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

IIC2343 – Arquitectura de Computadores

Ayudantía 2 – 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. **(I1 - I/2018)** Construya un circuito que obtenga la siguiente tabla de verdad, pero solo usando compuertas `OR` y `NOT`:

`R`	`S`	`Q(t+1)`	$\overline{\texttt{Q(t+1)}}$
0	0	-	-
1	0	0	1
0	1	1	0
1	1	`Q(t)`	$\overline{\texttt{Q(t)}}$

Cuadro 1: Tabla de verdad del *latch* RS.

Un circuito que cumple lo pedido es el siguiente.

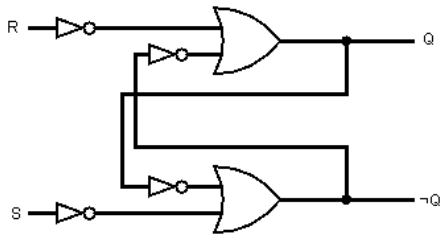

Figura 1: *Latch* RS con compuertas `OR` y `NOT`.

Este se deduce a partir de la **Ley de Morgan**. En el *latch* RS tradicional, las componentes siguen las siguientes fórmulas de lógica proposicional:

- `NOT(R AND` $\overline{\texttt{Q}}$`)`
- `NOT(S AND Q)`

Al desarrollar ambos términos con la Ley de Morgan, se obtiene lo siguiente:

- $\overline{\texttt{R}}$ `OR Q`
- $\overline{\texttt{S}}$ `OR` $\overline{\texttt{Q}}$

Que es lo que finalmente representa el circuito anterior.

b. **(I1 - I/2013)** Diseñe, utilizando todos los elementos de circuitos lógicos vistos en clases, un contador secuencial de 2 bits que se incrementa con cada flanco de subida de la señal de control.

Antes de presentar la solución, hay que ver intuitivamente lo que se necesita. Como el contador es de dos bits, lo que se busca replicar es la siguiente secuencia: 00, 01, 10, 11, 00. De aquí se aprecian las siguientes tendencias:

- El bit más significativo cambia su valor cada dos saltos y está sujeto al flanco de bajada (*i.e.*, un cambio de 1 a 0) del bit menos significativo.
- El bit menos significativo, independiente del otro bit, va alternando su valor constantemente.

De esta forma, un contador que cumple los comportamientos antes señalados es el siguiente:

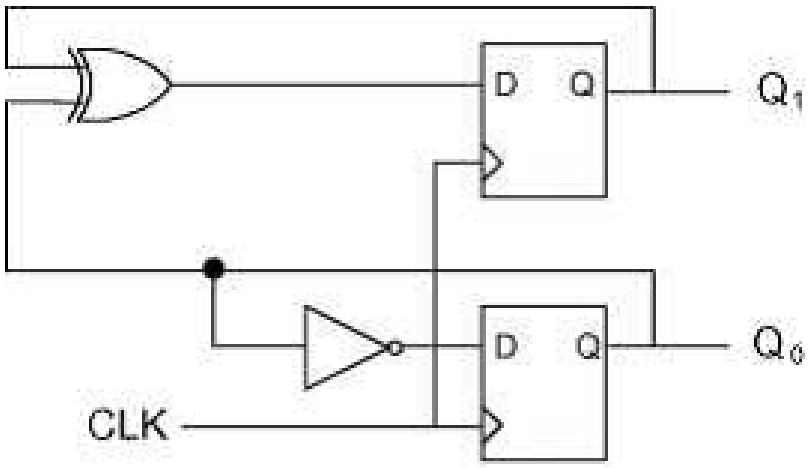

Figura 2: Contador secuencial de dos bits.

En este caso, el *flip-flop* inferior representa al bit menos significativo, mientras que el superior representa al más significativo. Se puede ver entonces que:

1) El *flip-flop* inferior recibe siempre como dato la negación del estado que poseía en la iteración previa, por lo que irá alternando su valor constantemente.
2) El *flip-flop* superior cambia su valor en dos casos:
   - Asumiendo que parte con un cero almacenado, se ve que de la compuerta `XOR` retorna un 1 una vez que el *flip-flop* inferior cambia su estado a 1. No obstante lo anterior, el *flip-flop* superior no cambia su valor hasta el próximo flanco de subida, lo que produce la secuencia 01-10.
   - Ya con un 1 almacenado, se ve que la compuerta `XOR` retorna un 0 cuando, nuevamente, el *flip-flop* inferior cambia su estado a 1[1]. De esta forma, en el siguiente flanco de subida el *flip-flop* superior cambia su estado a 0, generando ahora la secuencia 11,00.

   El uso de la compuerta `XOR` se hace evidente si vemos la siguiente tabla de verdad, siendo $\mathtt{A}_1, \mathtt{A}_0$ los bits más y menos significativos del contador, respectivamente:

$\mathtt{A}_1^t$	$\mathtt{A}_0^t$	$\mathtt{A}_1^{t+1}$
0	0	0
0	1	1
1	0	1
1	1	0

Cuadro 2: Tabla de verdad para determinar el siguiente valor de $\mathtt{A}_1$.

También es válido considerar para este contador la implementación de una secuencia de dos *flip-flops* para el bit más significativo, siendo el *clock* del poseedor del estado el bit la salida del primer *flip-flop*. De esta forma, este bit se actualizaría cada dos ciclos, generando el comportamiento correcto del contador de dos bits.

[1] Recordar que `1 XOR 1 = 0`

c. **(I1 - II/2012)** Implemente mediante compuertas lógicas, elementos de control y *flip-flops*, una memoria RAM de 16 palabras de 1 byte.

Ya conociendo mejor la estructura de los registros, los Mux y Demux, se puede crear la siguiente estructura para una RAM:

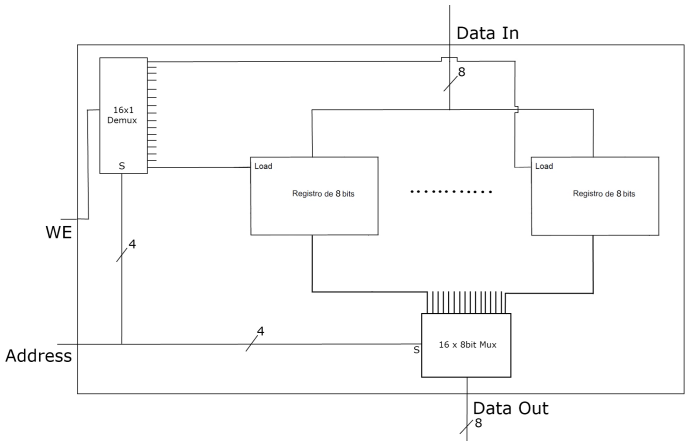

Figura 3: RAM de 16 palabras de 1 byte.

Se explica por parte cada una de sus componentes:

- **Señal *Data In*:** Corresponde al bus de un byte que busca ser almacenado en la RAM.
- **Señal *Address*:** Corresponde a la señal que determina la ubicación de un registro dentro de la RAM. Como la RAM posee 16 palabras de un byte, se necesitan 16 direcciones, *i.e.* 4 bits para poder ubicar un registro.
- **Señal WE:** Consiste en la señal que habilita la escritura en los registros de la RAM.
- **Señal Data Out:** Si `WE = 0`, corresponde a la palabra almacenada en la dirección indicada por *Address*. En cambio, si `WE = 1`, será la misma palabra que se acaba de almacenar (*i.e. Data In*).
- **Registro de 8 bits:** Corresponden a los registros de memoria de la RAM. Como se solicita en la pregunta, hay 16 en total.
- **16x1 bit Demux:** Consiste en un Demux de 16 salidas de un bit. Este se encarga de propagar la señal `WE` **solo** al registro del que se requiere su acceso (ya sea por lectura o escritura). La selección de la salida se hace a partir de la señal *Address*.
- **16x8 bit Mux:** Consiste en un Mux de 16 entradas de un byte (8 bits). A partir de la señal *Address*, retorna a través del bus *Data Out* el valor del registro seleccionado[2].

[2]Por esta razón, si `WE = 1`, el Mux selecciona el valor del registro que acaba de ser sobreescrito, explicando la coincidencia de los valores *Data In* y *Data Out*.

d. **(I1 - I/2018)** Diseñe un registro de 1 byte haciendo uso de *flip-flops* D y una señal de control `L` que habilite la sobreescritura del estado Q solo cuando `L = 1`. Luego, explique cómo puede modificar el circuito realizado para acceder individualmente (tanto para lectura como escritura) a cada uno de los bits del componente.

En primer lugar, se muestra el diagrama para un solo bit del registro, en particular, el bit menos significativo.

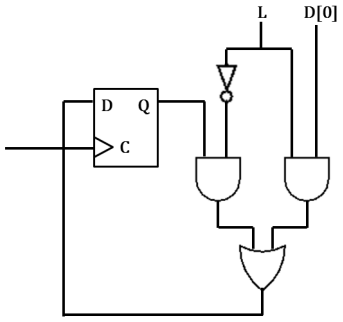

Figura 4: Diagrama del almacenamiento del bit menos significativo del registro.

En este caso, `D[0]` representa el bit menos significativo que se busca almacenar, mientras que `L` es la señal de carga. Así, se obtiene la siguiente tabla de verdad.

L	D[0]	Q
0	0	Q
0	1	Q
1	0	0
1	1	1

Cuadro 3: Tabla de verdad del almacenamiento del bit menos significativo del registro.

De esta forma, se evidencia el cumplimiento del primer objetivo: mantener el estado de un bit para `L = 0`. Para formar el byte, consiguientemente, basta por realizar este circuito para todos los bits del registro, conectados con todos los bits de `D`.

Para extender la lectura y la escritura a bits individuales del registro, existen muchas posibilidades, aquí se muestra una. En primer lugar, definimos un bus de control llamado S_{bit} que permita seleccionar el bit al que se desea acceder. Aquí, se debe cumplir que:

$$|S_{bit}| = \lceil \log_2(\text{Número de bits en el registro}) \rceil$$

Esto permitirá poder seleccionar todos los bits del registro. En este caso, se cumple que $|S_{bit}| = 3$.

Ahora, queremos que la señal `L` llegue **solo** al bit de interés. Para este propósito, podemos usar un Demux con entrada `L` y con bus de control igual a S_{bit}.

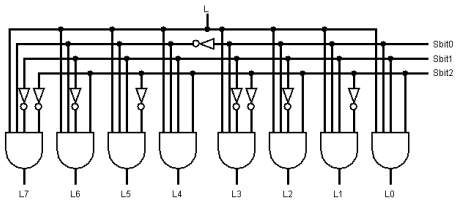

Figura 5: Demux con bus de control de 3 bits y señal de entrada `L`.

De esta forma, solo la señal $\texttt{L}_i$ va a ser igual a 1, habilitando la escritura del bit i del registro (sin importar el valor del resto de los bits que se encuentran en el bus de entrada).

Por último, necesitamos que el bus de salida solo posea el bit de interés. Para este propósito, usamos un Mux que tenga como entradas todos los bits del registro y que haga uso del mismo bus de control S_{bit}.

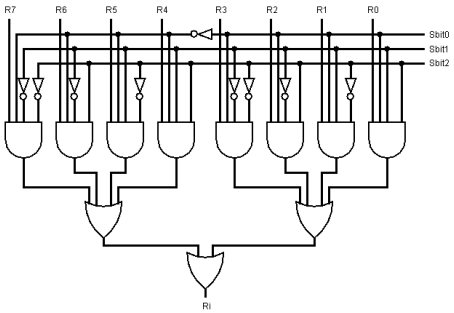

Figura 6: Mux con bus de control de 3 bits y bus de entrada `R` (valor del registro).

Finalmente, conectamos el bit resultante con un bus de 7 bits iguales a cero, de forma que el bus de salida del registro posea 8 bits (para respetar el formato del registro), siendo el bit menos significativo igual al seleccionado en un principio.

2. 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 el inverso multiplicativo del 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. Diseñe un circuito que le permita a la unidad de control del computador básico identificar un *opcode*, retornando una señal de 1 bit que indique si corresponde a la instrucción esperada o no.

   Supongamos que `Op` es el *opcode* recibido e `Ins` la instrucción esperada. Para cada bit i se espera una tabla de verdad como sigue:

`Op[i]`	`Ins[i]`	**Resultado**
0	0	1
0	1	0
1	0	0
1	1	1

Cuadro 4: Tabla de verdad esperada para la identificación de una instrucción.

De aquí se puede deducir que la compuerta que nos sirve es la negación de `XOR`, `XNOR`. Siguiente a eso, queremos verificar que todos los bits de la secuencia sean iguales a 1. El siguiente circuito, entonces, cumple con lo pedido:

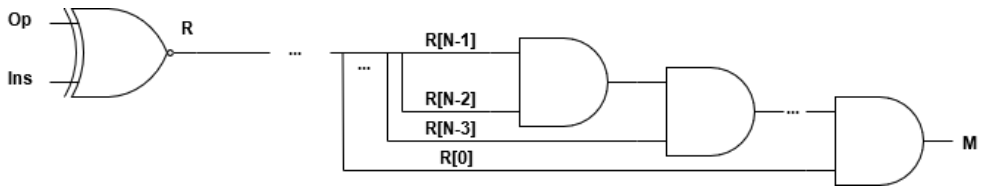

Figura 7: Circuito que retorna una señal igual a 1 para dos buses de bits equivalentes.

En este caso, `M = 1` si, y solo si todos los bits de `Op` coinciden con `Ins`, que es lo buscado.

c. Considere el siguiente programa:

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

En base a este:

I. 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
```

II. 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
```

III. A partir de los programas anteriores, explique el flujo resultante de cada uno de ellos en el diagrama del computador básico que se muestra a continuación.

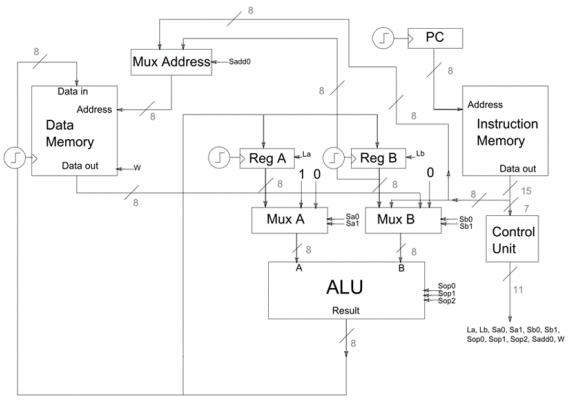

Figura 8: Computador básico visto hasta ahora.

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.

[3]Esto es muy útil al declarar arreglos.

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.
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.

d. **(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]Esto claramente no tiene mucho sentido.

e. **(I1 - II/2015)** Modifique el diagrama del computador básico de manera que soporte la ejecución de la instrucción `GOTO dir`, que fuerza que la siguiente instrucción en ejecutarse sea la ubicada en la dirección `dir`.

Para lograr esto, basta con agregar una conexión directa entre el literal de la memoria de instrucciones y el registro `PC`. A continuación se muestra esta idea:

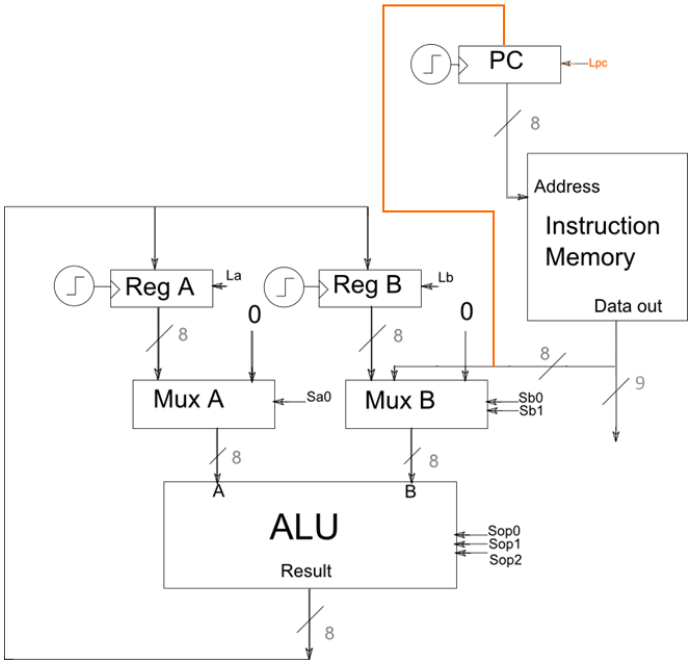

Figura 9: Computador básico con la implementación para la instrucción `GOTO dir`.

Basta con activar la señal L_{pc} una vez que esta instrucción sea llamada. Notar que esta instrucción es el equivalente al `JMP dir` que se encuentra en la versión final del computador básico, el que incluye saltos y subrutinas.

f. **(I1 - I/2018)** Modifique el computador básico para que acepte el comando `MOV A,[DIR]`, que toma el valor almacenado en la dirección `DIR` y luego considera ese valor como una dirección, va a esa dirección y almacena su valor en `A`.

Una posible solución consiste en agregar un registro interno temporal para este tipo de indirecciones, llamémoslo `T` de temporal. Este registro tendrá su entrada de datos del mismo bus de datos de `A` y `B` y su salida será conectada al `Mux A` del computador. Hay que agregar que la instrucción que se busca dar soporte **no puede** ser implementada en un ciclo de reloj, debido al funcionamiento de la RAM (necesita de dos flancos de subida para poder tener los dos accesos solicitados por la instrucción).

Lo que se hace por ciclo, entonces, es lo siguiente:

- **Ciclo 1:** Se hace uso del literal para cargar el valor almacenado en la dirección `DIR` de la RAM en el registro `T`.
- **Ciclo 2:** Se usa la salida del registro `T` como entrada para la dirección de la RAM (lo que implica su conexión con el `Mux Address`) y se obtiene el valor solicitado por la instrucción.

La siguiente figura muestra la adición del registro en conjunto con sus conexiones:

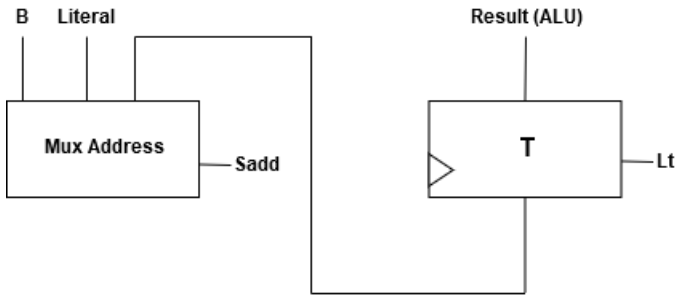

Figura 10: Registro `T` en conjunto con sus conexiones en el computador básico.

Note que es necesario el uso de la señal L_T, dado que no siempre queremos cambiar el valor del registro. Por otra parte, considerando la arquitectura del computador básico sin saltos y subrutinas, tenemos que ahora el `Mux Address` recibe tres entradas, por lo que se necesitan dos bits para seleccionar la salida de dicha componente.