ASIC/FPGA Design Spring 2025 Assignment 2

Professor Shabany

Problem 1

Simple FIFO Design

Design a **Synchronous FIFO buffer** with a depth of 16 items. Each item is 8 bits wide, so the module can hold up to 16 bytes of data in order.

- Example: if `wr_en = 1` and `wr_data = 1001_0010` at the rising edge of the clock, the value should be stored in the first FIFO location. Subsequent data are stored in later locations as long as `wr_en` remains active.

Inputs:

- `wr_data` – data to write
- `wr_en` – write enable
- `rd_en` – read enable

Outputs:

- `rd_data` – data being read
- `Empty` – buffer empty flag
- `Full` – buffer full flag

Functional requirements:

- If `wr_en = 1` and FIFO is not full, insert `wr_data`.
- If `rd_en = 1` and FIFO is not empty, output the next FIFO word.

Testbench:

1. Insert several values with `wr_en = 1`.
2. Continue writing until FIFO is full and verify `Full`.
3. Start reading with `rd_en = 1` and check output order.
4. Continue reading until FIFO is empty and verify `Empty`.

Problem 2

Signal Generation and Verification with MATLAB

1. Using MATLAB, generate one full period of a sine wave with length 1024. Convert the samples into **fixed-point format** (word length 16, fractional bits e.g. Q1.16.14). Store sine and cosine values in arrays and use `$readmemb` or `$readmemh` to initialize them in Verilog.
2. Design a Verilog module that uses these sine/cos arrays to generate output frequencies of **1×, 2×, 4×, and 8×** of the primary frequency. A 2-bit input selects which frequency to output.
3. Write a testbench that saves the module outputs to a `.txt` file. In MATLAB, verify outputs by plotting and analyzing the frequency spectrum using `fft` or `pwelch`. Compare expected and generated results.

Problem 3

Xilinx DSP48 Exploration

1. Provide a short explanation of the **DSP48 structure and components** from Xilinx documentation.
2. Design a **complex multiplier** using DSP48 blocks with two 18-bit inputs. Use pipelining for optimization and report the number of DSPs used after synthesis.
3. Repeat synthesis with two 19-bit inputs and explain how DSP usage changes.
4. Verify functionality via simulation.
5. (Bonus) Investigate the role of DSP48 *attributes* in synthesis and show how they can be used to force or avoid DSP usage.

Problem 4

8-bit ALU Design

Design an **8-bit ALU** with:

- Inputs: `i_a` (8-bit), `i_b` (8-bit), `i_cont` (4-bit control), `i_clk` (clock)
- Output: `o_out` (width depending on operation)

Notes:

- This unit must be able to be implemented therefore division can not be used.
- Operations execute on the rising clock edge.
- Operands are unsigned integers (0–255).
- Multiplication outputs only the lower 8 bits unless carry-out is required.
- Addition must include a carry-out bit.

Testbench: Implement a Verilog testbench that checks all ALU operations according to the operation table.

i_cont	Operation
0000	i_a + i_b
0001	i_a – i_b
0010	i_a * i_b
0011	i_a / i_b
0100	Logical Left Shift for "i_a"
0101	Logical Right Shift for "i_a"
0110	"i_a" Rotated Left by One
0111	"i_a" Rotated Right by One

i_cont	Operation
1000	"i_a" and "i_b"
1001	"i_a" or "i_b"
1010	"i_a" xor "i_b"
1011	"i_a" nor "i_b"
1100	"i_a" nand "i_b"
1101	"i_a" xnor "i_b"
1110	If equal, out =1 else out = 0
1111	If "i_a" >"i_b" out = 1 else out = 0

Figure 1: operation table

Problem 5

CRC Module Design
Design a **CRC-16 module** for error detection in a communication channel.

Transmitter (TX)

- For each 16-bit input word, compute CRC using the generator polynomial

$$G(x) = x^{16} + x^2 + x + 1$$

- Append the 16-bit CRC to the word (32-bit total).
- Save the results into an output file.

Receiver (RX)

- For each 32-bit received word, recompute the CRC.
- If correct, set `valid = High` for one cycle and output the 16-bit data.
- If incorrect, set `error = High` and discard the word.

Testing

- Use MATLAB or another tool to generate random test data and CRC values.
- Write a Verilog testbench to feed this data into the module and check correctness.
- Add noise/errors into the input data and verify the `error` signal.

Discussion

Explain why CRC is more effective than simple parity in detecting multi-bit errors.