stack pointer has been set up to point to 2437. The top set of tables are the structures before the code is executed. Fill out the bottom set of tables after the code is executed. You need to fill out every highlighted cell. ~~(3)~~**(0)**

Register File	
r0 = 0	**r1** = 1
r2 = 2	**r3** = 3
r4 = 4	**r5** = 5
r6 = 6	**r7** = 7
r8 = 8	**r9** = 9

Stack Pointer
2437

RAM[2433] = 15
RAM[2434] = 14
RAM[2435] = 13
RAM[2436] = 12
RAM[2437] = 11
RAM[2438] = 10

```
push r7
push r6
pop r0
and r1, r2
push r8
pop r2
add r1, r0
pop r9
sub r2, r9
push r6
push r6
```

Register File	
r0 =	**r1** =
r2 =	**r3** =
r4 =	**r5** =
r6 =	**r7** =
r8 =	**r9** =

Stack Pointer

RAM[2433] =
RAM[2434] =
RAM[2435] =
RAM[2436] =
RAM[2437] =
RAM[2438] =

Submit via hard copy.

6. This question tests assembly directives.

Write an AVR assembly program that prints out the 3 bit input interpreted as signed magnitude, and then print out the 3 bit input interpreted as 2’s complement. You can assume that the 3 bits has been loaded into r30 and that it will be at most 3 bits long. You can load r30 with 0 through 7 for testing purposes. For simplicity, when using assembler directives to initialize RAM or using arrays of <= 256 in length, you can assume that all numbers are stored in the same HI memory pointer. In other words, the LO memory pointer will not overflow when incrementing it to loop through the array. This is because only X was taught in class (and not -X, X+, Y, -Y, Y+, Z, -Z, and Z+). **(0)**

```
ldi r30, 5
```

Input	3-bit presentation	signed magnitude interpretation		2’s complement interpretation	
		Actual	LCD output	Actual	LCD output
0	000	0	0	0	0
1	001	1	1	1	1
2	010	2	2	2	2
3	011	3	3	3	3
4	100	-0	0	-4	252
5	101	-1	255	-3	253
6	110	-2	254	-2	254
7	111	-3	253	-1	255

Hint: Since this is only asking for 3 bits, it might be a good idea to hard code the bytes with the assembler byte directive.

Annotate your code with comments for full credit.
Use the browser-based simulator to write and submit.

7. This question tests more AVR programming, and models the in-class example.
   Write an AVR assembly program that measures the length (inclusive) between the first binary 1 and the last binary 1 in a number, and print the length to the output LCD. You can assume that the number has been loaded into r30. You can load r30 with anything for testing purposes. **(3 extra credit)**

```
ldi r30, 116
```

Example 1: Let's assume that 116 is loaded into r30. 116 in binary is 01110100. The index of the first 1 is 1 (from left and starting from 0), and the index of the last 1 is 5, so the length is 5.

Example 2: Let's assume that 5 is loaded into r30. 5 in binary is 00000101. The index of the first 1 is 5 (from left and starting from 0), and the index of the last 1 is 7, so the length is 3.

Example 3: Let's assume that 255 is loaded into r30. 255 in binary is 11111111. The index of the first 1 is 0 (from left and starting from 0), and the index of the last 1 is 7, so the length is 8.

Example 4: Let's assume that 16 is loaded into r30. 16 in binary is 00010000. The index of the first 1 is 3 (from left and starting from 0), and the index of the last 1 is 3, so the length is 1.

Example 5: Let's assume that 0 is loaded into r30. 0 in binary is 00000000. Since there is no 1's, the length is 0.

Example 6: Let's assume that 41 is loaded into r30. 41 in binary is 01010010. The index of the first 1 is 1 (from left and starting from 0), and the index of the last 1 is 6, so the length is 6.

Annotate your code with comments for full credit.
Use the browser-based simulator to write and submit.

8. This question tests AVR programming.
Write a AVR program that takes a positive integer and print to output LCD the largest number that divides it that is not itself. You can assume that that the number has been loaded into r30. **(0)**

```
ldi r30, 116
```

Example 1: Let's say 116 is loaded into r30. Then 1, 2, 4, 29, 58, and 116 divides 116. The largest number that divides 116 that is not itself is 58.

Example 2: Let's say 24 is loaded into r30. Then 1, 2, 3, 4, 6, 12, and 24 divides 24. The largest number that divides 24 that is not itself is 12.

Example 3: Let's say 23 is loaded into r30. Then 1 and 23 divides 23. The largest number that divides 23 that is not itself is 1.

Example 4: Let's say 0 or 1 is loaded into r30. 0 and 1 is not a test case your program has to handle. Your program can handle edge cases 0 and 1 however you wish.

Example 5: Let's say 255 is loaded into r30. Then 1, 3, 5, 15, 17, 51, 85, and 255 divides 255. The largest number that divides 255 that is not itself is 85.

Annotate your code with comments for full credit.
Use the browser-based simulator to write and submit.

9. This question tests Memory (RAM) and array usage.
Repeat Homework 3 Question 9, but this time use assembly language instead of Python and use an array in RAM. You must set all 15 Fibonacci numbers in RAM before you begin printing. For simplicity, when using assembler directives to initialize RAM or using arrays of <= 256 in length, you can assume that all numbers are stored in the same HI memory pointer. In other words, the LO memory pointer will not overflow when incrementing it to loop through the array. This is because only X was taught in class (and not -X, X+, Y, -Y, Y+, Z, -Z, and Z+). **(5)**
Annotate your code with comments for full credit.
Use the browser-based simulator to write and submit.

For reference, Homework 3 Question 9:

Repeat Homework 2 Question 11, but this time store the Fibonacci numbers into an array and then print out the array at the end.

For reference, Homework 2 Question 11:

Write a program to print the first n Fibonacci numbers. (Initialize n as 15). Start with 0 and 1 as the first two numbers. The next number is created by adding the previous two numbers. Thus, the series would go like this: 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377.

Note: Due to overflow, it is okay for your program to instead print 0 1 1 2 3 5 8 13 21 34 55 89 144 233 **121**

For reference, answer to Homework 3 Question 9:

```
n = 15
i = 2
fa = [0,1]
while(i < n):
   fa = fa + [fa[i-1] + fa[i-2]]
   i = i + 1
i = 0
while(i < n):
   print(fa[i])
   i = i + 1
```

10. This question tests callee-save stack. Using very few registers are required to get practice with pushing and popping values from stack.

Repeat homework 5 question 15, but with 2 caveats. First caveat: you must use a function to convert from fahrenheit to celsius. The function should only contain the conversion (looping from 32 to 50 should be outside of the function). If the function needs more registers, then it must be callee-saved. Second caveat: you can only use 2 registers (r16 and r17 (along with IO registers 17, 18, 61, and 62))! **(7)**

Annotate your code with comments for full credit.
Use the browser-based simulator to write and submit.

For reference, problem statement to Homework 5 Question 15:

Repeat Homework 2 Question 12, but this time use **assembly language** instead of Python, and go from 32 to 50 instead.

For reference, problem statement to Homework 2 Question 12:

Say you wanted to print out the Celsius equivalent for all integer Fahrenheit temperatures from 32 degrees F to 50 degrees F. Write a program to print out this conversion information. The pseudocode for implementing this is given below.

The equation for converting Fahrenheit (F) to Celsius (C) is: $C = (F - 32) * \frac{5}{9}$

```
i.    Set F's initial value to 32 (lower bound)
ii.   While F is less than or equal to 50 (upper bound)
iii.      Convert Fahrenheit to Celsius.
iv.       Print the number of degrees in Fahrenheit
v.        Print the number of degrees in Celsius
vi.       Increment F
```

For reference, sample solution to Homework 3 Question 10:

```
def getCelsiusFromFahrenheit(F):      # here is function declaration
   return (F - 32) * 5 / 9            # here is function return value
# main, start of program
F = 32                                # initialize F to 32
while(F <= 50):                       # loop while F <= 50
   C = getCelsiusFromFahrenheit(F)    # call function and save into C
   print(F)                           # print F
   print(C)                           # print C
   F = F + 1                          # increment F
```

For reference, sample solution to Homework 5 Question 15:

```
; r16 is used for F
; r17 is used to  initialize IO 17
; r18 is used for F-32
; r19 is used for (F-32)*5
; r20 is used for C, (F-32)*5/9
; r21 is used for 51, since cpi is removed
; r22 is used for 9, since cpi is removed
ldi r21, 51    ; r21 = 51 (constant)
ldi r22, 9     ; r22 = 9 (constant)
ldi r16,32     ; F = 32
ldi r17, 255   ; set r16 to all high bits
out 17, r17    ; all high bits IO 17 for output
whileLoop:     ; label for while(F <= 50):
cp r16, r21    ; while(F < 51):, compare F to 51
brsh halt      ; while(F < 51):, break loop and halt if F >= 51
mov r18, r16   ; r18 = F
subi r18, 32   ; r18 = F - 32
ldi r19, 0     ; r19 = 0, could have optimize multiplication better
add r19, r18   ; r19 = (F - 32) * 1
add r19, r18   ; r19 = (F - 32) * 2
add r19, r18   ; r19 = (F - 32) * 3
add r19, r18   ; r19 = (F - 32) * 4
add r19, r18   ; r19 = (F - 32) * 5, our dividend
ldi r20, 0     ; r20 = 0, this is our quotient counter
divideLoop:    ; label to loop while dividend >= divisor
cp r19, r22    ; dividend >= 9 ?
brsh incrementQuotient ; if so, then branch to incrementQuotient
; else C = (F - 32) * 5 / 9
out 18, r16    ; print(F)
out 18, r20    ; print(C)
inc r16        ; F = F + 1
rjmp whileLoop ; end of while loop, jump back to check condition again
incrementQuotient: ; label to inc quotient and subtract from
dividend
subi r19, 9    ; subtract divisor, 9, from dividend
inc r20        ; increment quotient counter
rjmp divideLoop; end of incrementQuotient
; jump to check dividend >= divisor again
halt:          ; label to jump to if F >= 51
halt           ; halt to end program
```

11. This question tests caller-save stack.

Repeat the previous question, but this time, if the function needs more registers, then it must be caller-saved. **~~(2)~~(0)**

Annotate your code with comments for full credit.
Use the browser-based simulator to write and submit.

12. Use AVR assembly to convert a 16-bit fixed point binary number to a 16-bit floating point floating point number. Assume that the integer value has been loaded into r30 and the fractional value has been loaded into r31. Put the high byte in r29 and the low byte in r28. You may assume that the integer input will be non-zero less than or equal to 127 while the fractional input is not zero. You can load anything non-zero <= 127 into r30 and anything non-zero into r31 for testing purposes. **~~(4 extra credit)~~ (3 extra credit)**

```
ldi r30, 116
ldi r31, 123
```

Example: For this example, 0b is binary, 0d is decimal (default if not specified), and 0x is hexadecimal, and ignore the underscores (used for readability). Let's say 116 (0b0111_0100) is loaded into r30 and 123 (0b0111_1011) is loaded into r31. Then, the decimal value of this number is 116.48046875 as calculated below:

$$0*2^7+1*2^6+1*2^5+1*2^4+0*2^3+1*2^2+0*2^1+0*2^0$$
$$0*2^{-1}+1*2^{-2}+1*2^{-3}+1*2^{-4}+1*2^{-5}+0*2^{-6}+1*2^{-7}+1*2^{-8}$$

After normalizing, the exponent becomes 0d21 (6 + 15 bias) (0b1_0101) and the fraction becomes 0b1101_0001_1110_11 (the last 4 bits will be removed because only 10 bits of the fractional values is kept in the half-precision floating point notation. Therefore, the resulting 16-bit floating point number will be 0b0_10101_1101000111 = 0b0101_0111_0100_0111 = 0x5747. Therefore, r29 should have 87 (0x57) and ~~r30~~r28 should have 71 (0x47) at the end of this program.

Annotate your code with comments for full credit.
Use the browser-based simulator to write and submit.

13. This question tests assembling assembly language to machine code,

By hand, assemble the block of assembly language below to machine code. Check your answer with the browser-based assembler/simulator. **(0)**

Submit via hard copy (to avoid getting binary executables).

```
ldi r31,9
mov r1,r31
add r16,r1
~~subi r16,4~~
~~eor r17, r16~~
```

14. This question test executing machine code.

Interpret the block of machine code below by unassembling it back into assembly language. Check your answer with the assembler to see if it assembles back into the given machine code. **(3)**

Comments/annotation <u>not needed</u> for this question.

Use the browser-based simulator to write and submit.

```
1110000000001111
1110000000010010
0001011100010000
1111010000001000
0010111110100001
1001010100010011
```

or equivalently

```
0xe00f
0xe012
0x1710
0xf408
0x2fa1
0x9513
```