ASM++ charts: an intuitive circuit representation ranging from low level RTL to SoC design

S. de Pablo, L.C. Herrero, F. Martínez
University of Valladolid
Valladolid (Spain)
sanpab@eis.uva.es

M. Berrocal
eZono AG
Jena (Germany)
manuel@ezono.com

Abstract

This article presents a methodology to describe digital circuits from register transfer level to system level. When designing systems it encapsulates the functionality of several modules and also encapsulates the connections between those modules. To achieve these results, the possibilities of Algorithmic State Machines (ASM charts) have been extended to develop a compiler. Using this approach, a System-on-a-Chip (SoC) design becomes a set of linked boxes where several special boxes encapsulate the connections between modules. The compiler processes all required boxes and files, and then generates the corresponding HDL code, valid for simulation and synthesis. A small SoC example is shown.

1. Introduction

System-on-a-Chip (SoC) designs integrate processor cores, memories and custom logic joined into complete systems. The increased complexity requires more effort and more efficient tools, but also an accurate knowledge on how to connect new computational modules to new peripheral devices using even new communication protocols and standards.

A hierarchical approach may encapsulate on black boxes the functionality of several modules. This technique effectively reduces the number of components, but system integration becomes more and more difficult as new components are added every day.

Thus, the key to a short design time, enabling "product on demand", is the use of a set of predesigned components which can be easily integrated through a set of also predesigned connections, in order to build a product.

Because of this reason, Xilinx and Altera have proposed their high end tools named Embedded Development Kit [1] and SoPC Builder [2], respectively, that allow the automatic generation of systems. Using these tools, designers may build complete SoC designs based on their processors and peripheral modules in few hours. At a lower scale, similar results may be found on the Hardware Highway (HwHw) web tool [3].

On the language side a parallel effort has been observed. In particular, SystemVerilog [4] now include an 'interface' element that allow designers to join several inputs and outputs together in one named description, so textual designs may become easier to read and understand. At a different scale, pursuing a higher level of abstraction, the promising SpecC top-down methodology [5] firstly describes computations and communications at an abstract and untimed level, and then descends to an accurate and precise level where connections and delays are fully described.

The aim of this paper is to contribute to these efforts from a bottom-up point of view, mostly adequate for academic purposes. First of all, we present several extensions to the Algorithmic State Machine (ASM) methodology, what we have called "ASM++ charts", allowing the automatic generation of VHDL or Verilog code from this charts, using a recently developed ASM++ compiler. Furthermore, these diagrams may describe hierarchical designs and define, through special boxes, how to connect different modules all together.

2. ASM++ charts

The Algorithmic State Machine (ASM) method for specifying digital designs was originally documented on 1973 by C.R. Clare [6], who worked at the Electronics Research Laboratory of Hewlett Packard Labs, based on previous developments made by T. Osborne at the University of California at Berkeley [6]. Since then it has been widely applied to assist designers in expressing algorithms and to support their conversion into hardware [7-10]. Many texts on digital logic design cover the ASM method in conjunction with other methods for specifying Finite State Machines (FSM) [11-12].

A FSM is a valid representation of the behavior of a digital circuit when the number of transitions and the complexity of operations is low. The example of fig. 1 shows a FSM for a 12x12 unsigned multiplier that computes *'outP = inA * inB'* through twelve conditional additions. It is fired by a signal named *'go'*, it signals the answer using *'done'*, and indicates through *'ready'* that new operands are welcome.

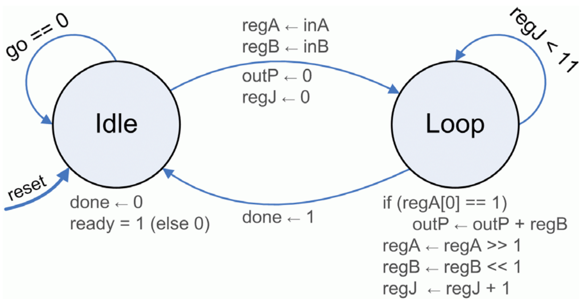

Figure 1. An example of FSM for a multiplier.

However, on these situations traditional ASM charts may be more accurate and consistent. As shown at fig. 2, they use three different boxes to fully describe the behavior of cycle driven RTL designs: a "state box" with rectangular shape defines the beginning of each clock cycle and may include unconditional operations that must be executed *during* (marked with '=') or *at the end* (using the delay operator '←') 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" –with oval shape– indicate those operations that are executed during the same clock cycle, but only when previous conditions are valid. Additionally, an "ASM block" includes all operations and decisions that are or can be executed simultaneously during each clock cycle.

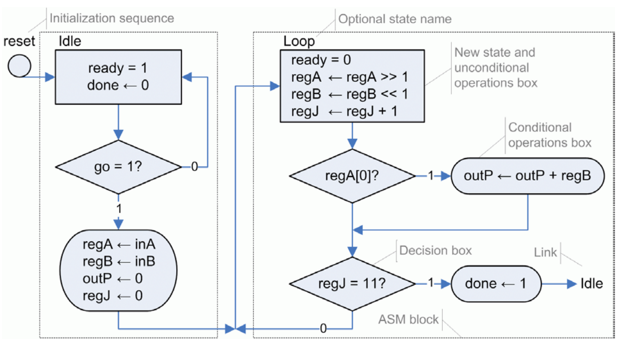

Figure 2. Traditional ASM chart for a multiplier.

The advantages of FSM for an overall description of a module are evident, but the ASM representation allows more complex designs through conditions that are introduced incrementally and detailed operations located where designer specifies.

However, ASM notation has several drawbacks:

- They use the same box, rectangular ones, 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. 2).
- Finally, the width of signals and ports cannot be specified when using the current notation.

Proposed ASM++ notation [13-14] tries to solve all these problems and extend far beyond the possibilities of this methodology. The first and main change introduced by this new notation, as seen at fig. 3, is the use of a specific box for states –we propose oval boxes, very similar to those circles used in bubble diagrams– thus now all operations may share the same box, a rectangle for synchronous assignments and a rectangle with bent sides for asynchronous assertions. Diamonds are kept for decision boxes because they are commonly recognized and accepted.

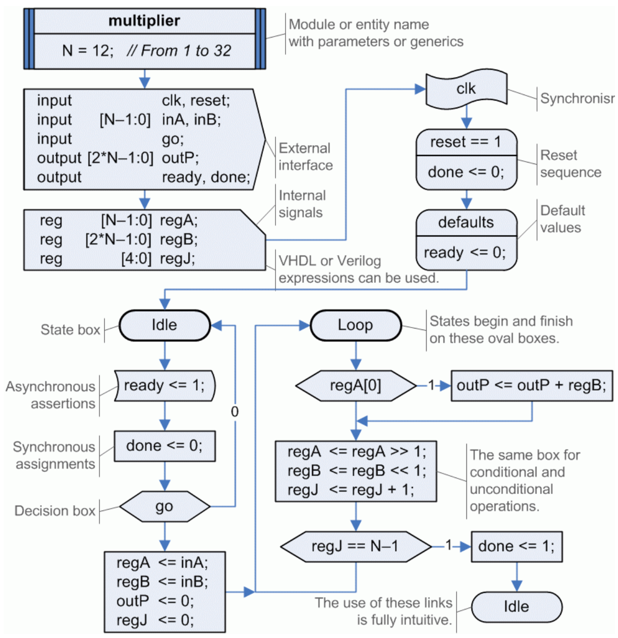

Figure 3. ASM++ chart ready for compilation.

Figure 3 shows additional features of ASM++ charts, included to allow their automatic compilation to generate HDL code. In addition to an algorithmic part, a declarative section may describe the design name, its implementation parameters, the external interface, one or more internal signals. The synchronization signal and its reset sequence can be fully specified in a very intuitive way too. A box for 'defaults' has been added to easily describe the circuit behavior when any state leave any signal free. Furthermore, all boxes use standard VHDL or Verilog expressions, but never both of them; the ASM++ compiler usually detects the HDL and then generates valid HDL code using the same language.

3. Hierarchical design using ASM++ charts

As soon as a compiler generates the VHDL or Verilog code related to an ASM++ chart, the advanced features of modern HDL languages can be easily integrated on them. The requirements for hierarchical design have been included through the following elements:

- Each design begins with a 'header' box that specifies the design name and, optionally, its parameters or generics.
- Any design may use one or several pages on a *MS Visio 2007* document[1], saved using its VDX format. Each VDX document may include several designs identified through their header boxes.
- Any design may instantiate other designs, giving them an instance name. As soon as a lower level module is instantiated, a full set of signals named "instance_name.port_name" (see fig. 5) is created to ease the connections with other elements. Later on, any 'dot' will be replaced by an 'underline' because of HDL compatibility issues.
- When the description of an instantiated module is located on another file, a 'RequireFile' box must be used before the header box to allow a joint compilation. However, the ASM++ compiler identifies any previously compiled design to avoid useless efforts and invalid duplications.
- VHDL users may include libraries or packages using their 'library' and 'use' sentences, but also before any header box.
- Nowadays, compiler does not support reading external HDL files, in order to instantiate hand written modules. A prototype of them, as shown at fig. 4, can be used instead.

Using these features, an example with a slightly improved multiplier can be easily designed. First of all, a prototype of a small FIFO memory is declared, as shown at fig. 4, thus compiler may know how to instantiate and connect this module, described elsewhere on a Verilog file. Then three FIFO memories are instantiated to handle the input and output data flows, as shown at fig. 5, so several processors may feed and retrieve data from this processing element.

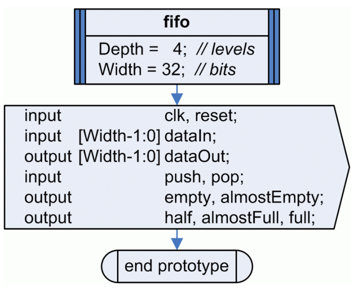

Figure 4. A prototype of an external design.

[1] Actually, designers may also use *MS Visio 2003* or *ConceptDraw*. However, the only supported file format is VDX.

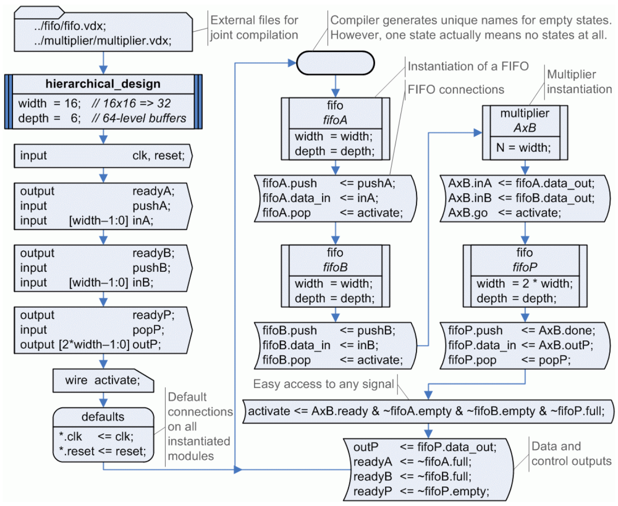

Figure 5. An example of hierarchical design.

The ASM++ chart of fig. 5 can be compared with its arranged compilation result, shown below. The advantages of this methodology on flexibility, clarity and time saving are evident. Not always a text based tool is faster and more productive than a graphical tool.

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

  parameter   width  = 16;      // 16x16 => 32
  parameter   depth =   6;      // 64-level buffers

  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;

  wire                  fifoA_clk,      fifoA_reset;
  wire   [width-1:0]    fifoA_dataIn, fifoA_dataOut;
  wire                  fifoA_push,     fifoA_pop;
  wire                  fifoA_empty,    fifoA_full;
  fifo  # (
    .width(width), .depth (depth)
  )  fifoA  (
    .clk       (fifoA_clk),         .reset      (fifoA_reset),
    .data_in (fifoA_dataIn),     .data_out (fifoA_dataOut),
    .push      (fifoA_push),        .pop          (fifoA_pop),
    .empty     (fifoA_empty),      .full          (fifoA_full)
  );

  wire                  fifoB_clk,      fifoB_reset;
  wire   [width-1:0]    fifoB_dataIn, fifoB_dataOut;
  wire                  fifoB_push,     fifoB_pop;
  wire                  fifoB_empty,    fifoB_full;
  fifo  # (
    .width(width), .depth (depth)
  )  fifoB  (
    .clk       (fifoB_clk),         .reset      (fifoB_reset),
    .data_in (fifoB_dataIn),     .data_out (fifoB_dataOut),
    .push      (fifoB_push),        .pop          (fifoB_pop),
    .empty     (fifoB_empty),      .full          (fifoB_full)
  );
```

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

wire                 fifoP_clk,      fifoP_reset;
wire [2*width-1:0]   fifoP_dataIn, fifoP_dataOut;
wire                 fifoP_push,     fifoP_pop;
wire                 fifoP_empty,    fifoP_full;
fifo # (
  .width(2 * width), .depth (depth)
) fifoP (
  .clk      (fifoP_clk),       .reset     (fifoP_reset),
  .data_in  (fifoP_dataIn),    .data_out  (fifoP_dataOut),
  .push     (fifoP_push),      .pop       (fifoP_pop),
  .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_dataIn    = inA;
assign  fifoA_pop       = activate;

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

assign  AxB_inA         = fifoA_dataOut;
assign  AxB_inB         = fifoB_dataOut;
assign  AxB_go          = activate;

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

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

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

endmodule   /// hierarchical_design
```

4. Encapsulating connections using *pipes*

Following this bottom-up methodology, the next step is using ASM++ charts to design full systems. As stated above, a chart can be used to instantiate several modules and connect them, with full, simple and easy access to all port signals.

However, system designers need to know how their available IP modules can or must be connected, in order to build a system. Probably, they need to read thoroughly several data sheets and try different combinations, to finally match their requirements. Nonetheless, when they become experts on those modules, newer and better IP modules are developed, so system designers must start again and again.

This paper presents an alternative to this situation, called *"Easy-Reuse"*. During the following explanations, please, refer to figures 6 to 9.

- First of all, a fully new concept must be introduced: an ASM++ chart may describe an entity/module that will be *instantiated*, like 'multiplier' at fig. 3, but additionally it may be used for a description that will be *executed* (see figs. 8 and 9). The former will just instantiate a reference to an outer description, meanwhile the later will generate one or more sentences inside the modules that call them. To differentiate those modules that will be executed, header boxes enclose one or more module names using '<' and '>' symbols. Later on, these descriptions will be processed each time an instance or a *'pipe'* (described below) calls them.
- Furthermore, the ASM++ compiler has been enhanced with PHP-like variables [15]. They are immediately evaluated during compilation, but they are available only at compilation time, so no circuit structures will be directly inferred from them. Their names are preceded by a dollar sign ('$'), they may be assigned with no previous declaration and store integer values, strings or lists of freely indexed variables.
- In order to differentiate several connections that may use the same descriptor, variables are used instead of parameters or generics. The corresponding field at a header box, when using it to start a connection description, is used to define default values for several variables (see fig. 8); these specifications would be changed by *pipes* on each instantiation (see fig. 6).
- Usual ASM boxes are connected in a sequence using arrows with sense; a new box called *"pipe"* can be placed out of the sequence and connect two instances through single lines, with no arrows.
- When compiler finishes the processing of the main sequence, it searches all *pipes*, looks for their linked instances, and executes the ASM charts related to those connections. Before each operation, it defines two automatic variables to identify the connecting instances. As said above, the *pipe* itself may define additional variables to personalize and differentiate each connection.
- As soon as several *pipes* may describe connections to the same signal, a resolution function must be defined to handle their conflicts. A tristate function would be used, but HDL compilers use to refuse such connections if they suspect contentions; furthermore, modern FPGAs do not implement such resources any more because of their high consumption, thus these descriptions are actually replaced by gate-safe logic. Subsequently, a wired-OR, easier to understand than a wired-AND, has

been implemented when several sources define different values from different *pipe* instantiations or, in general, from different design threads.

- The last element required by ASM++ charts to manage automatic connections is *conditional compilation*. A diamond-like box, with double lines at each side, is used to tell the ASM++ compiler to follow one path and fully ignore the other one. Thus, different connections are created when, for example, a FIFO memory is accessed from a processor to write data, to read data or both.

Using these ideas, a SoC design may now encapsulate not only the functionality of several components, but also their connections.

Figure 6 describes a small SoC that implements a Harvard-like DSP processor (see [13]) connected to a program memory, a 32-level FIFO and a register. First of all, two C-like compiler directives are used to specify the HDL language and a definition used later; a VDX file that describes the DSP processor is also included before giving a name to the SoC design. Then, all required modules are instantiated and connected using *pipes*.

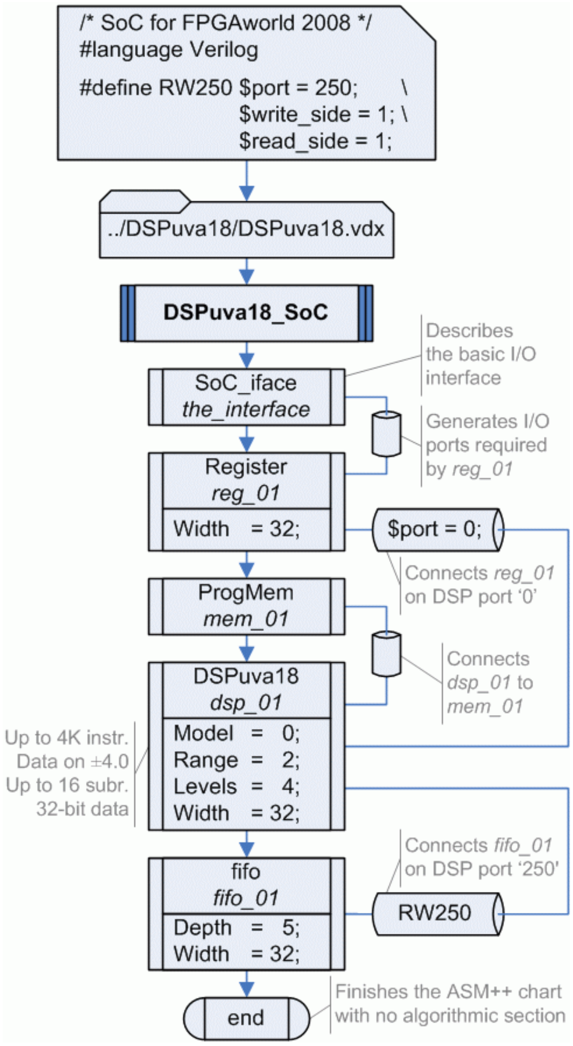

Figure 6. A small SoC design using *pipes*.

A small program memory has been designed for testing purposes, as shown at fig. 7: the upper chart describes a ROM memory with a short program that emulates the behavior of a Xilinx Block RAM, and the lower chart describes how this synchronous memory must be connected to the DSP. This figure illustrates the use of automatic variables ('$ProgMem' and '$DSPuva18', whose values will be "mem_01" and "dsp_01", respectively) and the difference between modules that can be instantiated or executed.

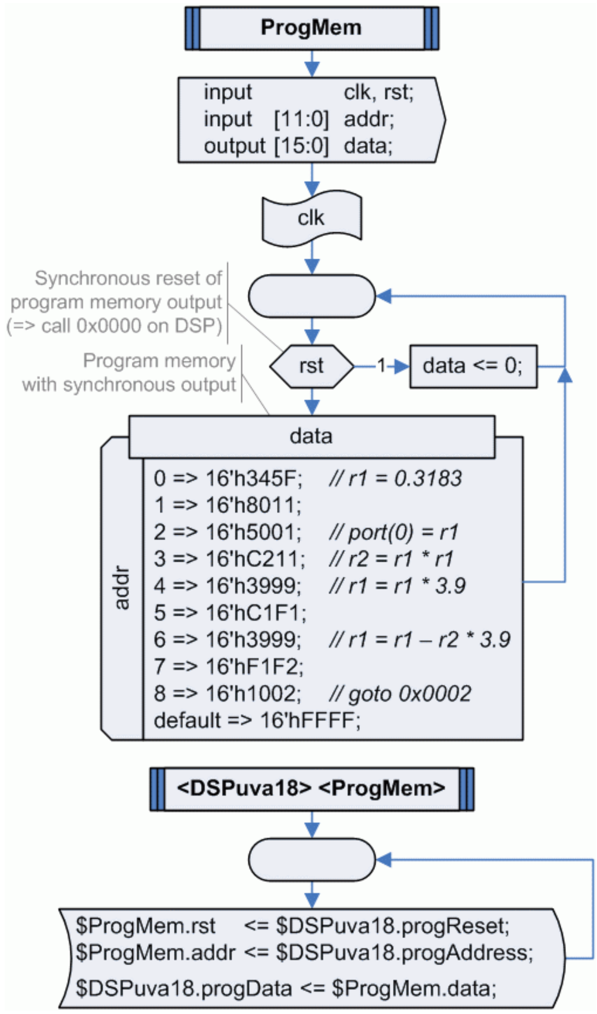

Figure 7. Charts may describe connections.

The *pipe* at figure 6 with text "RW250" describes the connection of a FIFO memory (see fig. 4) to a DSPuva18 processor [13], thus it executes the ASM++ chart shown at fig. 8. When executing this pipe, a '0' value is firstly assigned to variables '$port', '$write_side' and '$read_side', as stated by the header box; then these values are changed as specified by the *pipe* box (see the defined value of 'RW250'); finally, the chart of figure 8 generates the HDL code that fully describes how *"fifo_01"* device is connected to *"dsp_01"* processor for reading and writing using port '250' for data and port '251' for control (getting the state through a read and forcing a reset through a write).

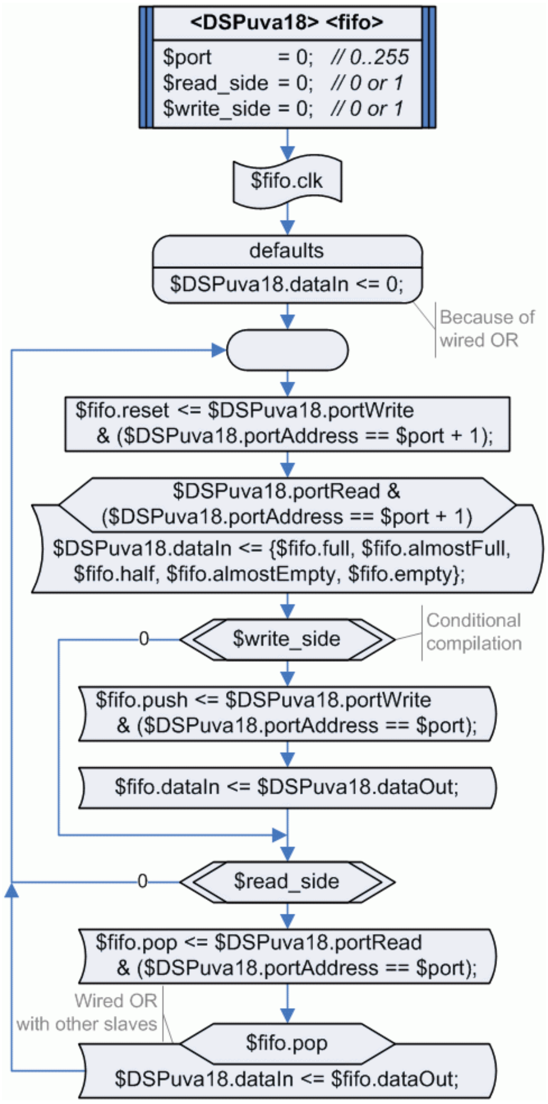

Figure 8. An ASM++ chart that describes how a FIFO must be connected to a DSP processor.

Two final ASM++ charts will be described at figure 9, but other required charts have not been included for shortness. The chart at left specifies how the instance named 'SoC_iface' at figure 6 must be executed, not instantiated, in order to generate two control inputs and to connect them to all modules. The diagram at right generates additional I/O signals and connects them to the register controlled by the DSP through its port '0'.

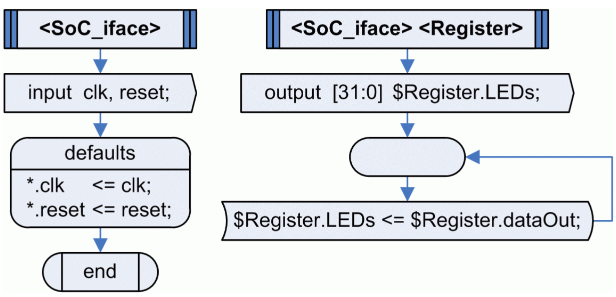

Figure 9. Charts may describe I/O interface too.

Several sentences of the HDL code generated by the ASM++ compiler when processing these diagrams are displayed following, revealing that ASM++ charts are fully capable of describing SoC designs using an intuitive, easy to use and consistent representation.

```
// I/O interface described by 'SoC_iface' instance and pipe (see figure 9):
input          clk, reset;
output  [31:0] reg_01_LEDs;

// A connection described by "<SoC_iface> <Register>" pipe:
assign  reg_01_LEDs      = reg_01_dataOut;

// Connecting dsp_01 to mem_01, its program memory (see figure 6):
assign  mem_01_rst       = dsp_01_progReset;
assign  mem_01_addr      = dsp_01_progAddress;
assign  dsp_01_progData  = mem_01_data;

// Connecting reg_01 to dsp_01 (at port '0'):
assign  reg_01_we        = dsp_01_portWrite & dsp_01_portAddress == 0);
assign  reg_01_dataIn    = dsp_01_dataOut;

// Connecting fifo_01 to dsp_01 (at ports '250' and '251'):
always @ (posedge fifo_01_clk)
begin
    fifo_01_reset  <=  dsp_01_portWrite & (dsp_01_portAddress == 250 + 1);
end
assign  fifo_01_dataIn   = dsp_01_ dataOut;
assign  fifo_01_push     = dsp_01_portWrite & (dsp_01_portAddress == 250);
assign  fifo_01_pop      = dsp_01_portRead & (dsp_01_portAddress == 250);

// Connecting several sources to dsp_01 using a wired-OR:
assign  asm_thread_1017_dsp_01_dataIn =
    (dsp_01_portRead & (dsp_01_portAddress == 0)) ? reg_01_dataOut : 0;
assign  asm_thread_1021_dsp_01_dataIn =
    (fifo_01_pop) ? fifo_01_dataOut :
    (dsp_01_portRead & (dsp_01_portAddress == 250+1)) ?
    {fifo_01_full, fifo_01_almostFull, fifo_01_half, fifo_01_almostEmpty, fifo_01_empty} : 0;
assign  dsp_01_dataIn =
    asm_thread_1017_dsp_01_dataIn  |  asm_thread_1021_dsp_01_dataIn;
```

5. Conclusions

This article has presented a powerful and intuitive methodology for SoC design named *Easy-Reuse*. It is based on a suitable extension of traditional Algorithmic State Machines, named ASM++ charts, its compiler and a key idea: charts may describe entities or modules, but they also may describe connections between modules. The ASM++ compiler developed to process these charts in order to generate VHDL or Verilog code has been enhanced further to understand a new box called *pipe* that implements the required connections. The result is a self-documented diagram that fully describes the system for easy maintenance, supervision, simulation and synthesis.

6. Acknowledgments

The authors would like to acknowledge the financial support for these developments of eZono AG, Jena, Germany, ISEND SA, Valladolid, Spain, and the Spanish Government under grant CICYT ENE2007-67417/ALT.

References

[1] Xilinx, “Platform Studio and the EDK”, on-line at *http://www.xilinx.com/ise/embedded_design_prod/platform_studio.htm*, last viewed on July 2008.

[2] Altera, “SoPC Builder”, on-line at *http://www.altera.com/products/software/products/sopc/sop-index.html*, last viewed on July 2008.

[3] epYme workgroup, “HwHw: The Hardware Highway web-tool for fast prototyping in digital system design”, on-line at *http://www.epYme.uva.es/HwHw.php*, 2007.

[4] SystemVerilog, “IEEE Std. 1800-2005: IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language”, IEEE, 3 Park Avenue, NY, 2005.

[5] R. Dömer, D.D. Gajski and A. Gerstlauer, “SpecC Methodology for High-Level Modeling”, 9th IEEE/DATC Electronic Design Processes Workshop, 2002.

[6] C.R. Clare, *Designing Logic Systems Using State Machines*, McGraw-Hill, New York, 1973.

[7] D.W. Brown, “State-Machine Synthesizer – SMS”, Proc. of 18th Design Automation Conference, pp. 301-305, Nashville, Tennessee, USA, June 1981.

[8] J.P. David and 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, Banff, Alberta, Canada, July 2004.

[9] 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 (NEWCAS 2004), pp. 81-84, June 2004.

[10] D. Ponta and G. Donzellini, “A Simulator to Train for Finite State Machine Design”, Proc. of 26th Annual Conference on Frontiers in Education Conference (FIE'96), vol. 2, pp. 725-729, Salt Lake City, Utah, USA, November 1996.

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

[12] Roth, *Fundamentals of Logic Design*, 5th edition, Thomson-Engineering, 2003.

[13] S. de Pablo, S. Cáceres, J.A. Cebrián and M. Berrocal, “Application of ASM++ methodology on the design of a DSP processor”, Proc. of 4th FPGAworld Conference, pp. 13-19, Stockholm, Sweden, September 2007.

[14] S. de Pablo, S. Cáceres, J.A. Cebrián, M. Berrocal and F. Sanz, “ASM++ diagrams used on teaching electronic design”, International Joint Conferences on Computer, Information, and Systems Sciences, and Engineering (CISSE 2007), on-line conference, December 2007.

[15] The PHP Group, on-line at *http://www.php.net*, last release has been PHP 5.2.6 at May 1st, 2008.