Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencias de la Computación

IIC2343 – Arquitectura de Computadores

Ayudantía 4 – Solución propuesta

Profesores: Hans-Albert Löbel Díaz, Jurgen Dieter Heysen Palacios

Ayudante: Germán Leandro Contreras Sagredo (glcontreras@uc.cl)

Nota al lector

El título dice "solución propuesta" por una razón bien sencilla: Estos ejercicios pueden tener más de un desarrollo correcto. Lo que se pretende hacer aquí es mostrar un camino a la solución, sin excluir la posibilidad de rutas alternativas igual de correctas.

Preguntas

1. a. **(I2 - II/2015)** Compare las arquitecturas Harvard y Von Neumann desde el punto de vista del tiempo de ejecución de las instrucciones. Fundamente y explique claramente las diferencias.

   En la arquitectura de Harvard (utilizada para el computador básico estudiado en el curso) se tiene un tiempo promedio menor en ejecución si se compara con la arquitectura Von Neumann. Esto, debido a que en la primera (salvo por las instrucciones `POP` y `RET`) utiliza un ciclo por instrucción, mientras que Von Neumann utiliza al menos dos o tres (un ciclo donde se obtiene el *opcode* de la instrucción y se transfiere, otro donde se obtiene el literal y se transfiere -estas dos primeras se podrían realizar en uno asumiendo un *tradeoff* con respecto al rango de valores de los literales y *opcodes*-, y el último para ejecutar la instrucción en sí). Si bien hay instrucciones que se podrían seguir ejecutando en un ciclo (por ejemplo, las operaciones de la `ALU` sobre los registros `A` y `B`), todo lo que conlleve a accesos a memoria necesitaría al menos dos ciclos (uno para acceder a la instrucción y literal en memoria, y el otro para acceder a la variable almacenada). Por esto, se asume que para toda instrucción en la arquitectura Von Neumann se necesitan dos o tres ciclos en promedio (por comodidad, en general se asumirán tres).

b. **(I2 - I/2017)** ¿Cuántos ciclos como mínimo puede tomar en un computador con arquitectura Von Neumann, una instrucción que lea y luego modifique el contenido de una posición de memoria?

Antes de desarrollar la pregunta, sin pérdida de generalidad, se asume que **leer** hace referencia a una instrucción del tipo `MOV A,(Dir)`, mientras que **modificar** usa una instrucción del tipo `MOV (Dir),A`, habiendo realizado un cambio sobre el registro donde se almacenó el dato de la memoria (por ejemplo, `ADD A,3`).

Aquí nos encontramos con dos casos:

- Asumiendo que no podemos obtener el *opcode* y el literal en un solo ciclo:
  1) Se obtiene el *opcode* de `MOV A,(Dir)` y se envía a la unidad de control, la que posteriormente propaga las señales de control necesarias para la ejecución.
  2) Se obtiene el literal (dirección de memoria) y se propaga a los multiplexores correspondientes (en este caso, al `Mux Address`).
  3) Se ejecuta la instrucción completa en el flanco de subida del tercer ciclo, donde se obtiene el dato almacenado en `Dir` en el registro `A`.
  4) Se obtiene el *opcode* de la operación a realizar sobre el registro `A` y se envía a la unidad de control, la que posteriormente propaga las señales de control necesarias para la ejecución.
  5) Se obtiene el literal para la operación y se propaga a los multiplexores correspondientes (podemos asumir, sin pérdida de generalidad, un literal `Lit` a sumarse con `A`).
  6) Se ejecuta la instrucción completa en el flanco de subida del sexto ciclo, donde el resultado de la operación se almacena, nuevamente, en el registro `A`.
  7) Se obtiene el *opcode* de `MOV (Dir),A` y se envía a la unidad de control, la que posteriormente propaga las señales de control necesarias para la ejecución.
  8) Se obtiene el literal (dirección de memoria) y se propaga a los multiplexores correspondientes (en este caso, al `Mux Address`).
  9) Se ejecuta la instrucción completa en el flanco de subida del noveno ciclo, donde el dato del registro `A` se almacena en la dirección de memoria `Dir`.
- Asumiendo que sí podemos:
  1) Se obtiene el *opcode* de la instrucción `MOV A,(Dir)` y el literal de la dirección, propagándose a los componentes correspondientes.
  2) Se ejecuta la instrucción, almacenando el dato de la dirección `Dir` en el registro `A`.
  3) Se obtiene el *opcode* de la instrucción a ejecutar sobre el registro `A` y el literal para operar, propagándose a los componentes correspondientes.
  4) Se ejecuta la instrucción, almacenando el resultado de la operación en el registro `A`.
  5) Se obtiene el *opcode* de la instrucción `MOV (Dir),A` y el literal de la dirección, propagándose a los componentes correspondientes.
  6) Se ejecuta la instrucción, almacenando en la dirección `A` el resultado que se encuentra en el registro `A`.

En ambos casos, se ve que la cantidad de ciclos necesaria para la ejecución de lo pedido aumenta considerablemente si se hace el contraste con una arquitectura Harvard.

c. **(I2 - II/2014)** Modifique el computador básico, para que este utilice un esquema Von Neumann, *i.e.*, memoria de datos e instrucciones unificadas en una sola.

La principal modificación que se le debe realizar al esquema del computador básico es unir la memoria de instrucciones con la memoria de datos en una sola. Esto claramente modifica los ciclos requeridos por instrucción, por lo que una forma de solucionarlo es dejar por *default* 3 ciclos: El primero se utiliza para enviar el *opcode* y que la unidad de control propague las señales correspondientes (cuidando que no se altere el contenido de los registros), el segundo para enviar el literal correspondiente para realizar las operaciones y, el tercero, para hacer finalmente la ejecución de la instrucción. Entre las consideraciones importantes se tiene el manejo del registro `PC`, donde debe suspender su incremento durante el tercer ciclo de ejecución. Si no se hiciera esto, durante el tercer ciclo saldría de la memoria el *opcode* de la siguiente instrucción, afectando la ejecución original. Por otra parte, también se debe tomar en cuenta que en la unidad de control es necesario realizar un bloqueo de la entrada durante el segundo y tercer ciclo de una instrucción. Esto, con el fin de evitar un cambio en las señales de control provenientes de esta componente. A continuación, **un posible** diagrama de esta modificación.

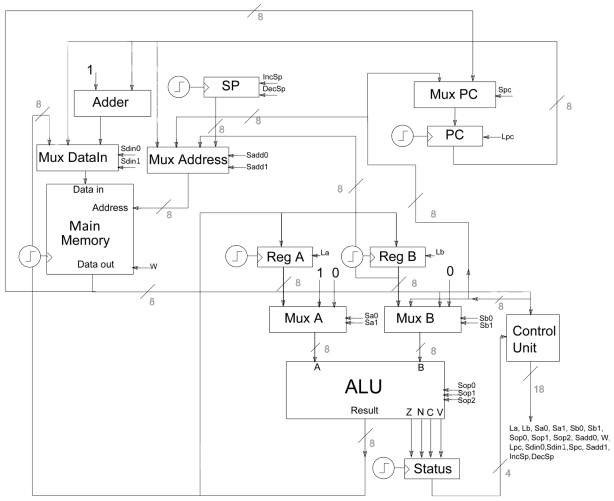

Figura 1: Diagrama del computador básico con arquitectura Von Neumann.

De aquí notamos lo siguiente:

- La salida de la memoria principal **debe** estar conectada directamente a la unidad de control, ya que puede corresponder a un *opcode*. Esta misma salida se usa como el literal de las instrucciones y los datos de memoria.
- `Mux Address` ahora también recibe el valor del registro `PC` como dirección, dado que las instrucciones se encuentran en la memoria principal.
- Al hacer uso de la memoria principal, se trabaja con una RAM de palabras de 8 bits. Por lo tanto, los *opcodes* y los literales se ajustan a estas dimensiones. Si se dejara un *opcode* y literal por palabra, ambos se verían perjudicados por un rango menor.

d. **(I2 - II/2014)** Dada la microarquitectura del computador básico, ¿es posible crear una ISA distinta la actual? Argumente su respuesta.

Sí, es posible, ya que esta se puede modificar de dos formas:

- Se puede usar un nuevo *opcode* que haga uso de una combinación de señales no utilizada antes para un nuevo comando. Por ejemplo, la combinación de señales que obtiene la suma del registro `A` y `B`, y luego guarda el resultado en ambos ($L_A = 1, L_B = 1, S_A =$ `A`, $S_B =$ `B`, $S_{op} =$ `ADD`).
- Usando una nueva instrucción que corresponda a la combinación de distintos *opcodes*. Por ejemplo, una instrucción que incremente en una unidad una variable almacenada en una dirección de memoria: `INC (dir) = PUSH B - MOV B,(dir) - INC B - MOV (dir),B - POP B`. Notar que la primera y última instrucción se usan de forma que no perdamos el valor almacenado en el registro `B`, pues la funcionalidad de la instrucción definida **no debiese** modificar el contenido de los registros si no se indica.

e. **(I2 - I/2015)** ¿Es posible agregar al `Assembly` del computador básico la instrucción `MOV A,(A+B)`, sin modificar la microarquitectura? Justifique su respuesta en cualquiera de los dos casos.

Sí, es posible. Basta con asignarle al nuevo comando los *opcodes* de las siguientes instrucciones existentes de forma consecutiva: `PUSH B - ADD B,A - MOV A,(B) - POP B`.

f. **(I2 - II/2016)** Modifique (solo) la ISA del computador básico para soportar la instrucción `CALL reg`, que permite llamar a la subrutina ubicada en la dirección de memoria almacenada en el registro `reg`.

Definimos en nuestra ISA la instrucción `CALL reg` de la siguiente forma:

- Primero se ejecuta la instrucción `MOV (dirreg),reg` para almacenar en la dirección `dir` el valor almacenado en uno de los registros (ya sea `A` o `B`). Se puede asumir que `dirreg` es una dirección fija con uso exclusivo para esta instrucción, lo que se puede definir en el *Assembler*.
- Añadimos la instrucción `CALL (dir)`, que carga en el registro `PC` el valor almacenado en la dirección `dir`. Notar que esto debe ocurrir en dos ciclos, ya que primero tenemos que guardar en el *stack* la posición `PC+1` para poder volver al retornar el llamado (*i.e.* como en las instrucciones `PUSH, CALL`, pero con $L_{pc} = 0$), y otro para obtener el valor almacenado en `dir` y cargarlo en el registro `PC`, lo que se puede hacer con la siguiente combinación de señales en el computador básico: $L_{pc} = 1$, $S_{pc} =$ `RAM Data out`, $S_{add} =$ `Lit`. En este caso, `Lit` será el literal correspondiente a la dirección `dir`, por lo que en este caso se define `dir = dirreg`.

El cuidado que tendría que tener el programador es de no alterar el contenido de la dirección `dirreg` en ningún momento.

2. a. **(I1 - I/2013)** ¿Cuál es el valor del número 11000001100000000000000000000000, representado mediante el tipo de dato `float`?

Sabemos que los `float` siguen la siguiente estructura (en el orden mencionado):

1) **Signo:** Un bit.
2) **Exponente:** 8 bits.
3) **Significante:** 23 bits.

Entonces, tenemos que:

- Signo = 1 (que representa un número negativo).
- Exponente = 10000011 = 131 (en base 10).
- Significante = 00000000000000000000000

En este caso, nuestro número es:

$$x = (-1)^1 * 1,00000000000000000000000 * 10^{(131-127)_2}$$

Entonces, tenemos finalmente que $x = -(10^{100})_2 = -2^4 = -16$.

b. **(I1 - II/2012)** Escriba en formato `float` el número 16,375 (decimal). Indique cómo se divide y qué significa cada una de las partes del *string* de bits.

Primero, necesitamos pasar el número a base binaria. Para ello, usamos el método de las restas sucesivas, que funciona de la siguiente forma:

- Obtenemos el mayor número 2^N que sea menor o igual al número que deseamos transformar.
- Al número original se le resta 2^N, ponemos un 1 en su posición (que en este caso sería la primera), y pasamos a una potencia de una unidad menor.
- Si 2^{N-1} es mayor a nuestro resultado anterior, disminuimos nuevamente la potencia en una unidad, y esta no la utilizamos (ponemos un 0 en su posición). En caso contrario, se resta (poniendo un 1 en su posición), y seguimos el mismo procedimiento con el nuevo número ya sustraido, y la potencia 2^{N-2}
- Esto se realiza hasta que el resultado sea 0. Si nunca nos da ese número, entonces estamos en presencia de un número binario infinito (que puede o no tener periodo).

En este caso, tenemos el siguiente desarrollo:

$$\begin{aligned}
16,375 - 1 * 2^4 &= 00,375 \\
00,375 - 0 * 2^3 &= 00,375 \\
00,375 - 0 * 2^2 &= 00,375 \\
00,375 - 0 * 2^1 &= 00,375 \\
00,375 - 0 * 2^0 &= 00,375 \\
00,375 - 0 * 2^{-1} &= 00,375 \\
00,375 - 1 * 2^{-2} &= 00,125 \\
00,125 - 1 * 2^{-3} &= 0
\end{aligned}$$

Por lo tanto, nuestro número en binario corresponde a 10000,011 (notar que la coma se coloca después de que comienzan a utilizarse potencias negativas). Esto es lo mismo que 1,0000011 $*10^{100}$ (notar aquí que multiplicar por 10 en binario es equivalente a hacer un `shift left`, y la potencia es 100 ya que este número corresponde a 4 en binario).

Para escribir nuestro número, analizamos las 3 partes importantes:

- **Signo:** Como el signo es positivo, tenemos que el bit de signo es 0.
- **Exponente:** El exponente es 4, pero como está desfasado: $x - 127 = 4 \rightarrow x = 131$. Utilizamos los 8 bits asignados para representar este número: 10000011
- **Significante:** Tenemos que el significante es 0000011, y para completar los 16 bits faltantes, utilizamos solo ceros.

Finalmente, el número resultante es 0 10000011 00000110000000000000000.

c. **(I1 - II/2011)** Se tienen dos números de punto flotante de precisión simple en formato IEEE754: A = 0x3E200000 y B = 0x00000000. ¿Cuál es el resultado, en formato IEEE754, de A : B?

Aquí, la gracia es ver que bajo el estándar IEEE754, la división por el 0 entrega un número infinito. Antes de anotar el resultado, necesitamos saber si el número es positivo, lo que vemos a partir de A:

0x3E200000 = 00111110001000000000000000000000

Como este estándar nos dice que el primer bit indica el signo, sabemos que A es positivo. Finalmente, la representación de un infinito positivo en IEEE754 es:

0x7F800000 = 01111111100000000000000000000000

d. **(Examen - I/2016)** Explique por qué el número $2^{50}+5$ no puede representarse de manera exacta usando el tipo de dato `float` de 32 bits.

Es necesario recordar la composición de un `float` de 32 bits, en el orden dado (de izquierda a derecha):

- **Signo:** 1 bit.
- **Exponente:** 8 bits.
- **Significante:** 23 bits.

Recordando que el exponente está desplazado en 127 unidades, tenemos que 2^{50} se expresa de la siguiente forma:

- **Signo:** 0 (positivo).
- **Exponente:** 10110001 (177).
- **Significante:** 00000000000000000000000.

Ahora, es importante notar que si se le suman 5 unidades al número, estas se deben ver reflejadas en el significante. Esto corresponde a un 1 que se encuentra 50 posiciones a la izquierda (proveniente de 2^{50}), y un 101 al final (proveniente de 5). Como la cantidad de ceros entre el 1 y el 101 supera el número de bits a disposición para el significante, $2^{50}+5$ no se puede representar.

Nota: Esto pasa para todos los números de la forma $2^{24}+x$ en adelante, $2^{23}+x$ se puede representar.

e. **(I1 - I/2017)** Sea P, el conjunto de representaciones de punto flotante de $s + e + 2$ bits, con s bits para el significante normalizado, 1 bit para el signo de este, e bits para el exponente (no desplazado) y 1 bit para el signo de este. Considere además x y $\tilde{x}$, números pertenecientes a $\mathbb{Z}$, ambos codificados usando una representación perteneciente a P, tales que:

- $sucesor(x) \neq x + 1$
- $\forall \tilde{x} < x, sucesor(\tilde{x}) = \tilde{x} + 1$

Donde $sucesor(y)$ es una función que retorna el siguiente número entero mayor que y (el sucesor de y).

En base a esto, responda las siguientes preguntas:

I. Indique cuál es el valor de x en función de los parámetros s y e y muestre que dado x, siempre existe un número $x^{'} \in \mathbb{Z}$, codificado en la misma representación que x, tal que $x^{'} + 1 = x$.
Lo primero que necesitamos para determinar el valor de x es entender sus propiedades. Supongamos que $s = 2$ y $e = 4$, que el bit 0 representa el signo positivo y por ahora obviemos los números negativos. La representación a utilizar tendrá el siguiente orden[1]:

$$se + exp + ss + sig$$

Donde:

- se: Signo exponente.
- exp: Exponente.
- ss: Signo significante.
- sig: Significante.

Entonces, la secuencia de números enteros que podemos generar partiendo de 1 es la siguiente:

$$\begin{array}{rcl} 1 & \rightarrow & 00000000 \\ 2 & \rightarrow & 00001000 \\ 3 & \rightarrow & 00001010 \\ 4 & \rightarrow & 00010000 \\ 5 & \rightarrow & 00010001 \\ 6 & \rightarrow & 00010010 \\ 7 & \rightarrow & 00010011 \\ 8 & \rightarrow & 00011000 \end{array}$$

¿Cómo podemos representar ahora el número 9? Es fácil ver que no se puede, pues el siguiente número, bajo este esquema, será:

$$00010001 \rightarrow (1010)_2 = 10$$

[1] Para no causar confusiones, llamamos a exp el valor del exponente y sig el valor del significante. En cambio, e y s representan el número de bits de estos.

Entonces, tenemos que $sucesor(8) = 10 \neq 9$, por lo que 8 sería el x buscado en esta representación. Se puede ver que este número consiste en el que ya no es posible aumentar su tamaño de forma unitaria, sino que solo a través de saltos en potencias de 2. Esto, ya que nuestro significante no es capaz de representar el bit menos significativo con un 1 para valores mayores en el exponente[2].
De la conclusión anterior se puede ver que el valor de x está sujeto al valor de s. Cuando el valor del exponente supera al número de bits para el significante es que se genera esta propiedad, es decir, $exp > s \rightarrow exp >= s+1$. Como x es el primer número con estas características:

$$x = 2^{s+1} = 0[bin(s+1)]0[zero(s)]$$

Donde:

- $bin(s+1)$: $s+1$ representado en base 2.
- $zero(s)$: s bits iguales a cero.

Ahora, para probar la existencia de $x^{'}$ simplemente usamos la definición del enunciado:

$$\forall \tilde{x} < x, sucesor(\tilde{x}) = \tilde{x} + 1 \rightarrow x - 1 < x, sucesor(x-1) = x - 1 + 1 = x$$

Finalmente: $x - 1 = x^{'} \rightarrow x^{'} + 1 = x$ ■

II. ¿Existe un número con las características de x en el estándar IEEE754 de 32 bits? En caso positivo, indique su valor, y en caso negativo indique por qué no es posible encontrar este número.
Sí, existe. Las diferencias son las siguientes:

(1) Los valores de s y e son fijos: $s = 23$, $e = 8$.
(2) No existe el bit se. Para tener exponentes positivos y negativos, el valor de exp está desfasado en 127.

De esta forma, adaptando la respuesta anterior, el número generado es el siguiente:

$$x_{IEEE754(32b)} = 2^{23+1} = 0[bin(23+1+127)][zero(23)]$$

III. Caracterice, en función de los parámetros s y e, el conjunto de representaciones de punto flotante $\tilde{P} \subset P$, tales que no contienen un número con las características de x. Para caracterizar al conjunto $\tilde{P}$ basta con ajustar el parámetro e en función de s. Esto, de forma que s sea lo suficientemente grande para poder representar siempre al bit menos significativo, sin importar el rango de valores de exp. Como el problema se da para $exp = bin(s+1)$:

$$e \leq \sum_{i=0}^{s-1} 2^i = \frac{1-2^s}{1-2} = \frac{2^s-1}{2-1} = 2^s - 1$$

Notemos que al requerir $exp < bin(s+1)$, lo que debemos hacer es que e sea menor o igual a la suma de todos los bits de s, esto es, $\sum_{i=0}^{s-1} 2^i$. El resultado se desprende de la suma de los primeros n términos de una serie geométrica[3].

[2] Asumimos que, dado el caso donde $exp > s$, los bits generados después de lo que puede representar el significante son todos iguales a 0. Por ejemplo, si $s = 2$ y $e = 3$, tomando $sig = 11$ y $exp = 011$ generamos 1110.

[3] `https://es.wikipedia.org/wiki/Serie_geom%C3%A9trica`