High Performance Embedded Systems

EEE4120F

FINAL EXAM

PART C – Short Answers

10 July 2020

2 hours

Examination Prepared by:
Simon Winberg

Last Modified: 10-Jul-2020

REGULATIONS

This is an open-book exam. Scan through the questions quickly before starting so that you can plan your strategy for answering the questions. NB: you need to sign and abide by the Honor Pledge before beginning this assessment. TurnItIn will be used. Make sure that you **put** your **student name and student number,** the course code **EEE4084F** and a title **Final Exam** on your answer submission.

DO NOT TURN OVER UNTIL YOU ARE TOLD TO

Exam Structure

Marked out of 150 marks

Part A	Part B	Part C	Appendices
Multiple Choice [50 marks] (online: see Vula Tests & Quizzes)	GA2 Part B *(Long Answers)* [50 marks] (should have completed already)	Short Answers [50 marks] Approx. 1h pg 2	A: Verilog cheatsheet pg 8

RULES

- Make sure your name and student is clearly indicated in the answers that you submit.
- Write the question numbers of corresponding to you're answers clearly for each answer.
- You may choose to answer the questions using pen/pencil on paper and then scan/photograph your answers and upload them; in such a case it is preferred that you combine your scanned answers together into a single PDF as this will reduce the chance of answers not getting marked.
- Make to cross out material you do not want marked. Your first attempt at any question will be marked if two answers for the same question are found.
- Answer all questions, and note that the time for each question relates to the marks allocated.

Part C: Short Answers [50 marks]

The following design concept relates to both Question 1 and 2. Read the description of the design given here and then proceed to answering the questions that follow on the next page.

Note you are *not going to be asked to design the whole system* in Verilog or other program, but rather this is used as a design concept used to assess you response to design approaches and considerations you may recommend in the questions that follow.

Dual Sensor Monitor with single Trigger Design Concept

As indicated, there are two 8-bit ADCs providing inputs to this system (assume they are the same type, and their interface is explained below). You can choose how many modules to use (e.g. a top-level module and sub-modules), or you can code everything in a single module. The trigger is simply a single bit output. If needed, there is a clock, *clk*, that runs at 10MHz. Assume that *Limit* is an internal register for this system that is set up in the HDL code to be an 8-bit value, e.g. set to 10 as indicated in the diagram.

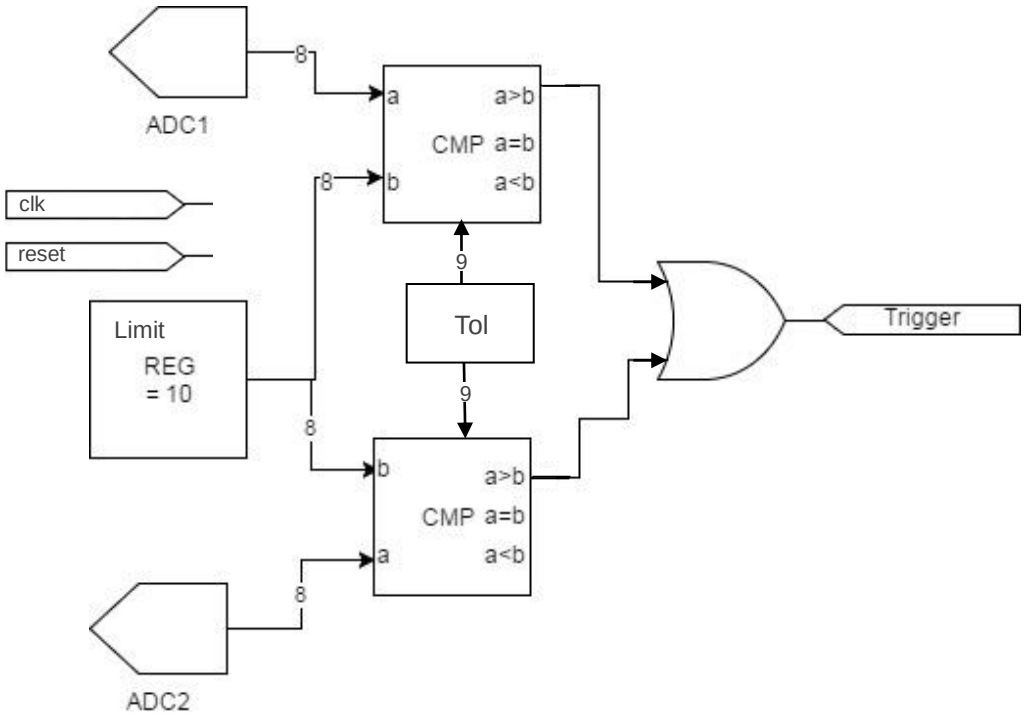

Figure 1 Dual sensor monitor with single trigger output.

Assume that *Tol* is also an internal register than is set up in code. It is a 9-bit value use to set a millisecond tolerance value (i.e. from 0 to 2^9-1) that specifies how quickly the trigger needs to respond. The CMP blocks are not normal comparators, but rather are delayed comparators which will only update its output every *Tol* milliseconds. The CMP must be sent a reset at system startup. The CMP block has two 8-bit inputs *a* and *b*, and three outputs, *a>b*, *a=b* and *a<b*. You need to decide suitable names as needed, and any port names you may decide to use for these inputs and outputs. Only the *a>b* output of the CMP at the top of the figure links to the OR gate, and similarly only the *a>b* for the second CMP links to the OR gate.

The interface for each ADC is as follows: The ADC must be sent a reset at system startup. The ADC has a *re* line that needs to be sent a positive edge to ask it to generate an output – you can assume it responds to a *re* positive edge instantly and the data lines will become valid; the data lines will remain valid until another *re* edge is sent. The ADC data output is an 8-bit bus, which you could call *adc_data*. Each ADC operates at a maximum speed of 1MHz, so sending it positive edges on *re* at a rate surpassing 1MHz will causing the operation of the device to become stochastic, the data bits taking on spurious values at unpredictable times until a reset is performed. During startup, the reset line is set high and will remain so for at least a few milliseconds before going low at which time the system should be operating. The *trigger* output must be set low initially when reset goes high, the *trigger* needs to remain low at least until the *Tol* duration has elapsed.

Please see questions on next page…

Question 1 [40 marks]

Please answer these sub-questions that relate to the design on the previous page …

(a) Based on the given design, would you say that the CMP components of this design are synchronous or asynchronous? Briefly motivate your answer. [4 marks]

(b) Consider the CMP component of this design. In order to better understand its operation, and particularly if we wanted to compare it to a Verilog equivalent, you are asked to develop a C version for this component. Accordingly, provide a C function to imitates the operation of this component. Add to your answer an explanation of what ports you think the CMP component should have and how you have emulated these ports, or emulating input changes, in C. [15 marks]

(c) Now implementation the CMP component as an actual Verilog module, you could call it cmp.v. Note you are not required to implement a testbench (see (d) below). [15 marks]
(*Hint:* Appendix A gives a Verilog cheat sheet.)

(d) Discuss how you could go about implementing a test bench for the CMP component that you have implemented. Note that *you are not* required to implement a testbench, however you can include code snippets to help illustrate you answer. It is important that plans for the testbench would be sufficiently thorough considering the important operational characteristics of the CMP component as explained in the design description. [6 marks]

Question 2 [10 marks]

A generally desirable aspect of a dynamically reconfigurable system is having aspects of the design that can fit on a single slice, so that just that slide can be reprogrammed without having to spend more time reprogramming other slices in the design. Some of the sub-questions below relate to the design on the previous page.

(a) What are potentially limiting factors on deciding whether or not a design, such as the design given on the previous page, would be able to fit into a single slice? [3 marks]

(b) If you had recently given a client your BOM (i.e. bill of materials) for the dual sensor monitor system, and he had come back to you with a complaint, saying that he had found on the web a PLD that costs just $2, which could replace the FPGA you proposed and thereby much reduce the cost. Considering that part of the requirement is it being dynamically scalable to over 5 simultaneous, what would your response be to motivate for sticking with an FPGA? [4 marks]

(c) Comment on the potential benefits that may be achieved by doing simulation of an Verilog or other HDL design rather than diving in to trying to get it working on hardware first (you could refer to a design concept, such as the design concept on the previous page, to use if needed as an example to aid your explanation). [3 marks]

END OF PART C EXAMINATION

Appendix A: Verilog Cheat sheet

Numbers and constants

Example: 4-bit constant 10 in binary, hex and in decimal: 4'b1010 == 4'ha -- 4'd10

(numbers are unsigned by default)

Concatenation of bits using {}

4'b1011 == {2'b10 , 2'b11}

Constants are declared using parameter:

parameter myparam = 51

Operators

Arithmetic: and (+), subtract (-), multiply (*), divide (/) and modulus (%) all provided.

Shift: left (<<), shift right (>>)

Relational ops: equal (==), not-equal (!=), less-than (<), less-than or equal (<=), greater-than (>), greater-than or equal (>=).

Bitwise ops: and (&), or (|), xor (^), not (~)

Logical operators: and (&&) or (||) not (!) note that these work as in C, e.g. (2 && 1) == 1

Bit reduction operators: [n] n=bit to extract

Conditional operator: ? to multiplex result

Example: (a==1)? funcif1 : funcif0

The above is equivalent to:

((a==1) && funcif1)

|| ((a!=1) && funcif0)

Registers and wires

Declaring a 4 bit wire with index starting at 0:

wire [3:0] w;

Declaring an 8 bit register:

reg [7:0] r;

Declaring a 32 element memory 8 bits wide:

reg [7:0] mem [0:31]

Bit extract example:

r[5:2] returns 4 bits between pos 2 to 5 inclusive

Assignment

Assignment to wires uses the assign primitive outside an always block, e.g.:

assign mywire = a & b

Registers are assigned to inside an always block which specifies where the clock comes from, e.g.:

```
always@(posedge myclock)
   cnt = cnt + 1;
```

Blocking vs. unblocking assignment <= vs. =

The **<=** assignment operator is non-blocking (i.e. if use in an always@(posedge) it will be performed on every positive edge. If you have many non-blocking assignments they will all updated in parallel. The <= operator must be used inside an always block – you can't use it in an assign statement.

The blocking assignment operator **=** can be used in either an assign block or an always block. But it causes assignments to be performed in sequential order. This tends to result in slower circuits, so avoid using it (especially for synthesized circuits) unless you have to.

Case and if statements

Case and if statements are used inside an always block to conditionally update state. e.g.:

```
always @(posedge clock)
  if (add1 && add2) r <= r+3;
  else if (add2) r <= r+2;
  else if(add1) r <= r+1;
```

Note that we don't need to specify what happens when add1 and add2 are both false since the default behavior is that r will not be updated. Equivalent function using a case statement:

```
always @(posedge clock)
  case({add2,add1})
  2'b11  : r <= r+3;
  2'b10  : r <= r+2;
  2'b01  : r <= r+1;
  default: r <= r;
endcase
```

Module declarations

Modules pass inputs, outputs as wires by default.

```
module ModName (
  output reg [3:0] result,  // register output
  input [1:0] bitsin,  input clk, inout bidirectnl  );
 … code …
endmodule
```

Verilog Simulation / ISIM commands

```
$display ("a string to display");
$monitor ("like printf. Vals: %d %b", decv,bitv);
#100   // wait 100ns or simulation moments
$finish  // end simulation
```