Originalmente publicado en los números 165 (enunciado) y 166 (solución) de la revista Novática (`http://www.ati.es/novatica/`)

Problema A: ¿Dónde está mi interrupción?

Manuel Carro
Facultad de Informática
Universidad Politécnica de Madrid
`mcarro@fi.upm.es`

Oscar Martín
Unidad de Organización y Sistemas
Caja Madrid
`oscarm@epersonas.net`

1. Descripción del problema

Los microprocesadores modernos pueden interrumpir el flujo de control de un programa cuando se solicita, bien desde *hardware* (respondiendo al funcionamiento de un dispositivo físico), bien desde *software*. El control puede saltar entonces a una rutina de atención previamente especificada (el *manejador de interrupciones*), atender la interrupción ejecutando el código de la rutina y continuar con el programa en el punto en que su flujo fue derivado. Asumiremos que las interrupciones están numeradas desde 1 hasta $w, 1 \leq w \leq 32$, y que cada interrupción está asociada a un manejador distinto.

La recepción de la interrupción i puede ser habilitada o deshabilitada ajustando el bit i-ésimo de un registro de w bits (denominado *máscara de interrupciones*) a 1 (desbloqueo) o a 0 (bloqueo). En aras de la simplicidad, identificaremos de ahora en adelante la máscara de interrupciones con una variable denominada `imr0`. Simplificando, cada manejador de interrupciones puede cambiar el valor de `imr0` para deshabilitar la recepción de la interrupción que está siendo atendida (y así evitar caer en una sucesión ilimitada de reentradas en la rutina) y también para deshabilitar la recepción de otras interrupciones que tienen menos prioridad.[1]

Conocer el valor que un manejador de interrupciones deja en `imr0` es extremadamente interesante: si su valor final es 0, entonces ninguna otra interrupción podrá ser recibida tras el final de ese manejador. Nos interesa detectar si esta situación es posible analizando los valores que los manejadores de interrupciones pueden dejar en `imr0`. Las operaciones para manejar el valor de la máscara de interrupciones `imr0` están expresadas por el lenguaje que aparece en la figura 4, donde `imr`i, $i \geq 0$, son variables de w bits. Asumiremos, por tanto, que no hay saltos condicionales ni bucles.

Manejador	::=	*Instrucción* ; *Manejador* | *Instrucción* ;
Instrucción	::=	*Registro* `:=` *Op BinOp Op*
BinOp	::=	`and` | `or` | `xor`
Op	::=	*Valor* | `not` *Valor*
Valor	::=	*Registro* | *Número*
Registro	::=	`imr`i
Número	::=	`0` | `1` | `2` | ... | $2^w - 1$

Figura 1: Lenguaje de manejo de interrupciones

En las figuras 2 y 3 tenemos dos ejemplos de programas que asumen $w = 8$. El programa de la figura 2 deja un valor final 0 en `imr0` para los valores iniciales de `imr0` pertenecientes a $\{168 \ldots 175\} \bigcup \{184 \ldots 191\} \bigcup \{232 \ldots 239\} \bigcup \{248 \ldots 255\}$, mientras que el de la figura 3 nunca dejará un valor `imr0` = 0. La tarea a realizar es leer una serie de manejadores de interrupciones y determinar si es posible que, para algún valor inicial de `imr0`, el valor final de `imr0` sea 0, o si para **ningún** valor inicial de `imr0` su valor final será 0.

[1] En una arquitectura real existirían también interrupciones que no se pueden deshabilitar, cuya existencia ignoraremos aquí.

```
imr1 := not imr0 or not 23;
imr2 := imr1 xor not imr0;
imr3 := 87 or imr2;
imr0 := not imr3 and imr1;
```

Figura 2: Un programa que puede deshabilitar todas las interrupciones

```
imr1 := imr0 or not 23;
imr2 := imr1 xor not imr0;
imr3 := 87 or imr2;
imr0 := not imr3 or imr1;
```

Figura 3: Un programa que nunca deshabilitará todas las interrupciones

Descripción de la entrada

La entrada consistirá en el código de una serie de manejadores de interrupciones. Cada definición empieza con un número natural $k, 0 < k \leq 20,$ que se corresponde con el número de instrucciones de que consta el código del manejador y un número natural w que indica el número de interrupciones ($\equiv$ número de bits en cada `imr`i) a considerar. A continuación siguen k líneas con una instrucción en cada una de ellas. En un manejador con k instrucciones, `imr`i puede variar de `imr0` a `imr`k. Con la excepción de `imr0`, ningún `imr`i se usará en la parte derecha de una asignación antes de haber aparecido previamente en la parte izquierda de otra asignación.

Descripción de la salida

Para cada manejador, el programa debe imprimir la palabra `cero` si es posible que el manejador deje un valor 0 en `imr0` y `no cero` si esta situación es imposible para cualquier valor inicial de `imr0`.

Ejemplo de entrada

```
4 8
imr1 := not imr0 or not 23;
imr2 := imr1 xor not imr0;
imr3 := 87 or imr2;
imr0 := not imr3 and imr1;
4 7
imr1 := imr0 or not 23;
imr2 := imr1 xor not imr0;
imr3 := 87 or imr2;
imr0 := not imr3 or imr1;
1 32
imr0 := imr0 and imr0;
1 32
imr0 := imr0 or not imr0;
```

Salida para el ejemplo de entrada

```
cero
no cero
cero
no cero
```

2. Resumen del problema

Recordemos brevemente el problema propuesto en el número anterior: tenemos un extracto de un manejador de interrupciones del cual sólo nos interesan las órdenes que realmente cambian el valor de la máscara de interrupciones, `imr0`. El lenguaje con el que trataremos está generado

por la gramática de la figura 4; la semántica es la esperada (asignaciones y operaciones lógicas que son extensiones de las de bit a bit).

Un ejemplo de rutina aparece en la figura 5. En ella deseamos saber si el valor final de `imr0` puede ser 0 para algún valor inicial de `imr0`, o si, por el contrario, esa situación no es posible.

Manejador	::=	*Instrucción* ; *Manejador* |
		Instrucción ;
Instrucción	::=	*Registro* `:=` *Op BinOp Op*
BinOp	::=	`and` | `or` | `xor`
Op	::=	*Valor* | `not` *Valor*
Valor	::=	*Registro* | *Número*
Registro	::=	`imr`i
Número	::=	`0` | `1` | `2` | ... | $2^w - 1$

Figura 4: Lenguaje de manejo de interrupciones

```
imr1 := imr0 or not 23;
imr2 := imr1 xor not imr0;
imr3 := 87 or imr2;
imr0 := not imr3 or imr1;
```

Figura 5: Una rutina P de manejo de interrupciones

Cada rutina P implementa una función

$$f_P : \mathbb{B}^w \longrightarrow \mathbb{B}^w$$

donde w es el ancho (el número de bits) de cada `imr`i. Para reflejar mejor que cada registro se comporta como un vector de bits (para [des]inhibir interrupciones) utilizaremos explícitamente una notación vectorial en las variables y constantes. El código de la figura 5 implementa, por ejemplo, la función

$$f_P(\overline{x}) = \overline{x} \vee \neg\overline{23} = \overline{x} \vee \overline{11101000}_2$$

donde hemos asumido, como en el primer problema de ejemplo del enunciado, que el número de bits w en el registro de interrupciones de 8, y por tanto $\neg\overline{23}$ tiene 8 dígitos binarios. El problema inicial puede reformularse como averiguar si para una rutina P dada existe un valor inicial de la máscara de interrupciones $\overline{\texttt{imr0}}$ tal que $f_P(\overline{\texttt{imr0}}) = \overline{0}$.

Hay que resaltar que, dado que el lenguaje generado por la gramática de la figura 4 no incluye bucles, la ejecución de cualquier rutina escrita en este lenguaje terminará siempre para cualquier valor inicial de $\overline{\texttt{imr0}}$ y que, por la naturaleza de las operaciones lógicas, $f_P(\overline{x})$ está definida para cualquier $\overline{x} \in \mathbb{B}^w$.

3. Enumerando

Como la ejecución de cualquier P siempre termina (por no haber bucles en el lenguaje), un método de resolución consiste en enumerar todos los valores iniciales posibles de $\overline{\texttt{imr0}}$ (no más de 2^{32}, de acuerdo con los límites fijados en el enunciado del problema) y realizar una interpretación de la rutina P para cada uno de esos valores. El problema de este enfoque es el tiempo que esta interpretación puede llevar: recordemos que en los concursos de programación los tiempos de ejecución de programas están acotados para evitar, precisamente, soluciones triviales. Es posible acelerar esta interpretación usando técnicas conocidas de compilación:

- Precompilar el código de la rutina en una suerte de *código de byte*, que puede ser interpretado mucho más rápidamente. La única operación a realizar es asignación, y los números de

registros pueden utilizarse directamente para indexar un vector donde se guardan los valores de los registros.

- La compilación puede, en realidad, llevarse hasta el punto de sintetizar la función f_P correspondiente al programa P. Por ejemplo, como ya vimos, $f_P(\overline{x}) = \overline{x} \vee \neg\overline{23}$ para el programa de la figura 5. Esta función siempre puede tener la forma

$$f(\overline{x}) = (\overline{x} \wedge \overline{k_1}) \vee (\neg\overline{x} \wedge \overline{k_2})$$

en la que sólo hay que averiguar los valores de $\overline{k_1}$ y $\overline{k_2}$, y debería ser más rápida de interpretar que una versión de código de byte.

Sin embargo cualquiera de esas opciones parece demasiado trabajosa para realizar en el tiempo limitado disponible en un concurso de programación. Adicionalmente, no es seguro que la ejecución sea suficientemente rápida como para cumplir con las restricciones de tiempo impuestas por el procedimiento de corrección de los problemas. Para hacer una estimación, asumamos el siguiente bucle de comprobación:

```
for (imr = 0; imr != ~0; imr++) {
  if (((imr & k1) | (~imr & k2)) == 0) {
    cero = 1;
    break;
  }
}
```

El resultado de compilarlo (usando gcc 3.3 y nivel de optimización -O2 para i686) utiliza 10 instrucciones de ensamblador (entre las que hay dos de salto) para cada iteración. Un procesador de la familia i686 a 2 GHz de velocidad y que sea capaz de realizar un par de instrucciones por ciclo podría explorar 2^{32} valores en aproximadamente $(10 \times 2^{32})/(2 \times 2 \times 10^9) \approx 10{,}7$ segundos para un solo caso de prueba (el tiempo experimental ronda los 9,6 segundos). Esto podría exceder fácilmente los límites admisibles en un concurso de programación impuestos, precisamente, para evitar este tipo de algoritmos de (casi) fuerza bruta.

4. Bits independientes

Hay una propiedad importante que no hemos aprovechado: en las constantes y los registros el valor del bit i-ésimo no afecta al valor de otros bits en los registros, porque no hay ni operaciones de rotación ni aritméticas, y las lógicas son extensiones de operaciones bit a bit. La función f_P puede por tanto descomponerse en un vector de funciones $f_P = (f_0, f_1, \ldots, f_{w-1})$ (figura 6) que aplica cada uno de sus componentes a un bit x_i del valor de entrada $\overline{x}$: $f_P(\overline{x}) = (f_0(x_0), f_1(x_1), \ldots, f_{w-1}(x_{w-1}))$. Una condición equivalente a que $f_P(\overline{x})$ sea $\overline{0}$ para algún $\overline{x}$ es que $f_i(0) = 0 \vee f_i(1) = 0$ para todo i, y el valor $\overline{c}$ tal que $f_P(\overline{c}) = \overline{0}$ se puede hallar componiendo el valor de los bits i-ésimos que hacen que $f_i(c_i) = 0$. Por otro lado, si en algún momento hallamos que $f_i(0) = 1 \wedge f_i(1) = 1$, para algún i, entonces ya podemos concluir que el manejador nunca terminará con $\overline{\texttt{imr0}} = \overline{0}$.

Esta idea permite realizar una búsqueda mucho más inteligente: en vez de comprobar todos los valores desde 0 hasta $2^w - 1$ sólo es necesario comprobar los valores 0, 1, 2, 4, $\ldots$, 2^{w-1} (que sólo tienen un 1 en cada posición) — ¡una aceleración exponencial con respecto a la enumeración inicial! El algoritmo, a grandes rasgos, sería: poner cada bit a cero y a uno, independientemente. Si alguna f_i siempre se evalúa a uno (es decir, $\exists i f_i(0) = 1 \wedge f_i(1) = 1$), entonces $\overline{\texttt{imr0}}$ no puede ser $\overline{0}$ al final. En otro caso, existe un $\overline{\texttt{imr0}}$ inicial tal que $f_P(\overline{\texttt{imr0}}) = \overline{0}$.

Es posible refinar aún más esta idea: dado que en el cálculo del valor de salida cada bit es independiente de los demás, sólo es necesario calcular una vez $f_P(\overline{0})$ y $f_P(\overline{1})$, introduciendo primero el valor $\overline{\texttt{imr0}} = \overline{0}$ y luego $\overline{\texttt{imr0}} = \overline{1}$ (es decir, un registro inicial con todos los bits a cero o todos a uno). Por tanto, lo único que hay que comprobar es si $f_P(\overline{0}) \barwedge f_P(\overline{1}) \neq \overline{0}$, en cuyo

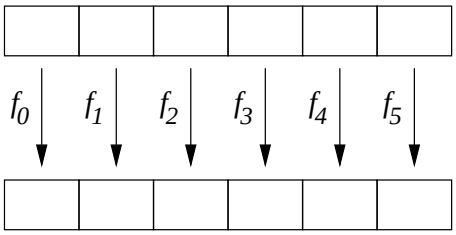

Figura 6: $f_P = (f_0, f_1, \ldots, f_{w-1})$

caso el manejador será seguro. La única precaución a tener en cuenta en la práctica es realizar las comparaciones pertinentes usando sólo w bits; de lo contrario las negaciones de valores (p. ej. `not 0`) podrían dejar bits a uno en la parte superior de los registros que falseasen los resultados finales.

Un esquema del algoritmo, en un seudocódigo procedimental, sería el siguiente:

```
read(Ancho, Manej, Fin);
while not Fin loop
   Unos    := 2 ** Ancho - 1;
   ResUno  := Interp(Manej, Unos);
   ResCero := Interp(Manej, 0);
   if (ResUno && ResCero && Unos = 0) then
       Print(cero);
   else
       Print(no cero);
   end if;
end loop;
```

donde `Interp()` evalúa el resultado que el manejador `Manej` deja en $\overline{\texttt{imr0}}$.

5. Código final

El código que sigue implementa en Prolog el algoritmo anterior, centrándose en la determinación del resultado para la definición de un manejador dado. No se presenta la lectura del programa en sí por ser algo relativamente mecánico: asumiremos que se ha construido ya una estructura de datos que representa el programa de forma adecuada.

El fragmento inicial, `ceroable/3`, recibe la definición de un manejador y el número de bits (el número de interrupciones) de ancho de los registros. Obsérvese que se llama sólo dos veces al intérprete del programa (`imr0_final/3`): una con un registro inicial $\overline{\texttt{imr0}} = \overline{0}$ y otra con $\overline{\texttt{imr0}} = \overline{1}$.

```
ceroable(Manej, Bits, Ceroable):-
    imr0_final(Manej, 0, V0),
    Unos is (1 << Bits) - 1,
    imr0_final(Manej, Unos, V1),
    Veredicto is V0 /\ V1 /\ Unos,
    (
        Veredicto =:= 0 ->
        Ceroable = cero
    ;
        Ceroable = 'no cero'
    ).
```

Un ejemplo de consulta (con la rutina de la figura 5) es:

```
?- ceroable((imr1 := imr0 or not 23;
             imr2 := imr1 xor not imr0;
             imr3 := 87 or imr2;
             imr0 := not imr3 or imr1),
             7, Z).

Z = 'no cero' ?
```

La evaluación del programa utiliza llamadas a un diccionario (una implementación de una tabla que almacena pares *clave-valor*) para guardar los valores de los registros intermedios y del registro final. El uso de variables lógicas permite que las operaciones de consulta y de primera inserción usen la misma llamada: obsérvese como `dic_lookup/3` se usa tanto para insertar el valor inicial `Ini` asociado a $\overline{\texttt{imr0}}$ como para recuperarlo en la variable `Fin`.

```
imr0_final(Manej, Ini, Final):-
    dic_lookup(Dic, imr0, Ini),
    evaluar(Manej, Dic, DicSal),
    dic_lookup(DicSal, imr0, Final).
```

El esquema del programa es un reflejo casi exacto de la gramática (figura 4), que se acompaña con los efectos derivados de la ejecución de cada instrucción. El bucle principal distingue el caso de la última instrucción o de una instrucción que aún no es la última. En cualquiera de esos casos el efecto de la instrucción a ejecutar se refleja en el cambio del estado del diccionario.

```
% Manejador ::= Instrucción; Manejador |
%               Instruccion;
evaluar((Ins;Insns), DicEnt, DicSal):-
    eval_inst(Ins, DicEnt, DicMed),
    evaluar(Insns, DicMed, DicSal).
evaluar(Ins, DicEnt, DicSal):-
    eval_inst(Ins, DicEnt, DicSal).
```

Las instrucciones tienen siempre la forma *Registro := Operación*; éste es el único punto en que se altera el estado del programa, y por tanto es el único sitio en que se cambia la asignación de valor a los registros.

```
% Instruccion ::= Registro := Operacion
eval_inst(Reg := Op, DIn, DOut):-
    eval_op(Op, DIn, Res),
    dic_replace(DIn, Reg, Res, DOut).
```

Todas las operaciones son binarias y se evalúan descomponiéndolas en sus operandos,[2] evaluando cada uno de ellos y después realizando la operación correspondiente con los valores ya obtenidos.

```
% Operacion ::= Op BinOp Op
eval_op(Op, Dic, Res):-
    Op    =.. [BinOp, Op1, Op2],
    ValOp =.. [BinOp, V1, V2],
    operando(Op1, V1, Dic),
    operando(Op2, V2, Dic),
    operar(ValOp, Res).
```

[2]Como nota para aquellos que usen Prolog habitualmente: aunque usualmente no se recomienda utilizar `=../2` en los programas, debido a la lentitud y gasto de memoria, nosotros lo hemos empleado aquí porque la velocidad no es un problema y proporciona un código algo más corto que otras alternativas.

La forma de cada operando distingue tres casos:

- La negación de una expresión más simple, que se resuelve determinando el valor de la expresión y después negándola.
- Una constante numérica, que tiene como valor a ella misma.
- Un nombre de registro, que se resuelve accediendo al valor asociado al mismo en el diccionario (recordemos que, exceptuando imr0, un registro sólo puede aparecer en la parte derecha de una asignación si antes ha aparecido en la izquierda, y por tanto ya tiene un valor asignado).

```
% Op    ::= Valor | not Valor
% Valor ::= Registro | Número
% Reg   ::= imri
% Num   ::= 0 | 1 | 2 | ... |  2^w - 1
operando(not Valor, R, Dic):-
    operando(Valor, V, Dic),
    R is \ V.
operando(Valor, Valor, _Dic):-
    number(Valor).
operando(Reg, Valor, Dic):-
    atom(Reg),
    dic_lookup(Dic, Reg, Valor).
```

Finalmente, las operaciones con valores ya determinados se realizan utilizando las operaciones bit a bit directamente disponibles en Prolog.

```
% BinOp ::= and | or | xor
operar(V1 and V2, R):- R is V1 /\ V2.
operar(V1  or V2, R):- R is V1 \/ V2.
operar(V1 xor V2, R):- R is V1  # V2.
```

6. Más datos

La estrategia habitual para evitar que una interrupción no inhibida provoque llamadas (reentrantes) a un manejador de interrupciones es deshabilitarla de forma automática en el momento de ser recibida — pero entonces el manejador debe ocuparse de habilitarla de nuevo al terminar su ejecución.

El problema (expuesto de forma simplificada aquí) es de especial importancia en la implementación de sistemas empotrados y de tiempo real, en los que el sistema operativo debe ser lo más liviano posible y se aprovechan al máximo las capacidades del *hardware*. En estos casos el arranque de tareas puede realizarse mediante el uso de interrupciones, y las prioridades relativas entre las tareas del sistema pueden ser implementadas mediante la (des)habilitación explícita de interrupciones, como se ha descrito.

En un caso real el esquema debe ser más complicado: las (des)habilitaciones pueden aparecer entremezcladas con el resto del código, dentro de expresiones condicionales o dentro de bucles. El análisis del enmascaramiento necesita tener en cuenta el control de flujo; éste es un campo del análisis de programas en el que se realiza investigación en estos momentos [CMM+03].

Referencias

[CMM+03] Krishnendu Chatterjee, Di Ma, Rupak Majumdar, Tian Zhao, Thomas A. Henzinger y Jens Palsberg. Stack Size Analysis for Interrupt-Driven Programs. En *Tenth Interna-*

tional Static Analysis Symposium (SAS), número 2694 en Lecture Notes in Computer Science, páginas 109–126. Springer-Verlag, 2003.