SISTEMAS BASADOS EN MICROPROCESADOR **Primera convocatoria 2024-2025**

Apellidos:.....SOLUCIÓN.....

Nombre:.....

P1	P2	P3

Duración 4:00 horas

1.- **[3 Puntos]** Analice el siguiente código e indique qué hace. Proponga un código alternativo que haga lo mismo, pero de forma más eficiente. Señale si sigue o no el convenio de llamada de C. En caso afirmativo, muestre el prototipo. En caso negativo, enumere los parámetros de entrada/salida, ubicación y tipo. Calcule el número de ciclos de ejecución, así como el tiempo total para f_{MCLK}=1MHz. Ensamble manualmente el programa y dé el código máquina en hexadecimal.

```
;                                Emulación       CiclosCódigo máquina
;-------------------------------------------------------------------------------
algo        mov.w  r12, r13      ;_________________|____|_____________________
            clr.w  r14           ;_________________|____|_____________________
            mov.w  #16, r15      ;_________________|____|_____________________
algobuc     rrc.w  r13           ;_________________|____|_____________________
            rlc.w  r14           ;_________________|____|_____________________
            dec.w  r15           ;_________________|____|_____________________
            jnz    algobuc       ;_________________|____|_____________________
            cmp.w  r12, r14      ;_________________|____|_____________________
            jne    algono        ;_________________|____|_____________________
            mov.w  #1, r12       ;_________________|____|_____________________
            jmp    algofin       ;_________________|____|_____________________
algono      clr.w  r12           ;_________________|____|_____________________
algofin     ret                  ;_________________|____|_____________________
```

SOLUCIÓN

a) Análisis:

```
algo        mov.w    r12, r13  ;Copia el número de entrada en R13
            clr.w    r14       ;Inicializa R14 a 0
            mov.w    #16, r15  ;R15=Contador de bucle 16 iteraciones
algobuc     rrc.w    r13       ;Rota R13 a la derecha. lsb al carry
            rlc.w    r14       ;Rota R14 a izquierda. C a lsb. R14 es R13 invertido
            dec.w    r15       ;Decrementa el contador
            jnz      algobuc   ;Procesados los 16 bits? ...no; vuelve a bucle
            cmp.w    r12, r14  ;...sí. Compara número original con invertido
            jne      algono    ;Son iguales? ...no; devolver 0 por R12
            mov.w    #1, r12   ;...sí; devolver 1 por R12
            jmp      algofin   ;Salta a final
algono      clr.w    r12       ;R12 = 0
algofin     ret
```

Esta rutina determina si un número de 16 bits es un palíndromo en binario (se lee igual de izquierda a derecha que de derecha a izquierda en su representación binaria). Por ejemplo:

- 0x0000 sí es palíndromo (0000000000000000)
- 0x0FF0 sí es palíndromo (0000111111110000)
- 0x1234 no es palíndromo (0001001000110100)

El algoritmo funciona rotando el número original bit a bit hacia la derecha mientras construye simultáneamente el número invertido rotando hacia la izquierda, aprovechando el bit de carry como intermediario. Después compara ambos números para determinar si son iguales.

Hay muchas formas de optimizar el código. La alternativa propuesta se basa en invertir solo el byte bajo y compararlo con el byte alto sin invertir, lo que reduciría el tiempo del bucle a la mitad, ya que solo se hacen 8 iteraciones en vez de 16. Además, se usará el mismo registro (R12) para albergar la constante de control de bucle y el dato invertido. Se carga el dato 1 en R12 y se va desplazando a la izquierda guardando los nuevos bits por la derecha. Después de

8 iteraciones sale un 1 por la izquierda que puede ser detectado inspeccionando el carry. Obsérvese cómo se comparan el inverso del byte bajo con el alto y después se carga en R12 uno de los valores de salida.

```
palin       mov.w    r12, r13  ;Copia el número de entrada en R13
            mov.b    #1, r12   ;R12 contador de bucle e inverso de R13 (8 iterac.)
palinbuc    rrc.w    r13       ;Rota R13 a la derecha a través del carry
            rlc.b    r12       ;R14 = número original invertido
            jnc      bucle     ;Procesados los 8 bits? ...no, vuelve a bucle
            cmp.b    r13, r12  ;...sí, compara número original con invertido
            clr.w    r12       ;Por defecto devolver NO
            jne      palinfin  ;Son iguales? ...no, devolver 0 por R12
            mov.w    #1, r12   ;...sí, devolver 1 por R12
palinfin    ret
```

El número de ciclos es 1+1+8*(1+1+2)+1+1+2+1+3=42 ciclos si el número SÍ es palíndromo, un ciclo menos si NO lo es.

Convenio de llamada de C:

- Entrada por R12 Sí
- Salida por R12 Sí
- No modifica R4-R10 Sí
- Usa R13 como temporal Sí

SÍ sigue el convenio de llamada en C. El prototipo sería:

```
int palindromo(uint16_t num);
```

Tiempo de ejecución:

```
;                                Emulación       Ciclos
;-------------------------------------------------------------------------------
algo        mov.w  r12, r13      ;                  | 1 |
            clr.w  r14           ;mov.w #0, r14     | 1 |
            mov.w  #16, r15      ;                  | 2 |
algobuc     rrc.w  r13           ;                  | 1 | \
            rlc.w  r14           ;addc.w r14, r14   | 1 | | 16 iteraciones
            dec.w  r15           ;sub.w #1, r15     | 1 | |
            jnz    algobuc       ;                  | 2 | /
            cmp.w  r12, r14      ;                  | 1 |
            jne    algono        ;                  | 2 |
            mov.w  #1, r12       ;                  | 1 |
            jmp    algofin       ;                  | 2 |
algono      clr.w  r12           ;mov.w #0, r14     | 1 |
algofin     ret                  ;mov.w @sp+, pc    | 3 |
```

Se ha comentado el código para expresar las instrucciones emuladas y el número de ciclos de cada instrucción. Si R12 es palíndromo, la subrutina tarda 4+4*16+3+3+3=93 ciclos. En caso contrario, 4+4*16+3+1+3=91 ciclos. Para una frecuencia de 1MHz, el tiempo total sería 93 μs o 91 μs respectivamente.

Ensamblado:

Se ha comentado el código para expresar la codificación de las instrucciones de tipo I (las que tienen 2 operandos), las de tipo II (las que tienen 1 operando) y las de salto relativo. En éstas sólo hay que codificar la condición y el desplazamiento, que se calcula como el número de palabras desde el origen (instrucción posterior al salto), al destino, es decir, -4 para `jnz algobuc`, que en complemento a 2 es 1111111100; +2 para `jne algono` (0000000010) y +1 para `jmp algofin` (0000000001). Cada instrucción ocupa una palabra dando un total de 13 palabras o 26 bytes.

```
;                                Emulación          Código máquina
;                                                   OpCd -Rf- DdBDf -Rd-(tipo I)
;                                                   000100 OpC BDf -Rd-(tipo II)
;                                                   001 Cnd Desplazami (salto)
;-------------------------------------------------------------------------------
algo        mov.w  r12, r13      ;                | 1 |0100 1100 0000 1101 = 4C0D
```

```
            clr.w  r14          ;mov.w #0, r14     |    0100 0011 0000 1110 = 430E
            mov.w  #16, r15     ;                  |    0100 0000 0011 1111 = 403F
algobuc     rrc.w  r13          ;                  |    0001 00 00 0 000 1101 = 100D
            rlc.w  r14          ;addc.w r14, r14   |    0110 1110 0000 1110 = 6E0E
            dec.w  r15          ;sub.w #1, r15     |    1000 0011 0001 1111 = 831F
            jnz    algobuc      ;salto -4          |    001 000 11 11111100  = 23FC
            cmp.w  r12, r14     ;                  |    1001 1100 0000 1110 = 9C0E
            jne    algono       ;salto +2          |    001 000 00 00000010  = 2002
            mov.w  #1, r12      ;                  |    0100 0011 0001 1100 = 431C
            jmp    algofin      ;                  |    001 111 00 00000001  = 3C01
algono      clr.w  r12          ;mov.w #0, r14     |    0100 0011 0000 1100 = 430C
algofin     ret                 ;mov.w @sp+, pc    |    0100 0001 0011 0000 = 4130
```

CRITERIO DE CORRECCIÓN

- Análisis: 40%
- Tiempo de ejecución: 20%
- Codificación: 40%

2.- **[3 Puntos]** Realizar la función `strrchr` que sigue la convención de llamada tipo C y que atiende al siguiente prototipo:

```
char *strrchr(const char *s, char c);
```

donde `c` es el carácter a buscar en la cadena de caracteres `s`. La función devuelve un puntero a la última ocurrencia de `c` o `NULL` (0) si no se encontró.

a) Explique con palabras el algoritmo que usará para resolver el problema.
b) Escriba la subrutina en ensamblador del MSP430.

SOLUCIÓN

a) Se recorren los caracteres de la cadena s avanzando uno a uno y comparando con c. Si hay coincidencia, se guarda la dirección en un registro auxiliar que inicialmente es 0, y se sigue recorriendo la cadena hasta llegar al '\0'. Al final, el registro auxiliar contendrá la dirección de la última ocurrencia encontrada o NULL si no hubo coincidencias. Si c == '\0', se devolverá el puntero al final de la cadena.

b)

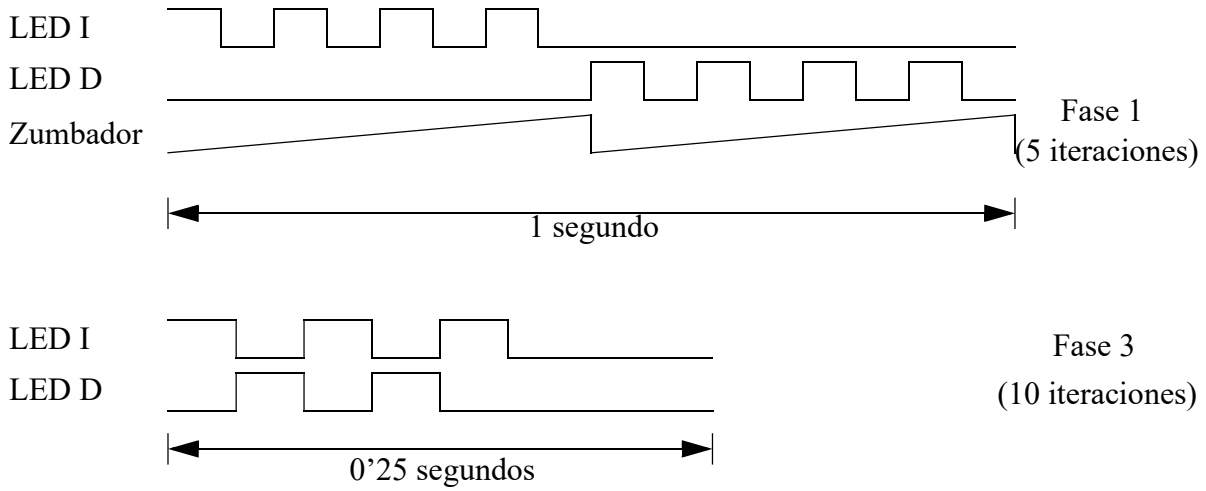
```
;-------------------------------------------------------------------------------
; char *strrchr(const char *s, char c)                                    v1.0
;
; Devuelve un puntero a la última ocurrencia de c en s o NULL si no se encontró
;-------------------------------------------------------------------------------
strrchr     clr.w     r14        ; Inicializar última coincidencia como NULL
strrchrBuc  mov.b     @r12+, r15; Leer carácter de s y avanza
            cmp.b     r15, r13   ; Es el carácter buscado?
            jne       strrchr1   ; ...no, ver fin de s
            mov.w     r12, r14   ; ...sí, guardar dirección de ocurrencia...
            dec.w     r14        ; ...compensando el incremento que se hizo en r12
strrchr1    tst.b     r15        ; Fin de s?
            jnz       strrchrBuc; ...no, seguir buscando
            mov.w     r14, r12   ; ...sí, devolver puntero a última ocurrencia o NULL
            ret
```

CRITERIO DE CORRECCIÓN

3.- **[4 Puntos]** Las señales de emergencia de un vehículo de policía disponen de 2 leds y un zumbador piezoeléctrico. Funcionan en tres fases que se repiten indefinidamente:

a) Fase 1: led izquierdo parpadea 4 veces en medio segundo con ciclo de trabajo del 50%. El led derecho apagado. En el siguiente medio segundo, los dos leds intercambian sus papeles. El zumbador hace un barrido lineal de 512Hz a 1536Hz cada medio segundo. Repetir 5 veces (ver figura).
b) Fase 2: leds igual que en la anterior, pero 3 repeticiones sólo; el zumbador hace un barrido de 512Hz a 1536Hz y vuelta a 512Hz en los 3 segundos.
c) Fase 3: Se alterna el encendido de los leds izquierdo-derecho 5 veces (I-D-I-D-I),

permaneciendo cada led encendido 1/32s; esperar 3/32s con los leds apagados y empezar nueva iteración, invirtiendo el orden de los leds (es decir, en la siguiente sería D-I-D-I-D); 10 iteraciones para dar 2'5 segundos en total con el zumbador en silencio.



Realice un programa en ensamblador del MSP430 que realice la parte óptica de este sistema de señalización por interrupciones y con el mínimo consumo. Datos:

- El led izquierdo se encuentra conectado a P1.1, cuya función primaria es TA0.1.
- El led derecho se encuentra conectado a P1.2, cuya función primaria es TA0.2.
- El zumbador se encuentra conectado a P1.3, cuya función primaria es TA0.3.
- Considere el perro guardián desactivado, los puertos desbloqueados, ACLK funcionando a 32768Hz y la pila inicializada.

SOLUCIÓN

Inicialmente se configurarán P1.1 a P1.3 como salidas con función primaria para ser controlados por TA0. De las figuras se deduce que la lógica de los leds es positiva.

De tener que implementar la parte de sonido, se usarían las capacidades PWM del TA0, empleando el CCR0 para configurar la frecuencia y resolución de la señal de audio. Dado que no se realizará en este problema, queda libre para ayudar en la generación de las señales de los leds, si fuera necesario.

Aunque en el problema se habla de 3 fases, desde el punto de vista de los leds sólo son dos: 10 repeticiones de la fase 1 y 10 de la 3.

El control de los tiempos se hará usando los canales 1 y 2 del TA0 configurados en modo comparación y con el valor adecuado del semiperiodo que toque en cada caso. Los cambios en los leds se harán automáticamente por el TA0. Si el led debe conmutar, se configurará el modo OUTMOD_4 (toggle). Si debe permanecer apagado, el OUTMOD_5 (reset). Aunque se gestionarán los CCR de ambos canales para generar las señales automáticamente, sólo se activará la IRQ del CCR1, ya que la del CCR2 se producirá simultáneamente. En la ISR se gestionarán ambos canales.

La fase 1 consiste en una señal de 8Hz que alimentará al led izquierdo durante medio segundo y al led derecho durante otro medio. Se repite 10 veces. Para generar las señales, se configurará el timer con una frecuencia de 16Hz, para generar cada semiperiodo. La variable ContFase1 se usará para saber en qué semiperiodo nos encontramos. Los 3 bits menos significativos indicarán el semiciclo (0 a 7); el bit 3 si parpadea el led izquierdo (0) o el derecho (1); y los bits 4 a 7 indicarán la iteración (entre 0 y 9).

La fase 3 consiste en una señal de 16Hz que alimentará a los leds izquierdo y derecho en contrafase durante 5 semiperiodos y otros 3 apagados. Se repite 10 veces. En las iteraciones pares se comenzará ilumniando el led izquierdo y en las pares el derecho. En el código se usan los registros R5 y R6 para guardar las direcciones de los registros de control de CCR1 y CCR2 según convenga para no repetir el código según el semiciclo en el que nos encontremos. La variable ContFase3 se usará para seguir la pista del semiciclo en el que nos encontramos. Se inicializará a -1 para indicar que se está en la fase 1. La codificación es parecida a la de

ContFase1: bits 2 a 0 para recoger el número de semiciclo, y bits 6 a 3 para almacernar el número de iteración (entre 0 y 9).

La frecuencias de trabajo van a ser 16Hz y 32Hz, lo que habilita a usar la máxima división de frecuencia del TA0 para reducir al máximo el consumo y sin incurrir en errores de frecuencia por los redondeos, ya que todos los números son potencia de 2 (`NFASE1 = (32768/64)/(2*16)-1`).

En main recae la tarea de configuración e inicialización: puertos como salida en función primaria. El led izquierdo encendido y en modo conmutación y el derecho apagado y en modo reset. El TA0 se arrancará con el divisor de 64 en modo continuo. Sólo se habilitarán las interrupciones del CCR1. Finalmente se entra en modo LPM3 ya que el resto del trabajo se hará por el TA0 (cambios en los leds) e interrupciones.

```
;Constantes de configuración
LEDI        .equ   BIT1
LEDD        .equ   BIT2
ZUMB        .equ   BIT3
ALARMA      .equ   LEDI|LEDD|ZUMB

FACLK       .equ   32768         ;Frecuencia de ACLK. En Hz
FFASE1      .equ   8             ;Frecuencia de parpadeo en fase 1 (Hz)
FFASE3      .equ   16            ;Frecuencia de parpadeo en fase 3 (Hz)

DIVPRI      .equ   8             ;Divisor primario del TA0
DIVSEC      .equ   8             ;Divisor secundario del TA0

;Constantes calculadas
FTA         .equ   FACLK/(DIVPRI*DIVSEC)         ;Frecuencia del TA0. En Hz
NFASE1      .equ   FTA/(2*FFASE1)-1              ;Valor del CCR para fase 1
NFASE3      .equ   FTA/(2*FFASE3)-1              ;Valor del CCR para fase 3

;Variables
            .bss   ContFase1,1   ;Contador de la fase 1
            .bss   ContFase3,1   ;Contador de la fase 3

;-------------------------------------------------------------------------------
;main                                                                      v1.0
;-------------------------------------------------------------------------------
main        ;Inicializar variables
            clr.b  &ContFase1    ;Contador de fase 1. Empezamos en 0
            mov.b  #-1, &ContFase3;Contador de fase 3. Igual a -1, indica fase 1 activa

            ;P1.1 a P1.3 salidas con función primaria
            ;P1.1 a P1.3 salidas función 1 (TA0.1 a TA0.3)
            bis.b  #ALARMA, &P1DIR;Salidas
            bis.b  #ALARMA, &P1SEL0;Función 1

            ;LED I encendido y D apagado
            mov.w  #OUTMOD_0|OUT, &TA0CCTL1
            mov.w  #OUTMOD_0    , &TA0CCTL2
            ;LED I parpadea (conmuta) a 8Hz y LED D espera medio segundo apagado
            ;Temporización guiada por CCR1
            mov.w  #OUTMOD_4|CCIE, &TA0CCTL1
            mov.w  #OUTMOD_5     , &TA0CCTL2
            mov.w  #NFASE1, &TA0CCR1
            mov.w  #NFASE1, &TA0CCR2

            ;TA0 con ACLK/DIVTA modo CONT
            mov.w  #DIVSEC-1, &TA0EX0;Divisor extra
            mov.w  #TASSEL__ACLK|ID__8|MC__CONTINUOUS|TACLR, &TA0CTL
            bis.w  #LPM3|GIE, sr ;Ea, a dormir

;-------------------------------------------------------------------------------
; TA01ISR                                                                  v1.0
;
; ISR del CCR1
;-------------------------------------------------------------------------------
```

```
TA01ISR     bic.w  #CCIFG, &TA0CCTL1;Borrar IRQ
            push.w r4            ;Salvar reg en ISR
            cmp.b  #-1, &ContFase3;Estamos en fase 3?
            jne    TA01ISRF3     ;...sí, procesar fase 3
            ;Fase 1
TA01ISRF1   add.w  #NFASE1, &TA0CCR1;...no. Fase 1. Preparar siguiente semiciclo
            add.w  #NFASE1, &TA0CCR2;Preparar siguiente semiciclo
            inc.b  &ContFase1    ;Contar un semiciclo más
            mov.b  &ContFase1, r4;R4: Contador de la fase 1
            and.b  #0xF0, r4     ;Dejar sólo el número de iteración (bits 7-4)
            cmp.b  #0xA0, r4     ;Es la iteración 10?
            jhs    TA01CmbF3     ;...sí, cambia a fase 3
            mov.b  &ContFase1, r4;R4: Contador de la fase 1
            and.b  #7, r4        ;Dejar sólo el semiciclo (bits 2-0)
            cmp.b  #7, r4        ;Es la iteración 7?
            jne    TA01ISRFin    ;...no, hemos terminado
            xor.w  #OUTMOD0, &TA0CCTL1;...sí. Conmuta parpadeando-apagado led I
            xor.w  #OUTMOD0, &TA0CCTL2;Conmuta parpadeando-apagado led D
            jmp    TA01ISRFin    ;Salir
            ;Fase 3
TA01ISRF3   push.w r5            ;Salvar R5. Guardará la dirección del led de inicio
            push.w r6            ;Salvar R6. Guardará dirección del led en contrafase
            add.w  #NFASE3, &TA0CCR1;Preparar siguiente semiciclo
            add.w  #NFASE3, &TA0CCR2;Preparar siguiente semiciclo
            inc.b  &ContFase3    ;Contar un semiciclo más
            mov.b  &ContFase3, r4;R4: Contador de la fase 3
            and.b  #0xF8, r4     ;Dejar sólo el número de iteración (bits 7-3)
            cmp.b  #0x50, r4     ;Es la iteración 10?
            jhs    TA01CmbF1     ;...sí, cambia a fase 1
            bit.b  #BIT3, r4     ;Es una iteración par?
            jne    TA01Impar     ;...no, empieza el led derecho
TA01Par     mov.w  #TA0CCTL1, r5 ;R5: dirección del led izquierdo
            mov.w  #TA0CCTL2, r6 ;R6: dirección del led derecho
            jmp    TA01Iter      ;Seguir
TA01Impar   mov.w  #TA0CCTL2, r5 ;R5: dirección del led derecho
            mov.w  #TA0CCTL1, r6 ;R6: dirección del led izquierdo
TA01Iter    mov.b  &ContFase3, r4;R4: Contador de la fase 3
            and.b  #7, r4        ;Dejar sólo el semiciclo (bits 2-0)
            rla.w  r4            ;Multiplicar por 2 para tabla de saltos
            add.w  r4, pc        ;Saltar
            jmp    TA01CmbD      ;0, enciende led de contrafase
            jmp    TA01ISRFin    ;1, salir
            jmp    TA01ISRFin    ;2, salir
            jmp    TA01ISRFin    ;3, salir
            jmp    TA01CmbD      ;4, apaga led de contrafase
            jmp    TA01CmbI      ;5, apaga led de inicio
            jmp    TA01ISRFin    ;6, salir
            ;jmp   TA01CmbI      ;7, enciende led de inicio
TA01CmbI    ;Cambia led de inicio (I/D, indicado por R5) entre conmutación/apagado
            xor.w  #OUTMOD0, 0(r5);Cambia led de inicio conmutando-apagado
            jmp    TA01ISRFin2   ;Salir
TA01CmbD    ;Cambia led de contrafase (D/I, indicado por R6) entre conmutación/apagado
            xor.w  #OUTMOD0, 0(r6);Cambia led de contrafase conmutando-apagado
            jmp    TA01ISRFin2   ;Salir
TA01CmbF1   ;Cambiar a fase 1
            clr.b  &ContFase1    ;Empezar fase 1 desde el principio
            mov.b  #-1, &ContFase3;Fase 3 apagada
            jmp    TA01ISRFin2   ;Salir
TA01CmbF3   ;Cambiar a fase 3
            clr.b  &ContFase3    ;Empezar fase 3 desde el principio
            jmp    TA01ISRFin    ;Salir
TA01ISRFin2;Punto de salida si estamos en la fase 3
            pop.w  r6            ;Recuperar registros
            pop.w  r5
TA01ISRFin ;Punto de salida si estamos en la fase 1
            pop.w  r4
            reti
```

```
.intvecRESET_VECTOR, main
.intvecTIMER0_A1_VECTOR, TA01ISR
```

CRITERIO DE CORRECCIÓN

Definiciones e inicialización (main): 20%

ISR: 80%