1 Modifying SiLabs ADC-to-DAC Code (3 points)

We’d like you all to get a sense of the what it’s like to use an integrated controller—one with peripherals included on-chip. Below we show a program listing taken from SiLabs Controller Lab 4. The program listed in § 1.2 on the following page takes a sample from ADC and sends it to display and to DAC, using only the 8 MSB’s of the 12-bit result delivered by the ADC.

This little question gets extra weight not because it is difficult but because it obliges you to read through a couple of SiLabs programs in lab C4 and probably also C2.

1.1 The modification we’d like you to make: 12-bits to DAC and display

We’d like you to pencil in the changes you would make to the listing below in order to send all 12 bits from the ADC to the DAC and to the display. To keep your code readable, please add necessary definitions or “equates” (EQU) at the head of the program.

For example, we defined the high-bytes as follows, in the listing of § 1.2 on the next page:

```
DISPLAY_HI EQU P1 ; display high 8 bits of 12
SAMPLE_HI EQU R7
```

Total points: 21. Due Monday, April 27, 2015
Please do the equivalent *equates* for DISPLAY_LO and SAMPLE_LO (put that wherever you like), and make necessary modifications to...

- GETSAMPLE
- CHECK_FOR (the main program loop)

### 1.2 8-bit program, borrowed from Lab C4 (no additional work for you in this subsection—this is just where you’ll pencil in your changes)

This program neglects the low 4 bits of the conversions. Incidentally, this listing, from C4, omits an initialization block that appears at the end of the listing in the lab version. We have also assigned DISPLAY_HI to P1 rather than P0, to make this consistent with the wiring of lab C2—which we suspect you may want to consult, as you do this little problem.

```
; adc_dac_int_jan11.a51 : splices ADC_TEST with DAC_TEST, to do in-out on interrupt
; redone using Wizard, pins reassigned Jan 11

$NOSYMBOLS ; keeps listing short, lest...
$INCLUDE (C:\MICRO\8051\RAISON\INC\c8051f410.inc) ; ...this line might produce huge list
; of symbol definitions (all ’51 registers)
$INCLUDE (C:\MICRO\8051\RAISON\INC\VECTORS320.INC) ; Tom’s vectors definition file
STACKBOT EQU 080h ; put stack at start of scratch indirectly-addressable block (80h and up)
SOFTWARE EQU 0 ; software flag at bit 0: used by ISR to say ‘time to sample’
CNVRSN_DONE EQU ADC0CN.5 ; flag bit indicates ADC sample is ready
CNVRT_START EQU ADC0CN.4 ; ADC start bit: needs low-going pulse
DISPLAY_HI EQU P0 ; display high 8 bits of 12 (this assignment consistent with Lab C2, not with the listing in C4 as

SAMPLE_HI EQU R7

; INT0 at P0.5, to signal time to take a sample
; DAC1 at P0.1
; ADC0 at P0.0

ORG 0h
LJMP STARTUP

ORG 080h

STARTUP: mov SP, #STACKBOT-1
  acall USUAL_SETUP
    acall Init_Device
    acall INT_EDGE ; make INT1 edge-sensitive
    acall PORT_SETUP ; set bits high, to free them for ADC and DAC
CHECK_FOR: jnb SOFTWARE, $ ; hang here till interrupt says ‘time to transfer’
  clr SOFTWARE
  clr EA ; disable interrupts, to make sure it get played out to DAC
  acall GET_SAMPLE

  mov DISPLAY_HI, SAMPLE_HI ; this to let us watch as it proceeds
```
mov IDA0H, SAMPLE_HI
setb EA ; re-enable interrupts
sjmp CHECK_FOR

;--- SUBROUTINES ----
; ISR0: This is response to INT ONE: set softflag
ORG INT0VECTOR ; this is defined in VECTORS1210.INC, included above.
; It is address 13h, the address to which micro hops
; in response to interrupt ONE
ISR0: setb SOFTFLAG ; pseudo-polling: a flag that MAIN will check
RETI

GET_SAMPLE: CLR CNVRT_START ; low on AD0BUSY (to permit rising edge)
SETB CNVRT_START ; rising edge on AD0BUSY starts conversion
JNB CNVRSN_DONE, $; hang here till conversion-done flag
CLR CNVRSN_DONE ; not required, but tidy to clear this flag
MOV SAMPLE_HI, ADC0H ; high byte
RET

; ------INITS ---------
USUAL_SETUP: anl PCA0MD, #NOT(040h) ; Disable the WDT.
; Clear Watchdog Enable bit
; Configure the Oscillator

    mov OSCICN, #087h ; max speed sysclk: 24.5 Mhz
ret

INT_EDGE: setb IT0 ; make INT0 Edge-sensitive
ret

PORT_SETUP: setb P0.0 ; make sure latch is high (this for DAC)
             setb P0.1 ; ...and this for ADC
ret

2 Do_A, Do_B, Using Interrupt (8 points, total)

Last time, you showed how to use polling of two pushbuttons so as to make a controller do one thing or another. This time, we’d like you to show how to do this DO_A, DO_B program using interrupts. Since you told us earlier how to debounce, you needn’t show that again. If you need a debouncer, just show it as a box so labelled.

2.1 Hardware (3 points)

Show how to use two SPST pushbutton switches to get this behavior. You may use the external bus or the 8051’s internally-defined ports. Use interrupts rather than polling. Your little computer should do each action just once in response to one press of each pushbutton.
2.2 Code: Interrupt (5 points)

Again, write assembly-language code that would check whether A or B had been pressed, and would carry out action “A” or “B” in response, once, then would revert to checking for a button-pressing. In order to “DO A,” you need only call a subroutine named “DO_A.” Same for B. If both A and B buttons have been pushed, do A, then B. Include comments, please. You need not write either “DO_A” or “DO_B.”

Use INTERRUPT 0, whose “interrupt vector” is a set of 8 bytes beginning at location 03h, or use both INTO and INTI (its vector begins at address 13h).

An ISR can set a flag, tested by the Main program Sometimes it is useful to have an ISR set a FLAG, in software, so as to let the MAIN program know what’s up. The example below shows code using a bit that happens to be unused in the PSW (a register used mostly for ‘flags,’ such as the Carry flag). To make the code more readable, we say this up at the head of the program:

SAMPLEFLAG EQU 0h ; first bit-addressable location in internal "scratchpad" RAM:
    ; (if you simulate this, watch out for weird addressing, here:
    ; this lives at location 20h! used to let ISR talk to MAIN

This has its own weirdness, though—as noted in the comment, above: the “address” of this bit is an 8-bit value describing the bit’s offset from the base address of 20h. Thus 0 is the lsb of 20h, 1 is d1 of that byte; 8 is the lsb of location 21h; and so on. But you needn’t think about this: you can specify a bit by its number, as in the SAMPLEFLAG definition just above.

Then the ISR is as tiny as this:

    ;--Interrupt Response: ISR----------
    ORG 03h ; this is INTO vector address
    SETB SAMPLEFLAG ; set flag bit, to inform Main
    RETI
3 Do_A, Do_B, using Keypad (5 pts)

Now, let’s do the same task, but using your little computer’s keypad. Let your computer call “DO_A” when you press the key “A” followed by the WR key (which sets the READY flag in your computer, data line seven of port 1—see Lab μ3 or the Big Picture); let it call “DO_B,” if you have pressed B and then WR; let the program just loop back and check again if neither key is pressed. Note that the most recent key value appears as the 4 LSB’s of the byte that the keypad presents to the computer at PORT 0. (This task calls for only code. Include comments, please.)

Suggestions: the only compare operation that the 8051 offers probably can be useful here: “CJNE A, #REFERENCE, DESTINATION”, for example. This compares the value of the accumulator against the immediate “REFERENCE” constant, and jumps to the label “DESTINATION” if the two values don’t match. (CJNE is not restricted to the A register, by the way. See the instruction set reference for details.)

If you’re interested in the value of less than all of the accumulator, you’ll have to mask out what doesn’t interest you. Masking is described in Class notes μ3.

Note on ways to handle more than one port address

<table>
<thead>
<tr>
<th>There’s more than one way to take care of this:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• you might use a DPTR to do business with one port, then re-load it with a new port value;</td>
</tr>
<tr>
<td>• you might increment the DPTR (if the two ports are adjacent) in order to use it for the second port; then you could decrement DPL, the low half of the dptr: DEC DPL (strange to tell, there is no “DEC DPTR” operation!).</td>
</tr>
<tr>
<td>• you might use the MOVX @Rn addressing mode—which forms the address not from DPTR but from P2 paired with either R0 or R1. This way of doing business with off-chip locations is described in a note called “8051 Addressing Modes.”</td>
</tr>
<tr>
<td>• You may want to take advantage of the your processor’s second DPTR. You invoke that by setting a bit that is the LSB of a register called “Data Pointer Select” (DPS). After DPS has been set high, to use this data pointer just refer to it as “DPTR” as usual. Only DPS distinguishes the two DPTR’s.</td>
</tr>
</tbody>
</table>

4If this isn’t confusing enough, the 8-bit halves of the DPTR do, however carry different names: DPH, DPL for the main pointer; DPH1, DPL1 for the secondary pointer.
4 Reaction Timer, done with Verilog (5 for design, 3 points extra credit if you demonstrate it (any time))

This week we invite you to use Verilog to implement a reaction timer like the one that some people designed for the project lab—except that we have simplified it somewhat by asking for natural-binary rather than decimal output.

We have started the design file, doing the easy part: we have implemented PROCEED. We have also attached, to the end of these HW papers, a solution to the reaction timer problem. You need not follow this exactly; but provide a design that works.

The Verilog template file includes inputs for a debouncer; one of the outputs of this logic is the signal, “debounced,” which is used in the PROCEED logic: when it goes high, it clocks a flop to start the reaction timer.

4.1 Debouncer (1 pt)

Show (pencil and paper) logic to use $S^*$ and $R^*$ to form a cross-coupled-NAND debouncer, driven by an SPDT (double-throw) pushbutton. (This debouncer will operate on the STOP pushbutton.)

```verilog
input s_bar,
input r_bar,
output debounced_stop,
output debounced_stop_bar;

// debouncer
assign debounced_stop = ;
assign debounced_stop_bar = ;
```

4.2 ...the rest (4 pts)

Now we’d like you to write the full verilog file, including the debouncer done by hand just above. The template file appears at the end of these notes, and is posted on the course website.

**We do the easy part:** a PROCEED flip-flop that can make the counter count. This flop is clocked by $\text{start}_{\text{bar}}$ (sending its Q high); it is cleared by the debounced $\text{stop}$ pushbutton, or by the overall clear, called (appropriately) $\text{clr}_{\text{all}}_{\text{bar}}$.

```verilog
always@ (negedge start_bar or negedge debounced or negedge clr_all_bar)
begin
  if (!clr_all_bar) // this is the overall clear...
    proceed <= 0;
  else
    if (debounced_stop) // this is resetting by the debounced STOP pushbutton
      proceed <= 0;
    else
      proceed <= 1; // this is the case for edge-triggering by start_bar
end
```

We have used “debounced” to clear the $\text{proceed}$ signal, which we presume will be used to determine whether the counter increments or holds its present value. We did this because in our scheme for cheat, we needed to debounce the $\text{stop}$ signal (your scheme may differ).
4.3 Code we’d like you to write

We would like you to finish the file. Specifically, we’d like you to add:

```vhdl
// COUNT LOGIC
// OVERFLOW LOGIC
// CHEAT LOGIC
```

You will see sections in the template file for each of these. Each is of the form “always @(...).” The template file and test bench are posted, as usual. We’d like you to submit your .v file and your simulation waveform results.

4.4 ...Building It (3 points)—no particular deadline, for this

4.4.1 Wiring It

We’ve assigned pins, in such a way that you can burn the reaction timer into a PLD and you then could plug that PLD into the breadboard that you used for counter and memory labs. Only a few control signals have changed function. You could leave all the counter Q’s wired where they were, so that you can see the count on your LCD display. We will provide the necessary .UCF file to assign your signals to appropriate pins, when it’s time to burn the part.

But note that we have wired up one breadboard that way, to save you the trouble. You can use ours. All you will need do, to try your reaction timer code, is burn a PAL and install it in our board.

![Figure 1: You need to rewire only a half dozen lines to the counter PAL](image)

You would need to add 3 LED’s as outputs, and an SPDT switch to the s_bar and r_bar inputs (for debouncing); an SPST switch to start (never mind debouncing that). Since all those 3 pins are simply pulled up in the Lab D3 wiring (so long as you don’t hold them at ground, with the DIP switch), there’s no need to remove the existing wiring.

The new signals using oe* and cin* do require removing the ground connection that drove those points in Lab D3.

4.4.2 Burning the Part

In lab, you can “burn” (program) a PLD with your reaction timer code, and try it out.

Template files

```
`timescale 1ns / 1ps
```
module reaction_timer_april2011(
    input clk,
    input start,
    // next lines replace stop with debouncer SR latch
    input s_bar,
    input r_bar,
    output debounced_stop,
    output debounced_stop_bar,
    input clr_all_bar,
    output reg [15:0] count, // all these are flop outputs, and are class "reg," though "reg" does not quite mean "flop"
    output wire c_out_bar, // ...except this is the one combinational output
    output reg proceed,
    output reg ovflw,
    output reg cheat
);
// next line for full-size 16-bit counter
assign c_out_bar = !(proceed & (count == 16'hFFFF));
assign c_out_bar = !(proceed & (count == 16'b1111111111111111));

// debouncer
assign debounced_stop = ~s_bar | ~debounced_stop_bar;
assign debounced_stop_bar = ~r_bar | ~debounced_stop;
// next line for warmup, on 3-bit counter
assign c_out_bar = !(proceed & (count == 3'b111));

// PROCEED
always@(posedge start or posedge debounced_stop or negedge clr_all_bar)
    begin
        if (!clr_all_bar) // overall clear (this clears all flops and the counter)
            proceed <= 0;
        else
            if (debounced_stop) // this is the case when the stop pushbutton has been hit
                proceed <= 0;
            else
                proceed <= 1; // this is the case for edge-triggering by start
    end

// COUNT
always@(posedge clk or negedge clr_all_bar)
    begin
end

// OVERFLOW
always@(posedge c_out_bar or negedge clr_all_bar)
begin

end

// CHEAT
always@(posedge debounced_stop or negedge clr_all_bar)
begin

end

endmodule