Implementación de componentes de un procesador SPARC utilizando simulación de eventos discretos

Autores:
Enrique, Sebastian e-mail: senrique@dc.uba.ar
Teléfono: 02320-490197
Fax: 4756-9391
Glinsky, Ezequiel J. e-mail: eglinsky@dc.uba.ar
Teléfono: 4805-5318
Fax: 4806-4044

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Director:
Dr. Gabriel Wainer e-mail: gabrielw@dc.uba.ar

Materia:
Organización del Computador 1 – 2° año, 1° cuatrimestre

Mayo 1999

Resumen:
La herramienta CD++ sigue el formalismo DEVS de simulación de eventos discretos. Con ella se desarrollan varios componentes de un procesador SPARC utilizando lógica digital.
El objetivo es armar una biblioteca para que, juntos a otros componentes, puedan ser integrados para simular la arquitectura y funcionamiento del procesador completo.
En este informe se hace una introducción a DEVS, continuando con la descripción, desarrollo, e implementación de los componentes, algunos de los resultados obtenidos en las pruebas, y conclusiones.

Palabras Clave: sistemas de eventos discretos, DEVS, organización de computadoras, CD++, lógica digital

INTRODUCCIÓN

DEVS (Discrete EVents Systems specification) es un formalismo propuesto por Zeigler [Zei76] para el modelado de sistemas de eventos discretos. La característica de éstos para poder ser tratados mediante este formalismo es que deben poder representarse por variables discretas y tiempo contínuo. El conjunto de estas variables discretas forman los estados del sistema.

DEVS permite simular un sistema mediante modelos jerárquicos. Los modelos básicos son llamados atómicos, y el conjunto de estos forman estructuras más complejas conocidas como modelos acoplados. De esta manera, puede refinarse la simulación tanto como uno quiera, permitiendo reutilización de modelos, fácil mantenimiento, menor tiempo de testeo, y mayor productividad.

Un modelo atómico DEVS se describe como:
$\mathbf{M} = < \mathbf{X}, \mathbf{S}, \mathbf{Y}, \boldsymbol{\delta}_{int}, \boldsymbol{\delta}_{ext}, \boldsymbol{\lambda}, \mathbf{D} >$

donde,
X: conjunto de eventos externos de entrada;
S: conjunto de estados secuenciales;
Y: conjunto de eventos externos generados para salida;
$\boldsymbol{\delta}_{int}$: $\mathbf{S} \rightarrow \mathbf{S}$, función de transición interna, que define los cambios de estado por causa de eventos internos;

δ_{ext}: **Q** x **X** → **S**, función de transición externa, que define los cambios de estado por causa de eventos externos; donde el conjunto de eventos totales del sistema es **Q** = { (**s**, **e**) / **s** **S**, **e** ∈ [0, **D**(**s**)] }, siendo **e** el tiempo transcurrido desde la última transición de estado con **s** -es decir, la función de transición externa depende del estado actual y del tiempo transcurrido-.
λ: **S** → **Y**, función de salida; y
D: **S** → $\Re_0^+$, función de duración de un estado -**D**(**s**) es el tiempo que el modelo se queda en el estado **s** si no hay un evento externo-.
Cada modelo puede verse como con puertos de entrada/salida para comunicarse con otros modelos. Los eventos de entrada y salida determinan los valores en esos puertos. Los eventos de entrada externos se reciben por el puerto de entrada, y la especificación del modelo debe definir las propiedades bajo los mismos. Los eventos internos producen cambios de estado, cuyos resultados se envian a través de los puertos de salida.

CD++ [BBW98a] es una herramienta que implementa los conceptos teóricos para el mecanismo de simulación del paradigma DEVS. Utilizando C++, los modelos atómicos se programan e incorporan en la jerarquía de clases. Un lenguaje de especificación permite definir el acoplamiento de los modelos, valores iniciales y los eventos externos de entrada. Al correr la simulación, la herramienta envía a la salida el desarrollo, eventos, y transiciones de la misma. Cuatro tipos de mensajes son generados:
*****: indica un cambio de estado a causa de un evento interno;
X: se produce la llegada de un evento externo -incluye port y valor-;
Y: salida del modelo;
done: indica que el modelo finalizó con su tarea.
Las funciones que deben tenerse en cuenta para programar un modelo son:
externalFunction: (función de transición externa) es activada cuando el valor de alguno de los puertos de entrada cambia. En esta función se programa la reacción del modelo a cambios externos.
outputFunction: (función de salida) aquí se programan las salidas que debe emitir el modelo.
internalFunction: (función de transición interna) aquí se programan la reacción del modelo a cambios internos.

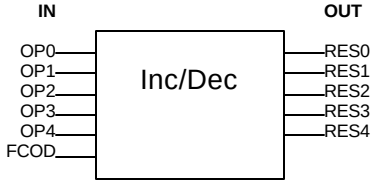
DESCRIPCIÓN DE LOS COMPONENTES A SIMULAR

El diseño de la arquitectura del modelo a simular se basa en la especificación del *Integer Unit* del procesador SPARC de Sun Microsystems. Se trata de la misma arquitectura RISC, a la que se le han simplificado el set de instrucciones y el manejo de la memoria.
Para el desarrollo de los modelos utilizamos CD++ y C++, compilando con GCC 2.8.1. Se utilizaron además algunos componentes de una biblioteca de lógica digital [FRW97].
Los modelos utilizan un tiempo configurable de estabilización de los registros. Es decir, que luego de recibida una entrada, el modelo tarda el tiempo de estabilización en retornar a un estado correcto; emitiendo las salidas necesarias y realizando las acciones pertinentes.
Por último, todos los componentes están inicializados en cero, y están armados de tal forma que el primer evento externo genere una salida cuando exista un cambio de fase.

INC/DEC

La arquitectura tiene 520 registros de uso general, organizados en ventanas que se superponen. En un instante dado, sólo 32 registros están disponibles. El esquema se implementa con un arreglo de 8 registros que contiene los registros globales, y otro de 512, donde una ventana de 24 registros aloja los registros de salida, los locales y los de entrada. CWP es la variable que apunta la ventana activa. Inc/Dec es el componente que se utiliza para desplazar la ventana, incrementando o decrementando el valor de CWP. Éste último es de 5 bits. Opera en forma circular.
Para su implementación, utilizamos un componente de la biblioteca de lógica digital (dlatch) para guardar los valores de los distintos ports del modelo, que cumplen virtualmente la función de un arreglo. El valor FCOD –que indica si es incremento o decremento- también se almacena en un dlatch. Utilizamos además el componente ALU para incrementar o decrementar la salida, según corresponda.
Los puertos de entrada y salida del modelo son:

Línea	In	Out	Descripción
OP4..0	X		Operando
FCOD	X		1 = INC; 0 = DEC
RES4..0		X	Resultado



En el gráfico superior derecho se muestra el componente Inc/Dec y sus puertos.

A continuación, con el fin de ejemplificar el uso de la herramienta y de mostrar parte del trabajo realizado, mostramos el código fuente correspondiente a las funciones de salida y de transición externa:

```
Model &IncDec::externalFunction( const ExternalMessage &msg )
{
    // Pregunta por el port de entrada y asigna el valor recibido a quien corresponda
    if( msg.port() == OP0 ) _OP[0].activar((char)msg.value()+48, '1');
    if( msg.port() == OP1 ) _OP[1].activar((char)msg.value()+48, '1');
    if( msg.port() == OP2 ) _OP[2].activar((char)msg.value()+48, '1');
    if( msg.port() == OP3 ) _OP[3].activar((char)msg.value()+48, '1');
    if( msg.port() == OP4 ) _OP[4].activar((char)msg.value()+48, '1');
    if( msg.port() == FCOD ) _FCOD.activar((char)msg.value()+48, '1');
    if( _FCOD.salida() == '1' ) // incrementa con FCOD=1
    {
    // Si recibí que debo incrementar, lo hago
    char s[6];
    for (int i=0; i<=4; i++)              // Guardo en vector s el operando
        s[4-i] = _OP[i].salida();         // que voy a incrementar
    s[5] = '\0';
    c_alu  alu(5);
    alu.activar(s,"00000","11",'1');      // Incremento con la ALU
    char *res = alu.salida();

    for (int i = 0; i<=4; i++)
        _RES[i].activar(*(res+(4-i)), '1');
    }
    else
    {
    // Si recibí que debo decrementar, lo hago
    char s[6];
    for (int i=0; i<=4; i++)              // Guardo en vector s el operando
        s[4-i] = _OP[i].salida();         // que voy a decrementar
    s[5] = '\0';
    c_alu  alu(5);
    alu.activar(s,"11111","11",'0');      // Decremento con la ALU
    char *res = alu.salida();

    for (int i = 0; i<=4; i++)
        _RES[i].activar(*(res+(4-i)), '1');
    }
    this->holdIn(active, preparationTime); // Programo la próxima salida dentro
                                          // del preparation time establecido
    return *this;
}


Model &IncDec::outputFunction( const InternalMessage &msg )
// Debe chequear si alguna de los nuevos valores obtenidos en el registro
// RES difiere de la última salida emitida
{
    if (_RES[0].salida() !=_OLD[0].salida() || _RES[1].salida() !=_OLD[1].salida() ||
      _RES[2].salida() !=_OLD[2].salida() || _RES[3].salida() !=_OLD[3].salida() ||
      _RES[4].salida() !=_OLD[4].salida() || _first )
    {   // Si existe algún valor que difiere de la última salida
        // emitir por el puerto el valor correspondiente
        // (-48 por el ASCII del “0”, para emitir 0 o 1 según corresponda)
        sendOutput( msg.time() , RES0, _RES[0].salida()-48 );
        sendOutput( msg.time() , RES1, _RES[1].salida()-48 );
        sendOutput( msg.time() , RES2, _RES[2].salida()-48 );
        sendOutput( msg.time() , RES3, _RES[3].salida()-48 );
        sendOutput( msg.time() , RES4, _RES[4].salida()-48 );
        // Almacenar la salida actual como última salida emitida
        _OLD[0].activar(_RES[0].salida(), '1');
        _OLD[1].activar(_RES[1].salida(), '1');
        _OLD[2].activar(_RES[2].salida(), '1');
        _OLD[3].activar(_RES[3].salida(), '1');
        _OLD[4].activar(_RES[4].salida(), '1');
        _first=false;
    }     // si no, continua

      return *this ;
}
```

MEMORIA

La memoria del modelo es plana, sin proveer mecanismos de paginación o segmentación. No soporta multiprogramación. La implementación incluye dos registros, BASE y LÍMITE, que determinan el espacio de direcciones disponible para el programa. El direccionamiento es a byte, y la información es guardada según el standard Little-Endian.

La operatoria es la siguiente: cuando el bus necesita una operación con la memoria, prepara todas las señales, datos, dirección, etc.; una vez que está todo listo, envía la señal AS, que la memoria interpreta e identifica como que puede comenzar a operar. Toma la dirección –A2..A31-, y establece el tipo de operación, lectura o escritura, según lo recibido en RD_WR. Si es lectura, se toman los 4 bytes desde la dirección especificada y se envían por las líneas de datos; si es escritura, se graban únicamente los bytes indicados en el byte select –BSEL0..3, tomados también del bus-. Si la dirección tomada desde el bus es incorrecta, es decir, se intenta apuntar a una posición no disponible por el tamaño de la memoria real, se enciende la señal de error –ERR-. Una vez que está concluido el proceso de búsqueda, lectura/escritura, se informa de ello enviando al bus la señal de data acknowledge –DTACK-.

Los puertos de entrada y salida se muestran en la siguiente tabla:

Línea	In	Out	Descripción
DATA0..31	X	X	Data
A2..31	X		Address
BSEL0..3	X		Byte Select
AS	X		Address Strobe
RD_WR	X		Read or Write
DTACK		X	Data Acknowledge
ERR		X	Error
RESET	X		Reset

Para una mejor implementación, puede configurarse el tamaño y cargarse la imagen inicial de la memoria desde un archivo.

En la codificación, se utilizaron arreglos de enteros para guardar las distintas líneas del modelo. Éste tiene la particularidad de utilizar un archivo, de tamaño definido por el usuario, para guardar los datos actuales de la memoria. Además puede inicializarse leyendo un archivo con los datos que se desean tener en la memoria al momento de iniciar la simulación. Si la simulación acaba, un archivo contendrá la imagen del último estado de la memoria.
Para direccionar los datos se utiliza el archivo de manera secuencial, es decir la posición 0 del archivo equivale al byte 0 de la memoria, y así sucesivamente.
El parámetro *memsize* contiene el tamaño de la memoria en bytes. Si no se especifica, el valor por defecto es 32kb. Si el valor especificado es mayor que el tamaño de la imagen provista, se rellena con ceros; si es menor, se trunca. Si no existiera el archivo `MEMORIA.MEM`, se crea una memoria vacía del tamaño que indique *memsize*.
Para la ubicación de una posición de memoria se utiliza una función (calc_dir()), que en base a los valores de las líneas de entrada se posiciona en el lugar correspondiente dentro del archivo.

Parámetros
tiempo_de_preparación; memorysize

Transición externa
Interpreta la señal recibida por algún puerto de entrada y toma acción según corresponda
Cuando recibo **x** en el port **y**, **_y ← x**
Si **_RESET** = 0 /* resetar */
    Cargar la imagen inicial.
    _RESET ← 1
    enviar 1 al port **DTACK**
    continuar
Si **_AS** = 1 /* la dirección está lista */
    dir **← A2..31**
    Si dir > memorySize-4 /* la dirección está por encima de la memoria disponible */

enviar 1 al port **ERR**
continuar
Si **_RD_RW** = 1 /* lectura */
dato ← memoria[dir] /* 4 bytes */
_DATA_OUT0..31 ← dato0..31
Sino /* escritura */
Si **_BSEL0** = 1 entonces dato0..7 ← **_DATA_IN0..7**; memoria[dir] ← dato0..7
Si **_BSEL1** = 1 entonces dato8..15 ← **_DATA_IN8..15**; memoria[dir+1] ← dato8..15
Si **_BSEL2** = 1 entonces dato16..23 ← **_DATA_IN16..23**; memoria[dir+2] ← dato16..23
Si **_BSEL3** = 1 entonces dato24..31 ← **_DATA_IN24..31**; memoria[dir+3] ← dato24..31
mantener ENVIANDO durante tiempo_de_preparación

Transición interna
Al existir cambios sólo por eventos externos, no hace nada
pasivar

Salida
Envia el valor incrementado o decrementado por los puertos de salida
Si **_AS** = 1 /* La dirección está lista */
_AS ← 0
enviar 1 al port **DTACK**
enviar 0 al port **ERR**
_DTACK ← 0
Si **_RD_WR** = 1 /* lectura */
enviar **_DATA_OUT0..31** a los ports **DATA_OUT0..31**
continuar

En el cuadro siguiente puede apreciarse parte de la especificación del modelo:

```
[top]
components : Mem@Mem
out : DATA_OUT0 DATA_OUT1 DATA_OUT2 DATA_OUT3 DATA_OUT4 DATA_OUT5 DATA_OUT6 DATA_OUT7
...
in : DATA_IN0 DATA_IN1 DATA_IN2 DATA_IN3 DATA_IN4 DATA_IN5 DATA_IN6 DATA_IN7 DATA_IN8
...
     A20 A21 A22 A23 A24 A25 A26 A27 A28 A29 A30 A31
     BSEL0 BSEL1 BSEL2 BSEL3
     AS RD_WR RESET

Link : DATA_OUT0@Mem DATA_OUT0
Link : DATA_OUT1@Mem DATA_OUT1
Link : DATA_OUT2@Mem DATA_OUT2
...
Link : A29 A29@Mem
Link : A30 A30@Mem
Link : A31 A31@Mem
Link : BSEL0 BSEL0@Mem
Link : BSEL1 BSEL1@Mem
Link : BSEL2 BSEL2@Mem
Link : BSEL3 BSEL3@Mem
Link : AS AS@Mem
Link : RD_WR RD_WR@Mem
Link : RESET RESET@Mem

[Mem]
preparation : 0:0:5:0
memsize : 512
```

En [top] se indican qué componentes forman el modelo, en este caso sólo la memoria (es atómico). Las etiquetas out e in se utilizan para indicar los puertos de entrada y salida que contiene. Link se utiliza para especificar qué puertos del modelo tienen conexión con los puertos de cada componente. Por ùltimo, en [Mem] se especifican los parámetros para ese modelo. El parámetro *preparation* indica el tiempo de preparación (estabilización) del modelo, mientras que *memsize* permite definir el tamaño de la memoria.

MUX4

El multiplexor de cuatro entradas es un componente que, a partir de una señal de selección, toma la entrada indicada y la envía por la salida. Varios multiplexores de este tipo se encuentran dentro de la arquitectura a simular, formando parte de las unidades aritmético-lógica y de direccionamiento, entre otras.

Las cuatro entradas están formadas por datos de 32 bits cada una. Para seleccionarlas, se recibe la señal select de cuatro bits, donde el bit encendido en la misma indicará cuál de las entradas emitir por la salida.

A continuación se muestra la tabla con los puertos de entrada y salida:

Línea	In	Out	Descripción
A31..0	X		Operando A
B31..0	X		Operando B
C31..0	X		Operando C
D31..0	X		Operando D
SELA	X		Select A
SELB	X		Select B
SELC	X		Select C
SELD	X		Select C
OUT31..0		X	Salida

El pseudocódigo de operación de este modelo es:

Si **SELA** = 1, envía a la salida (**OUT31..0**) el valor en **A31..0**;
Si **SELB** = 1, el de **B31..0**;
Si **SELC** = 1, el de **C31..0**;
Si **SELD** = 1, el de **D31..0**.

Para su implementación, utilizamos un multiplexor definido en la biblioteca de lógica digital de 4 entradas. En primer lugar, convertimos el select de 4 bits que llegan al modelo al de 2 bits que interpreta el multiplexor digital, luego lo activamos por cada bit de los datos de entrada. Finalmente los valores resultado se envian si y solo si difieren de la última salida.

TRAPLOGIC

Este componente es el encargado de determinar cuál es el trap que deberá atenderse. Está basado en un sistema de prioridades.
Consta de una serie de entradas para ciertos traps distinguidos, una entrada que indica que ha ocurrido un trap “común” y 7 bits para indicar el número de trap “común”. Como salida tiene un bit que indica si hay algún trap para atender y 8 bits para indicar cuál es el trap (*trap type*).
Consta de los registros (simulados mediante los componentes de la biblioteca de lógica digital) TN y Traps de 7 y 12 bits respectivamente. Se encargan de almacenar los valores recibidos por los ports de entrada y del registro lastTT (8 bits) y el latch lastTF para almacenar los últimos valores de salida y emitir resultados sólo en el caso de que los mismos difieran de ellos.
El registro Traps tiene un bit por cada tipo de trap existente y los mismos están ordenados descendentemente por prioridad.
En el siguiente cuadro pueden verse los distintos ports de entrada y salida (In y Out), prioridades y tipos de los distintos traps existentes:

Línea	In	Out	Descripción	Prioridad	Trap Type
INST_ACC_EXCEP	X		Instruction access exception	5	0x01
ILLEG_INST	X		Illegal instruction	7	0x02
PRIV_INST	X		Privileged instruction	6	0x03
WIN_OVER	X		Window overflow	9	0x05
WIN_UNDER	X		Window underflow	9	0x06
ADDR_NOT_ALIGN	X		Address not aligned	10	0x07
DATA_ACC_EXCEP	X		Data access exception	13	0x09

Línea	In	Out	Descripción	Prioridad	Trap Type
INST_ACC_ERR	X		Instruction access error	3	0x21
DATA_ACC_ERR	X		Data access error	12	0x29
DIV_ZERO	X		Division by zero	15	0x2A
DATA_ST_ERR	X		Data store error	2	0x2B
TRAP_INST	X		Trap instruction	16	0x80+TN
TN6.. 0	X		Trap number		
TF		X	Trap found		
TT7.. 0		X	Trap type		

De acuerdo al cuadro anterior, el componente que simulamos registra cuál es el trap con mayor prioridad que debe ser atendido y luego del tiempo de preparación emite por los puertos de salida el índice correspondiente.

Parámetros
tiempo_de_preparación

Transición externa
Almacena en sus registros los valores recibidos por los ports de entrada y programa el tiempo de la próxima ejecución de la función de salida y transición interna.

Transición interna
El modelo sólo se ve afectado por cambios externos, por ende lo único que se hace aquí es pasivar
pasivar

Salida
Verificar el registro TF (trap found) para chequear si existe algún trap que requiere ser atendido y emitir la salida del trap correspondiente
Si **_TF** /* existe algún trap pendiente */
TT0..7 ← **_TT0..7** /* índice del trap de mayor prioridad */

MANEJO DEL BLOQUE DE REGISTROS

Dos componentes fundamentales para el manejo de los registros de la computadora simulada son el Regblock y el Wimcheck. Ambos componentes fueron implementados y su funcionamiento se describe a continuación:

REGBLOCK

Es el bloque de registros a utilizarse en los procedimientos (sin incluir los 8 registros globales). Este componente permite almacenar o leer valores sobre cada uno de los registros existentes.
Mediante un arreglo de 512 registros de 32 bits simulamos los registros manejados por el componente. Los registros lastAOUT y lastBOUT sirven para almacenar la última salida y emitir señales sólo para los bits que cambien.
En el siguiente cuadro pueden verse los distintos ports de entrada y salida (In y Out) existentes:

Línea	In	Out	Descripción
ASEL8..0	X		Select A
BSEL8..0	X		Select B
CSEL8..0	X		Select C
CEN	X		Enable C
RESET	X		Reset
AOUT31..0		X	Output A
BOUT31..0		X	Output B
CIN31..0	X		Input C

Transición externa
Si **_CEN**=1 /* Almacenar el valor indicado en CIN31..0 en el CSEL-ésimo registro */
Registros[_CSEL0..8] = _CIN31..0
Si **_RESET**=1 /* Resetear todos los valores */
for (i=0; i<512; i++) Registros[i] = 0
_AOUT31..0 = BOUT31..0 = ASEL8..0 = BSEL8..0 = CSEL8..0 = CIN31..0 = CEN = 0
_AOUT0..31 = Registros[**_ASEL0..8**] /* AOUT = ASEL-ésimo registro*/
_BOUT0..31 = Registros[**_BSEL0..8**] /* BOUT = BSEL-ésimo registro*/

Transición interna
El modelo sólo se ve afectado por cambios externos, por ende lo único que se hace aquí es pasivar
pasivar

Salida
Solo emite salida si difiere de la anterior
Si **_AOUT0..31 <> LastAOUT** /* la salida AOUT cambió */
enviar **_AOUT0..31** a los ports **AOUT0..31**
LastAOUT = _AOUT0..31
Si **_BOUT0..31 <> LastBOUT** /* la salida BOUT cambió */
enviar **_BOUT0..31** a los ports **BOUT0..31**
LastBOUT = _BOUT0..31

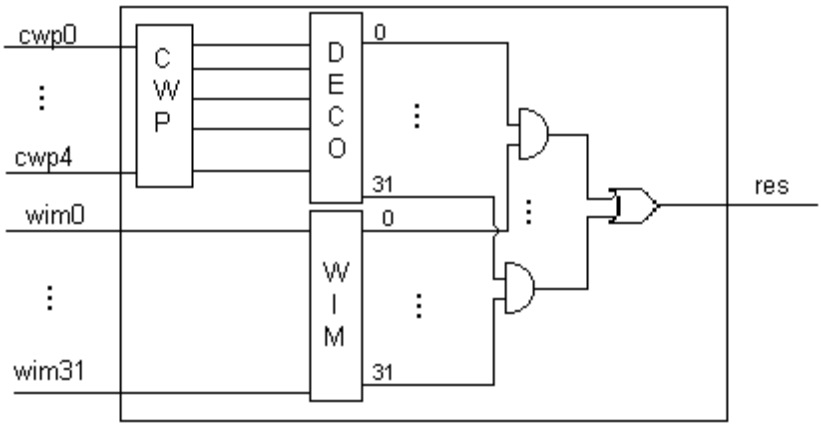
WIMCHECK

Es el componente de hardware que se encarga de hacer el chequeo que indica si la próxima ventana de registros a utilizar sobreescribirá a una que ya está utilizada. Para ello recibe los registros *WIM* y *CWP* y devuelve el valor del CWP-ésimo bit del *WIM*. Si el bit vale 1, el CWP está apuntando a la ventana más antigua que está siendo utilizada.
Este componente consta de un registro para almacenar los valores que recibe por los ports WIM (32 bits) y otro para los CWP (5 bits). Utiliza un decodificador de 5 líneas de entrada y un latch para almacenar el último resultado y no emitir una nueva señal de salida en caso de que no se produzca un cambio de voltaje.

En el siguiente cuadro pueden verse los distintos ports de entrada y salida (In y Out) existentes:

Línea	In	Out	Descripción
CWP4..0	X		Current Window Pointer
WIM31..0	X		Window Invalid Mask
RES		X	Res = WIM_{CWP}

Se utilizó un decodificador de la biblioteca de lógica digital. A continuación mostramos un gráfico del circuito simulado:



RESULTADOS

A modo de ejemplo mostramos a continuación algunas pruebas realizadas con los modelos implementados [AG98] [ER99]. El tiempo de preparación utilizado en todos los casos es de 00:00:00:040.

La primera corresponde al modelo Regblock. El test consiste en el seteo del valor 1 al registro 2 y la consulta posterior del mismo.
Inicialmente están todas las líneas en cero. Al tiempo 00:00 de simulación, se envía un 1 en CEN para prenderlo. Al tiempo 00:10 el bit 7 de CSEL se pone en 1. En 00:20, el bit 31 del input C se enciende. Lo mismo sucede para el bit 7 del ASEL en 10:00. En 10:40 el puerto de salida AOUT envía su bit 31 encendido.

```
Starting simulation ... stop at ...
00:00:00:000 / cen / 1.000
00:00:00:010 / csel7 / 1.000
00:00:00:020 / cin31 / 1.000
00:00:10:000 / asel7 / 1.000
00:00:10:040 aout31 1
Simulation ended!
```

Este segundo test corresponde al modelo Wimcheck. Muestra el resultado obtenido luego de consultar por el bit indicado con el CWP. Luego de la primer consulta efectuada el resultado esperado por el puerto de salida **Res** es 1, la respuesta es correcta y se obtiene en el instante 1:00:040 debido a que el Wim2 se encuentra encendido. El resultado obtenido en el instante 2:00:040 es un 0 por el puerto **Res**, lo cual es correcto debido a que refleja que el bit de Wim2 (por el cual se sigue consultando) ha sido apagado.

```
Starting simulation ... stop at ...
00:00:00:001 / cwp0 / 0.000
00:00:00:002 / cwp1 / 0.000
00:00:00:003 / cwp2 / 0.000
00:00:00:004 / cwp3 / 1.000
00:00:00:005 / cwp4 / 0.000
00:01:00:000 / wim2 / 1.000
00:02:00:000 / wim2 / 0.000
00:01:00:040 res 1
00:02:00:040 res 0
Simulation ended!
```

Este tercer test prueba el modelo Wimcheck en un caso un poco más complejo:

1) Se setea el WIM en sus bits 8 y 9 en 1.
2) Se consulta por el WIM 8. Se espera como respuesta un 1.
3) Se consulta por el WIM 9. Se espera que no haya una nueva respuesta, ya que la anterior fue un 1 (y recordemos que sólo se emite respuesta en caso de que hubieran cambios de estado).
4) Se consulta por el WIM 0. Se espera como respuesta un 0.
5) Se consulta por el WIM 9. Se espera como respuesta un 1.
6) Se setea el WIM 8 en 0.
7) Se consulta por el WIM 8. Se espera como respuesta un 0.

```
Loading models from c:\source\examples\wimcheck.ma
Loading events from c:\source\examples\wimt35.ev
Message log: /dev/null
Output to: -
Starting simulation ... stop at ...
00:00:00:001 / wim8 / 1.000
00:00:00:002 / wim9 / 1.000
00:00:00:003 / cwp1 / 1.000
00:01:00:000 / cwp4 / 1.000
00:02:00:000 / cwp1 / 0.000
00:02:00:001 / cwp4 / 0.000
00:03:00:000 / cwp1 / 1.000
00:03:00:001 / cwp4 / 1.000
```

```
00:04:00:000 / wim8 / 0.000
00:04:00:001 / cwp4 / 0.000
00:00:00:043 res 1
00:02:00:041 res 0
00:03:00:041 res 1
00:04:00:041 res 0
Simulation ended!
```

En el último test, correspondiente al modelo Traplogic, se verifica el resultado obtenido luego de encender todos los bits de trap. Debido a esto, se espera obtener el índice que indique el trap de mayor prioridad pendiente. El resultado obtenido luego del tiempo de estabilización es correcto: (43 = 2Bh) corresponde a *Data Store Error*. Puede consultarse la tabla de prioridades en la descripción del modelo Traplogic para más información.

```
Starting simulation ... stop at ...
00:00:00:001 / inst_acc_excep / 1.000
00:00:00:002 / illeg_inst / 1.000
00:00:00:003 / priv_inst / 1.000
00:00:00:004 / win_over / 1.000
00:00:00:005 / win_under / 1.000
00:00:00:006 / addr_not_align / 1.000
00:00:00:007 / data_acc_excep / 1.000
00:00:00:008 / inst_acc_err / 1.000
00:00:00:009 / data_acc_err / 1.000
00:00:00:010 / div_zero / 1.000
00:00:00:011 / data_st_err / 1.000
00:00:00:012 / trap_inst / 1.000
00:00:00:052 tf 1
00:00:00:052 tt7 1
00:00:00:052 tt6 1
00:00:00:052 tt4 1
00:00:00:052 tt2 1
Simulation ended!
```

CONCLUSIONES Y TRABAJO A FUTURO

- El formalismo tomado para la simulación de estos componentes forma una sólida base que permite abstraer cualquier tipo de problema discretizable.
- La herramienta utilizada provee los mecanismos necesarios, facilitando la tarea y el enfoque sobre el problema a modelar, sin tener que preocuparse por el "código sucio".
- La reusabilidad de los componentes permite formar una biblioteca de los mismos con acceso a todos los investigadores que utilicen el mismo formalismo y herramienta, con todas las ventajas que ello implica.
- A futuro queda hacer acoplados los componentes simulados en este trabajo, dándole el papel de atómicos a las compuertas lógicas AND, NOT, OR, etc.
- El trabajo que se está llevando a cabo en la actualidad es el simulado de nuevos componentes de la arquitectura, como los que forman parte del subsistema de entrada/salida, para poder incorporarlos en una única unidad funcional que es la máquina en sí.

BIBLIOGRAFÍA Y REFERENCIAS

[**AG98**] Altman, D.; Glinsky, E. "Wimcheck, Traplogic, Regblock". Seminario de Simulación de Eventos Discretos. Departamento de Computación. Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires. 1998.

[**BBW98a**] Barylko, A.; Beyoglonián, J.; Wainer, G. "GAD: a General Application DEVS environment". Technical Report No. 98-001. Departamento de Computación. Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires. 1998.

[**BBW598**] Barylko, A.; Beyoglonián, J.; Wainer, G. "A General Application DEVS environment". Technical Report No. 98-005. Departamento de Computación. Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires. 1998.

[**DDTZW98**] Daicz, S.; Diuk, C.; Troccoli, A.; Zlotnik, S.; Wainer, G. “Arquitectura del Modelo”. Seminario de Simulación de Eventos Discretos. Departamento de Computación. Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires. 1998.

[**ER99**] Enrique, S.; Rubinstein D. “Mem, Mux4, Inc/Dec”. Seminario de Simulación de Eventos Discretos. Departamento de Computación. Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires. 1999. http://members.xoom.com/senrique/files

[**FRW97**] Ferrari, A.; Romano, S.; Wainer, G. “Diseño e Implementación de una Biblioteca de Lógica Digital”. Curso de Organización de Computadoras I. Departamento de Computación. Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires. 1997.

[**WG96**] Wainer, G. “Introducción a la Simulación de Eventos Discretos”. Technical Report No. 96-005. Departamento de Computación. Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires. 1996.

[**Zei76**] Zeigler, B. “Theory of modeling and simulation”. Wiley, 1976.