CIS 240 Fall 2019: Final

Please put all answers in the exam booklet and remember to number them clearly.

Question 1 {10 pts}

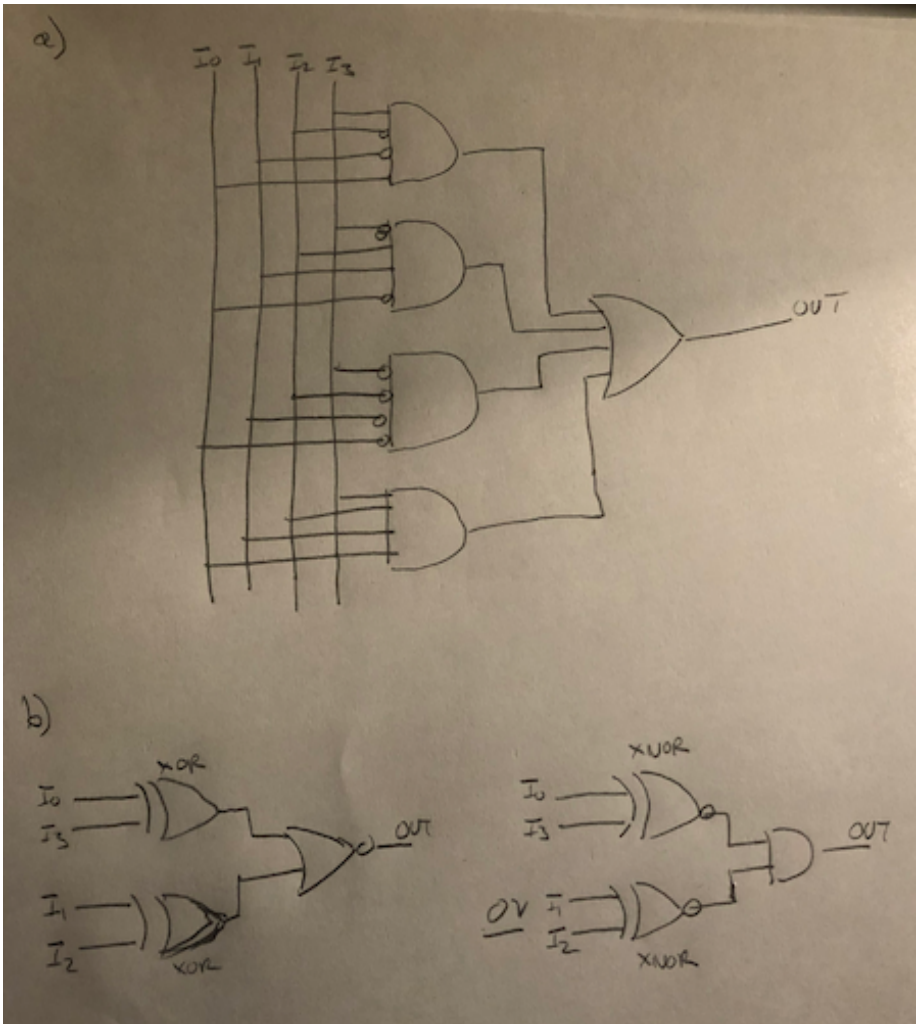
Part 1: {6 pts}
Your job is to design a PLA circuit that takes as input a 4 bit input I where I_0 is the LSB and I_3 is the MSB and returns a high output if and only if the number is a palindrome, which means that you get the same sequence of bits if you read it forward or backwards. As an example, 1001 and 0110 are 4-bit palindromes while 1100 is not. Please make sure to label your input signals clearly.

Part 2: {4 pts}
For this part you are going to redesign the circuit that you designed in part 1. This time it does not have to be a PLA but you are only allowed to use 2 input logic gates and you cannot have more than 5 of them. You can use any 2 input gates you wish, AND, OR, NAND, NOR, XOR, XNOR. More points will be given for designs that use fewer gates.

Answer:

There are only 4 4-bit palindromes. 0000, 1111, 0110, 1001. To see this note that the last 2 bits must mirror the first two bits. This leads to a straightforward PLA implementation with 4 and gates – one for each pattern.

For the second part we observe that we can use the XOR gate to compare two bits. The XOR output is high if the two inputs differ and the XNOR output is high if the two inputs are the same. So, there are two optimal implementations – one uses two XOR gates and a NOR gate and the other uses two XNOR gates and an AND gate. Both solutions use 3 gates which is the minimum possible.



Question 2 {10 pts}

For your CIS 240 homework you write a TRAP routine that is designed to place a single character on the ASCII output console. It is called TRAP_PUTC and it works perfectly. You are then asked to write a second routine, TRAP_PUT_STRING that puts a sequence of characters on the ASCII console. Not wanting to waste effort, you figure that you can just call your TRAP_PUTC trap repeatedly as the following pseudo code shows.

```
TRAP_PUT_STRING
        WHILE (current character is not null – i.e. not at end of string)
                Send current character to screen by using TRAP
                        instruction to call TRAP_PUTC
                Advance to next character
```

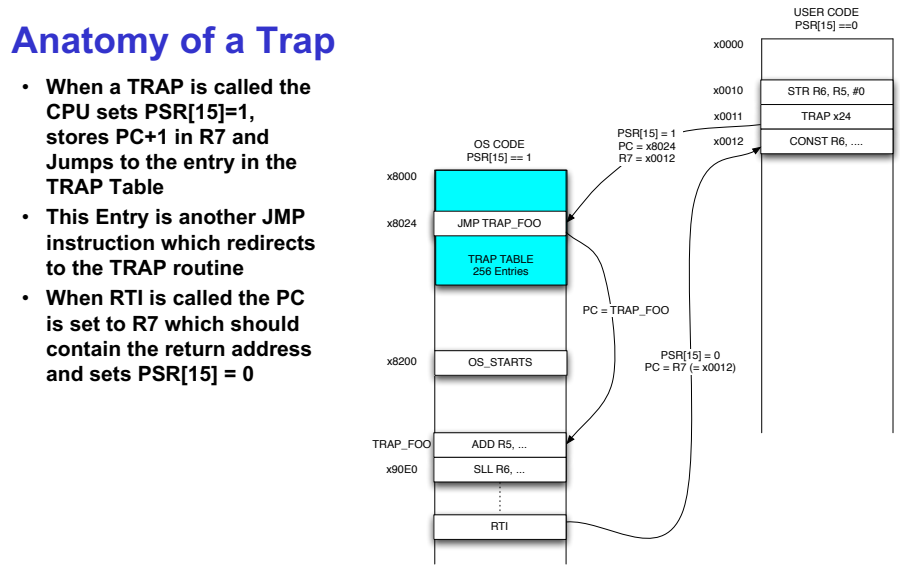
Would this idea of using the trap instruction to call TRAP_PUTC from inside TRAP_PUT_STRING work? Carefully explain why or why not.

Answer:

This would actually work it TRAP_PUTC was a regular subroutine. However, since it is a trap routine it ends with a call to RTI which returns control to the calling context but also reduces privilege to user level. The problem then is when TRAP_PUT_STRING called TRAP_PUTC it would dutifully place the first character on the output and then return to the TRAP_PUT_STRING routine with privilege set to USER level. Since TRAP_PUT_STRING is in the operating system part of memory this would cause a problem since we would be trying to execute OS code without OS privilege which would cause an exception.

Question 3 {10 pts}

Your cousin, Crazy Eddie, is looking over your slides for CIS 240 particularly the ones on how TRAPs are handled. When he comes across the slide shown below, he snorts and says: "This is way too complicated! Why don't we just place the trap routine we want to call, TRAP_FOO, at the address that TRAP is going to jump to, x8024, and avoid the intermediate JMP instruction. That should work just fine."



Question 1: Is Crazy Eddie right, would the trap routine work correctly if it were moved up to start at x8024 avoiding the JMP instruction.

Question 2: Would this change create other potential problems? Explain your answer.

Answer:

Crazy Eddie is right in that the TRAP would work correctly if it were placed at x8024 instead of further down in memory.

The problem is that if we do that the user could jump into arbitrary parts of the TRAP_FOO routine by calling trap with values greater that x24. For example TRAP x26 would start execution at the second instruction of the TRAP_FOO which isn't desirable. Since the user can call TRAP with any value this would create all kinds of potential problems and security vulnerabilities.

Question 4 {10 pts}

Consider the following C program

```
#include <stdlib.h>

/* allocate a buffer to store 100 ints */
void allocate_buffer (int *ptr) {
  ptr = malloc(100*sizeof(int));
}

int main () {
  int i, *buf;

  allocate_buffer (buf);
  for(i=0; i < 100; ++i) buf[i] = i;
}
```

Does the program compile without issue? Could the code cause a segmentation violation or not? Explain your answer briefly.

Answer:

The code will in fact compile since there are no syntactic errors.

The problem, however, is that the code doesn't actually do what it looks like it's doing. It looks like the allocate_buffer function allocates memory that we can store into and it does in fact call malloc. The issue is that C is a call by value language which means that when we call allocate_buffer we pass in the current value of the pointer buf but the value of buf is not changed by the function. This means that when we get to the for loop buf had the same value it had before and since it started off uninitialized, we have no idea what it points to. When we start writing into locations referenced to buf we could absolutely get a segmentation violation. The storage allocated by malloc is lost in this implementation, a memory leak.

If we wanted to fix this code one way to do it would be as follows.

```
void allocate_buffer (int **ptr) {
  *ptr = malloc(100*sizeof(int));
}
```

Then call the routine as follows

```
allocate_buffer(&buf);
```

Here we would be passing the ADDRESS of the variable buf to the allocate_buffer routine as opposed to the value of the variable. We can then store the value we want into this address.

An even better idea would be to have allocate_buffer return the address to the memory it allocated.

```
int * allocate_buffer() {
  return malloc(100*sizeof(int));
}
```

Then you could simply assign buf this value in the main routine.

```
buf = allocate_buffer();
```

Question 5 {10 pts}

Your cousin, Crazy Eddie, is really annoyed by the fact that he needs to FALIGN all of his subroutines in LC4 assembly. He thinks that alignment wastes space and that he should be able to start subroutines on any address he pleases. He thinks about it for a bit and then brightens and replaces this line in his assembly code.

```
JSR my_subroutine
```

With this sequence.

```
LEA R0, my_subroutine
JSRR R0
```

Will Crazy Eddie's code work even when the subroutine being called starts on an address that isn't a multiple of 16? What are the performance consequences of this change?

Answer:

Eddie is correct this will work even if my_subroutine does not start on an aligned address. The only issue is that this implementation involves 3 instructions – the LEA expands to a CONST, HICONST pair then the JSRR instruction so we are using 3 instructions to call a subroutine instead of a single JSR instruction.

Question 6 {10 pts}

Consider the following portion of a C library which provides two routines one for adding an element to a doubly linked list and another for deleting an element. A few of the lines have been blanked out. Your job is to tell us what each of the missing lines should be to correctly complete the code.

```
#include <stdlib.h>

typedef struct list_elt_tag {
  int number;
  // Pointers to the previous and next elements in the list
  struct list_elt_tag *prev, *next;
} list_elt;


/*
 * Creates a new list element and pushes it on the front of the list
 * returns a pointer to the newly created element.
 */

list_elt *push (list_elt *first_elt, int number)
{
  list_elt *elt;

  // Allocate a new list element with malloc
  elt = malloc (sizeof(*elt));

  // If malloc fails end the program
  if (elt == NULL) {
    exit (1);
  }

  elt->number = number;

  MISSING_LINE_1;   elt->prev = NULL;

  elt->next = first_elt;

  if (first_elt != NULL)
    first_elt->prev = elt;

  // return the pointer to the new list_elt
  return elt;
}
```

```
/*
 * delete: Deletes an element from the list returns a pointer to the new
 * first element of the list which may just be the old first element.
 */

list_elt *delete (list_elt *first_elt, list_elt *elt)
{
  list_elt *prev, *next;

  if (elt == NULL || first_elt == NULL) {
    return first_elt;
  }

  MISSING_LINE_2; prev = elt->prev;

  next = elt->next;

  /* First we fix the pointers of the next and previous elements */
  if (prev) {
    prev->next = elt->next;
  }

  if (next) {
    MISSING_LINE_3;  next->prev = elt->prev; or just prev
  }

  MISSING_LINE_4; free(elt)

  // Check if elt was the first element in the list
  if (elt == first_elt)
     MISSING_LINE_5; return next;
  else
    return first_elt;
}
```

Question 7 (10 pts)

The following piece of C code was compiled with the lcc compiler.

```
void strcpy (char *src, char *dest, int n) {
  int i;
  for (i=0; i < n; ++i) {
    if (src[i]) {
      dest[i] = src[i];
    } else {
      break;
    }
  }
}
```

The resulting LC4 assembly code fragment is shown below. Several of the assembly instructions have been blacked out. Your job is to figure out what those assembly instructions must have been.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;strcpy;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                .CODE
                .FALIGN
strcpy
        ;; prologue
        STR R7, R6, #-2         ;; save return address
        STR R5, R6, #-3         ;; save base pointer
        ADD R6, R6, #-3
        ADD R5, R6, #0
        ADD R6, R6, #-1         ;; allocate stack space for local variables
        ;; function body
        MISSING_INSN_1 -- CONST R7, 0
        STR R7, R5, #-1
        JMP L5_final2019
L2_final2019
        LDR R7, R5, #-1
        LDR R3, R5, #3
        ADD R7, R7, R3
        LDR R7, R7, #0
        CONST R3, #0
        CMP R7, R3
        MISSING_INSN_2 – BRz L4_final_2019 or L1_final_2019 these
        LDR R7, R5, #-1
        LDR R3, R5, #4
        ADD R3, R7, R3
        LDR R2, R5, #3
        ADD R7, R7, R2
```

```
        LDR R7, R7, #0
        MISSING_INSN_3 – STR R7, R3, 0
L7_final2019
L3_final2019
        LDR R7, R5, #-1
        MISSING_INSN_4 – ADD R7, R7, #1
        STR R7, R5, #-1
L5_final2019
        LDR R7, R5, #-1
        LDR R3, R5, #5
        CMP R7, R3
        BRn L2_final2019
L4_final2019
L1_final2019
        ;; epilogue
        ADD R6, R5, #0      ;; pop locals off stack
        ADD R6, R6, #3      ;; free space for return address, base pointer, and return
value
        STR R7, R6, #-1     ;; store return value
        LDR R5, R6, #-3     ;; restore base pointer
        MISSING_INSN_5 – LDR R7, R6, #-2
        RET
```

Question 8 (10 pts)

For the final assignment you were asked to implement a compiler that converted programs written in the stack-based J language into assembly instructions. We want to add the following two new commands to the J language so that we can write programs that access a fixed array in global memory:

global_write : Pops the first value off the stack and uses that as an index into the global array. Then pops the next value off of the stack and stores this value into the indicated location in the array. Egs 7 2 global_write would store the value 7 into the location global_array[2]. The entries 7 and 2 would be removed from the stack.

global_read : Pops the first element off of the stack and uses this as an index into the global array. It stores the value at that location in the global array at the top of the stack. Egs. 3 global_read – would place the value stored in global_array[3] at the top of the stack, the 3 would be removed.

For each of these two J commands indicate the sequence of assembly instructions that your revised J compiler would emit when it encountered them in a program. You can assume that there is a label in the assembly code entitled “global_array” that marks the location where the global array will be stored in memory. Following C convention, the array indices start at 0 and you can assume that the at run time the array index values will be legal. Please use comments to indicate your thinking.

Answer:

global_write:

To execute this we have to do the following:

1) Load the top element on the stack into a register – this is the array index
2) Compute the address we want to store into
3) Load the second element on the stack into a register – value to write
4) Store the value in the global array
5) Pop 2 entries from the stack

Here is one implementation

```
LEA R0, global_array    ; load the address of global array
LDR R1, R6, #0          ; load the index
ADD R0, R0, R1          ; compute the address to store to
LDR R2, R6, #1          ; load the value
STR R2, R0, #0          ; store the value in the global array
ADD R6, R6, #2          ; pop 2 values from the stack
```

global_read:

To execute this we have to do the following:

1) Load the top element on the stack into a register – this is the array index
2) Compute the address we want to store into
3) Get the value from the global array
4) Load the value onto the stack replacing the current top element

Here is one implementation

```
LEA R0, global_array    ; load the address of global array
LDR R1, R6, #0          ; load the index
ADD R0, R0, R1          ; compute the address to load from
LDR R2, R0, #0          ; load the value
STR R2, R6, #0          ; store the value on top of the stack
```