High Performance Embedded Systems EEE4120F

FINAL EXAM
19 June 2025

Venue: LESLIE SOCIAL: LS2B Time starting: 12h30 3 hours

Examination Prepared by:Simon Winberg *Last Modified: 19-May-2025*

NB! NB! FIRST ENTER YOUR NAME AND STUDENT NUMBER HERE:

Name (first and last name)	Student Number

REGULATIONS

This is a closed-book exam. Scan through the questions quickly before starting, so that you can plan your strategy for answering the questions. If you are caught cheating, you will be referred to University Court for expulsion procedures. Answer on the answer sheets provided.
Make sure to **put your student number on any separate answer sheets!** Also **put your name and student number** and course code **EEE4120F** and title **Final Exam** in your UCT exam answer book (this answer book is mainly for rough work, you need to return this question paper containing your answers).

FOLLOW THE EXAMINATION PROCEDURES ON THE **PROCEDURES PAGE** ON YOUR DESK

DO NOT TURN OVER UNTIL YOU ARE TOLD TO

NB!!! ANSWER QUESTIONS ON THIS PAPER, USE EXAM BOOKLET FOR ADDITIONAL SPACE IF NEEDED

Exam Structure

Marked out of 100 marks / 180 minutes. Time per mark = 1min 48sec

Section 1	Section 2	Section 3	MCQ answer block
Short Answers *(2 x 10-mark questions)* [20 marks] pg 2	Multiple choice *(10 x 5-mark questions)* [50 marks] pg 5	Long Answers *(2x question design/code)* [30 marks] pg 8	Appendices A: Verilog cheatsheet (can detach last page) pg 15

(hint: consider taking a quick look at Q3.1 and Q3.2 to help you plan your answering)

RULES

NB
- You must write your name and student number on each answer script book you hand in.
- Write the question numbers attempted on the cover of each answer book that you use and hand in.
- Make sure that you cross out material you do not want marked. Your first attempt at any question will be marked if two answers are found.
- Answer all questions, and note that the time for each question relates to the marks allocated to it.

Section 1: Short Answers [20 marks]

NB: Answer **any two** of the following questions. Have a quick look over the questions and then choose which one to skip and which two to attempt answering.

Question 1.1 [10 marks] *(remember: answer two of the Section 1 questions)*

High-performance embedded systems often utilize a complex memory hierarchy to achieve performance targets. Discuss the different levels typically found in a modern embedded system's memory hierarchy (e.g., caches, main memory, non-volatile storage) and explain how each level contributes to overall system performance. Include in your discussion the concepts of locality of reference (temporal and spatial) and how it is exploited by the memory hierarchy. [10 marks]

.

[Q1.1c total = 4 marks]

Question 1.2 [10 marks] *(remember: answer two of the Section 1 questions)*

Compare and contrast the use of interrupts and polling for handling I/O events in an embedded system. Discuss the advantages and disadvantages of each approach in the context of high-performance and real-time requirements. Provide scenarios where each method would be preferred (for example, you might want to refer to the Raspberry Pi part of Section 3 for ideas, where instead of bit-banging you decide on using a timer interrupt to send data to the ToneSoC being controlled).

Question 1.3 [10 marks] *(remember: answer two of the Section 1 questions)*

List the eight steps described in the lectures for the “steps in designing parallel programs” [3 marks]. Explain briefly why, even in the first step ("Understanding the Problem"), when you’re planning to develop a new version of an existing system, you still want to remember (as I kept reminding people in class) the saying “don’t lose sight of the forest for the trees” [4 marks]. Considering the types of benchmarking one might do on a system, explain why keeping this saying in mind is important during the start of a project. Finally, regardless of whether you remember which step it is in the line, briefly explain what the term granularity means in the context of computing [3 marks].

(exam continues next page)

Section 2: Multiple Choice [50 marks]

NOTE: Use the Multiple Choice Answer Sheet to record your answers for this section

Choose the most appropriate answer for each of the following questions. Each question is worth 5 marks.

Q2.1 Which of the following is a primary characteristic differentiating a High-Performance Embedded System (HPES) from a general-purpose computing system?

(a) Lower clock speeds

(b) Unlimited power consumption

(c) Dedicated function and often real-time constraints

(d) Large physical size

[5 marks]

Q2.2 Cache memory in an embedded system is primarily used to: …

(a) Store program instructions permanently

(b) Provide high-speed storage for frequently accessed data and instructions

(c) Interface with external peripherals

(d) Execute arithmetic and logical operations

[5 marks]

Q2.3 Which of the following is a common means that a multi-threaded application uses in order to avoid having multiple threads attempting to write to a variable or data address at the same time?

(a) Poll all (i.e. each thread must ask all other threads if that thread can change the data item)

(b) Busy-waiting flag (i.e. the thread wanting to write to a variable must just waiting in a while (1) loop on a flag variable that indicates permission to continue)

(c) Semaphores (i.e. using a system feature atomic operation whereby one thread at a time is given operation to proceed and all other threads are denied or paused until the semaphore is released)

(d) Infinite loops (i.e. where you basically just give up and sit in a while(1) and wait for an interrupt to happen to get out of the loop and do something)

[5 marks]

Q2.4 Pipelining in a processor is a technique used to: …

(a) Reduce the power consumption of the CPU

(b) Increase the number of instructions executed per unit of time

(c) Expand the available memory capacity

(d) Simplify the instruction set architecture

[5 marks]

Q2.5 Which of the following best describes the purpose of a Direct Memory Access (DMA) controller?

(a) To manage the execution of multiple threads

(b) To accelerate arithmetic calculations

(c) To allow peripherals to transfer data to or from memory without direct CPU intervention

(d) To handle interrupt requests from peripherals

[5 marks]

Q2.6 In the context of embedded systems, a 'watchdog timer' is typically used to: …

(a) Measure the execution time of critical code sections

(b) Provide a high-resolution system timer

(c) Detect and recover from system malfunctions (e.g., due to software errors) by resetting the system

(d) Synchronize access to shared resources

[5 marks]

Q2.7 Fixed-point arithmetic is often preferred over floating-point arithmetic in some embedded systems due to: …

(a) Higher precision

(b) Simpler implementation in hardware and often faster execution

(c) Larger dynamic range

(d) Easier debugging

[5 marks]

Q2.8 Which of the following is a key challenge in designing software for multi-core embedded processors?: …

(a) Lack of available compilers

(b) Ensuring proper synchronization and avoiding race conditions between cores

(c) Limited memory bandwidth (compared to single-core)

(d) Reduced power consumption

[5 marks]

(exam continues next page)

Q2.9 The term 'determinism' in a real-time embedded system refers to …

(a) The speed of task execution

(b) The ability of the system to complete tasks within a guaranteed time frame

(c) The total number of tasks that can be executed

(d) The amount of memory available to tasks

[5 marks]

Q2.10 Which of the following peripheral interfaces is commonly used for communication with sensors and other low-speed devices in embedded systems?…

(a) PCIe

(b) Ethernet

(c) I2C (Inter-Integrated Circuit)

(d) HDMI

[5 marks]

(exam continues next page)

Section 3: Long Answers [30 marks]

This section involves one rather lengthy answer and one not-so-lengthy answer

ToneSoC Development Scenario

This section requires you to apply your knowledge of high-performance embedded systems design, including digital logic design using Verilog, state machines, and understanding basic digital signal generation. You have approximately 1 hour for this section.

You are working with a System-on-Chip (SoC) design, prototyped on an FPGA, called the **ToneSoC**. The ToneSoC is designed to generate audio tones and is intended to be a simple, low-cost audio output device. It includes a basic Digital-to-Analog Converter (DAC) output (aout) that, when connected via a capacitor to an amplifier and speaker, can produce sound.

The ToneSoC receives commands and data via a simple serial communication link. It only receives messages and does not send replies or perform handshake procedures. The serial input consists of two lines: din (data input) and sclk (serial clock input). The communication is synchronous, with the ToneSoC reading one bit from the din line on each **positive edge** of the sclk signal.

Command packets sent to the ToneSoC are 12 bits long with the following structure:

```
Packet part:  <MSG: 3 bits> <DAT: 9 bits>
Bit Index:    11..9         8..0
```

The MSG field specifies the command, and the DAT field provides data for that command. The relevant commands and their effects are:

- **MSG 000 (Nop):** Does nothing. The DAT field is ignored.
- **MSG 001 (tone):** Selects a tone to produce. The DAT field specifies the tone number (ntone).
- **MSG 010 (volume):** Sets the output volume. The DAT field specifies the volume level (volume). A value of 0 means no sound, and 511 is maximum volume.
- **MSG 011 (fadeout):** Initiates a volume fade-out. The DAT field specifies the fade-out duration in units of 10 milliseconds.

The ToneSoC has its own internal clock running at 100KHz, available as clk.

Figure 1 on the next page shows a block diagram of the ToneSoC and its connection to a Raspberry Pi host computer, which acts as the sender of the serial commands.

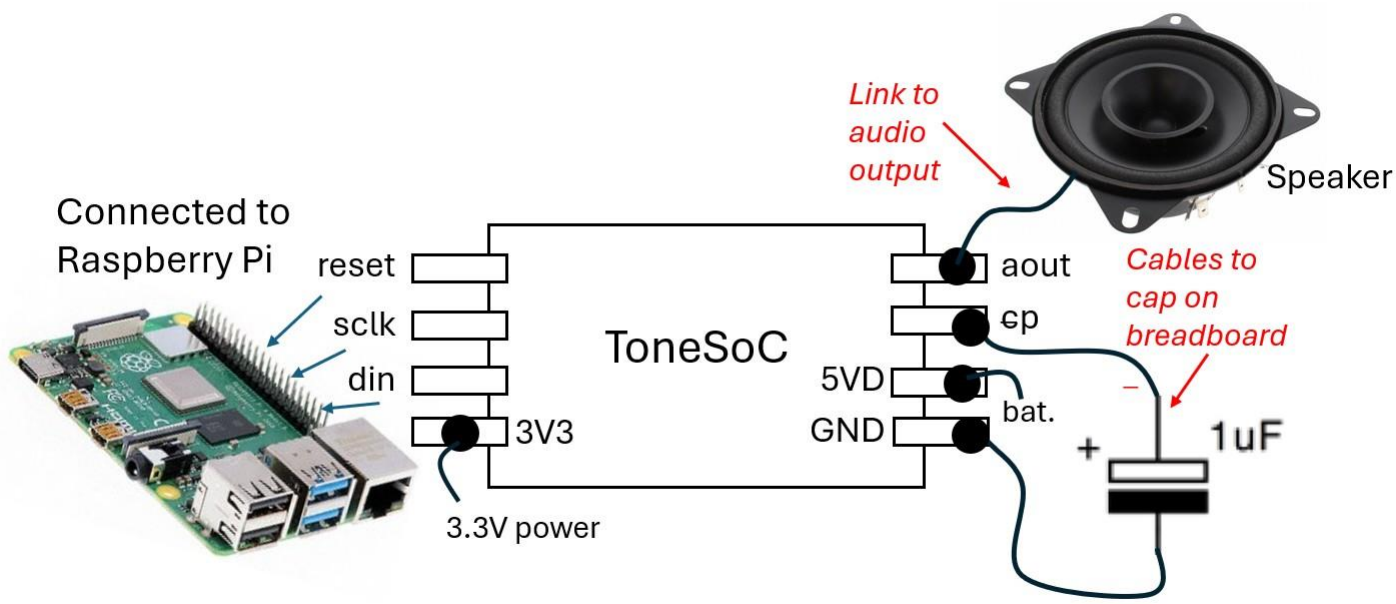

Figure 1: Block diagram of the ToneSoC chip connected up to a Raspberry Pi host computer. The diagram shows the aout line of the ToneSoC connecting to a capacitor and speaker which produces the tones.

The Raspberry Pi uses a bit-banging approach to send data, controlling the sclk and din lines directly from its General Purpose I/O (GPIO) pins. An example C function sendcmd is provided below, illustrating how a 12-bit packet is sent:

```
void sendcmd ( unsigned char MSG, unsigned short DAT )
{
    // Assuming p_sclk and p_din are pointers to GPIO memory locations
    // Assuming MSB first transmission based on packet structure
    // <MSG:11..9><DAT:8..0>

    // Send MSG (3 bits, MSB first)

    // Send MSG bit 2 (bit 11 of packet):
    *p_sclk = 0; *p_din = (MSG >> 2) & 0x1; *p_sclk = 1;

    // Send MSG bit 1 (bit 10 of packet)
    *p_sclk = 0; *p_din = (MSG >> 1) & 0x1; *p_sclk = 1;

    // Send MSG bit 0 (bit 9 of packet)
    *p_sclk = 0; *p_din = (MSG >> 0) & 0x1; *p_sclk = 1;

    // Send DAT (9 bits, MSB first)
    for (int i = 8 ; i >= 0 ; i--) { // Loop from bit 8 down to 0 (MSB first)
        *p_sclk = 0; *p_din = (DAT >> i) & 0x1;
        *p_sclk = 1; // send DAT bit i
        }
    *p_sclk = 0; // Leave sclk low
}
```

Now procced to the next page to answer the questions …

Question 3.1: Design and Implementation of build_msg (20 marks)

You are required to design and implement the Verilog module build_msg. This module receives serial data on the din input, synchronized by the external sclk. It needs to assemble the 12-bit packet and, upon receiving the final bit, decode the MSG field and take appropriate action by updating signals that control the tone_gen module. The ToneSoC's internal 100KHz clock (clk) is available, although the core serial reception is synchronized by sclk.

The tone_gen module has the following interface (you do **not** need to implement tone_gen, just interface with it):

```
module tone_gen (
    input wire clk,      // 100KHz internal clock
    input wire en,       // Tone generation enable (active high)
    input wire [8:0] ntone, // Tone selection (0-511)
    input wire [8:0] vol     // Volume level (0-511)
);
    // ... internal tone generation logic ...
endmodule
```

Q3.1a: Design (8 marks)

First, design the functionality of the build_msg module. You are recommended to think in terms of a state machine to handle the serial reception and command decoding.
Provide a pseudocode description explaining how build_msg would work. This helps clarify your logic and can serve as a backup plan if you run out of time for coding. More importantly, provide a state machine diagram illustrating the operation of build_msg. Include states for receiving bits and decoding the command, and show transitions based on sclk edges and bit count. Clearly indicate how the module will update the signals intended for the tone_gen module based on the received command.

Your Q3,1a answer:

P.T.O…

(more answer space available on the back on the next page, Appendix A, if you choose not to detach it)

Q3.1b: Verilog Implementation (12 marks)

Now, implement the Verilog code for the build_msg module. Your implementation should be based on your design from Q3.1a (although you can adapt your design if necessary during implementation). The module should receive sclk and din and output the control signals required by the tone_gen module (en, ntone, vol). A starting point for the module interface is provided below. You can add internal registers and wires as needed. Remember that serial data is read on the positive edge of sclk, and the packet is 12 bits long with the structure <MSG:11..9><DAT:8..0>.

```
module build_msg (
    input wire clk,      // 100KHz internal clock (optional for serial logic)
    input wire sclk,     // External serial clock input
    input wire din,      // Serial data input
    input wire reset,    // Asynchronous reset (active high)

    // Outputs to control tone_gen
    output reg en,
    output reg [8:0] ntone,
    output reg [8:0] vol
);

    // Add internal signals, parameters, state register, counters, etc. here

    // Add sequential logic (always @(posedge ...)) for state transitions,
    // counter updates, shift register operations, and output updates

    // Add combinational logic (assign or always @(*)) if needed

    // add your code here ...

endmodule
```

Your Q3.1b answer:

P.T.O…

(more answer space available on the back on the next page, Appendix A, if you choose not to detach it)

Question 3.2: Tone Generation Design (10 marks)

Explain how a tone could be produced by the tone_gen module using the available 100KHz clk input, the ntone input (0-511), and the vol input (0-511). The desired tone frequency is ntone * 100 Hz. You do **not** need to provide Verilog code for this question, just a clear description in English (you may include pseudocode or diagrams if they help). Describe how the 100KHz clock, ntone, and vol would be used to generate the appropriate analog output signal via the DAC.

Your Q3.2 answer:

(more answer space available on the back on the next page, Appendix A, if you choose not to detach it)

END OF EXAMINATION

Multiple Choice Answer Block

Please use the table below to fill in answers to Section 2 Multiple Choice Questions

Use a cross X to clearly mark your choice

You may use pencil to fill in answers

If using pen, and wanting to change an answer, then thoroughly scribble over your answer to cancel it

Question	(a)	(b)	(c)	(d)	(e)
2.1					
2.2					
2.3					
2.4					
2.5					
2.6					
2.7					
2.8					
2.9					
2.10					

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
```

(additional answer space, remember to indicate question numbers)