Arquitectura de Computadores – IIC2343

I1

14 septiembre 2017

1. Complemento de 2

a) [**1 pt.**] ¿Cómo se detecta *overflow* después de una operación aritmética?, tanto para números positivos como para números negativos. Justifica.

El resultado es del signo opuesto a los operandos, o, equivalentemente, el bit de signo es incorrecto: un 1 (a la izquierda) cuando el número es positivo; un 0, cuando el número es negativo.

El número positivo de mayor valor es 011…1; si sumamos 1 a este número, obtenemos 100…0, pero éste es un número negativo en complemento de 2

b) [**1 pt.**] Sea x un número binario de 8 bits; y sea $\tilde{x}$ una secuencia de 8 bits tal que cada bit de $\tilde{x}$ es la negación del correspondiente bit de x; p.ej., si x = 01101001, entonces $\tilde{x}$ = 10010110. ¿Cuál es la relación aritmética/algebraica entre $-x$ y $\tilde{x}$?

Para negar un número x *en complemento de 2, negamos cada bit —* ~x *— y a este valor le sumamos 1. Es decir,* –x = ~x + 1 => x + ~x = –1. (*Otra manera de ver esto, es que a partir de esta última ecuación se deduce la "regla" para determinar* –x *a partir de* x *en complemento de 2.*)

c) [**1 pt.**] Sea x un número binario de 8 bits; ¿cómo se lo lleva a 16 bits?, tanto para números positivos como para números negativos. Justifica.

Para números positivos (el bit de más a izquierda es 0), simplemente rellenamos con ocho 0's; es (más o menos) obvio que el valor numérico no cambia.

Para números negativos (el bit de más a la izquierda es 1) … rellenamos con 1's ! Esto se puede comprobar, ya que en el número original (de 8 bits) el 1 de más a la izquierda se interpreta como $1 \times -2^7 = -128$*; en cambio, en el nuevo número (de 16 bits) este mismo 1 se iterpreta como* $1 \times 2^7 = 128$*. Si a este valor sumamos el valor de los nuevos ocho 1's a su izquierda (recordando que el nuevo 1 de más a la izquierda ahora vale* 1×-2^{15} *), obtenemos nuevamente –128. Es decir, el valor numérico de los 8 bits originales* ***no cambia*** *al agregar ocho 1's a la izquierda.*

d) [**3 pts.**] He aquí la multiplicación de 1000_{10} por 1001_{10} :

multiplicando		$1\,0\,0\,0_{10}$
multiplicador	×	$1\,0\,0\,1_{10}$
		$1\,0\,0\,0$
		$0\,0\,0\,0$
		$0\,0\,0\,0$
		$1\,0\,0\,0$
producto		$1\,0\,0\,1\,0\,0\,0_{10}$

Es decir, tomamos los dígitos del multiplicador uno a uno de derecha a izquierda, multiplicamos el multiplicando por el dígito del multiplicador, y desplazamos el producto intermedio un dígito a la izquierda con respecto al producto intermedio anterior. Cuando los dígitos involucrados son solo 0 y 1 (como en el ejemplo y como en el caso de los números binarios), vemos que cada paso del proceso es simple:

i) Escribir una copia del multiplicando en la posición correcta si el dígito del multiplicador es 1; o
ii) Escribir 0 en la posición correcta si el dígito es 0.

Dibuja el diagrama de bloques del hardware para multiplicación de números de 32 bits. Inicialmente, el multiplicando y el multiplicador están en registros , y usamos un tercer registro para el producto, inicializado en 0; además, contamos con un sumador y con una unidad de control que controla a cada registro y al sumador. Explica tu diagrama en función del algoritmo de multiplicación sugerido.

Además, responde y justifica: Para cada uno de los tres registros, ¿cuántos bits tiene que tener? y ¿qué operación tiene que poder hacerse en el registro?

El registro del multiplicando debe tener la capacidad de hacer shift a la izquierda sin perder sus bits originales —es decir, con cada shift multiplica su valor por 2— por lo que debe tener 64 bits; la señal de cuándo hacer shift a la izquierda proviene de la unidad de control.

Este valor del multiplicando ingresa como uno de los dos operandos al sumador de 64 bits, cuando el bit que se está mirando del multiplicador es 1 (por i). En cambio, cuando el bit que se está mirando del multiplicador es 0, entra un 0 al sumador (por ii). Esto significa que debe ser posible mirar uno por uno los bits del multiplicador, de derecha a izquierda. Por lo tanto, el registro del multiplicador debe tener la capacidad de hacer shift a la derecha y "enviar" a la unidad de control su bit de más a la derecha (que va cambiando con cada shift). La señal de cuándo hacer shift a la derecha proviene de la misma unidad de control. Este registro puede ser de solo 32 bits.

La unidad de control, cada vez que recibe el bit de más a la derecha del multiplicador (bit que va cambiando con cada shift a la derecha del registro del multiplicador), decide si el valor del multiplicando o un 0 debe ingresar como operando al sumador.

Finalmente, la salida del sumador va al registro de 64 bits que almacena el producto; por lo tanto, este registro debe tener no solo 64 bits, sino también la capacidad de ser escrito. La señal de cuándo este registro debe actua-lizar su valor según la salida del sumador proviene también de la unidad de control. Además, el valor de este registro es a su vez el otro operando del sumador.

2. Considera el siguiente diagrama de bloques del computador básico, aún incompleto.

a) Explica el rol del multiplexor *Address*; da un ejemplo de su funcionamiento

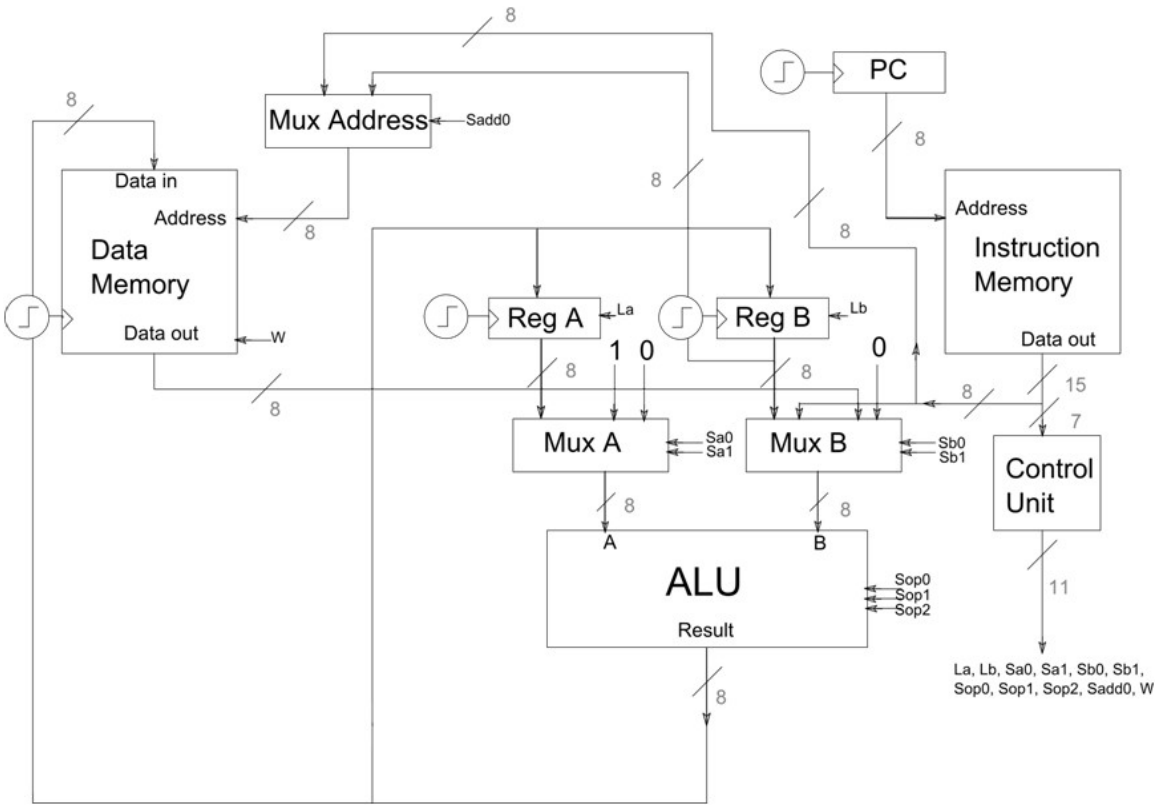
Decidir cuál dirección se "ingresa" a la memoria: la que viene especificada como literal en la instrucción (desde la instruction memory*) o la que está almecenada en el registro B.* ***Falta el ejemplo.***

b) Explica qué es necesario agregar para permitir instrucciones del tipo saltos incondicionales; y explica cómo funcionaría en este caso un salto incondicional.

Para saltos incondicionales es necesario poder poner en el registro PC *dirección de la instrucción a la cual se quiere ir: hay que conectar la salida de la* Instruction Memory *a la entrada del* PC*, y controlar con una señal de la* Control Unit *esta actualización del* PC *cuando la instrucción sea un salto.*

c) Explica qué es necesario agregar a tu respuesta a b) para permitir instrucciones del tipo saltos condicionales; y explica cómo funcionaría en este caso un salto condicional.

Es necesario poder evaluar una condición (aritmética) antes de decidir si se salta o no. El resultado de la evaluación de la condición (hecho en la ALU*) se codifica en varios bits, p.ej.,* N (negative) *y* Z (zero)*, que se cargan en un nuevo registro* Status. *La* Control Unit *lee el valor de estos bits al momento de ejecutar un salto condicional.*



3. Considera el código en *assembly* que se muestra a la derecha. **a**) Explica el propósito de, y cómo funciona, la secuencia de instrucciones PUSH B, CALL *func* y POP B; y cuál es el efecto de que no haya una instrucción PUSH A (justo antes) ni una instrucción POP A (justo después). **b**) Explica el propósito de, y cómo funciona, la secuencia de instrucciones CMP A,0 y JEQ *end*, y en particular qué ocurre cuando el contenido del registro A es 0. Tanto en a) como en b), identifica las componentes del computador básico que están involucradas y explica qué rol juegan.

```
DATA:
   r 3

CODE :
   MOV  A, ( r )
   MOV  B, 2
   PUSH  B
   CALL  func
   POP  B
   JMP  f i n i s h

finish:
```

```
func :
      s h i f t :
            MOV  A,B
            CMP  A, 0
            JEQ end
            DEC  A
            MOV  B,A
            MOV  A, ( r )
            SHL  A,A
            MOV  ( r ) ,A
            JMP  s h i f t
      end :
            RET
```

a) El propósito es ejecutar la subrutina *func*, pero de modo que el valor del registro *B* al terminar la ejecución de *func* y volver al programa principal (CODE) sea el mismo que *B* tenía justo antes de hacer la llamada; es decir, las actualizaciones del registro *B* al interior de *func* no se ven reflejadas en CODE. Esto ocurre porque el valor de *B* justo antes de la llamada es guardado en el stack, mediante PUSH B, y es restaurado justo después de la llamada mediante POP B.

Por esto mismo, la ausencia de las instrucciones PUSH A y POP A tiene el efecto contrario en el caso del registro *A*: los cambios que el registro *A* sufre dentro de *func* se ven reflejados en CODE cuando *func* termina su ejecución.

Las instrucciones PUSH y POP acualizan el valor del *stack pointer* —justo después y justo antes, respectivamente— de tener acceso a la dirección de memoria apuntada por el *stack pointer*.

b) El propósito es terminar la ejecución de *func* cuando el valor del registro *A* sea igual a 0. La instrucción COMP A, 0 ejecuta *A*–0 (resta 0 al contenido del registro *A*); el resultado no es almacenado sino procesado, de modo que solo si es 0 (todos los bits del resultado son 0), entonces el bit *Z* del registro *Status* queda en 1. Luego, JEQ *end* salta a la instrucción identificada con la etiqueta *end* (coloca en el registro *PC* la dirección de esta instrucción) si el bit *Z* es 1.