A proposal for ASM++ diagrams

Santiago de Pablo, Santiago Cáceres, Jesús A. Cebrián
University of Valladolid
Valladolid, Spain
Email: sanpab,sancac,jesceb@eis.uva.es

Manuel Berrocal
eZono GmbH
Jena, Germany
Email: manuel@ezono.com

***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 actions must be annotated as lateral comments. However, current notation for these diagrams has several limitations for medium-large designs, and often lateral annotations are finally needed. This paper presents an alternative notation for ASM diagrams, trying to overcome these limitations. This new notation is more 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 Claire [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][5]. Whereas most designers simply use them as a means to specify the control of digital systems using complex FSM models [6][7][8][9], few texts actually use them to design whole systems.

State diagrams are weak at capturing the structure behind complex sequencing. The problem is that they 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. Nevertheless, some authors consider them impractical for large algorithms and hard to manage because of their graphical interface [10], so modern Hardware Description Languages (HDL) are usually preferred.

Nowadays ASM diagrams are used at different stages of the design flow in multiple designs:

- They are useful during the concept capturing of a digital development [6], 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.
- They can be used for design documentation, after they have been written and verified, because they describe in detail the actions performed and the timing of those actions.
- They are also used to generate testbenches.

Current notation for ASM diagram is compact and valid for control-oriented designs, those where the main task of the circuit is to generate control signals. But when a design focuses mainly on its data path, this notation has several limitations.

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 describe 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 during each execution cycle.

These ideas are illustrated in fig. 1, where a 12x12 unsigned multiplier has been implemented using two states: during 'Idle' state it waits for two new operands given at 'in_a' and 'in_b' 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 'p'. This circuit is initialized using an asynchronous signal called 'reset'.

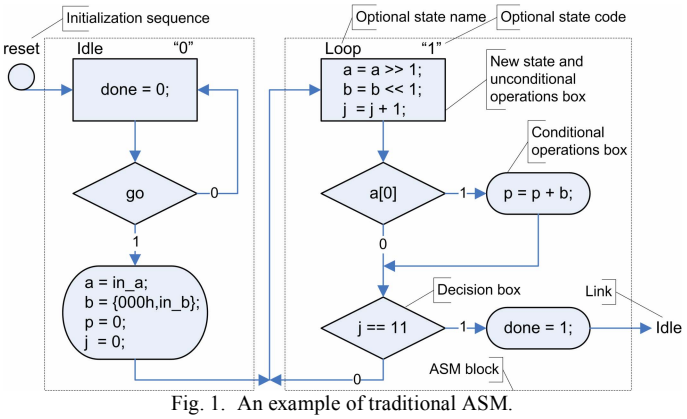

Fig. 1. An example of traditional ASM.

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 [6]. However, they have several properties that may be seen as disadvantages:

- They use the same box –rectangular ones– for new states and unconditional operations at those states. Because of this property, ASM diagrams are more compact, but they are also more difficult to read.
- Sometimes it is difficult to differentiate the frontier between states. The complexity of some states requires the use of dashed boxes or even different colors for different ASM blocks.
- Due to the *double meaning* of rectangular boxes, conditional operations must be represented using a different shape, the oval boxes.
- Additionally, designers must use lateral annotations for state names and codes, 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 by current notation.

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

III. Proposed New Notation: ASM++ Diagrams

The first and main change introduced by this new notation 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. There is no need for ASM blocks because limits between states are clearly defined by state boxes. All lateral annotations –the nightmare of CAD developers– have disappeared, and designers have more freedom to write the operations in the order they want: 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. The use of a 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.

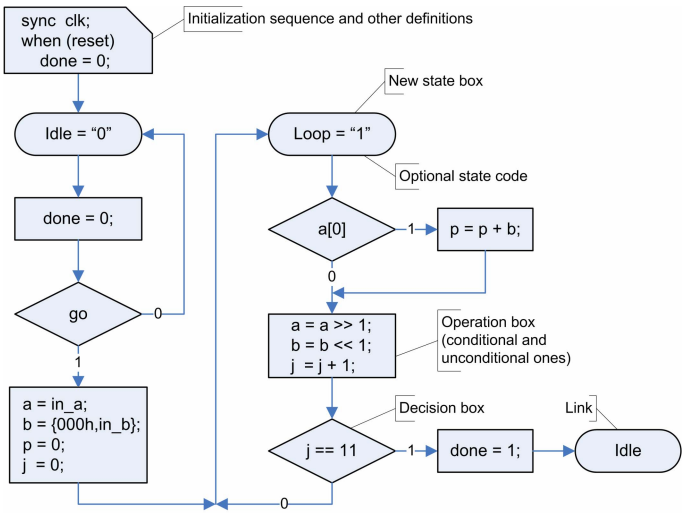

Fig. 2. An example of the new ASM++ notation.

The following code shows how ASM++ diagrams can be easily translated into Verilog (or VHDL) code. After a quick look the correspondence between this diagram and its code is evident.

```
// All signals must be declared before this line
parameter  Idle   = 1'b0,
           Loop = 1'b1;            // State codes
always @ (posedge clk or posedge reset)
begin
    if (reset) begin               // Initialization sequence
        state <= Idle;             // Going to the first state
        done <= 0;                 // Indicated by designer
        {a, b, p, j} <= 0;         // They depend on reset!
    end else case (state)

        Idle:                      // First state
        begin
            done <= 0;
            if (go) begin              // Waiting for 'go'
                a <= in_a;             // 12 bits assignment
                b <= {12'h000, in_b};  // 24 bits assignment
                p <= 0;                // 24 bits assignment
                j  <= 0;               //  4 bits assignment
                state <= Loop;         // Jump to next state
            end
        end

        Loop:                      // Second state
        begin
            if (a[0])
                p <= p + b;            // Conditional operation
            a <= a >> 1;               // Unconditional operations
            b <= b << 1;               // The order ...
            j  <= j + 1;               //  ... is decided by user
            if (j == 11) begin
                done <= 1;             // Indicate it finish
                state <= Idle;         // Conditional jump
            end
        end

        default:                   // Because of security reasons
            state <= Idle;

    endcase
end
```

A second change proposed in this paper –the initialization box at the beginning of the ASM++ diagram will be explained later– is to use different operation boxes for *assignments* and *assertions*. All signals used in the previous example have a synchronous behavior, so they are all assigned to new values *at the end* of each clock cycle. But other signals may need to be asserted to a different value *during* the current clock cycle, asynchronously.

The following example, where a synchronous RAM memory is initialized to a fixed value, illustrates this idea. As above, a C-like notation for all written expressions is also suggested, but most designers may prefer the use of VHDL-like or Verilog-like expressions.

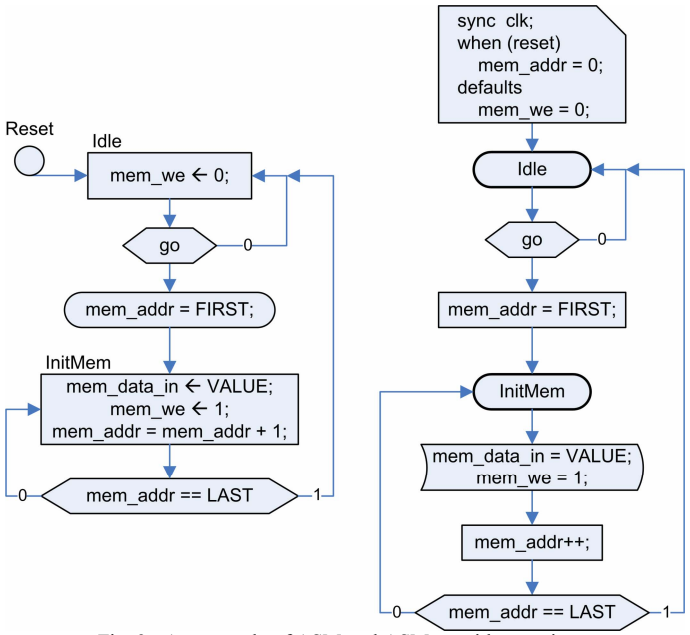

Fig. 3. An example of ASM and ASM++ with assertions.

Traditional ASM diagrams (left side of fig. 3) use different operators ("=" and "←") for synchronous assignments and asynchronous assertions. We propose (as shown on the right side of the same figure) the use of *rectangular* boxes for assignments, but boxes with *bowed* sides for assertions. Thus, the same equal symbol ("=") can be used for both, assignments and assertions.

The Verilog code for this diagram is shown below. 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.

```
// Use `define to set FIRST, LAST and VALUE.
parameter  Idle = 1'b0,  InitMem = 1'b1;
always @ (posedge clk or posedge reset)
begin
    if (reset) begin
        state <= Idle;
        mem_addr <= 0;
    end else case (state)

        Idle:
        begin
            if (go) begin
                mem_addr <= `FIRST;
                state <= InitMem;
            end
        end

        InitMem:
        begin
            mem_addr <= mem_addr + 1;
            if (mem_addr == `LAST)
                state <= Idle;
        end

        default:
            state <= Idle;

    endcase
end
assign mem_we      = (state == InitMem) ? 1'b1 : 1'b0;
assign mem_data_in = `VALUE;
```

This is the main reason for differentiating synchronous assignments –codified into an 'always' block– from asynchronous assertions –codified out of it. When the code is handwritten [11][12], 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.

The third proposed change is the introduction of a specific box for the initialization sequence and other global definitions. It has at least two applications:

- When the asynchronous reset signal arrives, the circuit must go immediately to a well defined state, 'Idle' in these examples. But this duty is usually not enough for most circuits, where other signals also need to be initialized. This is the case of the 'done' signal in fig. 2, that needs to be asserted to '0'. This one would be the default behavior.
- When synchronous signals are not used in one or more states, the default behavior of the circuit must be "to keep 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 preferred value for those situations is "don't care", as happens with 'mem_data_in' signal in fig. 3. But in the same figure, for example, signal 'mem_we' must be tied to '0' when not used. We propose the use of an optional "defaults" section at the beginning of all diagrams, because the alternative to it is to assert those signals in all states that do not use them, as shown in 'Idle' state with traditional notation.

For an ASM++ compiler –a program that generates Verilog and/or VHDL code from the ASM++ diagram– additional sections are required in this box (see fig. 4): 'in', 'out' and 'inout' must be used to declare the name, behavior and size of all inputs and outputs; 'signal' is required to declare the size of all signals –their behavior will be described later by the diagram–; 'define' can be used for definitions, 'sync' is needed for synchronous circuits and 'design' will be used to specify the design name and optionally its parameters/generics. If other sub-modules were hierarchically connected to this one, they may also be declared using this box. Anyway, these ones are only several examples.

With all these changes, ASM++ diagrams are obviously less compact, but more consistent. They are now easier to read and understand, and thanks to these changes they can be processed by CAD tools.

IV. A Complete ASM++ Example

A final example is shown in fig. 4, where a simplified version of a FIFO –with no generation of 'full' or 'empty' signals– is designed. Thanks to the global box this ASM++ diagram may lead to a Verilog/VHDL synthesizable code, because a compiler has all needed information to generate it. This code (not generated automatically yet, the compiler is still under work) can be seen later.

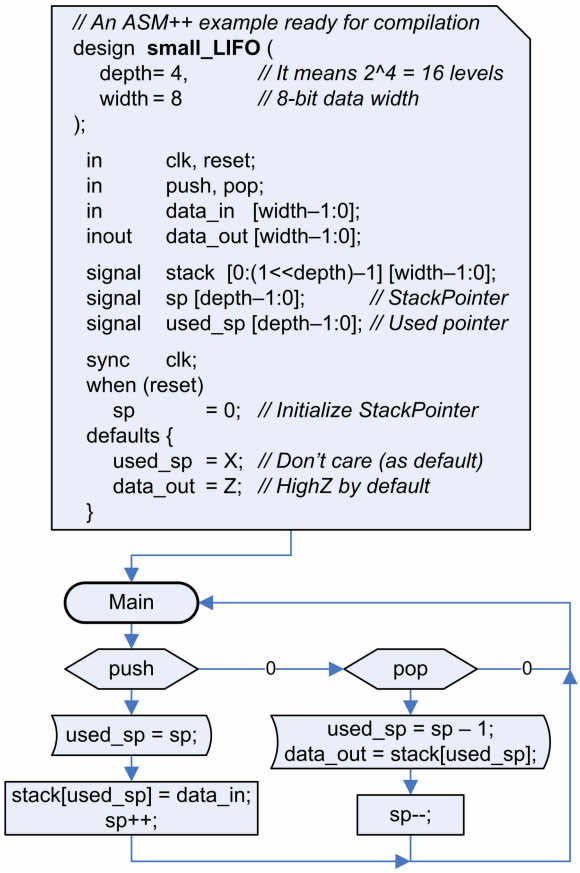

Fig. 4. A full ASM++ example ready for compilation.

```
// An ASM++ example ready for compilation
module small_LIFO (clk, reset, push, pop, data_in, data_out);

   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;

   reg      [width–1:0]     stack  [0:(1<<depth)–1];
   reg      [depth–1:0]     sp;         // StackPointer
   wire     [depth–1:0]     used_sp;    // Used pointer

   always @ (posedge clk or posedge reset)
   begin
      if (reset)        sp <= 0;       // Initialize StackPointer
      else if (push)    sp <= sp + 1;
      else if (pop)     sp <= sp – 1;
   end

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

   assign used_sp = push ? sp : sp – 1;
   assign data_out = ~push & pop ? stack[used_sp] : {width{1'bz}};

endmodule    /// small_LIFO
```

This circuit has a state box, named 'Main', but it has no states at all. Indeed, a circuit only needs states if it has two or more states, but this FIFO only has one. Additionally, for 'used_sp' and 'data_out' signals, the state box has been used as the beginning and end of their description, but there is no relation with any clock signal because they are asynchronous ones.

V. Conclusions

This article has presented an alternative notation for ASM diagrams. It makes diagrams easier to read, write and understand for large designs. It can be used as a primary tool for design or as a suitable representation for supervision and documentation.

The ASM++ notation can be Verilog/VHDL independent if the C-like proposal for expressions is accepted. It gives more freedom to designers to decide the writing order of operations and allows the specification of signal widths. In future releases it will also allow multiple threads and multiple clock sources. At last, this proposal is a more consistent notation, free from lateral annotations, thus more convenient for CAD tools. An ASM++ compiler is under work.

Acknowledgments

The authors would like to acknowledge the financial support of eZono GmbH, Jena, Germany. Our thanks also to Dolores García, David Ferrer, Ruben Herrero and Roberto Campo, students at the University of Valladolid, Spain, for their work and comments about this notation and methodology.

References

[1] C.R. Claire, *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*, revised 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] J.P. Hayes, *Introduction to Digital Logic Design*, Prentice-Hall, 1993.

[6] 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.

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

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

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

[10] 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.

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

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