1 Sampling (5 pts)

A telephone answering machine doesn’t need excellent sound quality, because the original telephone signal is pretty crummy.\textsuperscript{1} Telephone signals carry no information above about 3kHz, and the amplitude “fidelity” or accuracy doesn’t have to be very good. Here, we’d like you to help specify an ADC converter for an answering machine that is to store phone messages in digital form, then play them out through a DAC.

\textsuperscript{1}Also known as “crumby.”
1.1 **Number of Bits (1 pt)**

In order to resolve the signal to $\pm 1\%$ of full-scale, how many bits are required?

1.2 **Sampling Rate (2 pts)**

What sampling rate do you recommend? Assume that the output filter that will smooth the recovered analog signal is “steep” enough to reduce amplitude by a factor of at least 100 for frequencies 20% above the “cutoff” frequency, or $f_{3\text{dB}}$. Explain your choice, briefly.

1.3 **Unexpected Noise (2 pts)**

If your circuit includes no anti-aliasing filter, and strong electrical noise appears at the ADC input, at about 5kHz, will this noise be eliminated by your reconstruction filter (that is, a low-pass filter applied to the DAC output)? Please give a quantitative argument to support your answer, and if the effect of noise survives, at what frequency does it appear?
2 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.

2.1 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.)

2.2 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.

Figure 1: Timing diagram to explain your circuit design
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* (meaning 'Data read')
- WR* (meaning 'Controller write')
- PSEN* (meaning 'Code read')
- BR* (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* enabling.)

- A15 low and either
- BUSRQST* or
  - PSEN* (a signal that means "program read") or
  - WR* or RD* (signals that indicate a data transfer)

Figure 2: '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 (tilde flips each bit of a multi-bit variable; if the variable is a single bit, then “!” and “tilde” are equivalent).

```verilog
'timescale 1ns / 1ps

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 20 Test Program (2 points)

Modify the test program slightly, as follows:

- let the program loop begin at address 38h 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 lab 18:

MACRO ASSEMBLER FIRST_TEST 03/01/101 20:09:27 PAGE 1
LOC OBJ LINE SOURCE
0000 0 1 ; FIRST_TEST.A51 Lab 18: confirm that the circuit is a computer!
0003 3 ORG 0 ; tells assembler the address at which to place this code
0006 800E 5 SJMP DO_ZIP ; here code begins--with just a jump to start of
0009 7 ; real program. ALL our programs will start thus
0010 8 ORG 10H ; ...and here the program starts, at hex 10 ("...h")
0010 00 8 DO_ZIP: NOP ; the least exciting of operations: do nothing!
0011 80FD 11 SJMP DO_ZIP ; ...and do it again!
0013 END

6 Compiling the Glue PAL Design

We’d like you to implement the design that you did last week. To help with the compiling, we provide a self-checking test bench that will flag any cases where your logic doesn’t do what (we think-) it should.
6.1 Getting Verilog

Verilog is available, free, from the manufacturer of the PLD’s that we use. We will post a link to the site where you can get it. (Under pages/miscellany). The downloadable file is huge—several Gig, expanding to more (6 or 7) when installed. If you don’t want such a thing on your computer, just do the Verilog exercises on one of our lab laptops. (That’s the way Tom would do it; but some people like to have Verilog at home.)

6.1.1 Windows and Linux, but not Mac

Verilog comes in those two versions only. It can run on a Mac that uses a Windows emulator, such as Parallels.

We asked you last week to design little bite-sized fragments, using pencil and paper. This week, we’d like you to put your work into a file, which we’d like you to compile and simulate. This process is rather mechanical, but will introduce you to Verilog.

6.2 Note on Notation: Active-high versus Active-low

In a template Verilog file we have defined an active-high equivalent for every signal that is active-low. For example,

```verilog
// make all signals active-high
// inputs
assign br =˜BR_bar;
assign kreset =˜KRESET_bar;
assign kwr =˜KWR_bar;
assign ˜RD_bar = rd; // this may look backwards, but the active-low signal,
                  // the one that reaches the outside world goes on the left for an output

br = kwr | rd;
```

Doing this allows you to forget about whether any particular signal is active-high or active-low—so long as you write your equations for the active-high versions. So, if you wanted the active-low BR_bar signal to be asserted if KWR_bar or RD_bar is asserted, you could write

```verilog
br = kwr | rd;
```

Below we show the section of a Verilog file in which active-low variables are given active-high equivalents:

```verilog
// make all signals active-high
// inputs
assign br =˜BR_bar; // this is a "bus request" signal that indicates the humans
                  // have taken control of the computer’s buses, in place of the processor
                  // inputs
assign kreset =˜KRESET_bar;
assign kwr =˜KWR_bar;
assign rd =˜RD_bar;
assign wr =˜WR_bar;
assign psen =˜PSEN_bar;
assign in0 =˜IN0_bar;
assign loader =˜LOADER_bar;

// outputs
```
assign CTR_OE_bar = `ctr_oe;
assign KBUFEN_bar = `kbufen;
assign IOR_bar = `ior;
assign IOW_bar = `iow;
assign RAMWE_bar = `ramwe;
assign RAMOE_bar = `ramoe;
assign RAMCE_bar = `ramce;
assign DEMUXOE_bar = `demuxoe;

// ------------------
// this just for convenience
wire write_ok, full_speed;
// assign write_ok = ; // this is RAM address above 2K---you needn’t use this if you don’t want to

The last two lines can serve a different purpose: simply a short-hand reference that may make code more readable than otherwise. “full_speed” is used in the single-step logic (and you need not be concerned with that logic). “write_ok” is listed because you might want to define it, as a function of some address lines (defining the space above 2K, in the RAM’s 32K address range). But you are not obliged to use this. Just comment it out, if you don’t need it.

7 Put it all into one file and compile it (3 points)

Use the template file, stepglue_MT_nov12.v, which is posted on the course website, inserting into it your equations. When the file ceases to be empty, you could remove the “MT” from the filename. In any case, make sure that the filename matches the name of its testbench (apart from the “.tb”), a file that is posted as “stepglue_MT_nov12_tb.v.

All you need do is transfer the several handwritten equations you’ve already done—then compile it with Verilog. The simulation message will tell you whether you’ve done it right.

Please hand in your source file (.v) and a printout of your simulation results (which we hope will say ‘no errors.’ The template file (though lacking the testbench) appears at the end of this homework. The full template file, including self-checking testbench, is posted on the class website, so YOU NEED NOT TYPE IN THIS TEMPLATE FILE!

Template File for GLUE PAL

// stepglue_MT_nov12.v
// self-checking glue step, one function at a time
// add manual-write intermediate variable "OK_KWR" oct 2011

// Translated Verilog Source File
// Auto-Generated By Xilinx’s Blf2lang.
// Copyright (c) 2003. Xilinx, Inc.

//step and glue for 8051 lab computer, Xilinx version, Apr. 09, testbench slightly revised April 2011

timescale 1ns/1ns

module stepglue_MT_oct2011 ( input BR_bar,
                   input KRESET_bar,
                   input KWR_bar,
                   input RD_bar,
input WR_bar,
input PSEN_bar,
input IN0_bar,
input LOADER_bar,
input A15,
input A14,
input A13,
input A12,
input A11,
input CLK_IN,
input TRIG,
input STEP_RUNBAR,
input ALE,
output CTR_OE_bar,
output RESET51,
output KBUFEN_bar,
output IOR_bar,
output IOW_bar,
output RAMWE_bar,
output RAMOE_bar,
output RAMCE_bar,
output reg PSENRDY_bar,
output DEMUXOE_bar,
output CLK_OUT, // this one is wire, because used in continuous assignment
output reg TRIG_LATCH,
output reg TRIG_SYNC,
output reg ALE_LATCH,
output reg DELAY
);

// Now, a set of signals introduced just to allow us to write equations in a pure active-high world
wire br,kreset,kwr,rd,wr,psen,in0,loader,ctr_oe,kbufen,ior,iow,ramwe,ramoe,ramce,deluxoe;

//--
//-- make all signals active-high
//-- inputs
assign br = ~BR_bar;
assign kreset = ~KRESET_bar;
assign kwr = ~KWR_bar;
assign rd = ~RD_bar;
assign wr = ~WR_bar;
assign psen = ~PSEN_bar;
assign in0 = ~IN0_bar;
assign loader = ~LOADER_bar;

//-- outputs
assign CTR_OE_bar = ~ctr_oe;
assign KBUFEN_bar = ~kbufen;
assign IOR_bar = ~ior;
assign IOW_bar = ~iow;
assign RAMWE_bar = ~ramwe;
assign RAMOE_bar = ~ramoe;
assign RAMCE_bar = ~ramce;
assign DEMUXOE_bar = ~deluxoe;

//--
//-- this just for convenience
wire write_ok, ok_kwr, full_speed;
assign write_ok = (A14 | A13 | A12 | A11); // this is RAM address above 2K
assign ok_kwr = br & kwr;

//-- Logic implementation...
assign ctr_oe = // loader defeats manual turn-on of counter 3-states
assign ramce = // bottom half of address space--our manual control--turns on RAM
assign ramoe = // three conditions for reading memory: pgm rd, data rd, manual access
assign ramwe =
assign kbufen =
assign iord =
assign low =
    assign RESET51 =
assign demuxoe =

// now comes three-state turned on by loader
// assign PSENDRV_bar = (loader) ? 1'b0 : 1'bz; // if loader is asserted, drive low; otherwise, float PSENDRV_bar

always @(loader)
    if (loader)
        PSENDRV_bar = 1'b0;
    else
        PSENDRV_bar = 1'bz;

// ---------------------------------------------------------------
// here's the single-step logic

initial
    DELAY = 1'b0; // this initialization makes the sim work

always @(negedge ALE, posedge DELAY)
    if (DELAY)
        ALE_LATCH <= 1'b0;
    else
        ALE_LATCH <= 1'b1;

always @(posedge TRIG, posedge DELAY)
    if (DELAY)
        TRIG_LATCH <= 1'b0;
    else
        TRIG_LATCH <= 1'b1;

always @(negedge CLK_IN)
begin
    TRIG_SYNC <= TRIG_LATCH;
end

always @(negedge CLK_IN)
    DELAY <= (TRIG_SYNC & ALE_LATCH);

assign full_speed = (!STEP_RUNBAR | !BR_bar | !LOADER_bar | !KRESET_bar);
assign CLK_OUT = (CLK_IN & full_speed) | ((TRIG_SYNC & CLK_IN) & !full_speed);
endmodule

8 Optional Burner Bonus (3 points)—do it anytime; just show one of us

We’d like you to see how PAL burning works, so we offer a 3-point bribe to anyone who burns his GluePal design into a device. (Do it only after your design has passed its simulation.) You can do this anytime.

(hwd3.nov17.tex; October 29, 2017)