

Synthesis and Implementation of Parallel Logic Controllers in All Programmable Systems-on-Chip

Valery Sklyarov, Iouliia Skliarova and João Silva

Abstract The chapter is dedicated to the design of logic controllers with customizable behavior in all programmable systems-on-chip in such a way that the desired functionality is defined in software of a processing system and realized in hardware of reconfigurable logic. The controllers implement algorithms described in form of parallel hierarchical graph-schemes that are built in software from predefined modules. Parallel hierarchical circuits of the controllers are mapped to the reconfigurable logic customized from software through high-performance interfaces. The circuits generate control signals to determine the desired functionality of external devices. A number of experiments are done in Xilinx Zynq-7000 microchips and the results are reported.

Keywords Hardware/software architectures · Parallel logic controllers · Hierarchical finite state machines · Hierarchical algorithms · Hardware/software interactions

1 Introduction

Nowadays, the development of software and hardware becomes more and more interrelated [1]. The emphasis has significantly shifted from general-purpose to application-specific products in the form of embedded processing modules in various areas such as communications, industrial automation, automotive computers, and home electronics. There is a tendency to integrate components on a chip that not so long ago were separated and implemented as autonomous devices. For example, the Zynq-7000 [2] all programmable system-on-chip (APSoC) incorporates a processing

V. Sklyarov (✉) · I. Skliarova · J. Silva
Department of Electronics, Telecommunications and Informatics/IEETA,
University of Aveiro, Aveiro, Portugal
e-mail: skl@ua.pt

I. Skliarova
e-mail: iouliia@ua.pt

J. Silva
e-mail: jpss@ua.pt

system (PS) that combines the industry-standard ARM dual-core CortexTM-A9 RISC processor and a number of peripherals such as memory controllers, USB, Gigabit Ethernet, and UART. The same micro-chip contains a built-in gate array (programmable logic—PL) from the Artix-7 or Kintex-7 FPGA families that is linked with the PS through on-chip interfaces.

APSoCs like Zynq [2] can run software that interacts with parallel processing elements (PE) mapped to hardware. The main objective of any PE is to provide greater performance than an equivalent software component with similar functionality that is typically composed of a set of functions in C or methods in Java. A parallel logic controller can be seen as one of application-specific PEs that gets inputs from the controlled systems and generates outputs that ensure the desired functionality. Real-time systems may require high-speed control that can be provided more easily in hardware rather than in software. Besides, control circuits are often used in such hardware components that replace software functions [3].

For many practical applications (such as knowledge-based systems in [4]) interaction between programmable logic controllers and software in a PC is widely used. We suggest in this chapter to provide better support for such interactions using APSocS that run software in the dual-core processing system and hardware in the programmable logic. The emphasis is done on the following issues:

1. Support for modularity, hierarchy and parallelism in hardware (in the PL of APSoc) based on hierarchical (HFSM) and communicating (CFSM) finite state machines [5] with such functionality that can be customized and modified from software of APSoc running in the PS.
2. Interactions between a programmable parallel logic controller implemented in the PL and software in the PS through interrupts, general-purpose and high-performance ports.
3. Dynamic reconfiguration of the controller from software of the PS based on the methods [6] and potentially applying the knowledge-based technique from [4].

The remainder of the chapter is organized in five sections. Section 2 suggests architectures of parallel logic controllers implemented in APSocS and the methods of interaction between hardware and software components. Section 3 describes the design and implementation of parallel logic controllers with dynamically modifiable functionality providing support for modularity and hierarchy. Section 4 gives more details about hardware/software interactions. Section 5 discusses the details of implementations and examples. Section 6 concludes the chapter.

2 The Proposed Software/Hardware Architecture

Figure 1 shows the proposed hardware/software architecture. A reconfigurable parallel logic controller is implemented in the PL and we will consider below the following two models for such controllers: parallel hierarchical finite state machines (PHFSMs) [3] and communicating finite state machines (CFSMs) [5].

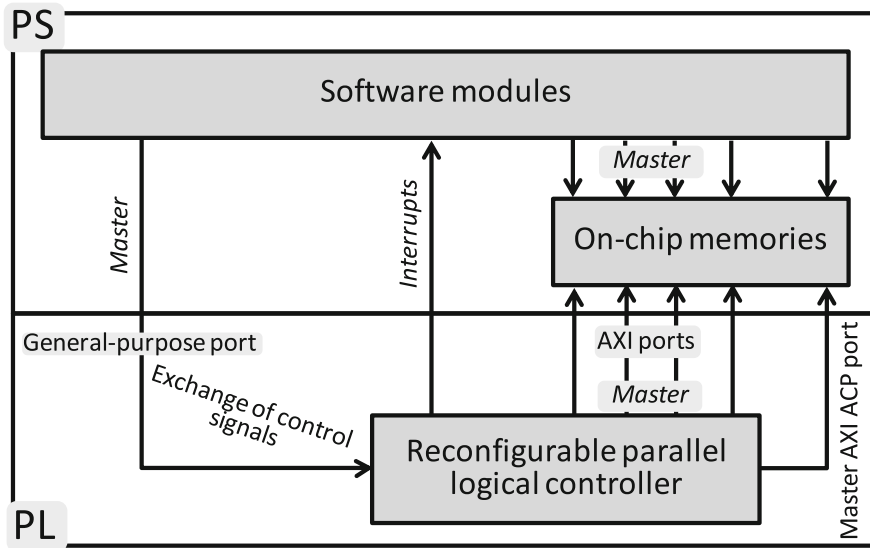


Fig. 1 Hardware/software architecture

Software modules in the PS are responsible for the following three functions:

1. Higher-level control that enables lower-level modules of PHFSM/CFSM to be managed. This means that the modules are not hard linked in the PL and can be activated/deactivated from software which much like [4] may use knowledge-based technique.
2. Run-time reconfiguration of lower-level modules allowing different functionalities to be implemented using the same hardware.
3. Test and debug of the lower-level modules.

Interaction between software and hardware modules is provided through the following interfaces:

1. General-purpose ports (GPP) [2] for exchange of control signals.
2. High-performance ports (HPP) to configure (reconfigure) modules of the parallel logic controller.
3. Interrupts generated in hardware and handled by software to support high-priority requests from hardware that need immediate reaction, which is important for real-time systems.

Figure 2 shows communication mechanisms between software and hardware with more details. The PHFSM/CFSM contains modules that can be executed in parallel. Any module is considered to be either a conventional finite state machine (FSM) or a hierarchical FSM (HFSM) and has pre-defined signals that are:

- a) An input signal *start* indicating that the module has to be reset to the initial state and begin execution.

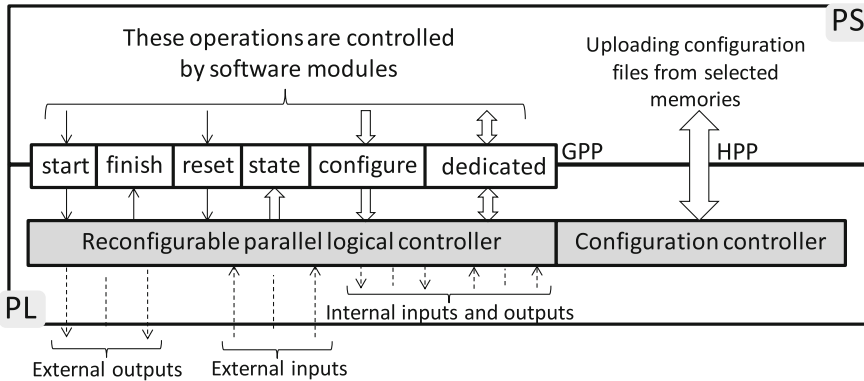


Fig. 2 Details of interactions between software and hardware modules

- b) An output signal *finish* designating that the module has completed the associated operations and is suspended.
- c) An input signal *reset* requiring transition to the initial state of the module. This signal may also reset the relevant registers in the attached execution unit (data-path).
- d) An output vector named *state* represents the current state of HFMSM/FMSM memory (state register). This vector can be used efficiently for debugging purposes. Indeed, software is capable of monitoring this signal and concluding if the desired functionality is properly provided or if there is an unusual situation. Many potential deadlocks can be found and eliminated.
- e) An input signal *configure* requests customization of the module and points to the first address in on-chip memory with the reconfiguration file. On such a request the module is reconfigured by a configuration controller and as soon as this operation is completed the signal *finish* is generated.
- f) Some signals are dedicated to particular module functionality and we will discuss them later.

Software modules set/check the GPP signals using two ways:

- a) Periodically and on internal requests generated according to the implemented algorithms. For example, as soon as one task is completely solved the hardware module responsible for the task may be reconfigured to solve the subsequent task.
- b) Immediately on interrupts from hardware modules.

Hardware modules may be configured statically or dynamically. Static configuration is done when the relevant bit-stream is uploaded to the PL section. Dynamic reconfiguration is provided during execution time, i.e. after bit-stream has been loaded. This is done with the aid of the methods described in [6] (see the next section). PHFSM/CFMSM may be used for the following three types of applications:

1. External devices connected to APSoC pins, such as those described in [7].
2. Internal blocks that may be used for different purposes, for example to accelerate time consuming segments of software modules.
3. A composition of external and internal devices, for example, some modules of the HFSM/CFSM may control components of an assembly line [7] and some other modules may be used for solving optimization problems such as planning the sequence of operations, etc.

Apart from applications described above, PHFSMs/CFSMs can be used as hardware accelerators of software programs, such as [1]. We will show below that for such applications capabilities of parallelism, modularity, hierarchy and dynamic reconfiguration are also very useful and important.

3 Design and Implementation of the Parallel Logic Controller

We have already mentioned that the parallel logic controllers considered here are based on different FSM models. Basically, we can distinguish three types of FSM models, which are *simple sequential*, *hierarchical*, and *parallel*. In turn, they can be further divided (for example, we can consider recursive and iterative hierarchical models).

Methods of synthesis for *simple sequential* FSMs are very well studied [8, 9] and they are considered just as a basis for more complicated hierarchical and parallel FSMs.

A *hierarchical* FSM is composed of other hierarchical and simple sequential FSMs (modules), which can be activated much like procedures in software programs. Thus, any module can be triggered from either another or the same module (see Fig. 3) [5].

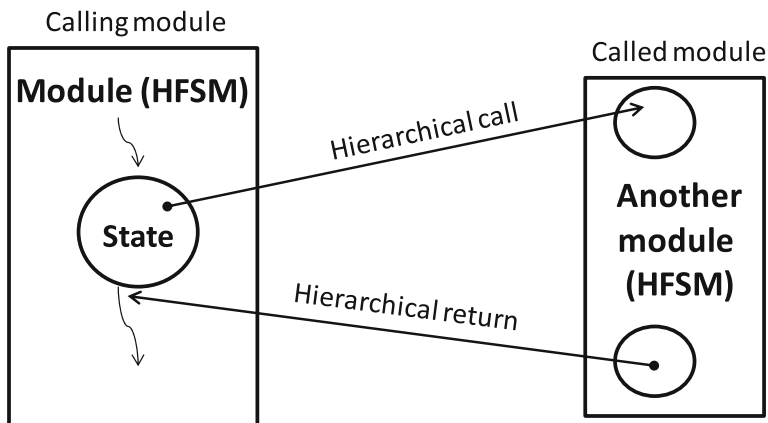


Fig. 3 Execution of hierarchical modules

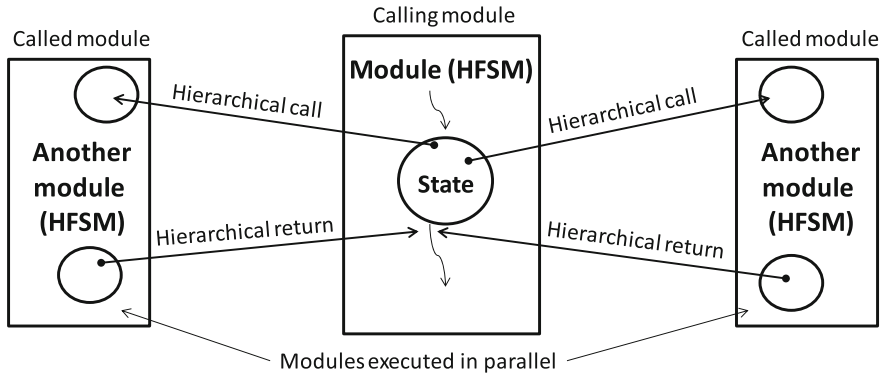


Fig. 4 Execution of parallel modules

A *parallel* FSM enables different modules to be executed in parallel (see Fig. 4). Note that generally any electronic device deals with simultaneous processing of analog/digital signals. Thus, it is parallel by definition. However, circuit level of parallelism does not give answers to many questions appearing at the algorithmic level of specification. For example, how can different branches of algorithms be executed in parallel, how can pipelining technique be applied, etc. In [5] all necessary answers to such questions are given.

The most interesting approach is a combination of parallel and hierarchical capabilities within the same FSM, which becomes a PHFSM.

Reconfiguration of HFSMs/CFSMs can be done with the aid of the methods [6] which permit HFSM/CFSM circuits to be built from reloadable memories that determine the desired functionality. The memories (that are embedded or distributed PL blocks) can be updated at execution time and thus the operations of the HFSMs/CFSMs can be changed in accordance with the requirements that might depend on some factors [3, 4].

Since HFSMs/CFSMs are composed of modules that may be replaced if required, different control algorithms specified by the modules can be selected during execution time in order to adjust parameters of the controlled devices. Thus, we can apply the strategy “try, test and replace if required”. Besides, any module can be updated with an improved version without modification of surrounding modules [3]. For example, the PS evaluates the functionality of the controlled devices and verifies if the established requirements are satisfied. If based on the result of evaluation the PS makes a conclusion that some modes or algorithms applied to the controlled devices may be improved then the set of active modules implemented in the PL can be updated and some of such modules may be reconfigured using the methods [6].

Hierarchy and parallelism can be described using various methods such as [3, 10–12]. We will use parallel hierarchical graph-schemes (PHGSs) [5]. An example of a PHGS which describes functionality of a self-controlled transport section from [13] is given in Fig. 5. The algorithm is composed of 7 modules Z_0, \dots, Z_6 . Some

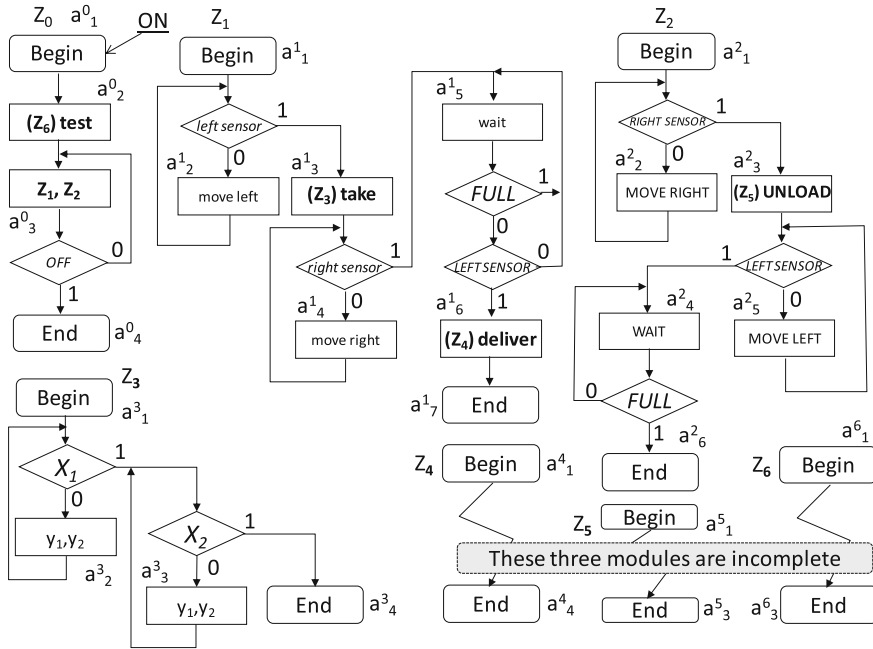


Fig. 5 An example of parallel hierarchical algorithms for a logic controller from [13]

of the modules, namely Z_1, \dots, Z_6 , are activated hierarchically and some of them, namely Z_1, Z_2 , are called in parallel. Labels like a_1^0 and a_2^0 represent states [5]. Rhomboidal nodes contain logical conditions that are formed by sensors of the logic controller and enable the sequence of execution of the algorithm to be properly selected. For example, if $OFF = 0$ in the node a_3^0 of the module Z_0 the execution of the rectangular node a_3^0 is repeated. If $OFF = 1$ in the node a_3^0 the module Z_0 is terminated. Microoperations (like y_1, y_2 , *move left*, etc.) affect actuators of the controlled device forcing the required operations to be executed.

The modules can be activated from each other in such a way that:

- the calling module is suspended;
- the called module is executed;
- as soon as the called module is terminated, the control has to be returned back to the calling module, i.e. the calling module continues its execution starting from a node following the node with the terminated called module. For example, the node a_2^0 of the calling module Z_0 activates the called module Z_6 . After Z_6 is terminated, the control has to be returned back to Z_0 and the node a_3^0 has to be activated.

If two or more modules are activated in the same rectangular node they have to be executed in parallel. For example, the modules Z_1 and Z_2 have to be activated in parallel from the module Z_0 . If two or more modules (the called modules) are called

in parallel from the calling module, the calling module is allowed to continue its execution if and only if all the called parallel modules have been completed. In other words if any of the called parallel modules is still functioning, the calling module has to be suspended. PHFSMs can formally be synthesized from PHGSs using the methods [3, 5, 13].

PHFSMs/HFSMs/FSMs may be connected in a network in such a way that they communicate with each other [5]. The communications we consider here are managed by software modules (see Fig. 2) in such a way that:

- Any FSM module can be *activated/reset/configured/tested* by software modules through GPPs and HPPs (see Fig. 2). Thus, many communication mechanisms in CFSMs [5] are provided by software.
- For such FSM states where some operations have to be immediately executed special interrupts from hardware to software are generated.
- Software modules check states of FSM modules and the interrupts from the FSM modules and make conclusion about subsequent operations.

4 Hardware/Software Interactions

Hardware/software interactions are supported by two hardware components that have been developed in the Vivado 2014.2 design suite for Zynq microchips. The first component GP_control provides support for interactions through GPPs and the second one, HP_control, enables dynamic reconfiguration to be done. Three Xilinx libraries proc_common, axi_lite_ipif, and axi_master_burst were used.

Data exchange through GPPs is provided through the PL registers mapped to an address range defined by the constant of Xilinx type SLV64_ARRAY_TYPE [14]. Interaction is organized through Xilinx modules in packages axi_lite_ipif and proc_common. From the side of hardware the constants C_ARD_ADDR_RANGE_ARRAY of Xilinx type SLV64_ARRAY_TYPE and C_ARD_NUM_CE_ARRAY of type INTEGER_ARRAY_TYPE have been properly customized selecting the required chip select and chip enable signals (many examples are given in [15]). The minimum allowed size of a memory segment is 1000_{16} (it is defined by the Xilinx constant C_S_AXI_MIN_SIZE) and it is almost always sufficient for all modules interacting with software in a way shown in Fig. 2. In rare cases when larger number of signals for GPPs is needed this constant can easily be increased (see details in [14]). Signals between the PS and the PL are transferred through registers in the PL addressed by the values in the constants and managed by the PS (the PS is the master and the PL is the slave). Hardware and software can be developed independently of each other using the defined transfer area to communicate. All projects for experiments were implemented as standalone. Other types of projects (such as running under Linux) can be prepared using the methods described in [15].

Reconfiguration of different FSM modules is done through HPPs and this requires the following customization:

1. The used memory (on-chip memory—OCM, or cache for our projects) was enabled and the size of transferred data (32 or 64 bits) was indicated.
2. The initial memory address needs to be chosen identically in software and in hardware. Software modules were developed in C language.

As soon as a request for configuration is set from software, the configuration controller in the PL copies data (allowing the chosen FSM module to be customized [6]) to the necessary memory blocks that are either embedded to the PL or distributed elements built from the PL look-up tables. It is done similarly to [16].

Reconfiguration data are kept in either OCM or cache filled in from a host PC. Copying data from the host PC to on-chip memories is done with the aid of projects from [15]. The memories are always considered to be slaves and the PS that copies data from the PC to the memories and the configuration controller in the PL that reads data and customizes the chosen FSM module are masters operating in different time slots. Configuration data are transferred from the PS to the PL in a burst mode as shown in Fig. 6.

The top module instantiates several components, two of which are *GP_control* and *Configuration module*. The remaining components are Xilinx intellectual property (IP) cores. The component *Burst reader* executes burst read (supported by the Xilinx component *axi_master_burst* [17]) and generates the signal *finished* as soon as reading is completed. After that HFSM/FSM memory blocks are loaded much like it is done in [16].

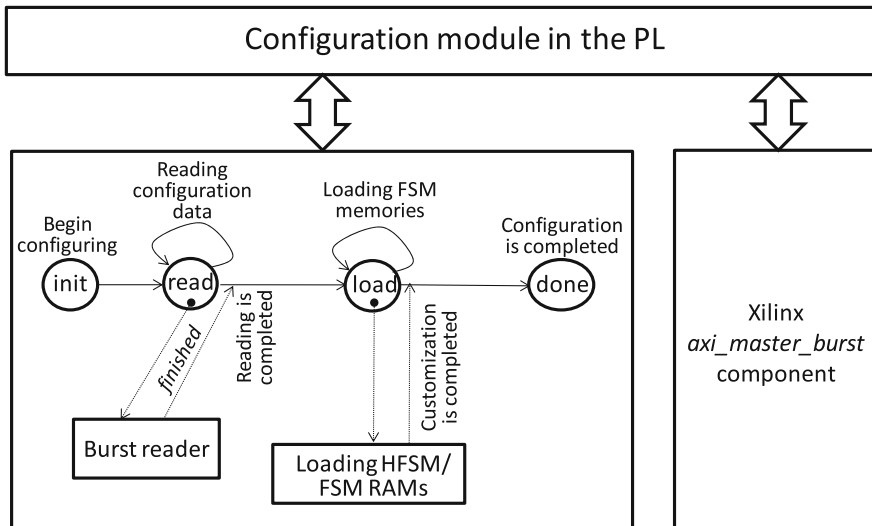


Fig. 6 Component diagram for configuration of FSM modules in burst mode

The sequence of operations *init*, *read*, *load*, and *done* is formed by a dedicated (not reconfigurable) HFSM module with the relevant states, two of which (*read* and *done*) involve hierarchical operations. The first operation is implemented in the *burst reader* and it is given in [15]. The second operation enables to load HFSM/FSM memory blocks that permit the desired customization of HFSM/FSM modules to be done.

Interrupts can be generated in any FSM module if immediate reaction is needed from the software modules. Interrupts are initiated by dedicated signals in some chosen HFSM/FSM states and processed in software by the interrupt handler. Many examples that demonstrate how interrupts can be processed in Zynq microchips are given in [15]. A similar technique is used in logic controllers that are considered here.

5 Implementations and Examples

Figure 7 shows the organization of the experiments. We used a multi-level computing system [18]. Configuration data are prepared in software of the host PC and saved in files that are copied to APSoC memories using projects from [15]. Modules of parallel logic controllers are created in the PL and managed from software of the PS. The latter and software of the host PC may also be responsible for verifying functionality of different HFSM/FSM modules. Standalone applications have been created and uploaded to the PS from Xilinx Software Development Kit (SDK) using methods described in [15]. Interaction is done through the SDK console window. All experiments were done in two Zynq-based prototyping systems: ZyBo [19] and ZedBoard [20]. Two examples are discussed in the subsequent sections.

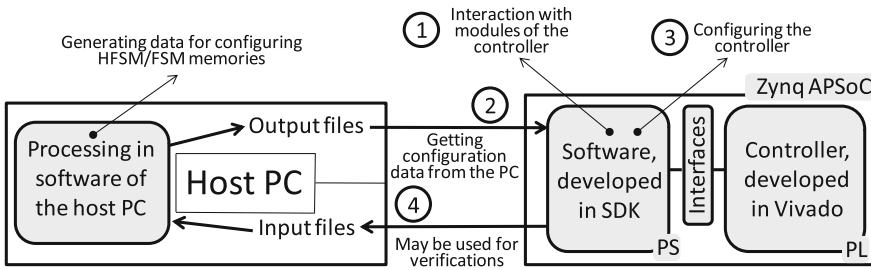


Fig. 7 Experimental setup

5.1 An Example of PHFSM-Based Hardware Accelerator

Let us consider a project demonstrating the use of PHFSM to accelerate computation of the greatest common divisor for N unsigned integers, where N is chosen to be 8. The intended functionality is demonstrated on an example of the following C function gcd with 8 arguments:

```
unsigned int gcd (unsigned int A, unsigned int B,
    unsigned int C, unsigned int D, unsigned int E,
    unsigned int F, unsigned int G, unsigned int H)
{
    return gcd(gcd(gcd(A,B) , gcd(C,D)) , gcd(gcd(E,F) , gcd
        (G,H)));
}
```

This function permits the greatest common divisor of 8 operands A, B, C, D, E, F, G, and H to be found and calls another function gcd with two operands:

```
unsigned int gcd (unsigned int A, unsigned int B)
{
    unsigned int tmp;
    while (B > 0)
    {
        if (B > A)
        {
            tmp = A;
            A = B;
            B = tmp;
        }
        else
        {
            tmp = B;
            B = A % B;
            A = tmp;
        }
    }
    return A;
}
```

Clearly, four functions gcd(A, B), gcd(C, D), gcd(E, F), gcd(G, H) can be executed in parallel at the first step giving the results Result_A_B, Result_C_D, Result_E_F, and Result_G_H. At the second step, these results will be used as arguments for the functions: gcd(Result_A_B, Result_C_D), and gcd(Result_E_F, Result_G_H), which can also be executed in parallel giving the results Result_A_B_C_D, and Result_E_F_G_H. At the next (last) step the function gcd(Result_A_B_C_D, Result_E_F_G_H) computes the final greatest common divisor of 8 unsigned integers A, B, C, D, E, F, G, and H. All the above functions will be implemented in the PHFSM

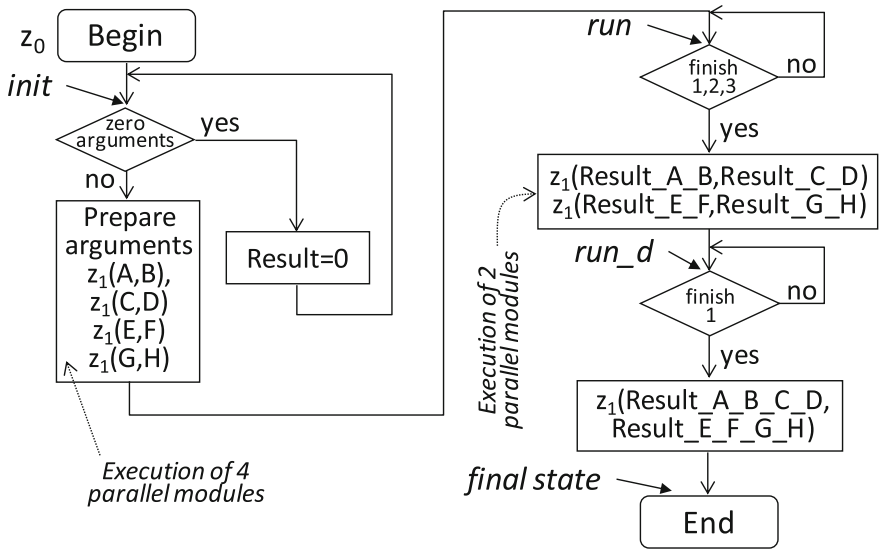
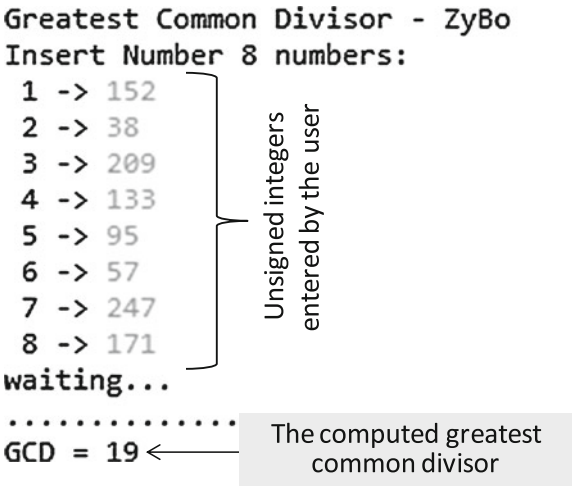


Fig. 8 Parallel hierarchical graph-scheme that permits the greatest common divisor of $N = 8$ non-negative integers to be found

Fig. 9 Interaction with the circuit that computes the greatest common divisor of eight unsigned integers



described by PHGS in Fig. 8. Possible results of interaction from the SDK console are demonstrated in Fig. 9.

At the beginning, the operands A, B, C, D, E, F, G, and H are examined and if there is at least one zero operand then the subsequent steps are not executed and the result is assigned to 0. If all the operands are not equal to zero then 4 modules Z_1 with different arguments are activated at the same time. As soon as all of them terminate, the results of these modules are used as operands for two new invocations of Z_1 also running in parallel. The final result is produced in the single module Z_1 . In [3] there are two complete synthesizable VHDL specifications that describe the hardware circuit that implements the algorithm in Fig. 8. The first specification (entity `Parallel_HFSM_iterative`) corresponds to the C function discussed above. The second specification (entity `Parallel_HFSM_recursive`) is based on a recursive C function given in [3]. Thus, there might be recursive calls in all modules Z_1 running in parallel. The modules `Parallel_HFSM_iterative` and `Parallel_HFSM_recursive` are given in [3] (see Sect. 5.4 in [3]) and can also be downloaded from <http://sweet.ua.pt/skl/Springer2014.html>.

Our example uses four address ranges [15] and respectively four chip select signals with one chip enable signal for each address pair. Let us look at the following constants:

```
constant C_CARD_ADDR_RANGE_ARRAY: SLV64_ARRAY_TYPE := (
    X"0000_0000_0000_0000", -- this pair is used for 8
        -- 32-bit operands: A, B, C, D, E, F, G, H
    X"0000_0000_0000_001F",
    X"0000_0000_0000_0020", -- this pair is used for the
        -- 32-bit result, i.e. for the greatest common
    X"0000_0000_0000_0023", -- divisor of the operands A,
        -- B, C, D, E, F, G, H
    X"0000_0000_0000_0024", -- 32-bit status (for
        -- overflow and ready signals)
    X"0000_0000_0000_0027",
    X"0000_0000_0000_0028", -- 32-bit control (for enable
        -- and reset signals)
    X"0000_0000_0000_002B");
constant C_CARD_NUM_CE_ARRAY : INTEGER_ARRAY_TYPE := (
    0 => 1,
    1 => 1,
    2 => 1,
    3 => 1);
```

The complete project that includes hardware and software modules is available at <http://sweet.ua.pt/skl/TUT2014.html>. Additional details may also be found in [15]. Verification of the project demonstrates high performance. Similar experiments have been done with recursive and iterative algorithms that enable traversing binary trees from [5] to be implemented partially in software and partially in hardware.

5.2 An Example of a Parallel Hierarchical Controller

The second example explains how to execute different operations with PHFSMs/CFSMs that implement the algorithm depicted in Fig. 5. Parallel module executions are organized with the aid of the methods [3]. The main difference between HFSMs and CFSMs is in connections between the modules that are FSMs without hierarchical calls. In HFSM all links are organized through common stack memories [3] and in CFSM they are organized through semaphores [5]. The following steps have been done:

1. Incomplete in Fig. 5 PHGSs Z_4 , Z_5 , and Z_6 have been entirely described.
2. Nodes of the PHGSs have been marked with labels: a_1^0, a_2^0, \dots in accordance with the rules [5] (see also Fig. 5).
3. A combinational circuit for each PHFSM module Z_0, \dots, Z_6 is built from memory blocks and has the structure shown in Figs. 8 and 10 of [6]. The configuration controller for memory blocks is built in a way [16].
4. The PHFSM has been synthesized and implemented in the PL with the aid of the methods [3, 6].
5. Initial configuration corresponding to the extended PHGS from Fig. 5 is done statically in the PL. Connections to the controlled devices are provided through external APSoC pins.
6. Reconfiguration that permits functionality of some modules of the PHFSM to be changed is done from software running in the PS and verified according to the methods described in Sect. 2.

We have found that reconfiguration can be done very fast. Thus, for many practical cases customization of modules may be done even during execution time. The circuit occupies less than 1 % of the PL resources, which permits many additional hardware components to be built in the same microchip.

6 Conclusion

The chapter suggests the design method for parallel logic controllers in Zynq-7000 all programmable systems-on-chip. It is proposed to model the controller by a parallel hierarchical finite state machine implemented in hardware (in the programmable logic) with additional support from software (in the processing system). The machine is composed of modules communicating with each other and managed by software, which also allows verifications and changes in the functionality of the modules applying the technique of dynamic reconfiguration. Finally, the proposed controllers provide support for modularity, hierarchy (including recursion), parallelism and run-time reconfiguration.

Acknowledgments This work was supported by National Funds through FCT—Foundation for Science and Technology, in the context of the project PEst-OE/EEI/UI0127/2014.

References

1. Sklyarov, V., & Skliarova, I. (2013). Hardware implementations of software programs based on HFSM models. *Computers and Electrical Engineering*, 39(7), 2145–2160.
2. *Zynq-7000 All Programmable SoC Technical Reference Manual* (2014). http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf
3. Sklyarov, V., Skliarova, I., Barkalov, A., & Titarenko, L. (2014). *Synthesis and Optimization of FPGA-based Systems*. Heidelberg: Springer.
4. Zmaranda, D., Silaghi, H., Gabor, G., & Vancea, C. (2013). Issues on applying knowledge-based techniques in real-time control systems. *International Journal of Computers, Communications and Control*, 8(1), 166–175.
5. Sklyarov, V., Skliarova, I., & Sudnitson, A. (2012). *Design of FPGA-based Circuits using Hierarchical Finite State Machines*. Tallinn: TUT Press.
6. Sklyarov, V. (2002). Reconfigurable models of finite state machines and their implementation in FPGAs. *Journal of Systems Architecture*, 47(14–15), 1043–1064.
7. Sklyarov, V. (2002). Hardware/software modeling of FPGA-based systems. *Parallel Algorithms Application*, 17(1), 19–39.
8. Baranov, S. (1994). *Logic Synthesis for Control Automata*. Boston: Kluwer Academic Publishers.
9. De Micheli, G. (1994). *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, Inc.
10. Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8, 231–274.
11. Uchitel, S., Kramer, J., & Magee, J. (2003). Synthesis of behavioral models from scenarios. *IEEE Transactions on Software Engineering*, 29(2), 99–115.
12. Zakrevskij, A. (1981): *Logical Synthesis of Cascade Networks*. Science, Moscow (in Russian).
13. Sklyarov, V., & Skliarova, I. (2008). Design and implementation of parallel hierarchical finite state machines. In *Proceedings of 2nd International Conference on Communications and Electronics* (pp. 33–38). Hoi An, Vietnam.
14. *LogiCORE IP AXI4-Lite IPIF v2.0. Product Guide for Vivado Design Suite* (2013). http://www.xilinx.com/support/documentation/ip_documentation/axi_lite_ipif/v2_0/pg155-axi-lite-ipif.pdf
15. Sklyarov, V., Skliarova, I., Silva, J., Rjabov, A., Sudnitson, A., & Cardoso, C. (2014). *Hardware/Software Co-design for Programmable Systems-on-Chip*. Tallinn: TUT Press.
16. Sklyarov, V., & Skliarova, I. (2007). Synthesis of reconfigurable hierarchical finite state machines. *Studies in Computational Intelligence, Autonomous Robots and Agents* (pp. 259–265). Berlin: Springer.
17. *LogiCORE IP AXI Master Burst v2.0. Product Guide for Vivado Design Suite* (2013). http://japan.xilinx.com/support/documentation/ip_documentation/axi_master_burst/v2_0/pg162-axi-master-burst.pdf
18. Sklyarov, V., Skliarova, I., Silva, J., & Sudnitson, A. (2014). Design space exploration in multi-level computing systems. In *Proceedings 15th International Conference on Computer Systems and Technologies* (pp. 40–47). Bulgaria.
19. *ZyBo Reference Manual* (2014). http://digilentinc.com/Data/Products/ZYBO/ZYBO_RM_B_V6.pdf
20. *ZedBoard (Zynq™ Evaluation and Development) Hardware User's Guide* (2014) Version 2.2. http://www.zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf

Design of Reconfigurable Logic Controllers

Karatkevich, A.; Bukowiec, A.; Doligalski, M.; Tkacz, J.
(Eds.)

2016, VIII, 185 p. 76 illus., 13 illus. in color., Hardcover

ISBN: 978-3-319-26723-4