How not to write a bad report

T J Kazmierski

Report is written to be read!

- Always write for your readers – you know who they are: your examiners
- Know the purpose of your report: academic assessment
- Hence
  - No need to repeat material from notes, data sheets
  - But you must
    - **Demonstrate understanding and knowledge**
    - **Demonstrate results**
      - State what you achieved in the Introduction as well as in the Conclusion

Structure

- **Introduction (short)**
  - What is the problem?
  - How have you solved the problem?
  - What remains unsolved
  - What are your main results?
  - How is the rest of the report organized?

Structure (cont)

- **Technical sections and results**
  - What are the results?
  - Why do they look the way they do? – Analysis!
    - Saying “Modelsim waveforms in fig X show that the sequencer works correctly” is not explaining anything.
  - Be organised: introduce-explain-summarise
  - Be succinct: make your point, then move on.

- **Presentation matters**
  - High-quality technical work requires high-quality presentation
    - Clear figures
    - Annotated code
    - Do not leave much blank space, especially around figures; use effectively the space you have

Structure (cont)

- **Conclusions and further work**
  - Remember: examiners usually read the Introduction and Conclusion - FIRST.
    - In Introduction: tell what you are going to tell (set the scene)
    - In Conclusion: tell what you have told (summarise)
  - State the main points you took away from your work. Show what have you learned. Do more analysis!

Example of clear figure, waveforms explained

The ModelSim simulation (Figure 7) shows the register being reset on the first rising clock edge whilst 'reset' is high, then the register does not change even when 'Sum' is assigned a value until 'shift' goes high, the shifted values are as predicted above.

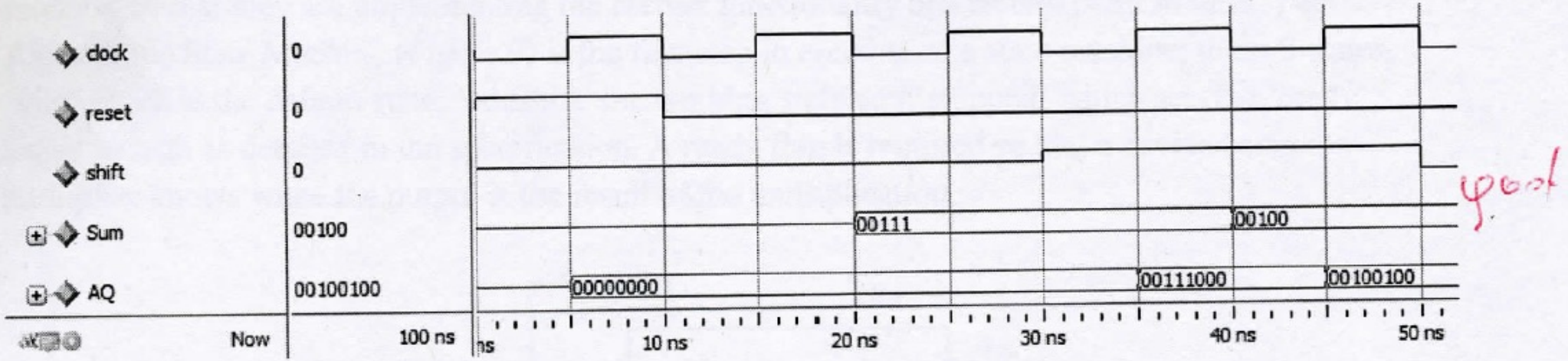


Figure 7: ModelSim simulation of register module

Example of efficient use of space

```
initial
begin
    add = 0;
    shift = 0;
    C = 0;
    Sum = 4'b0000;

    // testing reset
    reset = 0;
    Creg = 1;
    Qin = 4'b1011;
    #50ns reset = 1;
    #100ns reset = 0;

    // testing add
    Creg = 0;
    C = 1;
    Sum = 4'b1111;
    #50ns add = 1;
    #100ns add = 0;

    // testing shift
```

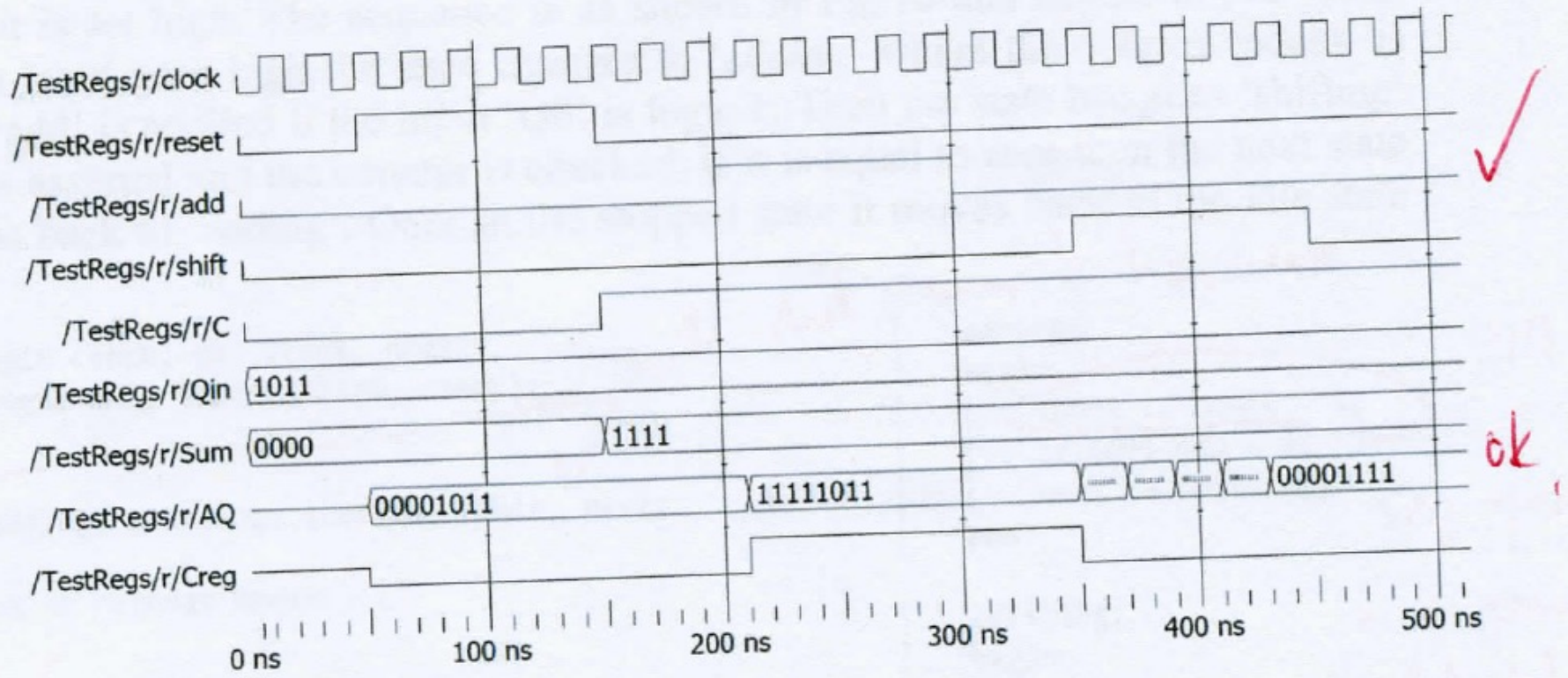


Figure.5: Register simulation waveform

Example of bad figure:

unclear, black background, no explanation what it shows

The following waveform suggests appropriate behaviour (figure 4).

Did it work? How can we tell

Figure 4: Modelsim simulation of the register module.

```
Ln#
 1  module regs_as_ext #(parameter n=8)(input logic clk, reset, add, shift, C, input logic[n-1:0] Q, sum, output logic[2*n-1:0] AQ);
 2
 3  always_ff @ (posedge clk)
 4  if(reset)
 5    begin
 6      AQ[2*n-1:n] <= 0;
 7      AQ[n-1:0] <= Q;
 8    end
 9  else if (add && shift)
10    begin
11      AQ[2*n-1] <= C;
12      AQ[2*n-2:n-1] <= sum;
13      AQ[n-2:0] <= AQ[n-1:1];
14    end
15  else if (~add && shift)
16    begin
17      AQ <= {1'b0, AQ[2*n-1:1]};
18    end
19  endmodule
20
```

Fig5 the code for an n-bit register with add and shift in one cycle

The testbench and simulation results: (fig6 and fig7)

too much

blank space

```
Ln#
 1  module regs_as_ext_t #(parameter n=8);
 2
 3  logic clk, reset, add, shift, C;
 4  logic[n-1:0] Q, sum;
 5  logic[2*n-1:0] AQ;
 6
 7  regs_as_ext #(.n(n)) r(.*);
 8
 9  initial
10    begin
11      clk = 0;
12      forever #20ns clk = ~clk;
13    end
14
15  initial
16    begin
17      reset=1;
18      Q=10;
19      C=0;
20      sum=9;
21      add=0;
22      shift=0;
23
24      #60ns add=1; shift=1; reset=0;
25      #40ns add=0;
26      #80ns add=1;
27      #120ns add=0;
28      #80ns shift=0;
29    end
30  endmodule
31
```

Fig6 the code for the n-bit register (fig5)

illegible!

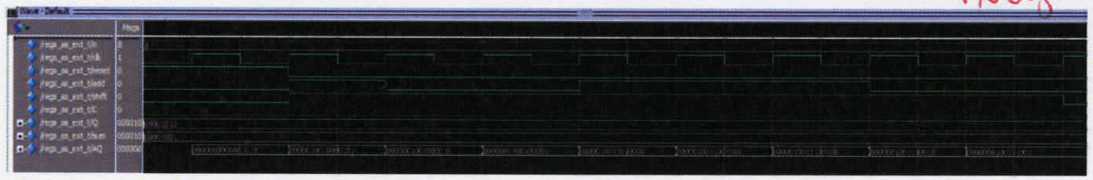


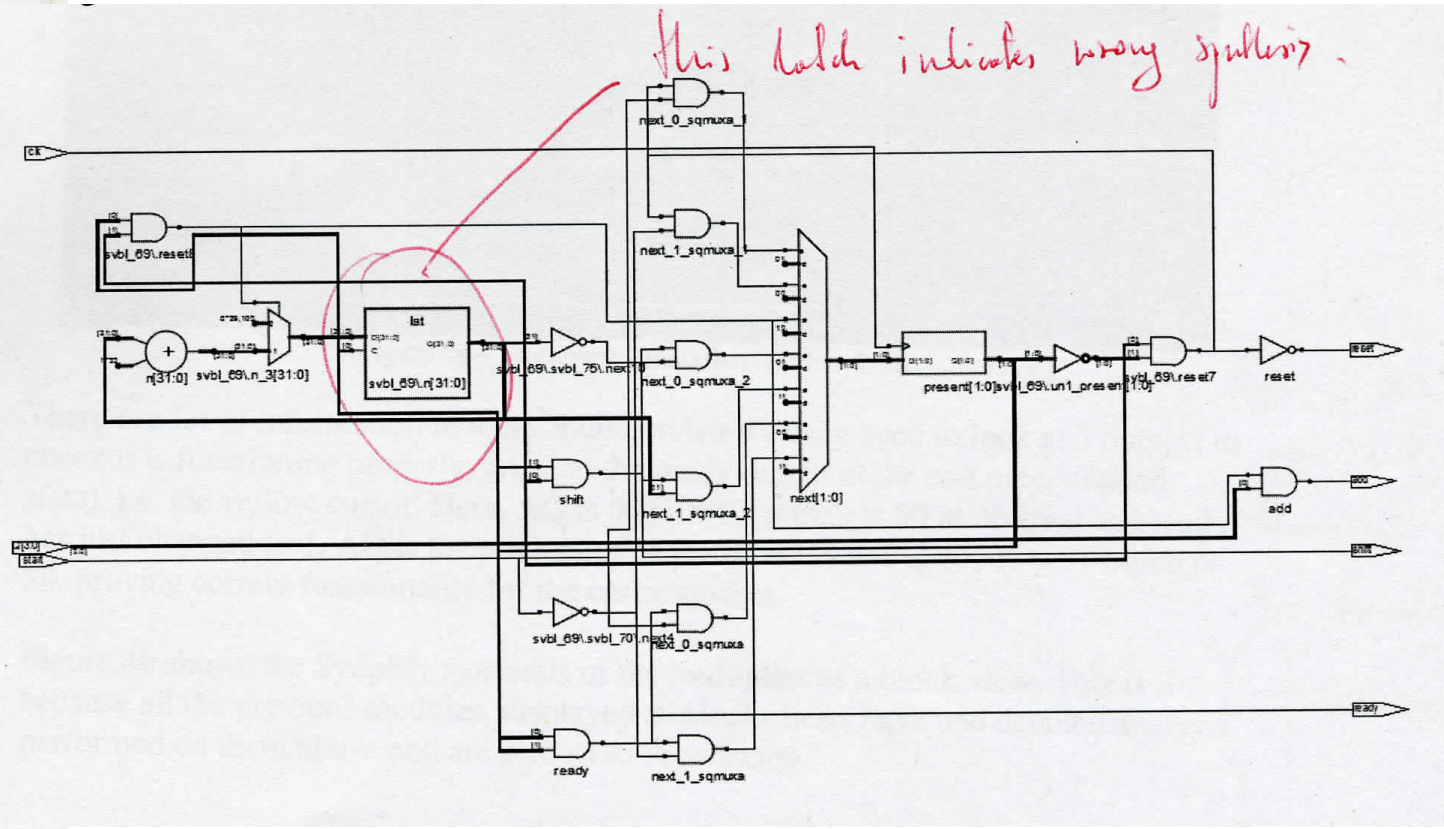
Fig7 the simulation results for the n-bit register (n=8)

```
module sequencer(input logic start,clk, input logic[3:0] Q, output logic
ready,add,shift,reset);

enum logic[1:0] {idle, adding ,shifting ,stopped} present = idle, next;
 int n = 4;
always_ff @(posedge clk)
begin
present <= next;
end

always_comb
begin
  add = 1'b0;
   ready = 1'b0;
   shift = 1'b0;
reset = 1'b0;
  case (present)

  idle:
  begin
    reset = 1'b0;
    if (start)
      begin
        next = adding;
      end
     else
        next = idle;
     end

    adding:
    begin
     reset = 1'b1;
     add = 1'b0;
     ready = 1'b0;
     shift = 1'b0;
      n = n-1;
      if (Q[0])
        begin
          add = 1'b1;
        end
     else
     begin
       add = 1'b0;
     end
     next = shifting;
    end

    shifting:
```

better to use logic not int

this is a sequential operation and should be implemented in an always-ff block to infer a counter in h/w.

As it stands, the synthesiser h/w will most likely not work. Always_comb is for logic only.



always_comb is for combinational logic only!

```
// correct counter implementation:
logic[2:0] n;
always_ff @ (posedge clk)
 if (present== idle)
          n <= 4;
 else if (present == adding)
          n <= n-1;
```