ECE190 Exam 2, Fall 2008

Thursday 30 October

Name:

- **Be sure that your exam booklet has 13 pages.**
- **The exam is meant 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 handwritten notes.**
- **Absolutely no interaction between students is allowed.**
- **Challenging problems are marked with ***.**
- **Show all of your work.**
- **Don't panic, and good luck!**

"...if rationality were the criterion for things being allowed to exist,
the world would be one gigantic field of soya beans!"
—from the play *Jumpers* by Tom Stoppard

Problem 1	20 points	_____
Problem 2	15 points	_____
Problem 3	25 points	_____
Problem 4	20 points	_____
Problem 5	20 points	_____
Total	100 points	_____

Problem 1 (20 points): Short Answers

Please answer concisely. If your answer requires more than a few words or a simple figure, it is probably wrong.

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

```
int mystery ()
{
    int x = 1;
    int y = 10;

    do {
        y = y + x;
        if (4 < x) {
            break;
        }
        x = x * 2;
    } while (20 > y);
    printf ("%d\n", y); /* A(i)  refers to this line. */
    return x;           /* A(ii) refers to this line. */
}
```

i) What number does the call to `printf` in the `mystery` function output to the display?

ii) What value does the `mystery` function return?

Part B (5 points): Consider the C code snippet below.

```
int a = 10;
int b = 5;
int c = 10;

c = (a++) + (--b) + c;
```

Write the values of the three variables after all of the assignments have completed.

a _____ b _____ c _____

Part C (5 points): Consider the following C program.

```
#include <stdio.h>

int glob = 42;

void fn ();

int main ()
{
    double mine = 3.8;
    printf ("glob=%d\n", glob);
    fn ();
    return 0;
}

void fn ()
{
    glob = mine;
    printf ("glob=%d\n", glob);
}
```

The code contains a single error. Explain the error.

Part D (5 points): Most functions require a specific number of arguments. A few, such as `scanf`, take a variable number of arguments. In the C declaration below, the ellipsis ("...") following the first argument indicates that a variable number of additional arguments can be passed.

```
void variable_args (int num_args, ...);
```

For the `variable_args` function, the `num_args` argument indicates the number of additional arguments. For example, the call `variable_args (2, 4, 5)` indicates that two additional arguments (4 and 5) are being passed.

When a compiler generates assembly code for a call to the `variable_args` function, should the `num_args` argument be pushed first or last (or does it not matter, if all compilers are required make the same choice)? Explain your answer.

Problem 2 (15 points): Assemblers and Assembly Language

This exam you are now taking was written by a program outputting the text to the console and redirecting that to a text file (think of comparisons you made for MP2). The first function we wrote was HEADER, which is shown below. **There are no typographical errors in the code.**

```
          .ORIG x3000
MAIN      LEA R1,TABLE
          JSR HEADER
          LEA R0,NICE
          HALT
NICE      .STRINGZ "Good Luck!"

HEADER    ST R7,TEMP
          LDR R0,R1,#0
          BRz DONE
          ADD R1,R1,#1
          PUTS
          LD R0,LINEFEED
          OUT
          BRnzp HEADER
DONE      LD R7,TEMP
          RET

TEMP      .BLKW #1
LINEFEED  .FILL x0A
TABLE     .FILL FIRST       ; address of first string
          .FILL SECOND      ; address of second string
          .FILL x0000
FIRST     .STRINGZ "ECE 190"
SECOND    .STRINGZ "Midterm 2"
          .END
```

Part A (8 points): Fill out the symbol table below as it would appear after the first pass of the LC-3 assembler. Some entries have been given. **Note that the symbol entries are NOT in order of appearance in the code.**

Symbol	Address
MAIN	
HEADER	
LINEFEED	
SECOND	
DONE	

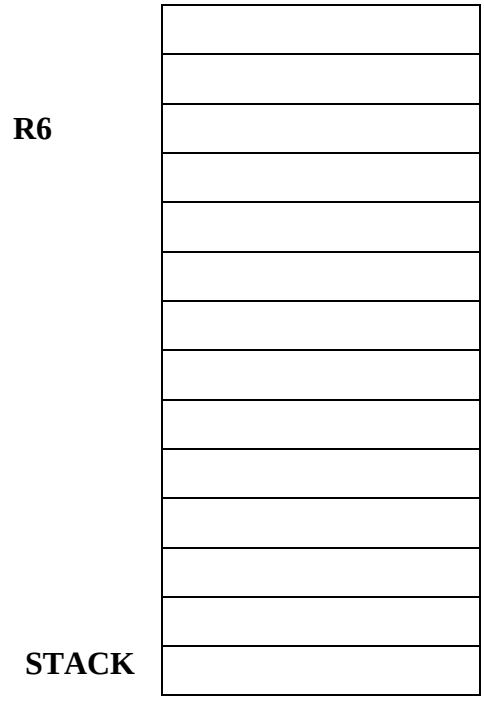
Symbol	Address
	x3004
	x301B
	x301E
TEMP	x3019

Part B (7 points): Show the output of the program and describe the program's behavior.

Problem 3 (25 points): Stacks and Subroutines

This problem pertains to the LC-3 assembly program shown on the left below.

```
        .ORIG x3000
        LEA R6,STACK
        LEA R1,STRING

        ADD R2,R1,#1

PART1   LDR R0,R1,#0
        BRz PART2
        ADD R6,R6,#-1
        STR R0,R6,#0
        ADD R1,R1,#2
        BRnzp PART1

PART2   LDR R0,R6,#0
        ADD R6,R6,#1
        OUT
        LDR R0,R2,#0
        OUT
        ADD R2,R2,#2
        LDR R0,R2,#-1
        BRnp PART2

        HALT
MESSAGE .STRINGZ "SOOGTALUTARINNC!"
        .BLKW #20
STACK
        .END
```



Part A (7 points): Assuming that an LC-3 processor has executed the program until it first reaches the instruction at PART2, fill in the contents of the stack diagram on the right above with one ASCII character per location. Any location not changed by the program must be left blank. Note the position of the STACK label, and **draw an arrow from R6** in the diagram to the memory location to which it currently points.

Part B (5 points): Write the output of the program.

Part C (6 points): The string provided in the program above has a length of 16 ASCII characters (not counting NUL). For what string lengths does this program do something predictable?

Problem 3, continued:

Part D (5 points): Add necessary instructions before and after the main portion of the program to turn it into an **assembly subroutine** (not a C subroutine—you do NOT need to create a stack frame/activation record).

R1 holds the address of a string of appropriate length when the subroutine is called, and R6 points to a stack with plenty of space left on top.

Your subroutine may not change the value of ANY register (except R7); R0, R1, R2, R3, R4, R5, and R6 **must be returned with their original values** at the end of your subroutine.

*******(2 points): For full credit on this problem, add only instructions in your code (no new directives such as .BLKW, no .FILL, no .STRINGZ, *etc.*).

```
STRINGSUB
; you may add code here









        ADD R2,R1,#1    ; this code was copied from previous page

PART1   LDR R0,R1,#0
        BRz PART2
        ADD R6,R6,#-1
        STR R0,R6,#0
        ADD R1,R1,#2
        BRnzp PART1

PART2   LDR R0,R6,#0
        ADD R6,R6,#1
        OUT
        LDR R0,R2,#0
        OUT
        ADD R2,R2,#2
        LDR R0,R2,#-1
        BRnp PART2

; you may also add code here
```

Problem 4 (20 points): I/O and Systematic Decomposition

While working at a summer internship, your boss informs you that the company has just bought several LC-3 machines. Your company is working on a top-secret defense contract, and you are charged with writing the password entry subroutine. As you might expect, the public version of the LC-3 operating system cannot be used on a secure machine, so **your code must interact directly with the devices**.

The display registers on your machines behave identically to a standard LC-3 platform: DSR[15] indicates that the display is ready to receive a new ASCII character; when the display is ready, writing an ASCII character to DDR[7:0] delivers it to the monitor.

The keyboard registers are similar, but have been extended with a fingerprint scanner. KBSR now returns one of the following bit patterns (only bits 15 and 3 are defined; others may hold any value):

15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	
0	x	x	x	x	x	x	x	x	x	x	x	0	x	x	x	no key, no fingerprint
0	x	x	x	x	x	x	x	x	x	x	x	1	x	x	x	impossible
1	x	x	x	x	x	x	x	x	x	x	x	0	x	x	x	keystroke—read KBDR
1	x	x	x	x	x	x	x	x	x	x	x	1	x	x	x	fingerprint match

When a key is ready, the 8-bit ASCII code can be read from KBDR[7:0]. No additional data is available when a fingerprint match is indicated by KBSR, thus KBDR holds no meaningful bits in that case.

For confidentiality reasons, every time the user types a password character, an asterisk ('*') must be echoed to the monitor in place of the character actually typed, while the character typed must be recorded in memory for later processing. After typing their password, a user swipes their finger over the scanner and, if authorized to use the machine, produces a fingerprint match response from KBSR.

Part A (10 points): Complete the systematic decomposition of the password entry routine on the next page by drawing missing arrows and filling in the boxes with brief labels in English and the symbolic names listed below.

KBSR KBDR DSR DDR

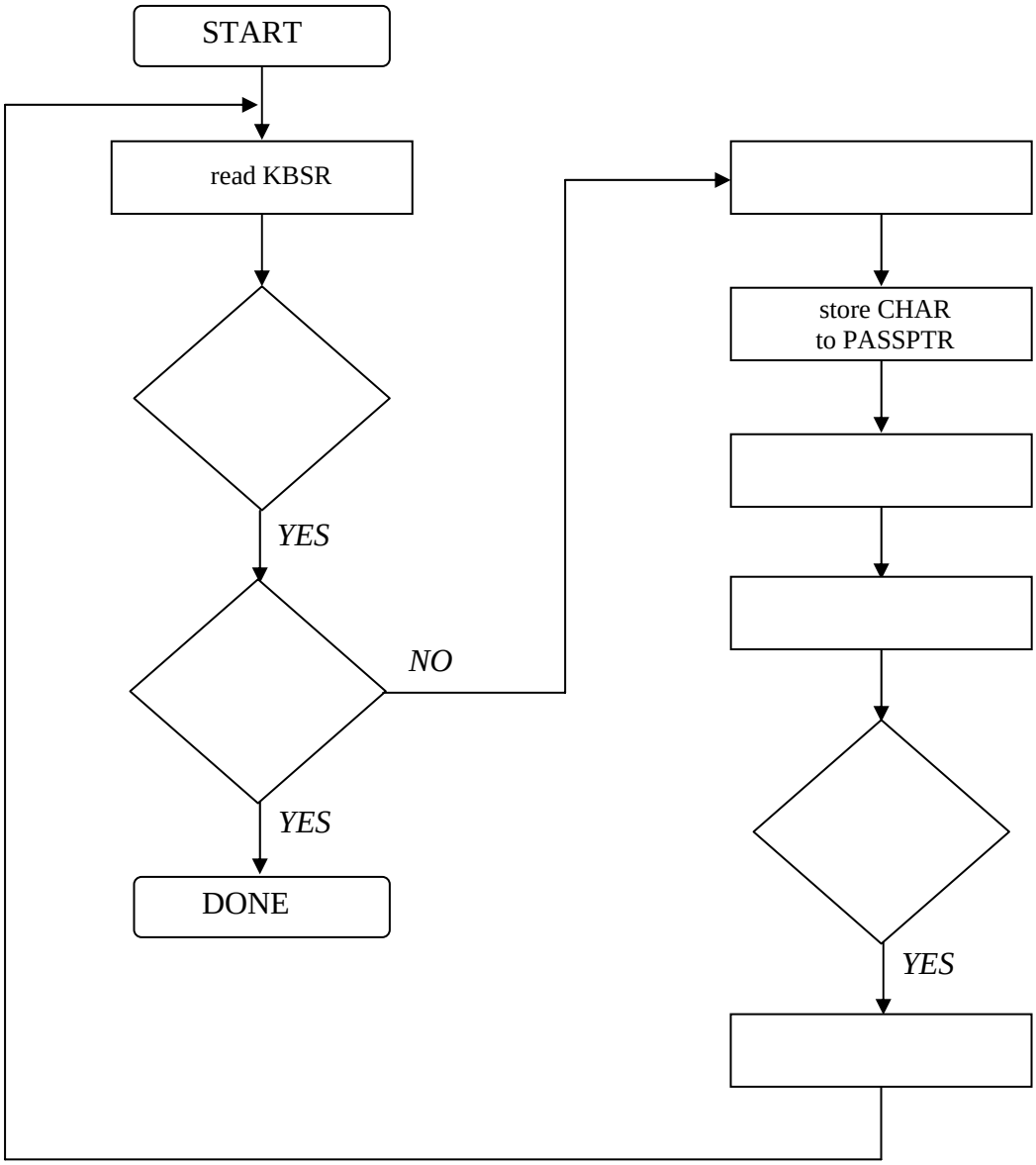
READY — the ready condition (as read from either KBSR or DSR)
FINGER — the fingerprint match flag (as read from KBSR)

CHAR — a character read from the keyboard
PASSPTR — the address of the first location in a block of memory into which you must store the password characters
ASTER — a register holding the ASCII character '*'

Note that labels in the form of LC-3 instructions earn no credit.

Problem 4, continued:

Part A answer diagram



Problem 4, continued:

Part B (10 points): Now write an LC-3 assembly subroutine to implement your password entry routine.

Some specifics for your subroutine:

- Device register addresses constants are provided for your use.
- R2 is an input to your subroutine and holds the address of the first location in the block of memory into which you must store the password characters (PASSPTR in **Part A**).
- R3 is loaded for you with an asterisk (ASTER in **Part A**).
- You may use any register that you like for the character read from KBDR (CHAR in **Part A**), reading the device registers, and so forth.
- You need not preserve any register values for this subroutine (all registers are caller-saved).

- **You may NOT add other constants or space for storage.**

```
PASS_ENTER
        LD R3,ASTERISK
        ; your code goes here
```

```
        RET
ASTERISK .FILL x002A
KBSR     .FILL xFE00
KBDR     .FILL xFE02
DSR      .FILL xFE04
DDR      .FILL xFE06
```

Problem 5 (20 points): C to LC-3

Consider the following C function.

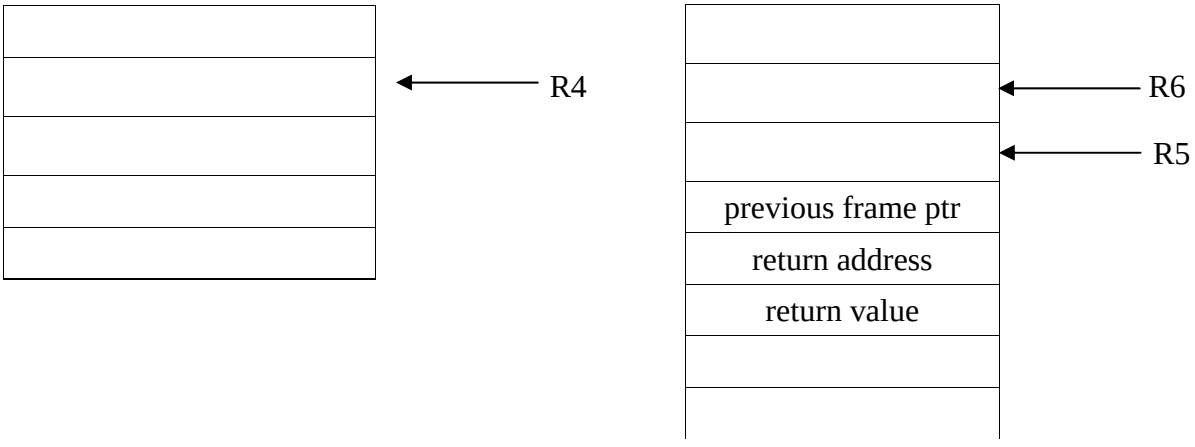
```
int a;

int a_function (int arg)
{
    int x;
    int y;

    x = arg - 1;
    y = (x << 3);
    if (10 > y) {
        y = a + 20;
    }
    return y;
}
```

Remember that in the LC-3 calling convention, C programs use R4 to point to the beginning (base) address of the global data section, R5 to point to the stack frame/activation record, and R6 to point to the top of the stack.

Part A (2 points): For the code above, fill the blank spaces in the partial symbol table shown below.

Identifier	Type	Offset	Scope
a	int	2	global
x	int	0	a_function
y	int	-1	a_function
arg	int		

Part B (6 points): Based on the symbol table, mark the locations of a, x, y and arg in the global data section diagram on the left and the stack frame diagram on the right below. Do not mark other boxes.



Part C (12 points): Fill in these five boxes with LC-3 assembly code for each of the statements listed at the top of the column. Write each section independently of the others: your code may not rely on registers loaded in a previous section (other than R4 and R5), and must update variables appropriately rather than leaving results in registers. **You may overwrite the values in R0 and R1 in your code for this problem.**

`x = arg - 1;`	`y = (x << 3);`

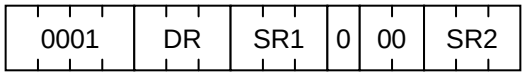
`if (10 > y) – branch if FALSE to NOPE`	`y = a + 20;`

`return y;`
`NOPE ; branch target for if test failure (above left)`

Use this page for scratchwork.

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

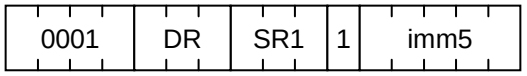
ADD



ADD DR, SR1, SR2

DR ← SR1 + SR2, Setcc

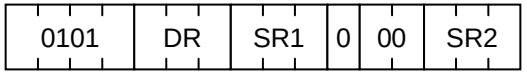
ADD



ADD DR, SR1, *imm5*

DR ← SR1 + SEXT(imm5), Setcc

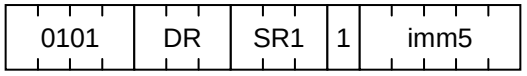
AND



AND DR, SR1, SR2

DR ← SR1 AND SR2, Setcc

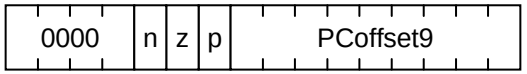
AND



AND DR, SR1, *imm5*

DR ← SR1 AND SEXT(imm5), Setcc

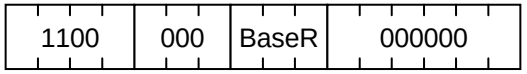
BR



BR{nzp} *PCoffset9*

((n AND N) OR (z AND Z) OR (p AND P)):
PC ← PC + SEXT(PCoffset9)

JMP



JMP BaseR

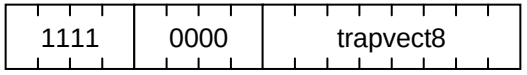
PC ← BaseR

JSR

JSR *PCoffset11*

R7 ← PC, PC ← PC + SEXT(PCoffset11)

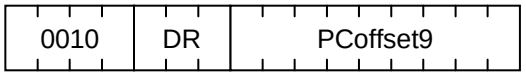
TRAP



TRAP *trapvect8*

R7 ← PC, PC ← M[ZEXT(trapvect8)]

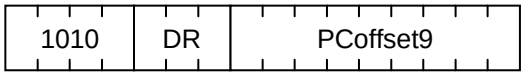
LD



LD DR, *PCoffset9*

DR ← M[PC + SEXT(PCoffset9)], Setcc

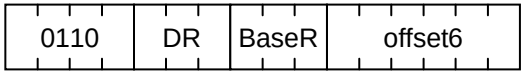
LDI



LDI DR, *PCoffset9*

DR ← M[M[PC + SEXT(PCoffset9)]], Setcc

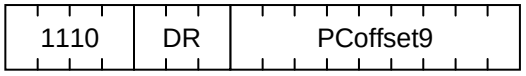
LDR



LDR DR, BaseR, *offset6*

DR ← M[BaseR + SEXT(offset6)], Setcc

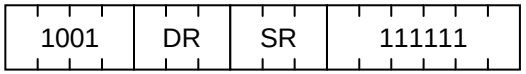
LEA



LEA DR, *PCoffset9*

DR ← PC + SEXT(PCoffset9), Setcc

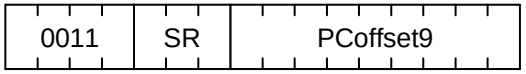
NOT



NOT DR, SR

DR ← NOT SR, Setcc

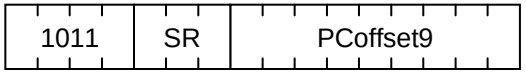
ST



ST SR, *PCoffset9*

M[PC + SEXT(PCoffset9)] ← SR

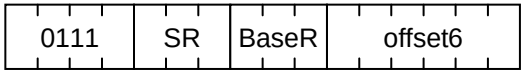
STI



STI SR, *PCoffset9*

M[M[PC + SEXT(PCoffset9)]] ← SR

STR



STR SR, BaseR, *offset6*

M[BaseR + SEXT(offset6)] ← SR