ECE190 Final Exam, Fall 2006

Tuesday 12 December

Name:

- **Be sure that your exam booklet has 14 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 THREE 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!**

"He was like a man who had served a term in prison or had been to Harvard College ..."
—C. McCullers in *The Heart is a Lonely Hunter*

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): In most systems, the heap and stack grow towards each other in memory. For example, data pushed on the stack use progressively smaller addresses, while data allocated from the heap use progressively larger addresses.

Describe one disadvantage of each of the following two alternatives:
1. having the stack and heap grow in the same direction, and

2. having the stack and heap grow away from each other.

Part B (5 points): Given a finite state machine with Q states, how many flip-flops are necessary to store the state of the machine? Write your answer as expression in terms of Q.

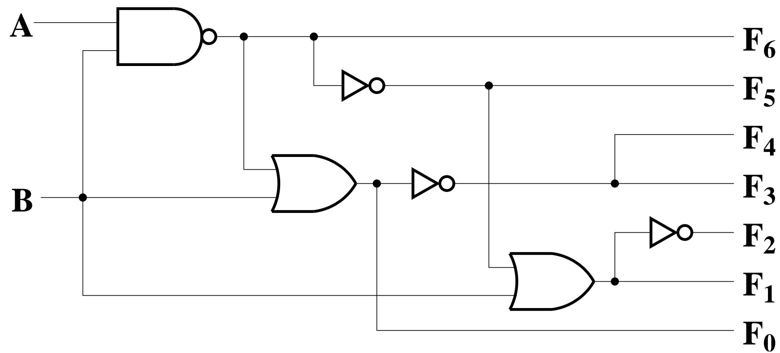
Problem 1, continued:

Part C (5 points): What is printed to the monitor when the function below is called with parameter `y=3`?

```
void func (int y)
{
    int x;

    x = y + 1;
    while ( (0 < x--) || 1 < (x += y--) ) {
        printf ("%d %d\n", y, x);
    }
}
```

Part D (5 points): Fill in the truth table below for the circuit shown.



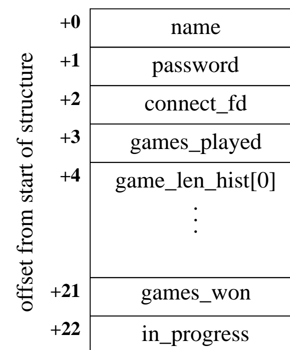
A	B	F_6	F_5	F_4	F_3	F_2	F_1	F_0
0	0							
0	1							
1	0							
1	1							

Problem 2 (20 points): Structures, Assembly, and I/O

The structure below is similar to the structure defined in class to describe a player in an online Risk server. The figure on the right below illustrates how this structure maps into LC-3 memory.

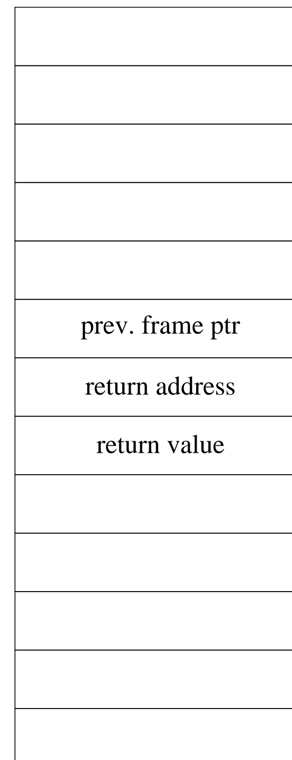
```
typedef struct player_t player_t;
struct player_t {
    char*   name;              /* pointer to dynamically-allocated name         */
    char*   password;          /* pointer to dynamically-allocated password     */
    int     connect_fd;        /* connection file descriptor (-1 if not online) */
    int     games_played;      /* number of games played                        */
    int     game_len_hist[17]; /* 0 to 8 hours in 1/2-hour blocks               */
    int     games_won;         /* number of games won                           */
    game_t* in_progress;       /* game being played (NULL if not playing)       */
};
```

```
int player_login (player_t* p, char* pswd, int desc)
{
    if (0 == strcmp (p->password, pswd)) {
        p->connect_fd = desc;
        return 0;
    }
    return -1;
}
```



Part A (4 points): The code shown above is called when a network connection specified as file descriptor `desc` attempts to associate itself with a player structure `p`. If the password provided matches the password stored in the player structure, the association is permitted, and the function returns 0. If they do not match, the function returns -1 without changing the player structure.

Using the diagram to the right, fill in the stack with names of parameters and local variables as they are located **during the execution** of the `player_login` function. You need only fill in names (e.g., "desc"), not values. Also, draw an arrow from R6 to the address to which it points. Leave any extra locations blank—you will lose points for writing in them.



Problem 2, continued:

(replicated for your convenience)

```
int player_login (player_t* p, char* pswd, int desc)
{
    if (0 == strcmp (p->password, pswd)) {
        p->connect_fd = desc;
        return 0;
    }
    return -1;
}
```

offset from start of structure

offset	field
+0	name
+1	password
+2	connect_fd
+3	games_played
+4	game_len_hist[0]
	⋮
+21	games_won
+22	in_progress

Part B (10 points): Using the diagram and your stack frame from **Part A**, translate the `if` statement's condition (shown in bold) from the `player_login` function into LC-3 assembly language. You may assume that R0 through R3 are available for your use and that the function `strcmp` is within reach of a JSR instruction. Remember that in the LC-3 calling convention, the return value is left on top of the stack when a function returns.

```
; write your code below
```

```
; code should fall through to here if passwords match

; translation of then block (NOT WRITTEN BY YOU!)

FAIL ; code should branch to this label if passwords do not match
```

Problem 2, continued:

(replicated for your convenience)

```
typedef struct player_t player_t; struct player_t {
    char*   name;              /* pointer to dynamically-allocated name          */
    char*   password;          /* pointer to dynamically-allocated password      */
    int     connect_fd;        /* connection file descriptor (-1 if not online) */
    int     games_played;      /* number of games played                         */
    int     game_len_hist[17]; /* 0 to 8 hours in 1/2-hour blocks               */
    int     games_won;         /* number of games won                            */
    game_t* in_progress;       /* game being played (NULL if not playing)        */
};
```

Part C (6 points): Write the function `save_player` below to write the contents of the player structure `p` to the file `f` as illustrated in the figure below. Note that the `connect_fd` and `in_progress` fields are temporary and are not stored into the file. You may assume that none of the file output functions fail.

name	**Napoleon**
password	**wat4rl00**
games_played	**137**
games_len_hist	**0 0 0 0 1 8 14 25 40 22 16 5 3 2 0 0 1**
games_won	**136**

File output function declarations are provided at the back of this test.

```
void save_player (player_t* p, FILE* f)
{

}
```

Problem 3 (20 points): Debugging

The two pieces of C code on this page have errors (other than syntax errors, of which there should be none). Given the explanations of the intended behavior, identify and explain one error for each function.

Part A (5 points):

```
/*
 * print the given string to stdout character by character,
 * followed by a line feed
 */
void
print_string (char* string)
{
    while ('\0' != *(string++)) {
        printf ("%c", *string);
    }
    printf ("\n");
}
```

Part B (5 points):

```
/* choose one of two strings to be printed */
void
string_select (int which)
{
    char str1[8] = "string1";
    char str2[8] = "string2";

    switch (which) {
        case 1:
            return str1;
        case 2:
            return str2;
        default:
            return str1;
    }
}


/* no errors in main -- included as a hint for string_select */
int
main ()
{
    char* s;

    s = string_select (1);
    printf ("string is %s\n", s);
    return 0;
}
```

Problem 3, continued:

A TA instructs each person in a class to pick a number P between -32767 and 32767 and to calculate $N = -P$ (both are thus representable using 16-bit 2's complement).

The students' numbers are placed into two arrays of 16-bit integers: pos[] for the P values and neg[] for the N values, and the following code is executed on the LC-3.

```
void
print_sum (int pos[], int neg[], int n_students)
{
    /* note that these are 16-bit integers */
    int sum_pos = 0;
    int sum_neg = 0;
    int sum;
    int i;

    for (i = 0; n_students > i; i++) {
        sum_pos = sum_pos + pos[i];
        sum_neg = sum_neg + neg[i];
    }

    printf ("%d\n", sum_pos + sum_neg);
}
```

The TA says that this code will print the value zero, since each student's P and N sum to zero.

Part C (5 points): Some students complain that the code shown won't work properly. Explain their concern.

Part D (5 points): The code does in fact print the value zero regardless of the students' choices of P. Explain why.

Name: ______________________________

Problem 4 (20 points): Pointer-based Data Structures and Recursion

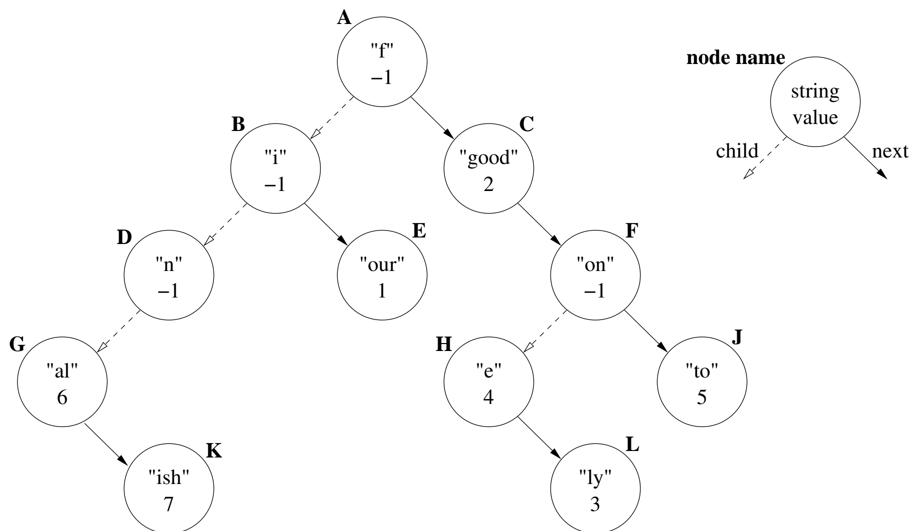
This problem makes use of an advanced, pointer-based data structure designed to efficiently store a set of string-value pairs. The structure is shown below, along with a function that prints all of the string-value pairs stored in the data structure. Below the code is an example of the data structure for use in the problem. The letters outside of the nodes are only for use in the problem and are not stored in the node structure. **All parts of the problem are on the next page; please write your answers there.**

```
/* string functions used
 * - strlen returns the number of characters in a string (not counting NUL)
 * - strcpy copies the second string argument into the first
 */

typedef struct node_t node_t;
struct node_t {
    char*   string;
    int     value;
    node_t* child;
    node_t* next;
};

void
print_nodes (node_t* node, char start[256])
{
    int len;

    if (-1 != node->value) {
        printf ("%s%s %d\n", start, node->string, node->value);
    }
    if (NULL != node->child) {
        len = strlen (start);
        strcpy (start + len, node->string);
        print_nodes (node->child, start);
        start[len] = '\0';
    }
    if (NULL != node->next) {
        print_nodes (node->next, start);
    }
}
```



Problem 4, continued:

Part A (8 points): Write the sequence of recursive calls made when `print_nodes` is called as `print_nodes (<node A>, "");` (the characters after a NUL in the `start` parameter don't matter).

call #	node	start
1	A	(empty string)
2		
3		
4		
5		
6		

call #	node	start
7		
8		
9		
10		
11		
12		

Part B (7 points): Write the output printed when the function is called as in **Part A**.

Part C (3 points): If this code is run on the LC-3, and 1,000 memory locations are available for the `print_nodes` stack frames (ignore other calls' frames), how many nested calls can be made? *Hint: a picture of the LC-3 linkage appears on page 4, if you need it.*

Part D*** (2 points): Let the value N be your answer to **Part C**. In terms of N, how many *total* recursive calls can be made?

Problem 5 (20 points): Dynamic Allocation and Stacks

In this problem, you will write functions to implement a stack to store integers using a dynamically allocated array. You must implement the stack using the three file-scope variables shown below and the dynamic resizing technique discussed in lecture (and reiterated in **Part C** on the next page). Note that your stack *does not need to grow backwards,* i.e.*, downward in memory.* Make your implementation simple.

```
/* stack variable declarations */
static int* stack = NULL;    /* array used for stack                     */
static int  size  = 0;       /* current array size for stack             */
static int* sfree = NULL;    /* pointer to FIRST UNUSED LOCATION on stack */
```

Allocation function declarations are provided at the back of this test.

Part A (6 points): Write the function `init_stack`, which takes a stack size as a parameter, creates a stack of that size, and updates the file-scope variables appropriately. Return 0 on success and -1 if an error occurs. You may assume that the stack variables have their initial values (shown in declarations above) when this routine is called.

```
int init_stack (int stack_size)
{

}
```

Part B (2 points): Write the function `clear_stack`, which releases any dynamic memory used in the stack and restores the file-scope variables to their initial values. You may assume that the stack exists (*i.e.*, `init_stack` was called) when this function is called.

```
void clear_stack()
{

}
```

Problem 5, continued:

Part C (8 points): Write the function `push`, which takes an integer parameter and pushes the integer onto the stack. Return 0 on success and -1 on failure. If the stack already has enough space for the new integer, add the integer to the stack and update the file-scope variables accordingly. If the stack is full, dynamically resize it to twice its current size, then push the integer and return success (0). If reallocation fails, return failure (-1). You may again assume that the stack exists when this function is called. *Hint: you will need to do arithmetic with pointers!*

```
int push (int item)
{

}
```

Part D (4 points): Write the function `pop`, which removes the topmost value from the stack and writes it into a memory location provided by the caller. Your function should return 0 on success, and -1 if the stack is empty. You may assume that `item_ptr` points to valid storage for an item and that the stack exists when this function is called.

```
int pop (int* item_ptr)
{

}
```

```
/* memory allocation function declarations from stdlib.h */
void* malloc  (size_t n_bytes);                    /* returns NULL on failure */
void* calloc  (size_t n_elts, size_t elt_size);    /* returns NULL on failure */
void* realloc (void* old_ptr, size_t n_bytes);     /* returns NULL on failure */
void  free    (void* old_ptr);

/* file output function declarations from stdio.h */
int    fprintf (FILE* stream, const char* format, ...);
int    fputc   (int c, FILE* stream);
int    fputs   (const char* s, FILE* stream);
size_t fwrite  (const void * ptr, size_t size, size_t nmemb, FILE* stream);
```

Use the rest of this page for scratch paper.

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

Instr	Encoding	Syntax	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