Información Tecnológica
Vol. 21(2), 91-102 (2010)
doi:10.1612/inf.tecnol.4167it.09

Diseño de Circuitos Digitales a Nivel de Registro empleando Diagramas ASM++

Santiago de Pablo[1], Santiago Cáceres[1], Jesús A. Cebrián[1] y Francisco Sanz[2]
(1) Universidad de Valladolid, Escuela de Ingenierías Industriales, Departamento de Tecnología Electrónica, Paseo de Cauce 59, 47011 Valladolid, España, (2) Universidad Europea Miguel de Cervantes, C/ Padre Julio Chevalier N° 2, 47012 Valladolid, España (e-mail: sanpab@eis.uva.es)

Resumen

Este artículo muestra la estrecha relación que existe entre los diagramas de estado algorítmicos (ASM charts) y los modernos lenguajes de descripción de circuitos, ambos empleados en el diseño de circuitos digitales. Se proponen sustanciales mejoras sobre la notación actual con el objetivo de desarrollar un compilador capaz de procesar automáticamente estos diagramas y generar el código VHDL o Verilog correspondiente. El uso de esta metodología facilita el aprendizaje del diseño de circuitos digitales a nivel de transferencia entre registros (RTL). El lenguaje gráfico propuesto es fácil de aprender y es entendido sin dificultad por estudiantes universitarios, quienes lo emplean como parte de la metodología de diseño para producir circuitos digitales sobre dispositivos reconfigurables tipo FPGA y CPLD.

Palabras clave: gráficos ASM, compilador, metodología de diseño, lenguajes de descripción, nivel de transferencia de registro

Digital Circuit Design at Register Transfer Level using ASM++ Charts

Abstract

This article shows the close relationship between Algorithmic State Machines (ASM charts) and modern hardware description languages, both applied to digital electronic design. Important improvements on current notation have been proposed in order to develop a compiler capable of processing these charts and generating VHDL or Verilog code automatically, The use of this methodology facilitates the learning of electronic design at the register transfer level (RTL). The language proposed is easy to learn and comprehend with no much difficulty by university students who used it as part of design methodology to produce digital circuits on reconfigurable devices of the type FPGA and CPLD.

Keywords: ASM charts, compiler, design methodology, description languages, register transfer level

INTRODUCCIÓN

El diseño de circuitos digitales puede realizarse empleando diferentes niveles de abstracción, cada uno con metodologías, técnicas y herramientas distintas. El nivel más utilizado y aceptado en la enseñanza universitaria para el diseño de circuitos digitales complejos es el conocido como "RTL" (en inglés, *Register Transfer Level*), en donde el diseñador ha de especificar todas las operaciones y transferencias entre registros que se suceden a lo largo de varios ciclos de reloj (Gajski, 1997; Pollán, 2008). En cierta medida se pierden de vista la geometría física y las conexiones concretas del circuito final, para centrar los esfuerzos de diseño en controlar únicamente su comportamiento ciclo a ciclo. Sin embargo, aunque las posibilidades de estos circuitos son enormes, muchos métodos empleados habitualmente para describir estos comportamientos son, o demasiado sencillos y limitados, o demasiado complejos y poco intuitivos. Esto dificulta el aprendizaje por parte del alumno y reduce la productividad del diseñador.

En este artículo se presenta una metodología de diseño basada en los diagramas de estado algorítmicos (ASM), documentados hace unos 35 años por Clare en 1973, que basó su trabajo en desarrollos previos de Osborne, de la Universidad de California en Berkeley (Clare, 1973). Desde entonces estos diagramas han sido ampliamente utilizados por diseñadores y docentes (Gajski, 1997; Roth, 2004; Pollán, 2008), aunque apenas han evolucionado: su primer uso se limitaba prácticamente a diseñar unidades de control (Brown, 1981; Baranov, 1997; Nixon, 1997; Bahill et al., 1998; Rizzi et al., 2007), más adelante han ido incorporando elementos más propios de la parte operativa de los circuitos (David y Bergeron, 2004), y finalmente se han aplicado como ayuda para desarrollar y documentar circuitos completos de elevada complejidad (Örs et al., 2003; Jang et al., 2006; Arnold y Vouzis, 2007; Soviani et al., 2009). De todas formas, hasta ahora sólo se han utilizado estos diagramas como una forma de materializar ideas, como una representación gráfica del comportamiento de los circuitos que, en todo caso, habría que diseñar manualmente empleando un esquema o un lenguaje de descripción de circuitos (HDL) (Chang, 1996).

La presente propuesta, denominada ASM++, modifica ligeramente y amplía la notación actual para hacerla más clara, flexible, intuitiva y productiva, resultando un lenguaje gráfico más consistente y fácil de entender. Esto ha permitido desarrollar un compilador que procesa automáticamente estos diagramas y genera los circuitos correspondientes empleando los lenguajes VHDL y Verilog.

METODOLOGÍAS DE DISEÑO 'RTL'

Un método ampliamente extendido para diseñar circuitos digitales de cierta complejidad es el uso de máquinas de estado (FSM, en inglés *Finite State Machine*), donde se representa con círculos los diversos estados por los que puede ir pasando el circuito con cada ciclo de reloj (véase la figura 1), y diversas flechas indican las posibles transiciones entre estados (Gajski, 1997; Wakerly, 2001; Roth, 2004). Las operaciones que ha de realizar el circuito se van anotando como comentarios cerca de los estados o, si son condicionales, de las transiciones entre estados. A veces también se distingue entre asignaciones síncronas, que se almacenarán sobre registros al terminar cada ciclo de reloj, y operaciones puramente combinacionales, asíncronas, que mostrarán su resultado durante el propio ciclo de reloj (Clare, 1973; Örs et al., 2003). Las primeras se denotan con una flecha ('←' ó '→'), para indicar que implican un cierto retraso (Clare, 1973), y las últimas con el operador de asignación ('=').

Cuando la función del circuito es principalmente numérica puede ser conveniente describir previamente el circuito como un algoritmo (Deschamps y Angulo, 1989; Pollán, 2008), es decir, como una secuencia de operaciones. El lenguaje matemático facilita la escritura de descripciones progresivamente más detalladas, y el proceso termina cuando las operaciones se pueden realizar con circuitos aritméticos o lógicos físicamente implementables. Mientras tanto, la secuencia se puede controlar con una máquina de estados o un secuenciador, que genera las señales de control apropiadas.

La figura 1 trata de ilustrar estas ideas mostrando un algoritmo (a la izquierda) y su correspondiente máquina de estados (a la derecha) para un circuito que multiplica dos números enteros positivos de doce bits: para ello, el circuito indica que está preparado para operar a través de una señal *'ready'*,

activa a nivel alto, y espera hasta recibir simultáneamente dos operandos a través de dos entradas *'inA'* e *'inB'* validadas por una señal externa *'go'*; a continuación ejecuta doce multiplicaciones parciales –que resultan ser sumas condicionales– y termina validando con una señal *'done'* el resultado mostrado en la salida *'outP'*. Este circuito es inicializado asíncronamente con una señal *'reset'*, activa a nivel alto, y sincronizado con los flancos ascendentes de una señal *'clk'*, no mostrada.

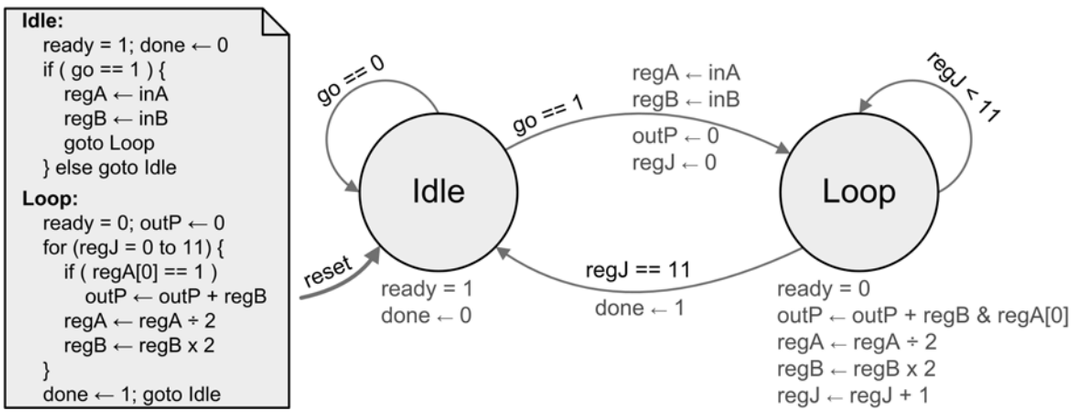


Fig. 1: Un circuito multiplicador de 12 bits como ejemplo de diseño a nivel RTL.

La máquina de estados de la figura 1 muestra de una forma muy intuitiva cuál será la evolución temporal del circuito, pero no expresa con suficiente claridad ni consistencia las operaciones que realiza. Al menos, en situaciones con muchas transiciones entre estados y muchas operaciones condicionales, estos diagramas se pueden volver confusos. El algoritmo, por su parte, permite manipular con rigor y consistencia las operaciones matemáticas necesarias para la ejecución de la tarea especificada, pero presenta limitaciones al tratar de detallar una secuencia concreta para los diversos cálculos implicados. Por este motivo se suelen emplear grafos que muestran la dependencia entre operandos y recursos de cálculo, lo que ayuda a distribuir temporalmente las operaciones (Deschamps y Angulo, 1989; Gajski, 1997).

Las máquinas de estado algorítmicas (ASM), como su propio nombre indica, presentan las características y ventajas de ambos métodos: manteniendo una representación gráfica intuitiva, expresan con detalle y de forma consistente todas las operaciones que ha de realizar el circuito en cada uno de los ciclos de reloj, como muestra claramente la figura 2. Para ello emplean tres tipos de cajas (Clare, 1973; Gajski, 1997; Roth, 2004): las rectangulares especifican el inicio de cada estado o ciclo de reloj y las operaciones incondicionales que se han de ejecutar durante ese periodo de tiempo; las cajas con forma de rombo o diamante permiten tomar decisiones y así modificar la línea de ejecución del algoritmo; por último, las cajas con forma ovalada muestran las operaciones que hay que realizar de forma condicional en cada ciclo, sólo si las decisiones anteriores lo permiten. En algunas situaciones, y para facilitar la interpretación del diagrama, se rodean todas las operaciones que se realizarán simultáneamente en cada ciclo de reloj con lo que se conoce como "bloque ASM"; en diseños complejos resulta casi imprescindible su uso.

Esta representación presenta algunas ventajas sobre los diagramas de estado convencionales: no sólo muestran y especifican la evolución entre estados (*'Idle'* y *'Loop'*, en este caso), sino que también detallan todas las operaciones que el circuito ha de realizar en cada estado y en cada transición entre estados; además, las condiciones se pueden crear de forma progresiva y más acorde con nuestra forma de pensar, aunque luego sean implementadas a través de funciones booleanas completas (Bahill et al., 1998).

Con todo, los diagramas ASM tradicionales tienen algunos inconvenientes:

En primer lugar, estos diagramas emplean un mismo tipo de caja, las rectangulares, para dos funciones completamente distintas y dispares: por una parte, indican que estamos ante un nuevo estado o ciclo de reloj, pero a la vez describen las operaciones incondicionales que se desea realizar

durante ese periodo de tiempo. Esta propiedad permite desarrollar diagramas muy compactos, pero impone al diseñador un orden muy estricto a la hora de especificar las operaciones, lo que en algunas situaciones resulta artificial e incluso molesto.

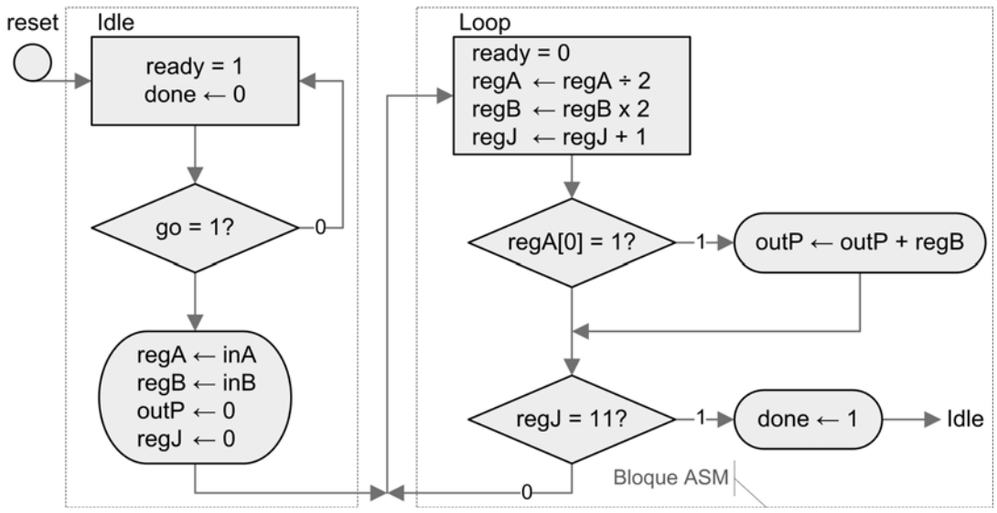


Fig. 2: Diagrama ASM tradicional correspondiente a un circuito multiplicador de 12 bits.

Precisamente por lo anterior, algunas veces es difícil diferenciar los límites de cada estado. En esos casos se añaden recuadros con líneas discontinuas (Clare, 1973) o incluso sombreados (Gajski, 1997), es decir, bloques ASM que envuelven todas las operaciones que se ejecutan de forma simultánea en cada estado. La propia existencia de los bloques ASM indica que estos diagramas no son realmente demasiado intuitivos, pues es necesario acudir a recursos adicionales para facilitar la interpretación de los propios diagramas.

Debido también al doble significado de las cajas rectangulares, las operaciones condicionales no pueden emplear el mismo tipo de caja que las incondicionales, pues indicarían una transición entre estados que no se desea. Esta característica de los diagramas tradicionales resulta confusa para los alumnos y dificulta de forma significativa la labor de edición por ordenador, pues durante el desarrollo de un circuito es frecuente la reordenación de las operaciones, que en este caso requiere también un cambio en la propia forma de las cajas. La única alternativa posible parece ser reservar las cajas rectangulares para los estados y emplear las cajas ovaladas para describir todas las operaciones (Örs et al., 2003; Arnold y Vouzis, 2007), tanto las condicionales como las incondicionales.

Además de todo lo anterior, y como se muestra claramente en la figura 2, el diseñador ha de emplear con frecuencia anotaciones fuera de las cajas para indicar el nombre de los estados, el proceso de inicialización del circuito, el valor por defecto de diversas señales e incluso los enlaces entre partes distantes de un diagrama. Estos textos no siguen una notación consistente (Soviani et al., 2009), pero han de ser tenidos en cuenta durante la posterior codificación manual del circuito.

Por último, pero especialmente importante, es la imposibilidad de especificar qué señales son internas y cuáles se corresponden con entradas o salidas, o también, cuántos bits emplea cada una de esas señales. Algunas herramientas comerciales como *StateCAD* o *HDL Designer* permiten introducir estos datos, pues sin estas especificaciones es imposible generar automáticamente un código HDL que permita simular y sintetizar el circuito, pero lo hacen fuera de los diagramas.

La nueva notación propuesta en este artículo trata de resolver todos estos problemas.

RESULTADOS Y DISCUSIÓN

En este artículo se propone una nueva notación, que se denomina “ASM++” por ampliar notablemente las posibilidades de los diagramas ASM tradicionales (Clare, 1973), en los que se basa

(Pablo et al., 2007a, 2007b, 2008). La primera y principal modificación que se introduce es el uso de una caja dedicada específicamente a denotar los estados: se ha elegido la forma oval (véase la parte derecha de la figura 3) por parecerse a los círculos empleados universalmente en las FSM y también por parecerse a las cajas que representan el inicio y el final de una subrutina en los diagramas de flujo de programación secuencial, pues aquí representan el inicio y el final de cada ciclo de reloj. Con ello, las cajas rectangulares quedan libres para introducir operaciones, tanto condicionales como incondicionales, indistintamente. Se han asignado las cajas rectangulares para describir todas las operaciones síncronas, que almacenarán sus resultados en biestables, y se han añadido unas cajas rectangulares con lados combados para especificar las operaciones asíncronas, que se materializarán empleando únicamente lógica combinacional. Por último, mantenemos las cajas con forma de rombo o diamante para las bifurcaciones, pues son comúnmente reconocidas y aceptadas.

Como muestra la mitad derecha de la figura 3, en la zona denominada “algorítmica”, el diagrama ASM++ es menos compacto que el analizado previamente en la figura 2, pero a la vez es más fácil de interpretar, se adapta mejor a la forma de pensar y a las decisiones que va tomando el diseñador, y se edita en un ordenador con mayor comodidad. Preguntados informalmente por esta cuestión, los alumnos se han mostrado absolutamente unánimes al preferir la nueva notación. Además, los diagramas ASM++ tienen otras propiedades interesantes que se enumeran a continuación.

En la nueva notación propuesta ya no son necesarios los denominados “bloques ASM”, porque los límites entre estados están ahora claramente definidos por las cajas de estado: transmiten de forma intuitiva la idea de que con ellas comienza y termina cada ciclo de reloj.

También han desaparecido todas las anotaciones laterales, excepto obviamente los comentarios: los nombres de los estados tienen un espacio reservado dentro de las cajas de estado; los enlaces entre estados o partes separadas de un diagrama se pueden realizar a través de las propias cajas de estado o con conectores –no incluidos en la figura– que simplemente indican la continuidad del diagrama sin introducir mayor funcionalidad.

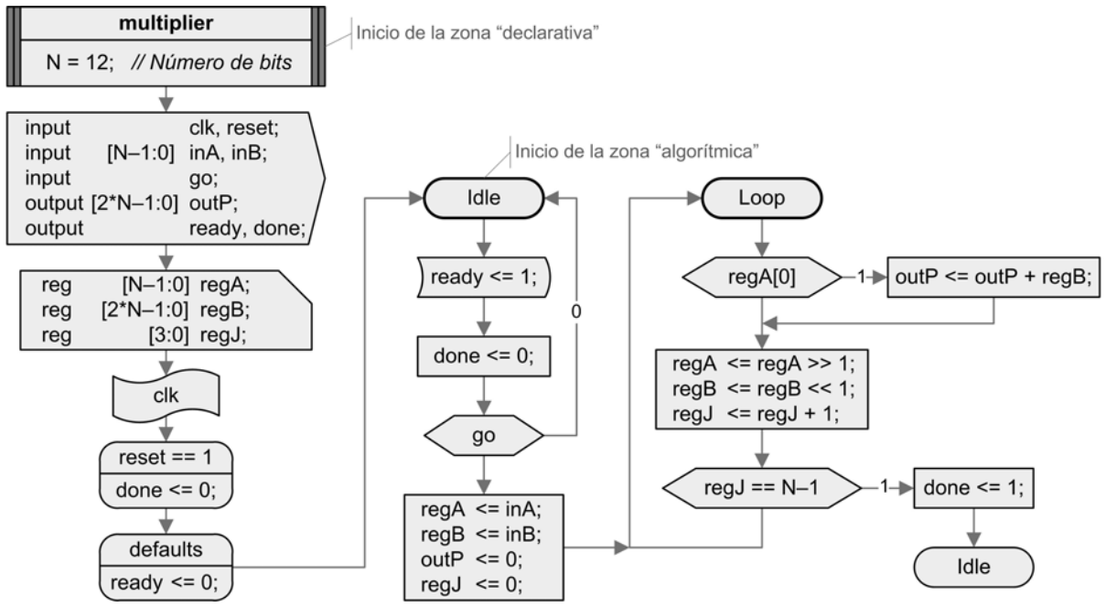


Fig. 3: Ejemplo de la notación ASM++ con cajas básicas y, a la izquierda, elementos adicionales.

El diseñador tiene ahora completa libertad para escribir las ecuaciones en el orden que prefiera, pues las operaciones incondicionales se pueden escribir sin ningún problema después de las condicionales (como ocurre en el estado *‘Loop’* de la figura 3).

Por último, todas las expresiones que aparecen en las cajas de operación y de condición emplean notación HDL estándar, o bien Verilog (como ocurre en este caso) o bien VHDL (véanse las figuras 6

y 7). Esta característica aumenta la productividad y ha resultado imprescindible para el desarrollo de un compilador de diagramas ASM++, actualmente operativo, que produce un código Verilog o VHDL –según el lenguaje empleado en las cajas– cuyo comportamiento es equivalente al especificado por el diagrama. Este código puede ser empleado inmediatamente para simular el circuito y también para sintetizarlo sobre dispositivos reconfigurables tipo FPGA o CPLD.

Tanto el diagrama de la figura 2 como la parte "algorítmica" del diagrama de la figura 3 describen la estructura interna del módulo multiplicador propuesto como ejemplo, pero no especifican nada acerca de su interfaz externo, ni sobre la anchura de sus señales, ni tampoco indican qué señal ha de ser empleada para sincronizar los registros de almacenamiento. Por tanto, si lo que se pretende es especificar completamente un circuito empleando estos diagramas, de modo que sirvan para algo más que como una primera aproximación abstracta a la funcionalidad que se desea implementar, resulta evidente que se necesitan más cajas para introducir nuevas funcionalidades. Como muestra la primera columna de la figura 3, que se corresponde con su zona "declarativa", la nueva notación ASM++ incorpora una serie de cajas que tienen una relación directa con los principales lenguajes de descripción de circuitos, pero que supera ampliamente las posibilidades de los diagramas ASM tradicionales.

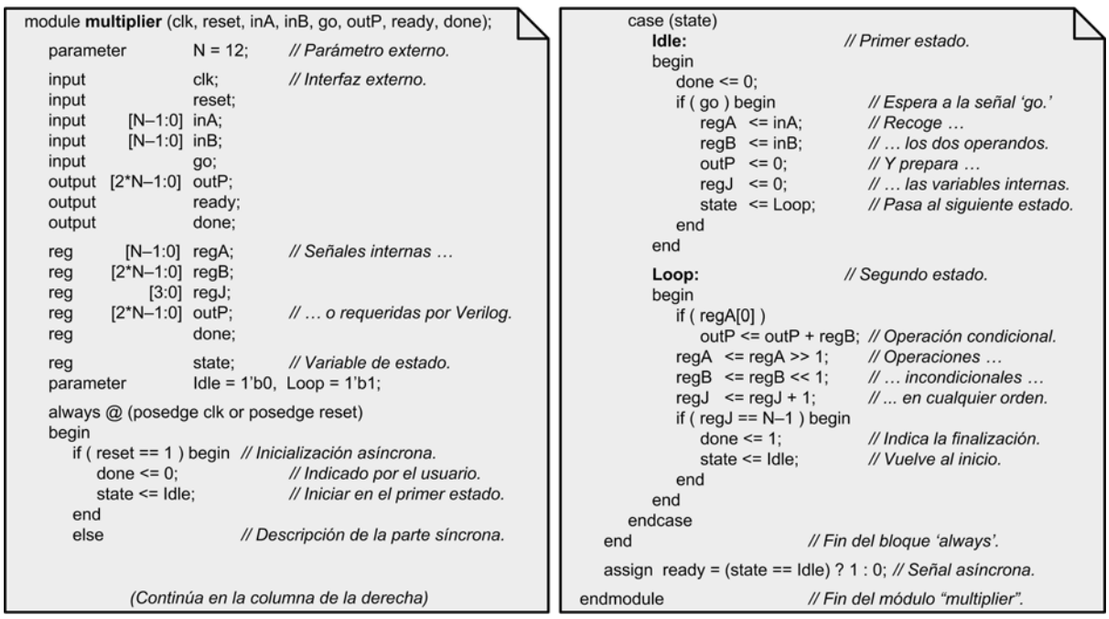
Empezando por la esquina superior izquierda de la figura, la primera caja especifica el nombre del diseño (*"multiplier"* en este caso) y asigna un valor por defecto a sus parámetros optativos (lo que se conoce como *"generics"* en VHDL y *"parameters"* en Verilog). Estos parámetros podrán ser modificados posteriormente cuando se inserte este módulo en un circuito de nivel superior (véase Pablo et al., 2008). Después se detallan las entradas y salidas de este módulo empleando código Verilog (las figuras 6 y 7 ilustran el uso de VHDL) dentro de una caja que hace referencia directa a esta funcionalidad. Por supuesto, el diseñador puede emplear varias cajas de este tipo si lo desea, facilitando así la agrupación de señales en función de su utilidad y mejorando la interpretación del diagrama. La tercera caja se emplea para introducir código HDL genérico y directivas del compilador (para especificar el lenguaje de entrada en caso de no ser reconocido, por ejemplo). En este caso describe únicamente las señales internas necesarias para la ejecución del algoritmo, es decir, varios registros que almacenarán los datos. Las variables que contienen el estado, sin embargo, se generan de forma automática y no es necesario especificarlas.

A continuación, una pequeña caja indica que la señal *'clk'* sincronizará todo el circuito. Otra caja con bordes redondeados detalla la secuencia de inicialización, tanto de la máquina de estados como de otras señales, en este caso la salida *'done'*, que se ejecutará de forma asíncrona cuando se cumpla la condición expresada, es decir, cuando la señal *'reset'* se active a nivel alto. Por último, hemos añadido una caja que permite asignar valores por defecto tanto a las señales internas como a las salidas: cuando una señal registrada no se modifica en alguno de los estados del circuito, se espera que los biestables correspondientes retengan su valor anterior; las señales asíncronas combinacionales, en cambio, no pueden aportar esa funcionalidad, así que hemos optado por asumir para ellas un valor indistinto, de modo que se simplifiquen las funciones lógicas resultantes. La caja con el texto *'defaults'* permite modificar de forma fácil e intuitiva estas especificaciones y simplifica en muchos casos la edición de los diagramas. En este caso define el valor asíncrono de la señal *'ready'* para todos los estados distintos de *'Idle'*.

La transcripción manual del diagrama ASM++ de la figura 3 a código HDL en sencilla, inmediata y de gran valor didáctico, como se observa en la figura 4. El código generado automáticamente por el compilador que hemos desarrollado es un poco distinto, pues siempre crea máquinas de estado con un biestable por estado (lo que se conoce como codificación *"one-hot") y* se centra más en la lógica correspondiente a cada señal individual (véase el código de la figura 7), pero en cualquier caso resulta equivalente desde el punto de vista de la simulación y, en gran medida, de la síntesis.

Gracias a estas nuevas cajas ya es posible describir completamente los circuitos RTL síncronos y centrarse únicamente en el comportamiento deseado para cada una de las señales en cada uno de los ciclos de reloj. Cuando el diseñador dibuja los diagramas empleando una herramienta adecuada, como *MS Visio* o *ConceptDraw*, y graba los ficheros empleando el formato VDX, que internamente es XML, el compilador lee y procesa rápidamente los diagramas. En cualquier momento el diseñador

puede verificar si el código HDL generado se corresponde con la estructura y/o el comportamiento deseados, y también puede modificar el diagrama buscando mejorar el tamaño, la velocidad o el consumo. Resulta muy fácil y cómodo probar diversas alternativas y comparar los resultados producidos por las herramientas de simulación y síntesis de circuitos.



```
module multiplier (clk, reset, inA, inB, go, outP, ready, done);
   parameter           N = 12;      // Parámetro externo.
   input               clk;         // Interfaz externo.
   input               reset;
   input      [N–1:0]  inA;
   input      [N–1:0]  inB;
   input               go;
   output [2*N–1:0]    outP;
   output              ready;
   output              done;
   reg        [N–1:0]  regA;        // Señales internas …
   reg      [2*N–1:0]  regB;
   reg            [3:0] regJ;
   reg      [2*N–1:0]  outP;        // … o requeridas por Verilog.
   reg                 done;
   reg                 state;       // Variable de estado.
   parameter           Idle = 1'b0, Loop = 1'b1;
   always @ (posedge clk or posedge reset)
   begin
      if ( reset == 1 ) begin  // Inicialización asíncrona.
         done <= 0;            // Indicado por el usuario.
         state <= Idle;        // Iniciar en el primer estado.
      end
      else                     // Descripción de la parte síncrona.

            (Continúa en la columna de la derecha)
```

```
      case (state)
         Idle:                    // Primer estado.
         begin
            done <= 0;
            if ( go ) begin       // Espera a la señal 'go.'
               regA  <= inA;      // Recoge …
               regB  <= inB;      // … los dos operandos.
               outP  <= 0;        // Y prepara …
               regJ  <= 0;        // … las variables internas.
               state <= Loop;     // Pasa al siguiente estado.
            end
         end
         Loop:                    // Segundo estado.
         begin
            if ( regA[0] )
               outP <= outP + regB; // Operación condicional.
            regA  <= regA >> 1;   // Operaciones …
            regB  <= regB << 1;   // … incondicionales …
            regJ  <= regJ + 1;    // ... en cualquier orden.
            if ( regJ == N–1 ) begin
               done <= 1;         // Indica la finalización.
               state <= Idle;     // Vuelve al inicio.
            end
         end
      endcase
   end                          // Fin del bloque 'always'.
   assign  ready = (state == Idle) ? 1 : 0; // Señal asíncrona.
endmodule                       // Fin del módulo "multiplier".
```

Fig. 4: Código Verilog correspondiente al diagrama ASM++ de la figura 3.

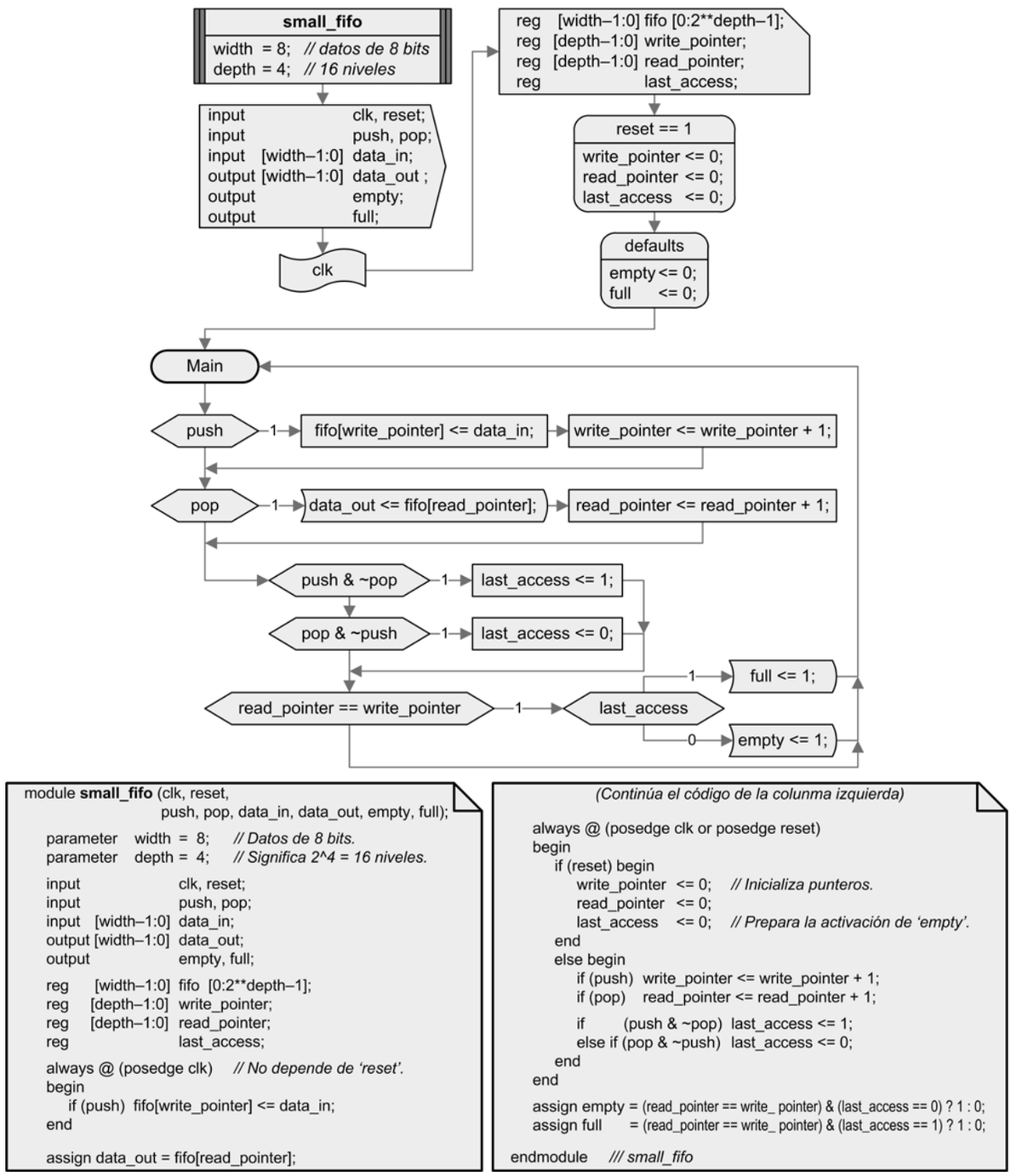
Para poder apreciar mejor la simbiosis entre estos diagramas y los lenguajes HDL, la figura 5 muestra un ejemplo de un diagrama ASM++ que se corresponde con una pequeña memoria tipo FIFO, que almacena varios valores y los va entregando posteriormente en el mismo orden que los ha recibido. Puede recibir órdenes simultáneas de escritura (validadas con *'push'*) y de lectura (indicadas con *'pop'*), e informa de su estado a través de dos señales, una que indica que la memoria está vacía (*'empty'*) y por tanto no debe ser leída, y otra que indica que la memoria está llena (*'full'*) y no puede recibir órdenes de escritura sin perder datos previos.

El código Verilog completo correspondiente a este diagrama, casi idéntico al generado automáticamente por el compilador, se muestra en la parte inferior de la propia figura. Este ejemplo muestra claramente que de cada diagrama ASM++ se pueden extraer hasta tres bloques de código distintos: el primero es sensible a las señales *'clk'* y *'reset'*; e incluiría en este caso a las señales *'write_pointer'*, *'read_pointer'* y *'last_access'*; el segundo bloque es sensible únicamente a la señal de sincronización *'clk'*, por lo que en este caso se correspondería únicamente con la señal *'fifo'*; el último bloque estaría formado por todas las señales asíncronas, es decir, *'data_out'*, *'empty'* y *'full'*. El diferente comportamiento de unas y otras señales justifica el uso de cajas distintas: esto no sólo facilita el análisis de los diagramas para evaluar su funcionamiento, sino que también ayuda durante la fase de escritura manual del código (Chang, 1996; Bahill et al., 1998).

Se destaca también que durante la construcción de estos diagramas, el diseñador se centra principalmente en la funcionalidad que desea conseguir y no en el circuito que se va a generar. Sin embargo, la disposición relativa de las cajas podrá tener un impacto notable sobre el circuito sintetizado o, al contrario, será indiferente gracias a posibles simplificaciones. En este caso, por ejemplo, la señal asíncrona *'data_out'* puede ser simplificada −eliminando su dependencia con la señal *'pop'*− al no estar definido su comportamiento en aquellos ciclos en los que no se retiran datos. El compilador es capaz de detectar muchas de estas situaciones y aplica automáticamente las simplificaciones correspondientes.

La figura 6 muestra otros dos diagramas ASM++ que describen un total de cuatro multiplexores, empleando el lenguaje VHDL en el diagrama de la izquierda y Verilog en el de la derecha. Además,

el primer diagrama genera multiplexores puramente combinacionales, con salida asíncrona, mientras el segundo sincroniza las salidas con una señal de reloj. Estos diagramas ilustran el uso de cajas de decisión múltiple para expresar la lógica de las señales *'mux1a'* y *'mux2a'* e introducen dos cajas compactas que describen una funcionalidad idéntica a las anteriores para *'mux1b'* y *'mux2b'*. Este ejemplo muestra claramente la conveniencia de utilizar formas distintas para dos comportamientos tan distintos, el síncrono y el asíncrono, y la ventaja adicional que supone haber liberado las cajas rectangulares de su función tradicional, que era especificar el comienzo de cada estado.




```
module small_fifo (clk, reset,
                   push, pop, data_in, data_out, empty, full);
   parameter   width = 8;    // Datos de 8 bits.
   parameter   depth = 4;    // Significa 2^4 = 16 niveles.

   input                clk, reset;
   input                push, pop;
   input  [width–1:0]   data_in;
   output [width–1:0]   data_out;
   output               empty, full;

   reg    [width–1:0]   fifo  [0:2**depth–1];
   reg    [depth–1:0]   write_pointer;
   reg    [depth–1:0]   read_pointer;
   reg                  last_access;

   always @ (posedge clk)    // No depende de 'reset'.
   begin
      if (push)  fifo[write_pointer] <= data_in;
   end

   assign data_out = fifo[read_pointer];
```

```
                    (Continúa el código de la columna izquierda)

   always @ (posedge clk or posedge reset)
   begin
      if (reset) begin
         write_pointer  <= 0;    // Inicializa punteros.
         read_pointer   <= 0;
         last_access    <= 0;    // Prepara la activación de 'empty'.
      end
      else begin
         if (push)  write_pointer <= write_pointer + 1;
         if (pop)   read_pointer <= read_pointer + 1;

         if       (push & ~pop)  last_access <= 1;
         else if (pop & ~push)   last_access <= 0;
      end
   end

   assign empty = (read_pointer == write_ pointer) & (last_access == 0) ? 1 : 0;
   assign full  = (read_pointer == write_ pointer) & (last_access == 1) ? 1 : 0;

endmodule    /// small_fifo
```

Fig. 5: Un diagrama ASM++ que describe una memoria tipo FIFO (arriba) y su código Verliog (abajo).

Adicionalmente, los tres diseños de las figura 5 y 6 tienen una peculiaridad interesante: como sólo tienen una caja de estado, no tendrán que ejecutar ningún tipo de secuencia, por lo que no

necesitarán de hecho ninguna máquina de estados. Sin embargo, la caja de estado sigue facilitando la representación del comportamiento de estos circuitos y la interpretación de su modo de funcionar cíclico y repetitivo, gobernado por una señal de reloj en el caso de que contengan señales síncronas.

A diferencia de los lenguajes de programación secuenciales, tanto en los diagramas ASM como en los lenguajes HDL es posible describir múltiples operaciones que se ejecutan en paralelo. De hecho, todas las operaciones especificadas en cada estado, en caso de llegar a ejecutarse, se realizan simultáneamente, con independencia de que se incluyan antes o después dentro del estado. Ése es el sentido de las máquinas de estado: ejecutan operaciones siguiendo una secuencia, pero permiten describir y ejecutar múltiples operaciones en cada uno de los ciclos de trabajo.

Sin embargo, en los circuitos de la vida real con frecuencia nos encontramos con situaciones en las que una única máquina de estados o una única línea de ejecución resulta insuficiente (Örs et al., 2003) o conduce a una solución muy compleja (Baranov, 1997). Es necesario describir señales con comportamientos tan distintos que sería conveniente disponer en un mismo circuito de la posibilidad de incorporar varias líneas de ejecución independientes, cada una siguiendo su propia secuencia de operaciones, e incluso su propia sincronización. Los diagramas ASM++ incorporan esta posibilidad, como muestra la figura 7, que incluye una versión simplificada de una FIFO con dos relojes, uno para la recepción de datos y otro para su emisión. Este diseño emplea VHDL como lenguaje de entrada, y la salida, expresada en el mismo lenguaje, se muestra en la parte inferior de la misma figura. En este caso, además, se ha incluido el código literal −salvo los comentarios− generado automáticamente por el compilador.

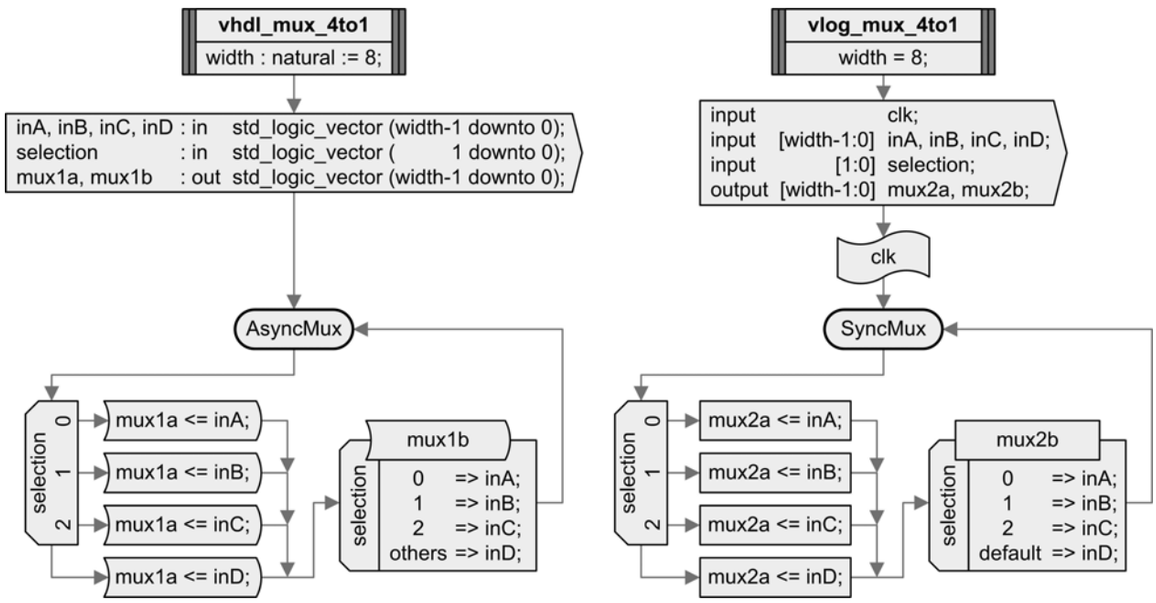
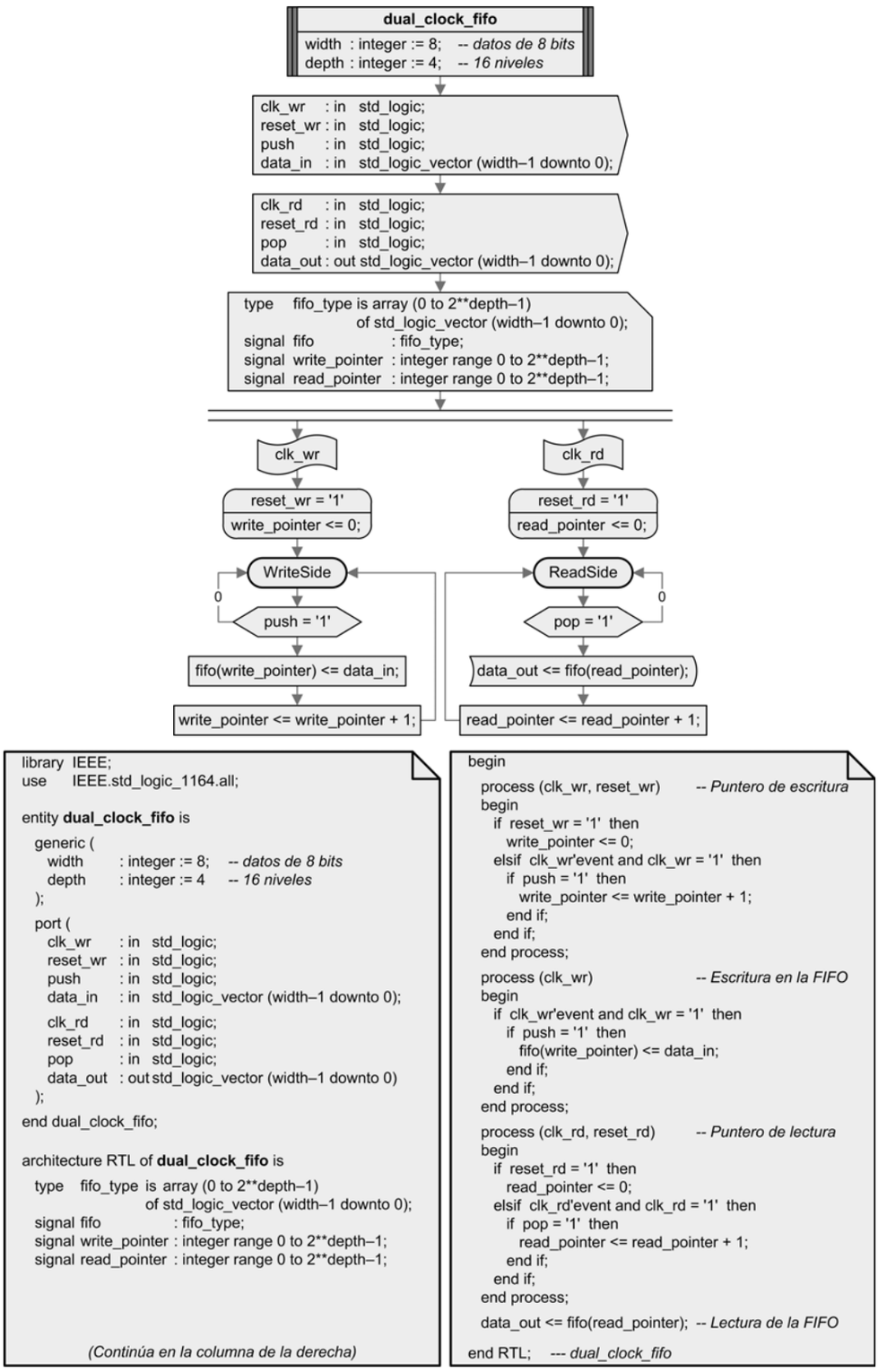


Fig. 6: Diagramas ASM++ que describen multiplexores asíncronos (izquierda) y síncronos (derecha).

De todo lo anterior se deduce que el nivel de abstracción de los diagramas ASM++ es equivalente al aportado por los principales lenguajes de descripción de circuitos, VHDL y Verilog, y sus posibilidades también son similares. Sin embargo, la forma de representar el comportamiento deseado es más cercana a nuestra forma de pensar, lo que permite abordar, incluso en menos tiempo, diseños de mayor complejidad. Además, y esto es importante desde el punto de vista didáctico, estos diagramas limitan en cierta medida las posibilidades del diseñador, sin perder flexibilidad, pues emplean únicamente aquellas estructuras de los lenguajes HDL que se han demostrado sólidas y libres de riesgos, de modo que así se asegura la generación de circuitos robustos.




```
library IEEE;
use     IEEE.std_logic_1164.all;

entity dual_clock_fifo is
  generic (
    width      : integer := 8;    -- datos de 8 bits
    depth      : integer := 4     -- 16 niveles
  );
  port (
    clk_wr     : in  std_logic;
    reset_wr   : in  std_logic;
    push       : in  std_logic;
    data_in    : in  std_logic_vector (width–1 downto 0);
    clk_rd     : in  std_logic;
    reset_rd   : in  std_logic;
    pop        : in  std_logic;
    data_out   : out std_logic_vector (width–1 downto 0)
  );
end dual_clock_fifo;

architecture RTL of dual_clock_fifo is
  type   fifo_type is array (0 to 2**depth–1)
                   of std_logic_vector (width–1 downto 0);
  signal fifo          : fifo_type;
  signal write_pointer : integer range 0 to 2**depth–1;
  signal read_pointer  : integer range 0 to 2**depth–1;

          (Continúa en la columna de la derecha)
```

```
begin
  process (clk_wr, reset_wr)       -- Puntero de escritura
  begin
    if  reset_wr = '1'  then
      write_pointer <= 0;
    elsif  clk_wr'event and clk_wr = '1'  then
      if  push = '1'  then
        write_pointer <= write_pointer + 1;
      end if;
    end if;
  end process;
  process (clk_wr)                 -- Escritura en la FIFO
  begin
    if  clk_wr'event and clk_wr = '1'  then
      if  push = '1'  then
        fifo(write_pointer) <= data_in;
      end if;
    end if;
  end process;
  process (clk_rd, reset_rd)       -- Puntero de lectura
  begin
    if  reset_rd = '1'  then
      read_pointer <= 0;
    elsif  clk_rd'event and clk_rd = '1'  then
      if  pop = '1'  then
        read_pointer <= read_pointer + 1;
      end if;
    end if;
  end process;
  data_out <= fifo(read_pointer); -- Lectura de la FIFO
end RTL;    --- dual_clock_fifo
```

Fig. 7: Un diagrama ASM++ con dos ramas paralelas (arriba) y su código VHDL (abajo).

CONCLUSIONES

Este artículo ha presentado una metodología gráfica, muy potente a la vez que intuitiva y fácil de usar, que facilita la representación del comportamiento de los diseños digitales a nivel de registro, lo que es especialmente útil en el proceso de enseñanza de la electrónica digital. Se basa en las

conocidas máquinas de estado algorítmicas (ASM), pero incrementa y actualiza sus posibilidades empleando recursos comunes y robustos de los lenguajes de descripción de circuitos, en particular VHDL y Verilog.

La introducción de estos diagramas, denominados ASM++, es fácil y cómoda a través de múltiples programas gráficos como *MS Visio* o *ConceptDraw*, empleando en cualquier caso el formato VDX. El compilador programado en C++ para este lenguaje gráfico recoge todos los ficheros solicitados, los procesa si han sido modificados, y genera automáticamente uno o varios ficheros VHDL o Verilog, según el lenguaje empleado por los diagramas. De este modo es posible simular y sintetizar el circuito inmediatamente, sin introducir manualmente ninguna modificación. Todos los ficheros intermedios se pueden editar y revisar para verificar el correcto funcionamiento de la herramienta o, habitualmente, para probar diversas alternativas de descripción buscando reducir el consumo o mejorar la síntesis en términos de área o velocidad.

El lenguaje gráfico propuesto es fácil de aprender y es entendido sin dificultad por estudiantes universitarios, tanto de nivel básico como de nivel avanzado, que lo emplean como parte de su metodología de diseño para producir circuitos digitales sobre dispositivos reconfigurables tipo FPGA y CPLD. También es empleado para el desarrollo de módulos, para la supervisión de diseños y para la documentación de los resultados finales en proyectos realizados desde la Universidad para empresas externas.

AGRADECIMIENTOS

A la financiación aportada para estos desarrollos por eZono AG, empresa ubicada en Jena, Alemania, por ISEND SA, empresa ubicada en el Parque Tecnológico de Boecillo, Valladolid, España y por el Ministerio de Educación y Ciencia de España, incluyendo fondos FEDER (referencia CICYT ENE2007-67417/ALT).

REFERENCIAS

Arnold, M.G. y P.D. Vouzis.; “A Serial Logarithmic Number System ALU”, Actas de 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools, 151-156, Lübeck, Alemania, 29 a 31 de agosto (2007).

Bahill, A.T. y otros nueve autores; *The design-methods comparison project*, IEEE Transactions on Systems, Man and Cybernetics, Part C: Applications and Reviews: 28(1), 80-103 (1998).

Baranov, S.; “Synthesis of control units for mobile robots”, Actas de 2nd Euromicro Workshop on Advanced Mobile Robots, 80-86, Brescia, Italia, 22 a 24 de octubre (1997).

Brown, D.W.; “State-Machine Synthesizer - SMS”, Actas de 18th Design Automation Conference (DAC 1981), 301-305, Nashville, Tennessee, USA, junio (1981).

Chang, M.; “Teaching Top-down Design Using VHDL and CPLD”, Actas de 26th Annual Conference on Frontiers in Education Conference (FIE'96), vol. 2, 514-517, Salt Lake City, Utah, USA, 6 a 9 de noviembre (1996).

Clare, C.R.; *Designing Logic Systems Using State Machines*, McGraw-Hill, Nueva York, USA (1973).

David, J.P. y E. Bergeron; “A Step towards Intelligent Translation from High-Level Design to RTL”, Actas de 4th IEEE International Workshop on System-on-Chip for Real-Time Applications, 183-188, Banff, Alberta, Canadá, 19 a 21 de julio (2004).

Deschamps, J.P. y J.M. Angulo; *Diseño de sistemas digitales: metodología moderna,* Paraninfo, Madrid, España (1989).

Gajski, D.D.; *Principios de diseño digital*, Prentice Hall Iberia, Madrid, España (1997).

Jang, J., Kang, H. y Jeon, H.; *The FPGA implementation of the RC-DBA algorithm in the EPON network*, International Journal of Computer Science and Network Security: 6(6), 44-51 (2006).

Nixon, M.; *On a Programmable Approach to Introducing Digital Design*, IEEE Transactions on Education: 40(3), 195-206 (1997).

Örs, S.B., L. Batina, B. Preneel y J. Vandewalle; "Hardware Implementation of an Elliptic Curve Processor over GF(p)", Actas de Application-Specific Systems, Architectures and Processors (ASAP'03), 1-11, La Haya, Holanda, 24 a 26 de junio (2003).

Pablo, S. de, S. Cáceres, J.A. Cebrián y M. Berrocal; "A proposal for ASM++ diagrams", Actas de 10th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems, 121-124, Cracovia, Polonia, 11 a 13 de abril (2007a).

Pablo, S. de, S. Cáceres, J.A. Cebrián y M. Berrocal; "Application of ASM++ methodology on the design of a DSP processor", Actas de 4th FPGAworld Conference, 13-19, Estocolmo, Suecia, 13 de septiembre (2007b).

Pablo, S. de, S. Cáceres, J.A. Cebrián, M. Berrocal y F. Sanz; "Los diagramas ASM++ como herramienta aplicada en la enseñanza de la Electrónica Digital", Actas de VIII Congreso de Tecnologías Aplicadas a la Enseñanza de la Electrónica (TAEE 2008), 1-12, Zaragoza, España, 2 a 4 de julio (2008).

Pollán, T.; *Electrónica Digital*, 3ª edición, Prensas Universitarias de Zaragoza, Zaragoza, España (2008).

Rizzi, M., M. Aloia y B. Castagnolo; *Synthesis of Finite State Machine Adopting the Controller-data Path Approach: Performance Evaluation of Different Methods*, Information Technology Journal: 6(6), 843-850 (2007).

Roth, C.H. Jr.; *Fundamentos de Diseño Lógico*, 5ª edición, International Thomson Editores Spain, Madrid, España (2004).

Soviani, C., I. Hadzic y Edwards, S.A.; *Synthesis and Optimization of Pipelined Packet Processors*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems: 28(2), 231-244 (2009).

Wakerly, J.F.; *Diseño digital: principios y prácticas,* 3ª edición, Pearson Educación, México (2001).