Test 2: Lectures 17 to 22

EEE4084F 2017-06-08

Instructions:

- Answer on separate pages. Verilog cheatsheet provided on last page.
- Make sure that your student number is on all your answer pages.
- There are **4** questions, each divided into sub-questions. Answer all questions.
- Total time: 1 hour.
- Total marks: 50.

Question 1: Thoughts about programmable logic devices and FPGAs [10 Total]

This question concerns Lecture 17 and Chapter 10.

Configurable Logic Blocks (CLBs) and I/O Blocks (IOBs) are major components of a FPGA design. Figure 1 below illustrates a simplified view of an IOB and CLB.

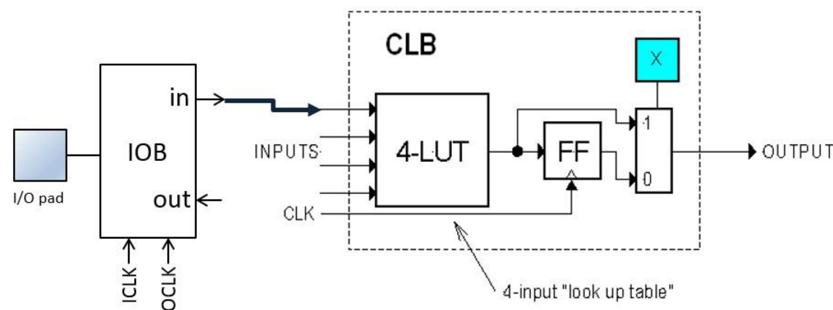

Figure 1: IOB (left) and CLB (right)

#1 Given the CLB above (module on right), what is the significance of the multiplexer on the output that is controlled by memory bit register X? Briefly explain what this aspect allows for and why it proves useful in the design. **[3]**

#2 The CLB design above is stated as being a 'simplification'. How so? Surely all FPGA CLBs follow this design with input lines that control what value a LUT sends to the output. Explain briefly why this is a simplification and how CLBs can be considerably more complex, you can (optionally) use a rough diagram to assist your explanation. **[4]**

#3 Figure 1 shows the IOB input line connected directly to an input line of the CLB. But surely this is not how it is actually done: You cannot just simply insert a wire into the FPGA to do this connection. Show how a PSM (programmable switch matrix) is used to make such programmable links. Provide a rough drawing showing how programmability is provided for linking the *in* port of the IOB to an input of the CLB, and what additional memory (if any) is needed to allow programmability of such a connection (only a little explanatory text or labelling of the diagram is needed for your answer). **[3]**

Question 2: Thoughts about computing devices [10 Total]

This question concerns Lecture 17 and Chapter 13.

#1 When quantifying performance of DSP processing it is typically arithmetic operations that are used in the basis for comparison. Surely other operations such leading from memory and exchanging values between registers are also critical… but maybe not so critical to DSP. Provide a brief argument motivating why a basis of arithmetic operations, as opposed to any operations, is useful in the case of assessing the suitability of a processor architecture for signal processing. **[4]**

#2 A DSP processor in a portable device is executing at its maximum clock rate of 100MHz, at 3.3V. But, the processor is guzzling too much power, drawing on 4W of power which drains the battery in an hour. According to application requirements, the processor does not need to be clocked at more than 20MHz and can operate reliably down to 1.2V. Propose how you could cut down the power consumption. Use the power formula[1] for relating clock speed to Watts to show how much you could cut this down by. Indicate what you would do to obtain the lowest operational power consumption. Roughly calculate how long the battery may likely last in this scenario (assume that unlikely case that the rate of discharge does not impact the total energy delivered by the battery). **[6]**

Question 3: Thoughts about computing devices [10 marks]

This question relates to ADCs covered in Seminar 6 / Chapter 7.

#1 Explain the difference between Positive Gain Error (PGE) and Negative Gain Error (NGE). You can use diagram(s) to illustrate your answer. **[5]**

#2 The ENOB of an ADC can be highly influential in selecting an ADC for an application. If a 12-bit ADC sampling a voltage range -2V to 2V has only an ENOB of 10, would it be able to detect a voltage differential of at least 0.005V between two consecutive reads of the ADC?[2] (Show your working to substantiate your answer! i.e. don't just give Yes or No.) **[5]**

Question 4: Of interfacing and HDL [20 marks]

Suppose that you have been hired to develop an HDL module, called PacketRCV, that needs to read a sequence of 8-bit data bytes that are sent in to the module's DIN port through an 8-bit bus line (as shown Fig(a) in the diagram below). The received bytes need to be placed into an internal BRAM array called PKT (which can store up to 1024 bytes). You don't need to worry about what is done with the received packet (the development company that hired you will take the module further to do some sort of secret processing on it). The process of receiving a packet and what is needed for the implementation is described below, illustrated by the waveform diagram in Fig (b) below.

Receiving packet and other required behaviour of PacketRCV:

- When a new packet is about to be sent to PacketRCV, the SOP (start of packet) line will be pulsed (raising edge indicates when transfer process starts, which involves sending in a sequence of bytes on DIN). The packet PKT array index (PIDX) reg must be set to 0. Output CSYNC set to 1 (and must be set back to 0 right after the next RDY posedge or in any

[1] In case you forgot, you should be thinking $P=C\cdot f\cdot V^2$

[2] Assuming of course that the electronics are set up to map -2V → 2V to the full range of the ADC.

error/reset condition if a RDY doesn't arrive). PTO (packet transfer over) and AHP must be set to 0 also.

- The sending module then sets the bits of DIN with the data to be sent in and sends a pulse on the RDY (ready) line once it is sure the data bits are stable. Immediately at the rising edge of RDY the data needs to be put into the PKT array at index PIDX and then PIDX incremented, also if the AHP is high it must be set to 0 at this point.
- The EOP (end of packet) line is used to signal, at the raising edge, that the complete packet of bytes has been sent and the transfer is complete. At this point the PTO (packet transfer over) line must be raised and then set back to 0 after two (raising edge) clock pulses.
- *[optional:]* If a RDY line or EOP signal is not received after 12 clock pulses (while receiving a packet) then the AHP (Abnormal or Halted Packet) line must be set high and the module must go back to waiting for a SOP (it is extremely important that PIDX is not set to 0 at this point as there may be important data in the array, but as mentioned before PIDX should still be set to 0 after a SOP pulse). Similarly if a SOP is received mid-way while handling a packet AHP needs to be raised, but the rest of the operation must proceed as described to start receiving a new packet. *[**NB:** up to 2 bonus marks for doing this part!!]*

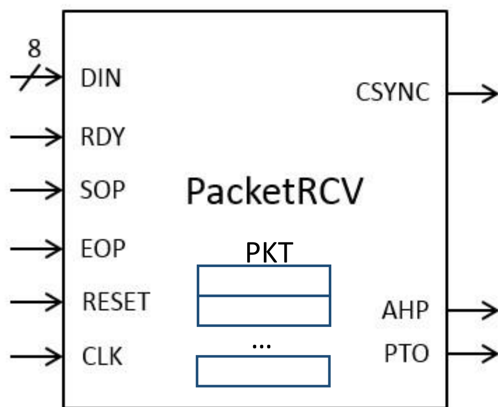

Fig (a) Diagram of the PacketRCV module

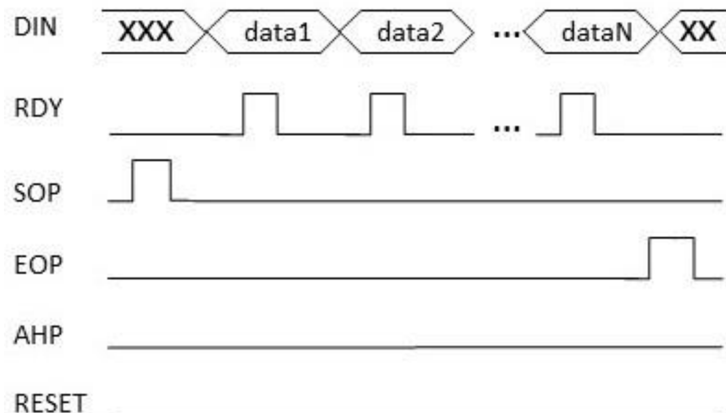

Fig (b) Waveform traces of the input lines to PacketRCV showing the sending of a data packet to the module.

There is furthermore a clock line that is sent to PacketRCV. This line operates at 100Mhz (about 10 times faster than pulses on the RDY line).

TODO: For this question you need to implement PacketRCV the module as described above. You can use either Verilog or VHDL to do this. You do not need to include any libraries (e.g. the tedious lines like "library ieee; use ieee.std_logic_1164.all;" you don't need to bother with). There is a Verilog cheat sheet on the last page.

(Q4 marking will be quite lenient as it might be tricky to implement this all in the time available)

END OF TEST

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)

<u>Concatenation of bits using {}</u>

4’b1011 == {2’b10 , 2’b11}

<u>Constants are declared using parameter:</u>

parameter myparam = 51

Operators

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

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

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

<u>Bitwise ops:</u> and (&), or (|), xor (^), not (~)

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

<u>Bit reduction operators:</u> [n] n=bit to extract

<u>Conditional operator:</u> ? 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/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  );
 … 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
```