LOS DIAGRAMAS ASM++ COMO HERRAMIENTA APLICADA EN LA ENSEÑANZA DE LA ELECTRÓNICA DIGITAL

S. DE PABLO[1], S. CÁCERES[1], J.A. CEBRIÁN[1], M. BERROCAL[2] y F. SANZ[3]
[1] *Departamento de Tecnología Electrónica, Universidad de Valladolid, Valladolid, España.*
[2] *eZono AG, Jena, Alemania.*
[3] *Universidad Europea Miguel de Cervantes, Valladolid, España.*
sanpab@eis.uva.es

Las máquinas de estado algorítmicas (ASM) son una herramienta de diseño digital que mejora algunas propiedades de los diagramas de estado, gracias a que describen con detalle no sólo las transiciones entre estados, sino también las operaciones que se realizan en ellos. Este artículo muestra la estrecha relación que existe entre los diagramas ASM y los modernos lenguajes de descripción de circuitos, y propone sustanciales mejoras sobre la notación actual para especificar completamente los diseños. Gracias a los cambios introducidos hemos desarrollado un compilador que procesa uno o varios diagramas y genera automáticamente el código VHDL o Verilog correspondiente, lo que permite simular y sintetizar sobre una FPGA los circuitos descritos de forma sencilla, cómoda y robusta. Consideramos que el uso de los diagramas propuestos, denominados ASM++, facilita el aprendizaje de la electrónica digital.

1. Introducción

Las máquinas de estado algorítmicas, también conocidas como diagramas ASM, fueron documentadas hace 40 años por Clare [1], quien trabajaba en *Electronics Research Laboratory* de *Hewlett Packard Labs*. Su libro se basó en los desarrollos previos de Osborne en la Universidad de California en Berkeley [2]. Desde entonces, los diagramas ASM se han aplicado con frecuencia para diseñar circuitos digitales que requieren la realización de tareas complejas [3,4]. Muchos textos de electrónica digital que describen el diseño a nivel de registro (RTL) dedican una atención especial a los diagramas ASM junto a otros métodos, en particular las máquinas de estados finitas (FSM), los diagramas de estado y las tablas de estado [5-7]. Sin embargo, después de un análisis más detallado, nos encontramos con que estos recursos de diseño prácticamente sólo se utilizan de forma marginal, y únicamente para definir las partes de control de los circuitos algorítmicos [8-11]. Realmente pocos autores emplean los diagramas ASM para desarrollar circuitos completos, excepto [12,13], que incrementan las posibilidades de estos diagramas con recursos más propios de lenguajes de alto nivel, pero finalmente codifican el circuito empleando texto.

Los diagramas ASM son una buena alternativa a los diagramas de estado porque, manteniendo su interfaz gráfico e intuitivo, permiten definir de forma más cómoda y consistente las transiciones entre estados y también las operaciones que se han de realizar durante y al final de cada estado. Sin embargo, muchos autores consideran que su interfaz gráfico es poco práctico [14]; posiblemente por ese motivo, en el diseño RTL se han impuesto los lenguajes de descripción de circuitos (HDL), pues su edición es en principio mucho más fácil y cómoda.

En el proceso de formación universitaria en Electrónica Digital, nuestros estudiantes comienzan usando herramientas de captura de esquemas para simular circuitos combinacionales y secuenciales, luego aprenden a diseñar circuitos algorítmicos y más tarde se enfrentan a situaciones más complejas empleando lenguajes VHDL y Verilog. En nuestra opinión, pensamos que las FSM son útiles para comprender de

forma genérica el comportamiento de un circuito, pero tienen muchas limitaciones para realizar descripciones detalladas. En cambio, hemos experimentado cómo los diagramas ASM tienen muchas ventajas durante el proceso de formación en electrónica digital [15], y también en el desarrollo y producción de circuitos digitales complejos [8], porque representan de forma muy directa el comportamiento del circuito, permiten materializar ideas y probar alternativas. Durante la fase de descripción detallada de un diseño, cuando la relación entre las diversas tareas se vuelve confusa, estos diagramas resultan particularmente útiles al clarificar el orden de las operaciones y sus interacciones.

Sin embargo, pensamos que la actual notación ASM tiene limitaciones importantes y muchas veces no es suficiente para describir los circuitos de la vida real, que suelen necesitar incluso varias líneas de ejecución funcionando en paralelo. Este artículo presenta una nueva notación, denominada "ASM++", que trata de mejorar la actual para aplicarla en diseños más complejos. Actualmente disponemos de un compilador capaz de generar automáticamente el código HDL correspondiente a los diagramas mostrados en este artículo y otros más complejos, y de momento reconoce más de 25 elementos distintos.

2. Diagramas ASM tradicionales

Los diagramas ASM clásicos son un conjunto de cajas enlazadas que describen las acciones que ha de realizar el circuito en cada ciclo de reloj. Emplean tres tipos de cajas: en primer lugar, las cajas 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. Adicionalmente se define un "bloque ASM", opcional, que incluye todas las operaciones, condicionales e incondicionales que han de ejecutarse de forma simultánea en cada ciclo de reloj. En diseños complejos resulta casi imprescindible el uso de los bloques ASM.

La figura 1 trata de ilustrar todas estas ideas mostrando el diagrama ASM tradicional de un circuito que multiplica dos números enteros de 12 bits sin signo: para ello espera hasta recibir simultáneamente los dos operandos a través de dos entradas *'inA'* e *'inB'* validadas por una señal *'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 es sincronizado por una señal *'clk'* no mostrada en el diagrama.

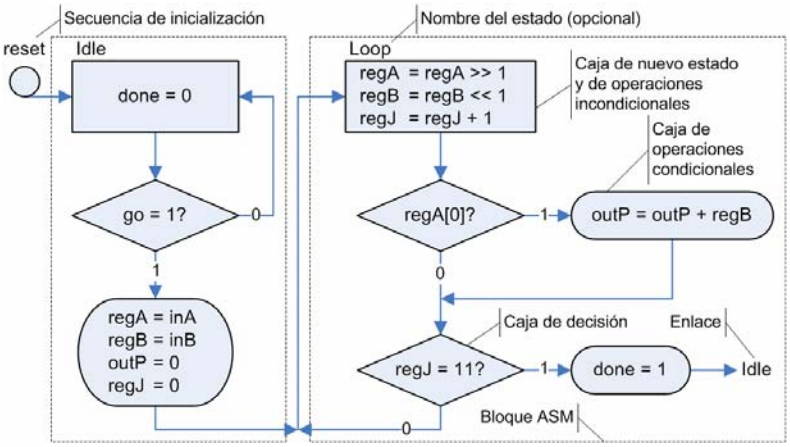

Figura 1. Un ejemplo de diagrama ASM tradicional.

Las ventajas de esta representación sobre los diagramas de estado convencionales (FSM) son evidentes: no sólo se muestra y especifica la evolución entre estados (*'Idle'* y *'Loop'*, en este caso), sino que también se incluyen 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 incremental, aunque luego serán implementadas a través de funciones booleanas completas [8]. En todo caso, pensamos que estos diagramas ASM tradicionales tienen algunas desventajas:

- Emplean un mismo tipo de caja, las rectangulares, para dos funciones completamente distintas: por una parte indican que estamos ante un nuevo estado o ciclo de reloj, pero a la vez especifican qué operaciones se han de realizar durante ese periodo de forma incondicional. Esta propiedad permite desarrollar diagramas muy compactos, pero impone al diseñador un orden a la hora de especificar las operaciones que resulta artificial e incluso molesto.

- Precisamente por lo anterior, algunas veces es difícil diferenciar los límites de cada estado. En esos casos se emplean cajas con líneas discontinuas que envuelven las operaciones de cada estado (lo que se conoce como "bloques ASM") o incluso se emplean diferentes colores para las cajas de los diferentes estados. 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 ASM tradicionales dificulta de forma significativa la labor de edición por ordenador, pues durante el desarrollo de un circuito es frecuente la reordenación de operaciones, que en este caso requiere también el 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, tanto condicionales como incondicionales.

- Además de todo lo anterior, y como se muestra claramente en la figura 1, 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, e incluso los enlaces entre partes distantes de un diagrama, que no tienen definida una forma estándar y simplemente se dejan como un texto que ha de ser interpretado por el diseñador durante la 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. Sin estas especificaciones es imposible generar automáticamente un código HDL que permita simular y sintetizar el circuito.

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

3. Características básicas de los diagramas ASM++

La primera y principal modificación que proponemos con esta nueva notación, que denominaremos ASM++ por ampliar notablemente las posibilidades de los diagramas tradicionales [16], es el uso de una caja dedicada específicamente a denotar los estados. Hemos elegido la forma oval (véase la figura 2) por parecerse a los círculos empleados por los diagramas de estado, y entonces dejamos las cajas rectangulares para introducir todas las operaciones síncronas, tanto incondicionales como condicionales. Mantenemos las cajas con forma de diamante para las bifurcaciones, pues son comúnmente reconocidas y aceptadas.

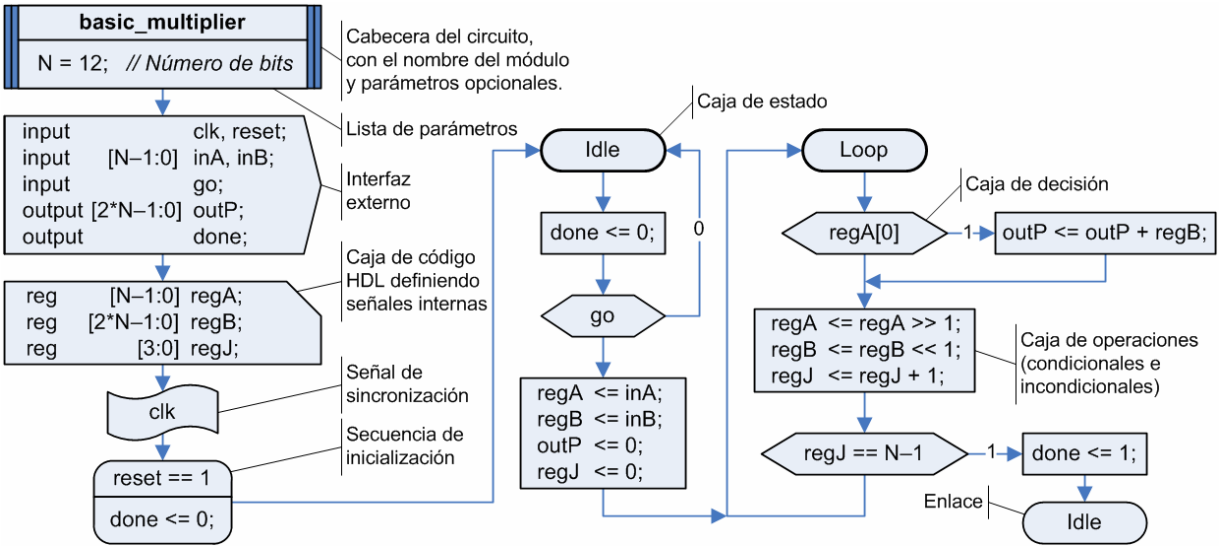

Figura 2. Un ejemplo de la nueva notación ASM++ con cajas básicas y, a la izquierda, elementos adicionales.

Como muestra la mitad derecha de la figura 2, el diagrama resultante es menos compacto que el de la figura 1, pero a la vez es más fácil de interpretar, se edita con mayor comodidad y se adapta mejor a las decisiones que va tomando el diseñador. Preguntados por esta cuestión, nuestros alumnos se han mostrado unánimes al preferir la nueva notación. Además tiene otras propiedades interesantes:

- 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.
- 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 –circulares o con forma de flecha, no incluidos en la figura– que simplemente realizan la función de enlace sin introducir mayor funcionalidad.
- 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 2).
- 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 (véase la figura 5). Esta característica ha resultado imprescindible para el desarrollo de un compilador de diagramas ASM++, actualmente operativo: el resultado es un código Verilog o VHDL –según el lenguaje empleado en las cajas– cuyo comportamiento es equivalente al especificado por el diagrama, que puede ser empleado inmediatamente para simular y sintetizar el circuito.

La trascripción manual del diagrama de la figura 2 a código HDL en sencilla, inmediata y de gran valor didáctico, como se observa en el código que se muestra en la página siguiente. El código generado automáticamente por el compilador es un poco distinto, pues se centra más en la lógica correspondiente a cada señal individual y crea máquinas de estado con un biestable por estado (*one-hot)*, pero en cualquier caso resulta equivalente desde el punto de vista de la simulación y, en gran medida, de la síntesis.

```
module basic_multiplier (clk, reset, inA, inB, go, outP, done);
   parameter            N = 12;      // Parámetro externo
   input                clk, reset;
   input      [N–1:0]   inA, inB;
   input                go;
   output  [2*N–1:0]    outP;
   output               done;
   reg        [N–1:0]   regA;
   reg     [2*N–1:0]    regB, outP;
   reg           [3:0]  regJ;
   reg                  done;
   reg                  state;
   parameter            Idle = 1'b0,  Loop = 1'b1;    // Estados

// Para el diseño 'multiplier' de la figura 4 hay que añadir aquí,
// además de una salida llamada 'ready', la siguiente sentencia:
// assign  ready = (state == Idle) ? 1 : 0;

   always @ (posedge clk or posedge reset)
   begin
      if ( reset ) begin          // Secuencia de inicialización
         done <= 0;               // Indicado por el usuario
         state <= Idle;           // Iniciar en el primer estado
      end else case (state)
```

(Continúa en la columna de la derecha)

```
      Idle:                       // Primer estado
      begin
         done <= 0;
         if ( go ) begin          // Espera a la señal 'go'
            regA  <= inA;         // Asignación de 12 bits
            regB  <= inB;         // Asignación de 24 bits
            outP  <= 0;           // Asignación de 24 bits
            regJ  <= 0;           // Asignación de  4 bits
            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'
endmodule            // Fin del módulo "basic_multiplier"
```

4. Elementos adicionales de los diagramas ASM++

Tanto el diagrama de la figura 1 como la parte derecha del diagrama de la figura 2 describen la estructura interna del módulo multiplicador propuesto, 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 pretendemos 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 necesitamos más cajas. Como muestra la primera columna de la figura 2, 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. En todo caso, la actual propuesta modifica ligeramente lo descrito anteriormente en [17] y [18].

Empezando por la esquina superior izquierda de la figura 2, la primera caja especifica el nombre del diseño (*"basic_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, como muestra la figura 6, cuando posteriormente se inserte este módulo en un circuito de nivel superior.

A continuación se detallan las entradas y salidas empleando código Verilog (la figura 5 ilustra 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, 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 las señales internas necesarias para la ejecución del algoritmo, es decir, varios biestables que almacenarán los datos. Las variables que contienen el estado, sin embargo, se generan de forma automática.

Finalmente, una pequeña caja indica que la señal *'clk'* sincronizará todo el circuito y 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'.

Gracias a estas nuevas cajas ya es posible describir completamente circuitos sencillos y generar automáticamente el código HDL correspondiente. Sin embargo, podemos dotar a estos diagramas de más posibilidades:

- En primer lugar, podemos diferenciar entre asignaciones síncronas (*synchronous assignments*, en inglés), efectuadas al finalizar cada ciclo de reloj, y operaciones asíncronas (en inglés *asynchronous assertions*), realizadas al comienzo de cada estado, por lo que permiten acceder al resultado durante la mayor parte del ciclo de reloj. Para diferenciar entre estas operaciones con tan distintos comportamientos en esta metodología se propone el uso de cajas rectangulares para las operaciones síncronas y cajas con lados combados para las asíncronas. A partir de la figura 3 se muestran ejemplos que emplean ambos tipos de cajas.

- Por otra parte, 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, 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.

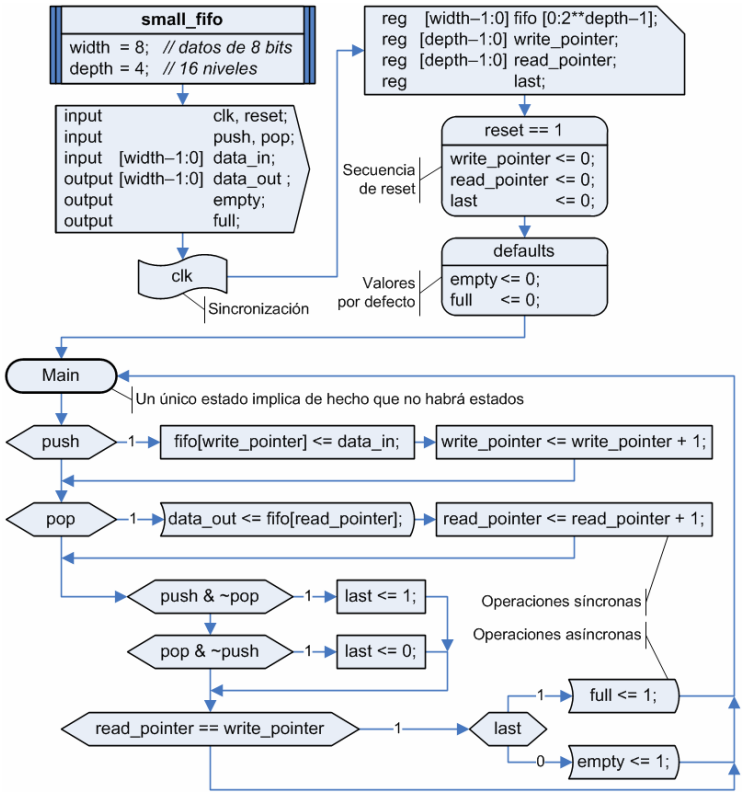

Figura 3. Un ejemplo de diagrama ASM++ con operaciones síncronas y asíncronas.

El código Verilog completo correspondiente al diagrama de la figura 3 se muestra a continuación. Conviene destacar que las operaciones asíncronas han de ser escritas fuera del bloque *"always"* (o *"process"* si usáramos VHDL): esto es debido a que no son sensibles a los flancos de la señal *'clk'* ni tampoco a la señal *'reset'*. También se observa que la señal asíncrona *'data_out'* ha sido simplificada al no estar definido su comportamiento en aquellos ciclos en los que no se recibe la señal *'pop'*.

```
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;

  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             <= 0;
      end
      else begin
          if (push)      write_pointer <= write_pointer + 1;
          if (pop)       read_pointer <= read_pointer + 1;

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

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

endmodule     /// small_fifo
```

Como se puede apreciar en el código anterior, de cada diagrama se extraen hasta tres bloques distintos: el primero es sensible a las señales *'clk'* y *'reset'*; el segundo es sensible únicamente a la señal de sincronización; el último estaría formado por todas las señales asíncronas. El diferente comportamiento de unas y otras justifica el uso de cajas distintas: esto facilita el análisis de un diagrama para evaluar su funcionamiento y, por supuesto, ayuda durante la fase de escritura manual del código [19,20].

Antes de continuar, y empleando los nuevos recursos, añadimos en el multiplicador de la figura 4 una salida *'ready'* que indica con un nivel alto que el circuito está preparado para recibir nuevos datos.

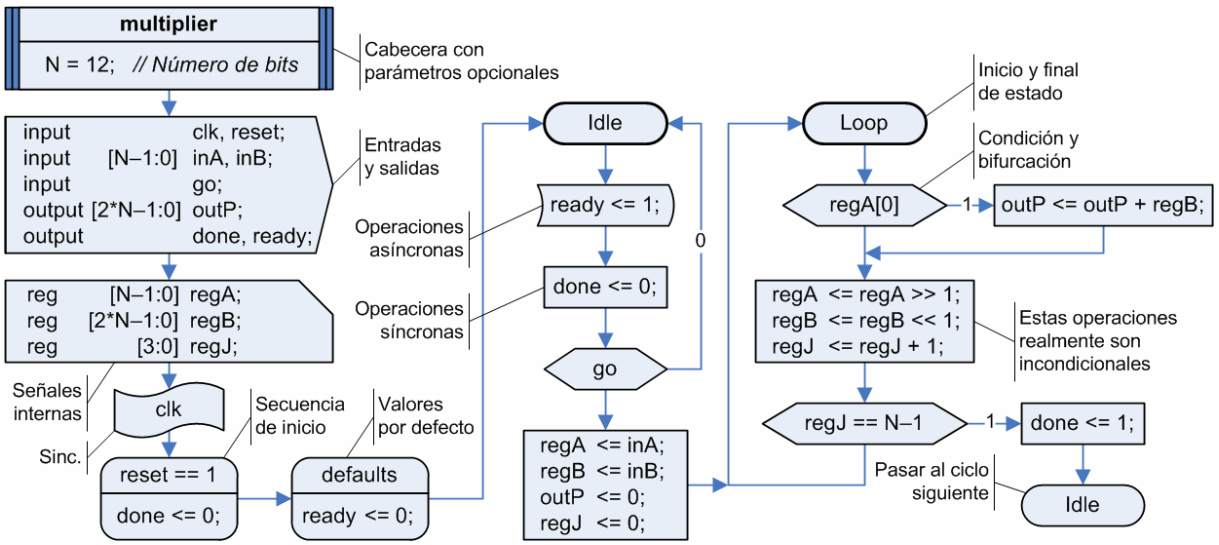

Figura 4. Circuito multiplicador modificado añadiendo una salida asíncrona de circuito preparado (*'ready'*).

5. Diagramas ASM++ con múltiples máquinas de estado

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 [18] con frecuencia nos encontramos con situaciones en las que una única máquina de estados o una única línea de ejecución resulta insuficiente. Necesitamos 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. Los diagramas ASM++ incorporan esta posibilidad, como se muestra en la figura 5 que incluye una versión simplificada de una FIFO con dos relojes. Este diseño emplea VHDL como lenguaje de entrada, y la salida, expresada en el mismo lenguaje, se muestra a continuación.

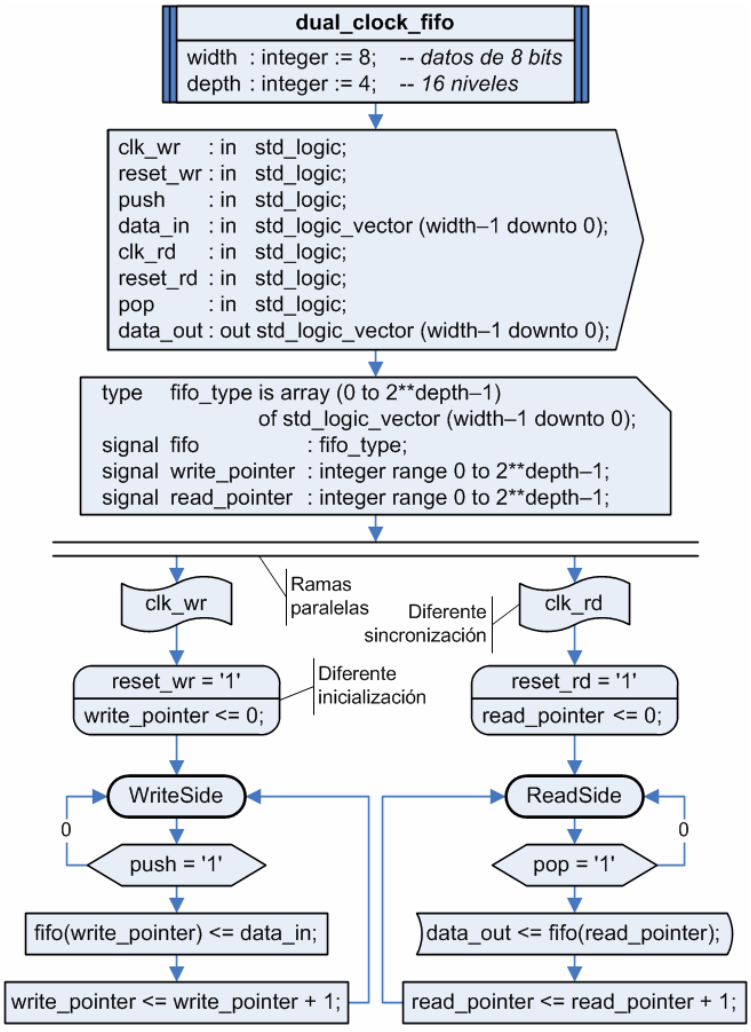

Figura 5. Un ejemplo con dos ramas paralelas y dos señales de sincronización distintas.

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

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

6. Diseño jerárquico empleando diagramas ASM++

Como se deduce de los apartados anteriores, las capacidades de los diagramas ASM++ son muy similares a las que aportan los lenguajes de descripción de circuitos, pues de hecho se basan en ellos y, lo que es más importante desde el punto de vista didáctico, aseguran una forma de utilizarlos robusta y a la vez flexible, orientada hacia circuitos reales sobre dispositivos reconfigurables.

Por todo ello ha sido necesario dotar al lenguaje ASM++ de la capacidad de abordar diseños de forma jerárquica. Esto se ha realizado en tres pasos: en primer lugar, es posible describir varios circuitos en un mismo fichero, que de hecho puede constar de una o varias páginas; a continuación, es posible incorporar un diseño dentro otro empleando una caja denominada *'Instance'*, similar a la de cabecera (ver figura 6), donde se indica el modelo de circuito, su nombre, y opcionalmente se asigna un valor a sus parámetros externos; finalmente es posible compilar conjuntamente varios ficheros empleando la caja *'RequireFile'*, que procesa los ficheros solicitados una sola vez –para evitar duplicaciones–, incluso cuando se solicitan varias veces.

El diseño propuesto como ejemplo en la figura 6 construye un multiplicador 16x16 con dos memorias FIFO a la entrada y una FIFO adicional a la salida, para mejorar el rendimiento del circuito. La propuesta de esta metodología es crear, con cada implementación de un módulo de nivel inferior, un conjunto de señales conectadas automáticamente a los puertos del módulo incorporado. La notación basada en un punto (".") entre el nombre del módulo y el nombre de la señal es empleada ampliamente en otros ámbitos y, al ser incompatible con VHDL y Verilog, permite diferenciarla. En el código generado se sustituye el punto por un signo de subrayado ("_"). Como se aprecia en la siguiente figura, el resultado es claro e intuitivo, y el diseñador sólo tiene que escribir lo estrictamente necesario.

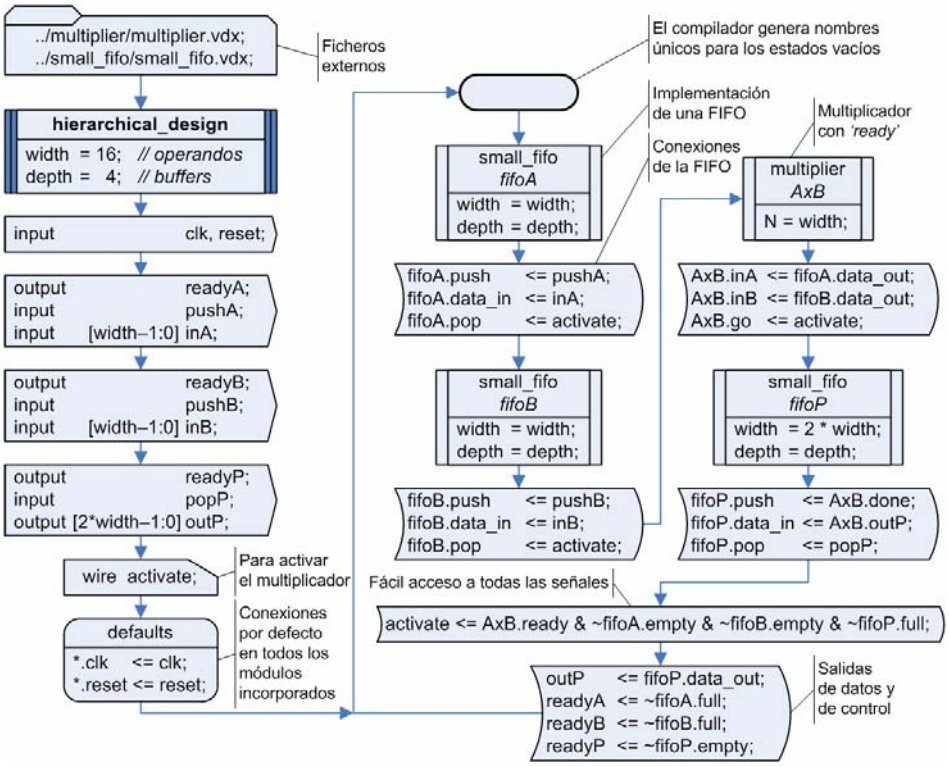

Figura 6. Un diseño jerárquico empleando diagramas ASM++.

El código Verilog correspondiente al diagrama de la figura 6 se muestra a continuación. De nuevo se aprecia la relación entre ambos y es significativa la cantidad de líneas de código que se ahorran al usar los diagramas ASM++: se evitan todas las líneas redundantes requeridas por VHDL y Verilog.

```
module hierarchical_design (clk, reset,   readyA, pushA, inA,
                            readyB, pushB, inB,   readyP, popP, outP);

  parameter  width  = 16;     // Openandos de 16 bits sin signo
  parameter  depth =  4;      // Buffers de 16 niveles

  input                    clk, reset;

  output                   readyA;
  input                    pushA;
  input      [width–1:0]   inA;

  output                   readyB;
  input                    pushB;
  input      [width–1:0]   inB;

  output                   readyP;
  input                    popP;
  output [2*width–1:0]     outP;

  wire                     activate;     // Señal interna

  wire                     fifoA_clk, fifoA_reset;
  wire       [width–1:0]   fifoA_data_in, fifoA_data_out;
  wire                     fifoA_push, fifoA_pop, fifoA_empty, fifoA_full;
  small_fifo  # (width <= width, depth <= depth)  fifoA  (
    .clk(fifoA_clk), .reset(fifoA_reset),
    .push(fifoA_push), .pop(fifoA_pop), .data_in(fifoA_data_in),
    .data_out(fifoA_data_out), .empty(fifoA_empty), .full(fifoA_full) );
```

(Sigue a la derecha)

```
  wire                     fifoB_clk, fifoB_reset;
  wire       [width–1:0]   fifoB_data_in, fifoB_data_out;
  wire                     fifoB_push, fifoB_pop, fifoB_empty, fifoB_full;
  small_fifo  # (width <= width, depth <= depth)  fifoB  (
    .clk(fifoB_clk), .reset(fifoB_reset),
    .push(fifoB_push), .pop(fifoB_pop), .data_in(fifoB_data_in),
    .data_out(fifoB_data_out), .empty(fifoB_empty), .full(fifoB_full) );

  wire                     AxB_clk, AxB_reset;
  wire                     AxB_go, AxB_done, AxB_ready;
  wire       [width–1:0]   AxB_inA, AxB_inB;
  wire   [2*width–1:0]     AxB_outP;
  multiplier  # (width)  AxB  (
    .clk(AxB_clk), .reset(AxB_reset),
    .inA(AxB_inA), .inB(AxB_inB), .go(AxB_go),
    .outP(AxB_outP) , .done(AxB_done), .ready(AxB_ready) );

  wire                     fifoP_clk, fifoP_reset;
  wire   [2*width–1:0]     fifoP_data_in, fifoP_data_out;
  wire                     fifoP_push, fifoP_pop, fifoP_empty, fifoP_full;
  small_fifo  # (width <= 2 * width, depth <= depth)  fifoP  (
    .clk(fifoP_clk), .reset(fifoP_reset),
    .push(fifoP_push), .pop(fifoP_pop), .data_in(fifoP_data_in),
    .data_out(fifoP_data_out), .empty(fifoP_empty), .full(fifoP_full) );
```

(Sigue en la página siguiente a la izquierda)

(Viene de la página anterior)

```
assign  fifoA_clk       = clk;      // Conexiones por defecto
assign  fifoB_clk       = clk;
assign  AxB_clk         = clk;
assign  fifoP_clk       = clk;
assign  fifoA_reset     = reset;
assign  fifoB_reset     = reset;
assign  AxB_reset       = reset;
assign  fifoP_reset     = reset;

assign  fifoA_push      = pushA;    // Conexiones del usuario
assign  fifoA_data_in   = inA;
assign  fifoA_pop       = activate;

assign  fifoB_push      = pushB;
assign  fifoB_data_in   = inB;
assign  fifoB_pop       = activate;
```

(Viene de la columna de la izquierda)

```
assign  AxB_inA         = fifoA_data_out;
assign  AxB_inB         = fifoB_data_out;
assign  AxB_go          = activate;

assign  fifoP_push      = AxB_done;
assign  fifoP_data_in   = AxB_outP;
assign  fifoP_pop       = popP;

assign  activate        = AxB.ready & ~fifoA_empty &
                          ~fifoB_empty & ~fifoP_full;

assign  outP            = fifoP_data_out;
assign  readyA          = ~fifoA_full;
assign  readyB          = ~fifoB_full;
assign  readyP          = ~fifoP_empty;

endmodule   /// hierarchical_design
```

7. Conclusiones

Este artículo ha presentado una metodología gráfica muy intuitiva y completa 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 en función de los usos comunes y robustos de los lenguajes de descripción de circuitos VHDL y Verilog.

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 FPGA. También es empleado para el desarrollo de módulos, para la supervisión de diseños y para la documentación de resultados finales en proyectos realizados desde la Universidad para empresas externas.

La introducción de diagramas ASM++ es fácil y cómoda a través de múltiples programas gráficos como *MS Visio*, *SmartDraw* o *ConceptDraw*, empleando en cualquier caso el formato VDX (que internamente es XML). 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.

Agradecimientos

Los autores agradecen la financiación aportada para estos desarrollos por eZono AG, ubicada en Jena, Alemania y por ISEND SA, ubicada en el Parque Tecnológico de Boecillo, Valladolid.

Referencias

[1] C.R. Clare, *Designing Logic Using State Machines*, McGraw-Hill, 1973.

[2] S. Leibson, "The NMOS II Hybrid Microprocessor: Fusing silicon, ceramic, and aluminum with rubber baby buggy bumpers", online en *http://www.hp9825.com/html/hybrid_microprocessor.html*, revisado en abril 2008.

[3] V.R.L. Shen y F. Lai, "Requirements Specification and Analysis of Digital Systems Using Fuzzy and Marked Petri Nets", *IEEE Trans. on Systems, Man and Cybernetics*, vol. 32, no. 1, pp. 149-159, enero 2002.

[4] S.B. Örs, L. Batina, B. Preneel y J. Vandewalle, “Hardware Implementation of an Elliptic Curve Processor over GF(p)”, Proc. of Application-Specific Systems, Architectures and Processors (ASAP’03), pp. 1-11, 2003.
[5] D.D. Gajski, *Principles of Digital Design*, Prentice Hall, Upper Saddle River, NJ, 1997.
[6] D.D. Givone, *Digital Principles and Design*, McGraw-Hill Professional, 2002.
[7] C.H. Roth, *Fundamentals of Logic Design*, 5ª edición, Thomson-Engineering International, 2003.
[8] A.T. Bahill et al., “The design-methods comparison project”, *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 28, No. 1, pp. 80-103, febrero 1998.
[9] S. Baranov, “Synthesis of control units for mobile robots”, 2nd EUROMICRO workshop on Advanced Mobile Robots, pp. 80-86, 1997.
[10] W.F. Lee et al., “An ASM-based ASIC for automobile accelerometer applications”, 1st IEEE Asia Pacific Conference on ASICs, pp. 127-130, 1999.
[11] M.S. Nixon, “On a Programmable Approach to Introducing Digital Design”, *IEEE Trans. on Education*, Vol. 40, No. 3, pp. 195-206, agosto 1997.
[12] J.P. David y E. Bergeron, “A Step towards Intelligent Translation from High-Level Design to RTL”, Proc. of 4th IEEE International Workshop on System-on-Chip for Real-Time Applications, pp. 183-188, 2004.
[13] E. Ogoubi y J.P. David, “Automatic synthesis from high level ASM to VHDL: a case study”, 2nd Annual IEEE Northeast Workshop on Circuits and Systems, pp. 81-84, 2004.
[14] E. Bergeron, X. Saint-Mleux, M. Feeley y J.P. David, “High Level Synthesis for Data-Driven Applications”, 16th IEEE International Workshop on Rapid System Prototyping, pp. 54-60, 2005.
[15] D. Ponta y G. Donzellini, “A Simulator to Train for Finite State Machine Design”, Proc. IEEE FIE’96, pp. 725-729, 1996.
[16] S. de Pablo y otros, “ASM++ diagrams: a modern methodology for RTL designs”, documento on-line disponible en *http://www.epYme.uva.es/ASM++.php*, actualizado en abril de 2008.
[17] S. de Pablo, S. Cáceres, J.A. Cebrián y M. Berrocal, “A proposal for ASM++ diagrams”, 10th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems, Cracovia, Polonia, pp. 121-124, 2007.
[18] S. de Pablo, S. Cáceres, J.A. Cebrián y M. Berrocal, “Application of ASM++ methodology on the design of a DSP processor”, 4th FPGAworld Conference, Estocolmo, Suecia, pp. 13-19, 2007.
[19] M. Chang, “Teaching Top-down Design Using VHDL and CPLD”, Proc. IEEE FIE’96, pp. 514-517, 1996.
[20] T.A. Giuma, D. Welch y K. MacDonald, “Computer-Aided-Design Platform for Sequential Systems”, Proc. IEEE Southeastcon’97, pp. 79-81, 1997.