Solución Examen de Arquitectura de Computadoras

27 de marzo de 2021

Instrucciones:

- **Indique su nombre, apellido y número de cédula en todas las hojas que entregue.**
- **Las hojas deben estar numeradas y en la primer hoja debe escribirse el total.**
- **Deben escanearse todas las hojas en orden para generar el pdf a entregar.**
-

Problema 1

Un sistema cooperativo requiere enviar información sensible a través de una red de datos poco segura. Por este motivo, los datos enviados son encriptados utilizando un mecanismo muy simple denominado ***tres-ofuscado***.
El algoritmo para transmitir ***tres-ofuscado*** asegura que cuando se reciben tres bits consecutivos con el mismo valor (bit) el emisor envió ese valor (bit) como un bit válido. Entre valores válidos se pueden colocar bits aleatorios que no forman bits válidos.
Es decir que cuando desea enviar un 0 el emisor envía tres bits en 0 (000) y cuando desea enviar un 1 debe enviar tres bits en 1 (111). Cualquier otra secuencia no representa un bit válido.
Se desea implementar un circuito receptor que sea capaz de decodificar el ***tres-ofuscado***.
Dicho circuito debe tener una entrada donde se reciben los bits de la comunicación (los que envía el emisor usando el algoritmo descripto), en forma serial sincronizados con cada flanco del reloj, y debe tener dos salidas, la primera indica si se recibió un bit válido y la segunda el valor del bit válido recibido.

Se pide:
Diseñar y dibujar el circuito receptor descripto, utilizando la metodología del curso. Se dispone de flip-flops tipo J-K y compuertas lógicas.

Problema 2

Considere una matriz de píxeles blancos y negros :

```
#define ALTO
#define ANCHO
char pixeles[ALTO][ANCHO];
```

Cada posición de la matriz se representa con 8 bits, pero se utilizan únicamente lo valores 1 (matriz ocupada) y 0 (matriz libre). Considere el siguiente algoritmo para rellenar una figura cerrada:

```
void ocupar(int i, intj){
   if (i >= 0 && j >= 0 && i < ALTO && j < ANCHO){
   if (pixeles[i][j] == 1){
          pixeles[i][j] = 0;
          ocupar(i + 1, j);
          ocupar(i, j + 1);
          ocupar(i – 1, j);
          ocupar(i, j – 1);
     }
   }
}
```

Se pide:
a) Compilar la función `ocupar(i ,j)` en assembler 8086. Los parámetros se pasan por stack, y la matriz se encuentra ubicada en memoria a partir de la dirección 0 del segmento extra.

Nota: se asume que las constantes ALTO y ANCHO no son mayores a 32.

b) Considere una matriz **pixeles** de 3 * 3 con todos sus casilleros en 1. Calcule el máximo consumo de stack si se realiza la siguiente llamada: `ocupar(0, 0)`.

Problema 3

Se desea implementar el controlador de un sistema anti-bloqueo de frenos (ABS por su sigla en inglés) para las ruedas delanteras de un auto.

Los sistemas ABS funcionan, simplificada y esquemáticamente, de la siguiente manera:

1. al apretar el pedal de freno se debe aplicar, inicialmente, la misma presión PRESMAX a los frenos de ambas ruedas.
2. se debe medir la velocidad de giro de ambas ruedas para detectar si la velocidad de giro de una de ellas es menor que la de la otra, en cuyo caso se debe disminuir la presión sobre el freno de la rueda que está girando más lentamente. La primera disminución consiste en restarle 1 a PRESMAX (es decir se debe aplicar PRESMAX – 1).
3. mientras la velocidad de la rueda que se detectó como mas lenta continúe en esa condición, se irá aplicando una disminución progresiva de la presión en el freno de dicha rueda. La segunda disminución resta 2 a PRESMAX, la tercera 4, y así sucesivamente restando progresivamente potencias de dos crecientes al valor original (es decir se aplica PRESMAX – 2^i) en cada evaluación, mientras la presión aplicada sea superior a 0 y hasta que la velocidad de la rueda más lenta iguale o supere la de la otra.
4. la evaluación de la diferencia de velocidades y la aplicación de la nueva presión debe realizarse cada 100 ms.
5. si luego de la disminución de la presión, en algún momento la velocidad de la rueda originalmente más lenta supera a la otra, se debe volver a aplicar la presión PRESMAX a dicha rueda y el algoritmo debe repetirse desde el paso 2.
6. cuando el pedal de freno deja de apretarse, debe dejarse de aplicar presión sobre los frenos en forma inmediata.

Se dispone de los siguientes elementos:

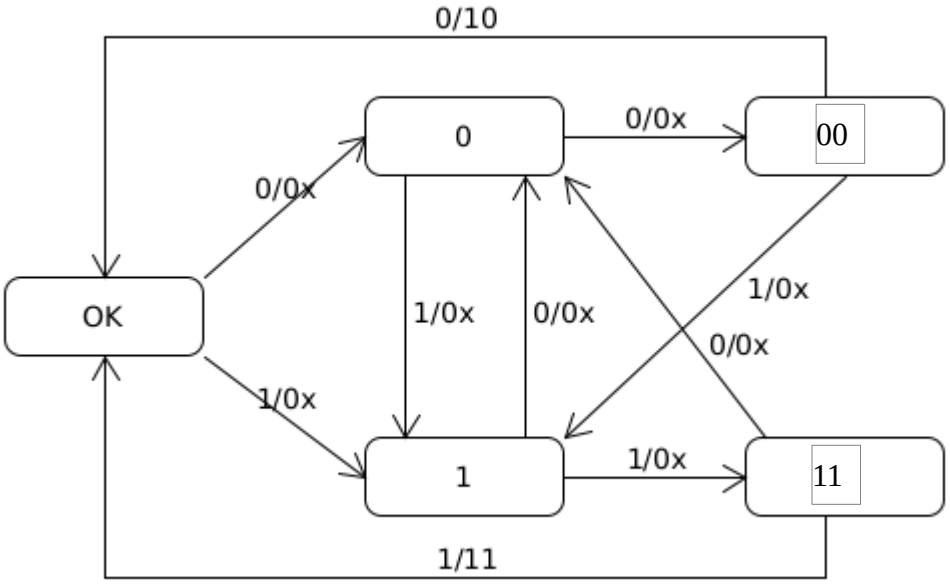
- sensores de giro en las ruedas que provocan una interrupción con cada giro de cada rueda, invocando a las rutinas **giro_der()** y **giro_izq()**, respectivamente.
- un timer que interrumpe con una frecuencia de 10 Khz invocando a la rutina **timer()**.
- un puerto de E/S, de tamaño byte y solo lectura, en la dirección **PEDAL**, en cuyo bit más significativo puede leerse el estado del pedal de freno (0 = no apretado, 1 = apretado).
- dos puertos de E/S, de tamaño palabra (2 bytes) y solo escritura, que aplican sobre los respectivos frenos el valor de presión que se escriba en ellos, en las direcciones **FRENO_DER** y **FRENO_IZQ**.

Se pide:
Escribir en un lenguaje de alto nivel (preferentemente C) todas las rutinas necesarias para implementar el controlador de frenos ABS descripto.

Solución Problema 1

Entradas: bit entrada

Salidas: valido / valor



Estado	Entrada	Estado siguiente	Salida		Estado	Entrada	Estado siguiente	Salida
OK	0	0	0x		000	0	010	0x
OK	1	1	0x		000	1	011	0x
0	0	00	0x		010	0	100	0x
0	1	1	0x		010	1	011	0x
1	0	0	0x		011	0	010	0x
1	1	11	0x		011	1	111	0x
00	0	OK	10		100	0	000	10
00	1	1	0x		100	1	011	0x
11	0	0	0x		111	0	010	0x
11	1	OK	11		111	1	000	11

Estado	Entrada	Estado siguiente	J2K2	J1K1	J0K0	Salida
000	0	010	0x	1x	0x	0x
000	1	011	0x	1x	1x	0x
010	0	100	1x	x1	0x	0x
010	1	011	0x	x0	1x	0x
011	0	010	0x	x0	x1	0x
011	1	111	1x	x0	x0	0x
100	0	000	x1	0x	0x	10
100	1	011	x1	1x	1x	0x
111	0	010	x1	x0	x1	0x
111	1	000	x1	x1	x1	11

Mapa para J2

q2q1\ q0e	00	01	11	10
00	0	0	X	X
01	1	0	1	0
11	X	X	X	X
10	X	X	X	X

J2=q1q0’e’+q0e

Mapa para K2

q2q1\ q0e	00	01	11	10
00	X	X	X	X
01	X	X	X	X
11	X	X	1	1
10	1	1	X	X

K2 = 1

Mapa para J1

q2q1\ q0e	00	01	11	10
00	1	1	X	X
01	X	X	X	X
11	X	X	X	X
10	0	1	X	X

J1=q2'+e

Mapa para K1

q2q1\ q0e	00	01	11	10
00	X	X	X	X
01	1	0	0	0
11	X	X	1	0
10	X	X	X	X

K1=q0’e’+q2e

Mapa para J0

q2q1\ q0e	00	01	11	10
00	0	1	X	X
01	0	1	X	X
11	X	X	X	X
10	0	1	X	X

J0 = e

Mapa para K0

q2q1\ q0e	00	01	11	10
00	X	X	X	X
01	X	X	0	1
11	X	X	1	1
10	X	X	X	X

K0=q2+e’

Mapa para Valido

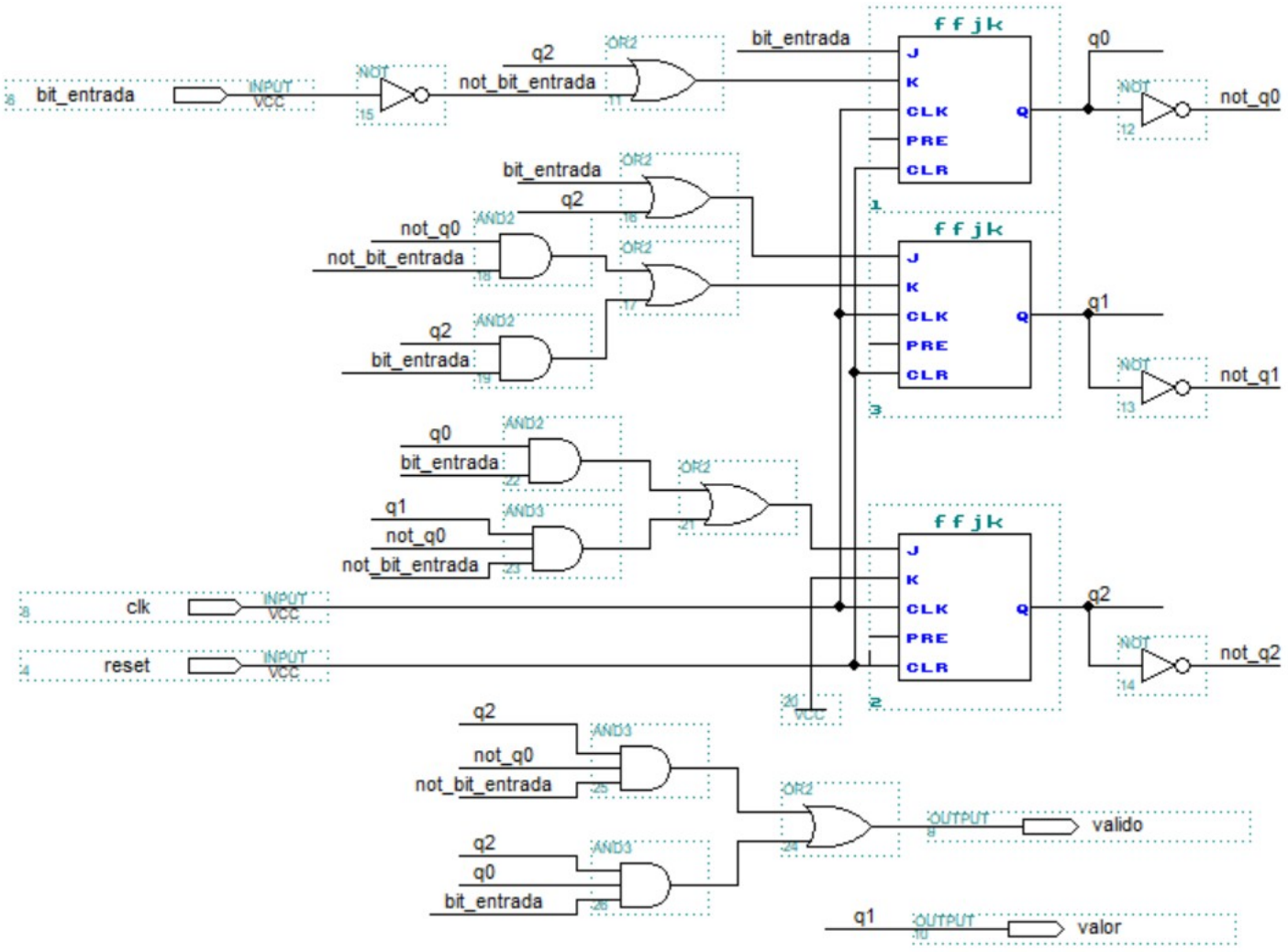
q2q1\ q0e	00	01	11	10
00	0	0	X	X
01	0	0	0	0
11	X	X	1	0
10	1	0	X	X

Valido=q2q0’e’+q2q0e

Mapa para Valor

q2q1\ q0e	00	01	11	10
00	X	X	X	X
01	X	X	X	X
11	X	X	1	X
10	0	X	X	X

Valor=q1


bit_entrada
INPUT
VCC
NOT
q2
not_bit_entrada
OR2
bit_entrada
f f j k
J
K
CLK
Q
PRE
CLR
q0
not_q0
not_q1
not_q2
q1
AND2
AND3
clk
reset
valido
valor
OUTPUT

Solución Problema 2

a)

```
ocupar proc
       push bp
       mov bp, sp
       push si
       push di
       push ax
       push bx

       mov si, [bp + 4]  ; obtener parámetos del stack
       mov di, [bp + 6]

       cmp si, 0         ; chequear condiciones de caso base
       jl fin
       cmp di, 0
       jl fin
       cmp si, ALTO
       jge fin
       cmp di, ANCHO
       jge fin

       mov ax, ANCHO     ; convertir índice en puntero
       mul si            ; i * ancho + j
       add ax, di
       mov bx, ax
       cmp byte_ptr es:[bx], 1
       jne fin
       mov byte_ptr es:[bx], 0

       inc si            ; llamadas recursivas
       push si
       push di
       call ocupar       ; ocupar(i+1, j)
       dec si            ; si contiene i + 1, se debe decrementar
       inc di
       push si
       push di
       call ocupar       ; ocupar(i, j+1)
       dec si
       dec di
       push si
       push di
       call ocupar       ; ocupar(i-1, j)
       inc si
       dec di
       push si
       push di
```

```
        call ocupar         ; ocupar(i, j - 1)
    fin:
        mov bx, [bp+2]
        mov [bp+6], bx      ; acomodar dirección de retorno
        pop bx
        pop ax
        pop di
        pop si
        pop bp
        add sp, 4           ; remover parámetro del stack
        ret
```

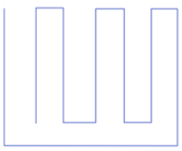
b)

Solución genérica:

Siguiendo el algoritmo, se puede comprobar que las direcciones tienen prioridades entre ellas: primero abajo, luego derecha, arriba, izquierda.

Para una matriz de N x N totalmente libre (la celda 0,0 es la ubicada a la izquierda arriba) la ejecución primero recorrerá de forma recursiva toda la primer columna hacia abajo, hasta llegar al paso base dado por i > ALTO. Luego, se recorrerá la fila inferior hacia la derecha, hasta llegar a la casilla inferior derecha y salir por el paso base j > ANCHO. En esa llamada recursiva, las llamadas a `ocupar(i + 1, j)` y a `ocupar(i, j + 1)` llegarán a un paso base, por lo tanto la siguiente recursión será hacia arriba. Se recorrerá la columna final hacia arriba hasta llegar a la casilla superior derecha, donde por fin se irá hacia la izquierda. En esta casilla (primer fila, penúltima columna), las casillas izquierda e inferior están libres, la casilla superior no es parte de la matriz y la de la derecha está ocupada. Por lo tanto, como la dirección hacia abajo es prioritaria, se toma esa dirección.

Si se continúa razonando del mismo modo, se puede ver que la matriz se recorre de la siguiente manera:



Se cubre toda la matriz de forma recursiva y por lo tanto hay tantas llamadas recursivas como casillas + un paso base.

Cada llamada, independientemente de si se trata del paso base o del paso recursivo, ocupa 16 bytes en el stack. 2 por IP, 4 por parámetros y 10 por contexto. De esta forma, como la función realiza ALTO * ANCHO llamadas recursivas y 1 paso base, se utilizarán (ALTO * ANCHO + 1) * 16 bytes.

Solución específica 3 x 3:

Las llamadas anidadas con las que se recorre la matriz son las siguientes:

ocupar(0, 0) → casilla superior izquierda

ocupar(1, 0) → casilla fila 1 columna 0
ocupar(2, 0)
ocupar(2, 1) → notar que en este caso se lllamó primero a ocupar(3, 0) pero se retornó por ser caso base
ocupar(2, 2)
ocupar(1, 2) → notar que se invocó a ocupar(2, 3) y ocupar(3,2) pero ambos son paso base
ocupar(0, 2)
ocupar(0, 1)
ocupar(1, 1)
paso base: ocupar(0, 1), que está ocupada

a partir de aquí se retorna liberando el stack.

Por lo tanto se recorren todas las casillas de forma recursiva y se ejecuta un paso base adicional. En el stack anidan 9 llamadas recursivas y 1 llamada a paso base. Cada llamada, independientemente de si se trata del paso base o del paso recursivo, ocupa 16 bytes en el stack. 2 por IP, 4 por parámetros y 10 por contexto. De esta forma, como la función realiza 9 llamadas recursivas y 1 paso base, por lo que se utilizarán 10 * 16 = 160 bytes.

Solución Problema 3:

```
#define 100_MS 1000;
#define ESPERA 0;
#define PRESIONES_IGUALES 1;
#define PRESIONES_DIFERENTES 2;
#define DER 0
#define IZQ 1

#define PRESMAX ...

unsigned int ticsIzq;
unsigned int ticsDer;
char estado;
char mult;
int presion[2]; // presion[0] = der, presion[1] == izq

void main(){
   // instalar interrupciones
   estado = ESPERA;
   ticsIzq = 0;
   ticsDer = 0;
   mult = 0;
   presion[DER] = PRESMAX;
   presion[IZQ] = PRESMAX;
   enable();
   while (true);
}

void interrupt timer(){
   ticsIzq++;
   ticsDer++;
   char pedal = IN(PEDAL);
   if ((pedal & 0x80) == 0) {                    // si el pedal no está presionado
      OUT(FRENO_DER, 0);
      OUT(FRENO_IZQ, 0);
      estado = ESPERA;
      tics = 0;
   } else {  // si el pedal está presionado
      tics++;
      // si estado = ESPREA (primera vez que entra) se reacciona inmediatamente
      if (tics == 100_MS || estado == ESPERA) {
         switch (estado) {
            case ESPERA:
               presion[DER] = PRESMAX;
               presion[IZQ] = PRESMAX;
               ESTADO =  PRESIONES_IGUALES;
               break;
            case PRESIONES_IGUALES:
               presion[DER] = PRESMAX;
               presion[IZQ] = PRESMAX;
               if (ticsGiroIzq < ticsGiroDer) {     // rueda der más lenta
                  estado = PRESIONES_DIFERENTES;
                  mult = 1;
                  ruedaABS = DER;
                  presion[DER] = presion[DER] - 1;
               } else if (ticsGiroDer < ticsGiroIzq) { // rueda izq más lenta
                  estado = PRESIONES_DIFERENTES;
                  mult = 1;
                  ruedaABS = IZQ;
```

```
                presion[IZQ] = presion[IZQ] - 1;
             }
             break;
          case PRESIONES_DIFERENTES:
             // si la rueda a la que se aplica el ABS sigue siendo la más lenta
             if ((ruedaABS == DER &&  ticsGiroIzq < ticsGiroDer) ||
                 (ruedaABS == IZQ &&  ticsGiroDer < ticsGiroIzq)) {
                mult++;
                presion[ruedaABS] = PRESMAX - (1 << mult);
                if (presion[ruedaABS] < 0) presion[ruedaABS] = 0;
             } else {
                presion[ruedaABS] = PRESMAX;
                estado = PRESIONES_IGUALES;
             }
             break;
       } // switch

       OUT(FRENO_DER, presion[DER]);
       OUT(FRENO_IZQ, presion[IZQ]);

    } // if tics == 100_MS
  }
}

void interrupt giro_der(){
  ticsGiroDer = ticsDer;
  ticsDer = 0;
}

void interrupt giro_izq(){
  ticsGiroIzq = ticsIzq;
  ticsIzq = 0;
}
```