# FPGA Basics

# EEE5117Z

FINAL EXAM – MEMORANDUM

26 July 2022

3 hours

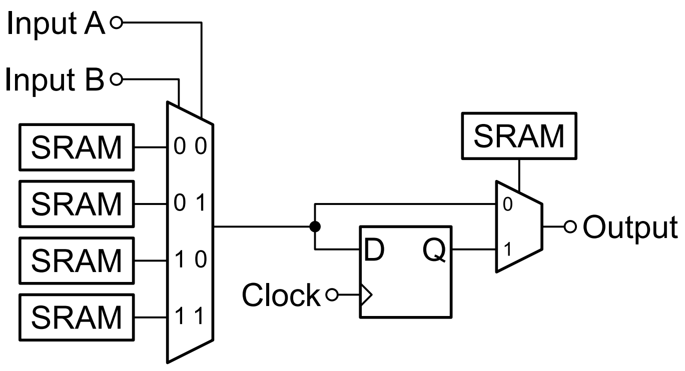
## Section 1: Short Answers [54 marks]

**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]**

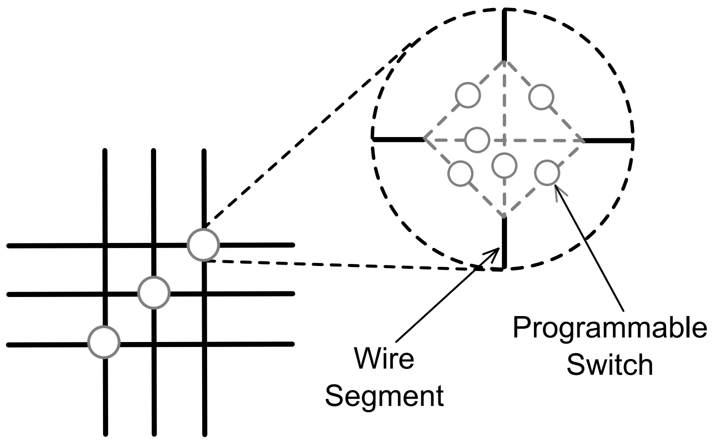
**Answer:** A simple 2-input LUT-based LE is shown below. It includes a register.



The SRAM cells are programmed with whatever the circuit requires. For example, the inputs to the multiplexer can be programmed with '0001' to implement an AND gate, or '0111' to implement an OR gate. The signal can either pass through or bypass the register by programming the correct selection into the SRAM cell above above the output multiplexer.

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

**Answer:** The elements of an FPGA (LEs, RAM blocks, etc.) are arranged in a grid. The spacing between elements are filled with wires, and each wire crossing happens through a switch-box, as shown below.



The switches of the switch-box are programmed in such a way that the signals are routed along the correct path.

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

**Answer:** The 4 marks are given for any 4 of the following list:

- A memory unit typically between 10 and 20 Mb in size
- Various programmable configuration, such as 1024$\times$18-bit, 4096$\times$4-bit, etc.
- Each RAM block as two independent read / write ports, including independent clocks, etc.
- The two independent ports can have different configurations
- The output ports can optionally be registered
- The contents can be initialised during FPGA configuration (i.e. boot-time)

**Question 1.2 [8 marks]**

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

**Answer:** Two marks each: one for the name, and another for the brief explanation.

- The **Analysis** and / or **Synthesis** stage converts the source code into a synthesisable netlist.
- The **Fitter** performs a place-and-route to realise the netlist in the FPGA fabric.
- The **Assembler** combines output from the fitter with other components (such as memory initialisation files) to generate the bit-stream.

Plus another 2 points for pointing out any one of the following list:

- The synthesis stage maps the circuit to internal components, such as multiplier, RAM blocks, etc.
- The synthesis stage performs circuit optimisation, such as removing circuits that does not drive further logic.
- The fitter takes timing and other design constraints into account.

**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]**

**Answer:** Blocking statements behave as if they are evaluated immediately, before the rest of the statements in the "always" block are evaluated. The order of statements is important.

Non-blocking statements behave as if their right-hand-sides are evaluated in parallel and independently, with the results assigned all at once when the evaluation has reached the end of the "always" block (or wait statement when simulating). The order of statements is **not** important.

```
// Blocking example:
always @(posedge ipClk) begin
  A = C & B;
  B = A | C; // The new value of "A" is used, giving "B = (C & B) | C"
end

// Non-blocking example:
always @(posedge ipClk) begin
  A <= C & B;
  B <= A | C; // The old value of "A" is used
end
```

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

**Answer:** When defining a combinational circuit (i.e. a circuit that is not sensitive to a clock edge), a latch is inferred when the source code does not fully define a signal for every branch of `if` and `case` statements. The circuit is built to "remember" the values in scenarios when the undefined branch happens. (2 marks)

It can be avoided by fully defining every possible branch in `if` statements (i.e. always assign all signals in both "main" and "else" bodies). For `case` statements, always include a "default" case and make sure to assign all signals in all cases. (2 marks)

It is generally a good idea to explicitly assign "don't care" when appropriate.

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

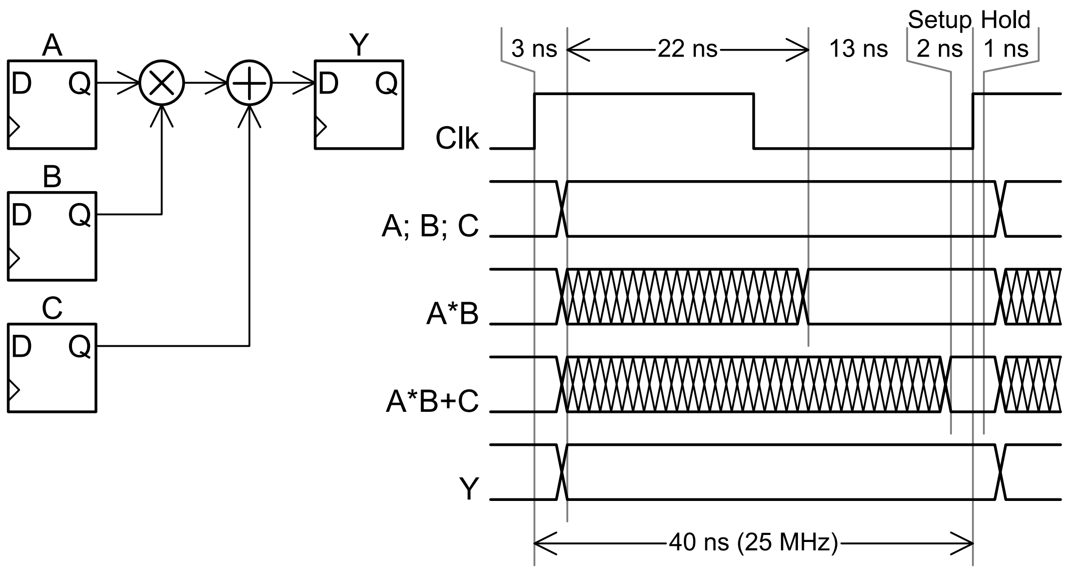
**Answer:** Synchronous resets are more deterministic, and the timing is better defined. (1 mark)

By localising reset signals, the reset distribution is pipelined. This reduces the fan-out of reset signals, which in turn makes it easier for the fitter to meet timing requirements on the reset network. (3 marks)

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]**

**Answer:** The input to a register has to be stable around the clock edge. The ‘setup’ time is the time of stability before the clock edge, and ‘hold’ time is the time of stability after the clock edge. (2 marks)

The combinational part between registers has a finite time to settle in order to meet the setup requirement of the target register. The compiler has to know what this time is in order to synthesise the circuit, which is why the clock frequency has to be defined. (2 marks)



The figure can be much simpler than shown above. The important aspects are that the input of the register takes time to settle, and it is stable around the clock edge, with ‘setup’ and ‘hold’ indicated. (2 marks)

**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]**

**Answer:** Latency is related to the number of stages of a pipeline. It is the number of clock cycles from the time a specific input is provided to the input of a pipeline to the answer related to that input appearing on the output. (2 marks)

Throughput specifies how often the processor can accept new input. (2 marks)

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

**Answer:** Pipelining generally involves inserting extra registers in order to break large combinational circuits into smaller blocks. These smaller blocks settle faster, meaning that the setup time of a faster clock can be met. A faster clock means that input can be accepted more often, implying an increase in throughput.

Each layer of extra registers introduces a clock-cycle of delay, which increases the latency.

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]**

**Answer:** Consider the case where a source is streaming answers to a sink for further processing.

The source marks valid answers by asserting a 'valid' signal. This tells the sink that it must take the data on the bus as valid input and perform the next step in the calculation.

The sink may not always be ready to receive data, however. It informs the source that it is busy by means of a 'back pressure' mechanism. This is typically implemented by means of a 'ready' line. The sink asserts the 'ready' signal to inform the source that it is allowed to send 'valid' data.

The details behind the handshaking is not important in the answer to this question (it is only worth 4 marks, after all). The important bit is that the data is transferred by means of 'transactions', which occur when both 'valid' and 'ready' are high during the same clock cycle.

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]**

**Answer:** Insert a FIFO queue between the two modules. Make sure that the FIFO is large enough to store at least two bursts.

The FIFO will be able to accept data from the source at the constant rate, as well as serve the data in fast bursts to the next block in the chain.

## 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. **Answer:** $\Leftarrow$ This one

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. **Answer:** $\Leftarrow$ This one

(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. **Answer:** $\Leftarrow$ This one

(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. **Answer:** $\Leftarrow$ This one

(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.**Answer:** False – FPGAs can implement a circuit that executes binary instructions (like an embedded microcontroller), but fundamentally FPGAs implement a digital circuit.

(b) FPGAs can control multiple external peripherals at the same time. **Answer:** True

(c) Most FPGAs include dedicated hardware for performing many multiplications in parallel. **Answer:** True

(d) Low level Verilog and VHDL modules (i.e. not the top level module) can define external FPGA pins directly. **Answer:** False

(e) It is generally safe to assume that all FPGA registers are in a known state after a power-on reset. **Answer:** False – Many FPGAs do not support register initialisation. It is much safer to assume a random state and issue an explicit reset.

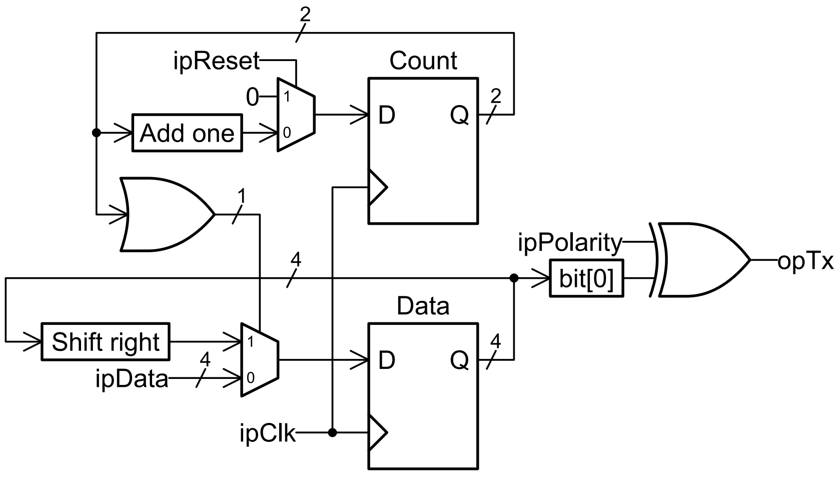
## Section 3: Long Answers [48 marks]

### Question 3.1 [16 marks]

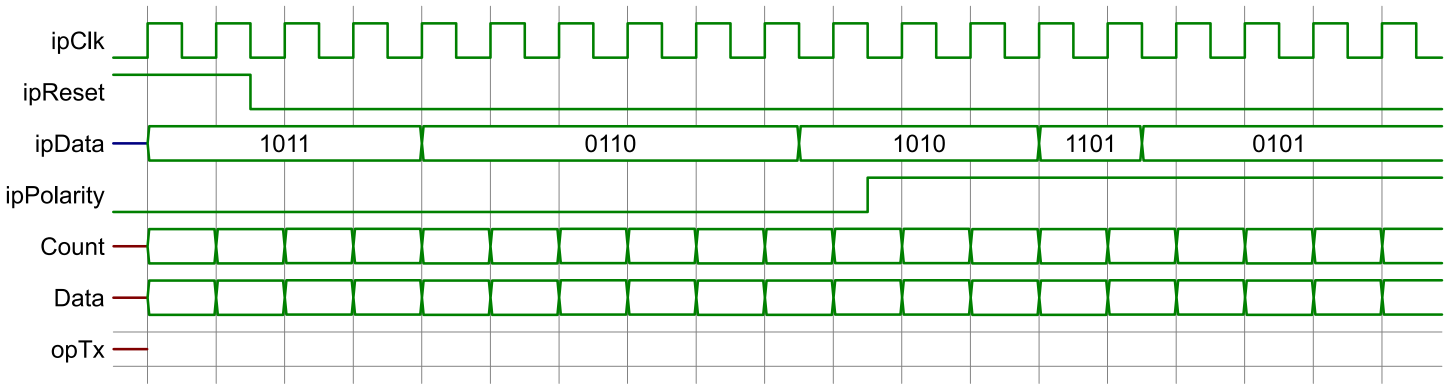
3.1 (a) Draw the circuit described by the Verilog code below. Don't draw the full gate-level circuit – rather make use of multi-bit registers, multi-bit multiplexers and high-level blocks such as `Add one` or `Shift right`. **[10]**

```
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
```

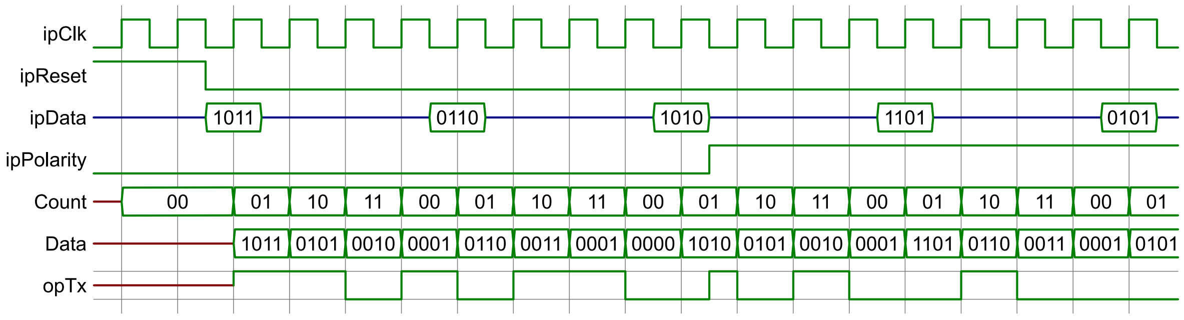
**Answer:** One possible circuit is:



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]**



**Answer:** The timing diagram should show at least that the `Polarity` function is asynchronous, the data is registered on the point where `Count` goes from 0 to 1 and data is transmitted least-significant bit first. In the diagram below, blue means "don't care" and red means "unknown".



If the answer includes a valid `Data` before the reset goes low, the answer is still valid, because the explicit 'don't care' assignment effectively removes the reset dependency.

**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]**

**Answer:** Something along the lines of:

```
import Streams::SoundStream;

module PCM1774(
  input ipClk, // 49.152 MHz
  input ipReset,

  input SoundStream ipStream,

  // External IC connection
  output     opLRCK,
  output     opBCK,
  output reg opDIN
);

// Module body goes here...

endmodule
```

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]**

**Answer:** The answer can choose to implement either 32 $f_s$ or 64 $f_s$ variations of the interface. The 48 $f_s$ requires a non-integer clock division, making it impractical.

The answer below assumes the 64 $f_s$ option. There is no reset required, because the system will auto-synchronise, so if the answer does not include a reset it will not be penalised.

```
reg      Reset;
reg [9:0]Count;

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

  if(Reset) Count <= 0;
  else      Count <= Count + 1;
end

assign opLRCK = Count[9];
assign opBCK  = Count[3];
```

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

**Answer:** The data must be double-buffered, because you cannot assume input data validity at all times. You also cannot assume that the `Valid` flag is in phase with your data framing, so the left and right shift registers must be initialised at the same time.

The data must be MSb first, and left-aligned, with a single cycle offset at the start of the frame. In other words, it must follow the timing diagram as specified in the appendix.

```
SoundStream Stream;
reg [15:0]Left;
reg [15:0]Right;

always @(posedge ipClk) begin
  if(ipStream.Valid) Stream <= ipStream;

  if(Count == 0) begin
    Left  <= Stream.Left;
    Right <= Stream.Right;

  end else if(opLRCK) begin // Right
    if(&Count[3:0]) {opDIN, Right} <= {Right, 1'b0};

  end else begin // Left
    if(&Count[3:0]) {opDIN, Left} <= {Left, 1'b0};
  end
end
```

---

END OF EXAMINATION