ECE331 Microcomputers (KEH) November 18, 2008
Test #2 – 100 Points, 4 Hours
Open Textbook and Microcontroller Interfacing Topics Notes
Dept. of Electrical and Computer Engineering
Rose-Hulman Institute of Technology

Name: Solution **CM Box:**________

1) **(14 Points, 1 point per blank)**
Assembly Language Program: Interrupt-driven White Noise Generator
The hardwired circuit shown below in Figure P1 is a 31-bit right-shift register consisting of D flip-flops FF0 – FF30, whose input is formed by EXCLUSIVE OR-ing the outputs of FF2 and FF30. This sequence generator produces a "maximum length" pseudorandom binary sequence (PRBS) that will not repeat until 2^{31}-1 = 2,147,483,647 (that is over 2 Billion!) clock pulses have elapsed. The system output may be taken from the output of any flip-flop in the shift register. The (normally closed) PRESET pushbutton is used to start the shift register in the state of all 1's, since the state of all 0's is the one state that is *not* allowed in a maximal length pseudorandom binary sequence generator (since it locks the generator into a sequence that is all 0's), and so we must not let this circuit start in the all 0's state. When clocked at 20 kHz, the sequence will take (2^{31}-1)/20000/60/60 = 29.8 hours to repeat itself! Thus the binary output is a rather random sequence of 0's and 1's! If this circuit drives a loudspeaker, it will produce white noise that might be used as a sleep aid.

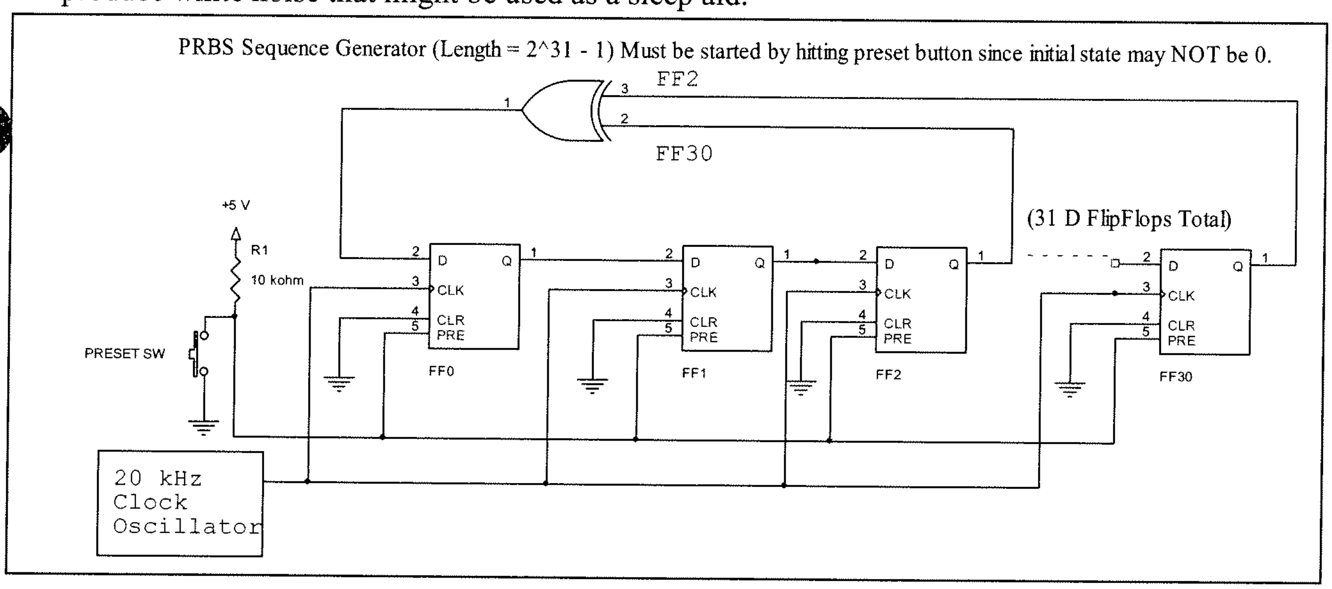


Figure P1. Pseudorandom Binary Sequence Generator

Below is an assembly-language program written for a Freescale 9S12C32 (specifically for our CSM12C32 module) that emulates this hardwired white noise generator in software as an interrupt routine. The calling program sets up the **RTI interrupt** to interrupt at a rate of 15.625 kHz. (Recall that we used the RTI interrupt in our DVM Lab (Lab 4b). Each RTI interrupt corresponds to a clock pulse in the hardwired system above. The main program also enables interrupts before it falls into an idle loop. The interrupt routine implements the rest of the system shown in Fig. P1. <u>*Note that this software emulation should behave exactly like the hardware system in Fig. P1, except it is clocked at a 15.625 kHz instead of 20 kHz.*</u>

The program is intended to run on our CSM12C32 lab modules, and that these modules employ a 16 MHz ceramic resonator, which sets the OSCCLK rate to 16 MHz. Note from Fig. 6.15 in the Huang Text that OSCCLK has nothing to do with the PLL that forms the bus clock, thus the RTI interrupt rate does ***NOT*** depend upon the bus clock rate, as set by the PLL.

Four byte-sized variables, or RAM locations, (SHR3, SHR2, SHR1, and SHR0) in the program below are used to implement the 31-bit shift register in Fig. P1, where SHR3 represents FF0 – FF7; SHR2 represents FF8 – FF15; etc. Note that with this assignment, the most significant bit of SHR3 (Bit #7) is FF0, the least significant bit of SHR3 (Bit #0) is FF7, and Bit #1 of SHR0 is FF30. Further note that Bit #0 of SHR0 is *not used*, since only 31 flip-flops (not 32) are needed in this design.

The system output is taken from FF30, and the state of FF0 driven out on I/O pin PM0. If a piezoelectric loudspeaker is connected to PM0, we will hear the broadband white noise as a steady "hiss".

Fill in the missing blanks in this assembly-language program.

```
; ECE331 White Noise
; PRBS.ASM - Generates 2^31-1 bit long PRBS (pseudorandom binary
;             sequence with a 15.625 kHz clock rate). Uses RTI interrupt.
;
            XDEF WHITENOISE
            ABSENTRY WHITENOISE
            INCLUDE 'mc9s12c32.inc'
            ORG $800
SHR3:       ds.b 1
SHR2:       ds.b 1
SHR1:       ds.b 1
SHR0:       ds.b 1
TEMP:       ds.b 1
            ORG $4000
WHITENOISE: lds #$1000
            bset DDRM,1
            bclr PTM,1

            ;Next two lines simulate depression of PRESET SW in Fig. P1
            movw #$FFFF,SHR3 ;***BLANK #1 ****
            movw #$FFFF,SHR1 ;***BLANK #2 ****

            ;Divide 16 MHZ OSCCLK to get
            ;RTI interrupts at 15.625 kHz rate (See Huang text, Table 6.4
            ;and Huang Text Fig. 6.11 and Fig. 6.12)
            movb #$10,RTICTL ;***BLANK #3 ****

            bset CRGINT,#$80    ;***BLANK #4 ****
            movb #$80,CRGFLG    ;Clear RTI interrupt flag
            CLI                 ;***BLANK #5 ****
loop_here_forever:
                          bra   loop_here_forever
;*********Here ends the main program "WHITENOISE"
WHITENOISEISR:
            CLR TEMP
            BRCLR SHR3,%00100000,FF2NOTSET
            MOVB #1,TEMP
FF2NOTSET:  CLRA
            BRCLR SHR0,2,FF30NOTSET ;***BLANK #6 ****
```

Rotate output of XOR gate (in Fig. P1) into "C" Flag.

```
              LDAA #1
FF30NOTSET:   EORA TEMP                        ;***BLANK #7 ****
              RORA (or LSRA, ASRA)              ;***BLANK #8 ****
              ROR SHR3
              ROR SHR2
              ROR SHR1
              ROR SHR0
```

Rotate Bit #30 into position

```
              ldaa  SHRØ                        ;***BLANK #9 ****
              RORA (LSRA, ASRA)                 ;***BLANK #10 ****
              STAA PTM ;Send Bit #30 out to PTM0

              ;Relax the RTI interrupt flag
              movb  ##80, CRGFLG                ;***BLANK #11 ****
              RTI                               ;***BLANK #12 ****
;************************************************************
;*    Initialize Reset Vector and RTI Interrupt Vector     *
;************************************************************
  ORG $FFFE
  dc.w     WHITENOISE       ;Make reset vector point to
                           ;entry point of WHITENOISE program
  ORG  # FFFØ                                   ;***BLANK #13 ****
  dc.w  Whitenoise ISR                          ;***BLANK #14 ****
```

2) ***(14 points, 1 point per blank) C Language Program: Interrupt-Driven Music Player***
Fill in the 14 blanks in the C program below that plays music. The TONE array is loaded with values N = 0 – 24, which represent two octaves of the musical scale: A1, Bb1, B1, C1, Db1, D1, Eb1, E1, F1, F#1, G1, Ab1, A2, Bb2 B2, C2, Db2, D2, Eb2, E2, F2, F#2, G2, Ab2, A3. Furthermore, let the value N = 25 correspond to the special case of silence (a musical "rest"). Let A1 correspond to 220 Hz, then A2 must correspond to 440 Hz, one octave above A1, and A3 corresponds to 880 Hz.

Since the Western musical scale varies in 12 logarithmically-spaced steps between octaves, the frequency of each note in this scale is given by

$$f = 220 \bullet 2^{N/12} \text{ Hertz, where N = the note number (0 – 25)}$$

The DURATION array is loaded with tone duration values that range from 1 up to 16. Let "1" represent the shortest possible note duration, let's call it a 16th note, then "2" represents a note that is twice as long (an 8th note), "4" represents a note that is 4 times as long (a quarter note), "8" represents a half note, and "16" represents a whole note. Note that with this scheme, a dotted 8th note, which is 1.5 times the length of a regular 8th note, would be represented by "3", and a dotted quarter note would be represented by "6", etc.

Note that the main program is quite short. It calls function ***music_init()*** that initializes the timer tick rate, TC0 "output compare" interrupt mechanism, and other important variables and registers, it ends by globally enabling interrupts. Then the main program enters an infinite "idle" (do nothing) loop.

The TC0 interrupt routine ***music_isr()*** performs the tasks of fetching the next tone and duration values from the TONE and DURATION arrays, generating the musical tones (as square waves) based upon the information fetched from the TONE array, and deciding how long to generate each tone, based upon the information fetched from the DURATION array.

Fill in ***all 14*** of the blanks in the music program below, so that it repeatedly plays “Dear Old Rose”.

```
#include <hidef.h>              /* common defines and macros */
#include <mc9s12c32.h>          /* derivative information */
#pragma LINK_INFO DERIVATIVE "mc9s12c32"
#define SONGSIZE 10             //There are 10 notes in this song
#define SIXTEENTHNOTE 300000    //SIXTEENTHNOTE = number of timer ticks in a 0.2
                                //                second sixteenth note
void INIT_PLL(void);
void music_isr(void);
void music_init(void);
char getnoteflag, noteptr, note_number;
long int dur_nr_half_cycles;
int tone_val, dur_counter;
//The "duration" and "tone" arrays below play the "Dear Old Rose" RHIT fight song
const char duration[SONGSIZE]={2, 2, 3, 1, 1, 1, 2, 2, 3, 4};
const char     tone[SONGSIZE]={6, 7, 8,10,11,13,15,13,11,25};//End with a rest
const int tone_table[25]=     {6818, 6435, 6074, 5733, 5412, 5108, 4821,
                                       //(***Blank 1 ***)
                               4550, 4295, 4054 ,3827, 3612, 3409, 3218,
                               3037, 2867, 2706 , 2554, 2411, 2148, 2027,
                                    //(***Blank 2 ***)
                               2027, 1806, 1705, 1609};

void main(void)
{     INIT_PLL();          // Set bus clock frequency to 24 MHz
      music_init();
      for(;;);
}

void music_init()
{ getnoteflag = 1;        // Set getnoteflag = 1, so first interrupt will fetch a
                          // new note from tone[] and duration[] arrays.
  noteptr = 0;            // Make noteptr point to first note (first element)
                          // in the tone[ ] and duration[ ] arrays.
  dur_counter = 0;        // Clear Duration Counter, which counts the number
                          // of half cycles that a note is played.
  DDRM_DDRM0 = 1;         // Make PTM0 (Bit #0 of Port M) an output.
                          // (PTM0 is connected to an amplified loudspeaker.)
  PTM_PTM0 = 0;           // Set PTM0 low.
  TSCR2 = 3               // Set Prescaler to divide bus clock by 8. (***Blank 3***)
                          // Assume 24 MHz bus clock,
                          // Thus the Timer Tick time = 8/24E6 = 333.3333 ns
  TSCR1 = 0x80            // Turn on timer                    (***Blank 4 ***)
  TIOS = 1                // Make TC0 an Output Compare       (***Blank 5 ***)
  TC0 = TCNT + 25;        // Schedule first TC0 interrupt in 25 timer ticks
  TFLG1 = 1               // Clear TC0 interrupt flag         (***Blank 6 ***)
  TIE = 1                 // Locally Enable TC0 interrupts    (***Blank 7 ***)
  EnableInterrupts;       // Globally Enable TC0 interrupts
}
```

N=0

N=3

N=16

$F = 220 \cdot 2^{N/12}$

Ticks in ½ period $= \frac{(1/F)/2}{333.333\text{ns}}$

```
// The following interrupt routine is entered when an output compare on TC0
// This TC0 interrupt should occur every half of a note cycle.

void interrupt music_isr( )
{ TFLG1 = 1;              //Relax TC0 interrupt
  if(getnoteflag == 1)
      {
        getnoteflag = 0;
        note_number = tone[noteptr] //Look up the number of the next note
                      //(***Blank 8 ***)

        if(note_number > 24) note_number = 25;  // If an invalid note number
                                                //(> 24) is entered,
                                                //make it a rest = 25.
        // tone_val = nr of ticks in half cycle of note
        tone_val = Tone_Table[note_number] //(***Blank 9 ***)
        dur_nr_half_cycles = duration[noteptr]*SIXTEENTHNOTE / tone-val );
                          //(***Blank 10 ***)          (***Blank 11 ***)
        // Hint: "SIXTEENTHNOTE" sets the speed at which
        // the musical composition is played. Note 'dur_nr_cycles"
        // is the total number of half cycles in the note that cause that
        // note to be played for the specified note duration.
        dur_counter = 0;                // Reset duration counter
        noteptr++;                      // Increment noteptr.
        if(noteptr > SONGSIZE) {noteptr = 0 ;  //(***Blank 12 ***)
          // If song is completed, wrap back to beginning.
      }
      else
      {
        if (note_number < 25) PTM_PTM0 = ~PTM_PTM0; // Toggle PTM0
        dur_counter++;              // increment duration counter
        if(dur_counter > dur-nr-half-cycles) getnoteflag = 1;
                         //(***Blank 13 ***)
        // Set getnoteflag = 1 if at the end of the note
      }
        TCØ = TCØ (or TCNT) + tone-val ;   // Schedule next TC0 interrupt
          //(***Blank 14 ***)
}
```

3. (16 points) LCD Display Multiplexing

a. (1 pts) A custom LCD display for a new product has 300 segments that must be individually controlled (turned on or off). If we choose to use 1:4 multiplexing on this display, implying 4 back plane signals are needed, what is the total number of wires (back plane wires *plus* front plane wires) that must be connected to this display?

$\frac{300}{4} + 4 = 75 + 4 = 79$

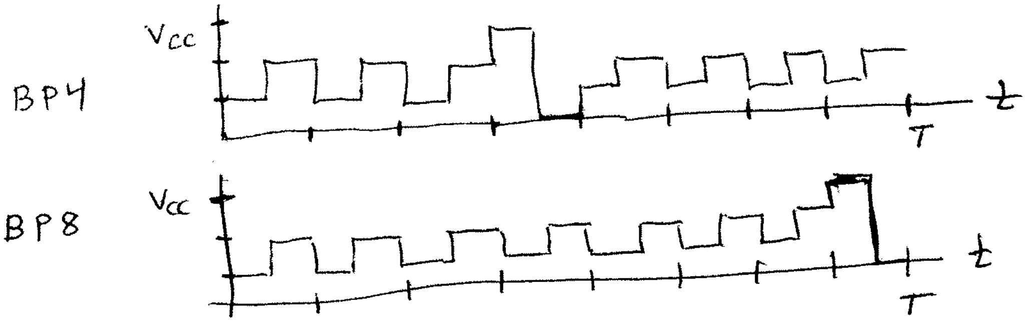
Total # Wires = 79

b. (1 pts) Repeat Part A for 1:8 multiplexing.

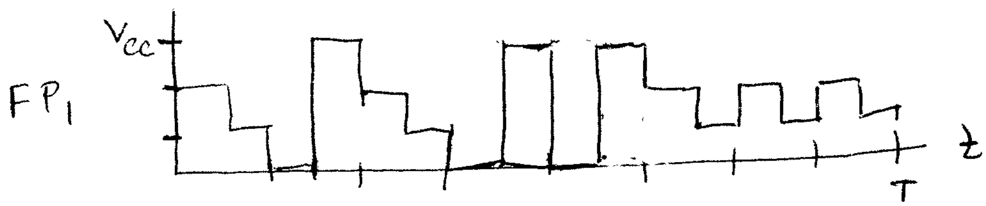
$\frac{300}{8} + 8 = 45.5 \Rightarrow$ 46 wires

Total # Wires = 46

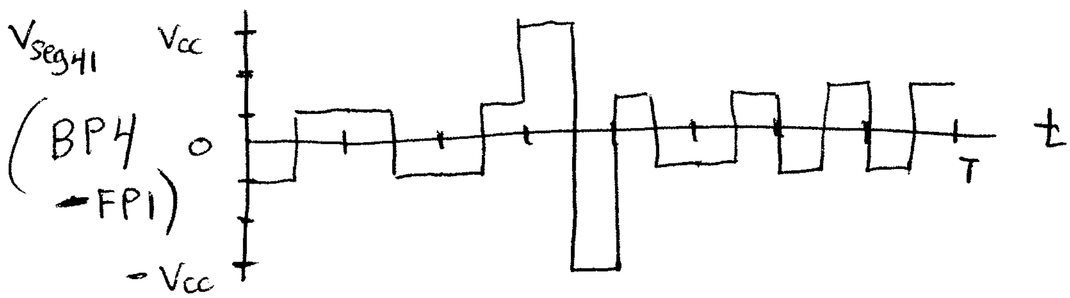
c. (2 pts) For the case of 1:8 LCD multiplexing, there are 8 backplane signals, BP1, BP2, BP3, BP4, BP5, BP6, BP7, and BP8. Assume that Vcc = 5 V, so the waveform voltage levels are 5 V, 3.333 V, 1.666 V, and 0 V. Sketch one frame of the **BP4** backplane signal and also one frame of the **BP8** backplane signal.



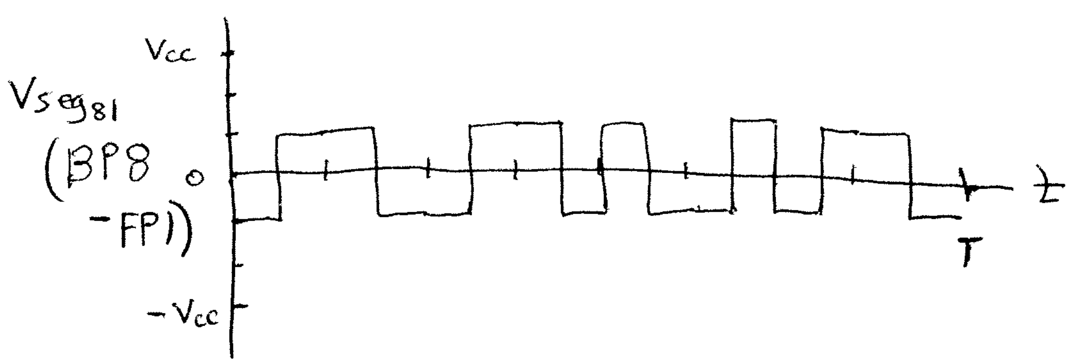
d. (3 pts) Sketch one frame of a single front plane signal, FP1, where the segments that pass over BP2, BP4, and BP5 are to be ON, and the remaining five segments are to be OFF.



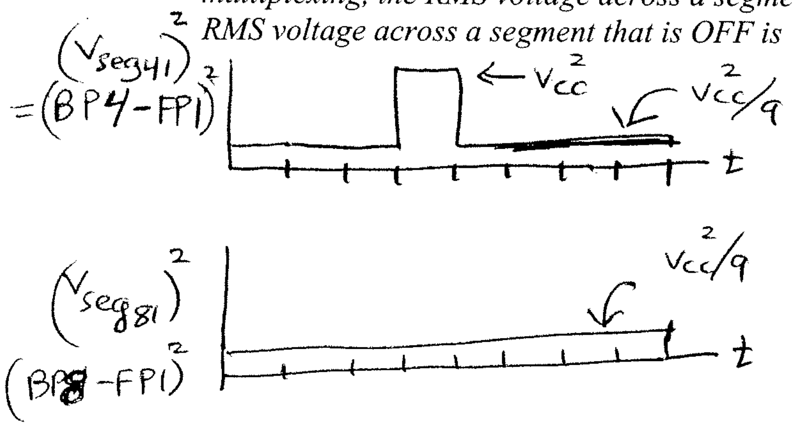
e. (3 pts) Sketch one frame of the voltage waveform $Vseg_{41}$, which represents the voltage across the "turned ON" segment that lies between FP1 and BP4. ($Vseg_{41}$ = BP4 voltage – FP1 voltage). Use the FP1 voltage waveform from Part d above.



f. (3 pts) Sketch one frame of the voltage across the "turned OFF" segment that lies between FP1 and BP8, $Vseg_{81}$. ($Vseg_{81}$ = BP8 voltage – FP1 voltage) . Use the FP1 voltage waveform from Part d above



g. (2 pts) Find the RMS value of the $Vseg_{41}$ waveform of Part e, which corresponds to the waveform of a turned **ON** segment, and also the RMS value of the $Vseg_{81}$ voltage waveform of Part f, which corresponds to a turned **OFF** segment. For credit on this problem, you ***must*** show the steps in your calculation (not just write down numbers) in the space below. *Recall that in the class notes, it was shown (in Figure 7.21) that for the case of 1:4 multiplexing, the RMS voltage across a segment that is ON is Vrmson = 2.899 V,rms; and the RMS voltage across a segment that is OFF is Vrmsoff = 1.67 V, rms.*



$$(V_{seg41})_{RMS} = \sqrt{\frac{1}{T}\left[\frac{7}{8}T\frac{V_{cc}^2}{9} + \frac{1}{8}T V_{cc}^2\right]} = 2.36V$$

$$(V_{seg81})_{RMS} = \sqrt{\frac{1}{T}\left[T\frac{V_{cc}^2}{9}\right]} = \frac{V_{cc}}{3} = 1.67V$$

RMS value of $Vseg_{41}$ = 2.36 V,rms   RMS value of $Vseg_{81}$ = 1.67 V,rms

F. (1 pt) Based upon comparing the results for 1:4 and 1:8 multiplexing,

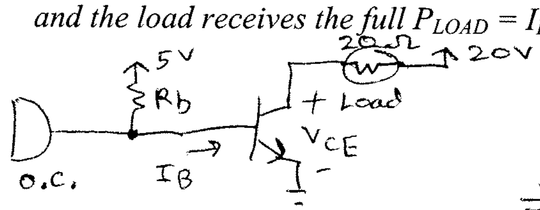
(a) which multiplexing method requires fewer connections? 1 : 8

(b) which multiplexing method yields higher contrast? 1 : 4

Since 2.36V < 2.899V
⇒ dark level is "darker" for 1:4 MUX

4. (7 pts) An NPN power BJT with a β = 150 is used to switch ON and OFF a 10 Ω, 20 V (40 Watt) resistive load using the ***upper-left circuit of Slide #57***. (Assume Vbe(on) = 0.7V and Vce(sat) = 0V.)

a. (3 pts) Draw this circuit in the space below, and determine the maximum permissible value of Rb that is necessary to keep the BJT saturated while the load is ON. *Note that with the BJT saturated, the switching BJT consumes essentially NO power ($P_{BJT} = Ic*Vce = 2*0 = 0W$), and the load receives the full $P_{LOAD} = I_L*V_L = 2*20 = 40$ W from the dc power supply.*



$$V_{CE} = 0 = 20 - \left(\frac{5-0.7}{R_b}\right)\beta \cdot 20\Omega \quad (\beta = 150)$$

$$\Rightarrow R_b = 322.5\Omega$$

Rb(MAX) = 322.5 Ω

b. (1 pt) How much current must the open-collector driving gate be able to sink while the load is turned off? (Assume that the value of Rb is the value calculated above in Part a, and that the output voltage of the driving gate is at a voltage of 0.5 V when sinking this current.)

$$\frac{5-0.5}{Rb_{max}} = 13.95mA$$

$Iout_{SINK}$ = 14.0 mA

c. (3 pts) If Rb = 500 Ω, and the open-collector driving gate is switched to its HIGH (floating) state, find the power that is delivered to the (10 Ω, "40 Watt") load
(*Hint: Because Rb = 500 Ω violates the calculation in Problem 4(a), you will find that the power delivered to the load will far less than the desired 40 Watts!*
Also find the power that is dissipated (as heat) in the BJT switching transistor. *(Hint: you may ignore the small amount of power consumed in the base-emitter junction of the BJT, and so $P_{BJT} = Vce * Ic$. Because Rb = 500 Ω violates the calculation in Problem 4(a), the power consumed (as heat) in the switching transistor will be unacceptably large, and it may even burn out the switching transistor!)*

$$V_{Load} = \left(\frac{5-0.7}{500}\right)(150)(10\Omega) = 12.9\text{V} \quad (150 \leftarrow \beta,\ 10\Omega \leftarrow R_L)$$

$$P_{Load} = \frac{V_{Load}^2}{10\Omega} = \frac{(12.9)^2}{10\Omega} = 16.64\text{W}$$

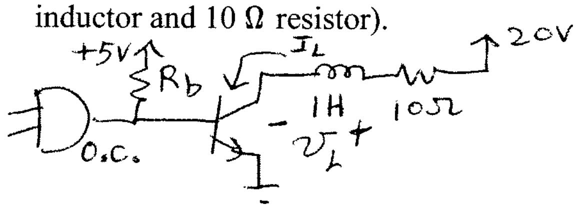
$$P_{BJT} = I_C \cdot V_{CE} = \underbrace{\left(\frac{12.9}{10}\right)}_{I_C}\underbrace{(20\text{V}-12.9)}_{V_{CE}} = 9.159\text{W}$$

$P_{10\Omega\ LOAD}$ = 16.64 W

P_{BJT} = 9.159 W

5. (4 points) Imagine that the 10 Ω resistive load of Problem 4 is replaced by an inductive load that may be modeled as a ***1.0 H inductance in series with a 10 Ω resistance***.

a. (1 pt) Sketch the modified switching circuit in the space below. This circuit consists of the open-collector driving gate, the 20 V dc power supply, Rb (assume Rb has a value that is less than the maximum value calculated in Problem 4(a)), the switching BJT, and the load (1 H inductor and 10 Ω resistor).



b. (3 points) Assume that the open-collector driving gate output voltage has been LOW for a long time, and then it suddenly is raised to its HIGH (floating) state. How long after that will it take for the load current to reach 90% of its final value (1.8 Amperes)? Let us regard this as the load "turn-on" time. (***Hint: Study Lecture 11, Slides 71-78***)

$I_{L_F} = \frac{20\text{V}}{10\Omega} = 2\text{A}$ $\quad I_{L_i} = 0\text{A}$ (BJT cut off if o.c. gate output Low)

$\tau = L/R = 1\text{H}/10\Omega = 0.1\text{s}$

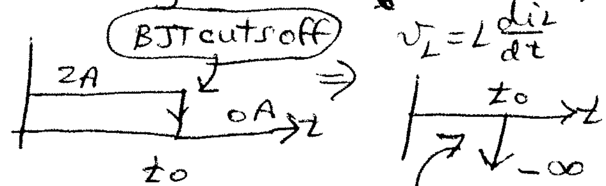
$1.8 = 2 - 2e^{-10t_{ON}} \Rightarrow t_{ON} = 0.23\text{s}$

Load Turn-On Time = 0.23s

6. (9 pts) Now imagine that the driving gate output voltage of the circuit in Problem 5 is suddenly changed from HIGH to LOW. ***Hint: See Lecture 11, Slides 71-78***

a. (1 pts) Using $v_L = L\,di_L/dt$ to explain why the switching BJT could burn out.

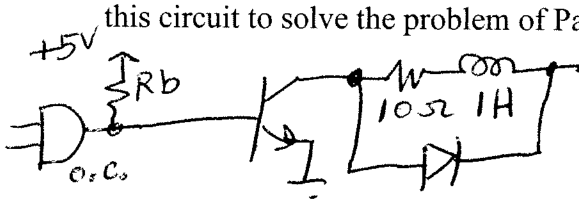
As driving gate output switches High → Low (at $t = t_0$), BJT suddenly cuts off.



(Negative $v_L(t)$ spike could burn out BJT!)

"Inductive Kick"

b. (1 pts) Redraw the switching circuit showing how a ***single*** fast-acting diode may be added to this circuit to solve the problem of Part (a).



↑ Inductive "Kick" suppression diode

c. (4 pts) For the circuit of Part (b), determine how long it will take for the load current to decay from its full value down to 10% of this value (0.2 A) when the driving gate output voltage is suddenly changed from HIGH to LOW . You might regard this as the "load turn-off time". ***Hint: See Lecture 11, Slides 71-78***

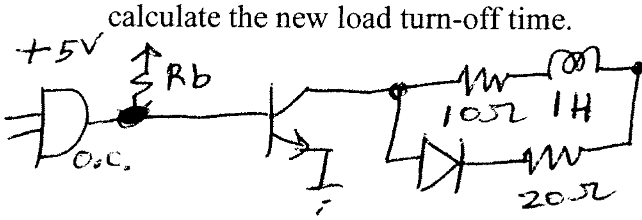
$(I_L)_0 = 2A \quad (I_L)_F = 0$

$0.2 = 0 - (0-2)e^{-\left(\frac{t_{off}}{(1H/10\Omega)}\right)}$

$0.2 = 2e^{-10t_{off}} \Rightarrow t_{off} = 0.23\,s$

Load Turn-Off Time = 0.23 s

d. (3 pts) How could you make this load turn off time shorter? Redraw the circuit showing how one additional resistor "Rspeedup" might be added, so the load will turn off faster, without affecting the load current of the load turn-on time. If Rspeedup = 20 ohms in this example, calculate the new load turn-off time.



$0.2 = 2e^{-t_{off}/\left[\frac{1H}{(10+20)}\right]}$

$\Rightarrow t_{off} = 0.0768\,s$

New Load Turn-Off Time = 0.0768 s

(Load Turns off quite a bit faster!!)

7. (8 pts) Using only ***TWO*** rising-edge sensitive D flip-flops (with D, CLK, CLR, Q and Q\ pins) and assorted inverters and logic gates, design a circuit that will derive the ***2x resolution*** ***CW output waveform*** shown in Fig. 6 from the A and B input waveforms. (See arrow below) YOU NEED NOT DERIVE THE CCW output waveform. ***Be sure to label your circuit's A and B inputs as well as your circuit's CW output.***

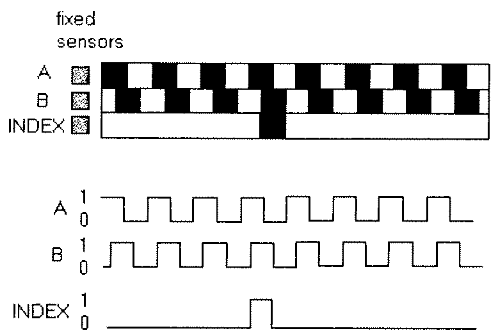


Fig 5. Incremental encoder disk track patterns

Desired output waveform→

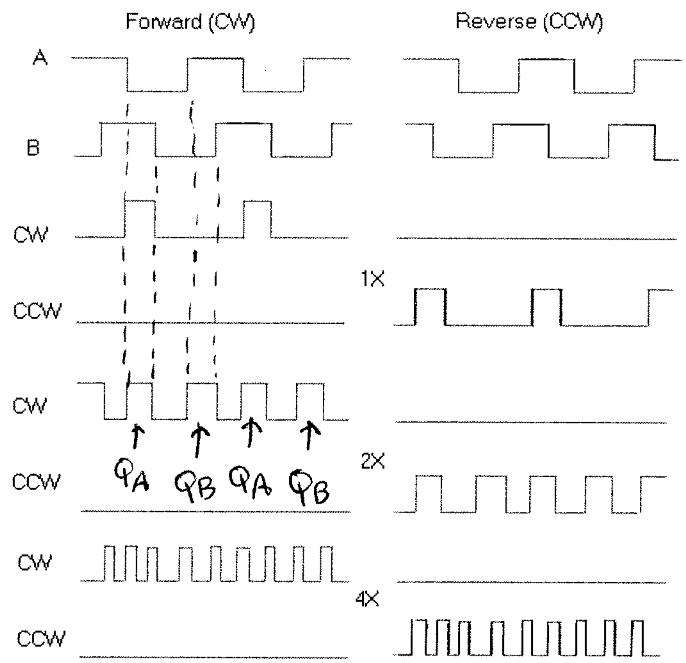
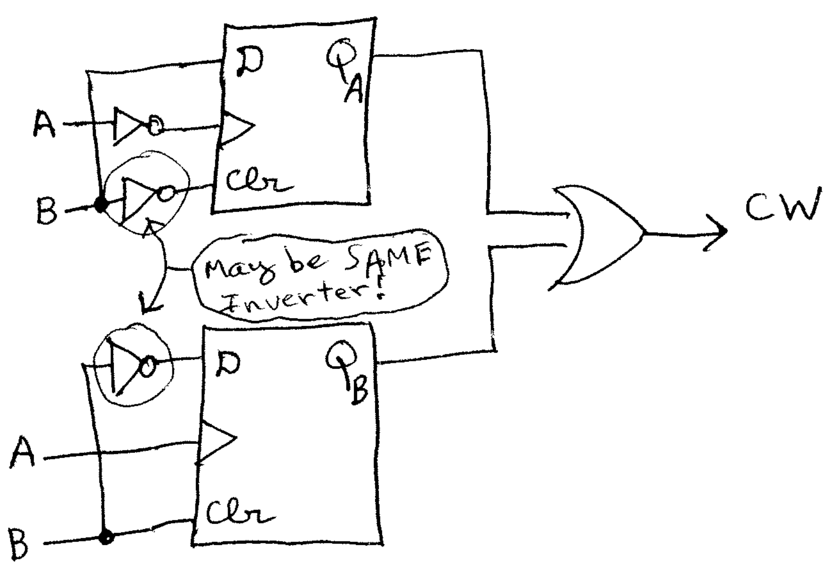
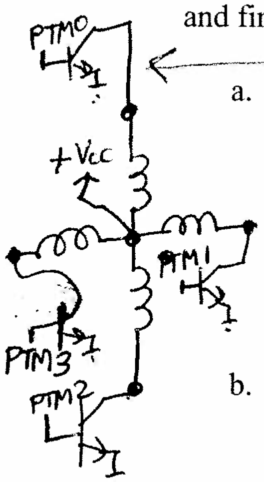


Fig 6. Quadrature direction sensing and resolution enhancement. (CW = clockwise, CCW= counter-clockwise)



8. ***Stepping Motor (8 points)***
Referring to the stepping motor circuit diagram shown in the course notes (Slide #68), imagine that the two bottom rows of 7407/7406 inverters are removed, leaving us with just one row of 2N6427 power Darlington BJT transistors. Then imagine that a microcontroller has PTM3 (Port M, Pin 3) connected to the base of the left-most power Darlington, PTM2 to the next one, PTM1 to the next, and finally PTM0 to the right-most power Darlington.



a. (2 pts) List the sequence of eight 4-bit numbers that would have to be output on the low 4 bits of PORT M (in the order PTM3:PTM2:PTM1:PTM0) in order to make the magnetic field vector developed by the stepping motor stator coils step in the clockwise (CW) direction, with 8 steps per revolution (45 degrees per step). Let your first number correspond to the magnetic field pointing directly up. (Hint: you may turn on either 1 or 2 coils at a time.) Note "0" cuts off BJT, "1" Saturate BJT

0001, 0011, 0010, 0110, 0100, 1100, 1000, 1001

↑ $\vec{B}_{0001}$ ↗ $\vec{B}_{0011}$ → $\vec{B}_{0010}$ ↘ $\vec{B}_{0110}$ ↓ $\vec{B}_{0100}$ ↙ $\vec{B}_{1100}$ ← $\vec{B}_{1000}$ ↖ $\vec{B}_{1001}$

b. (4 pts) Assuming a permanent magnet rotor with 7 permanent magnet poles (instead of the rotor with 3 permanent magnet poles considered on Slide #69 in the lecture notes), determine the number of steps per revolution of the shaft using the 8-value sequence of Part A. Do this by drawing, in the space provided below, the 7-pole rotor (showing only the 7 equal-angularly spaced south poles) with one of the 7 poles aligned with the initial **B** field. Then, when the **B** field steps 45 degrees to its next position, determine which south pole is closest to the new position of the **B** field, and hence is pulled into alignment. Determine the angle through which the shaft rotates, and determine its direction of rotation (CW or CCW). Also determine the total number of steps per one 360 degree revolution of the shaft.

Degrees of Shaft Angle Rotation Per Step = 6.429° Step Direction = CCW
Steps per one 360 degree revolution of the shaft = 56

$\frac{360°}{7} = 51.429°$ = Angular spacing between PM Rotor poles (Permanent Magnet)

Shaft Step Angle = 51.429° − 45° = 6.429°

shaft turns CCW

$\frac{360°}{6.429°}$ = 56 steps/revolution

c. (1 pt) What is the best name for the four 1N4001 power diodes in this stepping motor circuit? (circle one)
(1.) transient voltage suppression diodes 2. turn-on speedup diodes 3. turn-off speedup diodes
4. load current limiter diodes

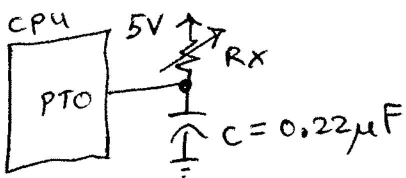
d. (1 pt) What is the best name for the 22-ohm resistor in this stepping motor circuit? (circle one)
1. turn-on speedup resistor (2.) turn-off speedup resistor 3. load current limiter
4. voltage transient suppression resistor

9. (2 pt) A magnetic reed switch will be most sensitive to an applied magnetic field (**B**) that is oriented in a direction that is
   1. perpendicular to the reeds (2.) parallel to the reeds 3. at a 45 degree angle to the reeds

10. (2 pt) What is the purpose of the diodes in the 8 x 8 scanned keyswitch matrix discussed in the course notes?
   (1.) short-circuit protection 2. over-voltage protection 3. speed up key scanning process

11. (9 pts) Imagine that a "poor man's A/D" circuit implemented in the C language is used to sense the value of a variable resistor Rx by connecting Rx between PT0 and Vcc = 5.0 V and a 0.22 μF capacitor between PT0 and ground. Assume that PT0 (when configured as an input) has an input logic high threshold of 3.00 V. If PT0 (when configured as an output) is driven low (to 0 V) for several seconds, and then suddenly released (allowed to float), the time elapsed before a logic 1 level is read by the microcontroller is measured.

a. (2 pts) Find the value of Rx if the time elapsed before a logic 1 is read is found to be 4 ms?



$$3 = 5 - 5e^{-4ms/(Rx \cdot 0.22\mu F)}$$

$$\Rightarrow \boxed{Rx = 19.843\ k\Omega}$$

b. (2 pts) Find the value of Rx if the time elapsed before a logic 1 is read is found to be 8 ms?

$$3 = 5 - 5e^{-8ms/(Rx \cdot 0.22\mu F)}$$

$$\Rightarrow \boxed{Rx = 39.686\ k\Omega}$$

(Note: doubled time ⇒ doubled value of Rx)

c. (2 pts) What is the lowest value of Rx that can be measured using this scheme if PT0 cannot sink more than ***25 mA*** when driving its output to a logic 0 level (which we will assume is precisely 0 V).

$$\frac{5-0}{(Rx)_{min}} = 25mA \Rightarrow (Rx)_{min} = \boxed{200\Omega}$$

d. (1 pts) How should the LSB of the PERT register be set in order to obtain the most accurate measurement of Rx? Explain your reasoning.

LSB of PERT = 0 (No internal pullup resistor should be enabled.)

e. (1 pts) How would you set the LSB's of the Port T data register and the Port T data direction register in order to drive PT0 to 0 V?

LSB PT DDR to "1"
LSB PT Data to "0"

f. (1 pts) How would you set the LSB's of the Port T data register and the PORT T data direction register in order to release (float) PT0?

LSB PT DDR to "0"
Then Read LSB Data, waiting for it to go to "1"!

12. (7 pts) **UPC-A Bar Code** (Used on groceries, pharmaceuticals, electronic items, but NOT on books!)

a. (2 pts) Using the UPC encoding table found in the notes, determine the six encoded UPC digits in the ***left half*** of the bar code. Recall that Black = 1, White = 0; there are 3 SYNC patterns: 101 at each end, and 01010 in the middle. (*Hint: first make sure you can successfully decode the six left digits in the example UPC code in the notes, or on any grocery product in your home.*)

101 SYNC (3) (7) (1) (5) (9) (8) 01010 SYNC ... 101 SYNC

b. (2 pts) Recalling that the UPC-A encoding table found in the notes must have its ***black and white regions exchanged*** for the ***right half*** of the UPC code, determine the six encoded UPC digits in the right half of the bar code. (*Hint: first make sure you can successfully decode the six right digits in the example UPC code given in the notes.*)

(6) (4) (2) (0) (5) (8)

c. (3 pts) The last (rightmost) digit you found in Part (b) is the UPC-A checksum digit. In the space below, show the step-by-step calculation of this checksum digit from the other preceding 11 digits. Your results must match the 12th digit you decoded above.

$$D_{12} = 10 - [3\cdot(D_1 + D_3 + D_5 + D_7 + D_9 + D_{11}) + (D_2 + D_4 + D_6 + D_8 + D_{10})] \% 10$$

$$D_{12} = 10 - [3\cdot(3+1+9+6+2+5) + (7+5+8+4+0)] \% 10$$

$$= 10 - [3\cdot(26) + 24] \% 10 = 10 - 102 \% 10 = 10 - 2$$

$$= \boxed{8} \checkmark$$