

Design of Digital Circuits on the Basis of Hardware Templates

Valery Sklyarov

University of Aveiro, Department of
Electronics and Telecommunications, IEETA
3810-193 Aveiro, Portugal

Iouliia Skliarova

University of Aveiro, Department of
Electronics and Telecommunications, IEETA
3810-193 Aveiro, Portugal

Abstract. *This paper presents a technique for the design of digital systems on the basis of reusable hardware templates (HT), which are circuits with modifiable functionality that might be customized to satisfy requirements of target applications, such as a highly optimized implementation of the selected problem-specific operations. It demonstrates how HTs can be modeled in software with the aid of suggested C++ classes and implemented in hardware (in FPGA in particular). It is shown that the desired functionality of HT-based digital systems might be provided through some changes to the behavior of reprogrammable finite state machines (RFSM) that are considered to be primary customizable blocks. The paper describes a parameterizable VHDL code of RFSM and demonstrates how the respective circuit can be implemented in FPGA.*

Keywords: Hardware template, Reusable architectures, FPGA, Reprogrammable FSM.

1. Introduction

The existing synthesis tools permit to construct circuits, which in general cannot be reused. Any desired modification to the circuit functionality requires repeating the synthesis from the beginning, including debugging, testing and other steps that are considered as a part of the design process. This procedure can be significantly simplified with the aid of field-programmable technology (such as that is implemented in PLD in general and FPGA in particular) but in any case it requires additional time and high-experienced human resources. This paper shows that for some practical applications the considered problem can be

essentially alleviated. It might be achieved for digital systems that can be decomposed in a reusable core (which is extendable in general case) and reprogrammable control unit(s). The latter is modeled by a reprogrammable finite state machine (RFSM). The proposed design method is based on reusable hardware templates (HT). The HT is a circuit with a predefined structure that has already been implemented in hardware (for example, in FPGA). The basic components of the circuit are RAM-blocks, and by reprogramming them we can implement a different functionality. This might be done with the aid of an additional removable component of the designed system called reconfiguration handler, which can be used in particular just for debugging purposes. It makes possible to evaluate various alternatives to the functionality of the developed system avoiding many traditional steps of digital design that are time and resource consuming.

2. Top-level model of hardware template

On the one hand any HT is an application-specific circuit but on the other hand it can be customized enabling us to implement different functionality, which might be provided in such a way that basic characteristics of the circuit (such as the execution time) are optimized. Examples of HTs and their use for practical applications were considered in [1].

A general architecture of a HT depends on the selected application-specific area. For example,

such a template might be constructed for different operations over binary and ternary vectors, for reprogrammable interfaces between digital circuits, for specific computational algorithms, etc.

Suppose we have to solve some tasks from a given area. For example, we want to construct a template for circuits that implement different types of interfaces. Note that basic operations for such type of applications are very similar but on the other hand each particular interface might be unique and this is very common for embedded systems. In order to model a template and to implement it in hardware the following technique has been employed. Initially the considered problem is studied in detail [1] and a basic architecture of a HT is proposed. This architecture has to be able to implement all the desired functionalities (all the required interfaces for our example). On the other hand it has to be built in such a way that it

can be optimized for any particular functionality that is needed for the considered device. In other words we would like to be able to personalize and to optimize this architecture for any customized circuit that belongs to the selected application-specific area. At the second step the HT has to be implemented and tested in software. This allows to validate its correctness and to analyze its effectiveness for solving various particular tasks. At the next step the template has to be implemented and verified in hardware. Finally we can use the HT for solving problems from the selected area [1].

In order to provide a more compact software model of a HT we will use an object-oriented description with the aid of C++ classes (see fig. 1). The top-level specification includes just two classes that are an abstract class *datapath_template* and a concrete class *FSM_template*. Note that datapath in fig.1 can also include a similar HT of a lower level.

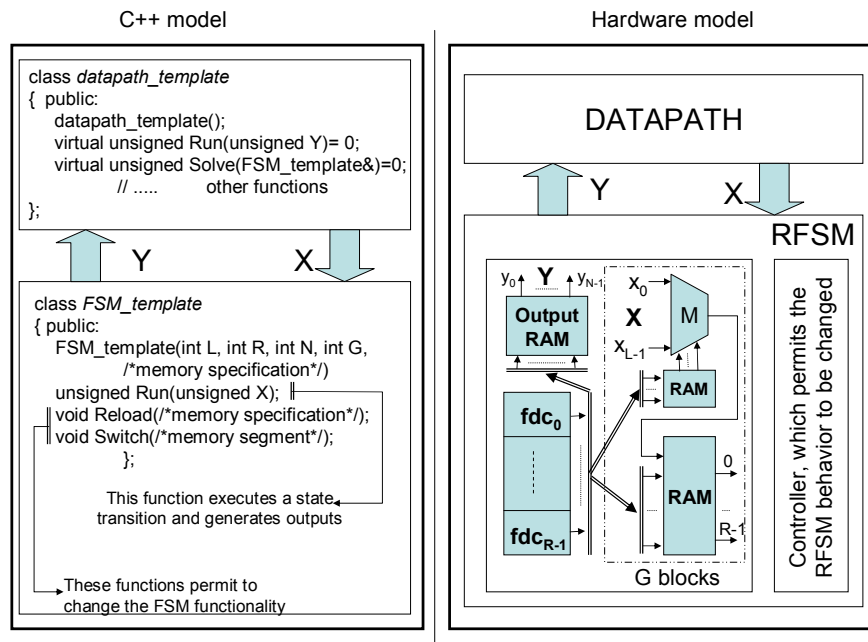


Figure 1. Relationship between software and hardware models

It is assumed that the problem can be solved with the aid of the function *Solve*, which executes an algorithm described by the argument *FSM_template&* (i.e. by a reference to an object of the class *FSM_template*). An interaction between the classes *datapath_template* and *FSM_template* has been

organized through the functions *Run* in such a way that the function *FSM_template::Run(X)* takes arguments presented in the vector *X* and returns outputs in form of vector *Y*; the function *datapath_template::Run(Y)* takes arguments presented in the vector *Y* and returns outputs in form of vector *X*. Each vector is considered to

be a binary value stored in a variable of type *unsigned int* (obviously any similar type or a user-defined type can be considered). Positions of the respective bits (from right to left) represent values of relevant binary variables. For example, the vector $X = \dots 101$ corresponds to values $x_1 = x_3 = 1$, $x_2 = 0$, ... ; the vector $Y = 0 \dots 01011$ specifies values $y_1 = y_2 = y_4 = 1$ and all the other outputs have the value 0.

The class *datapath_template* describes a general structure of hardware that can be used for solving an application-specific subset of tasks. The class constructor provides a default initialization (for instance, resets all available storage elements to 0), the functions *Solve* and *Run* are pure virtual functions and they are doing nothing because we intend to describe just a general structure for a subset of similar applications and we do not have yet any particular application. The latter can be described by a concrete class derived from the class *datapath_template* in such a way that the considered above general structure might be reutilized without any change or it can be extended. The functions *Solve* and *Run* in a new class have to be implemented and they specify the desired functionality. The primary target requirement is to construct a reusable circuit that might be employed for solving different problems from a given application-specific area. In order to solve different problems with the aid of the same datapath we have to be able to change sequences of operations that are provided by control circuits, i.e. we have to alter the behavior of the respective RFSM(s). It is important to provide such modifications not only for a software model (which is flexible by definition), but also for the relevant hardware model. Suppose that for modification purposes the class *FSM_template* contains 3 functions. The first one is a class constructor, which provides an initial (static) specification of the behavior; the second one is a function *Reload*, which allows the behavior to be changed and the last function *Switch* enables us to alter an active segment of the RFSM memory (the method [1] is used for such purposes). The last two functions offer dynamic changes to HT functionality. The next section shows how to realize such opportunities in hardware.

Note, that in general case it is impossible to describe concrete functions *Run* and *Solve* of a

class derived from the class *datapath_template* without taking into account a particularity of the concrete application. Examples of such applications with extensive list of references to the relevant publications can be found in [1,2].

3. Reprogrammable control unit

The considered reprogrammable control units have been built on the basis of an RFSM model that enables us to change the behavior (i.e. to alter the implemented control algorithm) without any modification of the respective hardware. Several types of RFSMs have been proposed [1,3] and any of them might be chosen for the considered HT. We will use a cascaded model of RFSM described in [1]. It is composed of RAM blocks, programmable (with the aid of RAM) multiplexers and a register. By modifying the contents of the RAM blocks we can implement any desired behavior within the scope of predefined constraints [1], which are the size R of the RFSM register, the number of RFSM inputs/outputs L/N and the number of reprogrammable levels G [1]. The latter are the primary building blocks of the RFSM combinational circuit.

A library of C++ classes that might be used for implementing the RFSM cascaded model in software was considered in [1]. We will use this library without any change and will show how to construct an identical RFSM in hardware. The latter will be described in VHDL and all the required parameterization will be provided through VHDL *generic* and *generate* statements.

Fig. 2 depicts basic building blocks of RFSM. They are a register, RAMs and programmable multiplexers. The latter are composed of a RAM and a multiplexer M . VHDL code in fig. 2 utilizes a library component *fdc* available for Xilinx Integrated Software Environment - ISE 5.2 [4]. The register of size R is generated from R Xilinx flip-flops of type *fdc* and R is a generic parameter. A programmable multiplexer is considered to be a parameterizable block of a lower level, which is composed of the following components:

```
component Gen_Mux is -- this is a multiplexer
generic (L : integer; R : integer);
port (
  SEL : in STD_LOGIC_VECTOR(R-1 downto 0);
  X : in STD_LOGIC_VECTOR(L-1 downto 0);
```



```

signal m_out    : std_logic;
signal composed : std_logic_vector(R downto 0);

begin
composed <= ds & m_out; -- ds is a dummy state

level_CC: Prog_Mux
    generic map(L=>8,R=>3)
    port map(clk,wem,di,ar,ds,X,m_out);

level_RAM: RAM
    generic map(N=>3,deep=>16,R=>4)
    port map (clk,we,di,composed,ar,T);

```

Finally the following VHDL code describes a top-level structure of a parameterizable RFSM:

```

entity RAM_FSM is
generic (
    L : integer := 8;
    R : integer := 3;
    N : integer := 4;
    G : integer := 2);
port (clk : in std_logic;
    we_a  : std_logic_vector(G-1 downto 0);
    -- we_a is an array of we for level RAMs
    wem_a : std_logic_vector(G-1 downto 0);
    -- wem_a is an array of we for RM RAMs
    weo   : std_logic;
    -- weo is we for the Output RAM (see fig. 2)
    di    : in std_logic_vector(N-1 downto 0);
    ar    : in std_logic_vector(R downto 0);
    rst   : in std_logic;
    X     : in std_logic_vector(L-1 downto 0);
    -- RFSM input vector xL-1,...,x0
    Y     : out std_logic_vector(N-1 downto 0));
    -- RFSM output vector yN-1,...,y0
end RAM_FSM;

architecture Behavioral of RAM_FSM is

component FDC
    -- Xilinx library component [4]
end component;

component level is
    generic (
        L : integer := 8;
        R : integer := 3;
        N : integer := 3);
    port (clk : in std_logic;
        we    : in std_logic; -- write enable for RAM
        wem   : in std_logic; -- write enable for RM
        di    : in std_logic_vector(N-1 downto 0);
        ar    : in std_logic_vector(R downto 0);
        X     : in std_logic_vector(L-1 downto 0);
        ds    : in std_logic_vector(R-1 downto 0);
        T     : out std_logic_vector(N-1 downto 0));
end component;

component RAM is
    -- the same as it was considered for RM
end component;

signal m_out : std_logic_vector(G downto 1);
type between_levels is array (G downto 0)
    of std_logic_vector (R-1 downto 0);
signal dummy_state : between_levels;

```

```

begin
    -- see the code in fig. 2
    output_RAM : RAM
        generic map(N=>4,deep=>8,R=>3)
        port map (clk,weo,di,dummy_state(0),ar,Y);

end Behavioral;

```

The considered code makes possible to construct and to implement in hardware RFSM with any required characteristics (all the details were considered in [1]) that can be provided through generic parameters $L/R/N/G$ and $deep$.

4. Reprogramming technique

Suppose the HTs for datapath and RFSM have been implemented in hardware (in FPGA in particular). They enable us to build any required circuit from a given application-specific area through reloading the considered above RAM blocks. Fig. 3 demonstrates the circuit (called reconfiguration handler - RH) that has been employed for such purposes. It is composed of a source for data (such as a memory block in fig. 3) and a controller that copies data from the source to the considered in fig. 2 RAM blocks. In other words the RH interacts with hardware described in VHDL in section 3.

In a trivial case the RH in fig. 3 begins reprogramming on external *reset* signal. In software this can be modeled either by *FSM_template* class constructor (see fig. 1) or by a special function. The controller generates memory addresses (for the source of data in fig. 3 and RFSM RAM-blocks) and copies data from the source to the RFSM RAM-blocks activated by appropriate *enable* signals that can also be read from the source. When all the blocks are programmed the controller resets the RFSM and the latter is set into a working mode. This method was implemented in a circuit designed with the aid of Xilinx ISE 5.2. All data needed for reprogramming the RFSM RAM-blocks were saved in a single Xilinx *.cgf* file [4] and used by the Xilinx Core generator to construct the memory block depicted in fig. 3.

Other circuits that provide changes to RFSM behavior from PC computer were also implemented and tested. In this case data that describe a new RFSM functionality can be loaded from PC through PCI or external ports. This allows in particular dynamic changes to the RFSM behavior.

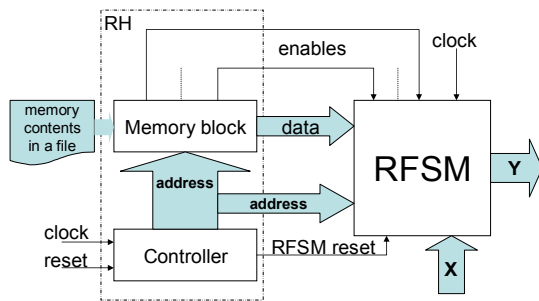


Figure 3. Reprogramming technique for RFSM RAM-blocks

It is important that all data needed for the RFSM RAM-blocks can be verified with the aid of the respective software model [1]. The latter might also be used for validation of the entire circuit including both RFSM(s) and datapath.

5. Using hardware templates for practical applications

Fig. 4 demonstrates one simplified potential application of the proposed HTs. The considered circuit consists of an RFSM and a datapath. Fig. 4,a depicts a graph-scheme of a trivial algorithm that enables us to count the number of ones in any input binary vector (see the signal *in_vector* in fig. 4). The RFSM has 2 inputs x_1, x_2 and 4 outputs y_1, \dots, y_4 , where x_1 is equal to 1 when all bits of the vector *in_vector* have been tested, x_2 is a value of the selected bit of the vector, y_1 copies the result of calculation to the circuit output (see the signal *result* in fig. 4), y_2 resets the datapath counters *count_ones* and *tmp*, y_3 and y_4 increment the counters *tmp* and *count_ones* correspondingly.

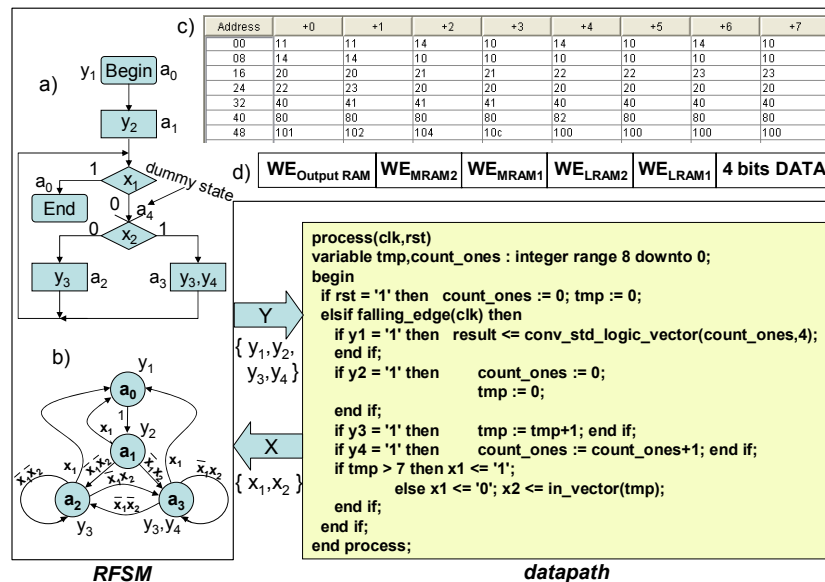


Figure 4. An example that demonstrates how the hardware templates can be used

Using method [5] the graph-scheme was converted to the RFSM state transition graph shown in fig. 4,b, where a_0, \dots, a_3 are the RFSM states, a_4 is a dummy state, which is not stored in the RFSM register. Note that any state transition between the states a_0, \dots, a_3 (see the graph in fig 4,b) is executed during just one clock cycle [1].

Fig. 4,c shows the contents of the RH memory block (see fig. 3), presented in a format of the Xilinx Core generator. The left column contains decimal addresses of memory and the

other columns show values written in memory on the respective addresses.

Fig.4,d describes a format of any 9-bit value. The first 5 bits permit to activate proper enable signals for the RAM-blocks depicted in fig. 2 in assumption that there are 2 levels in the RFSM combinational circuit. The last 4 bits keep the values that have to be copied to the RAM-blocks (see fig. 2). The right-bottom corner of fig. 4 demonstrates a trivial VHDL code for the datapath. Obviously the memory file (see fig. 4,c) can easily be changed in order to

implement other feasible operations over binary vectors, such as testing if a vector contains just one value I , or if it contains Q successive zeros, etc. Although the considered example is very simple it allows to illustrate the proposed technique. Of course, practical circuits, which might be used for such purposes require much more complicated datapath and RFSM but the latter can be generated from the same VHDL code considered in section 4.

Let us have a look at another example. A particular task might be, for instance, designing an interface between FPGA-based circuits and external devices. Very often it is necessary to support different protocols for data exchange and some of them might be unknown in advance. Fig. 5 depicts a circuit that provides programmable interfaces with a number of external devices, such as static RAMs, external controllers, etc.

In order to implement any particular interface it is not necessary to redesign the circuit. Any desired modification can be provided by reprogramming the RFSM.

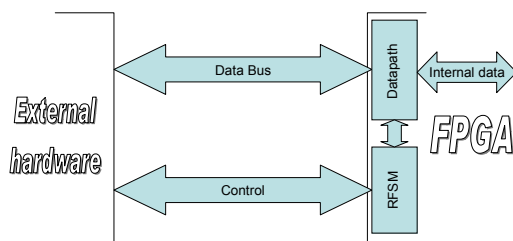


Figure 5. Example of a reusable circuit that supports different interfaces

Two HTs for applications similar to the considered above (see fig. 4 and 5) were designed with the aid of ISE 5.2 environment [4] and then implemented and tested in the FPGA XC2S300E of Xilinx Spartan-IIe family available on the prototyping board TE-XC2Se of Trenz electronic [6]. The respective circuits were validated for various operations over binary and ternary vectors and for data exchange between the FPGA and external components. The experiments with the HTs have proved all the expectable results. The designed hardware can easily be reused for implementing different operations without any

change in the relevant circuits. All the required modifications to the functionality have been provided just through reloading the HT RAM blocks. This simplifies many problems of hardware design especially the steps of testing and debugging.

6. Conclusion

The paper describes a method that can be used for the design of application-specific digital systems. It is based on circuits constructed in such a way that allows the same hardware to be employed for different types of closely related functionalities. A RAM-based reprogrammable finite state machine is considered to be a primary customizable component. The behavior of RFSM can be changed by reloading its RAM-blocks which might be done statically and dynamically. It is shown that such reprogrammable circuits called hardware templates can be utilized for a number of similar practical applications from the selected area. The paper describes how HTs can be modeled in software and implemented in FPGA. Two practical examples are examined.

Acknowledgements

This work was supported by the Portuguese Foundation of Science and Technology under grants POSI/43140/CHS/2001 and FCT-PRAXIS XXI/BD/21353/99.

References

- [1] V.Sklyarov, Reconfigurable models of finite state machines and their implementation in FPGAs. *Journal of Systems Architecture*, 2002, 47, pp. 1043-1064.
- [2] V.Sklyarov, I.Sklyarova. Architecture of a Reconfigurable Processor for Implementing Search Algorithms over Discrete Matrices. *Proceedings of ERSA'2003*, Las Vegas, 23-26 June, 2003.
- [3] V.Sklyarov, Synthesis and Implementation of RAM-based Finite State Machines in FPGAs. *Proceedings of FPL'2000*, Villach, Austria, August, 2000, pp. 718-728.
- [4] ISE 5.2; Xilinx FPGAs. [Online] available: <http://www.xilinx.com/>.
- [5] S. Baranov, "Logic Synthesis for Control Automata". Kluwer Academic Publishers, 1994.
- [6] Spartan IIE Development Platform, 2002. Available: www.trenz-electronic.de