FPGA Basics
EEE5117Z

FINAL EXAM

26 July 2022

3 hours

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 that you put your **student name and student number**, the course code **EEE5117Z** and a title **Final Exam** on your answer sheet(s).

DO NOT TURN OVER UNTIL YOU ARE TOLD TO

Exam Structure
Marked out of 120 marks / 180 minutes.

Section 1	Section 2	Section 3	Appendices
Short Answers (4 questions) [54 marks] pg. 2	Multiple Choice (4×2 mark questions + 5×2 mark true/false q's) [18 marks] pg 3	Long Answers (2 questions) [48 marks] pg 5	A: PCM1774 Datasheet B: Verilog Reference pg 7

RULES

- You must write your name and student number on each answer book.
- Some questions are to be answered on the question paper. Write your name and student number on the page that you answered on and insert that page into your answer book.
- Write the question numbers attempted on the cover of each book.
- **Start each section on a new page.**
- 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.
- Use a part of your script to plan the facts for your written replies to questions, so that you produce carefully constructed responses.
- Answer all questions.

Section 1: Short Answers [54 marks]

Question 1.1 [12 marks]

With the aid of diagrams, explain the following concepts related to FPGAs:

1.1 (a) What is a logic element, and how can it be used to implement logic gates? **[4]**

1.1 (b) How is signal routing generally implemented within an FPGA IC? **[4]**

1.1 (c) Briefly outline the features of at typical FPGA embedded RAM block. **[4]**

Question 1.2 [8 marks]

Briefly outline the three stages of compiling FPGA source code into a programmable bit-stream. **[8]**

Question 1.3 [18 marks]

1.3 (a) What is the difference between blocking an non-blocking statements. Illustrate your explanation with an example. **[4]**

1.3 (b) What is meant by an “inferred latch”, and how can it be avoided? **[4]**

1.3 (c) Explain the advantages of using localised and synchronous reset signals in state machines. **[4]**

1.3 (d) With the aid of a diagram, briefly explain what is meant by the ‘setup’ and ‘hold’ requirements of a register, and why it is important to define the clock frequency of all clock signals. **[6]**

Question 1.4 [16 marks]

1.4 (a) Explain what is meant by latency and throughput when referring to the properties of a processing module. **[4]**

1.4 (b) Explain why pipelining improves throughput,
and why this is at the cost of latency. **[4]**

1.4 (c) Explain how flow-control is generally implemented in streaming processors. Make specific reference to ‘valid’ signals and the concept of ‘back pressure’. **[4]**

1.4 (d) Explain how you would solve the problem where one processing module outputs data at a constant rate, but the next block in the chain can only accept data in bursts with relatively long ‘busy’ times. **[4]**

Section 2: Multiple Choice [18 marks]

Choose one option for each of the questions 2.1 to 2.4 in this section.

2.1 JTAG is: **[2]**

(a) A communication protocol primarily used for talking to FPGAs.

(b) A header tag added to the data when programming an FPGA.

(c) A USB device driver used to program FPGAs.

(d) A way to tag J-type devices.

(e) An industry standard primarily meant for testing integrated circuits, but also used for other purposes.

2.2 Consider the Verilog code for moduleX. Select the option that best describes what this module does. **[2]**

(a) This module returns active 1 or active 0 depending on whether the module has been tasked to count up or to count down.

(b) This is an up/down counter with reset, it counts up on every positive clock when active is high or counts down when active is low.

(c) This is an up counter with reset, it continuously counts up on every positive clock when rst is low.

(d) This is a down counter, on reset high the initial count is set and held; when reset is low it starts counting down with every clock until zero is reached at which point it sets active high.

(e) This module initially sets active high, then counts up from 0 and when the maximum value is reached it sets active to low.

```verilog
// Code for Question 2.2
module moduleX  (
  output reg [7:0]out,    // an output
  output          active, // another output
  input           clk,    // clock input
  input           rst     // reset input
);

// ------ Implementation -------
always @(posedge clk)
  if (rst) begin
    out    <= 8'b0;
    active <= 1'b0;
  end else begin
    out    <= out + 1;
    active <= 1'b1;
  end
endmodule
```

2.3 False path design constraints are generally used for: **[2]**

- (a) Informing the compiler that the signal is active-low.
- (b) Telling the compiler about asynchronous signals that should be removed from timing considerations.
- (c) Debug purposes: it temporarily removes a signal path from the circuit.
- (d) An apparent connection between two independent state machines that the compiler should remove.
- (e) Telling the compiler that the path should default to a low value.

2.4 “Hard IP” refers to: **[2]**

- (a) A protocol that is difficult to implement.
- (b) Intellectual property that is legally bound to a specific company.
- (c) A circuit that is difficult to implement.
- (d) Ready-built circuits that are included on the FPGA integrated circuit.
- (e) FPGA modules that are provided by the IDE.

2.5 Answer **true** or **false** to each question below (each answer is 2 marks). **[10]**

- (a) FPGAs, at the fundamental level, execute instructions provided in binary form.
- (b) FPGAs can control multiple external peripherals at the same time.
- (c) Most FPGAs include dedicated hardware for performing many multiplications in parallel.
- (d) Low level Verilog and VHDL modules (i.e. not the top level module) can define external FPGA pins directly.
- (e) It is generally safe to assume that all FPGA registers are in a known state after a power-on reset.

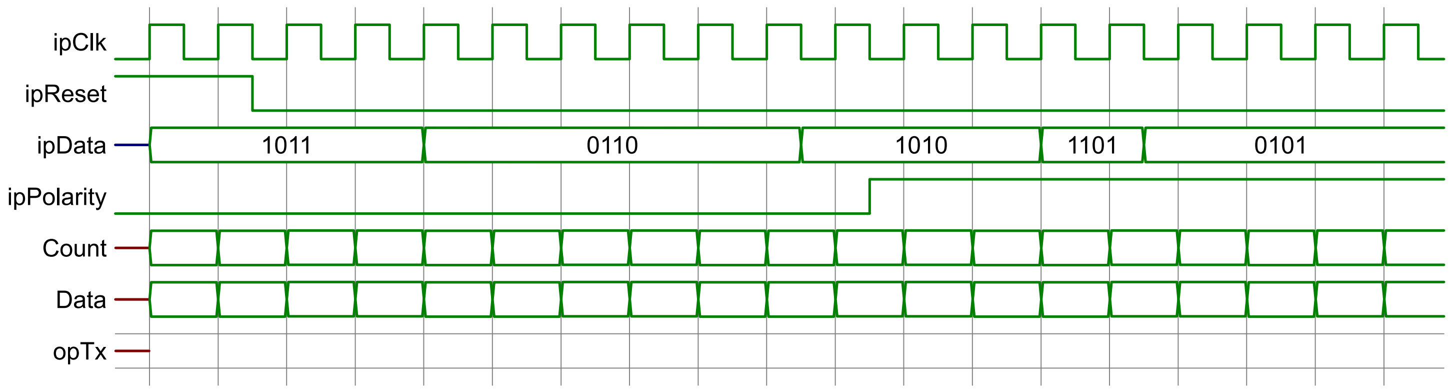
Section 3: Long Answers [48 marks]

Question 3.1 [16 marks]

3.1 (a) Draw the circuit described by the Verilog code below. Don't draw the full gate-level circuit – rather make use of multi-bit registers, multi-bit multiplexers and high-level blocks such as `Add one` or `Shift right`. **[10]**

```
module Shifter(input      ipClk , input ipReset,
               input [3:0]ipData, input ipPolarity,
               output     opTx);

reg [3:0]Data;
reg [1:0]Count;

assign opTx = Data[0] ^ ipPolarity;

always @(posedge ipClk) begin
  if(ipReset) begin
    Count <= 0;
    Data  <= 'hX;

  end else begin
    Count <= Count + 1'b1;
    if(|Count) Data <= {1'b0, Data[3:1]};
    else       Data <= ipData;
  end
end

endmodule
```

3.1 (b) Draw the timing diagram of all the signals in the Verilog above. Use the timing diagram skeleton below (i.e. draw on the question paper). Use `X` to mean "unknown". **[6]**



Question 3.2 [32 marks]

You are tasked to work on a project that implements a USB sound device. You are responsible for the output stage. In other words, your FPGA module must translate the data stream (see the `SoundStream` structure defined below) to something that the external digital to analogue converter (DAC) can understand.

Excerpts from the PCM1774 DAC datasheet is presented in the appendix. The device is set up in slave mode, implying that the `LRCK`, `BCK` and `DIN` signals are all output from the FPGA to the PCM1774.

```
// Streams.v
package Streams;
  typedef struct{
    logic [15:0]Left;  // 2's Compliment
    logic [15:0]Right; // 2's Compliment
    logic       Valid; // 48 kSps, continuous stream
  } SoundStream;
endpackage
```

Assume that your module does not require any other control. You only require a clock, reset, input data stream and external IC interface. The audio sampling rate is fixed at 48 kSps and the module is clocked at 49.152 MHz.

The `Valid` line of the input stream is high for exactly one clock cycle for every audio sample, and you cannot assume that the data is valid when the `Valid` line is low.

The parts of this question present the marking scheme, and does not imply that you need to answer each one separately. A single Verilog-based module definition is sufficient for all parts.

Timing constraints are outside the scope of this question, so do not include the SDC commands in your answer.

Trivial syntax errors (such as forgotten semicolons) will not be penalised, but the overall logic of the implementation must be sound.

3.2 (a) Design the skeleton of your module, including all module port definitions. **[8]**

3.2 (b) Generate the correct `LRCK` and `BCK` waveforms, including correct timing. You are free to choose an appropriate `BCK` frequency. If you use a reset signal, make it local and synchronous. **[10]**

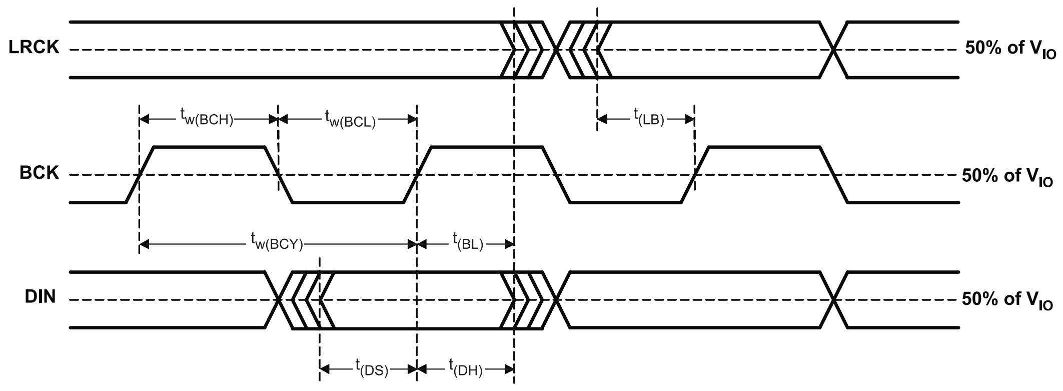
3.2 (c) Drive the `DIN` signal so that the correct audio is produced by the PCM1774. **[14]**

END OF EXAMINATION

Appendix A: PCM1774 Datasheet Excerpts

Audio Data Formats and Timing

The PCM1774 supports I^2S, right-justified, left-justified, and DSP formats. The data formats are shown in Figure 16 and are selected using registers 70 and 81 (RFM[1:0], PFM[1:0]). All formats require binary 2s-complement, MSB-first audio data. The default format is I^2S. Figure 14 shows a detailed timing diagram.



PARAMETERS		MIN	MAX	UNITS
$t_{(BCY)}$	BCK pulse cycle time (I^2S, left- and right-justified formats)	$1/(64\ f_S)$(1)		
	BCK pulse cycle time (DSP format)	$1/(256\ f_S)$(1)		
$t_{w(BCH)}$	BCK high-level time	35		ns
$t_{w(BCL)}$	BCK low-level time	35		ns
$t_{(BL)}$	BCK rising edge to LRCK edge	10		ns
$t_{(LB)}$	LRCK edge to BCK rising edge	10		ns
$t_{(DS)}$	DIN set up time	10		ns
$t_{(DH)}$	DIN hold time	10		ns
t_r	Rising time of all signals		10	ns
t_f	Falling time of all signals		10	ns

(1) f_S is the sampling frequency.

Figure 14. Audio Interface Timing (Slave Mode)

(b) I^2S Data Format; L-Channel = LOW, R-Channel = HIGH, LRPC = 0

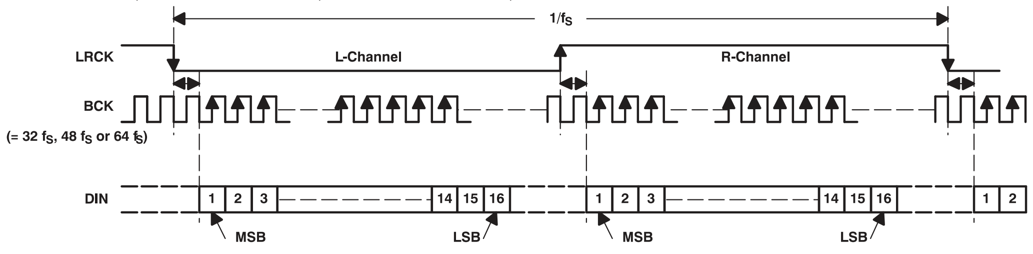


Figure 16. Audio Data Formats

Appendix B: Verilog Reference

Comments

```
// One-liner
/* Multiple
   lines */
```

Numeric Constants

```
// The 8-bit decimal number 106:
8'b_0110_1010 // Binary
8'o_152       // Octal
8'd_106       // Decimal
8'h_6A        // Hexadecimal
"j"           // ASCII

78'bZ         // 78-bit high-impedance
```

Nets and Variables

```
wire [3:0]w; // Assign outside always blocks
reg  [1:7]r; // Assign inside always blocks
reg  [7:0]mem[31:0];

integer j; // Compile-time variable
genvar  k; // Generate variable
```

Parameters

```
parameter  N     = 8;
localparam State = 2'd3;
```

Assignments

```
assign Output = A * B;
assign {C, D} = {D[5:2], C[1:9], E};
```

Module

```
module MyModule
#(parameter N = 8) // Optional parameter
 (input  ipReset, ipClk,
  output [N-1:0]opOutput);
// Module implementation
endmodule
```

Module Instantiation

```
// Override default parameter: setting N = 13
MyModule #(13) MyModule1(Reset, Clk, Result);
```

Packages

```
package MyPackage;
  //Define stuff
endpackage
```

Using Packages

```
import MyPackage::*;
```

Operators

```
// These are in order of precedence...
// Select
A[N] A[N:M]
// Reduction
&A ~&A |A ~|A ^A ~^A
// Compliment
!A ~A
// Unary
+A -A
// Concatenate
{A, ..., B}
// Replicate
{N{A}}
// Arithmetic
A*B A/B A%B
A+B A-B
// Shift
A<<B A>>B
// Relational
A>B A<B A>=B A<=B
A==B A!=B
// Bit-wise
A&B
A^B A~^B
A|B
// Logical
A&&B
A||B
// Conditional
A ? B : C
```

Case

```
always @(*) begin
  case(Mux)
    2'd0: A = 8'd9;
    2'd1,
    2'd3: A = 8'd103;
    2'd2: A = 8'd2;
    default:;
  endcase
end

always @(*) begin
  casex(Decoded)
    4'b1xxx: Encoded = 2'd0;
    4'b01xx: Encoded = 2'd1;
    4'b001x: Encoded = 2'd2;
    4'b0001: Encoded = 2'd3;
    default: Encoded = 2'd0;
  endcase
end
```

Synchronous

```
always @(posedge Clk) begin
  if(Reset) B <= 0;
  else      B <= B + 1'b1;
end
```

Loop

```
always @(*) begin
  Count = 0;
  for(j = 0; j < 8; j = j+1)
    Count = Count + Input[j];
end
```

Function

```
function [6:0]F;
  input [3:0]A;
  input [2:0]B;
  begin
    F = {A+1'b1, B+2'd2};
  end
endfunction
```

Generate

```
genvar j;
wire [12:0]Output[19:0];

generate
  for(j = 0; j < 20; j = j+1)
  begin: Gen_Modules
    MyModule #(13) MyModule_Instance(
      Reset, Clk,
      Output[j]
    );
  end
endgenerate
```

State Machine

```
enum { Start, Idle, Work, Done } State;

reg Reset;

always @(posedge ipClk) begin
  Reset <= ipReset;

  if(Reset) begin
    State <= Start;

  end else begin
    case(State)
      Start: begin
        State <= Idle;
      end
      Idle: begin
        State <= Work;
      end
      Work: begin
        State <= Done;
      end
      Done: begin
        State <= Idle;
      end
      default:;
    endcase
  end
end
```