PROBLEMAS: El repertorio de instrucciones

1. Un repertorio de instrucciones debe cumplir ciertas exigencias para ser considerado lenguaje de programación. A saber, debe ofrecer soporte para:

   - ejecutar saltos condicionales
   - invocar procedimientos
   - modificar porciones de memoria

   A la luz de estas exigencias, proponga un conjunto mínimo de operaciones para un repertorio de instrucciones que lo constituya como lenguaje de programación.

   SOLUCIÓN:

   El salto condicional más sencillo es saltar si una variable es 0. Con esta instrucción de salto condicional ya disponemos de soporte para toma de decisiones y la implementación de estructuras `if-then-else`, `switch-case`, `for` y `while`.

   Respecto a la invocación de procedimientos basta con disponer de un salto incondicional siempre que no exijamos recursividad, ni reentrancia y asumiendo que la dirección de retorno se escribe en una ubicación preestablecida. El salto incondicional se puede construir en base al salto condicional propuesto anteriormente siempre que nos aseguremos que la condición es `true`. En definitiva, no haría falta añadir una nueva instrucción.

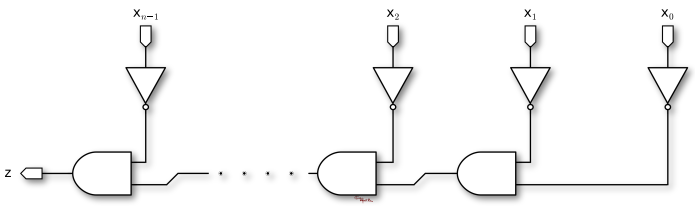
   En cuanto a la modificación de variables o porciones de memoria, basta con que las operaciones de proceso escriban sus resultados en memoria. ¿Qué operaciones serían indispensables? La operación de resta es la que contiene a todas las demás: sumas, negativos, productos, divisiones, etc.

2. Enumere qué información de estado es intrínseca al dato. ¿Qué operador *hardware* detecta cada una?

   SOLUCIÓN:

   Es intrínseca al dato toda aquella información que solamente requiera del propio dato para obtenerla. Es el caso de si es nulo (cero), si es negativo (signo), si tiene paridad par (par), si es todo unos, cuántos bits son cero a la derecha (ctz) o a la izquierda (clz), etc...

   Operadores *hardware* para estos estados son el detector de cero, el detector de signo, el detector de paridad... El detector de cero, ligeramente modificado, servirá como detector de todo unos...



Detector de ceros basado en propagación AND.
Sin inversores se convierte en un detector de unos.

   Respecto a la cuenta de bits a la derecha o a la izquierda, tenemos como ejemplo la extensión CIX (*Count Integer Extension*) al repertorio del procesador Alpha que consta de las siguientes instrucciones:

- `CTLZ`: *Count Leading Zero* cuenta la cantidad de ceros que hay desde el MSB.
- `CTPOP`: *Count Population* cuenta la cantidad de unos que hay en la codificación.
- `CTTZ`: *Count Trailing Zero* cuenta la cantidad de ceros que hay desde el LSB.

Estas instrucciones son útiles en la normalización de representaciones en coma flotante, para anticipar un posible desbordamiento, en la resolución de algunos algoritmos, en la planificación de procesos, etc..

Muchos repertorios las incorporan y también existen en funciones de librerías o macros de lenguajes de programación.

Desde el punto de vista *hardware* son operaciones relacionadas con las cadenas detectoras de primer '0' o primer '1'.

3. ¿Qué información de estado depende del dato y de la operación que lo generó?

SOLUCIÓN:

Aquella información que no depende solamente de la ristra de bits que observamos en el resultado. En general, el acarreo y el desbordamiento pero también aquellos estados que son resultado de la combinación de condiciones que involucran al acarreo o al desbordamiento, a saber:

- relaciones con signo: MAYOR-O-IGUAL-QUE combina el signo con el desbordamiento y el cero, es decir, $\overline{\text{signo} \oplus \text{des}}$ OR cero.
- relaciones sin signo: ENCIMA-O-IGUAL-QUE combina acarreo y cero, es decir, $\overline{\text{acarreo}}$ OR cero.

4. Sea un procesador que no tiene registro de estado. ¿Cómo se puede detectar un posible acarreo en suma? ¿Y en resta?

SOLUCIÓN:

Asumimos que estamos tratando números sin signo ya que en números con signo lo importante no es el acarreo sino el desbordamiento.

Suma:

Siempre que el resultado de la suma sea menor que cualquiera de los sumandos tenemos acarreo. El primer ejemplo está escrito en ensamblador del x86 aunque es un procesador con registro de estado. Para indicar que hay acarreo ponemos a '1' el registro `DX` y '0' en caso contrario:

```
        XOR DX, DX      ; el registro DX indicará el acarreo
        ADD AX, BX      ; AX <- AX + BX
        CMP AX, BX
        JAE #no_ac      ; si AX > BX no hay acarreo
        INC DX          ; en caso contrario sí y se indica en DX
#no_ac:  . . .
```

Los ejemplos que se muestran a continuación están escritos en el ensamblador del procesador ALPHA que no tiene registro de estado. Si hay acarreo se escribe '1' en un registro determinado por el programador y '0' en caso contrario:

```
        ; Rc = Ra + Rb, con Rd para señalar el acarreo
        ; el registro Rc es diferente al Ra
        ADDx    Ra, Rb, Rc      ; Rc = Ra + Rb
```

```
CMPULT  Rc, Ra, Rd      ; Rd = '1' si Rc < Ra => hay acarreo

; Rc = Ra + Rb, con Rd para señalar el acarreo
; el registro Rc es diferente al Rb
ADDx    Ra, Rb, Rc      ; Rc = Ra + Rb
CMPULT  Rc, Rb, Rd      ; Rd = '1' si Rc < Rb => hay acarreo

; Rc = Rc + Rc, con Rd para señalar el acarreo
; el registro Rd es diferente al Rc
BIS     Rc, Rc, Rd      ; Rd = Rc
ADDx    Rc, Rc, Rc      ; Rc = Rc + Rc
CMPULT  Rc, Rd, Rd      ; Rd = '1' si Rc < Rd => hay acarreo
```

La instrucción `ADDx` es una suma entera donde `x` puede ser `L` o `Q` para sumas de 32 o 64 bits respectivamente. Además, se puede añadir `/V` al mnemónico para que se dispare una excepción en caso de desbordamiento (se entiende que habrá desbordamiento solamente en caso de procesar números con signo ya que para números sin signo el desbordamiento es el acarreo).

Obsérvese que incluso en el caso de la instrucción `ADDL` que realiza la suma de operandos de 32 bits usando registros de 64 bits, no se utiliza la parte alta del registro para salvar un posible acarreo.

La instrucción `CMPUxx` es una comparación de números sin signo donde `xx` puede ser `LE` que examina la condición "menor que o igual" o `LT` que examina la condición "menor que". Si la comparación resulta verdadera se escribe el valor uno en el registro destino. En caso contrario se escribe cero.

La instrucción `BIS` es la `OR` o *bitwise or*. Se usa para mover datos entre registros ya que el procesador ALPHA no dispone de esa operación. Hacer la `OR` de un registro fuente consigo mismo y salvar el resultado en un registro destino es lo mismo que mover el operando fuente al destino.

Resta:

En resta, para detectar acarreo (o préstamo) hemos de comprobar si el sustraendo es mayor que el minuendo (en la operación $10 - 8 = 2$, el sustraendo es 8). La comprobación se puede realizar antes de hacer la operación:

```
; Rc = Ra - Rb, con Rd para señalar el acarreo
; el registro Rd es diferente a Ra y Rb
CMPULT  Ra, Rb, Rd      ; Rd = '1' si Ra < Rb
SUBx    Ra, Rb, Rc      ; Rc = Ra - Rb
```

La instrucción `SUBx` es una resta entera donde `x` puede ser `L` o `Q` para restas de 32 o 64 bits respectivamente. Además, se puede añadir `/V` al mnemónico para que se dispare una excepción en caso de desbordamiento.

5. Según establecieron Böhm y Jacopini en el teorema del programa estructurado (1966) toda tarea computable puede ser escrita combinando estructuras del tipo `if-then-else`, `switch-case`, `for` y `while`. Proponga una implementación de cada una de las cuatro usando instrucciones del lenguaje ensamblador.

SOLUCIÓN:

Primitiva para `if-then-else`:

```
if:     CMP <condición>
        Bcc #then
```

```
else:     ; código else
          . . .
          JMP #end_if
#then:    ; código then
          . . .
#end_if:  . . .
```

NOTA: las etiquetas que comienzan por '#' son del código mientras que las que no son explicativas.

Primitiva para `switch-case`:

```
          ; i es la variable
          CMP i, 1
          BNZ #case2
          CALL #subrutina_case1
#case2:   CMP i, 2
          BNZ #case3
          CALL #subrutina_case2
#case3:   CMP i, 3
          BNZ #default
          CALL #subrutina_case3
          . . .
#default: . . .
```

La siguiente primitiva para `switch-case` utiliza direccionamiento indirecto:

```
cases:    PTR #case1
          PTR #case2
          PTR #case3
          . . .
          ; bx es la variable
          JMP [cases + bx]

#case1:   ; código case1
          JMP #end_switch
#case2:   ; código case2
          JMP #end_switch
#case3:   ; código case3
          JMP #end_switch
          . . .
          . . .
#end_switch: . . .
```

Dado que el tamaño de los punteros será mayor que 1, lo más habitual es que el índice del caso que va sobre `BX` haya que multiplicarlo por 2 o por 4.

Es habitual que previamente a evaluar el índice del caso, se haga un pequeño control para ver si está o no en el rango y saltar, si no lo está, a la opción `default` si es que existe.

Primitivas para bucle por contador `for`:

```
          ; for (i=c; i=0; i--)
          MOV i, c
#bucle:   · · ·
          · · ·
          · · ·

          DEC i
          CMP i, 0
          BNZ #bucle
```

```
          ; for (i=0; i<c; i++)
          XOR i, i
#bucle:   · · ·
          · · ·
          · · ·

          INC i
          CMP i, c
          BB #bucle
```

Primitivas para bucle por condición `while`:

```
          ; while (condición)
#bucle:   · · ·
          · · ·
          · · ·

          CMP <condición>
          Bcc #bucle
```

Bucles `while` con 2 condiciones:

```
          ; while (cond1 AND cond2)
#bucle:   · · ·
          · · ·
          · · ·

          CMP <cond1>
          B!cc #end_while
          CMP <cond2>
          Bcc #bucle
#end_while: · · ·
```

```
          ; while (cond1 OR cond2)
#bucle:   · · ·
          · · ·
          · · ·

          CMP <cond1>
          Bcc #bucle
          CMP <cond2>
          Bcc #bucle
#end_while: · · ·
```

6. Utilizar la pila para dar soporte *hardware* a la recursividad de las subrutinas provoca una sobrecarga computacional tanto en número de instrucciones ejecutadas como en tiempo de acceso a memoria. Identifique qué instrucciones se ejecutan en la invocación, retorno y limpieza de una subrutina. Evalúe el tiempo empleado en todos los accesos a la pila suponiendo que cada acceso consume un tiempo igual a $12t_{reg}$ siendo t_{reg} el tiempo de acceso a un registro de la ruta de datos.

SOLUCIÓN:

A continuación se enumeran las instrucciones específicas utilizadas en la invocación, retorno y limpieza de una subrutina indicando si acceden a memoria o no. Asumimos que en promedio se pasan 3 argumentos, se declaran 2 variables locales y se utilizan 3 registros del banco de propósito general en el código de la subrutina. También suponemos que la salvaguarda de registros y el proceso de limpieza de la pila son responsabilidad del procedimiento llamador (*caller*).

Invocación:

- 3 instrucciones `PUSH` salvaguardan 3 registros en la pila; 3 accesos a memoria
- 3 instrucciones `PUSH` llevan los argumentos a la pila; 3 accesos a memoria

- 1 instrucción `CALL` salva en la pila la dirección de retorno y transfiere el control; 1 acceso a memoria
- 1 instrucción `PUSH` salva el marco de pila del procedimiento llamador; 1 acceso a memoria
- 1 instrucción `MOVE` carga el nuevo marco de pila; no hay acceso a memoria
- 2 instrucciones `ADD` reservan espacio para las variables locales; no hay acceso a memoria
- **TOTAL:** 11 instrucciones ejecutadas y 8 accesos a memoria

Retorno:

- 1 instrucción `MOV` ajusta la cima de la pila igualando al marco de pila; libera el espacio de las variables locales; no hay acceso a memoria
- 1 instrucción `POP` recupera el marco de pila del procedimiento llamador; 1 acceso a memoria
- 1 instrucción `RET` lee la dirección de retorno y transfiere el control al procedimiento llamador; 1 acceso a memoria
- **TOTAL:** 3 instrucciones ejecutadas y 2 accesos a memoria

Limpieza:

- 1 instrucción `SUB` limpia la pila de argumentos; no hay acceso a memoria
- 3 instrucciones `POP` recuperan los 3 registros compartidos con el procedimiento y que fueron salvaguardados justo al comienzo del proceso de invocación; 3 accesos a memoria
- **TOTAL:** 4 instrucciones ejecutadas y 3 accesos a memoria

En total, los procesos de invocación, retorno y limpieza propios de una llamada a procedimiento cuyo soporte *hardware* se basa en pila conllevan la ejecución de 18 instrucciones con 13 accesos a memoria. Podemos afirmar que cada instrucción `CALL` que aparezca en el código supone, en realidad, ejecutar aproximadamente unas 20 instrucciones adicionales dedicadas solamente a dar un adecuado soporte al procedimiento. Respecto al coste temporal, los 13 accesos a memoria suponen el siguiente consumo de tiempo aproximadamente:

$$13 \times 12t_{reg} = 156t_{reg}$$

Y todo esto, solamente en cuanto a transferencia de control sin contabilizar el procesamiento específico de la subrutina.

Es natural, por tanto, que cuando el compilador detecta que el procedimiento se usa muy poco lo elimine como tal copiando el código en lugar del `CALL`. Es lo que se conoce como optimización *inline expansion*.