ASM++ diagrams used on teaching electronic design

S. de Pablo, S. Cáceres, J.A. Cebrián
Department of Electronics Technology
University of Valladolid, Valladolid, Spain.
{sanpab,sancac,jesceb}@eis.uva.es

M. Berrocal
eZono AG
Jena, Germany.
manuel@ezono.com

F. Sanz
European University
"Miguel de Cervantes", Valladolid, Spain.
fsanz@uemc.es

***Abstract*– Algorithmic State Machines are a 40-year old tool for the design of digital circuits. They are a good alternative to Finite State Machines, where only states can be properly described, but all operations must be annotated as lateral comments. This paper shows the inner relationship between ASM diagrams and modern languages used to describe hardware and it proposes several modifications to the standard methodology to allow automated tools to produce Verilog or VHDL code from these diagrams. The new notation is more complete and consistent and thus more convenient for CAD tools.**

I. INTRODUCTION

The Algorithmic State Machine (ASM) method for specifying digital designs, in an abstract behavioral form, was originally documented by Clare [1] who worked at the Electronics Research Laboratory of Hewlett Packard Labs, based on previous developments made by Osborne at the University of California at Berkeley [2]. Since then it has been widely applied to assist designers in expressing abstract algorithms and to support their conversion into hardware [3]. Many texts on digital logic design cover the ASM method in conjunction with other methods for specifying Finite State Machines (FSM), namely state tables and state diagrams [4]. Whereas most designers simply use them as a means to specify the control of digital systems using complex FSM models [5], [6], [7], [8], few texts actually use them to design whole systems, except [9] and [10], that actually increment their possibilities with higher level features.

State diagrams do not describe properly the actions that must be executed as the control unit evolves through different states. Meanwhile, ASM are a good alternative because they prevent inconsistent diagram specifications and they are easier to read and maintain. However, most authors consider them impractical for large algorithms and hard to manage because of their graphical interface [11], so modern hardware description languages (HDL) are usually preferred to design at the register transfer level (RTL).

Our engineering students at the University learn basic electronic design using schematic capture tools and later on they increment their skills on more complex RTL designs using VHDL and Verilog. We have seen that FSM are useful to give the students an overall view of a design, but they are limited for detailed descriptions. Additionally, we have found that ASM diagrams are useful during the concept capturing of a digital development [5], because it is easy to materialize ideas using them. During the detailed design specification, when the order between different tasks becomes complex or confuse, these diagrams help clarifying the ordering and interactions between tasks.

However, we have found that current ASM notation cannot properly describe real-life circuits, which usually have several FSM running in parallel with complex interactions between them. This paper presents a different notation, called "ASM++ diagrams", aiming to improve ASM for more complex designs and more suitable for automatic conversion into HDL code.

II. TRADITIONAL ASM DIAGRAMS

Traditional ASM diagrams use three types of boxes: the "state boxes" –with rectangular shape– define the beginning of each clock cycle and may include unconditional operations that must be executed *during* or *at the end* of that cycle; "decision boxes" –diamond ones– are used to test inputs or internal values to determine the execution flow; and finally "conditional output boxes" –oval ones– indicate those operations that are executed only when previous conditions are valid. An "ASM block" includes all operations and decisions that are or can be performed simultaneously during each execution cycle.

These ideas are illustrated in fig. 1, where a 12x12 unsigned multiplier has been implemented through additions and shifts using two states: during 'Idle' state this circuit waits for two new operands given at 'inA' and 'inB' inputs when the 'go' signal is asserted to one; the second state, named 'Loop', executes twelve additions and shifts in twelve clock cycles to compute the desired product. At the end, a 'done' signal validates the result given at the output 'outP'. This circuit is asynchronously initialized using an active high signal called 'reset' and then it is synchronized with a 'clk' signal not included in this diagram.

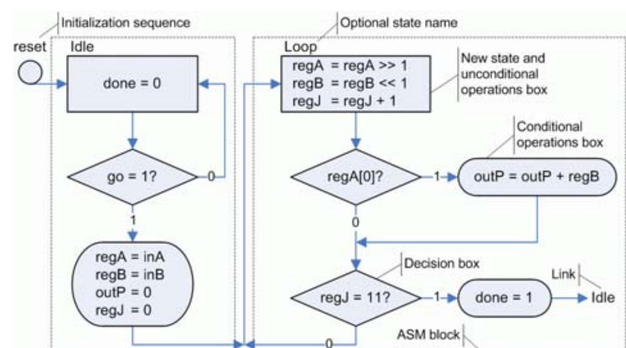

Fig. 1. An example of traditional ASM.

The advantages of this representation over FSM are evident: not only the evolution between states has been described, but also the operations in and between states have been included; additionally, conditions can be built up incrementally and later combined into a single boolean condition [5]. However, they have several properties that may be seen as disadvantages:

- They use the same box –with rectangular shape– for new states and unconditional operations executed at those states. Because of this property, ASM diagrams are compact, but they are also more rigid and difficult to read.

- Sometimes it is difficult to differentiate the frontier between different states. The complexity of some states requires the use of dashed boxes (named ASM blocks) or even different colors for different states.

- Due to the double meaning of rectangular boxes, conditional operations must be represented using a different shape, the oval boxes. But, actually, all operations are conditional, because all of them are state dependent.

- Additionally, designers must use lateral annotations for state names, for reset signals or even for links between different parts of a design (see fig. 1).

- Finally, the width of signals and ports cannot be specified when using the current notation.

The new notation proposed in this paper tries to solve all these problems.

III. Basic Features of ASM++ Diagrams

The first and main change introduced by this new notation, see fig. 2, is the use of a specific box for states –we propose oval boxes, very similar to those circles used in bubble diagrams– so now all operations may share the same rectangular box. Diamonds are kept for decision boxes because they are commonly recognized and accepted.

As shown in fig. 2, the resulting ASM++ diagram is less compact, but more clear and flexible. Former ASM blocks become useless because limits between states are clearly defined by state boxes. All lateral annotations have disappeared when using the new notation and designers have more freedom to write operations using any arbitrary order: all operations written from one state to the next one are executed in parallel, but sometimes unconditional operations are better written *after* conditional ones, as shown in 'Loop' state where shifts are specified below the conditional addition. The use of a specific box for states also simplifies the use of links between different parts of a diagram. This feature helps in writing complex designs that cannot fit in a single page.

More boxes are described in the following section, but an important property of ASM++ can be pointed out now: all boxes use standard HDL expressions, either Verilog (this case) or VHDL (see fig. 6). This feature helps during the phase of handwriting the final HDL code, but also it is decisive for the ASM++ compiler that is in progress: it will generate HDL code for simulators and synthesizers based on these diagrams.

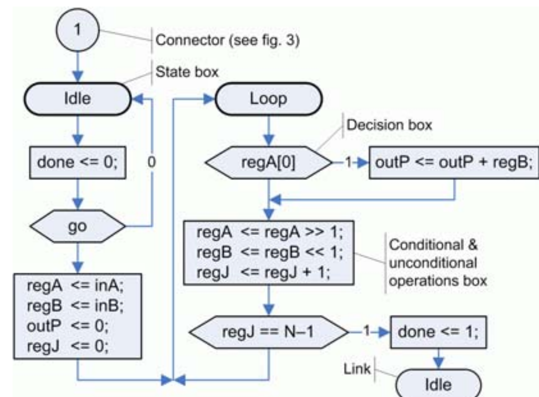

Fig. 2. Basic boxes of the new ASM++ notation.

The following code (see also fig. 3 to complete the circuit definition) shows how ASM++ diagrams can be easily translated into HDL code. After a quick look the correspondence between this diagram and its code is evident.

```
parameter  N = 12;                              // See fig. 3
// ...
parameter  Idle = 1'b0,  Loop = 1'b1;           // State codes
always @ (posedge clk or posedge reset)         // See also fig. 3
begin
    if ( reset ) begin                  // Initialization sequence
        done <= 0;                         // Indicated by designer
        state <= Idle;                     // Going to the first state
    end else case (state)
        Idle:                           // First state
        begin
            done <= 0;
            if ( go ) begin                // Waiting for 'go'
                regA <= inA;               // 12 bits assignment
                regB <= inB;               // 24 bits assignment
                outP <= 0;                 // 24 bits assignment
                regJ  <= 0;                //   4 bits assignment
                state <= Loop;             // Jump to next state
            end
        end
        Loop:                           // Second state
        begin
            if ( regA[0] )
                outP <= outP + regB;       // Conditional operation
            regA <= regA >> 1;             // Unconditional operations
            regB <= regB << 1;             // The order ...
            regJ  <= regJ + 1;              // ... is decided by user
            if ( regJ == N–1 ) begin
                done <= 1;                 // Indicate it finishes
                state <= Idle;             // Conditional jump
            end
        end
        default:                        // Because of security reasons
            state <= Idle;
    endcase
end     // always block
```

IV. Additional Features of ASM++ Diagrams

Figure 2 describes the internal architecture of the proposed module, but says nothing about its external interface, the width of internal signals, or about synchronization. For an automated

tool that generates HDL code from this specification other boxes are clearly required. Therefore, the new ASM++ notation adds more boxes to face these requirements, as shown in fig. 3. Though, this paper modifies what was proposed at [12] and [13].

From up to down, the first box of fig. 3 specifies the design name ("multiplier") and its optional parameters (or *generics*, using the VHDL notation). Following, a port-like box allows a detailed definition of the module interface: all inputs and outputs can be described using the preferred HDL language. Obviously, bigger designs may use more than one of these interface boxes.

The third box has a card-like shape and is used to introduce general HDL code and compiler directives: in this case, it is used to declare and specify the size of all internal signals (except the variable used for the resulting state machine).

Afterwards, a small box indicates that the synchronism of this whole circuit will be managed using the 'clk' signal, and finally the active high 'reset' signal will be used to asynchronously initialize the 'done' output and the internal state, starting at the first state named 'Idle' (see also fig. 2).

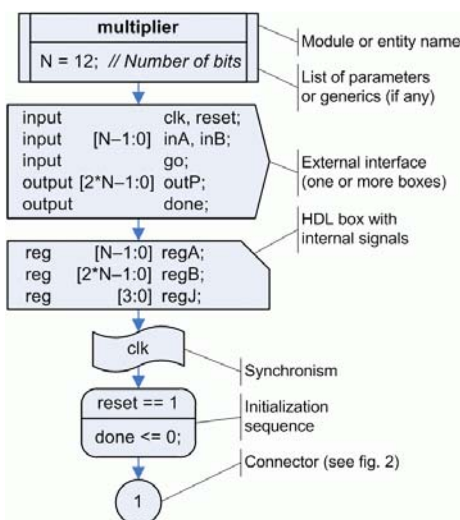

Fig. 3. Additional ASM++ boxes required to generate HDL code.

Using these new boxes, designers are now capable of completely describe a module and generate the subsequent HDL code for simulation and synthesis. However, looking with greater attention to the signals of these diagrams, we found that all of them exhibit a synchronous behavior: all operations included in rectangular boxes are registered *at the end* of each clock cycle. To improve the designer possibilities, a new box for asynchronous assertions is also provided: they have bent sides, as shown on fig. 4, and describe operations that are performed *during* the present clock cycle.

Furthermore, a new box to describe default values has been included: when synchronous signals are not used in one or more states, the default behavior of those signals must be *to keep constant* their last value, and that is the way all VHDL and Verilog compilers work. But what happens when an asynchronous signal is not used in one or more states? The proposed value for those situations is *"don't care"* to minimize the generated combinatorial logic (see for example the HDL code for 'data_out' on fig. 4, it actually does not depend on the value of 'pop'). A new box has been included to modify these default behaviors, for synchronous and asynchronous signals; otherwise, designers would be required to write their specifications once and again.

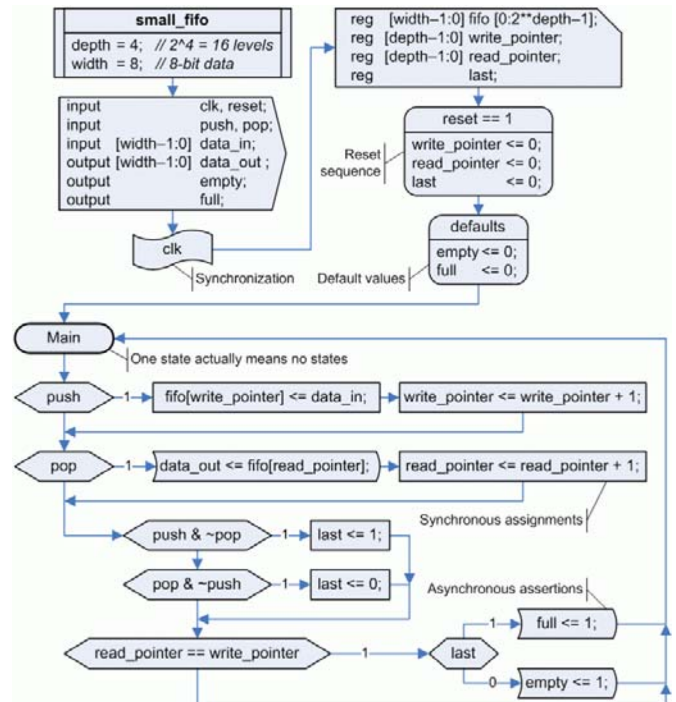

Fig. 4. An ASM++ example with asynchronous assertions.

The full Verilog code for the diagram of fig. 4 is shown following. It should be noted that asynchronous assertions ought to be written *out of* the 'always' (or 'process') block, because they are not sensible to 'clk' edges or to 'reset' signal.

```
module small_fifo (clk, reset,
                   push, pop, data_in, data_out, empty, full);

  parameter  depth =  4;              // It means 2^4 = 16 levels
  parameter  width  =  8;             // 8-bit data width

  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 or posedge reset)
  begin
     if (reset) begin
        write_pointer     <= 0;       // Initialize pointers
        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

    always @ (posedge clk)
    begin
        if (push)  fifo[write_pointer] <= data_in;
    end

    assign data_out = fifo[read_pointer];

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

endmodule    /// small_fifo
```

The main reason for differentiating synchronous assignments –codified into an 'always' block– from asynchronous assertions –codified out of it, is that when code is handwritten [14], [15], designer must look for all assertions *after* the main block is written down. In order to understand the behavior of the circuit, a different shape also helps.

V. HIERARCHICAL DESIGN USING ASM++ DIAGRAMS

A top level example with a hierarchical design is shown in fig. 5. It includes all previous modules to build a 12x12 multiplier with two input FIFOs and an output FIFO to maximize its throughput. As can be seen, when a module is instantiated in an ASM++ diagram, the proposal is that a full set of signals (with the name of the instantiated module, a dot, and the name of each module port) are automatically available to easy its connections with other modules. The result is clear, as can be seen below, and designers write only what they need.

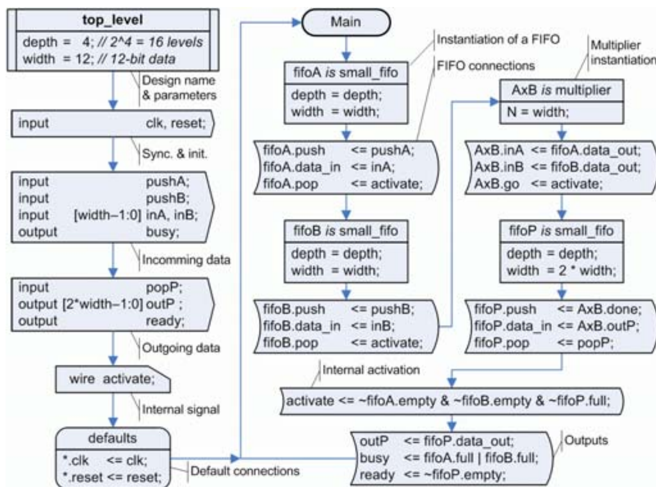

Fig. 5. A hierarchical design using ASM++ diagrams.

The Verilog code for the diagram of fig. 5 is shown at the right column. The relationship between diagrams and HDL code becomes evident again. It is also remarkable the amount of code saved when using ASM++ diagrams, because most redundant lines required by Verilog and VHDL are avoided.

```
module top_level (clk, reset,
                        pushA, pushB, inA, inB, busy,
                        popP, outP, ready);

    parameter  depth =  4;                    // 16 levels at each FIFO
    parameter  width  = 12;                   // 12-bit unsigned integer

    input                          clk, reset;

    input                          pushA, pushB;
    input     [width–1:0]          inA, inB;
    output                         busy;

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

    wire                           activate;       // Internal signal

    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  # (depth <= depth, width <= width)  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) );

    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  # (depth <= depth, width <= width)  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, AxB_go, AxB_done;
    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) );

    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  # (depth <= depth, width <= 2 * width)  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) );

    assign  fifoA_clk     = clk;                // Default connections
    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;          // User connections
    assign  fifoA_data_in     = inA;
    assign  fifoA_pop         = activate;

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

    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 = ~fifoA_empty & ~fifoB_empty & ~fifoP_full;
    assign  outP     = fifoP_data_out;
    assign  busy     = fifoA_full | fifoB_full;
    assign  ready    = ~fifoP_empty;

endmodule    /// top_level
```

VI. PARALLEL THREADS ON ASM++ DIAGRAMS

Operations at any state are (conditionally) executed in parallel. However, real life circuits usually need two or more state machines also running in parallel [13], with dependent or independent state sequences. ASM++ diagrams introduce that feature, as shown in fig. 6. This example uses VHDL to describe a simplified version of a dual-clock FIFO, with no flags, and also VHDL is used for the generated code.

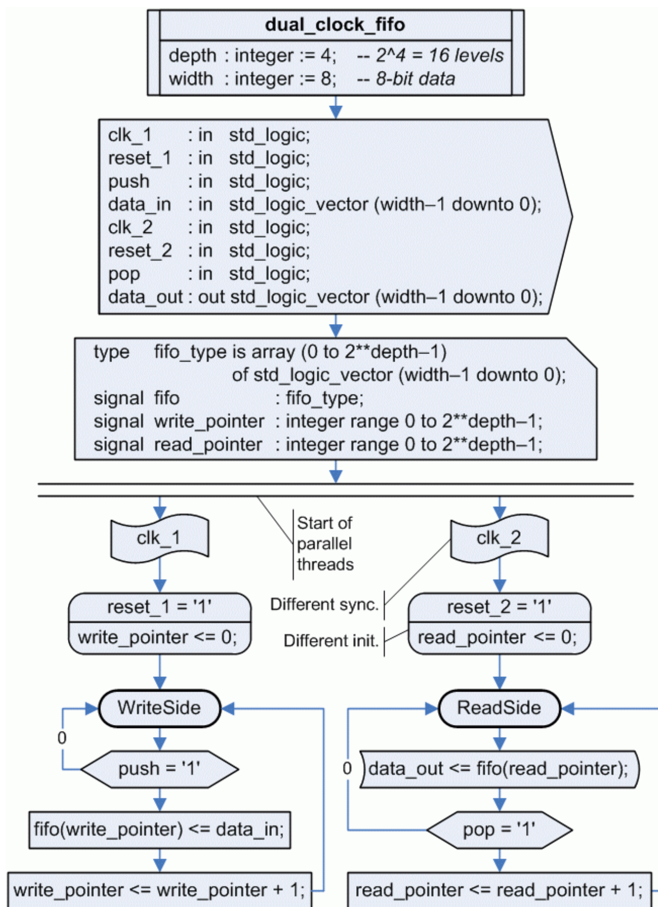

Fig. 6. A two threaded design with two clock signals.

The VHDL code that implements the functionality described on fig. 6 is shown following.

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

entity dual_clock_fifo is
  generic (
    depth  : integer := 4;     -- 2^4 = 16 levels
    width  : integer := 8      -- 8-bit data
  );
  port (
    clk_1, reset_1, push       : in  std_logic;
    data_in                    : in  std_logic_vector (width–1 downto 0);
    clk_2, reset_2, 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;
begin
  process (clk_1, reset_1)              -- Write pointer control
  begin
    if (reset_1 = '1') then
      write_pointer <= 0;
    elsif rising_edge(clk_1) then
      if (push = '1') then
        write_pointer <= write_pointer + 1;
      end if;
    end if;
  end process;

  process (clk_1)                       -- Writing of the FIFO
  begin
    if rising_edge(clk_1) then
      if (push = '1') then
        fifo(write_pointer) <= data_in;
      end if;
    end if;
  end process;

  process (clk_2, reset_2)              -- Read pointer control
  begin
    if (reset_2 = '1') then
      read_pointer <= 0;
    elsif rising_edge(clk_2) then
      if (pop = '1') then
        read_pointer <= read_pointer + 1;
      end if;
    end if;
  end process;

  data_out <= fifo(read_pointer);   -- Asynchronous FIFO output
end RTL;  --- dual_clock_fifo
```

VII. CONCLUSIONS

This article has presented a powerful graphical method for electronic design at register transfer level. It is based on standard ASM diagrams and most used structures of modern hardware description languages when implementing FPGA designs, improving their possibilities.

The resulting graphical language is easy to learn and it allows a quick circuit behavior understanding, thus it is a suitable representation for design supervision and documentation. It also has demonstrated that it is an excellent medium for teaching electronics, as a means of describing the circuit functionality before schematic capture or HDL handwriting.

However, this is an ongoing methodology and new boxes with advanced functionality will be added in a near future. An ASM++ compiler for Verilog and VHDL is under active development.

ACKNOWLEDGMENTS

The authors would like to acknowledge the financial support for these developments of eZono AG, Jena, Germany.

REFERENCES

[1] C.R. Clare, *Designing Logic Using State Machines*, McGraw-Hill, 1973. Referenced by [2].

[2] S. Leibson, "The NMOS II Hybrid Microprocessor: Fusing silicon, ceramic, and aluminum with rubber baby buggy bumpers", online at *http://www.hp9825.com/html/hybrid_microprocessor.html*, reviewed on March 2007.

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

[4] D.D. Gajski, *Principles of Digital Design*, Prentice Hall, Upper Saddle River, NJ, 1997.

[5] A.T. Bahill et al., "The design-methods comparison project", *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 28, No. 1, pp. 80-103, February 1998.

[6] S. Baranov, "Synthesis of control units for mobile robots", *Second EUROMICRO workshop on Advanced Mobile Robots*, pp. 80-86, 1997.

[7] W.F. Lee et al., "An ASM-based ASIC for automobile accelerometer applications", *First IEEE Asia Pacific Conference on ASICs*, pp. 127-130, 1999.

[8] M.S. Nixon, "On a Programmable Approach to Introducing Digital Design", *IEEE Trans. on Education*, Vol. 40, No. 3, pp. 195-206, August 1997.

[9] J.P. David and E. Bergeron, "A Step towards Intelligent Translation from High-Level Design to RTL", *Proceedings of 4th IEEE International Workshop on System-on-Chip for Real-Time Applications*, pp. 183-188, 2004.

[10] E. Ogoubi and 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.

[11] E. Bergeron, X. Saint-Mleux, M. Feeley, and J.P. David, "High Level Synthesis for Data-Driven Applications", *16th IEEE International Workshop on Rapid System Prototyping*, pp. 54-60, 2005.

[12] S. de Pablo, S. Cáceres, J.A. Cebrián, and M. Berrocal, "A proposal for ASM++ diagrams", *10th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems*, Krakow, Poland, 2007.

[13] S. de Pablo, S. Cáceres, J.A. Cebrián, and M. Berrocal, "Application of ASM++ methodology on the design of a DSP processor", *4th FPGAworld Conference*, Stockholm, Sweden, 2007.

[14] M. Chang, "Teaching Top-down Design Using VHDL and CPLD", *IEEE FIE'96 Proceedings*, pp. 514-517, 1996.

[15] T.A. Giuma, D. Welch, and K. MacDonald, "Computer-Aided-Design Platform For Sequential Systems", *IEEE Southeastcon'97 Proceedings*, pp. 79-81, 1997.