Microprocessor Systems Design ECS502U
Lab 2 (Peripheral Interfacing) Report
Hana Emma Hadidi

Contents:

Abstract:
The results of the ASM lab code in the .asm file and parts of which are included in this report, enables the user to press values on the keypad which is displayed on the screen and shifts accordingly. Greatest limitation of the final asm code is it is affected by debounce which results in a long key press printing the same value on the screen multiple times.

Introduction:
This lab report explains how the 8051 in conjunction with the 8255 Peripheral Interface Adapter can be used to allow for input from the keypad and present on the display interface. The hardware used is a microcontroller, and a keypad with a display interface shown in Figure 1, the software used was the MCU IDE to compile the ASM code and a HyperTerminal to download the code onto the microcontroller. Part A provides an analysis to the ASM code and a discussion on ways to increase the efficiency of the code or alternative hardware that could be used. Part B provides an explanation on how to extend the ASM code to allow for the keypad to act as an arithmetic calculator.

Part A:

Aim:
The function the ASM code should perform is being able to detect when there has been a keypress and have a debounce system to ensure the same value isn't displayed multiple times. The value pressed on the keypad should be correctly identified depending on the columns and rows and be shifted accordingly across R0-R3 and be shown on a multiplexed seven-segment display. If the key 'F' is pressed it should clear registers R0-R3.

Background:
Utilising the 8255 there are an increased number of ports, Port C is used to read from the keypad, Port A is used to provide the configuration of the LED for values 0-9 and A-F and Port B is used to manage on which of the 4 displays the values will appear on. Table 1 illustrates the combination of clearing or setting A1 pin and the A0 pin to access the different ports and control data bus. Figure 1 depicts the keyboard used; this is important as it establishes a 4x4 matric which is important to check for all possible combinations. Figure 2 illustrates how one byte should be inputted into P1, with the lower byte used to find rows and upper byte for columns. It is also important to establish the column inputs are pull down resistors hence, if a button is pressed while the row is high "1111" it will return "0", rather than "1"[1].

A_1	A_0	$\overline{RD}$	$\overline{WR}$	$\overline{CS}$	Input Operation (Read)
0	0	0	1	0	Port A - Data Bus
0	1	0	1	0	Port B - Data Bus
1	0	0	1	0	Port C - Data Bus
1	1	0	1	0	Control Word - Data Bus
					Output Operation (Write)
0	0	1	0	0	Data Bus - Port A
0	1	1	0	0	Data Bus - Port B
1	0	1	0	0	Data Bus - Port C
1	1	1	0	0	Data Bus - Control
					Disable Function
X	X	X	X	1	Data Bus - 3 - State
X	X	1	1	0	Data Bus - 3 - State

Table 1. Used for Port Access allowing read and write to Port A, B, C and the Control data bus.[1]

Fig 1. The Keyboard and Display Interface Module

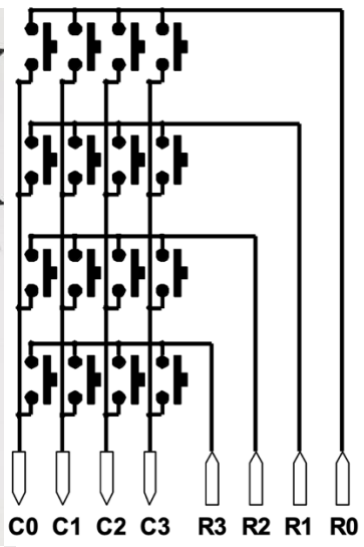
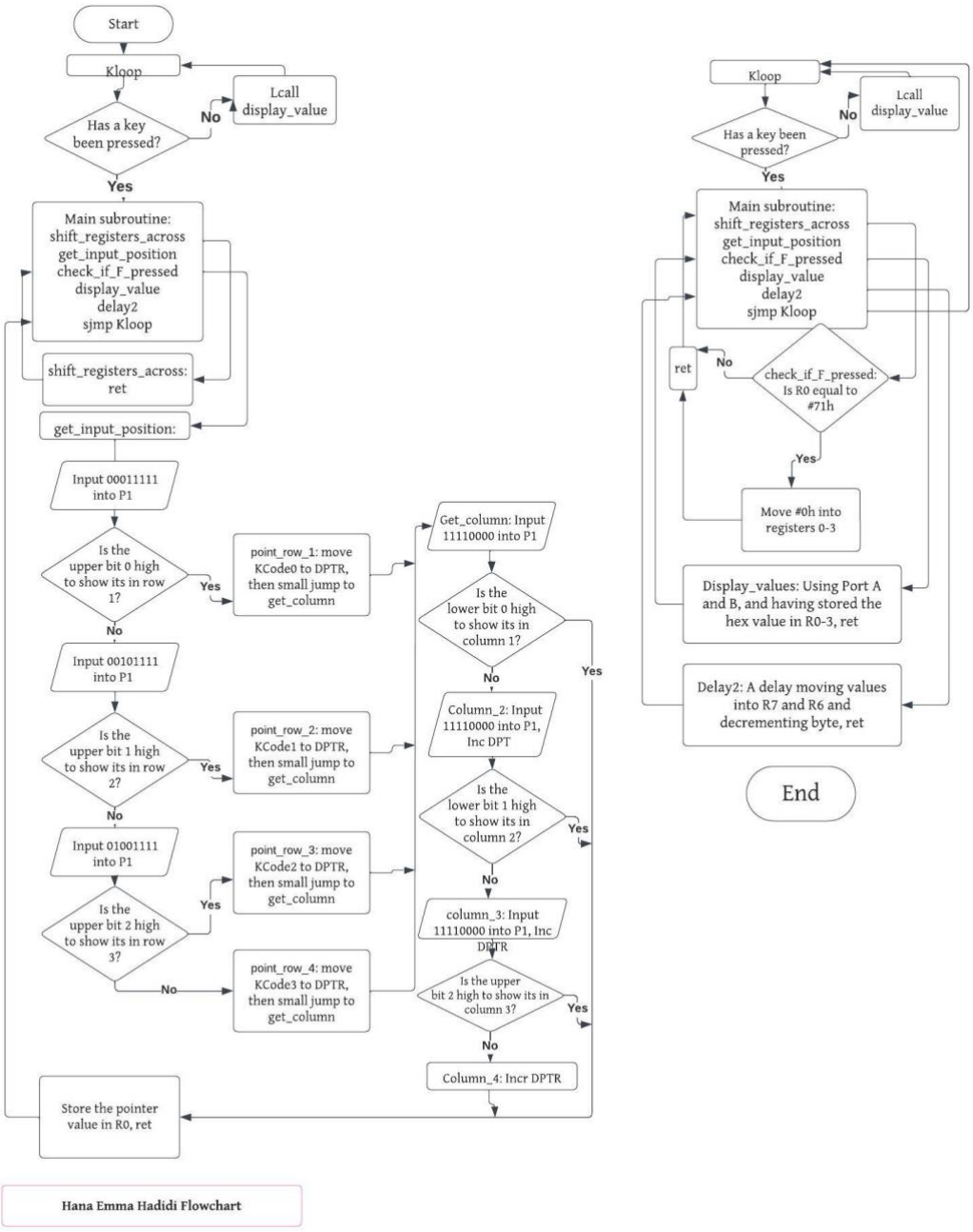


Fig 2. Block Diagram for Keypad Interfacing[2]

Flowchart:

Fig 3. A flowchart of the asm code showing the structure and brief description of each function.



Analysis of the ASM Code:

Parts of the of the code have been discussed below based on its purpose and the conscious decisions made to increase efficiency or readability.

```
RDpin equ 0B3h
WRpin equ 0B2h
A0pin equ 0B5h
A1pin equ 0B4h
```

Fig 4. Prior to the start of the programme to aid the readability of the code the hex values for read pin (RDpin), write pin (WRpin), A0 pin (A0pin) and the A1 pin (A1pin) are equated using the 'equ'. This allows for 'RDpin' to be used instead of 0B3h in the programme.

```
org 8000h

KCODE0: db 06h,5Bh,4Fh,71h
KCODE1: db 66h,6Dh,7Dh,79h
KCODE2: db 07h,7Fh,67h,5Eh
KCODE3: db 77H,3Fh,7Ch,58h
```

Fig 5. 'Org 8000h' is used to give the origin address for the lookup table. The lookup table is organised according to the 4x4 matrix keyboard shown in Figure 1with each 'KCODEO' correlating to a row on the keyboard. The hex values used represent the value on the keyboard by showing the correct configuration of the a-g seven-segment required for the LED (see Figure 5). In binary it is easier to see which a-g lights up by checking which bit is high (bit-0 corresponds to a, bit-1 to b, etc).

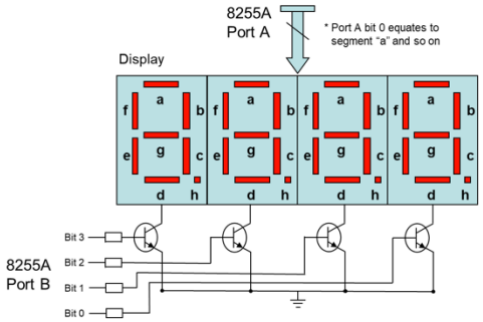


Fig 6. Showing the display of the 8255A and the respective relation between a-h and bit 0-8, bit 8 is not utilised in Part A.[1]

```
org 8100h
start:
setb A0pin
setb A1pin
mov P1, #81h
Lcall write
mov R0,#00h
mov R1,#00h
mov R2,#00h
mov R3,#00h
```

Fig 7. 'Org 8100h' is used to give the origin address for the start of the program. To configure the 8225 register setb is used to make the A0 pin and A1 pin high, meaning we can talk to the Control word data bus. Method 1 is required for strobed input/output and to make use of port A, B and C7-C4 in output and C3-C0 as input. To do this 81h is moved into P1 using 'Lcall write', which sets the mode flag 1 as active. 00h Is moved into all registers to ensure they are cleared at the start of the program.

```
Kloop:

clr A0pin
setb A1pin
mov a,#0FFh

mov P1, a
lcall write
clr RDpin
mov a,P1
setb RDpin
anl a,#0Fh

cjne a,#00h, main
acall display_value
sjmp Kloop
```

Fig 8. The label 'Kloop' is used to constantly check for a keypress using the polling method, if there has been it will branch out to 'main', otherwise it calls the display with the label 'display_value' and then calls itself. 'Setb A1pin' and 'clr A0pin' are used to enable communication with Port C, 0FFh is moved into P1 via the accumulator which writes all 1's to P1, to prepare the lower nibble for read access. 'clr RDpin' transfers the values into P1 from Port C. The values of P1 are moved into the accumulator, and 'setb RDpin' reverses the previous clr RDpin from earlier. Though lower nibble is configured as high, they are not affected when read, to prevent impacting any future tests the lower nibble is masked using anl a, #0Fh, which writes 1's to the lower nibble. Jump if not equal ('cjne') compares if the accumulator is equal to #00h which assumes no key has been pressed, if that's true it will continue to 'Lcall display_value' which is a long call to the label which displays values in R0-4 on the Display interface module, allowing for the values to be seen even without a keypress. After it returns using 'sjmp Kloop' (small jump) to itself to create a loop. If it's not true, then 'a' stores the destination of the keypress and a jump to the label 'main' is made.

```
main:
acall shift_registers_accross
acall get_input_position
acall check_if_F_pressed
acall display_value
lcall delay2
sjmp kloop
```

Fig 9. ‘Main’ is a label used to do an absolute or long call to other labels that shift the register, translate the position of the keypress to the configuration of that value, check if the ‘F’ key is pressed, provide a delay to reduce bounce and call the display screen respectively. It then loops back to the ‘Kloop’ to check if there has been another keypress. ‘Acall’ and ‘Lcall’ utilise the stack and relies on the ret function to allow the program to know to go back to the output label after completing the subroutine.’Lcall’ is used for Delay2 as is it at the end of the code and may be outside the 2Kbyte range.

```
shift_registers_accross:
mov a,R2
mov R3,a
mov a,R1
mov R2,a
mov a,R0
mov R1,a
ret
```

Fig 10. This label is only executed if there has been a keypress, hence, to prevent any value from being override the values stored within registers need to shift, register addressing is used to achieve this. The ‘ret’ is pushed automatically onto the stack and points the program back to the ‘main’ label.

```
get_input_position:
mov p1, #00011111b
lcall write
lcall read
anl a,#00001111b
cjne a,#0h,  point_row_()
```

Fig 11. ‘get_input_position checks which row is high by changing the row that is high, done by doing ‘mov P1, #00011111b’ , which is then written into the port to drive the row high, and the value read is used to decide whether the key was pressed in that row. Since the keys are pulled low by resistors if the row is driven high and a 0 is returned, no key is pressed. Hence the ‘cjne’ which compares the read value stored in ‘a’ to #0h (zero), if its high (1) it will branch to pointer0, otherwise it will carry on checking the column read against different rows driven high, this is achieved by copying the code underneath each other and changing the configuration of the upper 4 bits moved into P1 (#00101111b and #01001111b). It can be assumed if the keypad is limited to 4 rows, that by process of elimination row 4 would simply be ‘sjmp pointer3’ which helps make the code more efficient as it avoids extra logic.

```
point_row_1:
   mov DPTR, #KCODE0
   sjmp get_column
```

Fig 12. ‘point_row_1’ represents the row 1 being high, therefore points the data register pointer (‘DPTR’) to the correlating KCode set. This provides the base to get the key pressed row, it then performs a small jump to the label ‘get_column’ so it can increment to the column that is high. DPTR is used as its highly versatile, it can point to either data memory or external memory. In this case it is used to access look-up tables.

```
get_column:
   mov P1,#11110000b
   lcall write
   lcall read
   anl a,#0Fh
   cjne a,#00000001b,column_2
   sjmp read_and_store_value
   column_2:
   inc DPTR
   mov P1,#11110000b
   lcall write
   lcall read
   anl a,#0Fh
   cjne a,#00000010b,column_3
   sjmp   read_and_store_value
;…
```

Fig 13. ‘get_column’ finds which column is high, by making all rows high (‘mov P1,#11110000b’, which is written using ‘lcall write’) and check the read value against configuration in the lower nibble for different columns. This is done by ‘cjne a, #00000001b’, which is then written into the port to drive the rows high, and the value read is used to decide whether the key was pressed in that row. Since the keys are pulled low by resistors if the row is driven high and a 0 is returned, no key is pressed. Hence the ‘cjne’ which compares the read value stored in ‘a’ to #00000001 (column 1 high), if column 1 is high it will carry on to ‘sjmp read_and_store_value’, otherwise branches to test column 2. The code is changed to test for different columns by changing the lower nibble that is compared to ‘a’. For example, column 2 would be ‘cjne a, #00000010b, column_3’. It can be assumed if the keypad is limited to 4 columns, that by process of elimination column 4 would simply be ‘read_and_store_value’, which helps make the code more efficient as it avoids extra logic. Column 2 code is the same as get_column with the addition of starting with incrementing the DPTR (‘inc dptr’), to act as the shift required for the DPTR to point to the correct hex value in the KCode row.

```
read_and_store_value:
    clr A
    movc A,@A+dptr
    mov r0, A
    ret
```

Fig 14. ‘movc’ is used to store the contents on the DPTR into the register A using indexed addressing where the offset has already been added to the DPTR during the column checks with ‘inc DPTR’. ‘movc’ Has to be used instead of ‘mov’ as the DPTR is a special 16-bit register and to move the address into the ROM, the ‘c’ in ‘movc’ stands for code. In this case as ‘clr A’, equate register A to 0, therefore after ‘movc A,@A+DPTR’ . Register A gets the char value from the code space and stores it in ‘a’, then it is the stored in register 0.

```
check_if_F_pressed:
    cjne r0,#71h,return_to_main
    mov R0,#00h
    mov R1,#00h
    mov R2,#00h
    mov R3,#00h
    return_to_main:
    ret
```

Fig 15. To complete the task of clearing the display when “F” is pressed, it compares R0 to the hex configuration that displays F. If its untrue it returns to ‘main’, otherwise it moves the hex #00h into Registers 0-3, then returns to ‘main’.

```
display_value:
;write digit R0 to port A
    clr A0pin
    clr A1pin
    mov     P1, R0
    Lcall write
    ;activate digit R0 (port B)
    setb A0pin
    clr A1pin
    mov     P1, #00000001b
    Lcall write
    mov p1,#00000000b
    lcall write
```

Fig 16. To display the value, Port A is used to load in the configuration of the number, which is already stored in Registers 0-3. Port B has a constant binary value moved in P1 depending on which quarter segment of the display should display the value. Since R0 is the latest input in the rightest segment, this is done by having bit 0 high of the byte moved into P1. R1, R2 and R3, have bit 1,2 and 3 high respectively, of the byte moved into P1 for Port B. A ‘ret’ calls the end of the subroutine back to the main.

```
delay:
            mov r7,#7fh
delay1:
            mov r6,#0ffh
delay2:
            djnz r6,delay2
            djnz r7,delay1
            ret
```

Fig 17. Delay2, utilises Registers 7 and 6 to store large hex values (127_{10} and 255_{10} respectively) and used ‘djnz’ decrement the location of memory by one until 0 is reached, if it holds a non-zero value it will do a jump to label. Delay 2 was chosen as it had the greatest impact in decreasing the bounce that occurred when pressing the keypad.

Discussion:

The code is successful in showing the key pressed value on the screen, shifting values and clearing the display if ‘F’ is pressed. However, it only works effectively the keys are pressed quickly as it suffers from debounce. The “Delay2” subfunction is successful in allowing for 1 integer on the display if keys are pressed relatively fast and dealing with rapid keypress, however it was found additional delays results in lagging on the display and skipping keys if rapidly pressed. It is possible to get rid of debounce by not checking the switch for certain amount of time, but this wasn’t implemented as attempts didn’t give any output of the display segments. Another way of getting rid of the pulses on the switch is the implantation of alternative hardware such as a Schmitt Trigger. It is a switch that only goes either 0-1 or 1-0 and doesn’t switch back for a set amount of time based on its internal hardware.

The “Kloop” used to check of a keypress uses the polling method, it is inefficient as it constantly checks for if the keypad is pressed, an alternative to this is the use of interrupts which are a part of Port 3. Port 3 has 2 pins called external interrupts and either can be used; they function by being low until a key is pressed. This is more efficient as it allows the code to complete other programmes or sub function until a key is pressed rather than constantly looping around to check. When a key is pressed the processor runs an interrupt subroutine which in this case would be pointing the DPTR to the correct value and storing it in R0.

The code is designed to use the “main” subfunction to do absolute call to each function and long call ‘Delay2’. This enables testing for each function to be individually tested, however the code can be further split to return after getting the row and column and then storing the value, to increase clarity of the code.

Conclusion:
In conclusion, the code runs successfully to an extent as it can cope with rapid keypress and achieves the basic aim of shifting values displayed on the display interface and clearing when "F" is pressed. Improvements to the asm code to reduce debounce can be done via accessing a Port 3 external interrupt and swapping the switch in the keypad from Figure 1 to a Schmitt Trigger.

Part B:

Aim:
To extend the application from Part A to function as a simple arithmetic calculator, the code needs to be able to identify letters 'B-E' as arithmetic symbols and 'A' as an equal. It needs to be able to correctly display the value when equal is pressed.

Background theory:
The "B" which is a part of the special function register accumulator is required to use the multiplication and division function in the 8051. The arithmetic functions are "Add" for addition, "Sub" for subtraction, "Mul" for multiplication and "Div" for division. The registers can't have arithmetic functions be done on them directly hence a accumulator is used. Figure 18 below shows the working out for add. Figure 19 shows the keyboard layout assumed for Part B. Since division is done for binary configuration for the display "a-h" is considered as in division a decimal point may be required.

```
Mov a , R0
Mov B, R1
Mul AB
```

Figure 18 shows how multiplication is done between registers via the accumulator "a" and "b".

1	2	3	Clear
4	5	6	+
7	8	9	-
=	0	÷	x

Fig 19. Arithmetic keyboard layout assumption

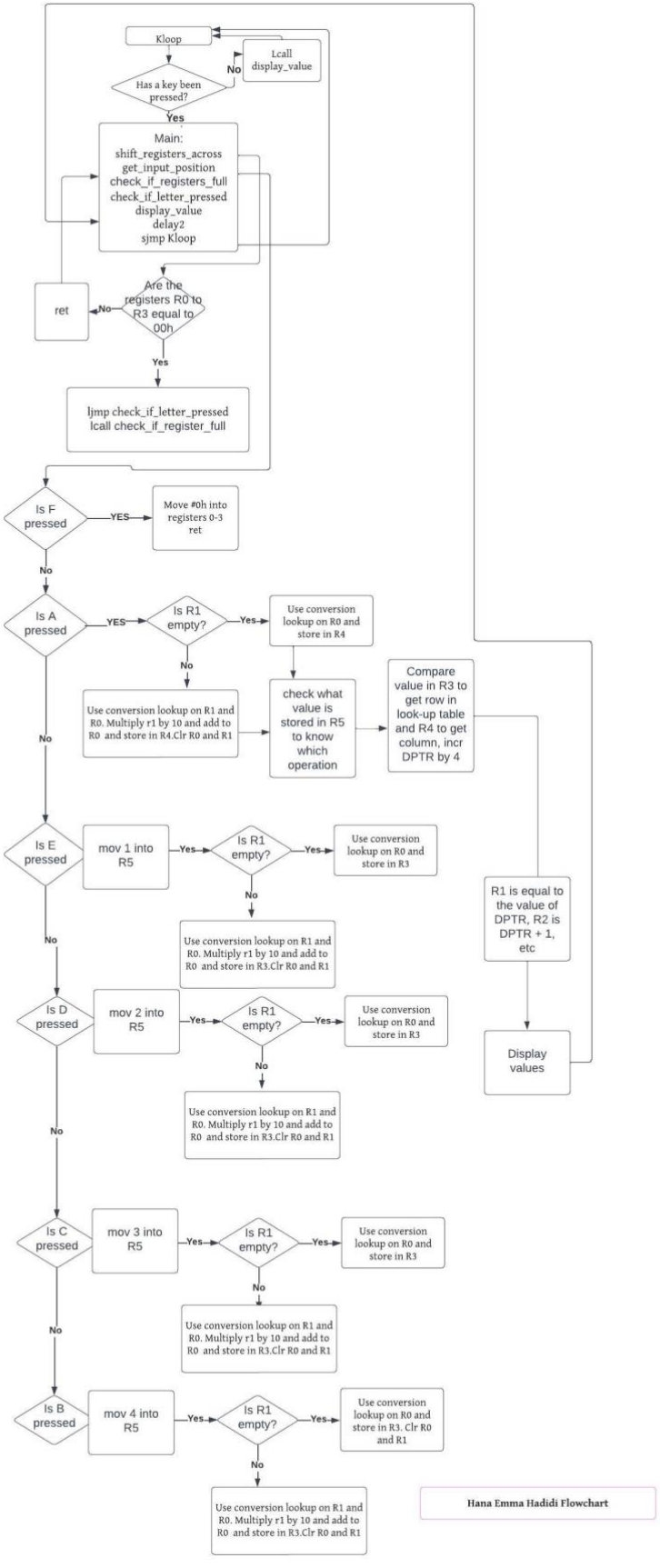
Theory:
Since the display interface only has 4 segments to display a number on the calculator, this theory to achieve the aim assumes a maximum input of 2 digits per turn (R0 and R1). This is to prevent numbers greater than 1000 significant figures appearing, as the maximum multiplication would be 99x99= 9801. (Figure 20 should be used to gain a better understanding of the structure of the arithmetic code explained below.)

After a button is pressed it will do a check for letters "A-F" and branch out depending on the functions of each letter or if it's a number it should check if R1 is empty or not. If R1 is not empty, should only convert values stored into R0 to its hex value rather than storing the configuration of the number, otherwise should do both. This can be done via comparing it upper and lower nibble to point the DPTR to the correct value on the conversion look-up table. If R1 is not empty it should multiple R1 by 10 using 'mul', adding R0 to it and storing the value in R3, using the accumulator 'a' to achieve this. Otherwise, it should make a jump and just store R0 in R3 after the conversion. If R1 is not empty it should loop to only recognize the input if it's an alphabetical as it shouldn't allow for numerical inputs. "F" will clear R0, R1 and R4 which is used to store the total amount. "A" represents equal, it should convert the latest input and store it R4. If 'B-E' are pressed it should clear the register R0-R1. It should then check the register R3 to the look-up table of that arithmetic symbol and treat it similar to

how row was found but comparing all bits. It should point the look-up row to that value. It should then compare all the bits in the R4 and increment by 4 the DPTR, to point to the correct final value. The pointed to value is stored in R1, the DPTR is incremented and stored in R1, incremented again and stored in R2 and once more and stored in R3. The total has been given already in the display configuration. Hence when 'A' is called it will display the result.

Flowchart:

Fig 20. Flowchart of the arithmetic code



Discussion:

The theory proposed does not utilise the functions that are provided within the 8051, which would directly perform arithmetic operations on the values. A limitation found when designing the code was converting the result back onto the configuration required for the display. Comparing every result using a look-up table was inefficient as the it may need to compare values in with 1000 significant figures. Another issue with doing direct operations is that registers can only store 8-bit values, so which multiplying or adding with greater values there may be an unaccounted-for carry. However, preparing each look-up table may be considered time consuming as well, while it is limited to values 0-99. A way to allow for greater inputs is by using a liquid crystal display which will enable calculations with greater numbers as it can display results to a greater significant figure by being able to do x10 to a power and it is far easier to convert the answer to the display as it has internal latches with 2416 characters[2.] The theory is limited into to a maximum input of 99 and minimum 0, as the display can only support 4 significant figures.

Conclusion:

In conclusion the theory provided is limited in effectiveness due to both limitations in the hardware display interface module and the amount of machine cycles required to do the comparisons and branch correctly. It is limited into to a maximum input of 2 significant which prevents complex arithmetic and useability.

References:

1. Chris Philips ECS502U, Lab Sheet 2 “Microprocessor Systems Design - Lab 2 (Peripheral Interfacing)”, 30/10/2023
2. Chris Philips, ECS502U Topic 6: Basic Interfacing Lecture slides