FIPBLOX: a graphical interactive design tool for FIPSOC

Daniel Simonelli
INCA/INTIA – Facultad de Ciencias Exactas – UNICEN
Paraje Arroyo Seco, Campus Universitario – Tandil, Argentina
dsimonel@exa.unicen.edu.ar

Abstract: FIPSOC is a field programmable mixed-signal integrated device consisting of a Field Programmable Gate Array (FPGA), a set of programmable and interconnectable analog cells, and a microprocessor core which can run general purpose user programs, handle the dynamic reconfiguration of the programmable blocks and probe, in real time, internal digital and analog signals. This device is specially suitable for development and fast prototyping of mixed signal integrated applications.
As FIPSOC project is currently under development, it has no yet any powerful tool for synthesis and a structural VHDL (components) approach is to be used for designing. Therefore, the user starts from simple design structures and through a bottom-up style must build more complex components. In this paper we present FIPBLOX, a tool that allows the user automatically generate VHDL code for implementing and customizing high-level modules using the basic resources provided by the FIPSOC FPGA.

1. Introduction

Typically, electronic system designs may include a digital part, an analog part and a software program running on a microprocessor or microcontroller. Up to recent years, these three domains (digital, analog and software) have to be designed and prototyped separately, using different CAD tools and hardware parts for each one. As an alternative for integrating these more or less isolated domains, industry gave origin to ASICs with embedded IP-cores, the so-called SOCs (System On Chip).
FIPSOC (Field Programmable System On Chip) prototyping and integration system, consists of a mixed-signal Field Programmable Device (FPD) with a standard microprocessor core (an 8051), a suitable set of CAD tools to effortlessly program it, and a set of library macros and cells which support a number of typical applications to be easily mapped onto the FPD and migrated to an ASIC afterwards, if required. The advantage of this approach relies upon the fully integrated design and prototyping methodology that the user can follow with such a system, because he can download his application onto the programmable hardware and then use the internal microcontroller to test it in real time (both digital and analog). A powerful integrated set of user-friendly CAD tools is being designed, with the final target of letting the user specify, simulate, emulate (probe in real time) and map the complete design on to a single chip using one design environment. Also, a suitable library has been developed providing a very easy path for migration to ASIC after the prototyping phase. FIPBLOX tool aims to enlarge this design possibilities.

1.1. FIPSOC Overview

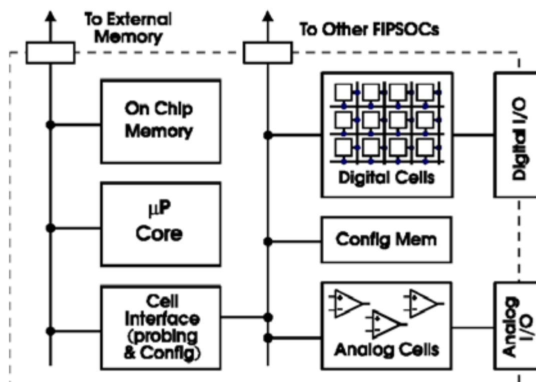

Fig. 1: Block diagram of the FIPSOC chip

The chip is a mixed signal field programmable device with an on-chip microcontroller. It includes a Field Programmable Gate Array (FPGA), a set of fixed functionality yet configurable analog cells, and a microprocessor core with RAM memory and some peripherals. The programmable digital and analog blocks are well defined and are separated from one another due to noise immunity considerations [Bra96]. Nevertheless, the different interfaces between these blocks themselves and to the microprocessor provide a very powerful interaction between software, digital hardware and analog hardware. FIPBLOX tool is specially concerned with the FPGA, so that configurable analog hardware, microcontroller core and interfaces between subsystems are not discussed here. For more detailed information regarding these topics refer to [Fau97a,b,c], [Lys93], [Mor97], [Hor97], [Nai97] and [Kra97].

1.1.1. Programmable Digital Hardware

The FIPSOC chip includes a two-dimensional array of programmable DMCs (Digital Macro Cell). The DMC is a large granularity, Look Up Table (LUT) based, synthesis targeted 4-bit wide programmable cell. Fig. 2 shows a simplified block diagram of the DMC.

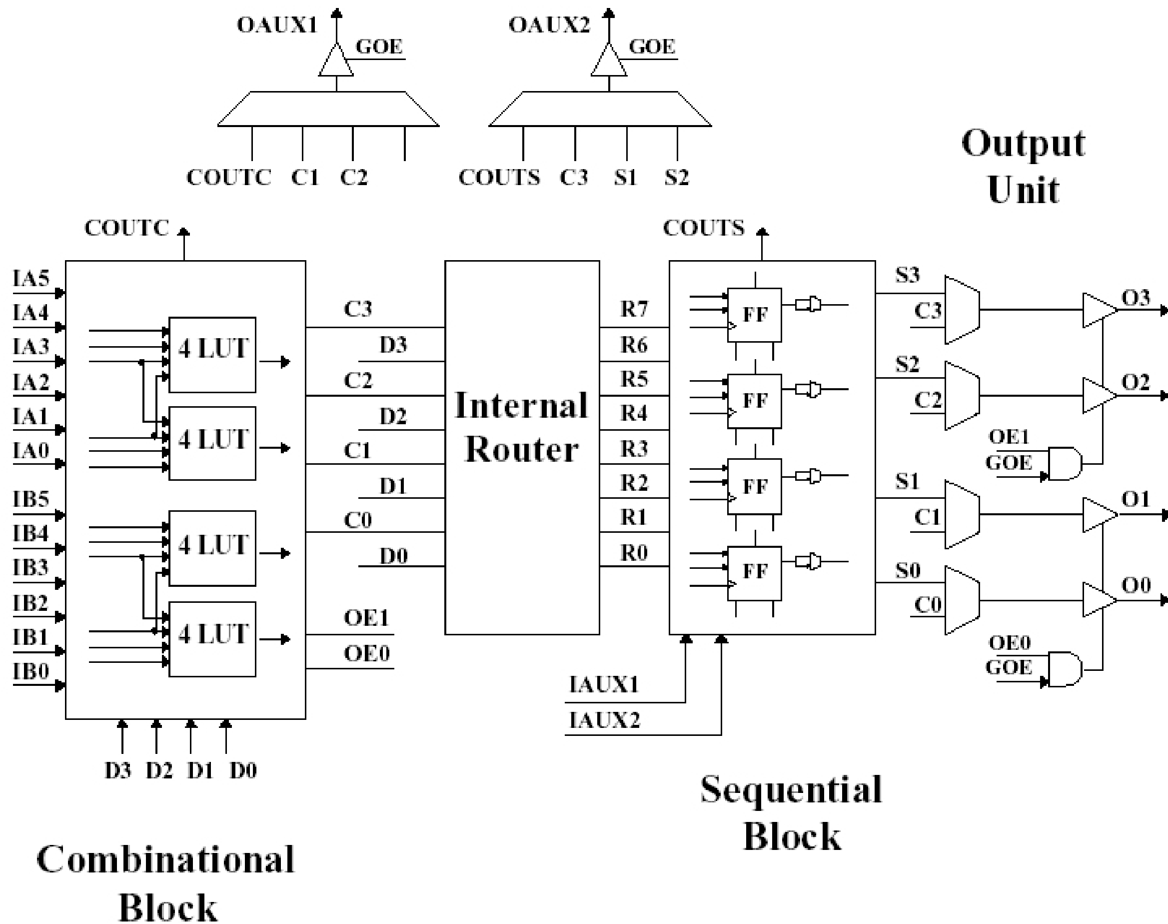

Fig. 2: Simplified DMC block diagram

The DMC has two main blocks: a combinational part, composed of four 4-input LUTs, and a sequential block including four FFs. Between them there is an internal router which provides the necessary connectivity, and makes it possible to feed direct inputs into the FFs rather than using the combinational outputs. This makes it possible to use the combinational and the sequential blocks

more or less independently. Each lookup table (LUT) can implement any Boolean function of 4 inputs. Every two 4-input LUTs share two inputs, and two LUTs can be combined to form a 5 input function or a 4 to 1 multiplexer (four inputs and two control bits). The four LUTs of a DMC can be combined to perform any 6 input Boolean function. The whole combinational part of the DMC can be configured as a 16x4 RAM memory (in fact, two independent 16x2 memories) or as a cascadable 4-bit adder or subtractor with carry-in and carry-out (also some other arithmetic functions are possible. See Table 1).
The sequential part of the DMC includes four two-input flipflops (FF), each of which can be independently configured as mux-type or enable-type, as latch or FF, and with synchronous and asynchronous set or reset. Again, the whole sequential part of the DMC can be configured as a cascadable shift register with load and enable or as a cascadable 4-bit up/down counter with load and enable.
These combinational and sequential macro functions are specially suitable to be used by synthesis programs [Sta95]. The routing architecture of this FPGA core has been designed according to this large granularity philosophy. Tracks spanning one, two and four DMCs (horizontally and vertically) are provided for general purpose interconnect.
Long lines spanning the whole height or width of a column or a row and dedicated tracks for global reset and clock spine distribution are provided. Interconnection switches composed of MOS transistors are controlled with RAM memory cells writeable by the microprocessor.

1.1.2. FIPSOC Standard Macro Block Library

This library is intended to help end-users to optimize routing resources and speed up their hardware designs. The macro blocks can be divided in:

· *Combinational Macros*: Set of standard macros related to the combinational sub-block of the DMC.
· *Sequential Macros*: Set of standard macros related to the sequential sub-block of the DMC.

COMBINATIONAL MODES	
ADD4: Basic Adder	The basic adder block adds two nibbles and can be propagated through the COUT output.
EADD4: Adder with Enable	The Adder with enable block adds two nibble only if enable is set; otherwise the result is 0.
SUB4: Subtraction	The sub4 block subtracts two nibbles
CPL4: Two Complementer	The CPL4 do the two complement of a nibble, and propagates the carry out through COUT
INC4: Incrementer	The INC4 block increments a nibble.
DEC4: Decrementer	The DEC4 block decrements a nibble.
MUX4: Multiplexer	2 nibbles multiplexer
MUL4: C-2 serial multiplier	The mul4 block multiplies a nibble by one bit (*and* function) and adds the result with a temp-adder nibble
RAM16x2: ram block	Ram of 16 words of 2 bits
SEQUENTIAL MODES	
COUNT4: 4-bit counter	In the Counter Macro Mode, the sequential part of the DMC is configured as a 4-bit up/down counter with load and enable.
SHIFT4: 4-bit shifter	When configured in this mode, the sequential part of the DMC becomes a shift register (SHR) with load and enable.

Table 1 – DMC macro modes

All of these macros are available to the final user as VHDL entities. FIPBLOX tool uses these entities as basic components for generating more complex structures (components). Generated VHDL code is fully compatible with Synopsys Synthesis Tool [SynSup] as this is the standard chosen by FIPSOC creators [Mad97]. It's worth noting that today's available FIPSOC CAE Tools, provide the ability of creating macros from the generated code so that new components can be used in the schematic editor.

2. FIPBLOX Tool

The front-end is based in Xilinx's Foundation 3.1i LogiBLOX [XilSup]. Fig. 3 shows its main screen.

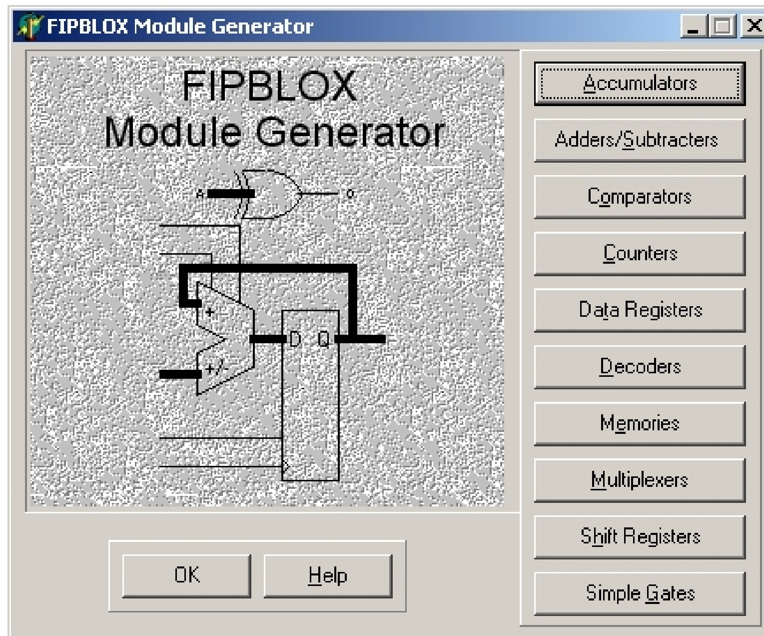

Fig. 3 – FIPBLOX Front-end

The algorithm for implementing n-size components is based on partitioning the required VHDL description into $k = \lceil n/4 \rceil$ basic components from the macro library and generating the corresponding signals for interconnecting them. FIPSOC Library provides a wide variety of Flip Flops and registers that cover easily the requirements for implementing the sequential part of components like accumulators which are not supported directly by macro library definitions. Code 1 shows the main VHDL structure involved for implementing a function with n-sized operands from 4-sized macro functions (*fn*4):

```
label: for i in 0 to k -1 generate
        comp1: fn4 port map(inp1(3+i*4),inp1(2+i*4),inp1(1+i*4),inp1(i*4),
                            inp2(3+i*4),inp2(2+i*4),inp2(1+i*4),inp2(i*4),
                            ............................,
                            inpj(i),
                            out1(3+i*4),out1(2+i*4),out1(1+i*4),out1(i*4),
                            ............................,
                            outm(i+1));
end generate;
```

Code 1 – Partitioning through generation of 4-sized components

As can be seen in Table 1, both combinational and sequential library components are nibble-sized. Remaining macros as simple gates, latches, flip flops and registers are available in sizes ranging 1 and 4, what adds flexibility through the use of minimum resources. This results in more efficient routing because of the locality at DMCs level.
Let's take as an example, the definition of a 16 bit accumulator, namely "Accu16", that accumulates (adding) the constant value 345. Carry input, not registered carry output and clock enable are required.
The input screen for the generation process is shown in fig. 4

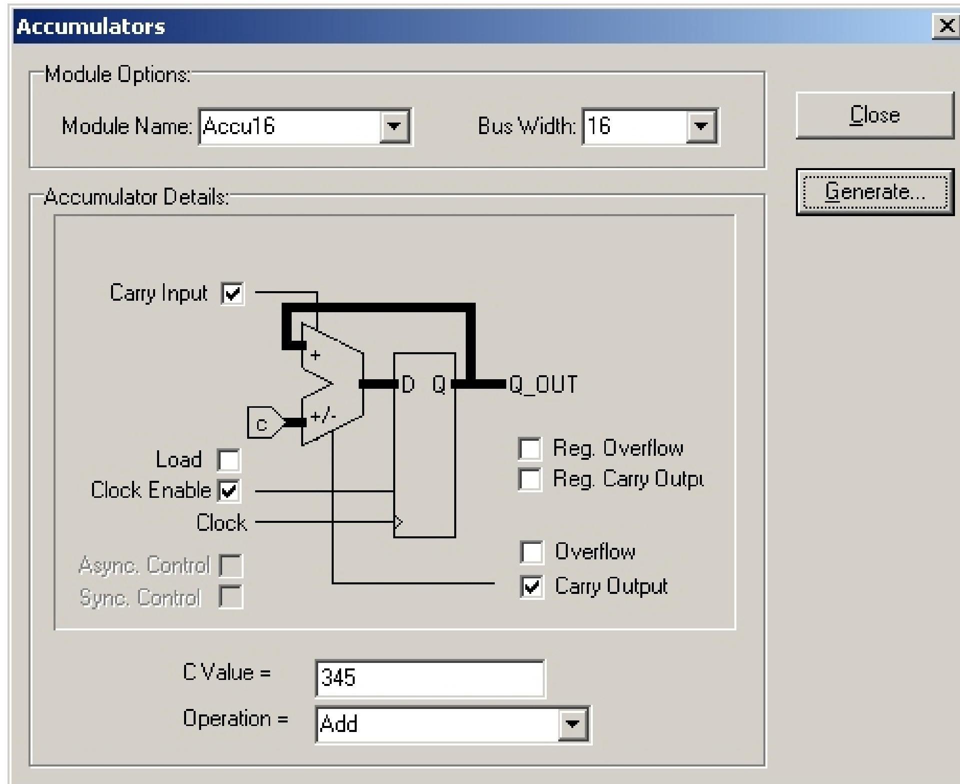

Fig. 4 – Accumulator generation screen

```
library ieee;
use ieee.std_logic_1164.all;
entity  Accu16 is
       port( cin: in std_logic;
              clkenable: in std_logic;
              clk: in std_logic;
              q_out: inout std_logic_vector(15 downto 0));
end Accu16;

architecture FIPSOC of Accu16 is

component add4
      port( A3, A2, A1, A0, B3, B2, B1, B0, CIN: in std_logic;
              Q3, Q2, Q1, Q0, COUT: out std_logic);
end component;

component SEQFFE
      port(D1, D2, D3, D4, CLK, E : in std_logic;  Q1, Q2, Q3, Q4 : out std_logic);
end component;
```

```
for all: s1 use entity work.add4(FIPSOC);
for all: regist use entity work.SEQFFE(FIPSOC);
signal c: std_logic_vector (0 to 4);
signal sint: std_logic_vector(15  downto 0);
signal cout: std_logic;
signal b: std_logic_vector(15 downto 0);

begin
     b <= "0000000101011001";
     c(0) <= cin;
     label1: for i in 0 to 3 generate
           comp1: s1 port map(q_out(3+i*4), q_out(2+i*4), q_out(1+i*4), q_out(i*4),
                              b(3+i*4),b(2+i*4),b(1+i*4),b(i*4),
                              c(i),
                              sint(3+i*4),sint(2+i*4),sint(1+i*4),sint(i*4),
                              c(i+1));
     end generate;

     cout <= c(4);

     label2: for i in 0 to 3 generate
          comp2: regist port map(sint(i*4),sint(1+i*4),sint(2+i*4),sint(3+i*4),
                     clk,
                     clkenable,
                     q_out(i*4), q_out(1+i*4), q_out(2+i*4), q_out(3+i*4));
     end generate;
end FIPSOC;
```

Code 2 – VHDL code generated for Accu16

2.1. FIPBLOX Modules

2.1.1. Accumulator Module

The Accumulator module takes the data on its B input port and its Carry Input port and adds this data to, or subtracts it from, the current value stored in the accumulator register. It then loads the result back into the register, making it available at the Q_OUT port.

There are two types of accumulators: those that accumulate the value applied to the symbol's B input port and those that accumulate a constant value.
The Carry Output and the Overflow outputs are generated by the Adder/Subtracter to indicate the status of the present arithmetic operation. Outputs can be latched by specifying the appropriate attribute. Also it is possible to load separate predefined values synchronously into any or all of the module's registers.

2.1.2. Adder/Subtracter Module

The Adder/Subtracter module adds or subtracts two data inputs and a Carry input. You can use this module as an adder, as a subtracter, or both. The Adders/Subtracters module provides a Carry Output and an Overflow output to indicate the status of the current arithmetic operation.
There are two types of Adder/Subtracter modules: those that add or subtract the values applied through two bus-pins, and those that add a constant to or subtract a constant from a value applied to one bus-pin.

The Adder/Subtracter module allows the user to register any or all of its outputs. Internally, if only add operation is selected the add4 macro library is used. Similarly, if only subtract is selected the sub4 macro library is used. If both operation modes are selected simultaneously, these two same macros are used together with a tristate buffer for choosing among them.

2.1.3. Comparator Module

The Comparator module compares the magnitude of two values, their equality, or both their magnitude and equality. There are two types of comparators: those that compare the values applied to the symbol's two bus-pins and those that compare a constant against the value applied to the A bus-pin. It is implemented by means of subtracters.

2.1.4. Counter Module

The Counter module generates a sequence of count values determined by the selected encoding and the status of the control inputs. The Counter module can be an up counter, down counter, or up/down counter with a predefined synchronous pre-load, and a dynamic synchronous parallel load. FIPBLOX counter modules can be loaded with a value applied to a bus-pin. Library macro COUNT4 is used for implementing n-sized counters.

2.1.5. Data Register Module

The Data Register module is used to capture the data applied to its D_IN port on active Clock transitions. The contents of the data register are always present on the Q_OUT port. The module is synthesized as an array of flip-flops that can be loaded with predefined asynchronous and synchronous data. There are several macro registers supplied as default in FIPSOC's macro library.

2.1.6. Decoder Module

The Decoder module converts unsigned binary data on the Select port to a 1-of-n one-hot output on the D_OUT port. The width (precision) of the Select and D_OUT ports is determined by the Bus Width attribute. This is a component completely synthesized by FIPBLOX tools because there is no any primitive in the macro library.

2.1.7. Memories Module

The Memories module allows the user to create ROM, RAM, Synchronous RAM, and Dual Port RAM modules. These modules store information in the form of words in a tabular fashion and in all cases use the library macro RAM16x2. These modules are to be used for the following purposes:

- Store information from a user-edited memory file (.txt file) and access that information as needed (ROM)

- Store dynamic information into memory and then read that information from memory (RAM)

Although it is possible create memories of up to 32 bits width with no any limitation in the depth, is strongly discouraged implementing big memories through DMC resources because of the small size of the FPGA. In these cases, is much better to access external memories through the internal microcontroller.

2.1.7.1. ROM Modules

The Read-Only Memory (ROM) module is a memory for storing information from a user-edited memory file (.txt file). The word storage capacity of the ROM is called the depth. The depth must be a multiple of 16.
The size of the data stored within the ROM ranges from 1 through 32 bits. The largest word stored in the ROM should be within the boundaries set by the bus width of the Data Out pin, DO.
To read a word from the ROM, specify a binary number on the Address bus that matches the location of the desired word. The data is immediately output at the Data Out pin. Only addressable locations that are within the valid depth will be synthesized.

2.1.7.2. RAM Memory Modules

Random-Access Memory modules are used to store dynamic data. There are three different types of RAM modules: level-sensitive Asynchronous RAMs, Synchronous RAMs, and Dual Port RAMs.
There are two ways of using RAM modules. One way consists of storing data dynamically by addressing locations from the Address bus pin and writing binary data furnished by the Data Input port into these locations. The other way consists of initializing the RAM module's contents at power-up by specifying the Mem File attribute. You can access the latter information the same way as you would in a ROM module. Furthermore, you can overwrite the information by writing new data in the RAM locations.

2.1.7.3. Storing Data Dynamically

To store a new word into the RAM and read it out on the output port, the address of the location to be written is placed on the address bus pin (A), the data to be written is placed on the data bus pin (DI), and Write Enable is driven from low to high.
For Asynchronous RAM modules, when Write Enable goes High, the data on the Data Input port is immediately stored into the currently addressed location and read out at the Data Out port. For Synchronous RAM and Dual Port RAM modules, the Write Enable Clock must also go High. Once a word is stored in the RAM, it is possible to re-access it from the Address pin independently of the Write Enable signal.

If Write Enable stays Low while a value appears at the Data In port, this data is ignored. Only values that occur on a rising Write Enable pulse can be written. The data on the Address bus and on the Data Input port cannot change for the duration of the Write Enable pulse.

2.1.7.4. Synchronous RAM Modules (SYNC_RAM)

This module is identical to the Level-Sensitive RAM except that writes to the RAM are synchronized to the Write Enable Clock. When the Write Enable input is high and the Write Enable Clock goes from low to high, the data on the Data Input pin is written to the location specified by the Address input pin. The data on the Data Input pin appears on the Data Output pin after it is written to the RAM. New data appearing in the Data Input pin must wait for the next high on the Write Enable pin and a low to high transition on the Write Enable Clock before it is written.

This module is faster than a Level-Sensitive RAM due to the synchronous nature of the write, which allows pipelining of data on the Data Input pin.

2.1.7.5. Dual Port RAM Modules (DP_RAM)

The Dual Port RAM module includes two independent pairs of Address and Data Output pins that share access to the same region of memory, allowing simultaneous read and write access to it. Writes are synchronized by the Write Enable Clock input.

2.1.8. Multiplexer Module

There are two types of multiplexers: Type 1 or Type 2. Type 1 includes one input bus and one select line. Type 2 includes at least two input buses and one select line.

- The Type 1 Multiplexer module routes one bit of an n-bit Input to the Output under the control of the Select input port, where the value of n is determined by the width of the input port. The Select input encoding can be binary or one-hot (not implemented yet)
- The Type 2 Multiplexer module routes one of two or more Input Buses to the Output Bus under the control of the Select input port. The Select input encoding can be binary or one-hot (not implemented yet).

2.1.9. Shift Register

The Shift Register module is multi-functional, with predefined asynchronous or synchronous pre-load, and dynamic synchronous parallel load.
The Shift Register can be synthesized through SHIFT4 macro library in any one of the following configurations.

- Serial-in/serial-out shift register (FIFO or LIFO)
- Serial-in/parallel-out shift register
- Parallel-in/parallel-out shift register
- Parallel-in/serial-out shift register

Also the shift style of the register can be selected. The shift styles include the following:

- **Logical:** Data is shifted either left or right. For a left shift, the value specified by LS_IN is shifted into the LSB position. For a right shift, the value specified by MS_IN is shifted into the MSB position.
- **Circular:** Data is shifted in a circular pattern. For a left shift, the MSB is shifted into the LSB position. For a right shift, the LSB is shifted into the MSB position.
- **Arithmetic:** Data is shifted left to effectively multiply it by 2 or shifted right to effectively divide it by 2. For a left shift, a value of 0 is stuffed into the LSB. For a right shift, the Sign bit is extended if the data is signed. If the data is unsigned, a value of 0 is stuffed into the MSB.

2.1.10. Simple Gates

This section describes the generic bused gate functions of FIPBLOX. Bused gate functions are defined as generalizations of the common logic primitives: AND, INVERT, NAND, NOR, OR, XNOR, and XOR. Except for the INVERT function, each of these can be implemented in three different ways, depending on the number and type of inputs.

The following functions are described for the AND gate, but the same bus expansion criteria apply to the other functions.

Attributes

- **Logic Type:** The following descriptions use the AND module for an example, but the logic applies to all of the Simple Gate modules except the INVERT module.

Type 1 (One input bus)	The Type 1 AND module logically ANDs the individual signals of an input bus of width (n) to produce a single output signal, where (n) varies from 2-64 bits.
Type 2 (One input net and one input bus)	The Type 2 AND module logically ANDs the individual signals of an input bus of width (n) with a single input to produce an output bus of width (n), where (n) varies from 2-16 bits.
Type 3 (2-8 input buses)	The Type 3 AND module logically ANDs (m) input buses of width (n), where (m) varies from 2-8 buses and (n) varies from 2-16 bits. The output is a single bus of width (n).

- **Gate Type:** this attribute is used to select the type of gate: AND, INVERT, NAND, NOR, OR, XNOR, or XOR.
- **Input Buses:** Specifies the number of Input buses (m) of the module. This attribute applies to Type 3 modules only. Its value must be in the range 2-8.

3. Conclusions

A VHDL code generator for FIPSOC has been presented. Some features regarding memories still are under development as well as some details for checking unsupportable components (i.e. wrong operands sizes) and some latched and registered outputs.

The author thanks to Dolores Bengochea and Noelia Di Guilmi for their assistance in programming the tool's interface.

4. References

[Bra96] Adrian Bratt and Ian Macbeth, *"Design and implementation of a field programmable analogue array"*, FPGA'96, Monterrey CA.

[Lys93] Patrick Lysaght and John Dunlop, *"Dynamic reconfiguration of FPGAs"*, European FPL'93, Oxford (UK), pp 82 to 94.

[Sta95] Anthony Stansfield and Ian Page, *"The design of a new FPGA architecture"*, European FPL'95, Oxford (UK).

[Fau97a] Julio Faura, Miguel A. Aguirre, Juan M. Moreno, Phuoc van Duong, Josep M. Insenser, "FIPSOC: A Field Programmable System On a Chip" , DCIS'97

[Fau97b] Julio Faura, Josep Maria Insenser, "Tradeoffs for the Design of Programmable Interconnections in FPGAs" , DCIS'97

[Fau97c] Julio Faura, Juan Manuel Moreno, Miguel Angel Aguirre, Phuoc van Duong, and Josep Maria Insenser, "Multicontext Dynamic Reconfiguration and Real Time Probing on a Novel Mixed Signal Device with On-Chip Microprocessor" , FPL'97

[Mor97] J. M. Moreno, J. Cabestany, J. Faura, C. Horton, P. Van Duong, M. A. Aguirre, J. M. Insenser, "FIPSOC. A Novel Mixed Programmable Device for System Prototyping, MIXDES'97 (re-edited in a post-proceeding book by Kluwer Academic Publishers)

[Mad97] Julio Faura, Juan Manuel Moreno, Jordi Madrenas, Josep Maria Insenser, "VHDL Modeling of Fast Dynamic Reconfiguration on Novel Multicontext RAM-based Field Programmable Devices", VHDL User's Forum in Europe 1997

[Hor97] Julio Faura, Chris Horton, Phuoc van Duong, Jordi Madrenas, Miguel Angel Aguirre, and Josep Maria Insenser, A Novel Mixed Signal Programmable Device with On-Chip Microprocessor", Custom Integrated Circuits Conference 1997 (CICC'97)

[Kra97] Julio Faura, Chris Horton, Bernd Krah, Joan Cabestany, Miguel Angel Aguirre, and Josep Maria Insenser, "A New Field Programmable System-on-a-chip for Mixed Signal Integration", European Design & Test Conference 1997

[Nai97] Julio Faura, Paul Naish, Paul Paddan, Bernd Krah, Phuoc van Duong, Juan Manuel Moreno, Antonio Torralba, Jose Launa, and Jose Maria Insenser, "FIPSOC: A New Concept to Mixed Signal Integration", Silicon Design Show'96

[XilSup] Xilinx LogiBlox Support, *http://support.xilinx.com/support*

[SynSup] Synopsys Support, *http://www.synopsys.com*