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

IIC2343 – Arquitectura de Computadores

Ayudantía 1 – Solución propuesta

Profesor: Yadran Francisco Eterovic Solano

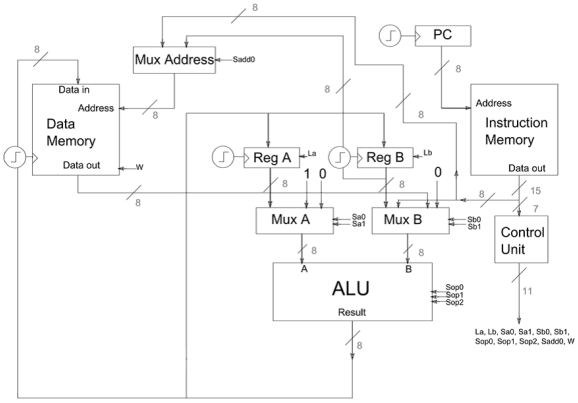
Ayudante: Germán Leandro Contreras Sagredo (glcontreras@uc.cl)

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.

Precalentamiento

1. **(I1 - II/2017)** Considere el siguiente diagrama de bloques del computador básico, aún incompleto.

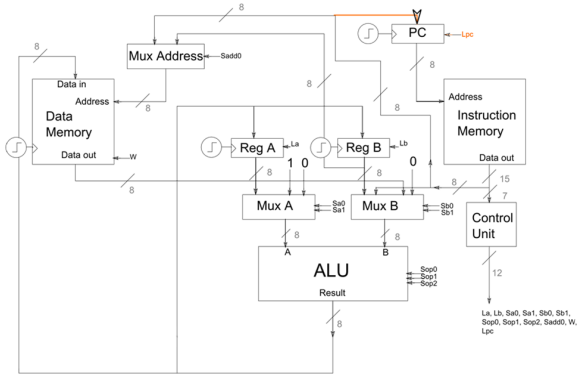


a. Explica el rol del multiplexor *Address*, da un ejemplo de su funcionamiento.

Su rol es poder seleccionar la dirección que se desea utilizar en la memoria de datos (tanto para lectura como escritura). Existen dos posibilidades, la dirección proveniente como literal de la memoria de instrucciones (direccionamiento directo) y la dirección obtenida desde el registro `B` (direccionamiento indirecto). Un ejemplo de esto es el contraste entre las instrucciones `MOV (Dir),A` y `MOV (B),A`. En el primer caso, sin pérdida de generalidad, la unidad de control le asignará el valor 0 a la señal S_{add0} para que el multiplexor `Address` seleccione el bus proveniente de la memoria de instrucciones (que será la dirección de `Dir`), mientras que en el segundo asignará el valor 1 para seleccionar el valor del registro `B`. No obstante, en ambos casos el valor escrito será el contenido en el registro `A`.

b. Explica qué es necesario agregar para permitir instrucciones de tipo salto incondicional; y explica cómo funcionaría en ese caso un salto incondicional.

Como los saltos incondicionales se realizan con la instrucción `JMP label` (siendo `label` la etiqueta de referencia de la instrucción que queremos ejecutar posteriormente), lo que se necesita es poder entregarle al registro `PC` el número de instrucción deseado y cargarlo para poder obtener el *opcode* y el literal correspondientes. Esto es posible transfiriendo la dirección de la instrucción[1] como literal desde la memoria de instrucciones[2] hacia `PC`. Como solo se transfiere una señal no es necesario un multiplexor por el momento, no obstante, sí se requiere una señal que permita decidir cuándo sobreescribir el valor del registro, que llamaremos L_{pc}. El diagrama resultante se muestra a continuación.



Entonces, su funcionamiento es el siguiente:

- Se obtiene el *opcode* y el literal de la instrucción `JMP label`.
- La unidad de control identifica la instrucción y habilita la escritura sobre el registro `PC`, es decir, $L_{pc} = 1$.
- El literal (correspondiente a la línea de instrucción que se desea saltar) llega al registro `PC` y es escrito en el mismo una vez que comienza el flanco de subida.

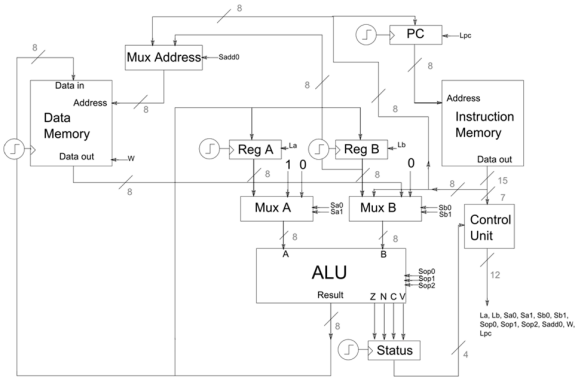
[1] En general, la dirección será equivalente al número de la línea de código del programa.
[2] En `JMP label`, el *Assembler* se encarga de convertir `label` en el literal correspondiente a la línea de la instrucción.

c. Explica qué es necesario agregar a tu respuesta en **b.** para permitir instrucciones de tipo salto condicional; y explica cómo funcionaría en ese caso un salto condicional.

La implementación de saltos condicionales implica dos tipos de instrucción esenciales: `CMP A,x` (comparación entre el valor del registro `A` y `x`) y `JEQ, JNE,` etc. (saltos según condición). En primer lugar, el objetivo de la instrucción `CMP` es poder obtener el resultado de $A - x$ sin tener que cambiar los valores de los registros, lo que en el siguiente ciclo nos ayudará a obtener información sobre la operación. Este resultado se interpreta a partir de *flags*, las que no son más que señales de un bit que nos permiten definir qué pasó con el resultado. Se suelen utilizar cuatro:

- **Z:** Indica si $A - x = 0$ o no.
- **N:** Indica si $A - x < 0$ o no.
- **C:** Indica si hubo *carry* o no.
- **O:** Indica si hubo *overflow* o no.

Por ejemplo, si queremos ejecutar la instrucción `JEQ label`, queremos ver si se activó la señal **Z** (ya que eso indicaría que la diferencia entre ambos es 0, siendo iguales). Para poder lograr esto entonces, primero se necesita añadir un registro `STATUS` que esté conectado al *clock* del computador básico y que reciba las *flags* provenientes de la `ALU`. Esto permitirá coordinar el estado del ciclo **anterior** con la unidad de control y la instrucción que esté leyendo en su momento, de forma que pueda decidir si habilitar o no la señal L_{pc} para permitir o no la escritura sobre el registro `PC`. El diagrama, entonces, se ve como sigue:



Finalmente, su funcionamiento es el siguiente:

- En el primer ciclo, se obtiene el *opcode* de la instrucción `CMP`, realizando la operación de resta en la `ALU` con el objetivo de actualizar las *flags* en `STATUS`.
- En el segundo ciclo, se obtiene el *opcode* y el literal de la instrucción de salto condicional.
- La unidad de control identifica la instrucción y habilita la escritura sobre el registro `PC` si, y solo si se cumplen las condiciones dadas por las *flags* provenientes de `STATUS`.
- Si se cumple la condición, el literal (correspondiente a la línea de instrucción a la que se desea saltar) llega al registro `PC` y es escrito en el mismo una vez que comienza el flanco de subida. En caso contrario, L_{pc} será igual a 0 y simplemente se obtendrá el *opcode* de la instrucción siguiente en el programa.

2. Considere el siguiente programa:

```
x = 2
y = 4
z = 0 # Variable auxiliar
z = x
x = x + y
y = y - z
```

En base a este:

a. Construya un programa en `Assembly` que obtenga el mismo resultado (considere que x, y y z parten con sus valores almacenados en memoria).

```
DATA:
    x 2
    y 4
    z 0
CODE:
    MOV A,(x) ;Guardamos x en A
    MOV (z),A ;Guardamos A en z, variable auxiliar
    MOV B,(y) ;Guardamos y en B
    ADD A,B ;Guardamos en A x+y
    MOV (x),A ;Guardamos en x el resultado de la suma
    MOV A,(y) ;Guardamos y en A
    MOV B,(z) ;Guardamos z en B
    SUB A,B ;Guardamos en A y-z
    MOV (y),A ;Guardamos en y el resultado de la resta
```

Aquí es importante notar que no se usó la operación `SUB B,A` debido a que esta resulta en $B = A - B$, lo que no corresponde a lo que se buscaba en el programa original.

b. Ahora, programe en `Assembly` un código que obtenga el mismo resultado de x e y, pero sin hacer uso de la variable z en el segmento `DATA`.

```
DATA:
    x 2
    y 4
CODE:
    MOV A,(x) ;Guardamos x en A
    MOV B,(y) ;Guardamos en B el valor de y
    ADD A,B ;Guardamos en A x+y
    MOV B,(x) ;Guardamos x en B
    MOV (x),A ;Guardamos en x el resultado de la suma
    MOV A,(y) ;Guardamos y en A
    SUB A,B ;Guardamos en A y-z
    MOV (y),A ;Guardamos en y el resultado de la resta
```

Así, se puede ver que una limitante conllevó a un programa con una instrucción menos.

c. A partir de los programas anteriores, explique el flujo resultante de cada uno de ellos en el diagrama del computador en **1**.

Antes de detallar el flujo, veremos cómo se almacena cada variable en la memoria de datos:

1) `MOV A,2`: Almacena en el registro `A` el valor de la variable x.
2) `MOV (0),A`: Almacena en la primera dirección de la memoria de datos el valor del registro `A` (correspondiente a x).
3) `MOV A,4`: Almacena en el registro `A` el valor de la variable y.
4) `MOV (1),A`: Almacena en segunda dirección de la memoria de datos el valor del registro `A` (correspondiente a y).
5) `MOV A,0`: Almacena en el registro `A` el valor de la variable z.
6) `MOV (2),A`: Almacena en la primera dirección de la memoria de datos el valor del registro `A` (correspondiente a z).

Es decir, por cada variable declarada se ejecutan dos instrucciones que permiten almacenar en la memoria de datos sus valores. Por lo general, el *Assembler* parte en la primera dirección y sigue de forma sucesiva para el resto de las variables declaradas[3].

Siguiendo con el código:

1) `MOV A,(x)`: El literal de la ROM (correspondiente a la dirección de x) es escogido por el multiplexor `Address` para obtener su valor y se habilita la carga en el registro `A` para que el número sea almacenado en él.
2) `MOV (z),A`: El literal de la ROM (correspondiente a la dirección de z) es escogido por el multiplexor `Address` y se habilita la señal W para poder escribir en la RAM. Luego, se configura la `ALU` de forma que el resultado sea el valor del registro `A`, escribiendo ese número en la dirección z.
3) `MOV B,(y)`: El literal de la ROM (correspondiente a la dirección de z) es escogido por el multiplexor `Address` para obtener su valor y se habilita la carga en el registro `B` para que el número sea almacenado en él.
4) `ADD A,B`: Se configuran los multiplexores de `A` y `B` para escoger los valores de sus registros. Luego, se configura la `ALU` para obtener la suma de ambos valores y, finalmente, se habilita la escritura en el registro `A` para que almacene el resultado.
5) `MOV (x),A`: El literal de la ROM (correspondiente a la dirección de x) es escogido por el multiplexor `Address` y se habilita la señal W para poder escribir en la RAM. Luego, se configura la `ALU` de forma que el resultado sea el valor del registro `A`, escribiendo ese número en la dirección x (resultado de $x + y$).
6) `MOV A,(y)`: El literal de la ROM (correspondiente a la dirección de y) es escogido por el multiplexor `Address` para obtener su valor y se habilita la carga en el registro `A` para que el número sea almacenado en él.
7) `MOV B,(z)`: El literal de la ROM (correspondiente a la dirección de z) es escogido por el multiplexor `Address` para obtener su valor y se habilita la carga en el registro `B` para que el número sea almacenado en él.
8) `SUB A,B`: Se configuran los multiplexores de `A` y `B` para escoger los valores de sus registros. Luego, se configura la `ALU` para obtener la suma de ambos valores y, finalmente, se habilita la escritura en el registro `A` para que almacene el resultado.

[3]Esto es muy útil al declarar arreglos.

9) `MOV (y),A`: El literal de la ROM (correspondiente a la dirección de y) es escogido por el multiplexor `Address` y se habilita la señal `W` para poder escribir en la RAM. Luego, se configura la `ALU` de forma que el resultado sea el valor del registro `A`, escribiendo ese número en la dirección y (resultado de $y - z$).

Se obvia el flujo del segundo programa, dado que es una versión simplificada del primero sin mayores cambios en las instrucciones utilizadas.

3. **(I1 - I/2016)** ¿En qué casos es posible soportar la instrucciones `ADD B,Lit` en el computador básico, sin modificar su hardware ni sobrescribir datos? Para los casos negativos, indique qué modificaciones al hardware y/o assembly se deberían hacer para soportarla.

   Si dentro del computador básico revisamos los componentes Mux `A` y Mux `B`, podemos ver que los únicos literales que podrían ser seleccionados para ser almacenados en `A` son 0 y 1. Por lo tanto, sin modificar el computador básico, podemos soportar `ADD B,0`[4] y `ADD B,1` (que existe y llamamos `INC B`). Si quisiéramos habilitar la instrucción `ADD B,Lit` para cualquier literal, sería necesario entonces tener una conexión entre el Mux `A` y la ROM (para así poder recibir el literal a utilizar, al igual como está incluido en `B`) y definir la combinación S_{a0}, S_{a1} para escogerlo y ajustar las señales de control para esta nueva instrucción a partir de un nuevo *opcode*.

4. **(Examen - I/2017)** Modifique la arquitectura del computador básico para que el registro `STATUS` se actualice solo después de la ejecución de una instrucción `CMP`.

   La modificación más simple que se puede hacer para lograr el objetivo consiste en crear una nueva señal (llamémosla L_{stat}). L_{stat} será la señal que habilita la carga de datos en el registro `STATUS`. Finalmente, basta con que la Unidad de Control se encargue de transmitir $L_{stat} = 0$ para todo *opcode* que no corresponda a la instrucción `CMP`.

[4] Esto claramente no tiene mucho sentido.

5. **(Apuntes - Saltos y subrutinas)** ¿Cómo se podría implementar en el computador básico la opción de que este avise luego de realizar una operación cuando el resultado es par o impar?

   Esto se puede hacer de forma muy sencilla si nos damos cuenta del siguiente hecho: Todo número par en representación binaria termina en 0 y todo número impar termina en 1. Esto, ya que todo bit representa una potencia de 2, salvo el último que representa un 1. Por lo tanto, bastaría con añadir a la salida de la `ALU` una señal P equivalente al último bit del resultado (o su negación) y que fuera almacenada dentro del registro Status de forma directa. Solo faltaría una nueva instrucción asociada a un nuevo opcode (que podríamos llamar `JEN` -*Jump Even Number*-, por ejemplo).

6. ¿Cuántos ciclos toma llamar una subrutina? ¿y cuántos toma retornarla? Justifique.

   Para llamar a una subrutina, se siguen los siguientes pasos:

   a) Guardar `PC+1` en la posición actual de `SP`.

   b) Decrementar en 1 `SP`.

   c) Guardar la dirección de la subrutina en `PC`.

   En cambio, para retornarla, se siguen estos:

   a) Incrementar en 1 `SP`.

   b) Guardar el valor de memoria por `SP` incrementado en `PC`.

   Si revisamos el computador básico, podemos ver que el llamado a la subrutina toma solo un ciclo, ya que:

   a) `PC+1` ya se encuentra esperando en el Mux DataIn, espera a que la unidad de control habilite su paso a la RAM y, en el flanco de subida, se almacena en la dirección `SP`.

   b) Se habilita la señal $DecSp$ en el registro `SP` y, llegado el flanco de subida, decrece correspondientemente.

   c) La dirección de la subrutina ya se encuentra en el Mux `PC`. Se habilita su paso a partir de Spc, y llegado el flanco de subida, se almacena en el registro `PC`.

   Es decir, los 3 cambios se realizan simultáneamente dentro de un solo flanco de subida, por lo que un ciclo es más que suficiente.

   Ahora, veamos el retorno:

   a) Se habilita la señal $IncSP$ en el registro `SP` y, llegado el flanco de subida, decrece correspondientemente.

   b) Se guarda en el `PC` lo almacenado en `SP+1` (que sería la siguiente línea de código desde la que se saltó). Sin embargo, `SP` lo cambiamos recientemente, por lo que el valor final llega a `PC` una vez que acaba el flanco de subida y debe esperar al siguiente flanco para habilitar su escritura en el registro.

   Por ende, el retorno es imposible realizarlo en un ciclo, son necesarios dos para el funcionamiento correcto de la instrucción.

Preguntas

1. a. **(I1 - I/2016)** ¿Cuál es la frecuencia máxima que puede tener el clock del computador básico? ¿Qué pasa si un clock con una frecuencia mayor a la máxima es conectado al computador básico?

   La frecuencia máxima del clock **debe** ser el tiempo que toma ejecutar la instrucción más lenta del listado. ¿Por qué? Porque en caso contrario, esta instrucción no alcanzaría a ejercutarse por completo y, por ejemplo, no se alcanzarían a almacenar los nuevos valores de los registros en el flanco de subida, causando errores de consistencia con las instrucciones posteriores.

   b. **(I1 - I/2016)** Modifique el computador básico para dar soporte a la instrucción `CLEAR`, que setea en 0 el valor de todos los registros (excepto `PC` y `STATUS`) y el de todas las palabras de la memoria de datos. Indique la(s) señal(es) de control de la nueva instrucción.

   Para los registros `A` y `B`, si se ejecuta la instrucción `CLEAR` (con *opcode* diferente al resto de las instrucciones), habilitamos la escritura en ellos ($L_a = L_b = 0$) y hacemos que el literal de la memoria de instrucciones (igual a cero) sea el resultante de la `ALU` (lo que se puede hacer escogiendo el 0 en el Mux `A` y habilitando la suma). Para la memoria de datos y el registro `SP` se requiere un trato distinto. Como para almacenar el valor 0 en los registros `A` y `B` el literal fue el valor resultante de la `ALU`, llegará al bus de entrada de la memoria de datos. Para poder escribir en esta, entonces, necesitamos habilitar la escritura en todos los registros de la RAM[5]. Como el bus *Address* en conjunto con la señal W permiten la escritura sobre **un solo** registro, definiremos una nueva señal Clr de forma que ingrese directamente a la memoria de datos y se conecte con cada salida del demultiplexor[6] interno a través de una componente `OR`, antes de llegar a la entrada L de cada entrada de la RAM. Así, si $Clr = 1$ (solo en la instrucción `CLEAR`), se terminará de habilitar la carga de todos los registros de la memoria, sobreescribiendo en ellos el valor 0. Finalmente, para el registro `SP` una opción sería usar una señal L_{sp} que permita la escritura en dicho registro y conectar el resultado de la `ALU` en su entrada, permitiendo su modificación al igual que en los registros `A` y `B`.

   c. **(Propuesto) (Examen - II/2016)** Modifique la arquitectura del computador básico para que funcione con lógica ternaria en vez de binaria. Más específicamente, modifique los tamaños de los elementos (buses, registros, señales de control, etc.) de modo que el nuevo computador tenga una capacidad similar a la versión binaria. Asuma que existen todos los componentes vistos en clases en versión ternaria. **Nota:** No es válido utilizar los valores ternarios como si fueran binarios.

   Aquí, lo relevante es ver cómo cambia la representación de datos que se tenía antes. Con un trit, podemos representar un número más que con un bit, con dos trits, podemos representar 4 números más que con dos bits, y así sucesivamente. Por lo tanto, iremos estudiando cómo cambiaría cada parte del computador básico:

   - **Registros y buses de datos:** Estos almacenaban 8 bits. Para poder abarcarlos todos, resolvemos el siguiente problema para K trits y N bits:

[5] Pensaremos en la memoria de datos como un conjunto de registros. Detalles de esto se verán más adelante.

[6] Componente que, a diferencia del multiplexor, dada una señal o bus de entrada y uno de selección S transfiere el número recibido a la salida seleccionada, enviando un 0 al resto. Más información aquí.

$$\begin{aligned} 3^K &= 2^N \\ \log_3(3^K) &= \log_3(2^N) \\ K &= N * \log_3(2) \\ K &= \lceil N * \log_3(2) \rceil \end{aligned}$$

Notar que como K debe ser un número entero, usamos la función techo para aproximar al entero mayor (si usáramos la función piso, podríamos perder números representables). Por lo tanto, para $N = 8$ bits, tenemos que $K = \lceil 5{,}047 \rceil = 6$.
Otra forma de obtener el resultado es simplemente ver el intervalo de representación con N bits[7]:

$$-\frac{2^N}{2} \leq x \leq \frac{2^N}{2} - 1$$

[7]Se verá en detalle en ejercicios futuros.

Y tomando el intervalo para los trits:

$$-\lfloor\frac{3^K}{2}\rfloor \leq x \leq \lfloor\frac{3^K}{2}\rfloor$$

Buscamos el K tal que el intervalo de representación de bits (con $N = 8$) sea considerado **por completo**. (Notar que nos importa representar todo, da lo mismo si nuestro poder de representación aumenta, no queremos que disminuya). De esta forma, también se llega a $K = 6$.

Es importante ver que esto no lo cambiamos para el registro `STATUS`, ya que aquí nos interesa ver si cada *flag* cumple o no para generar los saltos (utilizar base trinaria dificultaría un poco el funcionamiento de este).

- **Señales de control:** Algunas de las señales de control se pueden abreviar de la siguiente forma:
  - Selector de operaciones de la `ALU`: Aquí tenemos 8 operaciones que se seleccionan a partir de 3 bits. Sin embargo, bastan 2 trits para tener 8 combinaciones diferentes, y así, escoger la operación correspondiente.
  - Incrementar/decrementar `SP`: Ahora, podemos usar un trit para las siguientes combinaciones: 0 (no incrementar ni decrementar), 1 (incrementar) y 2 (decrementar).
  - Selector del Mux `A` y Mux `Address`: Como escogemos entre tres señales para estos dos multiplexores, podemos quedarnos con un solo trit para obtener las 3 combinaciones. Notar que esto no sirve para el `Mux B`, ya que se selecciona entre 4 entradas (por lo que se siguen necesitando dos trits).

Estos corresponden a los cambios aplicables para mantener el funcionamiento de nuestro computador, utilizando trits en vez de bits.

2. a. **(I1 - I/2016)** ¿Qué pasaría si se quita el registro `STATUS` del computador básico y se conectaran directamente las señales `ZNCV` a la unidad de control?

   Al no estar conectadas a la señal clock, estas señales perderán la sincronización con el flanco de subida. Esto implica que las señales que estén ingresadas en la unidad de control no necesariamente sean correspondientes a la última instrucción ejecutada, causando posibles errores en las instrucciones de salto (y por ende, en todo el programa).

   b. **(I1 - I/2017)** Si se elimina la instrucción `CMP` del computador básico, ¿cómo deben modificarse las instrucciones de salto, sin alterar el hardware, para que estas no dependan del resultado de la última instrucción ejecutada? Escriba detalladamente todas las modificaciones necesarias y sus implicancias. Asuma que solo es necesario resolver el caso de la comparación de los registros A y B y que no es posible sobreescribir los registros para realizar la comparación.

   Como no queremos que los saltos dependan de la instrucción anterior (esto es, que no se basen en cómo haya quedado el registro `STATUS` después de ejecutarlas), lo que se necesita hacer es añadir más operaciones a las instrucciones de salto. En este caso, lo que se debe hacer es replicar la instrucción `CMP` dentro de la ejecución que realizan los saltos. De esta forma, cada instrucción de salto contendrá dos opcodes:

   1) El primero corresponderá al mismo que solía tener `CMP`: Se realiza la resta entre los registros A y B sin guardar el resultado para poder actualizar las flags del registro `STATUS`.

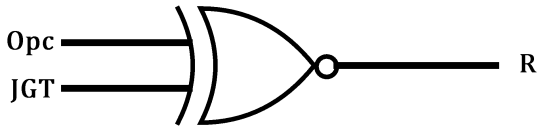
2) El segundo será el que tenía cada salto originalmente.

De esta forma, el cambio sustancial que se genera es que las instrucciones de salto toman **dos ciclos** en vez de uno.

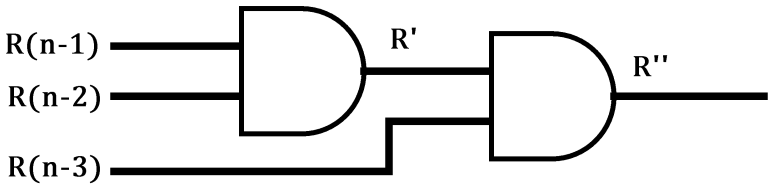
c. **(Propuesto)** Diagrame y explique una posible implementación de la instrucción `JGT Dir` (opcode 1000000) al interior de la unidad de control.

Se detallará por parte lo que debe suceder dentro de la unidad de control:

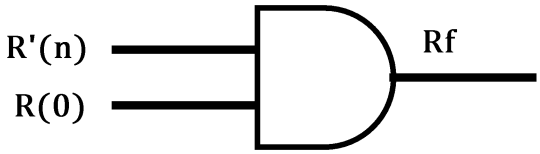
1) **Comprobar que el opcode es el correcto:** Para hacer esto, usamos los siguientes circuitos:



Primero, nos aseguramos de que la señal que recibe de la ROM (*Opc*) coincida con la señal para el comando `JGT` (nombrado de la misma forma).
Para verificar que sea solo una cadena de 1's, usamos el siguiente circuito:

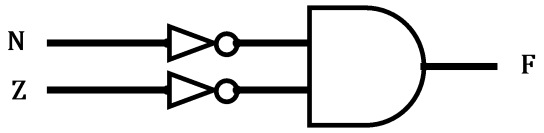


Hacemos esto de forma sucesiva hasta llegar al último bit de R (el menos representativo):



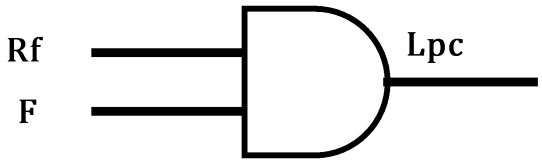
Si nuestra señal Rf es igual a 1, entonces los números coinciden.

2) **Comprobar que las condiciones de salto se cumplen:** Notemos que para realizar el salto, se debe cumplir que las *flags* N y Z deben ser iguales a 0, por lo tanto, usamos el siguiente circuito:



Así, si F = 1, entonces se cumple que el número es mayor exclusivo (ya que la diferencia no es 0, y tampoco es negativa).

3) **Cumplimiento simultáneo:** Finalmente, la señal más importante es L_{pc} (pues esta define si el salto se hace o no), por lo que la definimos como sigue:



Nota: Si tienen dudas sobre qué pasaría si en realidad es otra instrucción de salto la que se quiere ejecutar (`JGE`, por ejemplo), hacemos que todos los posibles L_{pc} se conecten a partir de un `OR` para ver si se hace el salto o no.

d. **(Propuesto)** A partir del siguiente código Assembly, explique el flujo resultante dentro del computador básico, indicando el valor final de los registros A y B:

```
DATA:
   r 3 ;Resultado final
CODE:
  MOV A,(r)
  MOV B,2
  PUSH B
  CALL func
  POP B
  JMP finish
  func:
    shift:
      MOV A,B
      CMP A,0
      JEQ end
      DEC A
      MOV B,A
      MOV A,(r)
      SHL A,A
      MOV (r),A
      JMP shift
    end:
      RET
  finish: ;Termina el programa
```

1) `r 3:` Aquí se declara que el label r (una dirección de memoria) almacenará el literal 3. Al ser la primera (y única), se guarda por *default* en la dirección 0. Entonces, esto es equivalente a las instrucciones `MOV A,3` y `MOV (0),A`. Esto implica que, en primer lugar, se escribe el literal 3 en el registro `A` (habilitando su escritura previamente) y en el siguiente ciclo se habilita la escritura en la RAM y se escribe el valor del registro `A` en la dirección 0.
2) `MOV A,(r):` Se obtiene el dato de memoria en la dirección de r (que ya sabemos que es 0). Este dato se extrae de la RAM, lo seleccionamos para que pase por Mux B y permitimos la carga en el registro A para el resultado (que será la operación suma de la ALU, con A = 0 y B = Mem[0]). Entonces, queda A = Mem[0] = 3.
3) `PUSH B:` Se almacena en la RAM el valor del registro B, en la posición SP (correspondiente a la última dirección). El valor de SP decrece (para almacenar el próximo valor).
4) `CALL func:` Guarda la posición PC+1 en la dirección de memoria SP, esta señal decrece (para almacenar el siguiente valor), y se carga en PC el número correspondiente a la línea de código donde se encuentra la subrutina func (para ejecutar desde ahí).
5) `MOV A,B:` Se guarda en el registro A lo que está almacenado en B (en un comienzo, el valor igual a 2).

6) `CMP A,0`: Se realiza la operación A-0 (en este caso, 2-0) y se guardan las flags en el registro Status para el próximo comando a ejecutar.
7) `JEQ end`: Desde la unidad de control se revisa si la flag Z es igual a 0. Como esto no pasa, no se habilita la señal Lpc y el PC aumenta en uno para proceder con la siguiente línea de código.
8) `DEC A`: Equivalente a SUB A,1. El registro almacenado en A decrece en una unidad (en este caso entonces, queda A = 1).
9) `MOV B,A`: Se guarda en el registro B el valor almacenado en A (por lo que queda B = 1).
10) `MOV A,(r)`: Se obtiene el dato de memoria en la dirección de r = 0. Este dato se extrae de la RAM, lo seleccionamos para que pase por Mux B, y permitimos la carga en el registro A para el resultado (que será la operación suma de la ALU, con A = 0 y B = Mem[0]). Entonces, queda A = Mem[0] = 3.
11) `SHL A,A`: Almacenamos en A el resultado de hacerle un `shift left` al número. En este caso, al ser equivalente a una multiplicación por 2, queda A = 6.
12) `MOV (r),A`: Se habilita la escritura en la RAM y en la dirección r = 0 se almacena el valor del registro A (entonces, Mem[0] = 6).
13) `JMP shift`: Se realiza un salto incondicional a la label shift (es decir, Lpc = 1, y se carga la línea de código correspondiente para la próxima ejecución).
14) `MOV A,B`: Se guarda en el registro A lo que está almacenado en B (ahora, un valor igual a 1).
15) `CMP A,0`: Se realiza la operación A-0 (en este caso, 1-0) y se guardan las flags en el registro Status para el próximo comando a ejecutar.
16) `JEQ end`: Desde la unidad de control se revisa si la flag Z es igual a 0. Como esto no pasa, no se habilita la señal Lpc y el PC aumenta en uno para proceder con la siguiente línea de código.
17) `DEC A`: Equivalente a SUB A,1. El registro almacenado en A decrece en una unidad (en este caso entonces, queda A = 0).
18) `MOV B,A`: Se guarda en el registro B el valor almacenado en A (por lo que queda B = 0).
19) `MOV A,(r)`: Se obtiene el dato de memoria en la dirección de r = 0. Este dato se extrae de la RAM, lo seleccionamos para que pase por Mux B y permitimos la carga en el registro A para el resultado (que será la operación suma de la ALU, con A = 0 y B = Mem[0]). Entonces, queda A = Mem[0] = 6.
20) `SHL A,A`: Almacenamos en A el resultado de hacerle un `shift left` al número. En este caso, al ser equivalente a una multiplicación por 2, queda A = 12.
21) `MOV (r),A`: Se habilita la escritura en la RAM y en la dirección r = 0 se almacena el valor del registro A (entonces, Mem[0] = 12).
22) `JMP shift`: Se realiza un salto incondicional a la label shift (es decir, Lpc = 1 y se carga la línea de código correspondiente).
23) `MOV A,B`: Se guarda en el registro A lo que está almacenado en B (ahora, un valor igual a 0).
24) `CMP A,0`: Se realiza la operación A-0 (en este caso, 0-0) y se guardan las flags en el registro Status para el próximo comando a ejecutar.

25) `JEQ end:` Desde la unidad de control se revisa si la flag Z es igual a 0. Como esto pasa, se habilita la señal Lpc y el registro PC carga la dirección de la línea de código del label end.

26) `RET:` SP aumenta en 1 y se carga en el registro PC lo almacenado en Mem[SP] (en este caso, la línea de código correspondiente a la siguiente de donde se hizo el llamado a la función).

27) `POP B:` SP aumenta en 1 y se carga en el registro B lo almacenado en Mem[SP] (en este caso, el valor original del registro B antes de ingresar a la función, B = 2).

28) `JMP finish:` Se realiza un salto incondicional a la label finish (es decir, Lpc = 1, y se carga la línea de código correspondiente). Como aquí ya no hay más código, es la última ejecución a realizar (la ROM suele tener después solo líneas de código que hacen que no se ejecute nada dentro del computador, manteniendo el estado final).

Finalmente, podemos ver que los valores finales de nuestros registros son A = 0 y B = 2. Notemos que el valor de A **no es** 12, ya que este cambió dentro de la rutina, y no terminamos por almacenarle el resultado escrito en memoria. Sí sabemos que en la memoria lo calculado se mantuvo (Mem[0] = 12).