Problem 1. (Short answers) Assume initially for each part below that:

(w0) = 0xFE9A, (w1) = 1E06, (w2) = 0x0952, and (w3) =0xFFFE, (0x0818) = 0x10C8, (0x0950) = 0x5090, (0x0A50) = 0x4020, (0x0B50) = 0x0000, (0x0E60) = 0x1111, (SR) =0x0109, and (PC) = 0x000BF8.

(a) The instructions "`cp0 w0`" and then "`bra LE,0x2E0`" are executed. Where are the operands, what are their values, and what will be the values of any registers changed after these instructions?

For the first compare with zero instruction, the operand is register `w0` which has a value of `0xFE9A = -358`. The status register is updated to reflect the result of this compare instruction, however, the resulting value of (SR) happens to be the same as its initial value (DC, N, C). Since `w0` is clearly less than or equal to zero, the next instruction `bra LE, 0x2E0` will take the branch and set the program counter to the second argument of the branch if less than or equal to instruction: `(PC) = 0x0002E0`. The following table represents the changed/affected registers:

Register	Before	After	
(SR)	0x0109	0x0109	(DC, N, C)
(PC)	0x000BF8	0x0002E0	

(b) After the instruction "mov 0x2804,w0" is executed what registers have changed and what are their new values?

While the instruction `mov 0x2804, w0` will compile without warning, on the PIC24FJ64GA002 device that is being used for this course, there is only 8KB of data memory available. Implemented data RAM addresses begin at address `0x0800` and go up to, but do not include, address `0x2800`. As such, the mov instruction above, which attempts to move the word contents of the file register located at address `0x2804`, is referencing a memory location which is unimplemented. According to the family datasheet for our device, the file registers in this unimplemented region of data memory should be read as '0'.

As such, the following registers are changed by the instruction `mov 0x2804, w0`:

Register	Before	After
(w0)	0xFE9A	0x0000
(PC)	0x000BF8	0x000BFA

(c) Suppose that the instructions "`mov [w2+w3],w0`" and then "`add w2,w0,w1`" are executed. Where are the operands, what are their values, and what will be the values of any registers changed after these instructions?

The operands of the first instruction are `[w2+w3] = [0x0950] = 0x5090` , and `w0` as the destination working register. After execution of the first instruction, the following registers have changed:

Register	Before	After
(w0)	0xFE9A	0x5090
(PC)	0x000BF8	0x000BFA

The operands of the second instruction are `w2 = 0x0952` and `w0 = 0x5090` which are to be summed, with the result being stored in the third argument `w1`. Hence, after execution of the second instruction, the following registers have changed:

Register	Before	After
(w1)	0x1E06	0x59E2
(SR)	0x0109	0x0000
(PC)	0x000BFA	0x000BFC

In conclusion, the following registers have changed as a result of the two instructions executed:

Continued on next page ⟶

Register	Before	After
(w0)	0xFE9A	0x5090
(w1)	0x1E06	0x59E2
(SR)	0x0109	0x0000
(PC)	0x000BFA	0x000BFC

(d) I have saved the SR register to data memory address 0x818, performed some operations (which may have changed SR) and now I want to restore the `SR` register to the saved value. How can I do this?

Note that the following instructions make use of `w0` as an intermediate destination for the saved SR value. To ensure that the data which is in `w0` is not lost, its value is pushed to the stack before it is changed, and restored to its initial value after it is no longer needed by popping it from the stack. Observe the following sequence of instructions which restore the SR register to the saved value:

```
push    w0
mov     0x818, w0
mov     w0, SR
pop     w0
```

(e) Suppose that "`subb w0,#0x1A,w1`" is executed. What registers will be read or written, and what will their values be afterward?

Registers `w0` and `SR` will be read, and register `w1` will be written to. Additionally, `PC` and `SR` will be modified. Note that the value of `SR` is `0x0109`, meaning that the carry flag is set. As such, the borrow flag–which is the inverse of the carry flag–is zero.

```
w0 - #0x1A - 0 = 0xFE9A - 0x1A = 0XFE80
```

As a result of the above computation, the DC N, and C flags will be set in the status register, which happens to be the same as the initial value of `SR`.

In summary,

Register	Before	After	
(w1)	0x1E06	0xFE80	
(SR)	0x0109	0x0109	(DC, N, C)
(PC)	0x000BFA	0x000BFC	

Problem 2. The subroutine DelayMS is to implement a 1 millisecond (ms) delay. The assembly listing of the subroutine DelayMS is shown below.

```
00000bea <_delayMS>:
  bea: 00 00 eb     clr.w     w0
  bec: -1 -- 2-     mov.w     #Number, w1

00000bee <.L2>:
  bee: 00 00 e8     inc.w     w0, w0
  bf0: 81 0f 50     sub.w     w0,w1, [w15]
  bf2: fd ff 3a     bra       NZ, 0xbee <.L2>
  bf4: 00 00 eb     clr.w     w0
  bf6: 00 00 06     return
```

Here Number is a 16-bit hex value to be determined. Assume an instruction frequency of 16 MHz for an instruction cycle time, Tcy, of 62.5 ns.

(a) How many bytes of program memory does the subroutine `DelayMS` occupy?

As each instruction is represented by a 24 bit op code (3 bytes), it follows that the total number of program memory bytes occupied by `DelayMS` is:

(7 instructions) $*$ (3 bytes per instruction) =21 bytes.

(b) What does `[w15]` correspond to and how is it used?

Register `w15` always points to the first available memory address on the stack. As such, `[w15]` corresponds to an indirect addressing to the location in data memory specified by `w15` which is the first available memory address on the stack. It is used as safe place to write the result of `w0 - w1` to without fear of overwriting useful information. Note that the value stored at this spot is not used by the subroutine, and so it does not matter if it gets overwritten/lost (i.e. a dummy value of sorts). The important thing is that `[w15]` is guaranteed to contain no useful information that needs to be saved before it is written to. This is done because a destination register must be specified for a subtract instruction of the form `Ws - Wb` even though we only care about the resulting status register flags and not the value of the subtraction operation. When debugging, however, the absolute value of the number written to `[w15]` indicates the number of iterations left in the loop, which can be insightful (when the loop terminates [w15] will read 0).

(c) What value should `Number` be for this routine to take as close to, but no greater than, 1 ms as possible? Include the time it takes the `rcall` instruction in the calling routine.

```
    2 cycles for rcall
    1 cycle for clr.w
    1 cycle for mov.w
    1 * Number cycle for inc.w
    1 * Number cycle for sub.w
    2 * (Number - 1) cycles for bra NZ taken
    1 cycle for bra NZ not taken
    1 cycle for final clr.w
+   3 cycles for return
 ------------------------------------
= 9 + 4*Number - 2 = (7 + 4*Number) cycles
```


Continued on next page ⟶

As each cycle is 62.5 ns, we can solve for `Number` as follows:

$$\begin{aligned} 1\ ms &= (7 + 4 \cdot Number)\ cycles\ \left(\frac{62.5\ ns}{cycle}\right) \\ \iff 7 + 4 \cdot Number &= 1\ ms\ \left(\frac{1}{62.5\ ns}\right)\left(\frac{10^6\ ns}{1\ ms}\right) \\ \iff Number &= \frac{1}{4}\left(1\ ms\ \left(\frac{1}{62.5\ ns}\right)\left(\frac{10^6\ ns}{1\ ms}\right) - 7\right) \\ \iff Number &= 3998.25 \end{aligned}$$

Since `Number` has to be an integer, however, we will round down to `Number = 3998` to avoid exceeding the 1 ms execution time target.

(d) To take exactly 1 ms how many, if any, `nop` instructions need to be inserted and where.

As determined in part (c), with a value of `Number = 3998`, the delay routine will take

$$(7 + 4 \cdot 3998)\ cycles \left(\frac{62.5\ ns}{1\ cycle}\right)\left(\frac{1\ ms}{10^6\ ns}\right) = 0.9999375\ ms$$

Hence, we need the delay routine to take $1 - 0.9999375 = 0.0000625ms = 62.5ns$ longer if we want it to delay exactly 1 ms. Luckily, this is the amount of time 1 `nop` instruction takes (1 cycle). As such, if we insert one `nop` instruction after the `bra` instruction, or before the `.L2` label, we will have hit the target delay time exactly.

Problem 3. The subroutine **power** raises an integer to the n-th power for $n > 0$. The assembly listing of the subroutine power is shown below.

```
power:
  2ce: 13 00 20     mov.w     #0x1, w3
  2d0: 01 00 e0     cp0.w     w1
  2d2: 06 00 34     bra       LE, 0x2e0
  2d4: 03 01 78     mov.w     w3, w2
L3:
  2d6: 00 9a b9     mul.ss    w3, w0, w4
  2d8: 84 01 78     mov.w     w4, w3
  2da: 02 01 e8     inc.w     w2, w2
  2dc: 82 8f 50     sub.w     w1, w2, [w15]
  2de: fb ff 3d     bra       GE, 0x2d6
L2:
  2e0: 03 00 78     mov.w     w3, w0
  2e2: 00 00 06     return
```

Assume `(w15) = 0x800` and the subroutine is called to compute 2^6 as:

```
2e4: 41 00 20       mov.w     #0x6, w1
2e6: 20 00 20       mov.w     #0x2, w0
2e8: f2 ff 07       rcall     0x2ce
```

Note that the exponent is passed in `w1` and the base in `w0`. Assume an instruction cycle time, `Tcy`, of 62.5 ns.

(a) How many bytes of program memory does the subroutine power occupy?

As each instruction is represented by a 24 bit op code (3 bytes), it follows that the total number of program memory bytes occupied by `power` is:

(11 instructions) $*$ (3 bytes per instruction) = 33 bytes.

(b) Register `w15` is the stack pointer. How much stack is used (number of bytes) for a single call to subroutine `power`? Explain.

The answer to how much stack is used is a bit nuanced. On the one hand, the stack pointer is never incremented during the execution of subroutine, which means that once the subroutine has started, the stack does not grow. However, when `rcall power` is executed, the `rcall` instruction pushes the return `PC` address to the top of the stack which has the effect of growing the stack by 4 bytes. Additionally, even though the stack is not grown during the execution of `power`, the stack pointer–which points to the first available memory address on the stack–is indirectly addressed, and used as a throw away destination address to store the result of `sub.w w1, w2, [w15]`. As such, an argument can be made that this constitutes using an additional 2 bytes of stack space. Because the value that is stored in `[w15]` does not get used, and the stack pointer is not incremented, however, it is likely more correct to say that only 4 bytes of stack memory is used for a single call to `power` instead of 6 bytes.

Continued on next page ⟶

(c) How long does it take (in microseconds) to compute 2^6 with this subroutine?

Including the time needed to initialize `w1, w2` registers with the arguments to be used in the `power` subroutine, we see the following instruction cycle breakdown:

```
   1 cycle for mov.w #0x6,w1
   1 cycle for mov.w #0x2,w0
   2 cycles for rcall
   1 cycle for mov.w
   1 cycle for cp0.w
   1 cycle for bra LE not taken
   1 cycle for mov.w
   1 * 6 cycles for mul.ss
   1 * 6 cycles for mov.w
   1 * 6 cycles for sub.w
   2 * 5 cycles for bra GE taken
   1 cycle for bra GE not taken
   1 cycle for mov.w
+  3 cycles for return
 ------------------------------------
=  41 cycles * 62.5 ns / cycle = 2562.5 ns
```

Converting to microseconds, then, yields:

$2562.5\ ns\left(\frac{1\ \mu s}{10^3\ ns}\right) = 2.5625\ \mu s$ to compute 2^6 with this subroutine.

(d) Where is the result computed by the subroutine power located when it is returned?

The result computed by `power` is stored in registers `w0, w3, w4` when it is returned. However, the intended return register appears to be `w0`, as this is where the result is moved just before returning.

(e) If we compute 2^n with this subroutine, what is the maximum value of n we can use?

The maximum value of n that can be used is $n = 15 = 0xF$. This is because $2^{16} - 1$ is the largest unsigned value that can be stored in a 16-bit register, which is obviously less than 2^{16}. It follows that $n = 15$ is the largest value that can be used since $2^{15} = 16384 \leq 2^{16} - 1$.

Problem 4. Assume a 32 MHz Focs (and corresponding 16 MHz instruction frequency). A code segment to initialize Timer 1 is

```
TMR1 = 0x0000;
PR1 = 25000-1;
T1CON = 0x8020;
```

(a) With this code what is the interval between Timer 1 events? (i.e. how long does one Timer 1 cycle take?)

Note that `TCKPS<1:0> = 0b10` which corresponds to a `Timer1` input clock prescale of $1:64$ since `T1CON = 0x8020 = 0b1000 0000 0010 0000` and bits 4 and 5 represent `TCKPS<1:0>`.

As such, the interval between Timer 1 events is:

$(25000 - 1 + 1) \cdot 62.5\ ns \cdot 64 = 100000000\ ns = 100\ ms$

(b) Change the values in (a) in `T1CON` and `PR1` to produce a 1 second interval.

To produce a 1 second interval, `Timer1` input clock prescale of $1:256$ is required. This corresponds to `TCKPS<1:0> = 0b11` which means `T1CON = 0x8030 = 0b1000 0000 0011 0000`.

Next, the value of PR1 is determined as follows:

$$\begin{aligned} 1second &= (PR1 + 1) \cdot (62.5\ ns) \cdot \frac{1\ sec}{10^9\ ns} \cdot 256 \\ \iff PR1 &= \left(\frac{10^9}{62.5}\right)\left(\frac{1}{256}\right) - 1 \\ \iff PR1 &= 62499 \end{aligned}$$

Hence, to produce a 1 second time interval, set `PR1 = 62499` and `T1CON = 0x8030`.