REV 0[1]; November 16, 2004

HW D2: Pal for Glue Logic: Fall '04

Question	Points
A. Odds and Ends (7 pts + 1 extra)	
1. Two Switch Debouncers	2 pts + 1
2. Counter Application	3 pts
3.. Counter Timing Diagrams	2 pts
D1 GLUE PAL (17 pts)	
4.1 RAM enable	1 pt
4.2 RAM 3-state enable	1 pt
RAM write:	
4.3 Write protect (processor write)	2 pts
4.4 Manual write	1 pt
4.5 RAM write, combined	2 pts
4.6 Keyboard buffer enable	1 pt
4.7 I/O enable	2 pts
4.8 8051 Reset	1 pt
4.9 PSEN drive	1 pt
4.10 Demux 3-state	1 pt
4.11 Put it all in a file, and compile wi ABEL	4 pts
TOTAL	24 pts

Total points: 24.

Due Monday, November 22, 2004

Intro

After a little introductory stuff, this homework jumps right into two tasks important to this course, for practical reasons: one, to introduce you to the *logic compiler* **ABEL** that we use to program PALs (or "PLD's," *programmable logic devices*, to speak more generically); two, to start you implementing the "glue" logic that will link the brains of your little lab computer to the memory and peripheral devices that make the thing a useful circuit—truly a *computer*.

[1]Revision: combined counter intro wi glue hw (D2, D1 from previous term) (11/04);intermed. vars. in template commented out, with explanatory note (4/03).

1 Switch Debouncers (2 pts, + 1 extra)

1.1 SPST

Show how to debounce a switch of the *single-pole, single-throw* form shown below:

Figure 1: Debouncer for SPST switch

(There's more than one way to do it.) *Two* ways, for extra credit.

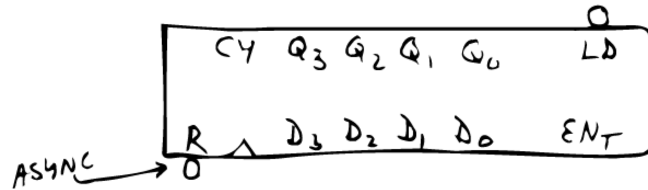
Show how to debounce a switch of the *single-pole, double-throw* form shown below:

Figure 2: Debouncer for SPDT switch

(There's more than one way to do it.) *Two* ways, for extra credit (please don't show any debouncing method *more than once!*

2 Counter Applications: Sync vs Async Function (3 points)

Use a 74HC161, a 4-bit binary up counter with synchronous load, asynchronous clear, to make a "divide-by-eleven" counter in several ways:



Note:

- CY is high if Count = 15, and if EN_T is asserted; CY is low otherwise;
- the counter simply holds its present state if EN_T is not asserted;
- "Synchronous Load": Data (at the 4 "D" inputs) is loaded into the counter on the next rising edge of the clock if LD* is low (with 2ns setup time—yes, just two ns)

2.1 Crummy: async clear (1 point)

Show how to use the asynchronous clear to make a crummy divide-by-eleven counter: By the way, what's so

crummy about this design?

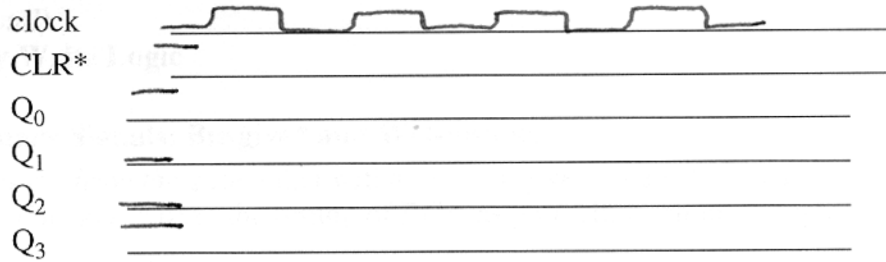
2.2 Good: fully synchronous divide-by-eleven (2 points, total)

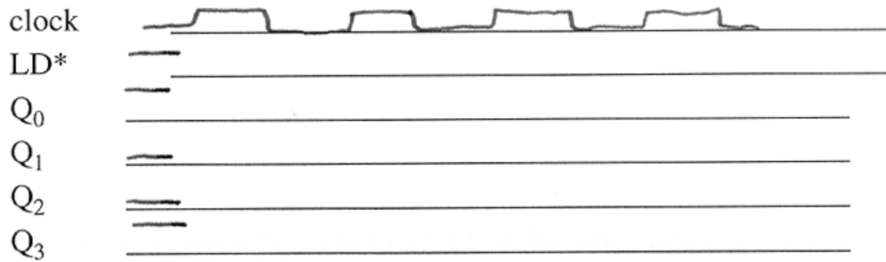
1- Let the counter run up from state zero (1 point)

2- Use any states you like, taking advantage of the CarryOut function (1 point)

3 Timing Diagram of Synchronous vs Asynchronous Schemes (2 points)

In the preceding question, you used both an asynchronous (crummy) method, and a fully-synchronous (classy) method. Your counters, in two cases, ran from zero up, and were to divide by 11. Use the timing diagrams below to show how the sync version is better than the async. Let your timing diagrams begin near the end of the cycle: at count **9**, and show what happens in the next **4** clock periods.





4 GLUE PAL

This homework asks you to implement the first of two PALs required by your lab computer

Today’s PAL (the “GLUE” PAL) forms the glue logic that well need to join a microcontroller—the brains of a little computer—to the memory and other odds and ends that will let the whole circuit serve as a small computer. Glue is jargon for the small bits and pieces of logic that join large functions, like CPU and memory. Normally, its pronounced with a slight sneer. PLD/PALs are good at implementing this kind of logic, and youll find nearly all the glue fits on one 22V10.

The other (the “STEP PAL”) implements the single-step logic that will allow us to watch our little computer run in slow motion—or, more accurately, run in “freeze-frame”. That PAL we will postpone till next homework.

Well ask you to design little bite-sized fragments, using pencil and paper, and then to put your work into an ABEL file, which wed like you to compile and simulate.

In a *template* file we have assigned variables to pins. This appears at the end of this homework, and is posted on the class website. Youll notice that **we have banged every pin that is active-low**: such variables show an exclamation point that defines them as active low. The address lines do not show a !, because they are not active low: they are as true when high as when low. We state this here just to remind you that as you write equations, below, we want you to assume the active levels have already been assigned, in the pin declarations.

Note on naming of variables:In the real circuit, if, for example, you were to probe WR*, you would find it low when true or asserted. In this homework, however, because all active-low pins are banged in pin declarations, we refer to the signal WR* by its variable name, WR: that is, we use the name of the variable as it appears within the PAL.

Below we show the section of the ABEL file in which variables are assigned to pins.

```
"inputs

      !BUSRQST   pin 1;
      !KRESET    pin 2; // This is inverted signal from keypad (squared up & debounced by HC14)
      !KWR       pin 3;
      !RD        pin 4;
      !WR        pin 5;
      !PSEN      pin 6;
      !INO       pin 7;
      !LOADER    pin 8;
      A15        pin 9;
      A14        pin 10;
      A13        pin 11;

      A12        pin 13;
      A11        pin 14;


"outputs

      RESET51    pin 23 ISTYPE 'com';  // Here is the ONLY active-high output!
      !KBUFEN    pin 22 ISTYPE 'com';
      !IOR       pin 21 ISTYPE 'com';  // these enable the two halves...
      !IOW       pin 20 ISTYPE 'com';  // ...of the I/O decoder
      !RAMWE     pin 19 ISTYPE 'com';
      !RAMOE     pin 18 ISTYPE 'com';
      !RAMCE     pin 17 ISTYPE 'com';
      !PSENDRV   pin 16 ISTYPE 'com';  // During boot load, this forces PSEN* low
      !DEMUXOE   pin 15 ISTYPE 'com';  // enables demux during 1) run, 2) boot load
```

This list is not very interesting—except in one respect: you *need to take note of the fact that all* ***active-low signals*** *are* ***banged*** in their pin declarations. This is *good news* for you, as you write equations: it means that for the purpose of your equations, *you should treat ALL signals as active HIGH.* We hope you’re pleased! If you’re not, then you haven’t yet appreciated how troublesome keeping track of active levels can be.

4.1 RAM enable (1 point)

And heres the RAM enable logic we want to implement with ABEL: enable the RAM (drive its CS* or CE* pin ((chip select or chip enable), here called by the variable name RAMCE)) if BUSRQST* is asserted or if A15 is low (indicating memory address range, rather than I/O—input/output). Assume that all variables have been banged in the pin declarations, except A15, which is NOT an active-low signal, since it is equally asserted or true in both low and high states.

RAMCE =

4.2 RAM 3-state Enable (OE*) (1 point)

The RAMs output enable, OE*, should be enabled if any of the following signals is asserted: PSEN, RD, or BUSRQST. (PSEN is a timing signal indicating an instruction- fetch; RD is similar, but used for a data read; BUSRQST signal that we use to disable the microcontroller outputs so that we can turn on memory and load it by hand). Again assume all signals, including the output, RAMOE, have been banged in the pin declarations.

RAMOE =

4.3 RAM Write-Protect (2 points)

We want to block or frustrate any attempt by the controller to write to low addresses; this is to protect our code from runaway programs that might destroy the fruits of our labor. Specifically, we want to permit a controller write only at addresses above the bottom 2K, in the RAMs 32K memory space. How many address lines will the 32K memory use?

What is the hexadecimal address of the first location above the bottom 2K?

What address lines would you require to be high, therefore, before you would permit a controller write to go through?

4.4 RAM manual-write enabling (1 point)

We want to be able to write into RAM by hand—but only at times that are safe: not when the computer is running a program. We'd make a mess of things if we mistakenly pushed a manual write key while a program was running and overwrote code or data. So, we want to permit a manual write—signaled by KWR, from the keypad—only if **BUSRQST** is also asserted. Lets call this permitted manual write OK_KWR. (It is an intermediate variable, not brought out to any pin, so we need not trouble ourselves with the thought that it might be active low. Show the equation for this intermediate variable, assuming as usual that the relevant signals have been banged in pin declarations.

OK_KWR =

4.5 Complete (combined) RAM write-enable (WR*) (2 points)

Now lets put together the two branches youve just designed: controller write and manual write. Show the equation for the RAMs WE* pin, whose variable is named RAMWE. Again assume that all the active-low signals have been banged in pin declarations. The two branches are:

1. controller write—which is the WR signal, conditioned by the write-protect logic that you described, above, and
2. The manual write. (Please show, once again, the constituents of OK_KWR, rather than use that intermediate variable in your equation.)

And we would like you to modify the write-enabling as follows:

- When a signal named LOADER is asserted, the Write-Protect feature should be disabled. That is, an 8051 WR should generate a RAMWE signal, *regardless of address*. (This behavior will allow our automated program-loader to install a program anywhere in memory.)

RAMWE =

4.6 Keyboard Buffer Enable (KBUFEN*) (1 point)

A few labs down the road, well want to be able to let the controller read a keypad value. So, well want to permit either of two events to turn on its 3-state: a manual write, or a controller input at port 0—an IN0* signal. Thus:

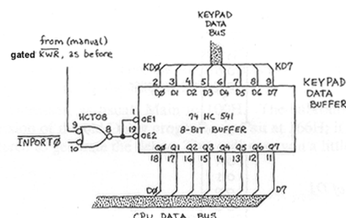


Figure 3: Logic that your KBUFEN function is to implement

Please write the ABEL equation for the OE* signals, here called by the variable name KBUFEN.

KBUFEN =

4.7 Two I/O Enables (IOR, IOW) (2 points)

These two signals indicate an I/O operation. We want one for each of the two possible directions: into the computer (an "input," an IORead), and out of the computer (an "output," an IOWrite). We will use these signals in Lab 19, as part of the enabling of an "I/O decoder" IC. The following are the two signals:

- Both signals—that is, all "I/O" or "input-output"—are asserted only when an address falls into the upper half of all address space: that is, where address line A15, the most-significant bit, is *high*.
- IOR: IOR should be asserted on an assertion of RD (indicating a data read) or PSEN (indicating a code or "instuction" read).
- IOW: IOW should be asserted on WR (no write-protect is involved, here).

IOR =

IOW =

4.8 Processor Reset (RESET51) (1 point)

This is a little strange, but turns out to be convenient: we let any one of *three* signals assert the processors RESET pin: either

1. A RESET from the keypad (debounced and inverted by an HC14), (the resulting, debounced, signal is named KRESET; it is active-low, but you need not worry about that, as usual: all active-low signals have been "banged" in the pin-declarations); or
2. BUSRQST (busrequest) from the keypad (this signal sort of puts the processor off duty); or
3. LOADER: this is a strange one, but it turns out that the code-loading-from-PC feature requires assertion of the processor's RESET line. (We won't use this feature till the very end of the course; but we want to set things up so we *can* use it.)

We drive the 8051's RESET this way because the RESET does double duty for us, as we may have mentioned: it resets the processor, forcing it to start up anew, on release of RESET; but it also makes the processor turn off its 3-states, which permits us to take control of the buses, as we do during a bus-request (fancy name: direct memory access). (BR* does another job at the same time, turning on our '469 counters 3-states.) Assuming *banged pins* as usual, write the ABEL equation for RESET51:

RESET51 =

4.9 DEMUX 3-state control (1 point)

The 8051 uses an annoying trick to save pins: it time-shares 8 lines between data and address. A '573 *octal transparent latch* "de-multiplexes" these lines, and needs to be told to stop driving the address lines, sometimes.

(You don't need to understand multiplexing, to do this exercise; we're just talking to anyone who can't stand to do an exercise with no idea *why*!).

The rule for the 3-state control on the '573, a variable named "DEMUXOE" ("...output enable") is the following:

assert DEMUXOE (enabling the 3-state) under the following conditions:

- if we are using the loader (LOADER is asserted); or
- if *neither* BUSRQST nor KRESET is asserted.

DEMUXOE =

4.10 PSEN driving, during Loading (1 point)

Here's an odd one: during LOADING, we need to indicate to the 8051 that we're loading by driving PSEN* low. The rest of the time, we must let PSEN* do what it likes (it is a *bidirectional* pin: usually it is an output—but in this exceptional case it can be driven). Therefore, we need to drive PSEN*—a variable named PSENDRV, in this file—in a way we have not tried before:

- drive the line LOW when LOADER is asserted;
- at all other times, *turn off PSENDRV's 3-state*.

To do this,

- we first say that whenever PSENDRV is active, we want the signal to be *TRUE*:

  `"PSENDRV = ^b 1;"`

- we use the following form to turn on the 3-state: "PSENDRV.oe = ...;".

PSENDRV.oe =

4.11 Put it all into one file and compile it (4 points)

Use the template file, glu51_mt.abl, inserting your equations. All you need do is transfer the several handwritten equations youve already done—then compile it with ABEL. The simulation vectors will tell you whether youve done it right. The template file is shown below, in case you have trouble getting it from the website `http://www.people.fas.harvard.edu/~thayes/phys123/`.

Please hand in your source file (.ABL) and your simulation results (.SM1). The **template file** appears at the end of this homework—but also is *posted on the class website*, so ***YOU NEED NOT TYPE IN THIS TEMPLATE FILE!***

Template File for GLUE PAL

```
module glu51_mt
TITLE   'Glue for 8051 lab computer, loader added 7/14/01'

//       glu51_mt device 'P22V10';  // This line commented out for students, whose computers
     //  lack the 'device library' this device specification assumes
"inputs

      !BUSRQST  pin 1;
      !KRESET   pin 2; // This is inverted signal from keypad (squared up & debounced by HC14)
      !KWR      pin 3;
      !RD       pin 4;
      !WR       pin 5;
      !PSEN     pin 6;
      !INO      pin 7;
      !LOADER   pin 8;
      A15       pin 9;
      A14       pin 10;
      A13       pin 11;

      A12       pin 13;
      A11       pin 14;

"outputs

      RESET51   pin 23 ISTYPE 'com';  // Here is the ONLY active-high output!
      !KBUFEN   pin 22 ISTYPE 'com';
      !IOR      pin 21 ISTYPE 'com';  // these enable the two halves...
      !IOW      pin 20 ISTYPE 'com';  // ...of the I/O decoder
      !RAMWE    pin 19 ISTYPE 'com';
      !RAMOE    pin 18 ISTYPE 'com';
      !RAMCE    pin 17 ISTYPE 'com';
      !PSENDRV  pin 16 ISTYPE 'com';  // During boot load, this forces PSEN* low
      !DEMUXOE  pin 15 ISTYPE 'com';  // enbls demux during 1) run, 2) boot load

// Constant assignments & intermed. vars.

      Z, X, C, H, L  =  .Z., .X. , .C., 1, 0 ;
      WRITE_ADDRESS = [ A14..A11 ];  // This 'field' is used in simulation vectors, to allow shorthand address specif'ctn
// next two lines are optional: may help to tidy equations, but you may prefer to ignore them.
// They are commented out.  If you choose to use and define these, you'll remove the comment delimiters
//      WRITE_OK =    // Blocks CPU write in low 2K
//      RUN =          // either of these signals can prevent
                          //  RUN; used to control demux oe equations

      RAMCE =            // Bottom 32K for RAM (top = I/O)
      RAMOE =    //3 conditions for reading
                                // mem: pgm rd, data read
                                // manual control
      RAMWE =    // Bottom 2K write protctd--but
                                           // no wrt-protct during download

      KBUFEN =

      IOR =          //Used to enable I/O decoder, hence
                                  // the name--but it really means
                                  // 'transaction in I/O space'
      IOW =

      RESET51 =   // 8051 RESET does double duty: pseudo-3stts buses
                                                  //  also required during download

      DEMUXOE =    // 573 demux is 3-stated on BR or Kreset
```

```
      PSENDRV = ^b 1;         //  Assert this (active low) during download
      PSENDRV.oe =     //  ...by turning on the 3-state

test_vectors 'test RAMCE logic'

      ([ A15, BUSRQST ] -> [ RAMCE])

        [1,0] -> [0];
        [0,0] -> [1];
        [1,0] -> [0];
        [1,1] -> [1];
        [1,0] -> [0];
        [0,1] -> [1];
        [1,0] -> [0];

test_vectors 'test RAM OE logic'
        ([PSEN,RD,BUSRQST] -> [RAMOE])
        [0,0,0] -> [0];
        [1,0,0] -> [1];
        [0,0,0] -> [0];
        [0,1,0] -> [1];
        [0,0,0] -> [0];
        [0,0,1] -> [1];
        [0,0,0] -> [0];

test_vectors 'test RAM write logic: first, normal run, write-protected...'

      ([ WRITE_ADDRESS, WR, KWR, BUSRQST,LOADER ] -> [ RAMWE ])

        [0,0,0,0,0] -> [0];
        [0,1,0,0,0] -> [0];
        [1,1,0,0,0] -> [1];
        [1,0,1,0,0] -> [0];
        [0,0,1,1,0] -> [1];
        [0,0,0,1,0] -> [0];
        [0,0,1,0,0] -> [0];

test_vectors 'test RAM write logic: ...second, loader allows write anywhere (defeats wrt protect)'

      ([ WRITE_ADDRESS, WR, KWR, BUSRQST,LOADER ] -> [ RAMWE ])

        [0,0,0,0,0] -> [0];
        [0,1,0,0,0] -> [0];
        [0,1,0,0,1] -> [1];     // write to low address, but effective with Loader asserted
        [0,0,0,0,1] -> [0];

test_vectors 'test micro reset logic'
        ([KRESET,BUSRQST,LOADER] -> [RESET51])

        [0,0,0] -> [0];
        [1,0,0] -> [1];
        [0,0,0] -> [0];
        [0,1,0] -> [1];
        [0,0,0] -> [0];
        [0,0,1] -> [1];
        [0,0,0] -> [0];



test_vectors 'test KBUFEN logic'
        ([BUSRQST,KWR,IN0] -> [KBUFEN])
        [0,0,0] -> [0];
        [1,0,0] -> [0];
        [1,1,0] -> [1];
        [0,1,0] -> [0];
        [0,0,0] -> [0];
        [0,0,1] -> [1];
        [0,0,0] -> [0];

test_vectors 'test I/O logic: first, use RD* or WR*'
        ([PSEN,RD, WR, A15] -> [IOR, IOW])
        [0,0,0,0] -> [0,0];
        [0,1,0,0] -> [0,0];         // RD, but not in I/O space
        [0,0,0,0] -> [0,0];
        [0,1,0,1] -> [1,0];         // First effective io read
        [0,0,0,1] -> [0,0];
        [0,0,1,1] -> [0,1];            //...effective io write
        [0,0,1,0] -> [0,0];         // attempted write, but not in I/O space
        [0,0,0,1] -> [0,0];
```

```
test_vectors 'test I/O logic: second, rely on PSEN*'
        ([PSEN,RD, WR, A15] -> [IOR, IOW])

        [0,0,0,0] -> [0,0];
        [1,0,0,0] -> [0,0];         // PSEN*, but not in I/O space
        [1,0,0,1] -> [1,0];         // PSEN*, in I/O space
        [0,0,0,1] -> [0,0];         // I/O space, but no read or write or PSEN*

test_vectors 'test LOADER drive of PSEN* as input'

      ([ LOADER ] -> [PSENDRV])

        [0] -> [Z];
        [1] -> [1];
        [0] -> [Z];

test_vectors 'try demux control'

        ([KRESET, BUSRQST, LOADER] -> [DEMUXOE])

        [0,0,0] -> [1];
        [1,0,0] -> [0];
        [0,0,0] -> [1];
        [0,1,0] -> [0];
        [0,0,0] -> [1];
        [1,0,0] -> [0];
        [1,0,1] -> [1];   // Amended per Felix's observation that "001" did not test Loader
END glu51_mt
```