Simulacro Primer Parcial de Arquitectura de Computadoras

2024

Instrucciones:

- **Indique su nombre, apellido y número de cédula en todas las hojas que entregue.**
- **Escriba las hojas de un solo lado. Empiece cada ejercicio en una hoja nueva.**
- **Las hojas deben estar numeradas y en la primer hoja debe escribirse el total.**
- **Apague su celular. No puede utilizar material ni calculadora.**
- **La duración del parcial es de tres horas. En dicho tiempo debe también completar sus datos.**

Pregunta 1

Compare los estilos de diseño CISC y RISC aplicados en tres de los elementos que caracterizan a una arquitectura Von Neumann.

Pregunta 2

Describa las entradas de un flip-flop D, indicando si son sincrónicas o asincrónicas. Explique las diferencias de funcionamiento entre un flip-flop D con señal de control por nivel y un flip-flop D con señal de control por flanco.

Pregunta 3

Dada la siguiente tira de bits:

1100100011000000

Para todas las representaciones enumeradas a continuación, provea el resultado numérico resultante de interpretar la tira provista como:

A) Punto Flotante de Media Precisión
B) Punto fijo (8 bits para la parte fraccionaria)
C) Desplazamiento d=215

Pregunta 4

Sea x un número entero con signo representado en complemento a dos de 3 bits. Implemente la función $f(x) = -x$ (siendo $-3 \leq x \leq 3$) utilizando únicamente multiplexores.

Problema 1

Sea una CPU RISC de 16 bits con instrucciones de 16 bits y 8 registros de propósito general de 16 bits (Reg0-Reg7)

Instrucción	**Descripción**
LOAD REG1, REG2	Guarda en REG2 el contenido de la dirección de memoria almacenada en REG1
STORE REG1, REG2	Escribe el contenido de REG1 en la dirección de memoria almacenada en REG2.
NOP	No realiza ninguna operación
MOVI INM, REG	Carga el valor constante INM de 9 bits en REG (pone en 0 los 7 bits más significativos)
ADD REG1, REG2, REG3	Suma REG1 y REG2 y guarda el resultado en REG3
SUB REG1, REG2, REG3	Resta REG2 a REG1 y guarda el resultado en REG3
AND REG1, REG2, REG3	Realiza el and bit a bit entre REG1 y REG2 y guarda el resultado en REG3
OR REG1, REG2, REG3	Realiza el or bit a bit entre REG1 y REG2 y guarda el resultado en REG3
NOT REG1, REG2	Guarda en REG2 el resultado de realizar un NOT bit a bit al REG1
SL REG1, INM, REG2	Realiza el shift a la izquierda de tantos bits de REG1 como indique el inmediato INM (de 4 bits) y el resultado lo pone en REG2
SR REG1, INM, REG2	Realiza shift a la derecha de tantos bits de REG1 como indique el inmediato INM (de 4 bits) y el resultado lo pone en REG2.
CMP REG1, REG2	Instrucción de comparación que realiza la resta REG1-REG2 sin guardar el resultado, actualizando las banderas de condición.
JZ INM	Si la bandera Z está en 1 salta INM instrucciones desde la posición actual. INM es un número con signo de 12 bits
JN INM	Si la bandera N está en 1 salta INM instrucciones desde la posición actual. INM es un número con signo de 12 bits
JMP INM	Instrucción de salto incondicional, salta INM instrucciones desde la posición actual. INM es un número con signo de 12 bits

Se pide:

a) Diseñar el formato de instrucción presentado e indicar la codificación de cada instrucción

b) Compilar el siguiente fragmento de código utilizando la arquitectura anterior (no se acepta el uso de etiquetas)

```
short suma = 0;
for (short a = 1023; a < 4097; a++){
     suma = suma + a;
}
suma = suma / 2;
```

c) Implemente en lenguaje de máquina las primeras 7 instrucciones

Problema 2

Se desea utilizar una ROM para implementar una función que determine la parte entera de un número representado en *Punto Flotante de Precisión Simple* (1 bit signo, 8 bits exponente, 23 bits parte fraccional). La parte entera a dar como salida deberá estar representada en *Complemento a 2* de 16 bits. La función también deberá indicar, mediante un bit, si la operación no puede realizarse por no ser representable la parte entera en ese formato (salida de Overflow).

Se pide:

A) Determinar tamaño y organización de la ROM necesaria para implementar la función. Construirla en base a ROMs de 1G x 8 y 1G x 4 y compuertas básicas. Se deberá utilizar la menor cantidad de ROMs posible.

Escribir en un lenguaje de alto nivel el programa que genera el contenido de la ROM. El lenguaje de alto nivel dispone solamente de aritmética entera y operaciones bit a bit (no dispone de aritmética de Punto Flotante).

Solución Pregunta 1

Con respecto al set de instrucciones, las arquitecturas CISC suelen tener un conjunto mucho más amplio de instrucciones que las arquitecturas RISC (que como su sigla lo sugiere tienen un conjunto reducido de instrucciones). A nivel de formato de instrucción, las arquitecturas CISC suelen implementar instrucciones de largo variable, mientras que las arquitecturas RISC utilizan una codificación de largo fijo. Por último, una tercer diferencia se puede encontrar en el set de registros, donde las arquitecturas CISC muchas veces ofrecen registros con propósitos específicos, mientras que las RISC disponen de registros uniformes, 'sin personalidad' y, muchas veces, un número mayor.

Solución Pregunta 2

Las entradas de un flip-flop D son:

- Entrada D: sincrónica
- Clock: sincrónica
- Clear: asincrónica
- Set: asincrónica
- Clock Enable: sincrónica

En el caso de los flip-flops con entrada de control por nivel, el nuevo valor de la salida corresponde al valor de la entrada D mientras la señal de control está en 1. Mientras la señal de control está en 0 la salida mantiene el valor de la entrada D inmediatamente antes de la transición de 1 a 0 de la entrada de control.

En el caso de los flip-flops con entrada de control por flanco, el nuevo valor de la salida corresponde al valor de la entrada D al momento de la transición de la entrada de reloj (CLK) de 0 a 1 (es decir el "flanco ascendente"). Este nuevo valor de la salida es adoptado inmediatamente después de ocurrido dicho flanco.

Solución Pregunta 3

a) La representación media precisión utiliza el bit más significativo para el signo, luego cinco bits para el exponente y los diez bits menos significativos para la mantisa (signo = 1, exponente = 10010 (d=15) y mantisa = 0011000000).

Por lo tanto el resultado numérico es:

$-1{,}0011 \cdot 2^3$ b = -1001,1b = -9,5.

b) Se representa en complemento a dos tomando los ocho bits menos significativos como la parte fraccionaria. Al descomplementar la representación se obtiene el código 0011011101000000. Tomando sus ocho bits menos significativos como la parte fraccionaria se obtiene el resultado numérico es:

-110111,01b = -55,25.

c) El código representa un número positivo. Para obtener el resultado numérico debe restarsele el desplazamiento ($d=2^{15}$ = 1000000000000000). Por lo tanto el resultado numérico es:

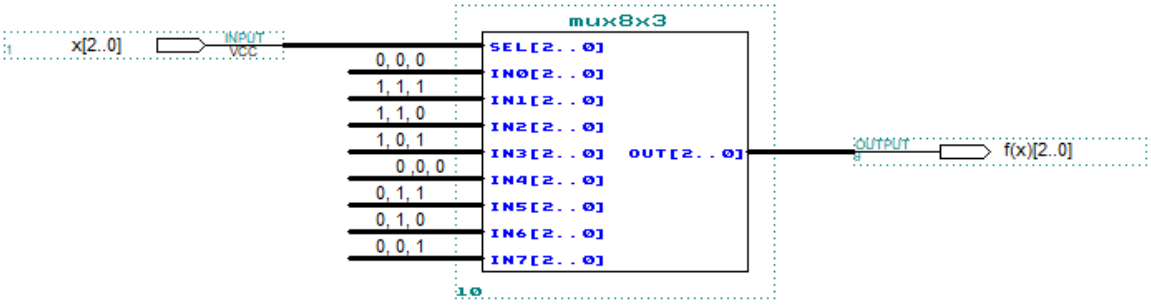
100100011000000b = 18624.

Solución Pregunta 4

f(0) = 0 => f(000) = 000
f(1) = -1 => f(001) = 111
f(2) = -2 => f(010) = 110
f(3) = -3 => f(011) = 101
f(-3) = 3 => f(101) = 011
f(-2) = 2 => f(110) = 010
f(-1) = 1 => f(111) = 001

Como la función no está definida para -4 (100), en forma arbitraria tomaremos la salida 100 para ese caso.

Esta función se implementará con un mux 8x3. Los tres bits de selección van a tomar como entrada el numero x y los tres bits de salida darán el resultado.



Solución Problema 1:

a) Dado que tenemos 15 instrucciones precisamos 4 bits para codificarlas. Los registros son 8, por lo tanto se precisan 3 bits. Rellenaremos con 0 a la derecha los bits excedentes en caso necesario.

Utilizaremos los siguientes códigos y formatos:

```
LOAD      → 0000RRRrrr000000
STORE     → 0001RRRrrr000000
NOP       → 0010000000000000
MOVI      → 0011RRRIIIIIIIII
ADD       → 0100RRRrrrDDD000
SUB       → 0101RRRrrrDDD000
AND       → 0110RRRrrrDDD000
OR        → 0111RRRrrrDDD000
NOT       → 1000RRRrrr000000
SL        → 1001RRRIIIIDDD00
SR        → 1010RRRIIIIDDD00
CMP       → 1011RRRrrr000000
JZ        → 1100IIIIIIIIIIII
JN        → 1101IIIIIIIIIIII
JMP       → 1110IIIIIIIIIIII
```

La codificación de los registros es:
R0 – 000
R1 – 001
R2 – 010
R3 – 011
R4 – 100
R5 – 101
R6 – 110
R7 – 111

b) La compilación del fragmento de código es la siguiente:

```
MOVI 0, R1      // R1 = suma
MOVI 256, R3
SL R3, 2, R2    // R2 = a = 1024
MOVI 1, R7
SUB R2, R7, R2 // a = 1023
SL R3, 4, R3    // R3 = 4096
ADD R3, R7, R3 // R3 = 4097
ADD R1, R2, R1 // suma = suma + a
ADD R2, R7, R2 // a++
CMP R2, R3
JN -3
SR R1, 1, R1
```

c)

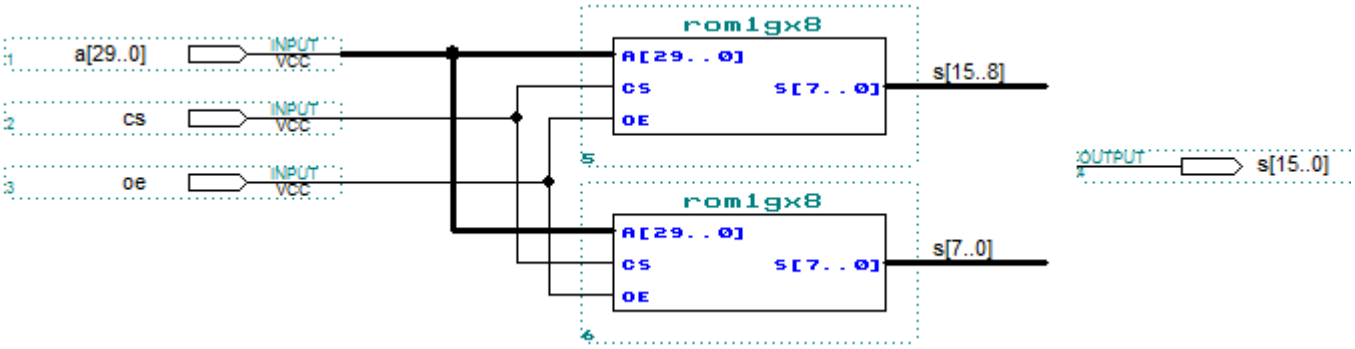
```
0011001000000000 // MOVI 0, R1 → MOVI:0011, R1:001, inm 0: 000000000
0011011100000000 // MOVI 256, R3 → MOVI:0011, R3:011, inm 256: 100000000
1001011001001000 // SL R3,2,R2 →SL:1001, R3:011, inm 2: 0010, R2:010
0011111000000001 // MOVI 1, R7 → MOVI:0011, R3:111, inm 1: 000000001
0101011111010000 // SUB R2,R7,R2 →SUB:0101, R2:010, R7: 111, R2:010
1001011010001100 // SL R3,4,R3 →SL:1001, R3:011, inm 4: 0100, R3:011
0100011111010000 // ADD R3,R7,R3 →SUB:0100, R3:011, R7: 111, R3:011
```

Solución Problema 2:

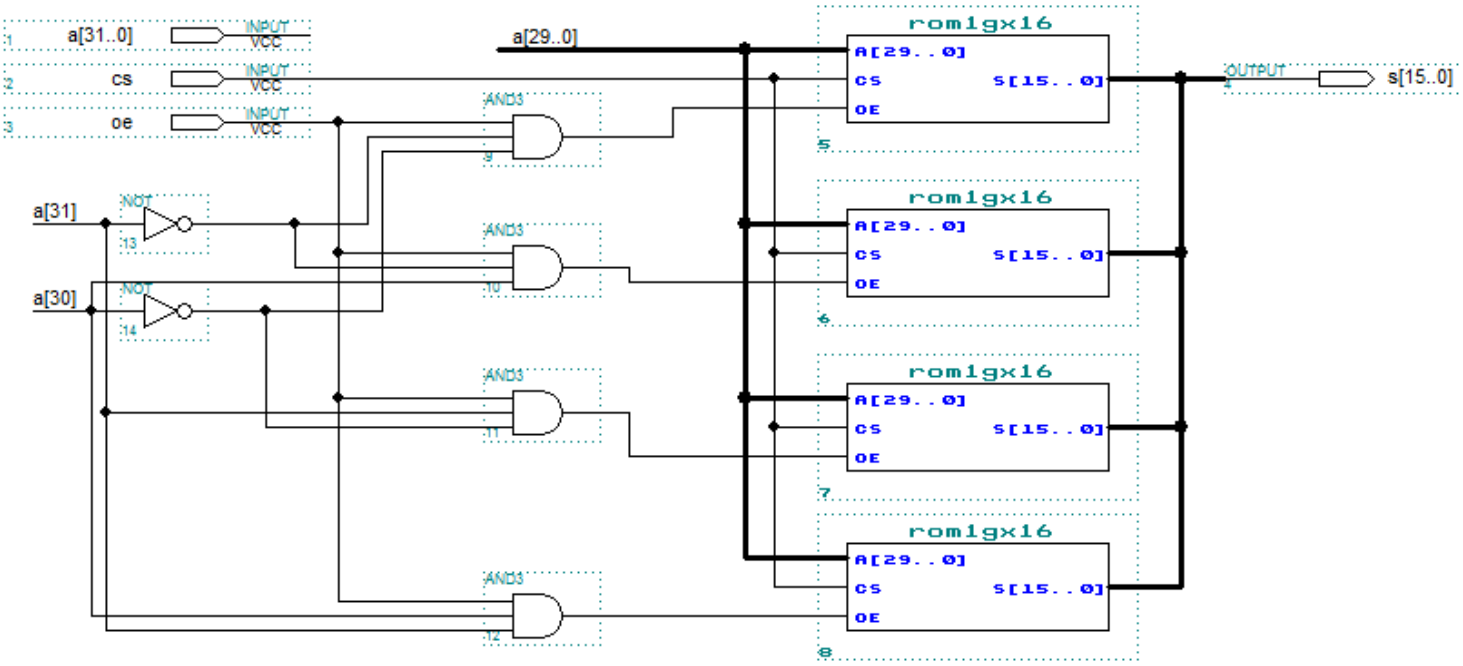
a) Necesitamos una ROM con 32 bits de entrada (E31..E0) que representan el número en punto flotante a convertir y 17 bits de salida (S16 que indica si hay Overflow y S15..S0 para representar la parte entera del número).

Por lo tanto se precisa construir una ROM de 4Gx17, que tiene un tamaño de 2^32*(2^4+1) bits = (8Gbytes + 0,5Gbytes) = 8,5GBytes

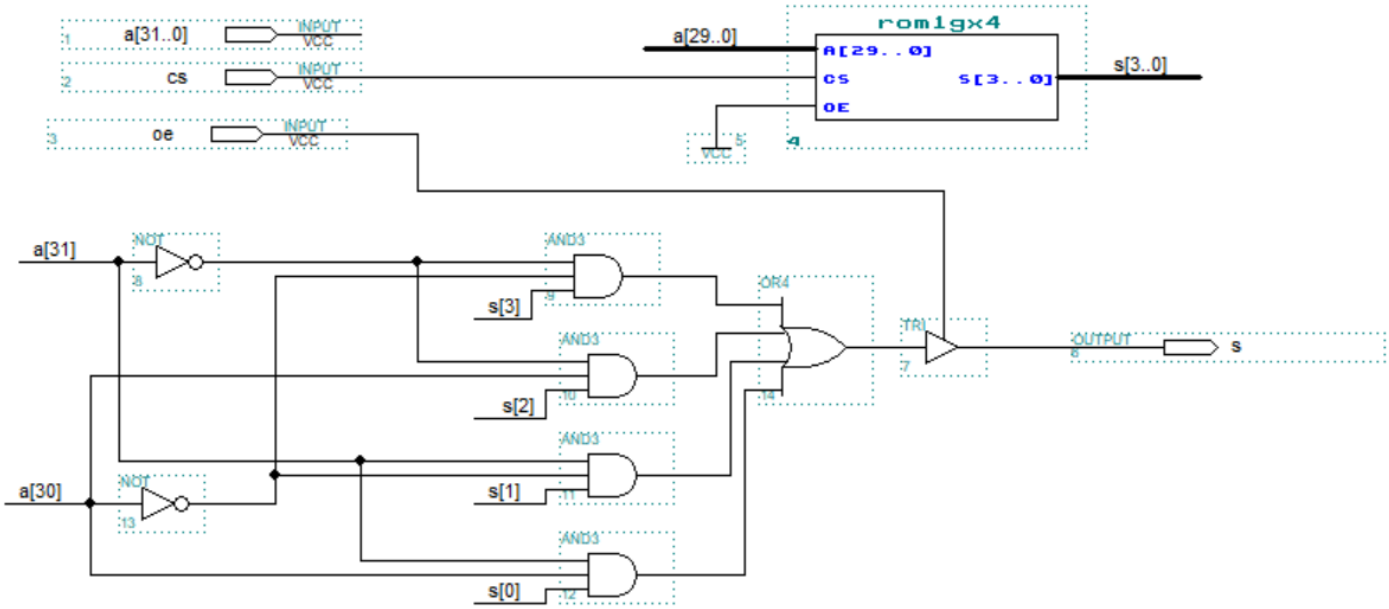
Primero construimos una ROM de 1Gx16 utilizando dos ROMs de 1Gx8, la cual llamaremos ROM A:



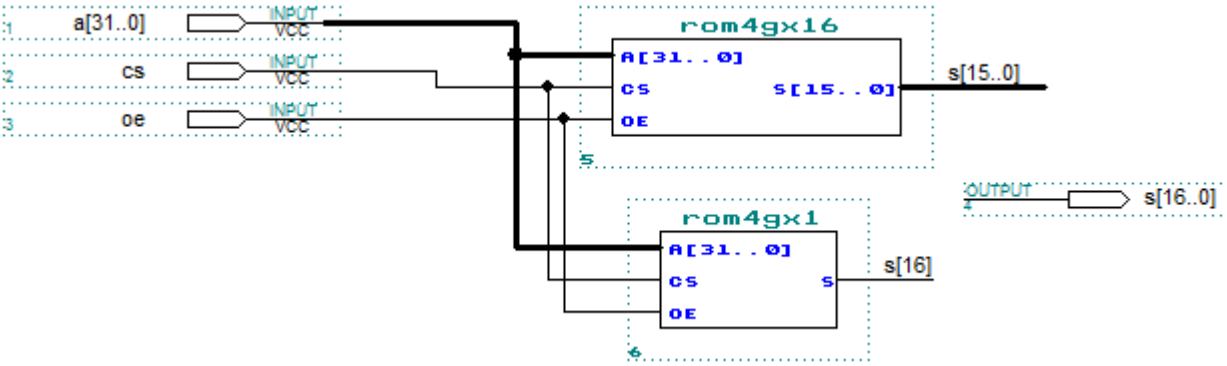
Luego construimos una ROM de 4Gx16 con 4 ROM A, la cual llamaremos ROM B:



Utilizamos una ROM de 1Gx4 para construir una de 4Gx1. La cual llamaremos ROM C:



Por último, utilizamos una ROM B y una ROM C para construir la ROM pedida:



b)

formato: [S|EXP8|FRAC23]

Los números en punto flotante de precisión simple normalizados se representan como (-1)^s*1,frac * 2^(exp8-127)

Por su parte, el rango de números en complemento a 2 de 16 bits va de -2^15 a 2^15-1

```
int ROM[1<<32]; // ints de 32 bits

void cargaROM()
{
  int entera;
  for(int codExp = 0; codExp < 255; codExp++)
  {
```

```
    int exp = codExp-127;
    for(int frac = (1<<23); frac >= 0; frac--)
    {
     entera = 0;
     overflow = 0;
     if(exp >= 0)
     {
       if(exp >= 15)
          overflow = 1<<16;
       else
          entera = 1<<exp | frac>>(23-exp);
     }
     ROM[codExp<<23|frac] = entera + overflow;
     if(exp == 15 && frac < (1<<8)) // Caso de borde
          overflow = 0;
     ROM[1<<31|codExp<<23|frac] = (-entera + overflow) & 0x1FFFF;
    }
  }
}
```

Solución alternativa:

```
int ROM[1<<32]; // ints de 32 bits

void cargaROM()
{
  for(int dir = 0; dir < 1<<32; dir++)
  {
     int signo = dir / 1<<31;
     int exp = ((dir / 1<<23) & 0xFF) - 127;
     int fracc =  dir & 0x7FFFFF;
     int overflow = 0;
     int entera = 0;
     if(exp >= 0)
     {
       if(exp >= 15)
          overflow = 1<<16;
       else
          entera = 1<<exp | frac>>(23-exp);
     }
     if((signo == 1) && (exp == 15) && (frac < (1<<8))) // Caso de borde
          overflow = 0;
     if (signo == 1) ROM[dir] = entera + overflow;
     else ROM[dir] = (-entera + overflow) & 0x1FFFF;
   }
 }
```