University of British Columbia
Electrical and Computer Engineering
Digital Systems and Microcomputers
CPEN312

Tutorial: How to do Lab 5

This tutorial is intended to help you get started with lab 5. Please note that as with any program, there are many possible different and equally valid implementations for the same functionality. You can write your program any way you want as long as it works correctly! I also want to point out that we can only display six decimal digits using the DE0-CV board, although we have up to ten decimal digits available when using 32-bit arithmetic.

As noted in the lab requirements, you have three programs available to help you with the lab:

1) ***Read_sw6.asm:*** This program reads BCD numbers from the DE0 board switches and shows them in the 7-segment displays. This could be the starting point for your program.
2) ***math32.asm:*** A library of 32-bit arithmetic functions as well as BCD-to-binary and binary-to-BCD conversion routines. We should have covered addition, subtraction, BCD to binary and binary to BCD in class already. These are the same routines we did in class.
3) ***math32test.asm:*** A program that shows how to use the functions in the ***math32.asm*** library. It also shows how to use some of the macros defined in ***math32.asm***.

Let us start by making a copy of ***Read_sw6.asm***. I will refer to that copy as ***calc32.asm***. This new program will be our starting point. The first thing we have to do is to add the library ***math32.asm*** to our program. The file ***math32test.asm*** shows what we need to do: include these few lines of code at the beginning of ***calc32.asm*** replacing lines 6 and 7:

```
dseg at 30h

x:    ds    4 ; 32-bits for variable 'x'
y:    ds    4 ; 32-bits for variable 'y'
bcd:  ds    5 ; 10-digit packed BCD (each byte stores 2 digits)

bseg

mf:         dbit 1 ; Math functions flag

$include(math32.asm)
```

Note: the lines of code above replace these two lines (lines 6 and 7):

```
DSEG at 30H
bcd:  ds 5
```

Save and compile. If there are any errors fix them. At this point we have all we need to start programming our calculator. We also need to be well aware of the requirements for the lab so we can plan how to arrange our program properly. We know from the requirements that:

```
'Function Select' is KEY.3
'Load Number' is KEY.2
'=' is KEY.1
```

Fortunately, the KEY.* pushbuttons are easy to work with because we can use the JB and JNB instructions to check them. If a pushbutton is pressed it reads as logic zero. If a pushbutton is not pressed it reads as logic one.

Now let us try to implement a simple calculator that just does addition. Start with the original 'forever' loop code in the program:

```
forever:
      lcall ReadNumber
      jnc forever
      lcall Shift_Digits
      lcall Display
      ljmp forever
```

It helps to add comments with the steps we need to follow to complete each task. It helps even more if you add the comments in the form of pseudocode[1]:

```
      ; Set addition as default operation after reset
forever:
      ; Display operation selected in LEDs 5..0
      ; if 'Function' key pressed then
      ;     increment function select mod 6
      ;     go to forever
      ; elsif 'Load Number' key pressed then
      ;     wait for 'Load Number' key to be released
      ;     convert BCD to binary
      ;     save number
      ;     go to forever
      ; elsif '=' key pressed then
      ;     wait for '=' key to be released
      ;     convert BCD to binary
      ;     perform selected operation
      ;     convert result to BCD
      ;     display result
      ;     go to forever
      ; else
      ; Read more decimal digits
      lcall ReadNumber
      jnc forever
      lcall Shift_Digits
      lcall Display
      ljmp forever
```

[1] Wikipedia has an excellent article on pseudocode. For some reason the style I used looked like VHDL, but any style will work just fine!

At this point we can start working on each step of the one-function calculator. Here are all the steps implemented from the pseudocode above:

```
      mov b, #0            ; b=0:addition, b=1:subtraction, etc.
      setb LEDRA.0         ; Turn LEDR0 on to indicate addition
forever:
      ; This is a good spot to set the LEDs for each operation...
      jb KEY.3, no_funct   ; If 'Function' key not pressed, skip
      jnb KEY.3, $         ; Wait for release of 'Function' key
      inc b                ; 'b' is used as function select
      mov a, b             ; make sure b is not larger than 5
      cjne a, #6, forever  ; ^
      mov b, #0            ; ^
      ljmp forever         ; Go check for more input
no_funct:
      jb KEY.2, no_load    ; If 'Load' key not pressed, skip
      jnb KEY.2, $         ; Wait for user to release 'Load' key
      lcall bcd2hex        ; Convert the BCD number to hex in x
      lcall copy_xy        ; Copy x to y
      Load_X(0)            ; Clear x (this is a macro)
      lcall hex2bcd        ; Convert result in x to BCD
      lcall Display        ; Display the new BCD number
      ljmp forever         ; Go check for more input
no_load:
      jb KEY.1, no_equal   ; If 'equal' key not pressed, skip
      jnb KEY.1, $         ; Wait for user to release 'equal' key
      lcall bcd2hex        ; Convert the BCD number to hex in x
      mov a, b             ; Check if we are doing addition
      cjne a, #0, no_add   ; ^
      lcall add32          ; Perform x+y
      lcall hex2bcd        ; Convert result in x to BCD
      lcall Display        ; Display the new BCD number
      ljmp forever         ; Go check for more input
no_add:
      ; Other operations maybe coded here
no_equal:
      ; get more numbers
      lcall ReadNumber
      jnc no_new_digit     ; Indirect jump to 'forever'
      lcall Shift_Digits
      lcall Display
no_new_digit:
      ljmp forever ; 'forever' is to far away, need to use ljmp
```

Note that the instruction:

```
      jnb KEY.3, $   ; Wait for user to release key 3
```

is an equivalent but more compact version of:

```
somelabel:
      jnb KEY.3, somelabel  ; Wait for user to release key
```

To add extra operations just keep checking the value of register b and add the code for each operation after label '`no_add`'. Don't forget to exchange ***x*** and ***y*** before subtraction, division, and remainder as these functions are not commutative. Actually you may as well exchange ***x*** and

y for all the operations (except integer square root), because addition and multiplication are commutative. In the ***math32.asm*** library there is an ***xchg_xy*** function you can use for this purpose.

Well, that should get you going. Once again, there are many different ways to implement the same functionality. By the way, I had left out many optimizations for the sake of clarity. Hopefully this short tutorial can help you with your own implementation of the program for lab 5.

One more thing: the library provided implements subroutines for addition, subtraction, multiplication, and division. You'll need to write your own code for the 'remainder' and 'square root' subroutines.