Examen de Arquitectura de Computadoras

21 de diciembre del 2017

Instrucciones:

- Indique su nombre, apellido y número de cédula en todas las hojas que entregue.
- Escriba las hojas de un solo lado.
- Las hojas deben estar numeradas y en la primer hoja debe escribirse el total.
- Empiece cada ejercicio en una hoja nueva.
- No puede utilizar material ni calculadora.
- **Apague su celular.**
- La duración del examen es de tres horas. **En dicho tiempo debe también completar sus datos**. Solo se contestarán dudas de letra. No se aceptan preguntas en los últimos 30 minutos del examen.
- Para aprobar debe contestar correctamente un ejercicio entero y dos preguntas.

Pregunta 1

Describa el mecanismo de atención a interrupciones en un computador 8086 que dispone de controlador de interrupciones

Respuesta:

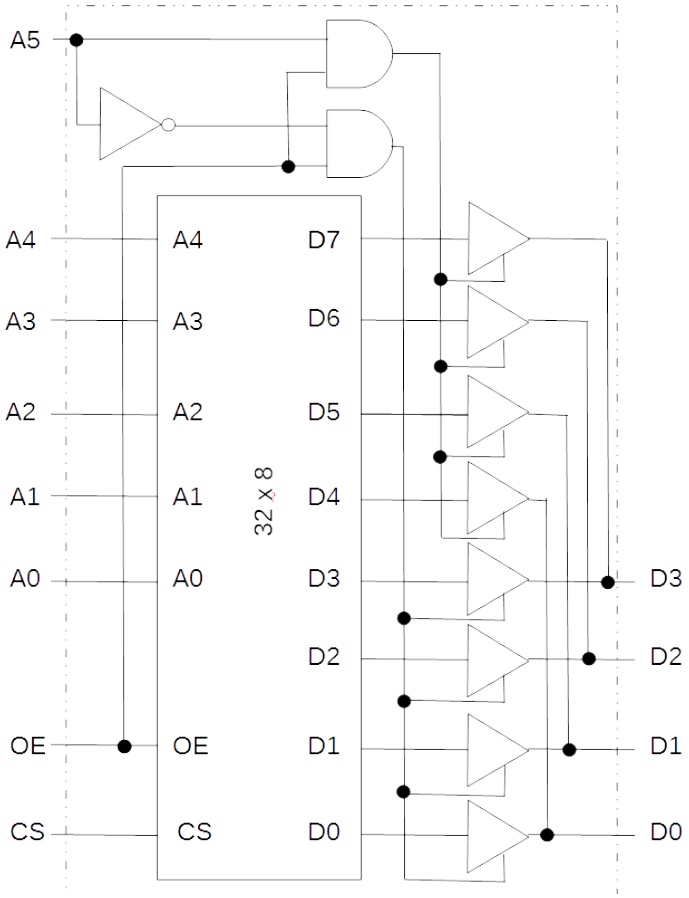
Un microprocesador 8086, al recibir un pedido de atención a la interrupción a través de la señal INT, realiza los siguientes pasos (en caso que las interrupciones estén habilitadas):

- Termina de ejecutar la instrucción actual (como consecuencia de que verifica la existencia del pedido al final del ciclo de una instrucción y antes de comenzar el siguiente).
- Obtiene el identificador de la línea de pedido de interrupción (IRQ) que está solicitando interrupción, el que es suministrado por el controlador de interrupciones en el bus de datos, luego de activar la señal INTA.
- Preserva la dirección de la próxima instrucción a ejecutarse y el estado del procesador, salvando en el stack el valor actual de los registros IP, CS, y FLAGS.
- Enmascara las interrupciones (las deshabilita)
- Accede al vector de interrupciones utilizando el identificador como indice (realiza accesos a memoria, en las direcciones id_interrupt * 4 y id_interrupt * 4 + 2) para obtener el desplazamiento y el segmento de la dirección de la primera instrucción de la rutina de servicio de la interrupción.
- Realiza un jump far a la dirección obtenida en el vector de interrupciones.

Pregunta 2

Construya una ROM (incluyendo las entradas CS y OE) de 64x4 a partir de una ROM de 32x8.

Respuesta:



Pregunta 3

Dada una arquitectura con direcciones de 20 bits, y una memoria cache de 4kb con líneas de 256 bytes. Indique como se intrpreta una dirección de memoria si se usa una función de correspondencia asociativa de 4 vias.

Respuesta:

Las direcciones emitidas por el procesador se interpretan por la cache como un registro con tres campos: tag, conjunto y byte.

Dado que el tamaño de la linea es de 256 bytes (2^8), se necesitan 8 bits para indicar el byte.

Dado que el tamaño de la cache es 4KB y el tamaño de la linea es 256B, se tienen 16 lineas. (4KB / 256B = 2^12 / 2^8 = 2^4 =16).

Como la cache es de 4 vías y se tienen 16 lineas, tendrá 4 conjuntos (16/4=4=2^2), entonces serán necesarios 2 bits para indicar el conjunto.

El campo tag utiliza el resto de los bits de la dirección (20-8-2=10)

TAG		CONJUNTO		BYTE	
19	10	9	8	7	0

Pregunta 4

Interpretando los hexadecimales como números representados en notación complemento a 2 de 16 bits, la siguiente secuencia de operaciones produce overflow intermedio. ¿Cómo reordenaría los términos para que en ningún momento se produzca overflow? Justifique.

0x7744 + 0x5499 + 0x88BD + 0x9879

Respuesta:

Una forma sencilla de asegurar que no se produzca overflow es siempre sumar positivos con negativos. Al sumar un número positivo y uno negativo, el resultado se acerca hacia el centro del rango de representación, asegurando que no se produce overflow. Podemos ir ordenando en función del signo de los operandos y de los resultados intermedios. Notar que no alcanza con “intercalar” positivos y negativos, ya que debemos tener en cuenta también los resultados intermedios.

0x7744 y 0x5499 son positivos, mientras que 0x88BD y 0x9879 son negativos (comienzan con 1).

Por tanto comenzamos con 0x7744 + 0x88BD = 0x0001, es decir positivo.

Por tanto ponemos como tercer elemento el 0x9879 (que es negativo).

Haciendo la operación del resultado intermedio y el nuevo término da 0x0001 + 0x9879 = 0x987A, es decir negativo, que sumado al último término que es positivo nos asegura que no hay overflow:
0x987A + 0x5499 = 0xED13 (resultado final negativo)

Si el último término hubiera sido negativo, aún así podría haber dado un resultado correcto, pero en ese caso ya se dependería de los valores absolutos de los números para determinar.
Y eventualmente podríamos haber llegado al caso en que el resultado no fuera representable, con lo que sería imposible lograr evitar el overflow.
En este ejemplo, si sumábamos en el orden original es fácil ver que ya la primera suma da overflow, aunque el resultado total sí es representable, como quedó demostrado con el nuevo orden propuesto:

0x7744 + 0x88BD + 0x9879 + 0x5499

Problema 1

Se desea construir un circuito con un bit de entrada y un bit de salida. Se recibe una tira de 0s y 1s de forma serial y la salida deberá ser 1 mientras la tira sea potencialmente válida. Si en algún momento la tira no es válida la salida deberá ser 0 hasta que el circuito se reinicie.

Las tiras válidas tienen la forma: `0*(1011)*`. Los paréntesis agrupan símbolos. El `*` indica cualquier cantidad de repeticiones (de cero a infinito) de lo precedente, sea un 0 o un grupo 1011.

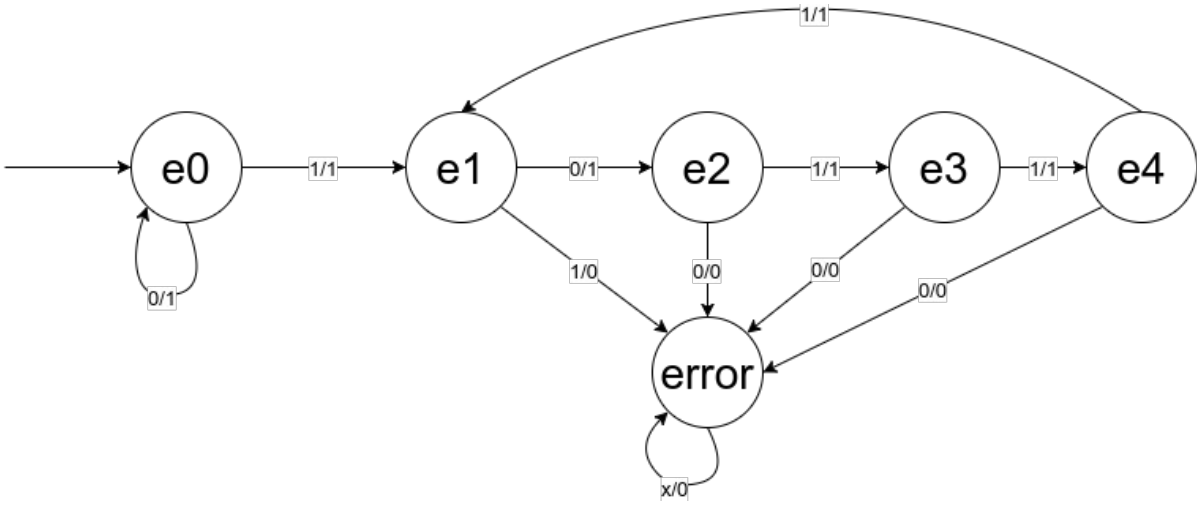
Por ejemplo, las siguientes tiras son válidas (se agregaron espacios para mejor visualización): “0 0 0”, “0 0 1011 1011 1011”, “1011 1011” y las siguientes no, a partir del dígito subrayado: “1<u>1</u>”, “0001<u>0</u>1”.

Se pide:

Construir el circuito aplicando la metodología del curso y usando flip-flops tipo D

Solución Problema 1:

El diagrama de estados de la máquina solicitada es:



La tabla de estados correspondiente se muestra a continuación:

E_n	Entrada	E_{n+1}	Salida
e_0	0	e_0	1
e_0	1	e_1	1
e_1	0	e_2	1
e_1	1	error	0

e_2	0	error	0
e_2	1	e_3	1
e_3	0	error	0
e_3	1	e_4	1
e_4	0	error	0
e_4	1	e_1	1
error	0	error	0
error	1	error	0

Se precisan 3 bits para codificar los 6 estados. A continuación se muestra la tabla de estados codificada:

E_n			**Entrada**	**E_{n+1}**			**Salida**
q_2	**q_1**	**q_0**	**e**	**q_2**	**q_1**	**q_0**	**s**
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	1
0	0	1	0	0	1	0	1
0	0	1	1	1	0	1	0
0	1	0	0	1	0	1	0
0	1	0	1	0	1	1	1
0	1	1	0	1	0	1	0
0	1	1	1	1	0	0	1
1	0	0	0	1	0	1	0
1	0	0	1	0	0	1	1
1	0	1	0	1	0	1	0
1	0	1	1	1	0	1	0

Se utilizarán 3 FF tipo D. La tabla de verdad se muestra a continuación:

q_2	q_1	q_0	e	s_2	s_1	s_0	s
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	1
0	0	1	0	0	1	0	1
0	0	1	1	1	0	1	0
0	1	0	0	1	0	1	0
0	1	0	1	0	1	1	1
0	1	1	0	1	0	1	0
0	1	1	1	1	0	0	1
1	0	0	0	1	0	1	0
1	0	0	1	0	0	1	1
1	0	1	0	1	0	1	0
1	0	1	1	1	0	1	0
1	1	0	0	x	x	x	x
1	1	0	1	x	x	x	x
1	1	1	0	x	x	x	x
1	1	1	1	x	x	x	x

Para la tabla de verdad anterior, los diagramas de Karnaugh pueden ser realizados de la siguiente manera:

s_2:

$q_0e\backslash q_2q_1$	00	01	11	10
00		1	x	1
01			x	
11	1	1	x	1
10		1	x	1

$s_2 = q_0e+e'q_1+e'q_2$

s_1:

$q_0e\backslash q_2q_1$	00	01	11	10
00			x	
01		1	x	
11			x	
10	1		x	

$$s_1 = q_1q_0'e+q_0e'q_2'q_1'$$

s_0:

$q_0e\backslash q_2q_1$	00	01	11	10
00		1	x	1
01	1	1	x	1
11	1		x	1
10		1	x	1

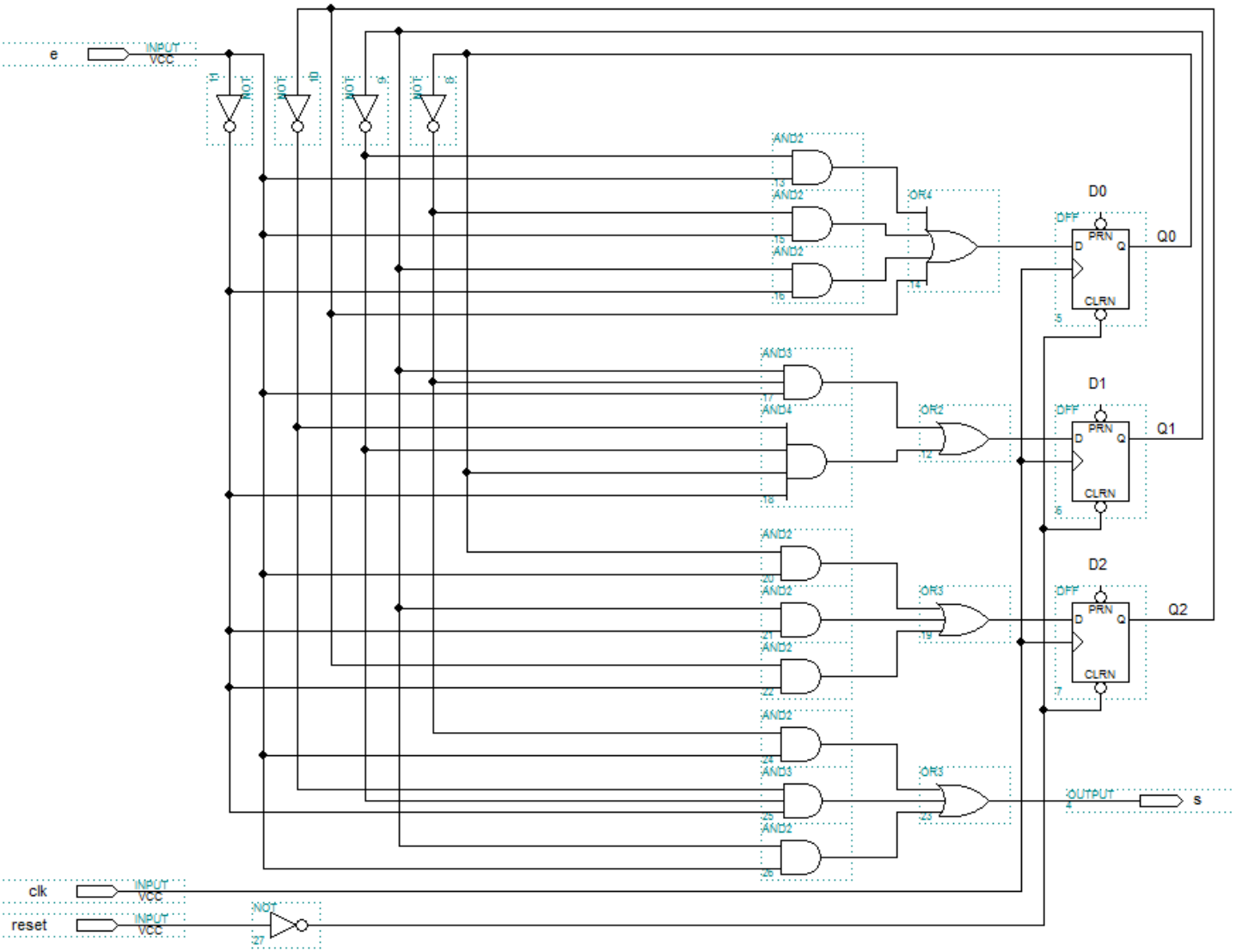
$$s_0 = q_1'e+q_0'e+e'q_1+q_2$$

s:

$q_0e\backslash q_2q_1$	00	01	11	10
00	1		x	
01	1	1	x	1
11		1	x	
10	1		x	

$$s = q_0'e+q_2'q_1'e'+q_1e$$

Circuito:


e
INPUT
VCC
NOT
AND2
AND3
OR4
OR2
OR3
D0
D1
D2
OFF
PRN
D
Q
CLRN
Q0
Q1
Q2
OUTPUT
s
clk
reset

Problema 2

Considere la siguiente estructura de árboles navideños.

```
typedef struct{
  char tipoChirimbolo;
  arbol* ramaDer;
  arbol* ramaIzq;
} arbol;
```

```
short contarArquiChirimbolos(arbol* arquiArbolito, char arquiChirimbolo){
      short chirimbolos = 0;
      if (arquiArbolito != NULL){
         chirimbolos += contarArquiChirimbolos(arquiArbolito→ramaDer, arquiChirimbolo);
         chirimbolos += contarArquiChirimbolos(arquiArbolito→ramaIzq, arquiChirimbolo);
         if (arquiArbolito->tipoChirimbolo == arquiChirimbolo){
           chirimbolos++;
         }
      }
      return chirimbolos;
}
```

Se pide:

Parte a)

Compilar la función *contarArquiChirimbolos* en assembler 8086. Considere que los punteros se representan como desplazamientos dentro del segmento ES, que los parámetros se reciben por stack y que el resultado se retorna en el stack. Es necesario preservar el valor de los registros.

El programa que va a invocar esta función lo hace de la siguiente manera (pseudocódigo assembler):

; PUSH arquiArbolito
; PUSH arquiChirimbolo
; CALL contarArquiChirimbolos
; POP resultado

Parte b)

Indique el máximo consumo de stack de su solución para un árbol de N nodos

Solución:

```
contarArquiChirimbolos PROC

      PUSH BP
      MOV BP, SP
      PUSH CX,
      PUSH BX
      PUSH DX
      XOR CX, CX
      MOV BX, [BP + 6]
      CMP BX, 0
      JZ FIN
; llamada recursiva derecha
      PUSH ES:[BX + 1]
      PUSH [BP + 4]
      CALL  contarArquiChirimbolos
      POP DX
      ADD CX, DX
; llamada recursiva izquierda
      PUSH ES:[BX + 3]
      PUSH [BP + 4]
      CALL contarChirimbolos
      POP DX
      ADD CX, DX
; compara tipos (1 byte)
      MOV DX, [BP + 4]
      CMP DL, ES:[BX]
      JNE FIN
      INC CX
FIN:
      MOV DX, [BP + 2]
      MOV [BP + 4], DX       ; acomodo IP
      MOV [BP + 6], CX       ; resultado
      POP DX
      POP BX
      POP CX
      POP BP
      ADD SP, 2
      RET

contarArquiChirimbolos ENDP
```

Parte b)

El peor caso ocurre cuando el árbol presentado es una lista, ya que para N nodos se realizan N llamadas recursivas más el paso base al llamar al árbol vacío desde el último nodo.

Luego, el paso base y recursivo utilizan la misma cantidad de espacio en el stack: 4 bytes para los parámetros, 2 bytes para el IP, y 8 bytes para guardar el contexto (4 registros) .

Por lo tanto, el consumo de stack para un árbol de N nodos es: 14 * (N + 1) bytes