Dynamically Reconfigurable Implementation of Control Circuits

Nuno Lau, Valery Sklyarov

Departamento de Electrónica e Telecomunicações da Universidade de Aveiro

Key words: FPGA, Reconfigurable Computing, Partial and Dynamic Configuration

Abstract: Control Circuits can be described using a top-down approach with the aid of Hierarchical Graph-Schemes (HGSs). The implementation of HGSs in a fine-grain FPGA has been done using a Hierarchical Finite State Machine structure where each sub-algorithm implementation is independent from the others. Static and dynamically reconfigurable implementations using the XC6200 FPGA have been obtained by applying a number of commercial design tools together with tools developed by the authors.

1. INTRODUCTION

FPGAs are being widely used in the design of complex digital systems involving both single and multiple FPGAs [Hauck 98], but the properties of plain, dynamic, or partial reconfigurability are not explored in most of these applications. The absence of CAD tools that support the design flow of reconfigurable circuits and the lack of published application expertise are certainly some of the main difficulties that a designer of reconfigurable circuits must handle [Hutchings95]. On the other hand several research groups have developed applications that prove the possibility (and in some cases the economic efficiency) of dynamically reconfigurable applications [Eldredge96, Wirthlin95, Robinson98, Shirazi98, Sklyarov98].

We think that one of the application areas where dynamic reconfiguration can prove its usefulness is in the implementation of control circuits. The specification of control circuits can be made using a top-down approach where the details of behaviour are at the lowest level and the top level gives an overview of the whole system. A top-down approach leads to a modular

---

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: [10.1007/978-0-387-35498-9_57](https://doi.org/10.1007/978-0-387-35498-9_57)

L. M. Silva et al. (eds.), *VLSI: Systems on a Chip*

© IFIP International Federation for Information Processing 2000

specification of the circuit that is also well adapted to a modular implementation. In our view, circuits that are modular and well-structured are good candidates for implementation using dynamic reconfiguration. A reconfiguration model has been developed that allows the management of sets of FPGA resources as if they were pages in a virtual memory system. A dynamically reconfigurable implementation of an HFSM has been used as a proof-of-concept example for the reconfiguration model. Section 2 presents the specification of control circuits using a top-down methodology known as hierarchical graph-scheme (HGS) [Rocha97]. Section 3 discusses the implementation of HGSs using a Hierarchical Finite State Machine model, the synthesis method adopted, and its static and dynamic implementation using a XC6200 FPGA.

2. HIERARCHICAL GRAPH-SCHEMES

The standard method to describe a control circuit is using a state transition table or a state transition diagram [Baranov94]. While these specification methods are very useful, the specification of control circuits can be done at a behavioural level using HGSs. An HGS specification of a control circuit gives us a visual description of the control algorithm. Any HGS must have one BEGIN node and one END node and it may have rectangular and rhomboidal nodes that link the BEGIN and END node forming a directed connected graph (*Figure 1*) [Sklyarov97]. Rectangular nodes determine which micro-operations (denoted y_i) and macro-operations (denoted z_j) have to be activated at each step. Rhomboidal nodes are used to select from two different execution flows of the HGS. The value of a conditional signal (denoted x_l) that is inserted in the rhomboidal node determines which path is executed. HGSs are powerful specification tools as they allow the specification of the control algorithm to be seen at several layers of abstraction. This is achieved by the use of macro-operations, i.e. one HGS can invoke another HGS in a way similar to procedure calls, and execution of the HGS that makes the call will only proceed when the HGS that was invoked has reached its END node.

3. HGS IMPLEMENTATION

Synthesis of a control circuit that is described by a set of HGSs is the process of transforming the HGS specification of the control algorithm to a hardware implementation. In our case we are interested in the implementation of the HGS in the Xilinx XC6200 FPGA family [Xilinx97].

Figure 1. Example of HGS specification

This FPGA has a fine-grain sea-of-gates architecture and is dynamically and partially reconfigurable.

3.1 HFSM implementation model

The HGS specification is to be implemented in hardware using the Hierarchical Finite State Machine (HFSM) model shown in Figure 2. This model is an extension of the FSM model, where the state register has been replaced by a combination of a stack memory and a normal register. This modification has been introduced to accommodate hierarchical calls between HGSs. When an HGS call is executed, the current state is stored on the stack and the stack pointer is incremented. The state of the HGS that was called is recorded and managed at the new stack level (Figure 3). This mechanism can even be used for recursive HGS calls. The standard FSM model has also been extended with a new component called the code converter, which is responsible for the selection of the invoked HGS. This is implemented as a RAM based device. Rewriting the contents of this RAM allows an easy (and fast) change in the behaviour of the HFSM. This is an important feature as it embodies the concept of virtual HGSs, i.e., dynamic binding of sub-algorithms. The management of the HFSM model demands a multi-phase clock synchronisation scheme.

By examining the variability of the components of the HFSM model of the behaviour of the control circuit, one can separate the components into two groups: one group containing components that will be parameterised and the other containing components that will be synthesised.

Figure 2. HFSM implementation model

The parameterised components are those that are almost independent of the behaviour of the HFSM. This group includes the stack memory, the code converter, and the synchronisation scheme. Optimised versions of structural VHDL descriptions of these components have been developed and are included in a library.

Figure 3. Execution of HGS

The synthesised group includes those components that are very dependent on the actual HFSM behaviour. These include the combinational scheme and the non-stack part of the state register. Synthesis tools have been developed to generate structural VHDL descriptions in accordance with the HGS behaviour. These tools use a textual description of the HGS (Figure 4). The syntax of the HGS text file was developed having in mind that it should be understandable for a human reader, its parser should be simple and it should be open to the inclusion of further HGS characteristics.

GRAPH 0 Z0									
-- node num type next									
NODE	0	0	Begin	2					
NODE	1	1	End						
NODE	2	3	Rect1	3	MICRO	1	4	MACRO	
NODE	3	4	Rhomb	1	1	6	4		
NODE	4	3	Rect2	5	MICRO	MACRO 1			
NODE	5	3	Rect3	1	MICRO	2	3	MACRO	
NODE	6	4	Rhomb	2	2	5	7		
NODE	7	3	Rect4	1	MICRO	3	4	MACRO	

Figure 4. Textual description of Figure 1's Γ_0 .

The synthesis of the combinational scheme is done considering each macro-operation as a different module, and generating a structural VHDL entity for each of them. Then the whole combinational scheme is an aggregation of the mentioned above entities [Oliveira98].

3.2 HGS synthesis

The synthesis of each macro-operation, representing a sub-algorithm, is done using a direct mapping of the macro-operation graph description onto its hardware implementation. To achieve a simple direct association between an HGS and its circuit implementation we have used one-hot state assignment This provides an implementation of a sub-algorithm that is independent of the behaviour of the others, and also allows for the possibility of internal states, as we will explain later. In the mapping for each of the rectangular nodes of an HGS there is a memory element and for each of the rhomboidal nodes there is a demultiplexer. *Figure 5* shows a simple example for which the HGS does not have macro-operations.

If one-hot state encoding is used, the mapping of the rectangular nodes of an HGS is dependent on their contents:

- Those that include macro-operations must be mapped onto stack memory. The corresponding states will be called hierarchical states.
- Those that include just micro-operations are mapped onto a D-flip-flop. The flip-flops can be instantiated as internal components of the VHDL entity that implements the macro-operation. This makes the routing of the system easier and provides a simpler interface for the entity.

The number of states that are mapped in RAM should be minimised in this implementation. If one hot state assignment is used inside the stack then the number of states that it can map is very limited, i.e., a stack of 8 bit registers would be able to map 8 states. One can minimise the problem by providing a coder and a decoder for stack contents, which enables the same stack of 8 bit registers to map 256 one-hot states, but this inevitably has significant costs in terms of area and performance of the whole system.

Figure 5. Mapping HGS onto the respective circuit

3.3 Static model

In this model of HGS implementation, all components of the HGS circuit are permanently mapped onto the FPGA. The combinational scheme is an aggregation of the various macro-operations optimised for each of them, i.e., if a certain sub-algorithm only uses conditional signals 1 and 2 from the four available then only these two signals are provided for that sub-algorithm. The synthesis tools generate this optimised aggregation.

The final step in the implementation of this type of circuit is the elaboration of the structural VHDL description of the circuits, which is performed by Velab and generates an EDIF file. Then the mapping, placement and routing of the circuit are executed by XACT6000, which generates the final configuration of the XC6200 FPGA.

3.4 Dynamically reconfigurable model

To explain the dynamically reconfigurable implementation of the HGS we should start by stating the general principles used in the development of our reconfiguration model.

We have defined some of the FPGA resources (functional cells and routing muxes) as “dynamically reconfigurable” (called “reconfigurable” in the rest of the paper), meaning that they are available for reconfiguration during run-time, while other (“fixed”) resources must be configured only at startup. These “reconfigurable” resources have been divided into several sets where the objective is to ensure that modifying a “reconfigurable” resource of a specific set is guaranteed not to change the circuit beyond the limits of the set where it belongs. In our case these sets are non-overlapping

rectangular areas inside the FPGA and in each area (set) all functional units and local routing resources have been considered “reconfigurable”.

Each of the “reconfigurable” sets has the same shape and the same topology with respect to the routing resources of the FPGA, i.e., if they all use length 4 routing resources then their position relative to a 4x4 boundary has to be the same for all “reconfigurable” sets.

Each of the “reconfigurable” sets has a fixed interface, both structurally, where they all have the same signals as inputs and outputs, and spatially, where the relative position of inputs and outputs inside the set is the same for all sets.

Each partial configuration describes a circuit that uses “reconfigurable” resources of only one of the sets. There should be more partial configurations than “reconfigurable” sets, otherwise the static model implementation is preferable.

If several partial configurations use resources that belong to just one “reconfigurable” set, they are available for swapping.

These assumptions are very important because they provide each reconfigurable set with characteristics similar to those of pages in virtual memory systems:

- The configuration of a particular set is independent of the configuration of the others and does not affect the others.
- Configurations can easily be moved from one set to another.
- The maximum size of a configuration can be easily calculated. In fact we could consider configurations as being of a fixed size, but as we will see, not all of the resources of each “reconfigurable” set need to be programmed to change the configuration of that set.
- All the information about each configuration can be saved in an external device.

Another principle of our dynamically reconfigurable model is that the circuit is responsible for signalling some external device in the event that reconfiguration is needed. A reconfiguration handler implemented in the external device is responsible for loading the new partial configuration to the desired set. Additional information may be needed by the external reconfiguration handler in order to choose which configuration has to be mapped onto which of the “reconfiguration” sets.

Let's turn our attention to the dynamically reconfigurable implementation of HGSs and to the methodology and tools that have been developed to support it. We'll also discuss some of the limitations of Xilinx tools when they are applied to dynamic reconfigurable circuits and how we have overcome these limitations.

As has already been said, our “reconfigurable” sets of resources have been defined as non-overlapping rectangular areas inside the FPGA. The interface of each “reconfigurable” area can be seen in *Figure 6*. Every input is fed to every reconfiguration area and every output is also received from all areas because neither the sub-algorithms that are going to be mapped nor their input/output needs are known,. Each of these areas can accommodate one sub-algorithm of the HGS if it is needed at a certain execution time (*Figure 7*).

Figure 6. Template for the external interface of an HGS component

The need for reconfiguration has been easily incorporated into the HFSM model by reserving one bit of the code converter entries to specify if a certain macro-operation is configured inside the FPGA or not. If it is configured, the other code converter bits specify the mapping area, and if it is not mapped, these bits specify the macro-operation code (*Figure 8*).

When reconfiguration is required the reconfiguration handler reads the code converter register (inside the FPGA) in order to know which macro-operation has to be loaded. In this model, reconfiguration only happens

Figure 7. HGS reconfiguration model

during macro-operation invocation and not during macro-operation return, as it is a constraint of this model that all macro-operations in the path from the main graph to the present macro-operation cannot be swapped out. The reconfiguration handler maps the macro-operation onto one of the available areas, determined by reading stack contents, then changes the contents of the code converter so that it reflects the actual configuration of these areas and indicates that execution may proceed.

Figure 8. Code Converter contents before (a) and after (b) reconfiguration

Most of the components belonging to the parameterised group didn’t have to undergo any modification to allow reconfigurability. Only the synchronisation scheme had to be changed. This is the circuit that signals the need for reconfiguration and that manages the protocol with the reconfiguration handler to resume HGS execution when reconfiguration is finished.

The synthesis tools had to be adapted to this new model as the interface of all macro-operations implementations is now the same, whether that particular implementation is using all interface signals or not (inputs that are not used are left open while unused outputs are connected to ground). The combinational scheme has a fixed structure that does not depend on the HGS specification and depends only on the specified interface and on the number of “reconfigurable” areas. The contents of each of these areas are only defined during run-time by loading the relevant partial configuration.

To respect the principles stated above, the “fixed” configuration should not use any of the “reconfigurable” resources. If they are used then partial configurations might affect the circuit implemented by the “fixed” configuration. XACT6000 provides a designer attribute to exclude all functional units within a certain area from being used, but it doesn’t provide anything similar with respect to routing resources. In order to generate a safe “fixed” configuration, we have made XACT think that all “reconfigurable” resources were already being used (so that it wouldn’t use them for other purposes) by generating a dummy layout (in the form of a XACT layout file [Xilinx97b]) that occupies all these resources (Figure 9).

In our case we have a different partial configuration for each area and for each macro-operation, so if we have two reconfiguration areas and 4 macro-

operations we need 8 partial reconfigurations. Only 4 partial configurations would be required if we had a translation operator for partial configurations available, so this operator is planned for the near future.

Figure 9. Dummy layout for reconfiguration area

The generation of usable partial configurations is not directly supported by XACT6000. We have created them by using XACT to generate a global configuration for the entities of each macro-operation located in each area, and then filtering these global configurations to obtain partial configurations where only “reconfigurable” resources are modified. The filtering process is based on a textual description of the “reconfigurable” resources of each area (Figure 10). The methodology to obtain the initial and partial reconfigurations is shown in Figure 11.

```
Cells (16,7-17)
  NORTH Reconfigurable
  SOUTH Reconfigurable
  EAST Reconfigurable
  WEST Fixed
  X1 Fixed
  RP Fixed
  ...
  End
```

Figure 10. Defining reconfigurable resources

The reconfigurable control circuit and design procedures have been tested in hardware using the Annapolis FireFly^{TM} PC board. Two versions of the reconfiguration handler have been used, one implemented in software using C++ and the other implemented in hardware using an XC4010XL FPGA mounted on a XS40 board [Xess98] that interfaces to the PC through the parallel port.

Figure 11. Creating initial and partial reconfigurations

4. CONCLUSION

Complex control circuits can be specified using HGSs by detailing the properties of the system at several layers of abstraction. These complex control circuits can be implemented in fine-grained FPGAs using the HFMSM structure proposed. This structure provides a high degree of modularity for the implementation of the sub-algorithms (macro-operations) of the HGS, and so it is well adapted to a dynamically reconfigurable implementation.

General principles for a model to use reconfigurability have been presented and these principles have been applied with success to the dynamically reconfigurable implementation of HGSs using the XC6200 FPGA. Several tools had to be developed to overcome the lack of proper support for dynamic reconfigurability in standard tools.

5. ACKNOWLEDGEMENTS

We are thankful to Ricardo Monteiro, António Rui Borges, Arnaldo Oliveira, Andreia Melo and António Ferrari for fruitful discussions concerning this work. We thank Ivor Horton for reading and useful comments on this paper.

6. REFERENCES

- [Baranov94] Samary Baranov, *Logic Synthesis for Control Automata*, Kluwer Academic Publishers (1994)
- [Eldredge96] James Eldredge, Brad Hutchings, "Run-time Reconfiguration: A Method for Enhancing the Functional Density of SRAM-Based FPGAs", in *Journal of VLSI Signal Processing*, Volume 12, pp 67-86 (1996)

- [Hauck98] Scott Hauck, "The Roles of FPGA's in Reprogrammable Systems", *Proceedings of the IEEE*, vol. 86, no. 4, pp. 615-638 April, 1998.
- [Hutchings95] B.L.Hutchings, M.J. Wirthlin, "Implementation Approaches for Reconfigurable Logic Applications", In *5th International Workshop on Field Programmable Logic and Applications*, pp 419-428, Oxford, England, August (1995)
- [Oliveira98] A. Oliveira, N. Lau, V. Sklyarov, "Synthesis of VHDL Code from Hierarchical Specification of Control Circuits for Dynamically Reconfigurable FPGAs", *CD Proceedings of VHDL International User Forum - VIUF*, Orlando (1998)
- [Robinson98] D. Robinson, P. Lysaght and G. McGregor, "New CAD Framework Extends Simulation of Dynamically Reconfigurable Logic", in *Field Programmable Logic and Applications*, pp 1-8, Tallinn, Estonia, Sept. (1998)
- [Rocha97] António Adrego da Rocha, Valery Sklyarov, António de Brito Ferrari, "Hierarchical Description and Design of Control Circuits Based on Reconfigurable and Reprogrammable Elements", *Proceeding of the Int. Workshop on Logic and Architectural Synthesis-IWLAS'97*, Grenoble, December (1997)
- [Shirazi98] N. Shirazi, W. Luk and P.Y.K. Cheung, "Automating production of run-time reconfigurable designs", in *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines*, IEEE Computer Society Press, (1998)
- [Sklyarov97] Valery Sklyarov, António Adrego Rocha, António Ferrari, "Synthesis of Reconfigurable Control Devices Based on Object-Oriented Specifications", In *Advanced Techniques for Embedded Systems Design and Test*, Kluwer Academic Publishers, pp. 151-177 (1997)
- [Sklyarov98] V. Sklyarov, N. Lau, A. Oliveira, A. Melo, K. Kondratjuk, A. Ferrari, R. Monteiro, I. Skliarova, "Design Tools for Dynamically Reconfigurable FPGAs", *Proceedings of PACT'98 - Workshop on Reconfigurable Computing*, Paris, Oct. (1998)
- [Xess98] "XS40, XSP and XS95 Board Manual", <http://www.xess.com/FPGA/manual.html>, XESS corporation (1998)
- [Xilinx97] "XC6200 Field Programmable Gate Arrays", Xilinx Product Description (Version 1.10). April 24 (1997)
- [Xilinx97b] "DES and CFG Format Reference", Xilinx (1997)
- [Wirthlin95] Michael J. Wirthlin, Brad Hutchings, "A Dynamic Instruction Set Computer", In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pp 99-107, Los Alamitos, California, April (1995)