ECE190 Exam 2, Fall 2006

Monday 30 October

Name:

- **Be sure that your exam booklet has 13 pages.**
- **The exam is <u>meant</u> to be taken apart!**
- **Write your name at the top of each page.**
- **This is a closed book exam.**
- **You may not use a calculator.**
- **You are allowed TWO 8.5×11" sheets of notes.**
- **Absolutely no interaction between students is allowed.**
- **Show all of your work.**
- **More challenging questions are marked with ***.**
- **Don't panic, and good luck!**

"Adaptation of the a priori to the real world has no more originated from 'experience' than adaptation of the fin of the fish to the properties of water."
—K. Lorenz, as quoted by N. Chomsky in *Language and Mind*, as quoted by O. Sacks in *Seeing Voices*

Problem 1	20 points	________________
Problem 2	20 points	________________
Problem 3	20 points	________________
Problem 4	20 points	________________
Problem 5	20 points	________________
Total	100 points	________________

Problem 1 (20 points): Short Answers

Please answer concisely. If you find yourself writing more than a few words or a simple drawing, your answer is probably wrong.

Part A (5 points): Consider the following C function:

```
void /* returns nothing */
func (int x)
{
    switch ((5 < x) - (3 > x)) {
        case -1:
            printf ("Too cold\n");
            break;
        case 1:
            printf ("Too hot\n");
            break;
        case 0:
            printf ("Just right\n");
            break;
        default:
            printf ("Weird weather!\n");
            break;
    }
}
```

Fill in the blanks below to re-implement the function using `if` statements.

```
void /* returns nothing */
func (int x)
{
    if (___________________________) {
        printf ("Too cold\n");

    } else if (___________________________) {
        printf ("Too hot\n");

    } else if (___________________________) {
        printf ("Just right\n");
    } else {
        printf ("Weird weather!\n");
    }
}
```

Part B (5 points): Describe one advantage and one disadvantage of making a variable global rather than local to a certain function in a C program. *Hint: the disadvantages outweigh the advantages in practice, particularly for large programs.*

Problem 1, continued:

Part C (5 points): The C program below is intended to print the numbers from 10 down to 0 with one number per line. What does it actually do, and how could you fix it with one simple change?

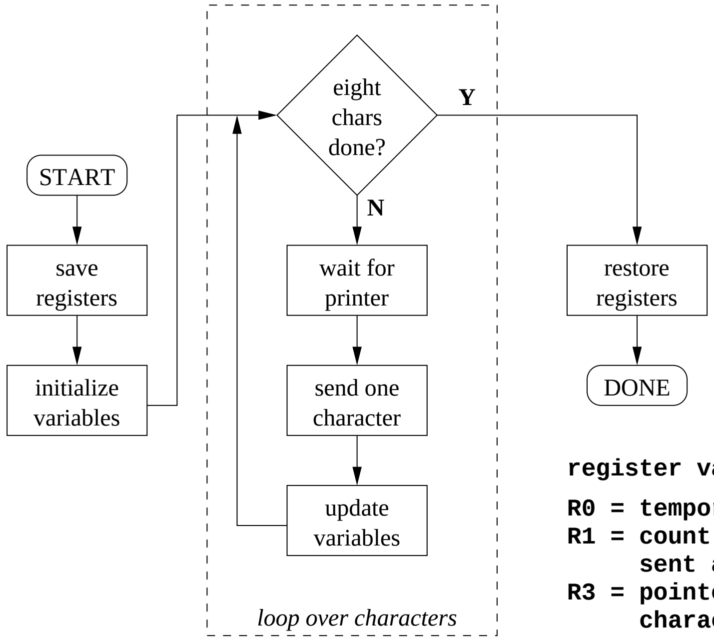
```
#include <stdio.h>

int
main ()
{
    int x;

    for (x = 10; 0 < x; --x) {
        printf ("%d\n", x);
    }
    return 0;
}
```

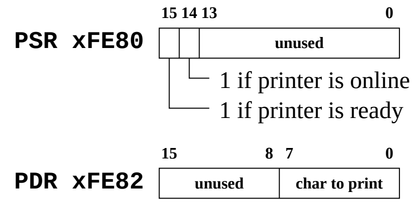
Part D*** (5 points): Your friend is developing a magic 8-ball program for the LC-3. He shows you the following assembly code:

```
        LEA      R1, SOURCE
        LEA      R2, DEST
LOOP    LDR      R0, R1, #0
        STR      R2, R0, #0
        BRz      DONE
        ADD      R1, R1, #1
        ADD      R2, R2, #1
        BRnzp    LOOP
DONE:   LEA      R0, DEST
        TRAP     x22         ; PUTS
        TRAP     x25         ; HALT
SOURCE  .STRINGZ "\"My sources say no\""
DEST    .BLKW    #20
MYDATA  .FILL    x0FFF
```

Your friend complains that when he runs this code with his test cases, it never finishes executing (in other words, it never reaches the HALT trap). Explain why. (Note that the two-character sequence \" inserts a single quotation mark, ASCII character x22, into a string.)

Problem 2 (20 points): Systematic Decomposition to LC-3 Assembly






```
register variables for trap routine

R0 = temporary values
R1 = count of characters
     sent already
R3 = pointer to next
     character to send
```

Prof. Lumetta needs your help: a new printer device has been added to the LC-3, but he has not been able to write one of the trap routines, and the next ECE190 assignment requires that trap routine! The trap routine in question sends a sequence of eight characters stored in memory starting at R3 (an input value) to the printer. Before sending each character to the printer, the trap routine must wait until both the online and ready bits of the PSR are equal to 1. The character can then be written to PDR.

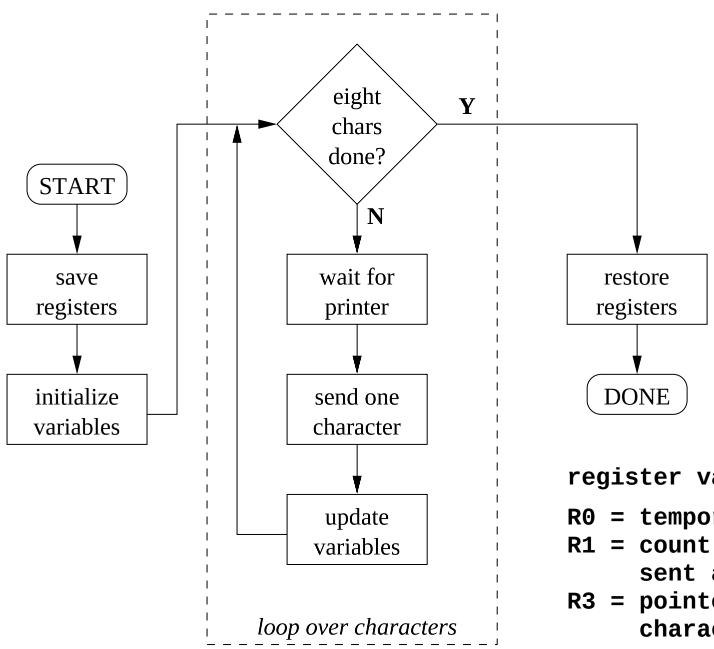
The figure above shows three things: on the left, a partial systematic decomposition for the trap routine (partial because it requires more than one LC-3 instruction for each box); in the upper right, the addresses and pictures of the new Printer Status Register (PSR) and Printer Data Register (PDR); in the lower right, the mapping from registers to data values that you'll need to use in the trap routine.

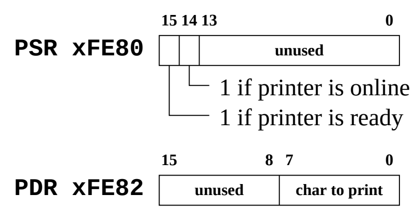
Part A (5 points): First protect the registers. The trap should preserve **all** register values. Fill in the code and allocate storage as necessary below to accomplish this goal. Two data values have been provided for Parts B and C.

```
; save registers (FILL IN)




; initialize variables and loop over characters (Part C)
; restore registers (FILL IN)




RET ; DONE
; data needed for trap routine (FILL IN)




TRAP_PSR .FILL xFE80
TRAP_PDR .FILL xFE82
```

Problem 2, continued:

Part B (5 points): The next step is to decompose the "wait for printer" box to the level of individual LC-3 instructions. Before sending a character to the printer, the trap routine must wait until both the online and ready bits of the PSR are equal to 1. Draw your answer as a flow chart with RTL or assembly inside each statement or test. For example, you might label a test with "BR" and write N, Z, and P on the appropriate output arcs. Use the register mapping shown in the figure (replicated below). Use data values from **Part A** (you should not need any others).






```
register variables for trap routine
R0 = temporary values
R1 = count of characters
     sent already
R3 = pointer to next
     character to send
```

Problem 2, continued:

Part C (8 points): You are now ready to write the main body of the code. Do so below. Remember that R3 initially points to the first of the eight characters to be sent, and that the others are in consecutive memory locations.

```
; save registers (Part A; NO NEED TO REWRITE)
; initialize variables (FILL IN; SEE REGISTER MAP FOR CONTENTS)



; all characters done?  (FILL IN)



; wait for printer (FILL IN from Part B)



; send one character (FILL IN)



; update variables (FILL IN)



; restore registers, DONE, and data (Part A; NO NEED TO REWRITE)
```

Part D*** (2 points): The printer has a button that turns it online/offline under human control. Using the protocol described, the printer **must** buffer one character even if the character is sent to PDR when the printer is offline. Explain why this buffering is necessary for correct behavior even though your code checks for the online bit before writing to PDR.

Problem 3 (20 points): The LC-3 Assembler

Consider the LC-3 program shown below. The numbers to the left are to help you answer the questions and **are not part of the program**. *Hint: what the program does is not important to the problem!*

```
01 .ORIG x3000
02
03 INIT
04     LEA   R0, START_STR
05     JSR   PRINT_STR
06     LD    R0, TEN
07     LEA   R1, DATA_B
08
09 STORE_LOOP
10     STR   R0, R1, #0
11     ADD   R1, R1, #1
12     ADD   R0, R0, #-1
13     BRp   ST_LOOP
14
15     LD    R0, TEN
16     ADD   R1, R1, #-1
17     AND   R2, R2, #0
18
19 ADD_LOOP
20     LDR   R3, R1, #0
21     ADD   R2, R3, R2
22     ADD   R1, R1, #-1
23     ADD   R0, R0, #-1
24     BRp   ADD_LOOP
25
26 STORE_SUM
27     ST    R2, RESULT
28     TRAP  #25
29
30 PRINT_STR
31     ST    R7, SAVE_R7
32     PUTS
33     LD    R7, SAVE_R7
34     RET
35
36 TEN       .FILL #10
37 SAVE_R7   .BLKW #1
38 DATA_B    .BLKW #10
39 START_STR .STRINGZ "Starting..."
40 RESULT    .FILL #0
41 .END
```

Label	Address
INIT	
STORE_LOOP	
ADD_LOOP	
STORE_SUM	
PRINT_STR	
TEN	
SAVE_R7	
DATA_B	
START_STR	
RESULT	

Part A (10 points): Fill in the addresses for the symbol table above as they would be generated by the assembler.

Part B (4 points): Write out the binary word that would be generated by the assembler for **line 15** of the program.

Part C (6 points): Assuming that both passes of the assembler were to execute, indicate which line numbers result in errors reported by the assembler, specify in which pass each error occurs, and *briefly* explain why each is an error.

Problem 4 (20 points): From C to LC-3 and Back Again

Part A (10 points): translate the C function below to LC-3 assembly instructions. The diagram of the stack frame for the function call has been provided for you.

Translate the `while` and `return` statements from the function body **independently, with no register values shared between sections**. The stack frame management and register save/store has been done for you (not shown in figures).

```
char* find_char
(char* str, char a)
{
   char* strcpy = str;

   while(*strcpy != a){
      strcpy++;
   }
   return strcpy;
}
```

`strcpy`	← R6,R5
prev. frame pointer	
return address	
return value	
`str`	
`a`	

stack frame for `find_char`

; create stack frame and save registers
…
; char* strcpy = str;
; DO NOT WRITE IN THIS BOX

; translation for while loop
;while(*strcpy != a) {
; strcpy++;
; }

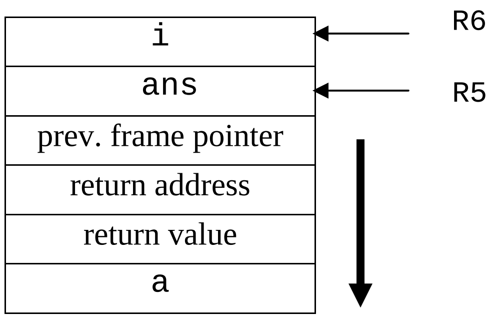
; translation for
; return i;

; restore registers and tear down stack frame
…
; DO NOT WRITE IN THIS BOX

Problem 4, continued:

Given below is part of the LC-3 translation of a C function `foo` and part of the function `foo` itself. Also given is the stack frame (activation record) for `foo`. Remember that memory addresses increase in the direction of the arrow. Answer the questions below.

```
int foo (int a)
{
  int ans = 0;
  int i;
  for (i = a; 0 < i; i--)  {

   /* body of loop written
    * by you in Part B
    */
  }
  return ans;
}
```



stack frame for `foo`

```
           .
           .
           .
        AND R0, R0, #0   ;1
        STR R0, R5, #0   ;2
        LDR R0, R5, #4   ;3
        STR R0, R5, #-1  ;4
LOOP
        LDR R0, R5, #-1  ;5
        BRnz DONE        ;6
        LDR R0, R5, #-1  ;7
        LDR R1, R5, #0   ;8
        ADD R1, R1, R0   ;9
        STR R1, R5, #0   ;10
        LDR R0, R5, #-1  ;11
        ADD R0, R0, #-1  ;12
        STR R0, R5, #-1  ;13
        BRnzp LOOP       ;14

DONE
        LDR R0, R5, #0   ;15
        STR R0, R5, #3   ;16
           .
           .
           .
```

Part B (6 points): Which LC-3 instructions correspond to (give the instruction numbers shown in the comments):

a. The initialization of the `for` loop?

b. The test part of the `for` loop?

c. The update (re-initialization) of the `for` loop?

Part C (4 points): Using the LC-3 translation of `foo`, write the body of the `for` loop here.

Problem 5 (20 points): C and Stack Frames

This question focuses on the program below, and particularly on the stack frames (also called activation records) that are used by each function in the program.

```
#include <stdio.h>

/* function declarations */
int bar (int a, int b);
int foo (int* p);

int bar (int a, int b)
{
    int x = a + b;

    if (0 < a) {
        printf ("%d\n", a * b);
    }
    return x;
}

int foo (int* p)
{
    *p = bar (-4, 11);
    return 6;
}

int main ()
{
    int x = 0;
    int y;

    y = foo (&x);
    bar (x, y);
    return 0;
}
```

Part A (3 points): When someone runs the program, what is the order of subroutine calls for the program, starting from `main`? In other words, what is the sequence of JSR target over the whole program execution? Give a comma-separated list, including only the `main`, `foo`, and `bar` functions.

`main`,

Part B (3 points): What, if anything, is printed by the program?

Problem 5, continued:

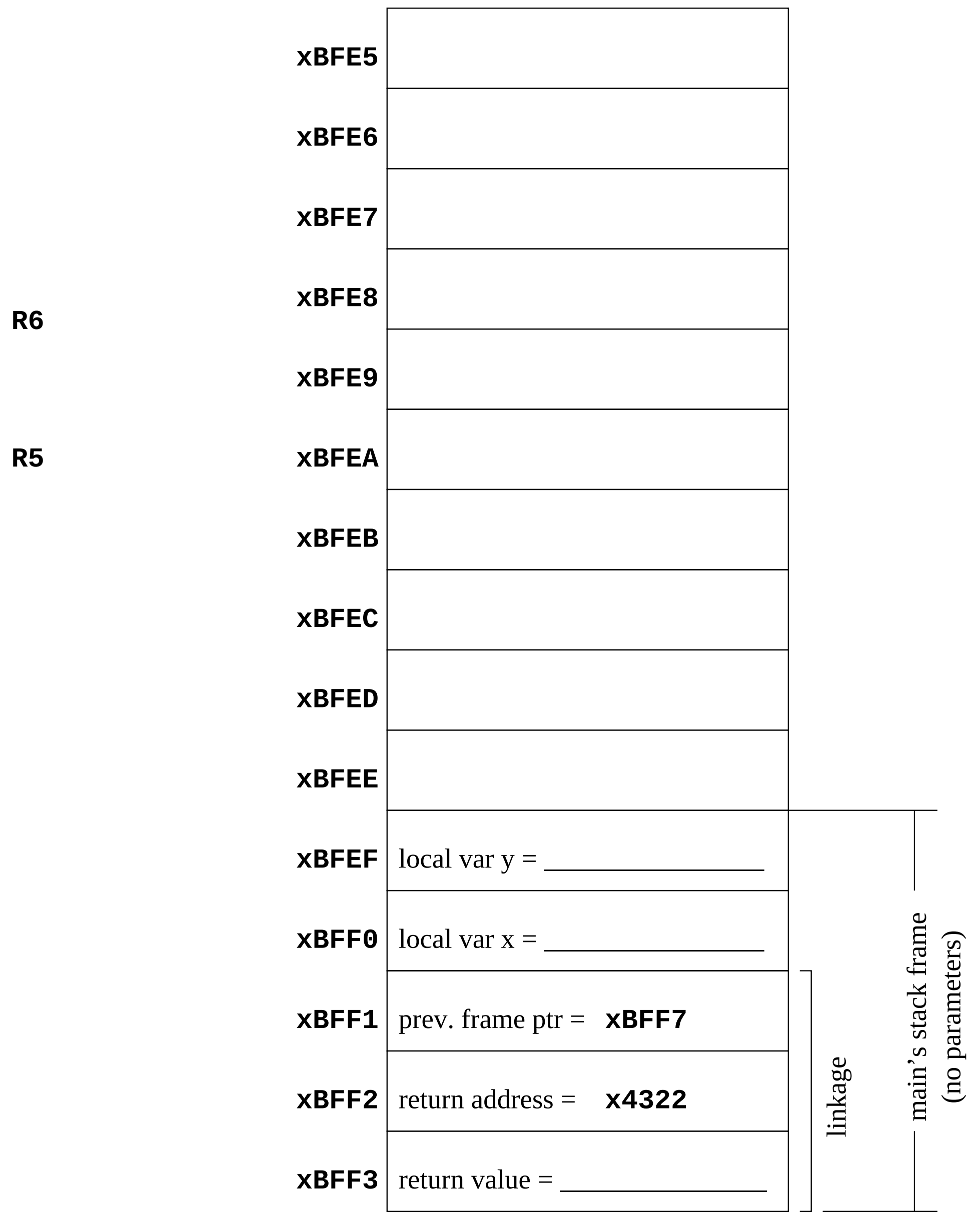
Part C (14 points): The stack frame for the `main` function is shown below. During execution of `main`, the stack pointer `R6=xBFEF`, and the frame pointer `R5=xBFF0`.

Use the figure to draw the stack just after completion of the `return` statement in the `bar` function **when it is called from** `main`, *i.e.*, just before `bar`'s stack frame is torn down and the subroutine returns to `main`.

Draw arrows to indicate the values of `R6` and `R5` at the point of program execution just described. For each memory location included in the stack (*i.e.*, between the stack pointer and the bottom of the figure), label the location with the type of information **and** the value stored there. If a memory location's value **cannot** be known, put a question mark by the description, *e.g.*, "x=?".

Do not mark or label any locations above the stack pointer, even if you know the values in those locations!

The address of the `JSR bar` instruction in `main` is `x3040`.



Use this page for scratch paper.

NOTES: RTL corresponds to execution (after fetch!); JSRR not shown

Instruction	Encoding	Assembly	RTL
ADD	0001 | DR | SR1 | 0 | 00 | SR2	ADD DR, SR1, SR2	DR ← SR1 + SR2, Setcc
ADD	0001 | DR | SR1 | 1 | imm5	ADD DR, SR1, *imm5*	DR ← SR1 + SEXT(imm5), Setcc
AND	0101 | DR | SR1 | 0 | 00 | SR2	AND DR, SR1, SR2	DR ← SR1 AND SR2, Setcc
AND	0101 | DR | SR1 | 1 | imm5	AND DR, SR1, *imm5*	DR ← SR1 AND SEXT(imm5), Setcc
BR	0000 | n | z | p | PCoffset9	BR{nzp} *PCoffset9*	((n AND N) OR (z AND Z) OR (p AND P)): PC ← PC + SEXT(PCoffset9)
JMP	1100 | 000 | BaseR | 000000	JMP BaseR	PC ← BaseR
JSR	0100 | 1 | PCoffset11	JSR *PCoffset11*	R7 ← PC, PC ← PC + SEXT(PCoffset11)
TRAP	1111 | 0000 | trapvect8	TRAP *trapvect8*	R7 ← PC, PC ← M[ZEXT(trapvect8)]
LD	0010 | DR | PCoffset9	LD DR, *PCoffset9*	DR ← M[PC + SEXT(PCoffset9)], Setcc
LDI	1010 | DR | PCoffset9	LDI DR, *PCoffset9*	DR ← M[M[PC + SEXT(PCoffset9)]], Setcc
LDR	0110 | DR | BaseR | offset6	LDR DR, BaseR, *offset6*	DR ← M[BaseR + SEXT(offset6)], Setcc
LEA	1110 | DR | PCoffset9	LEA DR, *PCoffset9*	DR ← PC + SEXT(PCoffset9), Setcc
NOT	1001 | DR | SR | 111111	NOT DR, SR	DR ← NOT SR, Setcc
ST	0011 | SR | PCoffset9	ST SR, *PCoffset9*	M[PC + SEXT(PCoffset9)] ← SR
STI	1011 | SR | PCoffset9	STI SR, *PCoffset9*	M[M[PC + SEXT(PCoffset9)]] ← SR
STR	0111 | SR | BaseR | offset6	STR SR, BaseR, *offset6*	M[BaseR + SEXT(offset6)] ← SR