ET1032 Informática Industrial

15 de junio de 2015

Soluciones al examen.

NOTA: El nivel de detalle con que se responde a las preguntas en este documento no es el que se requiere ni se espera para la calificación de los ejercicios. Al redactar este documento se pretende dar información más amplia y completa con fines documentales y didácticos.

Pregunta 1. (1 punto) - La instrucción `addwf DIR, f` de la arquitectura **PIC18** suma el contenido de la dirección especificada por `DIR` con el acumulador, un registro de la CPU, dejando el resultado de la suma nuevamente en la dirección `DIR` dada. Indicad detalladamente la interacción de los distintos componentes del ordenador para la ejecución de esta instrucción.

La ejecución de la instrucción se puede describir siguiendo las fases conceptuales que describen el funcionamiento general de los ordenadores:

- En una primera fase de adquisición de la instrucción, el procesador envía el contenido del contador de programa a la memoria, y ésta le devuelve el contenido de la dirección enviada, es decir la instrucción. En el caso de la arquitectura **PIC18**, por tratarse de una arquitectura *Harvard*, la memoria afectada en este caso es la de instrucciones.
- Una vez recibida la instrucción, y de nuevo el momento viene fijado por ser esta arquitectura, con ejecución solapada de instrucciones, se incrementa el contador de programa para apuntar a la instrucción siguiente.
- La instrucción adquirida se decodifica, es decir, se obtiene de ella la información necesaria para proceder a su ejecución: el tipo de operación a realizar y los datos con los que trabaja.
- Una vez conocidas las direcciones efectivas de los operandos fuente, estos deben adquirirse para operar con ellos. Así el procesador envía la dirección `DIR` a la memoria de datos y recibe de ella el valor con el que operar. Al mismo tiempo, el procesador obtiene el contenido de su registro acumulador `w`.
- Cuando se tienen los datos se realiza en los circuitos del procesador –en la unidad aritmético lógica en este caso- la operación pertinente, una suma para la instrucción que nos ocupa. A la salida del circuito se obtiene el resultado y, en esta arquitectura, los valores de los bits del registro de estado a modificar.
- Por último se deben escribir los resultados para completar la ejecución de la instrucción. Ahora el procesador vuelve a enviar a la memoria de datos la dirección `DIR` pero esta vez indica que quiere escribir y envía también como dato el resultado de la suma. Al mismo tiempo el procesador actualiza el registro de estado con los bits correspondientes. La memoria almacena en la dirección dada el dato, y concluye la ejecución.

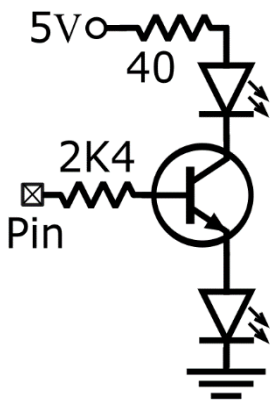
Pregunta 2. (1 punto) – Los procesadores de la familia **PIC18** implementan los registros **`PIEx`** –donde la x se sustituye por 0, 1, 2… dado que puede haber varios registros de este tipo- para almacenar los bits de habilitación de interrupciones de diferentes dispositivos. De manera análoga, los registros **`PIRx`** almacenan los *flags* que indican que el dispositivo reclama la atención del procesador, y los **`IPRx`** permiten configurar la prioridad de la interrupción como alta o baja. Indicad justificadamente cuáles de los anteriores registros son de control, cuáles de estado y cuáles de datos –no tiene por qué haber en el conjunto anterior registros de las tres clases-.

Los bits de habilitación de interrupción, almacenados en este caso en los registros **`PIEx`**, permiten que el programa pueda autorizar al dispositivo para generar interrupciones, es decir, sirven para configurar un cierto comportamiento del dispositivo. Son por tanto, de manera evidente, bits de control y los registros **`PIEx`** por ello son registros de control. Algo similar ocurre con los registros **`IPRx`**, dado que el programa decide según las especificaciones para las que ha sido diseñado, la prioridad de las interrupciones. De nuevo, los registros **`IPRx`** son claramente registros de control.

Los bits de los registros **`PIRx`** por el contrario reflejan el aviso que genera un dispositivo cuando requiere la atención del procesador por cualquier circunstancia; es decir, reflejan el estado del dispositivo, por lo que los registros **`PIRx`** son registros de estado evidentemente.

Pregunta 3. (1 punto) – Dado el montaje de la figura, con los valores en continua para el pin que aparecen en la tabla inferior y sabiendo que el valor de **Vd** para los diodos LED es de 1,8V y en el transistor se tiene **hfe** > 290, **Vcesat** < 0,2V y **Vbe** = 0,6V, indicad, cuando el transistor no está en corte, las corrientes máximas y mínimas que atravesarán cada una de las resistencias y de los diodos LED.

V_{IHmin}	3,6V	V_{OHmin}	4,2V	I_{Imax}	4 µA
V_{ILmax}	1,2V	V_{OLmax}	0,8V	I_{Omax}	23 mA



A la vista del circuito resulta sencillo deducir que el transistor conducirá únicamente cuando la tensión en el pin sea superior a la suma de **Vd** del diodo LED más **Vbe** del transistor, es decir, cuando tengamos un nivel lógico alto -1 lógico- en el pin. En este caso, si la tensión es suficiente para saturar el transistor, la corriente de colector vendrá dada por la expresión

$$Ic = \frac{5 - (2V_d + V_{cesat})}{40\ \Omega} = \frac{1{,}2\ V}{40\ \Omega} = 30\ mA$$

En el caso más desfavorable, cuando la tensión del nivel alto es **Vohmin** la corriente de base sería

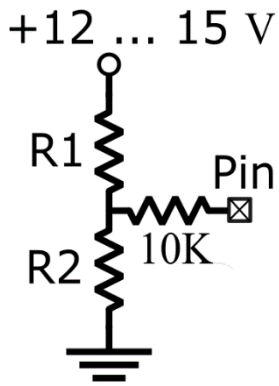
$$Ib = \frac{V_{ohmin} - (V_d + V_{be})}{2400\ \Omega} = \frac{1{,}8\ V}{2400\ \Omega} = 0{,}75\ mA$$

Que como se ve es suficiente para saturarlo, dado que **hfe** * **Ib** = 217,5 mA, muy superior a la **Ic** calculada. Con estos datos ya tenemos la Ic única que atraviesa el circuito de colector y la intensidad mínima que atraviesa la resistencia de base. Nos falta por calcular la intensidad máxima que atraviesa esta resistencia que sería

$$Ibmax = \frac{5 - (V_d + V_{be})}{2400\ \Omega} = \frac{2{,}6\ V}{2400\ \Omega} = 1{,}08\ mA$$

Nota: Estrictamente la intensidad que atraviesa el diodo inferior sería **Ic** + **Ib**, aunque es común despreciar la intensidad de base en estos cálculos.

Pregunta 4. (1 punto) – Calculad las resistencias R1 y R2 del montaje de la figura para que cualquier valor de entrada entre 12 y 15V, tal y como se ve, sea entendido como un 1 lógico en el pin. Sus características de continua aparecen en la tabla superior.



El ejercicio pide sencillamente diseñar un divisor de tensión, cuidando de que las tensiones máxima y mínima en el pin estén dentro de los rangos que son interpretados como un 1 lógico. Veamos sin embargo en primer lugar la influencia de la resistencia de protección de **10K** situada entre el punto medio del divisor y el pin. Consultando la tabla de características del pin vemos que la corriente máxima de entrada **Ii** es de 4 µA, con lo que la caída de tensión en esta resistencia será de 40 mV como mucho, aparentemente despreciable, aunque en su momento verificaremos que así sea.

Viendo las tensiones de entrada es sencillo ver que si las reducimos en un tercio mediante el divisor las tensiones en el pin estarán entre 5 y 4 V, ambas aceptables y superiores a **Vihmin** que es de 3,6 V. Así pues el divisor debe ser tal que R1 = 2 * R2 para que la tensión se reduzca en un tercio. Si elegimos las resistencias para que la corriente que las atraviesa sea de 0,1 mA en el peor caso, tenemos la expresión

$$I = \frac{V}{R} \Rightarrow 0{,}1\ mA = \frac{15\ V}{3\ R2} \Rightarrow R2 = \ 50\ K, R1 = 100\ K$$

Veamos ahora, con estos valores de resistencias, la tensión mínima en el pin a partir de la siguiente ecuación, con las resistencias expresadas en K Ω y las intensidades en mA:

$$12\ V = 100\ I + 50\ (\ I - 0{,}004) \Rightarrow I = 0{,}0787\ mA$$

$$Vpin = 50\ (\ I - 0{,}004) - 0{,}040V = 3{,}69\ V$$

Como se ve, la menor tensión posible en el pin es superior a **Vihmin**.

Pregunta 5. (1,5 puntos) – Se quiere modificar dos campos, uno de 7 y otro de 5 bits, de un mismo registro de control de 32 bits. El primer campo comienza en el bit 11 y el segundo en el 24 –recordad que los bits se numeran a partir del menos significativo, cuyo índice es 0-. Se tienen dos enteros, **`val1`** y **`val2`** que contienen en sus bits menos significativos los valores a poner en dichos campos. Realizad un fragmento de código en lenguaje C que modifique los campos dados poniendo en ellos tales valores y dejando sin modificar el resto de bits del registro.

Este ejercicio propone realizar una transformación común al tratar dispositivos de entrada/salida: modificar ciertos bits de un registro con valores dados, preservando el resto. Para ello, en primer lugar se crea una máscara de bits con **1** en aquellos que queramos preservar, y **0** en los que vamos a modificar. Al hacer una operación **`AND`** lógica preservaremos los bits adecuados y pondremos a cero los otros. Posteriormente los nuevos valores se añaden con una operación **`OR`** lógica que afecte solo a los bits adecuados.

Siguiendo los datos que se dan, el entero **`0x1F03F800`** tiene un campo de 7 unos a partir del bit 11, y otro de 5 a partir del bit 24. En realidad la máscara a usar, como se ha dicho, es la negada, cambiando unos por ceros y viceversa, lo que en leguaje C puede expresarse como **`~0x1F03F800`** o directamente con su valor **`0xE0FC07FF`**. Para añadir en esos campos los bits de los valores que se dan, es recomendable antes de desplazarlos a la posición correspondiente, 11 o 24, garantizar que el resto de los bits de los valores dados son 0, para lo que usaremos respectivamente las máscaras de 7 y 5 bits **`0x7F`** y **`0x1F`**.

Una vez comentadas las máscaras, el proceso a realizar en el código será el siguiente:

1. Realizar una operación **`AND`** entre el valor del registro y la máscara **`0xE0FC07FF`**.
2. Aplicar a la variable **`val1`** la máscara **`0x7F`** y desplazarla a la izquierda 11 posiciones; aplicar a **`val2`** la máscara **`0x1F`** y desplazarla 24 posiciones a la izquierda.
3. Realizar una operación **`OR`** entre el valor del registro y las dos variables desplazadas.

Con todas estas operaciones, una posible forma del código C sería la siguiente:

```
#define MASKPPAL    0xE0FC07FF     // Mascara principal
#define MASK7       0x7F           // Mascara para dejar 7 bits a la derecha
#define MASK5       0x1F           // Mascara para dejar 5 bits a la derecha

int registro, val1, val2;

//…

registro = lee_reg(REGXXX);         // leemos el valor del registro de control
registro = registro & MASKPPAL;     // Aplicamos la mascara para limpiar los campos

val1 = val1 & MASK7;                // Dejamos solo 7 bits
val1 = val1 << 11;                  // y desplazamos al bit 11
val2 = val2 & MASK5;                // Dejamos solo 5 bits
val2 = val2 << 24;                  // y desplazamos al bit 24

registro = registro | val1 | val2; // Y ponemos los bits en el registro

escribe_reg(REGXXX, registro);      // Escribimos el nuevo valor en el
                                    // registro de control
```

Pregunta 6. **(1,5 puntos)** – Utilizando la estructura que aparece más abajo, definid otra que represente un polígono cerrado e implementad una función que devuelva el perímetro de un polígono dado.

```
struct punto2d {double x, y; };
```

La manera más sencilla de definir un polígono es mediante el conjunto de sus vértices, que son puntos en el plano. Así pues la estructura pedida contendrá simplemente un entero que indique el número de vértices y un puntero a un vector de puntos que los contenga. Para que el polígono así representado sea correcto, cada par de puntos consecutivos en el vector debe definir un lado, estando el último conectado con el primero. Estos requisitos sin embargo no se manifiestan en la estructura, y deben ser respetados en las posibles funciones que creen polígonos, algo ajeno al objeto del ejercicio.

La función que calcule el perímetro simplemente sumará en un bucle las distancias entre cada dos puntos y finalmente añadirá la del lado que conecta el último con el primero. En cuanto a la posible detección de errores, no compete estrictamente a la función pedida, que espera un polígono bien formado. Sin embargo, se va a añadir una pequeña verificación para evitar errores de ejecución, que se darían si el número de lados es menor que tres.

A continuación aparece el código en C de la solución propuesta. La distancia entre puntos se calcula en una función auxiliar para simplificar la legibilidad y favorecer la modularidad del código.

```
struct poligono {                  // Estructura para el poligono
 int nv;                           // Numero de vertices
 struct punto2d *vertices;         // Vector de puntos
};
// Funcion para calcular las distancias. No se comenta por evidente
double distancia(struct punto2d a, struct punto2d b)
{
  double dx, dy;

  dx = a.x - b.x;
  dy = a.y - b.y;
  return sqrt(dx*dx + dy*dy);
}
// Funcion que calcula el perímetro de un polígono dado
double perimetro(struct poligono p)
{
  int i;                          // Indice para el bucle
  double peri = 0.0;              // Valor acumulado del perimetro

  if (p.nv < 3) return peri;   // Devolvemos 0 en caso de error (menos de 3 lados)
  for (i = 1; i < p.nv, i++)   // Empezamos por el vértice uno pues calculamos la
     peri += distancia(p.vertices[i], p.vertices[i-1]); // distancia al anterior
  // Al acabar el bucle sumamos la distancia entre el ultimo vértice y el primero
  // recordemos que i se ha incrementado una vez de mas
  peri += distancia(p.vertices[0], p.vertices[i-1]);

  return peri;                    // Devolvemos el valor del perimetro
}
```

Pregunta 7. (3 puntos) - Se desea realizar un programa para un microcontrolador **PIC18F4685** como el visto en clase, que tome a frecuencia fija muestras con una resolución de 10 bits de una cierta señal analógica y muestre en 8 leds funcionando como un vúmetro el valor promedio de las 16 últimas muestras tomadas.

Los vúmetros se utilizan frecuentemente en amplificadores u otros dispositivos de audio. Consisten es una barra de leds que dan una idea de la intensidad de la señal que está sonando. En los más sencillos se ilumina siempre un conjunto de leds contiguo desde el de menor intensidad. Así para intensidades sonoras bajas se iluminarán los uno o dos primeros, para intensidades medias los cuatro o cinco primeros y en el máximo, los 8 leds, siempre formando un segmento contiguo de leds encendidos.

Nota 1: Se pide desarrollar el programa completo, incluyendo la configuración de los dispositivos del microcontrolador. No es necesario diseñar la electrónica, simplemente indicar a qué pines irán conectados los diodos LED. Se puede suponer que la temporización del bucle principal la da la conversión AD, de la que no es necesario configurar los tiempos de adquisición ni de reloj –se suponen adecuados-.

Nota 2: Para almacenar los 16 últimos valores, basta con tener un vector de 16 posiciones y un índice que identifique la siguiente posición a escribir. Cuando el índice llega a 15 vuelve a pasar a 0. Así se implementa lo que se llama un *buffer* circular.

Nota 3: No es necesario calcular el promedio desde cero con cada nueva conversión. Si al valor acumulado de las 16 muestras se resta la más antigua y se suma la nueva, el proceso es mucho más rápido y sencillo y se evita recorrer el *buffer* circular con un bucle en cada iteración.

A continuación se da la información más relevante acerca de los registros del conversor A/D

Registro ADCON0							
-	-	CHS3	CHS2	CHS1	CHS0	GO	ADON
		Selección de canal					

GO: Al escribir un 1 comienza la conversión; cuando ésta ha terminado, se pone a 0 automáticamente.

ADON: El módulo de conversión se habilita al escribir un 1 en este bit, y se deshabilita al escribir un 0.

Registro ADCON2							
ADFM	-	ACQT2	ACQT1	ACQT0	ADCS2	ADCS1	ACDS0
		Tiempo de adquisición			Reloj de conversión		

ADFM: Cuando vale 1el resultado de la conversión se escribe en los 8 bits de ADRESL y los dos más bajos de ADRESH. Cuando vale 0 se utilizan los 8 bits de ADRESH y los 2 más altos de ADRESL.

Este ejercicio consiste a grandes rasgos en leer valores de un canal del conversor AD y mostrar en una barra de leds, como un vúmetro, una indicación de dicho promedio. Como el promedio es de las últimas 16 muestras tomadas, usaremos un *buffer* circular para almacenarla, como sugiere el enunciado. Ahora bien, para calcular el promedio simplemente mantendremos el valor acumulado de las 16 últimas mediante una variable que con cada nueva muestra restará la más antigua del *buffer* y sumará la nueva, al tiempo que almacena ésta en el *buffer* para restarla en su momento. Como cada muestra ocupa 10 bits, la suma de 16 como mucho 14, con lo que no se desbordará el acumulador –que será un **`word`** de 16 bits-.

Consideremos ahora el promediado y la activación de los bits, que es el único cometido de esta aplicación. En primer lugar, es fácil ver que el promediar cada 16 muestras no es casual, sino que el 16 se ha elegido por ser potencia de 2. Así, dividir entre 16 el número de muestras consiste simplemente en desplazar a la derecha 4 bits el valor acumulado. Se podría proceder así y, posteriormente, comparar el resultado entre 9 valores para encender de 0 a 8 leds en la barra. Una solución así sería algo compleja pero correcta. Se puede sin embargo añadir simplicidad si nos quedamos con un valor entre 1 y 8. En este caso, y dado que la conversión es de 10 bits, si desplazamos 7 bits a la derecha el promedio nos quedaremos con sus 3 bits más significativos, un valor entre 0 y 7 fácilmente tratable. Esta solución puede aplicarse sin necesidad de promediar antes, sin más que desplazar 11 bits (7 + 4) a la derecha el valor acumulado. Con estos valores entre 0 y 7 seleccionamos como veremos más adelante un valor de leds contiguos encendidos entre 1 y 8. Ahora bien, dado que un vúmetro se usa en un dispositivo de audio, es agradable al usuario que en caso de silencio o sonido muy poco intenso, no haya ningún led encendido. Así, nuestra solución final logrará esta circunstancia en caso de silencio comparando el valor acumulado con un cierto umbral antes de desplazarlo. Si es menor, directamente la salida será cero. En otro caso, hemos de seleccionar el conjunto de bits mediante el valor entre 0 y 7 que sale del desplazamiento. Una posibilidad sencilla es crear un vector con 8 valores, **`{0x00, 0x01, 0x03, 0x07 … 0x7F, 0xFF}`** correspondientes a los leds encendidos. Sin embargo, este valor se puede conseguir por programa desplazando el valor 2 hacia la derecha tantas veces como indique el número obtenido, y restando 1.

Solo queda por indicar que para los leds se conectarán al puerto **`B`** con el peso de los bits, y el canal analógico será el **`AN0`**. La temporización la dará el propio conversor, aunque no es necesario especificar los puertos. El código, que aparece abajo, simplemente configura el puerto **`B`** y el módulo de conversión AD y entra en un bucle infinito temporizado por la conversión. Primero lee el valor analógico, actualiza el buffer y el valor acumulado y luego genera el valor para el puerto **`B`** y lo escribe en él.

A continuación aparece el código descrito, preparado para probarse en el entorno de programación **Engine18F4685**, por eso se incluye dentro de la función **`userMain()`**.

```
#define NSAMPLES    16      // Numero de muestras
#define DISPBYTES   11      // Division para quedarnos con 3 bits
#define UMBRAL      400     // Cero de los leds

word adhist[NSAMPLES];      // Muestras antiguas guardadas

void userMain()
{
     int i, mact;
     word  acum, adval, div2led;
     byte salida;

     LATB = 0;              // Usaremos el puerto B para los LEDs
     TRISB = 0;             // que ponemos a 0 y como salida

     ADCON2bits.ADCS = 7;   // Configuramos tiempos -no es necesario
     ADCON2bits.ACQT = 3;   // en el ejercicio
     ADCON0bits.CHS = 0;    // El canal de entrada siempre AN0
     ADCON0bits.ADON = 1;   // Activamos el modulo

     for (i = 0; i < NSAMPLES; i++) // Ponemos a 0 los almacenados
            adhist[NSAMPLES] = 0;
     acum = 0;              // Ponemos a 0 el acumulado y
     mact = 0;              // el indice al vector

     ADCON0bits.GO = 1;     // Activamos la conversion

     while(1) {               // Y entramos en el bucle
            while(ADCON0bits.GO);  // Esperamos que termine la conversion
            adval = (ADRESH << 2) + (ADRESL >> 6); // leemos el valor
            ADCON0bits.GO = 1;      // y programamamos la siguiente

            // Actualizamos acumulado restando el mas antiguo y sumando el nuevo
            acum = acum + adval - adhist[mact];
            adhist[mact++] = adval; // Guardamos el nuevo valor e incrementamos
            mact = mact & (NSAMPLES - 1); // el indice modulo 15 (16 - 1)

            // Valor para el VUMETRO
            if (acum < UMBRAL)      // Si el valor es muy pequeño no encendemos nada
                   salida = 0;
            // En otro caso, nos quedamoscon un valor entre 1 y 8 sacado de los
            // 3 bits mas altos del acumulado mas 1
            // Desplazando 1 ese numero de bits y restando 1, tenemos el valor para
            // el vumetro
            else {
                   div2led = acum >> DISPBYTES;
                   salida = (2 << div2led) - 1;
            }
            // Y lo escribimos en los leds
            LATB = salida;
     }
}
```