

Automatic Conversion of MatLab/Simulink Models to HDL Models

Bogdan Sbarcea, Dan Nicula

Dept. of Electronics and Computers, Transilvania University of Brasov, Romania,

FCD Ltd. Israel

e-mail: nicula@fcd.co.il

Abstract – This paper presents an automatic conversion method from MatLab/Simulink models to HDL models, and the benefits of using this method. It is a recent approach, appeared in the last 3 years due to the DSP exponential growth.

Index terms – EDA (Electronic Design Automation) tools, DSP (Digital Signal Processing), ASIC (Application Specific Integrated Circuit), FPGA (Field Programmable Gate Array), HDL (Hardware Description Language).

I. INTRODUCTION

Automatic conversion from MatLab/Simulink models to HDL models has arisen when digital signal processing (DSP) became a focus technology, with expectations of exponential growth.

Until now, most of the DSP designs have been implemented on general-purpose digital signal processors. These general-purpose processors are relatively cheap, are supported by high quality and inexpensive programming tools, facilitate rapid implementation of DSP algorithms and provide flexibility in development, reprogramming and debug phases.

However, today's electronic systems require a high-speed data processing. These requirements exceed the possibilities of general-purpose DSPs. If the tendencies would be the same in the next few years, it will be a big gap between the speed requirements of the future technology and the possibilities provided by general-purpose DSPs.

The only alternative to general-purpose DSPs is to cast the algorithm into an ASIC for hardware acceleration. Usually, the system prototype is implemented on FPGA (because of the flexibility) and the final implementation will be on ASIC (because of the high speed and low cost for a big production).

Together with the introduction of the advanced FPGA architectures such as *Xilinx Virtex II*, *Xilinx Virtex II Pro* and *Altera Stratix*, the two companies developed tools for DSP designers, which combine the advantages of general-purpose DSPs and the FPGA performance. This paper presents *Sim2HDL*, an EDA (Electronic Design Automation) tool used for automatic conversion of non-specific Simulink model to a HDL model. On this market, there are a couple of alternative solutions: AccelChip has DSP Synthesis, Xilinx released *System Generator* and Altera came with *DSP Builder*. There are some differences between the last two and *Sim2HDL*:

- Both tools use a self-defined Simulink library for conversion while *Sim2HDL* uses a limited set of Simulink blocks from the original Simulink libraries.
- *DSP Builder* offers support only for Altera, *System Generator* offers support only for Xilinx, while *Sim2HDL* can offer support for both platforms as well as for various ASIC technologies. The keyword for *Sim2HDL* is “**retargeting**”. Hard macros (such as multipliers, RAMs, ROMs, etc) are technology dependent.
- Altera has a 51-bit bus limitation while *Sim2HDL* does not have any such limitation.
- *Sim2HDL* offers support for MatLab variables from workspace that can be introduced directly in the Simulink block without being necessary the wiring up to the top level.

Section II presents the internal structure of *Sim2HDL*. Section III shows the generation of the HDL structural model and how to build HDL behavioral models for Simulink indivisible blocks. Testing and integration issues are presented in section IV. The conclusions are shown in the last section.

II. INTERNAL STRUCTURE

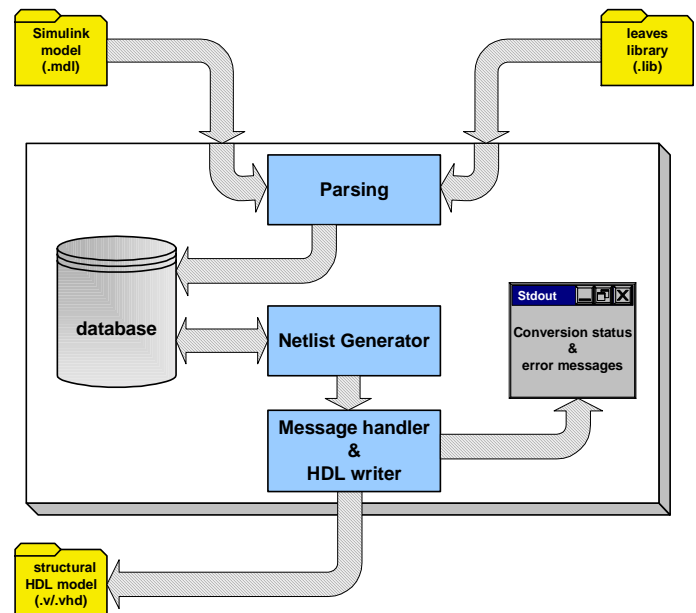


Fig 1. Sim2HDL internal structure

The software structure is shown in Fig 1. *Sim2HDL* has two input files, a Simulink *.mdl*, which contains the DSP algorithm description, and a library file *.lib* that contains the Simulink indivisible block instances that have a corresponding HDL behavioral description. The generated HDL structural model, together with all the behavioral models of the Simulink indivisible blocks are used in simulation and synthesis.

Four main blocks group all *Sim2HDL* functions:

- Parser
- Database
- Netlist generator
- Message handler and HDL code writer

A. Parser

The **Parser** block has been developed in ANSI C, lex and yacc. Lex and yacc are tools designed for compiler or code translation writers. Any application that looks for patterns in its input can be easily processed by lex and yacc. Furthermore, they allow a rapid application prototyping, easy modification, and simple maintenance of programs. This block is compound of two parts. The former part is responsible with parsing the Simulink code, while the latter one is responsible with parsing the leaves library.

- *Simulink parser*: An *.mdl* file opened in text mode, contains:
 - general parameters of the project,
 - description of all basic Simulink blocks used in the current project and
 - description of all Simulink subsystems, blocks and the interconnectivity.

The Simulink parser takes over all the significant parameters (the basic block’s relevant parameters, the block interconnectivity, etc) and stores them into the “*.mdl* database” in a suitable format.

- *Leaves library parser*:

The leaves library parser gets the port information (type and order) and fills in the “*.lib* database”.

Simulink block name instances must follow HDL naming conventions. However, if a block name has in composition an illegal character, the **Parser** functions will fix this inconvenience and the Simulink block is automatically renamed.

B. Database

The **Database** structure is presented in Fig 3, and it is developed in DataDraw. It is structured in 2 parts, each part communicating with its corresponding parser.

- “*.mdl* database” contains all the information about the Simulink blocks (parameters from “Block parameters” window and input/output ports) and about the interconnectivity between the blocks. “**Model**” stores only the name of the top module. “**System**” stores all the Simulink blocks that have in composition other blocks, while “**Line**” is filled in with the block

interconnections. “**Source**” and “**Dest**” objects have links to that block and port where “**Line**” is connected. In “**Block**” object are found all the relevant properties of a Simulink indivisible block. The building of an HDL behavioral model is based on these properties. “**Input**” and “**Output**” are in fact the **Block**’s input and output ports.

- “*.lib* database” contains information about the instances of Simulink indivisible blocks. “**Instance**” stores the type of the Simulink block, while “**Port**” specifies every port direction, location and name.

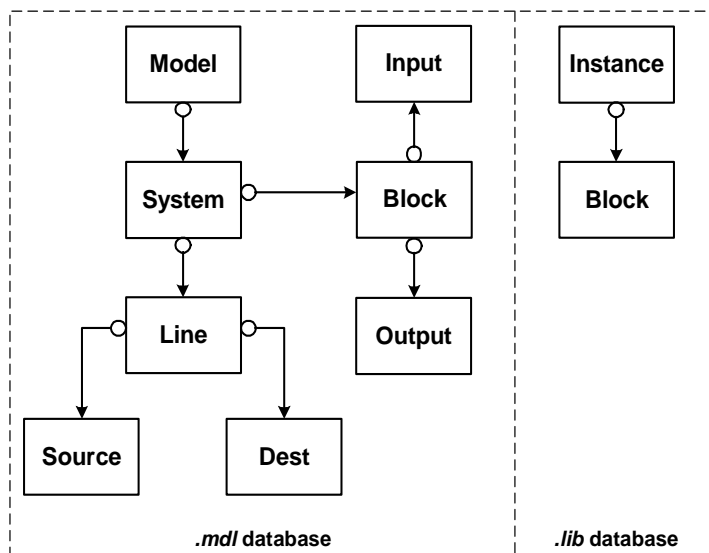


Fig 2. Database structure

C. Netlist Generator

Netlist Generator is compound of a set of functions that query the database and another set that process the information received from the database. The result is passed forward to Message Handler & HDL Writer. This block builds the HDL structural model. It is mandatory to establish all the net properties (width and sign) in order to obtain a correct HDL model. **Netlist Generator** has such of functions that settle the both net properties and Simulink input-output block properties (not specified in all cases).

D. Message Handler & HDL Writer

Message Handler & HDL Writer block is responsible with writing the HDL code and signaling if an error occurs. The modules are printed iteratively, beginning from the top module. Changing the output HDL language, (Verilog or VHDL) does not affect the previous described blocks. It is an option to have a Simulink indivisible block behavioral description in both HDL languages. HDL netlist is generated following each hardware description language’s specific rules.

If the parser finds a Simulink indivisible block that does not have a corresponding instance in **leaves library**, the user is informed that his library does not support all Simulink blocks present in his design.

III. CREATING A HDL BEHAVIORAL MODEL FOR THE SIMULINK BLOCK

The HDL behavioral model is written in order to support all the Simulink block parameters, presented in a specific “Block Parameters” window. Fig 4 shows an example for MinMax block.

MinMax block is a masked S-function that outputs either the minimum or the maximum element of the inputs. It can be chosen which function to apply with the “**Function**” parameter list.

The algorithm developer specifies the number of input ports with the “**Number of input ports parameter**”. If the block has one input port, the input must be a scalar or a vector. The block outputs a scalar equal to the minimum or maximum element of the input vector.

“**Output data type and scaling**” specifies the output data type and scaling via the dialog box, or inherit the data type and scaling from the driving block or by back-propagation.

“**Output data type**” accepts signed or unsigned data types, any number of bits.

If “**Saturate to max or min when overflows occur**” is selected, fixed-point overflows saturate.

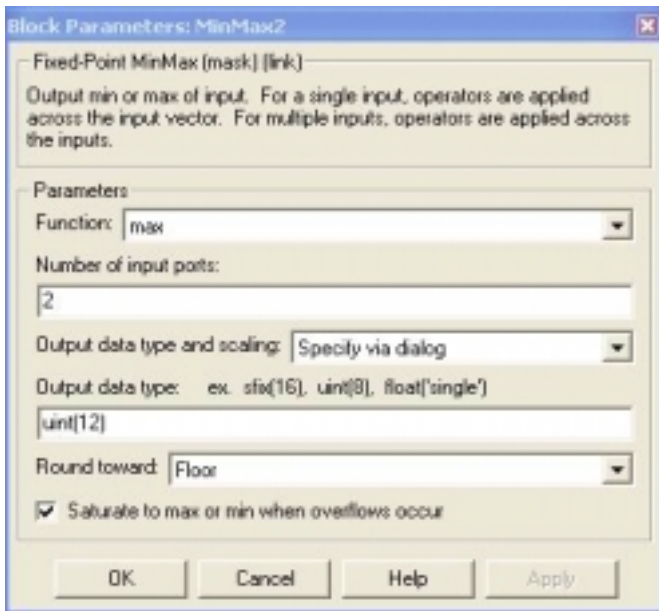


Fig 3. Block Parameters window for MinMax Simulink block

Based on the “Block Parameters” window, it is generated the *.mdl* code. A few parameters refer to the graphical arrangement of the Simulink block (**Position** and **ShowName**), block library (**BlockType**, **SourceBlock** and **SourceType**). “Block Parameters” window has direct access to the rest of the *.mdl* parameters. The mdl code below is a result of the previous described window:

```
Block {
  BlockType Reference
  Name       "MinMax2"
  Ports     [2, 1]
  Position  [200, 27, 255, 98]
  ShowName  off
  SourceBlock "fixpt_lib_4/Math/MinMax"
  SourceType "Fixed-Point MinMax"
  Function   "max"
  Inputs     "2"
  OutputDataTypeScalingMode "Specify via
                                dialog"

  OutDataType "uint(12)"
  OutScaling  "2^-10"
  LockScale  "off"
  RndMeth    "Floor"
  DoSatur    "on"
}
```

The Verilog behavioral description of MinMax model is shown below:

```
module MinMax (in1,in2, out);
  parameter width      = 32;
  parameter type_out = 0;
  // 0-signed; 1-unsigned
  parameter Function = 0;
  // 0-min; 1-max

  input      [width-1:0] in1;
  input      [width-1:0] in2;
  output reg [width-1:0] out;

  always @(in1 or in2)
  // min signed
    if((Function==0)&&(type_out==0))
      case({in1[width-1],in2[width-1]})
        2'b00: out<=(in1<in2)?in1:in2;
        2'b01: out<=in2;
        2'b10: out<=in1;
        2'b11: out<=(in1>in2)?in1:in2;
      endcase
  // min unsigned
    else if((Function==0)&&(type_out==1))
      out<=(in1<in2)?in1:in2;
  // max signed
    else if((Function==1)&&(type_out==1))
      out<=(in1>in2)?in1:in2;
  // max unsigned
    else if((Function==1)&&(type_out==0))
      case({in1[width-1],in2[width-1]})
        2'b00: out<= (in1>in2)?in1:in2;
        2'b01: out<= in1;
        2'b10: out<= in2;
        2'b11: out<= (in1<in2)?in1:in2;
      endcase
  endmodule
```

The HDL behavioral model was checked against the Simulink MinMax block in all possible parameter configurations, with random data. In the example shown below, MinMax module is instantiated in the Verilog netlist as follows:

```
defparam MinMax2.width = 12;
defparam MinMax2.type_out = 1;
defparam MinMax2.function = 1;
MinMax MinMax2 (
    .out(n72),
    .in1(n76),
    .in2(n78)
);
```

A very important observation is that for a specific set of parameters, the synthesis tool selects only the appropriate hardware structure. For example, MinMax behavioral model is synthesized with two set of parameters: Fig 5 presents the schematic resulted after synthesis for a 'Max' structure for unsigned input data (**width = 12, type_out = 1 and Function = 1**), while Fig 6 shows a Max structure for signed input data (**width = 12, type_out = 0 and Function = 1**).

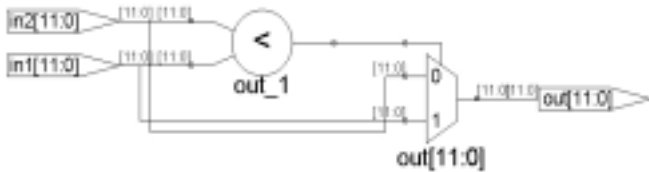


Fig 4. 'Max' function, unsigned input data type

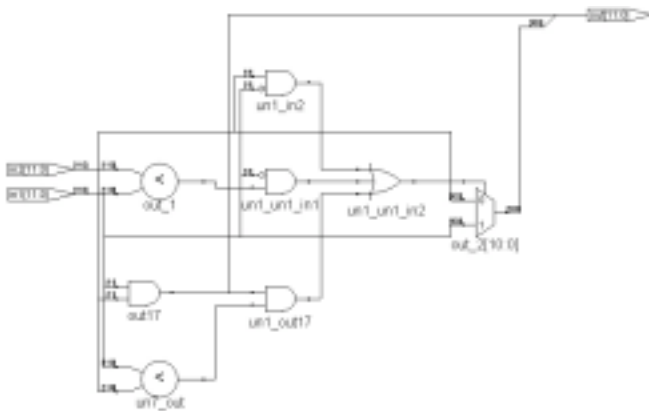


Fig 5. 'Max' function, signed input data type

It is obvious that from all the cases described in the HDL behavioral model, the synthesis tool has cut off the unnecessary logic. Therefore, the conclusion is that no matter how complex is the behavioral description, the synthesis tool will keep only the relevant logic for each instance, depending on the parameters introduced.

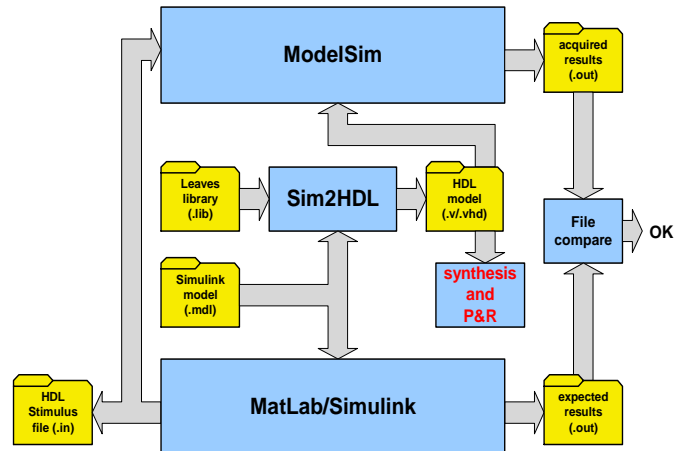


Fig 6. DSP design environment

IV. TESTING AND INTEGRATION

One of the significant benefits of the current approach of implementing DSP algorithms is the design data management. The main benefit is time saving. MatLab offers the test-bench, so the implementation engineer does not have to create a complicated test-bench in order to test his RTL model. The DSP developer tests intensively an algorithm before passing it to implementation engineer because a MatLab simulation consumes less time than a ModelSim simulation. Another thing is regarding the iterations between DSP developer and implementation engineer. Traditionally, a modification in the algorithm may take up to a couple of days to be re-designed. With *Sim2HDL*, a DSP design re-translation takes minutes. Any mistake or misinterpretation is avoided by using such an automatic translator.

V. CONCLUSIONS

This paper presented one of the most modern and recent methods to design digital systems and its main benefits. This method proved its efficiency by implementing various DSP algorithms on XILINX and ASIC platforms. Due to the drastic reduction of design time, iteration time between the DSP developer and implementation engineer and the absence of design misunderstandings, this automatic method of converting an algorithm into a hardware structure proved to be feasible.

REFERENCES

- [1] Automatic Conversion of Floating-point to Fixed-point MATLAB (www.accelchip.com)
- [2] True Top-Down DSP Design (www.accelchip.com)
- [3] Algorithms to Silicon (www.accelchip.com)
- [2] Xilinx System Generator for DSP (www.xilinx.com)
- [3] Altera DSP Builder User Guide (www.altera.com)
- [4] www.mathworks.com