# Contents

1. **Pulse Generator (4 points, total)**  
   1.1 Design It (3 points)  
   1.2 Show a Timing Diagram for your design (1 points)  

2. **Harvard Class? (2 points, total)**  
   2.1 How Does the Harvard design work? (1 point)  
   2.2 How does our non-Harvard design work? (1 point)  

3. **Memory Address Decode, with gates or decoder (3 points)**  

4. **Memory Address Decode...done with Verilog (2 points)**  

5. **Lab μ1 Test Program (2 points)**  

6. **Modify a Counter, using Verilog (5 points)**  
   6.1 Draw a modified design (1 point)  
   6.2 Modify our Verilog design (we suggest you start with ours) (4 points)  
   6.2.1 Testbench  

7. **Modify the design further: make it Up/Down (4 points)**  
   7.1 Modify the 3-bit design to make it an UP/Down counter (2 points)  
   7.2 Modify and run an UP/Down Testbench (2 points)
1 Pulse Generator *(4 points, total)*

We want to keep practicing the old skills, as we move into new territory, with the microcontroller. So here’s an “old” topic—but not necessarily easy because of that.

1.1 Design It *(3 points)*

You are given a clock signal, a square wave. When a button is pressed, allow exactly one full positive clock pulse through. Before and after the pulse, the output should be zero. No partial pulses permitted. (I don’t find this easy.)

1.2 Show a Timing Diagram for your design *(1 points)*

Complete the timing diagram begun below, showing your circuit’s output and enough signals internal to your design so that we can understand how you achieved your result. Make your “sweep rate” high enough so that one can see a gate or flip-flop delay.
2 Harvard Class? (2 points, total)

The 8051 is exceptional among contemporary processors (well, it was born a long time ago...): it is a so-called “Harvard class” computer, because it assigns separate memory spaces to Code, on the one hand, versus Data, on the other. The figure below (taken from a Dallas application note) shows a diagram of a “typical” 8051 configuration, using external ROM and RAM. The ROM and RAM even share addresses—yet they do not conflict. In case you’re puzzled by this circuit, you’ll find some discussion of this figure at http://www.maxim-ic.com/appnotes.cfm/an_pk/57, the note from which the figure is taken).

2.1 How Does the Harvard design work? (1 point)

How is it that the two memory IC’s—one for code and the other for data—do not conflict?

2.2 How does our non-Harvard design work?) (1 point)

Our ’Big Board’ computer is NOT a Harvard class computer, though it uses an 8051. What in our 8051 circuit’s wiring makes this so? (For us, just one RAM houses both code and data). You may want to consult the “Big Picture” full-circuit diagram.
3 Memory Address Decode, with gates or decoder (3 points)

Suppose that instead of a single 32K RAM (which is what your machine has), we had to make do with 4 of the smaller 8K RAMs that we used to rely on.

Show how to do that, using either a ’139 decoder or gates. Assume that you start with signals that are enough to say ‘turn on memory.’ Assume that we turn on memory in response to any of the following signals:

- RD\(^\ast\) (meaning ‘Data read’)
- WR\(^\ast\) (meaning ‘Controller write’)
- PSEN\(^\ast\) (meaning ‘Code read’)
- BR\(^\ast\) (in our machine, this means ‘the humans want control’)

(In this problem we have departed slightly from the wiring of the lab computer, in requiring A15 to be low during BUSRQST\(^\ast\) enabling.)

- A15 low and either
- BUSRQST\(^\ast\) or
  - PSEN\(^\ast\) (a signal that means “program read”) or
  - WR\(^\ast\) or RD\(^\ast\) (signals that indicate a data transfer)

Figure 3: ’139 decoder, used for address decode
4 Memory Address Decode...done with Verilog (2 points)

This time, we have not set up active-high signals for your convenience. You can do that yourself, or you can write your equations paying attention to active levels. The variable names indicate which signals are active-low.

You can do this with paper and pencil, if you like, rather than fire up Verilog. AND is &; OR is |, NOT is ! or tilde.

```verilog
module hw_adr_decode(
    input ,// you choose the appropriate address lines
    input ,
    input a15,
    input psen_bar,
    input rd_bar,
    input wr_bar,
    input br_bar,
    output cs0_bar,
    output cs1_bar,
    output cs2_bar,
    output cs3_bar
);

assign cs0_bar = ;
assign cs1_bar = ;
assign cs2_bar = ;
assign cs3_bar = ;
endmodule
```
5  Lab μ1 Test Program (2 points)

Modify the Big Board test program slightly, as follows:

- let the program loop begin at address 28h instead of 10h;
- let the program loop be “NOP, NOP, SJMP start” (in other words, include one additional NOP within the loop).

Please show hexadecimal codes and offsets as well as the usual assembly language. (Usually, when we ask for a “program,” we mean only assembly language; this is an exceptional case.)

We’ve reproduced, here, the code that appears at the end of Big Board Lab μ1:

MACRO ASSEMBLER FIRST_TEST

<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>000E</td>
<td>5</td>
<td>ORG 0 ; tells assembler the address at which to place this code</td>
</tr>
<tr>
<td>0000</td>
<td>800E</td>
<td>5</td>
<td>SJMP DO_ZIP ; here code begins--with just a jump to start of</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>real program. ALL our programs will start thus</td>
</tr>
<tr>
<td>0010</td>
<td>00</td>
<td>8</td>
<td>ORG 10H ; ...and here the program starts, at hex 10 (“...h”)</td>
</tr>
<tr>
<td>0010</td>
<td>00</td>
<td>9</td>
<td>DO_ZIP: NOP ; the least exciting of operations: do nothing!</td>
</tr>
<tr>
<td>0011</td>
<td>80FD</td>
<td>11</td>
<td>SJMP DO_ZIP ; ...and do it again!</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13</td>
<td>END</td>
</tr>
</tbody>
</table>
6 Modify a Counter, using Verilog (5 points)

Here, we’d like you to try using flip-flops in Verilog. We’ve provided (that is, we’ve posted) a very simple two-bit synchronous up counter design in Verilog. There is no carry-in or carry-out; just 2 bits, with an asynchronous reset. You know how to design such a thing, but we’ve posted it to spare you some of the fussy details.

6.1 Draw a modified design (1 point)

Now draw a 3-bit version synchronous DOWN counter with synchronous RESET*. Draw a design, starting with D flops.

6.2 Modify our Verilog design (we suggest you start with ours) (4 points)

Now write a Verilog design for the 3-bit synchronous Down counter with synchronous reset. We suggest that you download the file “two_bit_simplest_ctr.v” from our website. (It’s shown below.) Then modify it as necessary. Strictly a RESET* should force a DOWN counter to its all-ONE’s state, rather than ZEROes.

Run your design with the testbench that is posted, and print out the result, to submit with your HW. Don’t forget to make the internal file references match your filenames. The testbench, for example, refers to

```verilog
// Instantiate the Unit Under Test (UUT)
three_bit_down_sync_clear uut (..)
```

If your design is not called “three_bit_down_sync_clear” you’ll need to change that name or the reference, to make them match.
else
count <= count +1; // clock really IS edge-sensitive
endmodule

6.2.1 Testbench

Here is a testbench for the 3-bit DOWN COUNTER (the test bench works equally well for a down or up counter). You can try it with your design.

`timescale 1ns / 1ps

////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
// Create Date: 17:52:30 04/02/2012
// Design Name: three_bit_sync_clear
// Module Name: Y:/Desktop/PLD_xilinx/counters/three_bit_sync_clear/three_bit_sync_clear_tb.v
// Project Name: three_bit_sync_clear
// Target Device:
// Tool versions:
// Description:
///
// Verilog Test Fixture created by ISE for module: three_bit_sync_clear
// Dependencies:
///
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
///
////////////////////////////////////////////////////////////////////////////
module three_bit_sync_clear_tb;

// Inputs
reg clk;
reg reset_bar;

// Outputs
wire [2:0] count;

// Instantiate the Unit Under Test (UUT)
three_bit_sync_clear uut (
  .clk(clk),
  .reset_bar(reset_bar),
  .count(count)
);

initial
$monitor("clk = %b, reset_bar = %b, count=%b", clk, reset_bar, count);

initial begin
  // Initialize Inputs
  clk = 0;
  reset_bar = 1;

  // Wait 100 ns for global reset to finish
  #100;

  // Add stimulus here
  repeat (4)
#10 clk = ~ clk;
#10 reset_bar = 0; // force to zeroes, for UP counter, to all one’s for DOWN counter
repeat (2)
#10 clk = ~ clk;
#10 reset_bar = 1;
repeat (20)
#10 clk = ~ clk;
end
endmodule

7 Modify the design further: make it Up/Down (4 points)

7.1 Modify the 3-bit design to make it an UP/Down counter (2 points)

Modify the 3-bit counter so as to let it count UP or DOWN. Add one more input, named UP, which makes this selection. You can do the job, then, by adding one more IF...ELSE pair.

7.2 Modify and run an UP/Down Testbench (2 points)

Modify the 3-bit testbench file of § 6.2.1 on the preceding page to suit your up/down design. Here are some steps you’ll need:

- You’ll need to add UP to the set of signals; make it type REG;
- add its ’instantiation’ to the list already present in the testbench above;
- in the “stimulus” section, you should specify the level of UP;
- then, after testing reset_bar, give ten rising edges as in the existing bench;
- then try the other level of UP and give another ten rising edges.

Please print out the source file and the simulation result for your Up/Down counter.

(hwd3_apr15.tex; April 6, 2015)