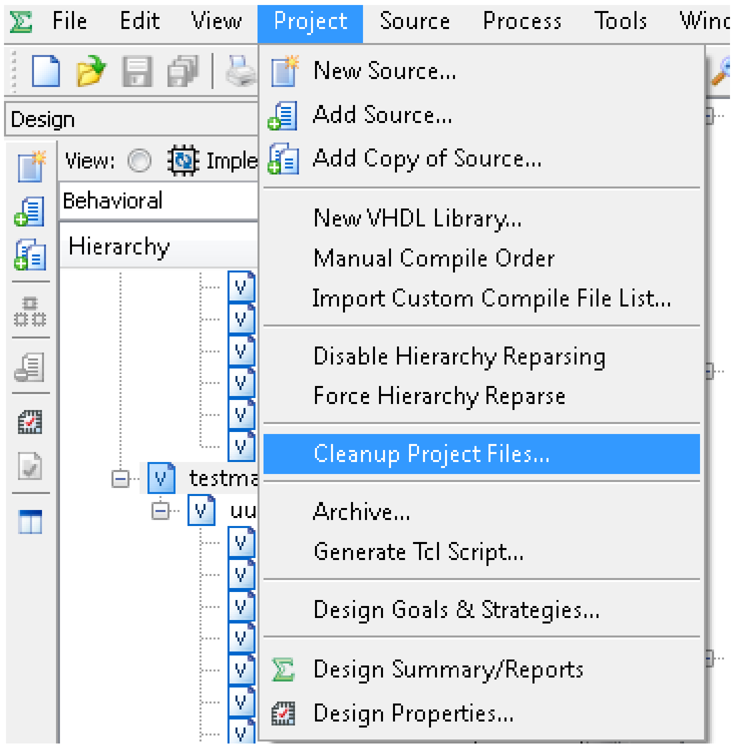
Dear Architects (CPU Architects ☺),
Hope you are doing well; Please pay attention and consider to this issues:

- Implement all modules in **the level specified in the question**.
- Implement all modules in one Xilinx project.
- For each module you have to create **Verilog test fixture** (test bench), If you didn't create test therefore you will lose half of the grade.
- In some questions it is needed first to provide a schematic and label the wire and then implement a circuit in Verilog language. So put your schematic besides your code.
- After finishing your work, it's better to **clean up** your project in Xilinx to reduce the file size.
- 

- Put all of your work in final folder and change the folder name to this format FnameLname_Assignment3.rar/zip and after that rar or zip folder and upload it on the site.
- There is a section in your assignment which labeled as **Bounce**, you can do it to use extra marks.
- If you faced with any problems or question, please feel free to ask your question in the site ASAP.
- Please answer the evaluation section seriously. We need and would like to see your comments.

Hint: To find the schematic it's enough to study your logic book (Mano), Also you can find the electronic version in week 0 of site.

Good Luck

1. Implement a circuit that counts the number of ones in a bit-vector.

```
Test: d = 01001100     , count = 3
Test: d = 0            , count = 3
Test: d = 11           , count = 2
```

```
module #(parameter N=32) ones_count (
   input  wire [N-1:0]          d,
   output wire [???-1:0]        count
);
```

What should be the value in '???' above (hint: see $clog2 function in Verilog).

Implement in behavioral Verilog using a for-loop.
Bonus: can you design a more efficient implementation using data-flow Verilog?

2. Implement a parity-bit calculation circuit. A parity bit circuit counts the number of 1's in a bit-vector, and outputs 0 if the count is even, and outputs 1 if the count odd. It is often used in detecting data corruption.

```
Test: d = 010111010110 => parity = 1
Test: d = 000000000000 => parity = 0
Test: d = 000000001100 => parity = 0
```

```
module #(parameter N=32) parity_calc (
   input wire [N-1:0] d,
   output wire        parity
);
```

Could you do a better job implementing this circuit with data-flow or behavioral Verilog?

Now implement a parity checker. The error signal should be high if the number of 1's in the data plus the parity-bit is odd.

```
Test: d = 010111010110 , parity = 1 => err = 0
Test: d = 010111010110 , parity = 0 => err = 1
Test: d = 000000000000 , parity = 0 => err = 0
Test: d = 000000000000 , parity = 1 => err = 1
Test: d = 000000001100 , parity = 0 => err = 0
Test: d = 000000001100 , parity = 1 => err = 1
```

```
module #(parameter N=32) parity_check (
   input wire [N-1:0] d,
   input wire         parity,
   output wire        err
);
```

Bonus

4. Implement an arbiter. This module should arbitrate between 3 requestors, with fixed priority (req0 is highest priority, req2 is always lowest priority). A grant signal should be sent back to a requestor if its request will be selected. The winner should have its data muxed to the output.

```
module arbiter #(parameter WIDTH=32) (

      input wire req0_v,      // Requestor 0 has a request

      input wire [WIDTH-1:0] req0_data, // Requestor 0 data

      output reg req0_grant, // Requestor 0 wins arbitration

      input wire req1_v, // Requestor 1 has a request

      input wire [WIDTH-1:0] req1_data, // Requestor 1 data

      output reg req1_grant, // Requestor 1 wins arbitration

      input wire req2_v, // Requestor 2 has a request

      input wire [WIDTH-1:0] req2_data, // Requestor 2 data

      output reg req2_grant, // Requestor 2 wins arbitration

      output reg req_out_v, // There is a request winner

      output reg [WIDTH-1:0] req_out_data // Data of arbitration winner

);
```