CLASS 7 HOMEWORK

- ## CHAPTER 3 EXERCISES:

1. Formula we use to convert Virtual Address (VA) to Physical Address (PA) is:

   PA = VA & 0x1FFFFFFF

   (a.) 0x80000020: PA = **0x00000020**
   This memory location is ***cacheable***, and it resides in the ***RAM***.

   (b.) 0xA0000020: PA = **0x00000020**
   This memory location is ***noncacheable***, and it resides in the ***RAM***.

   (c.) 0xBF800001: PA = **0x1F800001**
   This memory location is ***noncacheable***, and it resides in the ***SFRs***.

   (d.) 0x9FC00111: PA = **0x1FC00111**
   This memory location is ***cacheable***, and it resides in the ***Boot Flash***.

   (e.) 0x9D001000: PA = **0x1D001000**
   This memory location is ***cacheable***, and it resides in the ***Program Flash***.

2. The bootloader.ld installs your program at Virtual Address (VA): 0xBD000000 + 0x1000 + 0x970 = **0xBD001970**.

3. From the data sheet we get the *Input/Output* bits of the ports B-G:

   (a.) PORTB: **0-15** *bits*
   PORTC: **12-15** *bits*
   PORTD: **0-11** *bits*
   PORTE: **0-7** *bits*
   PORTF: **0-1** *bits and 3-5 bits*
   PORTG: **2-3** *bits and 6-9 bits*

   From the pin diagram, we can see that pin 60 corresponds to bit 0 of port E (RE0).

   (b.) ***Unimplemented*** bits in the **SFR INTCON** are: **bits 5-7, bit 11, bits 13-15 and bits 17-31.**

   Implemented bits in the **SFR INTCON** are:

   INT(0-4)EP – bits 0-4
   TPC<2.0> – bits 8-10
   MVEC – bit 12
   SS0 – bit 16

4. Here we just make 1 change:

    LATFbits.LATF0 = 1;

    Code is attached along with the submission. Please find simplePIC.c

6. Virtual Addresses (VAs) and Reset values of the following SFRs:

    (a.) I2C3CON: *VA* – **0xBF805000** *Reset value* – **0x1000**

    (b.) TRISC: *VA* – **0xBF886080** *Reset value* – **0xF000**

7. So the processor.o file contains all the virtual addresses of the SFR for our particular pic32 model. Now in the final linking process when we create a .hex which is the final executable which we send over to the pic32. In the linking process the bootloader is responsible for actual memory assignment of the SFR which we've sent over to the pic32. So, the final stage creates a .hex file which is a couple of kBs only because they are stripped off of the information and operations of that the bootloader handles. So, this file is sent directly to the Program Flash as instructions.

8.
    (a.) The following are the lines of code that calls the user's main function and when the C runtime startup completes:

    ```
    and     a0,a0,0
    and     a1,a1,0
    la      t0,_main_entry
    jr      t0
    nop

    .end _startup
    ```

    (b.) Name and addresses of the ***5 highest address*** SFRs (*ascending order*):

    (i.) *VA*: **BF88CB4C** *SFR*: **C2FIFOCI31INV**
    (ii.) *VA*: **BFC02FF0** *SFR*: **DEVCFG3**
    (iii.) *VA*: **BFC02FF4** *SFR*: **DEVCFG2**
    (iv.) *VA*: **BFC02FF8** *SFR*: **DEVCFG1**
    (v.) *VA*: **BFC02FFC** *SFR*: **DEVCFG0**

    (c.) There are 10 bit fields in total inside two structures in the field data type **__SPI2STATbits_t**, which are:

    (i.) SPIRBF – **1** *bit*
    (ii.) SPITBF – **1** *bit*
    (iii.) SPITBE – **1** *bit*
    (iv.) SPIRBE – **1** *bit*
    (v.) SPIROV – **1** *bit*

(vi.) SRMT – **1** *bit*
(vii.) SPITUR – **1** *bit*
(viii.) SPIBUSY – **1** *bit*
(ix.) TXBUFELM – **5** *bits*
(x.) RXBUFELM – **5** *bits*

9. TRISDSET = **0b1100** or **0xC**

   TRISDCLR = **0b100010** or **0x22**

   TRISDINV = **0b10001** or **0x11**

- <u>CHAPTER 4 EXERCISES:</u>

1. NU32_DESIRED_BAUD is a global variable private to on NU32.c and all other functions and constants defined are not private to NU32.c and can be used by other C files.

2.
   (a.) Code attached along with the submission. Please find invest.c source code for reference.

   (b.) Codes attached along with the submission. Please find main_2b.c, helper.c and helper.h for reference.

   So helper.h contains all the function prototypes of calculateGrowth(), getUserInput(), and sendOutput() and their function definition are in helper.c. There is a single main file which calls all of our functions. The datatype Investment and the constant MAX_YEARS is defined in the helper.h file. We use the include guard in helper.h and include helper.h in helper.c.

   (c.) Codes are attached along with the submission. Please find main_2c.c, io.c, and calculate.c.

   I have chosen to split invest.c into three .c files, namely main.c, io.c and calculate.c. io.c contains functions getUserInput() and sendOutput() and their definitions. Function prototypes of getUserInput() and sendOutput() are defined in io.h along with the datatype Investment and the include guard. calculateGrowth() function is defined in the calculate.c file and it's prototype is defined in the calculate.h file along with a include guard. MAX_YEARS is a constant defined in io.h as well.

4. <u>Function</u>:

```
Void LCD_ClearLine(int ln) {    //    Function to clear a single line of LCD.
char c = " ";
LCD_Move(ln, 0);                //    Moves the cursor to line 'ln' and
                                //    column '0'.
for (int i = 1; i<=16;i++) {
        LCD_WriteChar(c);       //    Writes a char c at the cursor
                                //    location.
```

```
}
```

- <u>CHAPTER 5 EXERCISES:</u>

3.

(a.) The following combinations of data types and arithmetic operations results in a jump to a subroutine:

(i.) long long int with division

Example:

<u>C - statement</u>: **j3 = j1 / j2;**

<u>Assembly Commands</u>:

```
9d0030c8:   8fc40020   lw     a0,32(s8)
9d0030cc:   8fc50024   lw     a1,36(s8)
9d0030d0:   8fc60028   lw     a2,40(s8)
9d0030d4:   8fc7002c   lw     a3,44(s8)
9d0030d8:   0f4008ae   jal    9d0022b8 <__divdi3>
9d0030dc:   00000000   nop
9d0030e0:   00400013   mtlo   v0
9d0030e4:   00600011   mthi   v1
9d0030e8:   00001012   mflo   v0
9d0030ec:   00001810   mfhi   v1
9d0030f0:   afc20050   sw     v0,80(s8)
9d0030f4:   afc30054   sw     v1,84(s8)
```

(ii.) float with all operations (addition/subtraction/multiplication/division)

(iii.) double with all operations (addition/subtraction/multiplication/division)

(b.) int with addition and subtraction is the combination of the data types and arithmetic operations that results in the fewest assembly commands:

Example:

<u>C - statement</u>: **i3 = i1 + i2;**

<u>Assembly Commands</u>:

```
9d002fd4:   8fc30014   lw     v1,20(s8)
9d002fd8:   8fc20018   lw     v0,24(s8)
9d002fdc:   00621021   addu   v0,v1,v0
9d002fe0:   afc2004c   sw     v0,76(s8)
```

Char doesn't have the fewest assembly commands because char has 1 additional line 'andi' that does the bitwise 'AND' operation between v0 and 0xFF.

(c.)

	char	int	long long	float	long double
+	1.25(5)	1.0 (4)	2.75 (11)	1.25(5) [J]	2.0(8) [J]
−	1.25(5)	1.0 (4)	2.75 (11)	1.25(5) [J]	2.0(8) [J]
*	1.5 (6)	1.25 (5)	4.75(19)	1.25(5) [J]	2.0(8) [J]
/	1.75(7)	1.75 (7)	3.0 (12)[J]	1.25(5) [J]	2.0(8) [J]

(d.) We can find the math subroutines in the .map file by looking at the kseg0 Program-Memory Usage section and get the VAs of the math subroutines:

.text.dp32mul	**0x9d001e00**	0x4b8	**1208**
.text.dp32subadd	**0x9d00276c**	0x430	**1072**
.text.dp32mul	**0x9d002b9c**	0x32c	**812**
.text.fpsubadd	**0x9d003498**	0x278	**632**
.text.fp32div	**0x9d003710**	0x230	**560**
.text.fp32mul	**0x9d003940**	0x1bc	**444**

4. 'AND' and 'OR' operations both use 4 commands each. Both 'Right-shifting' and 'Left-shifting' use 3 commands each. Please find qn4.c for reference.

6.

(a.) For this, we can use the core timer after each operation and record the time. Code attached along with the submission. Please find qn6_a.c for reference.

(b.) disassembly from f2 = cos(f1) are:

```
9d00229c:   8fc40010   lw     a0,16(s8)
9d0022a0:   0f4008fb   jal    9d0023ec <.LFE7>
9d0022a4:   00000000   nop
9d0022a8:   afc20020   sw     v0,32(s8)
```

disassembly from d2 = cos(d1) are:

```
9d00231c:   8fc40018   lw     a0,24(s8)
9d002320:   8fc5001c   lw     a1,28(s8)
9d002324:   0f400984   jal    9d002610 <__truncdfsf2>
9d002328:   00000000   nop
9d00232c:   00402021   move   a0,v0
9d002330:   0f4008fb   jal    9d0023ec <.LFE7>
9d002334:   00000000   nop
9d002338:   00402021   move   a0,v0
9d00233c:   0f400a3a   jal    9d0028e8 <__extendsfdf2>
9d002340:   00000000   nop
9d002344:   afc20028   sw     v0,40(s8)
9d002348:   afc3002c   sw     v1,44(s8)
```

Advantages of using long double:

We get a much more precise value compared to float. This is particularly useful if we are using trigonometric functions like cosine whose range is small but high precision can give us continuous like display for discrete signals.

Disadvantage of using long double:

It uses twice as much space as float in memory and hence there is 3 times more assembly commands compared to float. There are multiple jumps in the assembly commands.

(c.) Directory path: ***/Applications/microchip/xc32/v2.15/pic32mx/lib***

10. So, here our global variable *glob* takes up (5000*4 = 20000) bytes of memory because it is a type int array.

    Therefore, total stack memory that is available to us is 131032 bytes (128 kB of RAM) before the global variable memory allocation.

    So, glob would take up 20k bytes from 131032 bytes.

    Hence max. size of an array of ints we can define = $\frac{131032-20000}{4}$ = **27758**.