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

Tutorial: How to do Lab 6

This tutorial is intended to help you started with lab 6. 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:

```
+ is KEY.3 when KEY.0 is NOT pressed
- is KEY.2 when KEY.0 is NOT pressed
= is KEY.1
* is KEY.3 when KEY.0 is pressed
/ is KEY.2 when KEY.0 is pressed
```

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]:

```
forever:
	; if '+' key pressed then
	;     wait for '+' key to be released
	;     convert BCD to binary
	;     move x to y
	;     clear x
	;     convert x to BCD
	;     display BCD
	;     go to forever
	; elsif '=' key pressed then
	;     wait for '=' key to be released
	;     convert BCD to binary
	;     call add32
	;     convert x to BCD
	;     display BDC
	;     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:

```
forever:
      jb KEY.3, no_add    ; If '+' key not pressed, skip
      jnb KEY.3, $        ; Wait for user to release '+' key
      lcall bcd2hex       ; Convert the BCD number to hex in x
      lcall copy_xy       ; move x to y (this is a function)
      Load_X(0)           ; clear x (this is a macro)
      lcall hex2bcd       ; Convert binary x to BCD
      lcall Display       ; Display the new BCD number
      ljmp forever        ; Go check for more input
no_add:
      jb KEY.1, no_equal  ; If '=' key not pressed, skip
      jnb KEY.1, $        ; Wait for user to release '=' key
      lcall bcd2hex       ; Convert the BCD number to hex in x
      lcall add32         ; Add the numbers stored in x and y
      lcall hex2bcd       ; Convert result in x to BCD
      lcall Display       ; Display the new BCD number
      ljmp forever        ; Go check for more input
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
```

As pointed out in class, the instruction

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

is an equivalent but more compact version of:

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

The code above works fine for a one-function calculator, but we need to implement four functions: addition, subtraction, multiplication and division. In principle it should be easy to add more functions by just copying and pasting the code for the ‘addition’ and renaming accordingly, but there are a couple of problems we have to solve:

Problem 1: There are not enough pushbuttons in the DE0-CV board for each operation we need. From the requirements we know that KEY.3 is used both to select addition and multiplication depending on the state of KEY.0. The same situation exists for subtraction and division with KEY.2.

Problem 2: What happens when we press the ‘equal’ key? How do we know what operation we need to call?

Both these problems can be solved by remembering the function the user selected with an extra variable (or register if any are available). Let start by defining a one byte variable at the top of our program, in the same section where ***x***, ***y***, and ***bcd*** are defined:

```
operation:  ds 1
```

For the addition/multiplication part of the code we make bit 0 of ***operation*** equal to one for addition or bit 2 of operation equal to one for multiplication (just before the '`ljmp forever`' preceding the '`no_add:`' label):

```
      jnb KEY.0, is_mult
      mov operation, #0000_0001B
      ljmp forever
is_mult:
      mov operation, #0000_0100B
      ljmp forever
```

Similarly, for subtraction and division:

```
      jnb KEY.0, is_div
      mov operation, #0000_0010B
      ljmp forever
is_div:
      mov operation, #0000_1000B
      ljmp forever
```

In the '=' section we can check the bits of '***operation***' and jump to the corresponding section of code:

```
      jb KEY.1, no_equal ; If the '=' key not pressed, skip
      jnb KEY.1, $       ; Wait for user to release the '=' key
      lcall bcd2hex      ; Convert the BCD number to hex in x
      ; Select the function the user wants to perform:
      mov a, operation   ; The accumulator is bit addressable!
      jb acc.0, do_addition
      jb acc.1, do_subtraction
      jb acc.2, do_multiplication
      jb acc.3, do_division
```

Of course do_addition, do_subtraction, etc. have to be implemented somewhere in the program. For example:

```
do_addition:
      lcall add32   ; Add the numbers stored in x and y
      lcall hex2bcd ; Convert result in x to BCD
      lcall Display ; Display BCD using 7-segment displays
      ljmp forever  ; go check for more input
```

Also, don't forget to exchange ***x*** and ***y*** before subtraction and division as these functions are not commutative. Actually you may as well exchange ***x*** and ***y*** for all the operations, 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 6.