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. Write the AVR assembly code equivalent to the following Python code snippet. **(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.

Sample solution:

```
; r16 is used for i
; r17 is used for k
; r18 is used for j
; r19 is used for 13 for cp
; r20 is used for 30 for cp
ldi r16, 23                ; r16 = i = 23
ldi r17, 0                 ; r17 = k = 0
ldi r19, 13                ;load immediate cp values
ldi r20, 30                ;load immediate cp values
beginningOfOuterWhileLoop:; while(i > 12): begin label
cp r16, r19                ; i < 13,opposite i > 12
brlo halt                  ; halt if i < 13
mov r18, r16               ; r18 = j = i
subi r18, 2                ; r18 = j = i - 2
beginningOfInnerWhileLoop:; while(j < 30): begin label
cp r18, r20                ; j < 30
brsh endOfInnerWhileLoop ; end inner loop if j >= 30
subi r18, -10              ; r18 = j = j + 10
inc r17                    ; r17 = k = k + 1
rjmp beginningOfInnerWhileLoop; jmp to check condition
endOfInnerWhileLoop:       ;while(j <30): end label
subi r16, 3                ; r16 = i = i - 3
rjmp beginningOfOuterWhileLoop;while(i >12): end label
halt:                      ; halt label
halt                       ; end program
```

What is the value of i, j, and k after the program halts? **(0)**

Submit via hard copy.

i = 11; j = 32; k = 7;

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. **(3)**
   Annotate your code with comments for full credit.
   Use the browser-based simulator to write and submit.

Sample solution:

```
; r16 is used to load memory into
; r17 is used as 'c' counter
; r18 is used to configure IO reg 17 for output
; r19 is used for 'c' for cp
; r27 is used for HI memory pointer
; r26 is used for LO memory pointer
.string(text) "computer sciences" ; assembler directive
ldi r27, hi8(text) ; set up HI memory pointer
ldi r26, lo8(text) ; set up LO memory pointer
ldi r17, 0       ; set 'c' index to 0
ldi r19, 'c'     ; load immediate cp values
ldi r18, 255     ; set r18 to all high bits
out 17, r18      ; set IO reg 17 to all high for output

beginWhileLoop: ; label to loop while not end of string
ld r16, X        ; load character into r16
subi r16, 0      ; check if null terminating
breq endWhileLoop ; if so, then end loop
cp r16, r19      ; check if char = 'c' in ASCII
breq printIndex ; if so,then inc 'c' counter

endPrintIndex:  ; label to move LO memory pointer
inc r26          ; increment LO memory pointer
inc r17          ; increment string index
rjmp beginWhileLoop ; check while loop condition

printIndex:      ; label if detected 'c'
out 18, r17      ; print index of ‘c’
rjmp endPrintIndex ; jump to update LO memry ptr

endWhileLoop:    ; label for end of while loop
halt             ; end program
; per https://piazza.com/class/ie0hgwa9s8k623?cid=186
; it is okay to print in descending order
```

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 = -2
b. L2 = 1
c. L3 = 2

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.

Register File			
r0 = 114	r1 = 69	r2 = 121	r3 = 115
r4 = 97	r5 = 121	r6 = 101	r7 = 86
...			
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] = 115
RAM[65533] = 121
RAM[65534] = 101
RAM[65535] = 86

You can try it out for yourself!

```
; initialize stack pointer to bottom of memory
ldi r16, 255
out 62, r16
out 61, r16
; put values into register to push into stack
ldi r16, 86
ldi r17, 101
ldi r18, 114
ldi r19, 121
ldi r20, 69
ldi r21, 97
ldi r22, 115
ldi r23, 121
; stack manipulation line 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
halt      ; end program
```

Stack trace: Table 1 draws the contents of the stack and the stack pointer at the end of each line. Table 2 shows the sequence of popped values.

Table 1: Snapshots of the stack and stack pointer

Stack Pointer	65535	65534	65533	65532	65533	65532	65531	65532	65533	65532	65531	65532	65533	65532	65533	65534	65535
65531	0	0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
65532	0	0	**0**	**0**	**0**	**0**	**69**	**69**	**69**	**69**	**115**	**115**	**115**	**115**	**115**	**115**	**115**
65533	0	0	**0**	**114**	**114**	**121**	**121**	**121**	**121**	**97**	**97**	**97**	**97**	**121**	**121**	**121**	**121**
65534	0	0	**101**	**101**	**101**	**101**	**101**	**101**	**101**	**101**	**101**	**101**	**101**	**101**	**101**	**101**	**101**
65535	0	86	**86**	**86**	**86**	**86**	**86**	**86**	**86**	**86**	**86**	**86**	**86**	**86**	**86**	**86**	**86**
Line	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**

Table 2: Popped values in register file in [dec] format

Register File [dec]			
r0	114	**r1**	**69**
r2	**121**	**r2**	**115**
r4	**97**	**r3**	**121**
r6	**101**	**r4**	**86**

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. **(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.

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

Stack Pointer
2435

RAM[2433] = 15
RAM[2434] = 14
RAM[2435] = 13
RAM[2436] = 6
RAM[2437] = 6
RAM[2438] = 10

You can try it out for yourself!

```
; initialize stack pointer
ldi r16, 9      ; 2437 / 256
out 62, r16
ldi r16, 133    ; 2437 % 256
out 61, r16
; put values into register
ldi r16, 0
mov r0, r16
ldi r16, 1
mov r1, r16
ldi r16, 2
mov r2, r16
ldi r16, 3
mov r3, r16
ldi r16, 4
mov r4, r16
ldi r16, 5
mov r5, r16
ldi r16, 6
mov r6, r16
ldi r16, 7
mov r7, r16
ldi r16, 8
mov r8, r16
ldi r16, 9
mov r9, r16
; put values into memory
ldi r27, 9
```

```
ldi r26, 129
ldi r16, 15
st X, r16
inc r26
ldi r16, 14
st X, r16
inc r26
ldi r16, 13
st X, r16
inc r26
ldi r16, 12
st X, r16
inc r26
ldi r16, 11
st X, r16
inc r26
ldi r16, 10
st X, r16
; start stack operations
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
```

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. **(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.

Sample solution:

```
; r16 is used to store signed magnitude interpretation
; r17 is used to store 2's complement interpretation
; r18 is used to initialize DDRD port, IO reg 17
; r26 and r27 is used as memory pointer X
; r30 is used as program input
; assembler directive for signed magnitude
.byte(signedMagnitude) 0,1,2,3,0,-1,-2,-3
; assembler directive for 2's complement
.byte(twosComplement) 0,1,2,3,-4,-3,-2,-1
ldi r30, 5           ; user input
ldi r27, hi8(signedMagnitude) ; set up HI memory pointer
ldi r26, lo8(signedMagnitude) ; set up LO memory pointer
add r26, r30    ; get offset
ld r16, X       ; load signed magnitude interpretation
ldi r27, hi8(twosComplement) ; set up HI memory pointer
ldi r26, lo8(twosComplement) ; set up LO memory pointer
add r26, r30    ; get offset
ld r17, X       ; load 2's complement interpretation
ldi r18, 255    ; set r18 to all high bits
out 17, r18     ; set IO reg 17 to all high, for output
out 18, r16     ; display signed magnitude interpretation
out 18, r17     ; display 2's complement interpretation
halt            ; end program
```

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.

Note: The intention of this question is to provide more practice with AVR assembly programming that is modeled after in-class worksheet question 4. However, if you wanted to hard code a 256 length long look-up-table (LUT) to solve this question, then that's cool too.

Sample Solution:

```
; the general approach is the find the index of the leftmost 1
and the index of the rightmost 1
; if the input is 0, then the length is 0
; else the length is the difference + 1
; r16 is used as index of leftmost 1
; r17 is used as index of rightmost 1
; r18 is used as boolean, set false (0) upon detecting first 1
; r19 is used as indexCounter, loop while < 8
; r20 is used as shift register, used to check every bit of r30
; r21 is used to and with 1, to check rightmost bit
; r22 is used to set up output
; r23 is used for 1 for cp
; r30 is used as program input
ldi r30, 116   ; register that contains input
ldi r16, 0     ; right index
ldi r17, 0     ; left index
ldi r18, 1     ; boolean, isFirstBoolean, set to 0 if found
ldi r19, 8     ; indexCounter
mov r20, r30   ; shift register
ldi r23, 1     ; load immediate cp values

beginWhileLoop:; loop to check every bit
ldi r21, 1     ; r21 = 1
and r21, r20   ; r21 anded, will remain 1 if LSB is 1
cp r21, r23    ; compare to check if r21 == 1
breq isOne     ; update indexes if LSB is 1

endIfIsOne:    ; label to fall into if not 1
asr r20        ; shift the shift register
dec r19        ; decrement indexCounter
breq endWhileLoop ; break if indexCounter == 0
rjmp beginWhileLoop ; go check next bit

isOne:         ; label to update indexes if LSB is 1
cp r18, r23    ; check boolean if this is the first 1
breq isFirstOne; if so, then update both indexes
mov r17, r19   ; else just update the left index
rjmp endIfIsOne; jump to shift and increment counter

isFirstOne:    ; label if first 1
```

```
ldi r18, 0      ; set isFirstOne boolean to false
mov r16, r19    ; update right index
mov r17, r19    ; update left index
rjmp endIfIsOne; jump to shift and increment counter

endWhileLoop:   ; label when done checking all 8 bits
subi r30, 0     ; check if input was 0
breq isLength0  ; if so, then length is 0
mov r22, r16    ; else length is rightmost - leftmost + 1
sub r22, r17    ; subtract rightmost from leftmost
inc r22         ; plus 1
rjmp endCheckingLength ; jump to print output

isLength0:      ; label if input was 0
ldi r22, 0      ; then set length to 0

endCheckingLength: ; label done check length, go print output
ldi r23, 255    ; set r23 to all high bits
out 17, r23     ; set IO reg 17 to all high for output
out 18, r22     ; print answer
halt            ; end program
```

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.

Sample Python Solution:

```
import input                # import to take user input
numberInput = input.get_num("Enter a positive integer: ")
numberInput = 116
largestDivisor = 1          # keep track of largest divisor
index = 2                   # index to check if divides numberInput
while(index < numberInput - 1):     # check every smaller number
   if(numberInput % index == 0):    #if index divides numberInput
      largestDivisor = index        # update largestDivisor
   index = index + 1        # increment to check next index
print(largestDivisor)       # print largestDivisor
```

Sample AVR Solution:

```
;  this question was originally designed to test user input
; r16 used for numberInput
; r17 used for largestDivisor
```

```
; r18 used for index
; r19 used for numberInput - 1
; r20 used for dividend in divid for mod function
; import input by setting IO reg 17 and reading IO reg 16
; ldi r16, 0   ; set r16 to all low bits
; out 17, r16  ; move all low bits IO 17 for input
; numberInput = input.get_num("Enter a positive integer: ")
; read input from toggle switches to r16
; in r16, 16   ; r16 = numberInput
ldi r16, 116   ; r16 = numberInput
ldi r17, 1     ; r17 = largestDivisor = 1
ldi r18, 2     ; r18 = index = 2
mov r19, r16   ; r19 = numberInput
dec r19        ; r19 = numberInput - 1
whileLoopBegin:; while(index < numberInput - 1):, label
cp r18, r19    ; while(index < numberInput - 1):, compare
brsh whileLoopEnd ;while(index < numberInput - 1):, end if false
mov r20, r16   ; r20 = numberInput
modLoopBegin:  ; label to begin mod (%) operation
cp r20, r18    ; while(dividend >= divisor), compare
brsh decrementDividend ; while(dividend >= divisor), go subtract
rjmp modLoopEnd; else end while loop and jmp to modLoopEnd
decrementDividend:; label to subtract divisor from dividend
sub r20, r18   ; subtract divisor from dividend
rjmp modLoopBegin ; go back to while(dividend >= divisor)
modLoopEnd:    ; label when mod operation done
subi r20, 0    ; if(numberInput % index == 0):,if remainder is 0
breq storeLargestDivisior ; largestDivisor = index
ifBlockEnd:    ; label for after if(numberInput % index == 0):
inc r18        ; index = index + 1
rjmp whileLoopBegin ; jmp to check condition for while loop
storeLargestDivisior: ; label for inside if block
mov r17, r18   ; r17 = largestDivisor = index
rjmp ifBlockEnd; label for end of inside of if block
whileLoopEnd:  ; label for end of while block
ldi r16, 255   ; set r20 to all high bits
out 17, r16    ; all high bits IO 17 for output
out 18, r17    ; print(largestDivisor)
halt           ; end program
```

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. <u>You must set all 15 Fibonacci numbers in RAM before you begin printing.</u> **(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
```

Sample solution 1:
Translating the following with assembler directive

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

```
; r16 = n
; r17 = i
; r26 and r27 = X = &fa
; r18 = fa[i-1], and reused for fa[i-1] + fa[i-2]
; r19 = fa[i-2], and reused to set up DDRD
.byte(fa) 0,1          ; fa = [0,1,0,0,0,0,0,0,0,0,0,0,0,0,0]
ldi r16, 15            ; n = 15
ldi r17, 2                  ; i = 2
ldi r26, lo8(fa)       ; X = &fa
ldi r27, hi8(fa)       ; X = &fa

begin_store_loop:      ; while(i < n):
cp r17, r16            ; while(i < n):
brsh end_store_loop    ; while(i < n):
ldi r26, lo8(fa)       ; X = &fa
add r26, r17           ; X = &fa[i]
dec r26                ; X = &fa[i-1]
ld r18, X              ; r18 = fa[i-1]
dec r26                ; X = &fa[i-2]
ld r19, X              ; r19 = fa[i-2]
inc r26                ; X = &fa[i-1]
inc r26                ; X = &fa[i]
add r18, r19           ; r18 = fa[i-1] + fa[i-2]
st X, r18              ; fa[i] = fa[i-1] + fa[i-2]
inc r17                ; i = i + 1
rjmp begin_store_loop; while(i < n):
```

```
end_store_loop:      ; while(i < n):
ldi r17, 0                ; i = 0

begin_print_loop:    ; while(i < n):
cp r17, r16          ; while(i < n):
brsh end_print_loop ; while(i < n):
ldi r26, lo8(fa)     ; X = &fa
add r26, r17         ; X = &fa[i]
ld r18, X            ; r18 = fa[i]
ldi r19, 255         ; print(fa[i])
out 17, r19          ; print(fa[i])
out 18, r18          ; print(fa[i])
inc r17              ; i = i + 1
rjmp begin_print_loop; while(i < n):
end_print_loop:      ; while(i < n):
```

Sample solution 2 without assembler directive:

```
; r16 is used for n
; r17 is used for i
; r18 is used for fa[i]
; r18 is not clear from previous iteration
; therefore, r18 is also used for fa[i-1]
; r19 is used for fa[i-2]
; r26 is used as address pointer
ldi r16, 15     ; n = 15
ldi r17, 2      ; i = 2
ldi r18, 0      ; fa = [0]
ldi r26, 0      ; fa = [0]
st X, r18       ; fa = [0]
ldi r18, 1      ; fa = [0,1]
ldi r26, 1      ; fa = [0,1]
st X, r18       ; fa = [0,1]

storeLoop:      ; while(i<n):, while loop label to jump to
cp r17, r16     ; while(i<n), compare i to n
brsh endStoreLoop ; while(i<n), exit loop if i >= n
mov r26, r17    ; r26 = i
subi r26, 2     ; r26 = i-2
ld r19, X       ; r19 = fa[i-2]
add r18, r19    ; r18 = fa[i-1] + fa[i-2]
mov r26, r17    ; r26 = i
st X, r18       ; fa = fa + [fa[i-1] + fa[i-2]]
inc r17         ; i = i + 1
rjmp storeLoop  ; end of 1st while loop, go check condition again

endStoreLoop:   ; after first while loop
ldi r26, 0      ; i = 0
ldi r17, 255    ; set r20 to all high bits
out 17, r17     ; all high bits IO 17 for output

printLoop:      ; while(i < n):
cp r26, r16     ; while(i < n), compare i to n
brsh halt       ; while(i < n), exit loop if i >= n
ld r17, X       ; load fa[i]
out 18, r17     ; print(fa[i])
inc r26         ; i = i + 1
rjmp printLoop  ; end of while loop, go check condition again
halt:           ; label for after second while loop
halt            ; end program
```

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. 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):
```
~~; cpi r16,51 ; while(F < 51):, compare F to 51~~
```
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
```

Sample solution:

```
; Inside function
;    r16 is used for F - 32
;    r17 is used for (F - 32) * 5
;    r16 is reused for (F - 32) * 5 / 9
; Outside function
;    r16 used to set up stack pointer and DDRD
;    r16 is used to pass in fahrenheit value
;    r17 holds a copy of the fahrenheit value
;    r16 is used to catch the celsius value from the function
rjmp main       ; functions are declared on top, jump to main


getCelsiusFromFahrenheit: ; function label
push r17        ; callee-save register
subi r16, 32    ; r16 = F - 32
mov r17, r16    ; r17 = (F - 32) * 1
add r17, r16    ; r17 = (F - 32) * 2
add r17, r16    ; r17 = (F - 32) * 3
add r17, r16    ; r17 = (F - 32) * 4
add r17, r16    ; r17 = (F - 32) * 5, our dividend
ldi r16, 0      ; r16 = 0

divideLoop:     ; label to loop while dividend >= divisor
; cpi r17, 9    ; dividend >= 9 ?
subi r17, 9     ; dividend >= 9 ?
; the step above is weird since cpi is removed, so subtract 9
; and branch if it was less than 9
brsh incrementQuotient ; if so, then branch to incrementQuotient
; else C = (F - 32) * 5 / 9
pop r17         ; pop r17 since about to return from function
ret             ; return from function

incrementQuotient: ; label to inc quotient, subtr from dividend
; subi r17, 9   ; subtract divisor, 9, from dividend
inc r16         ; increment quotient counter
rjmp divideLoop; jump to check dividend >= divisor again
```

```
main:           ; label to start program
ldi r16, 255    ; set r16 to all high bits
out 61, r16     ; all high bits IO 61 for SP LO
out 62, r16     ; all high bits IO 62 for SP HI
out 17, r16     ; all high bits IO 17 for output
ldi r16,32      ; F = 32
beginWhileLoop:; label for while(F <= 50):
ldi r17, 51     ; load immediate to do comparison
cp r16, r17     ; while(F < 51):, compare F to 51
brsh halt       ; while(F < 51):, break loop and halt if F >= 51
mov r17, r16    ; r17 = F
rcall getCelsiusFromFahrenheit ; function call to get Celsius
out 18, r17     ; print(F)
out 18, r16     ; print(C)
mov r16, r17    ; r16 = F
inc r16         ; F = F + 1
rjmp beginWhileLoop ; jump to check while loop condition

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. **(0)**

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

Sample solution:

```
; Inside function
;    r16 is used for F - 32
;    r17 is used for (F - 32) * 5
;    r16 is reused for (F - 32) * 5 / 9
; Outside function
;    r16 used to set up stack pointer and DDRD
;    r16 is used to pass in fahrenheit value
;    r17 holds a copy of the fahrenheit value
;    r16 is used to catch the celsius value from the function
rjmp main       ; functions are declared on top, jump to main


getCelsiusFromFahrenheit: ; function label
; push r17      ; removed since no longer callee-save register
subi r16, 32    ; r16 = F - 32
mov r17, r16    ; r17 = (F - 32) * 1
add r17, r16    ; r17 = (F - 32) * 2
add r17, r16    ; r17 = (F - 32) * 3
add r17, r16    ; r17 = (F - 32) * 4
add r17, r16    ; r17 = (F - 32) * 5, our dividend
ldi r16, 0      ; r16 = 0

divideLoop:     ; label to loop while dividend >= divisor
subi r17, 9     ; dividend >= 9 ?
; the step above is weird since cpi is removed, so subtract 9
; and branch if it was less than 9
brsh incrementQuotient ; if so, then branch to incrementQuotient
; else C = (F - 32) * 5 / 9
; pop r17       ; removed since no longer callee-save register
ret             ; return from function
```

```
incrementQuotient: ; label to inc quotient, subtr from dividend
; subi r17, 9  ; subtract divisor, 9, from dividend
inc r16        ; increment quotient counter
rjmp divideLoop; jump to check dividend >= divisor again

main:          ; label to start program
ldi r16, 255   ; set r16 to all high bits
out 61, r16    ; all high bits IO 61 for SP LO
out 62, r16    ; all high bits IO 62 for SP HI
out 17, r16    ; all high bits IO 17 for output
ldi r16,32     ; F = 32
beginWhileLoop:; label for while(F <= 50):
ldi r17, 51    ; load immediate to do comparison
cp r16, r17    ; while(F < 51):, compare F to 51
brsh halt      ; while(F < 51):, break loop and halt if F >= 51
mov r17, r16   ; r17 = F
push r17       ; save r17 = F, added since caller-saved
rcall getCelsiusFromFahrenheit ; function call to get Celsius
pop r17        ; restore r17 = F, added since caller-saved
out 18, r17    ; print(F)
out 18, r16    ; print(C)
mov r16, r17   ; r16 = F
inc r16        ; F = F + 1
rjmp beginWhileLoop ; jump to check while loop condition
halt           ; label to jump to if F >= 51
halt           ; halt to end program
```

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 less than or equal to 127 while the fractional input is not zero. You can load anything <= 127 into r30 and anything non-zero into r31 for testing purposes. **(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 = 0d22343. Therefore, r29 should have 86 and r28 should have 71 at the end of this program.

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

Sample Solution from a student:

```
ldi r30, 116          ; f[15:8]
ldi r31, 123          ; f[7:0]

main:
ldi r16, 100
out 61, r16
out 62, r16
rjmp pushBitsToStack
finishPushing:
rjmp loadBitsFromStack
funishLoading:
rjmp exit

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;Push all the bits into a stack
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pushBitsToStack:
ldi r16, 1            ; 00000001 (used for extracting bit)
ldi r17, 0            ; counter = 0
mov r11, r31          ; temp = f[7:0]

pushLOBits:           ; extract the bits from f[7:0]
mov r12, r11  ; tempB = temp (don't want to lose all the bits)
cpi r17, 8            ; if counter == 8
breq endPushLOBits    ; end the loop
and r12, r16          ; extract the least significant bit
push r12              ; push the extracted bit into stack
inc r17               ; counter = counter + 1
asr r11               ; temp >> 1
rjmp pushLOBits

endPushLOBits:        ; extract the bits from f[15:8]
ldi r17, 0            ; reinitialize counter to 0
mov r11, r30          ; temp = f[15:8]

pusHIBits:          ; same mechanism as the above "pushLOBits"
mov r12, r11
cpi r17, 8
breq endPushBits
```

```
and r12, r16
push r12
inc r17
asr r11
rjmp pusHIBits

endPushBits:
rjmp finishPushing

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;Loads bits from the stack
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
loadBitsFromStack:
ldi r29, 0          ; new f[15:8]
ldi r28, 0          ; new f[7:0]
ldi r22, 22         ; exp = 22 (max expon rep. by fixing point)
ldi r23, 0          ; counterA = 0
ldi r24, 0          ; counterB = 0 (check if f[15:8] is full)
ldi r25, 0          ; counterC = 0 (check if f[7:0] is full)
ldi r26, 0          ; nonzero_check = false (begins with zeroes)
ldi r27, 0          ; temp = 0 (for storing loaded bits)

ldi r16, 85
out 61, r16
ldi r16, 100
out 62, r16

loadBitsLoop:
cpi r23, 16       ; loaded all the extracted bits from the stack
brsh endLoop
pop r27             ; load a bit from the top of the stack
cpi r27, 1
brne 1
ldi r26, 1        ; if the loaded bit == 1, nonzero_check = true
cpi r26, 0
breq doNothing ; if nonzero_check == false, cont to the next
loop
cpi r24, 3
brne loadHIBits  ; if f[15:8] not full, load new bit in f[15:8]
cpi r25, 8
brne loadLOBits   ; if f[7:0] not full, load new bit into f[7:0]
```

```
rjmp endLoop          ; else end the loop

doNothing:
dec r22               ; exp = exp - 1
rjmp nextLoop

loadHIBits:
add r29, r29          ; f[15:8] << 1
add r29, r27          ; f[8] = the loaded bit
inc r24               ; counterB = counterB + 1
rjmp nextLoop

loadLOBits:
add r28, r28          ; f[7:0] << 1
add r28, r27          ; f[0] = the loaded bit
inc r25               ; counterC = counterC + 1
rjmp nextLoop

nextLoop:
inc r23               ; counterA = counterA + 1
rjmp loadBitsLoop

endLoop:
subi r29, 4           ; f[15:8] - 00000100 (remove its MSB)
subi r22, 1           ; exp = exp - 1 (as the MSB got removed)
add r22, r22          ; exp << 2
add r22, r22
add r29, r22          ; integrate the exponent bits into f[15:8]
rjmp funishLoading

exit:
```

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~~

1110000011111001
0010111000011111
0000110100000001
~~**0101000000000100**~~
~~**0010011100010000**~~

or equivalently

0xe0f9
0x2e1f
0x0d01
~~**0x5004**~~
~~**0x2710**~~

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
```

```
ldi r16, 15
ldi r17, 2
cp r17, r16
brsh 1
mov r26, r17
inc r17
```