UNIVERSITY OF TORONTO

FACULTY OF APPLIED SCIENCE AND ENGINEERING
Final Exam, Dec 21, 2018
DURATION: 2 hour 30 min
Second Year – Engineering Science
ECE253 – Computer Organization

Calculator Type: 4
Exam Type: D

Examiners – T. Czajkowski and T. Kosteski

Instructions:

1. Write your name and student number.
2. **Do not remove any pages from this examination booklet.**
3. Answer all questions and justify all your work for full marks.
4. The grade for each question is given in the square brackets [ ].
5. Aid Sheets are provided at the end of this booklet. **DO NOT REMOVE.**

Last Name: _____

First Name: _____

Student #: _____

Question	Grade
Q1	
Q2	
Q3	
Q4	
Q5	
Q6	
Q7	
Q8	
Total (83)	

Question 1 [4 marks]: Simply the following expression:

$$A + \overline{A}B + \overline{B}C + \overline{C}D + \overline{D}E + EF$$

Question 2 [10 marks]: Design a minimum size combinational logic circuit that converts input data into a Binary Coded Decimal (BCD) representation. The table below lists the decimal number from 0 to 9, each corresponding BCD, and the corresponding input data representation.

Decimal Value	BCD ($f_3f_2f_1f_0$) Representation	Input Data ($X_3X_2X_1X_0$) Representation
0	0000	0000
1	0001	0011
2	0010	0101
3	0011	0110
4	0100	1010
5	0101	1011
6	0110	1100
7	0111	0111
8	1000	1000
9	1001	1001

Additional space on the next page

Additional space for Question 2

Question 3 [5 marks]: Given the expression, $\overline{E}$, below, design a circuit using only NAND gates that implements, **E** (i.e. the inverted form of $\overline{E}$). The only inputs are w, x, y and z.

$$\overline{E} = (\overline{x} + \overline{y})(\overline{w} + x)(x\overline{z})$$

Question 4 [10 marks]: For the truth table below, design a minimum-size combinational logic circuit using only NOR gates. The inputs are A, B, C, D and the output is Y.

A	B	C	D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

Additional space on the next page

Additional space for Question 4.

Question 5 [16 marks]: Design a finite state machine (FSM) with two inputs (x and y) with an output z, which is asserted every time x and y change state to opposing values at the same time. A sample output looks as follows:

x 0 0 0 1 1 1 0 1 0 1 0 0

y 0 1 1 0 1 1 0 1 1 0 1 0

z 0 0 0 0 1 0 0 0 0 1 1 0

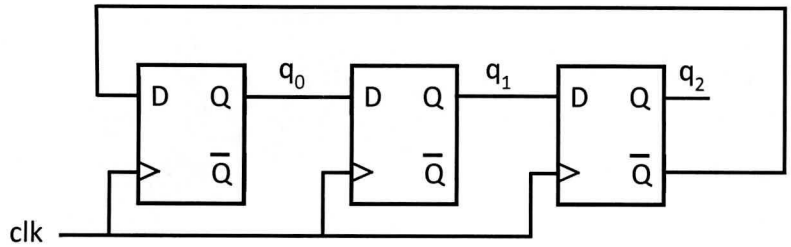
a) [4 marks] Show a state table for this FSM.

b) [4 marks] Show a state-assigned table for this FSM, using as few bits possible.

(Continued on next page)

c) **[8 marks] Derive minimum cost Sum-of-Products logic expressions for next state and output signals.**

Question 6 [3 marks]: The synchronous circuit below starts with $q_0,q_1,q_2 = 0,0,0$. After four clock pulses, determine the new values for q_0, q_1, and q_2. The flip-flops have both Q and $\bar{Q}$ outputs.



Question 7 [20 marks]: We want to implement a simple calculator using ARM assembly. The calculator only performs addition and subtraction, but what makes it more interesting is that it permits the use of parentheses that must be handled correctly.

The input the calculator needs to process starts at label INPUT and consists of a series of 32-bit data elements. Each data element comprises a 3-bit code as its most significant bits, and the remainder is an unsigned binary number. The 3-bit codes mean:

000 – left parenthesis

001 – right parenthesis

010 – add symbol

011 – subtract symbol

100 – value to be operated on.

111 – end of input

You are to write the program that implements the calculator. It **must** contain a recursive subroutine called **PROCESS**, which takes as input the position in the input it should start processing data. The position should point to the element right after a left parenthesis and this method should provide a correct result between this parenthesis and a corresponding right parenthesis. The result is to be returned in R1 and the position of the corresponding right parenthesis will be returned in R0. You may assume the input is properly formatted and checking input validity is not needed.

Additional Space on the next page

Additional Space for Question 7

Question 8 [15 marks]: Suppose that you have a computer system, very similar to the one used in the labs for this course. In this system, there is a new component connected with an ID of 80 (in decimal) to the GIC. This component is a message receiver with the following registers:

Address	Meaning
0xFF2ECC00	Data
0xFF2ECC04	Count
0xFF2ECC08	Threshold
0xFF2ECC0C	Status

The data register holds the information for a given message, if and only if count > 0. Count specifies the number of stored messages. Threshold defines how many messages must be stored to trigger an interrupt, and status only has one bit (bit 0) which is set to 1 when this component has triggered an interrupt.

To clear this interrupt you must reduce the count below the threshold, which can only be done by reading the data register. The reading of the data register will reduce the count by 1 (though it is possible another message arrives as you read the current message). Once you have read enough messages to reduce count below threshold, this will then trigger the status register to lower bit 0 and the interrupt will be cleared.

You are to write an Interrupt Service routine called MY_ISR to handle the interrupts generated by this component. You can assume that the exception vector table has been appropriately set and the only task at hand is that of writing MY_ISR. You will, however, need to handle the GIC as well as you did in lab 10. For the reference, the relevant registers of the GIC are:

Address	Register Name
0xFFFEC100	ICCICR
0xFFFEC104	ICCPMR
0xFFFEC10C	ICCIAR
0xFFFEC110	ICCEOIR

Additional Space on the next page

Additional Space for Question 8

AID SHEET

No.	Equation	No.	Equation	Description
1a	$\overline{0} = 1$	1b	$\overline{1} = 0$	
2a	$x + 0 = x$	2b	$x \bullet 1 = x$	
3a	$x + 1 = 1$	3b	$x \bullet 0 = 0$	
4a	$x + x = x$	4b	$x\,x = x$	Idempotent law
5a	$x + \overline{x} = 1$	5b	$x\,\overline{x} = 0$	
6	$\overline{(\overline{x})} = x$			Involution law
7a	$x + y = x$ $y + x$	7b	$x\,y = y\,x$	Commutative law
8a	$x + x\,y = x$	8b	$x\,(x + y) = x$	Absorption law
9a	$x + \overline{x}\,y = x + y$	9b	$x\,(\overline{x} + y) = x\,y$	
10a	$\overline{(x + y)} = \overline{x}\ \overline{y}$	10b	$\overline{(x\,y)} = \overline{x} + \overline{y}$	DeMorgan's law
11a	$(x + y) + z = x + (y + z) = x + y + z$	11b	$(x\,y)\,z = x\,(y\,z) = x\,y\,z$	Associative law
12a	$x + y\,z = (x + y)\,(x + z)$	12b	$x\,(y + z) = x\,y + x\,z$	Distributive law

Table1. Boolean Algebra Theorems

Verilog® Quick Reference Card

1. Module

```
module module_name (list of ports);
    input / output / inout declarations
    net / reg declarations
    integer declarations
    parameter declarations

    gate / switch instatnces
    hierarchical instances
    parallel statements
endmodule
```

2. Parallel Statements

Following statements start executing simultaneously inside module

```
initial begin
    {sequential statements}
end
always begin
    {sequential statements}
end
assign wire_name = [expression]};
```

3. Basic Data Types

a. Nets

e.g. **wire, wand, tri, wor**

- Continuously driven
- Gets new value when driver changes
- LHS of continuous assignment

```
tri [15:0] data;
        // unconditional
assign data[15:0] = data_in;
        // conditional
assign data[15:0] = enable ? data_in : 16'bz;
```

b. Registers

e.g. **reg**

- Represents storage
- Always stores last assigned value
- LHS of an assignment in procedural block.

```
reg signal;
@(posedge clock) signal = 1'b1;
        // positive edge
@(reset) signal = 1'b0;  // event (both edges)
```

-
```
if (reset == 0) begin
        data = 8'b00;
end
```
-
```
case (operator)
        2'd0 : z = x + y;
        2'd1 : z = x – y;
        2'd2 : z = x * y;
        default : $display ("Invalid Operation");
endcase
```
-
```
initial begin        // 50 MHz clock
        clock = 0;
        forever #10 clock = ~clock;
end                  // precission 1 ns
```
- `repeat (2) @(posedge clk) data;`
-
```
bus <= repeat (5) @ (posedge clk) data
        // evaluate data when the assignment is
        // encountered and assign to bus after 5
        // clocks.
```
-
```
repeat (flag) begin // looping
        .... action ....
```
-
```
while (i < 10) begin
        .... action ....
end
```
-
```
for (i = 0; i < 9; i = i + 1) begin
        .... action ....
end
```
- `wait (!oe) #5 data = d_in;`
- `@(negedge clock) q = d;`
-
```
begin                // finishes at time #25
        #10 x = y;
        #15 a = b;
end
```

- **The @(*) token adds to the sensitivity list all nets and variables that are read by the statements in the always block.**

7. Declarations

{}, {{}}	concatenation
+ - * /	arithmetic
%	modulus
> >= < <=	relational
!	logical negation
&&	logical and
||	logical or
==	logical equality
!=	logical inequality
===	case equality
!==	case inequality
~	bit-wise negation
&	bit-wise and
|	bit-wise inclusive or
^	bit-wise exclusive or
^~ or ~^	bit-wise equivalence
&	reduction and
~&	reduction nand
|	reduction or
~|	reduction nor
^	reduction xor
~^ or ^~	reduction xnor
<<	left shift
>>	right shift
? :	condition
or	event or

Mnemonic	Instruction	Action
ADC	Add with carry	Rd := Rn + Op2 + Carry
ADD	Add	Rd := Rn + Op2
AND	AND	Rd := Rn AND Op2
B	Branch	R15 := address
BIC	Bit Clear	Rd := Rn AND NOT Op2
BL	Branch with Link	R14 := R15, R15 := address
BX	Branch and Exchange	R15 := Rn, T bit := Rn[0]
CDP	Coprocesor Data Processing	(Coprocessor-specific)
CMN	Compare Negative	CPSR flags := Rn + Op2
CMP	Compare	CPSR flags := Rn - Op2
EOR	Exclusive OR	Rd := (Rn AND NOT Op2) OR (op2 AND NOT Rn)
LDC	Load coprocessor from memory	Coprocessor load
LDM	Load multiple registers	Stack manipulation (Pop)
LDR	Load register from memory	Rd := (address)
MCR	Move CPU register to coprocessor register	cRn := rRn {<op>cRm}
MLA	Multiply Accumulate	Rd := (Rm * Rs) + Rn
MOV	Move register or constant	Rd : = Op2
MRC	Move from coprocessor register to CPU register	Rn := cRn {<op>cRm}
MRS	Move PSR status/flags to register	Rn := PSR
MSR	Move register to PSR status/flags	PSR := Rm
MUL	Multiply	Rd := Rm * Rs
MVN	Move negative register	Rd := 0xFFFFFFFF EOR Op2
ORR	OR	Rd := Rn OR Op2
RSB	Reverse Subtract	Rd := Op2 - Rn
RSC	Reverse Subtract with Carry	Rd := Op2 - Rn - 1 + Carry

Mnemonic	Instruction	Action
SBC	Subtract with Carry	Rd := Rn - Op2 - 1 + Carry
STC	Store coprocessor register to memory	address := CRn
STM	Store Multiple	Stack manipulation (Push)
STR	Store register to memory	<address> := Rd
SUB	Subtract	Rd := Rn - Op2
SWI	Software Interrupt	OS call
SWP	Swap register with memory	Rd := [Rn], [Rn] := Rm
TEQ	Test bitwise equality	CPSR flags := Rn EOR Op2
TST	Test bits	CPSR flags := Rn AND Op2