Universidade de Aveiro
Departamento de Electrónica e Telecomunicações

Synthesis and Simulation of Reprogrammable Control Units from Hierarchical Specifications

António Manuel Adrego da Rocha
1999

Synthesis and Simulation of Reprogrammable Control Units from Hierarchical Specifications

António Manuel Adrego da Rocha

Departamento de Electrónica e Telecomunicações
Universidade de Aveiro
Portugal, May, 1999

Thesis submitted in fulfilment of the requirements for the degree of
Doutor em Engenharia Electrotécnica

Acknowledgements

I would like to thank to my thesis supervisors. To Valery Sklyarov for introducing me in the field of hierarchical finite state machines with stack memory and for having proposed and supervised this work. To António Ferrari for the many comments and suggestions in order to improve this thesis, and also for the corrections regarding the proper use of the English language. They deserve my sincere gratitude.

I am also indebted to Artur Pereira for his expertise in Petri nets and to António Rui Borges for the helpful discussions and suggestions concerning the contents of some chapters of this thesis.

I would also like to thank to every one in the Departamento de Electrónica e Telecomunicações and INESC, especially to all members in the Computer group, for their support and encouragement.

This work was supported by the grant PRODEP Formação nº 35/3/94. The financial support of the Human Capital and Mobility Programme of the European Community under contract CHRX-CT94-0459, the Network "Behavioural Design Methodologies for Digital Systems", that made possible the presentation and discussion of part of this work is also acknowledge.

Finally, thanks to my family and friends, in particular to my parents Maria and João and my sister Rosa for their love, patience and support. This work is dedicated to them.

Abstract

Finite state machines (FSM) have been a topic of great importance in the last five decades and have been used to specify and implement control units. Due to the increasing complexity of control units and since the FSM model does not explicitly support hierarchy and concurrency, new state-based models with hierarchical and concurrent constructions were proposed in order to overcome the limitations of the conventional FSM model and allowing the specification of complex control units in a top-down manner. Still, there are not many hierarchical FSM architectures (HFSM) that have been proposed to implement those hierarchical specifications and most of them cannot be seen as a whole FSM implementing internally in an efficient way the switching between the different hierarchical levels of the machine, except for the HFSM with stack memory.

This thesis tackles the synthesis of FSMs from hierarchical specifications and proposes two HFSMs and a parallel hierarchical FSM (PHFSM) with stack memory that can provide such facilities as flexibility, extensibility and reusability. It also presents the synthesis methodology from hierarchical specifications to the generation of state transition tables that can be used to carry out the logic synthesis of the proposed HFSM models.

Considering that the use of formal state-based models that provide hierarchical and concurrent constructions is highly recommended for specifying complex control units, hierarchical graph-schemes (HGS) and parallel hierarchical graph-schemes (PHGS) are used and some considerations about their execution and correctness are presented. It is also explained how HGSs can be used to specify a control algorithm and how it is possible to verify automatically its correctness and to validate the intended functionality through simulation.

Using the first model of a HFSM with stack memory as a starting model, two new models that can provide flexibility, extensibility and reusability and a PHFSM model that combines hierarchy and pseudo-parallel execution of operations are proposed. Their functionality, flexibility, extensibility, synchronisation and internal realisation are fully explained.

To implement a control unit specified with a set of HGSs/PHGSs it is necessary to perform the first step of the sequential logic synthesis, taking in consideration the pretended target model. The manual synthesis methodology required to build the state transition table of a HFSM/PHFSM starting from a hierarchical specification based on HGSs/PHGSs is explained for a Moore, a Mealy and a mixed Moore/Mealy FSM. A tool that automatically performs this first step for the two HFSM models proposed is also presented.

In order to validate the proposed HFSM/PHFSM models and their synthesis, the models were described in VHDL for a LUT-based implementation and simulated using the Synopsys simulation tools.

Resumo

As máquinas finitas de estados (FSM) têm sido usadas para especificar e implementar unidades de controlo e têm sido um assunto de grande importância nas últimas cinco décadas. Devido ao aumento da complexidade das unidades de controlo e uma vez que o modelo FSM não permite descrições hierárquicas e concorrentes, novos modelos formais que suportam hierarquia e concorrência têm sido propostos com o objectivo de ultrapassar as limitações do modelo FSM e que permitem a especificação de unidades de controlo complexas usando uma metodologia de decomposição hierarquizada. Apesar disso não têm sido propostas arquitecturas de máquinas finitas de estados hierárquicas, com excepção das máquinas construídas com memória *stack*, que possam ser vistas como uma máquina integral que implementa internamente e de forma eficiente a transição entre os diferentes níveis hierárquicos da máquina.

Esta tese aborda a síntese de máquinas de estados especificadas hierarquicamente e propõe duas arquitecturas de máquinas hierárquicas (HFSM) e uma máquina paralela hierárquica (PHFSM) contruídas com memória *stack*, que são flexíveis, extensíveis e reutilizáveis. Apresenta também, a metodologia de síntese lógica que permite construir a tabela de transição de estados a partir da especificação hierárquica, tabela essa que é utilizada na implementação dos modelos propostos.

Considerando que é altamente recomendável a utilização de modelos formais que permitam descrições hierárquicas e concorrentes na especificação de unidades de controlo complexas, os modelos de grafos hierárquicos (HGS) e grafos paralelos hierárquicos (PHGS) são apresentados e são feitas algumas considerações acerca da sua utilização, execução e correcção. É ainda explicado como se pode validar a especificação hierárquica da funcionalidade de unidades de controlo complexas através da verificação automática e simulação da especificação baseada em HGSs.

Os modelos propostos de máquinas de estados são apresentados detalhadamente tendo em atenção o seu funcionamento, implementação interna baseada em memórias e sincronização, bem como as novas facilidades de flexibilidade e extensibilidade que estes modelos apresentam.

É apresentada a metodologia manual da síntese lógica que é necessário implementar a partir das especificações hierárquicas baseadas em HGSs ou PHGSs de forma a construir a tabela de transição de estados que especifica a máquina hierárquica ou paralela hierárquica, para as máquinas de estados de Moore, Mealy ou mista Moore/Mealy. É também apresentado um programa que implementa automaticamente a síntese lógica dos dois modelos de máquinas de estados hierárquicas propostos a partir da especificação feita com HGSs.

Os modelos de arquitecturas propostas, bem como a metodologia de síntese, foram validadas através de uma simulação em VHDL que foi feita usando as ferramentas de simulação da Synopsys.

CONTENTS

LIST OF FIGURES

Chapter 1

Chapter 2

Chapter 3

Chapter 4

Chapter 5

Chapter 6

Chapter 7

LIST OF TABLES

Chapter 3

Chapter 5

Chapter 6

Chapter 7

1 INTRODUCTION

Summary

The goal of this thesis is the development of a methodology for the synthesis of reprogrammable control units from hierarchical specifications. The proposed methodology uses complex finite state machine models, i.e. hierarchical and parallel hierarchical finite state machines, that can provide such facilities as flexibility, extensibility and reusability and that can be easily reprogrammed.

This chapter starts by presenting an historical perspective of the evolution of the finite state machine model, starting from the Turing machine until the most recent proposals for hierarchical and parallel implementations of complex finite state machines.

Then it gives an overview of control units in particular those that can appear in embedded systems. Since control units are increasing in complexity their functionality should be specified in a top-down manner and therefore the advantages of such approach are explained. Control units can follow the finite state machine model but in order to simplify the implementation of complex control units several alternative architectures are presented. The automatic synthesis of digital circuits, with an emphasis on the sequential logic synthesis of control units, is also outlined.

Finally the objectives of the work and the structure of this thesis are presented.

1.1 Outline of the Evolution of the Finite State Machine

Finite state machine (FSM) or finite automaton is a mathematical model of a system, with discrete inputs and outputs and a finite number of states. The state of the system summarises the information concerning past inputs that is needed to determine the behaviour of the system for future inputs. Associated with a finite machine is a direct graph called a state transition diagram, which is its graphical counterpart.

Alan Turing proposed the first model of a machine or automaton in 1936, even before the appearance of the first computers. Since Turing was a mathematician he was interested in defining the fundamental relationships involved in making computations [Booth67]. The basic model of a **Turing machine** (see Figure 1.1) is a finite control, an infinite input tape and a read/write head [Booth67, Kohavi70, AleHan75, HopUll79].

The machine head scans one cell of the tape at a time and it is allowed to read from or write on the cell directly under it and to move its position one cell at a time to the right or to the left. The tape, which represents the external information store, is divided into cells and each cell holds a blank symbol or one symbol from a finite set of symbols.

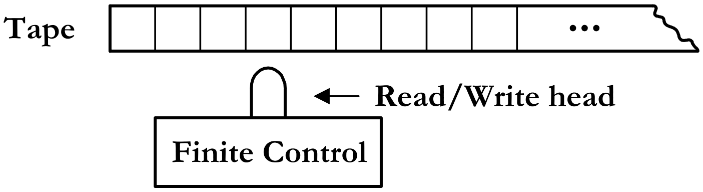


Figure 1.1 – General form of a Turing machine.

This machine can execute any process that is finitely described, consisting of discrete steps, each of which can be carried out mechanically and work as follows. During each cycle of operation the cell under the head is read to determine the symbol printed on the tape. After reading the symbol the control machine executes one of the four following possible moves: a new symbol can be written in the cell tape; the head is moved one position to the right of the current cell; the head is moved one position to the left of the current cell; the operation of the machine is halted.

Because the control element is a finite state machine, the actual operation performed will be influenced by the previous operations performed by the machine. The Turing machine can be considered a general-purpose machine and has been the base model of the finite automata.

However, a more attractive kind of machine is a Turing machine with multiple tapes, in particular the read-only machine. This kind of machine has an input tape, an output tape and a working tape and works as follows. The input tape can only move in a forward direction past its reading head, i.e. the machine can read the input tape but cannot write on the tape or recall past inputs. The output tape is similar to the input tape but it is initially blank and it can only be written when it passes through the printing head of the machine. The main tape of this machine is the working tape, which is both a read and a write tape on which all of the intermediate calculations are recorded and thus it represents the memory of the machine. Modified versions of Turing machines appear in [Booth67, HopUll79].

During the 1950s, several authors had proposed different types of read-only machines, being the two most know models the Moore and the Mealy machines. The former is due to the work of Huffman in 1954 and Moore in 1956, and the latter is due to the work of Mealy in 1955. In the **Moore machine** the symbol written in the output tape depends on the information stored in the working tape, while in the **Mealy machine** the symbol written in the output tape depends on the information stored in the working tape and on the symbol read from the input tape.

From the electronic point of view, a Moore (Mealy) machine is characterised by having a state register that holds its internal state, a next state logic function that generates the next state depending on the present state and on the inputs and an output logic function depending on the present state and in the case of Mealy depending also on the inputs. The formal definition of the FSM is presented in the Paragraph 2.3.2.

Oettinger in 1961 and Schutzenberger in 1963 conceptualised the **pushdown automaton**. The pushdown automaton is a second version of a read-only machine with a working tape that is restricted to be what is called a pushdown tape [Booth67, HopUll79]. This working tape can be written on, read from or moved in both directions, but as it moves from left to right past the reading head all the tape cells on the right are left blank [Booth67]. Such an arrangement is described as "last in first out" list. The pushdown automaton is also briefly presented in [AleHan75].

During the first half of the 1960s, Hartmanis and Stearns [HarSte66] and Kohavi [Kohavi70] have studied the **composition** and **decomposition** of finite state machines. There are three basic forms of machine composition/decomposition: **parallel**, **series** or **cascade** and **feedback** [Booth67, Kohavi70, Baranov79]. The decomposition of a FSM into several smaller interconnected FSMs is an alternative to a single monolithic implementation in order to deal with the complexity of a large machine.

The topic become less interesting during the 1970s due to the ROM-based implementation of machines, i.e. due to the use of microprogramming (concept first outlined by Wilkes in [Wilkes51]), but it had regained importance with the appearance of the first PLDs. Since they could not provide enough inputs or outputs or number of products to implement the next state and output functions of complex FSMs the realisation with a single PLD was not possible. Therefore, more recent authors [Bolton90, Baranov94, Katz94] have presented the decomposition of FSMs, with the introduction of idle states, in order to overcome the resource limitations of PLDs.

In order to deal with the increasing complexity of control units and since the state transition diagrams are not adequate to capture the notation of an algorithm, in 1973 Christopher R. Clare introduced the **algorithmic state machine** (ASM) in [Clare73]. The ASM looks like a program flowchart and it is an approach to move toward programming. Moreover, [Clare73] explores the use of ROMs in the synthesis of ASM-based designs, i.e. microprogramming. The ASM model, the synthesis of ASM-based designs and microprogramming are well documented in [WinPro80, Green86].

Clare also proposed the implementation of a finite state machine with several co-operating FSMs in [Clare73]. In other words FSMs can be linked in order to specify parallel algorithms. The linked FSMs can be completely independent or can communicate between them for synchronisation purposes and most commonly share the same clock. Linked FSMs are presented in [Clare73, Green86, Bolton90].

In 1974 Baranov proposed the **graph-scheme of algorithm**, which is very similar to the ASM, in [Baranov74]. In 1984 Sklyarov proposed an extension of the GS, the **hierarchical graph-scheme** (HGS) in [Sklyarov84]. The HGS adds support for hierarchical descriptions based on macro blocks, such as macrooperations and logic functions. This model is another step toward programming, in this case to procedural programming. The HGS can be used to describe the functionality of complex control units by decomposing them in a top-down manner, resembling an algorithmic decomposition using a structured programming language. The macrooperation can be seen as the hardware equivalent of the procedure in Pascal while the logic function as the hardware analogue of the function in Pascal.

For the hierarchical implementation once again the software model was imported and a **hierarchical FSM with stack memory** (**HFSM**), which is closely related with the pushdown automaton, was proposed in [Sklyarov84]. The stack memory controlled through push and pop instructions is used as the state register of the machine like it is used in a computer that executes a program specified with subprograms. Thus, when the execution of the machine must perform a macro block in a new hierarchical level, the stack keeps the present state unchanged and another register of the stack is used to hold the state during the execution of the macro block. When the macro block finishes executing the interrupted state of the previous hierarchical level is resumed. This HFSM can be seen as a procedural hardware implementation of a control unit.

In 1987, Sklyarov proposed an extension of the HGS, the **parallel hierarchical graph-scheme** (PHGS), which in addition to hierarchical descriptions also allows the parallel invocation of macrooperations, in [Sklyarov87]. This model can be seen as a set of FSMs that have their combinational schemes merged in a unique combinational scheme and with a state register composed of several state registers, one per machine, that are sequentially scanned by the FSM clock. The synchronisation of the parallel execution of the sub-FSMs is achieved through the introduction of waiting states. As a result a pseudo-**parallel FSM** (**PFSM**) that implements parallel tasks sequentially is achieved.

In 1987, Harel proposed the **Statechart** as an extension of the FSM state transition diagrams that allows hierarchical and concurrent descriptions in [Harel87]. And in 1989, he and Drusinsky proposed a hierarchical implementation based on a **tree of interconnected FSMs**, where each state at each level of the statechart hierarchy is represented by a machine implementing the FSM corresponding to its sub-states on the next immediate level, in [DruHar89].

In 1994, Micheli suggests representing a FSM diagram in a hierarchical way (**hierarchical FSM**) by splitting it into sub-diagrams, in [Micheli94]. Each sub-diagram, except the root has an entry and an exit state and it is associated with one or more calling states from other sub-diagrams. Each transition to a calling state is equivalent to a transition into the entry state of the corresponding sub-diagram and a transition to an exit state of the sub-diagram corresponds to return to the calling state. For the implementation of this hierarchical specification he proposes, a control unit built by **interconnecting independent control units**, each implementing a sub-diagram and having its own activation signal which start or halt its execution, in [Micheli94].

Also in 1994, Gajski proposes a **hierarchical concurrent finite state machine** (**HCFSM**) as an extension of the FSM with support for hierarchy and concurrency in [Gajski94]. According to him the statecharts model is well adapted to specify a HCFSM but he does not suggest any implementation.

1.2 Control Units Overview

1.2.1 Introduction

Embedded systems can be defined as computing and control systems dedicated to a certain application [Micheli94]. They are parts of larger systems [MicGup97] and they are widely used in the manufacturing industry, in consumer products, in vehicles, in communication systems, in industrial automation, in aerospace, etc. They are often used in life critical situations, where reliability, availability and safety are more important criteria than performance [Edwards97, MicGup97].

In the general case, an embedded system is composed of microcontrollers, application-specific integrated circuits (ASIC), field-programmable gate arrays (FPGA), as well as other programmable computing units such as digital signal processors (DSP) [Edwards97]. Since embedded systems interact with an analogue environment they often integrate components that implement A/D and D/A conversions [Edwards97, MicGup97].

The behaviour of an embedded system is defined by its interaction with the environment in which it operates [Gajski94] and in most cases they have to react continuously to their environment at the speed of the environment. These systems are called reactive systems [Micheli94, Edwards97]. Real-time systems implement functions that must execute satisfying timing constrains [Micheli94].

There are many kinds of devices that can be decomposed into a **datapath** (execution unit) and a **control unit** (see Figure 1.2) [Gajski94, Micheli94]. A particular kind of execution unit that can appear in an embedded system is a device depending on input data provided by sensors in the outside environment and generating output data that usually regulate mechanical components also in the outside environment via actuators. The actuators and sensors can be electronic, optical, mechanical, etc.

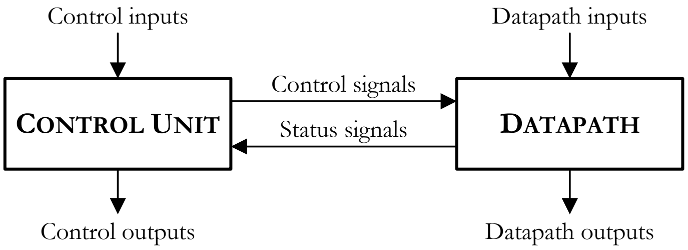


Figure 1.2 – Embedded system block diagram.

The datapath consists of registers, multiplexers and functional units such as ALUs, multipliers and shifters. A typical operation in the datapath reads the operands from the registers or external memory, computes the result in the functional units and writes the result into a destination register. The datapath is connected with external memory, being all memory accesses routed through registers with load and store operations.

The control unit is usually modelled as a FSM and consists of the state register, the next state logic to compute the next state to be stored in the state register and the control logic to drive its outputs. The control unit performs a set of instructions that depend on results of comparison operations carried by status signals from the datapath or external conditions carried by control inputs supplied by sensors in the outside environment and generates the control signals and the control outputs. The former defines what operations must be applied to which operands stored in the datapath while the latter controls actuators in the outside environment.

1.2.2 Specification of Control Units

There are two different approaches for implementing a system from simple components, namely **bottom-up assembly** and **top-down decomposition**.

Systems can be built using a bottom-up assembly procedure where primitive building blocks are clustered into more complex blocks until the desired functionality of the system is achieved. But, since it is easier to understand the operation of a whole system by looking to its components and their interactions, the top-down decomposition is a more attractive approach and it is a good strategy for constructing any kind of complex system.

Top-down decomposition is the application of the principle of "divide and conquer", which is the basic way of breaking down complexity. A top-down decomposition allows the decomposition of a complex problem into smaller pieces until manageable pieces are found. The main advantage of a top-down approach is the great flexibility allowed in the exploration of possible designs [McFKow90]. The design process starts with an initial solution where the most important decisions are made and more refinements are added at each step thus allowing the exploration of alternatives. The design representation is also simplified in a top-down decomposition, because it is never necessary to deal simultaneously with multiple levels of the design or with multiple design representations [McFKow90].

However, the specification of a complex system in a top-down manner requires the use of hierarchy. Hierarchical specifications are therefore essential to manage the complexity of systems in terms of specification size and readability and have the following advantages:

- to master the complexity through the creation of hardware (software) macro blocks using encapsulation;
- to allow the reuse of hardware (software) macro blocks;
- to allow for the migration of complex algorithms normally implemented in software to hardware.

The appropriate requirements for the specification of control units are presented in the next chapter.

The behaviour of control-dominated systems, such as embedded real-time reactive control systems, is more naturally represented in the form of states and transitions between them provoked by external events. Therefore, a state-oriented model is more suitable to describe the functionality of control-dominated systems. There are several state-oriented models and they are presented in the next chapter. However, since the FSM is the most popular state-oriented model generally the design of control units follows the FSM model. The next paragraph presents several FSM-based implementations of a control unit.

1.2.3 Implementation of Control Units

A control unit can follow the FSM model and therefore consisting of a state register, a next state logic and an output logic (see Figure 1.3a). But if a control unit has thousands of states this approach becomes very complex and the three alternative architectures depicted in Figure 1.3 and presented in [Gajski97] can be used to simplify the implementation of complex control units.

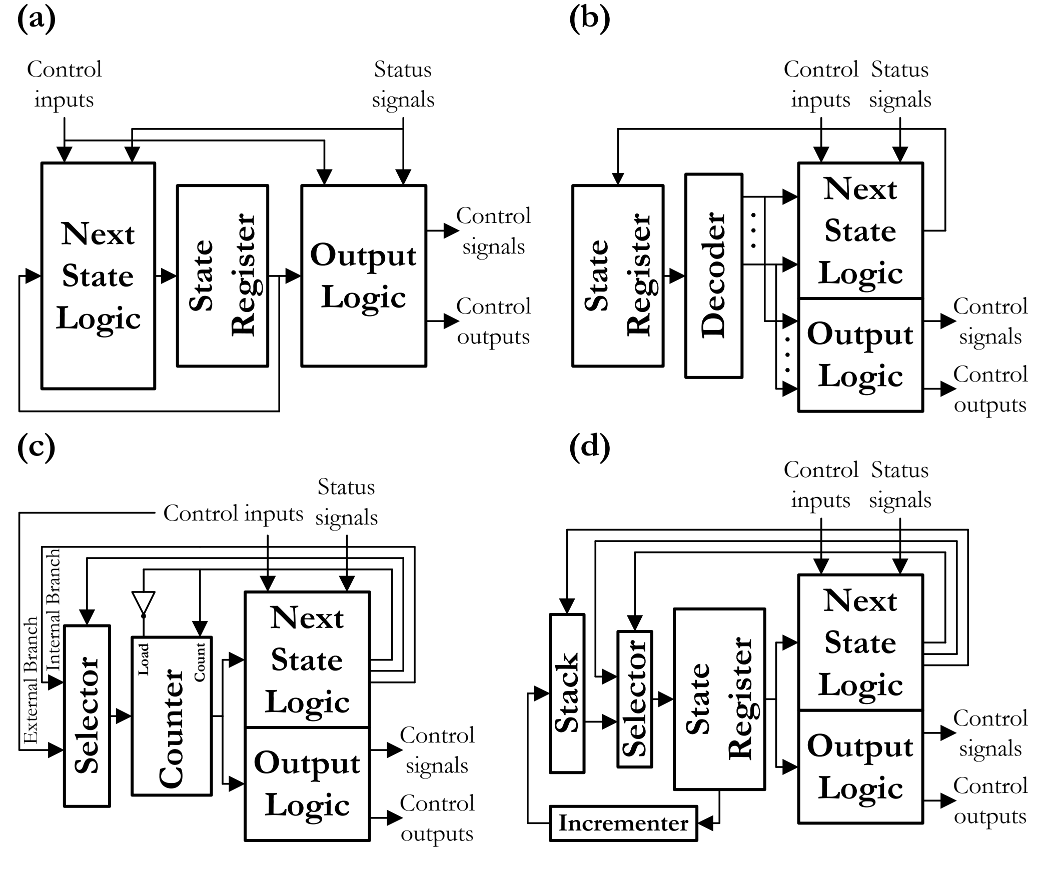


Figure 1.3 – (a) Control unit model. (b) Control unit with decoder.
(c) Control unit with counter. (d) Control unit with stack.

The first architecture (see Figure 1.3b) uses a decoder, in order to simplify the next state and output logic implementation. Since each state is identified by a state signal, which is 1 when the state register is in that particular state and 0 otherwise, the signals generated in the next state and output logic blocks, i.e. the next state, the control signals and the control outputs will fall in two situations. If they only depend on the present state they can be implemented with n-input OR gates where n represents the number of states in which each signal is asserted. If they depend on the present state and on input signals, i.e. the control inputs and the status signals, they can be implemented with AND-OR logic, having the AND gates normally only two inputs, one being the state signal and the other being the input signal.

If a control unit has many unconditional state sequences in which each state has only one next state, and if the states are encoded in a way that each state encoding can be obtained by incrementing the state encoding of its previous state, then the state register can be replaced with a counter (see Figure 1.3c). In this architecture, two more signals must be added to the next state logic [Gajski97]. A load/count signal that controls the counter behaviour, i.e. incrementing the state or loading a predefined state (branch state) to branch out of the sequence. A selector control signal that will select the proper value of the branch state, that can be supplied internally by the next state logic or provided externally through the control inputs.

In order to modularise the implementation of the control unit, frequently used tasks can be encoded as subroutines, instead of repeating the same sequence several times. For this purpose it is necessary a stack memory, which will save the state that follows the subroutine call (see Figure 1.3d). This architecture demands two more signals to be added to the next state logic [Gajski97]. A selector control signal that will select the proper state to be loaded in the state register, i.e. the next state or the state previously stored in the stack, and a push/pop signal to control the stack actuation. The state that follows the subroutine call that is saved on the stack is obtained by incrementing the state encoding of the previous state in the incrementer block.

A final strategy to simplify the control unit implementation is to replace the next state and the output logic blocks by read-only memories (ROM). When using this approach the state register acts as the ROM address register. In order to reduce the size of the next state ROM, it is very important to reduce the number of control inputs and status signals used in the next state generation. That can be done by selecting the minimal input signals with the introduction of a conditional selector in the above architecture [Gajski97]. This architecture of a control unit is usually called **microprogrammed control** and the task of converting state transition diagrams or ASM charts into ROM words is called **microprogramming**.

However, all these three architectures suggested in [Gajski97] are flattened implementations and cannot provide such facilities as flexibility, extensibility and reusability. Moreover, they cannot be implemented from a hierarchical specification, unless it is flattened into a non-hierarchical specification first.

1.2.4 Synthesis of Control Units

The synthesis process of a control unit always starts with the specification of its intended functionality and ends with the implementation of the control unit. During the process the control unit acquires different representations, which differ in the type of information they highlight.

At the specification step (**behavioural representation**) the control unit is viewed as a black box with inputs and outputs and its functionality is specified behaviourally by means of an algorithm or using a state-based formal model, like for example finite state machine diagrams or the equivalent state transition tables.

After the synthesis process the control unit is viewed as a set of components and their connections (**structural representation**). The components can be simple logic gates or alternatively programmable logic devices (PLD) such as PALs, PLAs, ROMs and basic memory elements such as flip-flops to serve as the control unit memory. Nowadays, designers can take advantage of sophisticated field-programmable devices such as FPGAs. Not only can they provide a large number of logic gates and flip-flops that can be connected in various ways, but some of them can also be reprogrammed as many times as the designer needs.

The process of generating a structural view of a logic level model with an interconnection of logic primitives is called (**sequential**) **logic synthesis** [Gajski94, Micheli94].

The first design methodology was based on a **capture-and-simulate** approach [Gajski94]. In this methodology an initial architectural block diagram specification would be produced and each functional block would be converted into a circuit schematic that could be captured by a schematic tool and then its functionality could be verified through simulation.

In recent years logic synthesis became an integral part of the design process and once logic synthesis was accepted by the design community, designers began to use Boolean expressions and finite state machine diagrams to describe logic, instead of capturing gates with schematic tools. Finally, this new methodology encouraged the practice of capturing a design through behavioural descriptions based on hardware description languages and the capture-and-simulate methodology has given way to a **describe-and-synthesise** methodology [Gajski94].

In this new methodology the design structure is generated by automatic synthesis using CAD tools instead of by manual synthesis that is very tedious for all but trivial circuits. Since this methodology can be applied on several levels of abstraction it had evolved to higher levels of abstraction with large productivity gains [Gajski94].

The automatic synthesis of digital circuits is normally divided into the four main steps depicted in Figure 1.4.

The first step is the **system-level synthesis**. At this level, the abstract functionality of a system is decomposed into different tasks, which are partitioned between hardware and software implementation. A lot of research is currently being developed and the system-level synthesis of embedded systems is normally named as software-hardware codesign or software-hardware cosynthesis [GupMic93, ThoAdaSch93, Wolf94, IsmJer95, MicGup97, StaWol97].

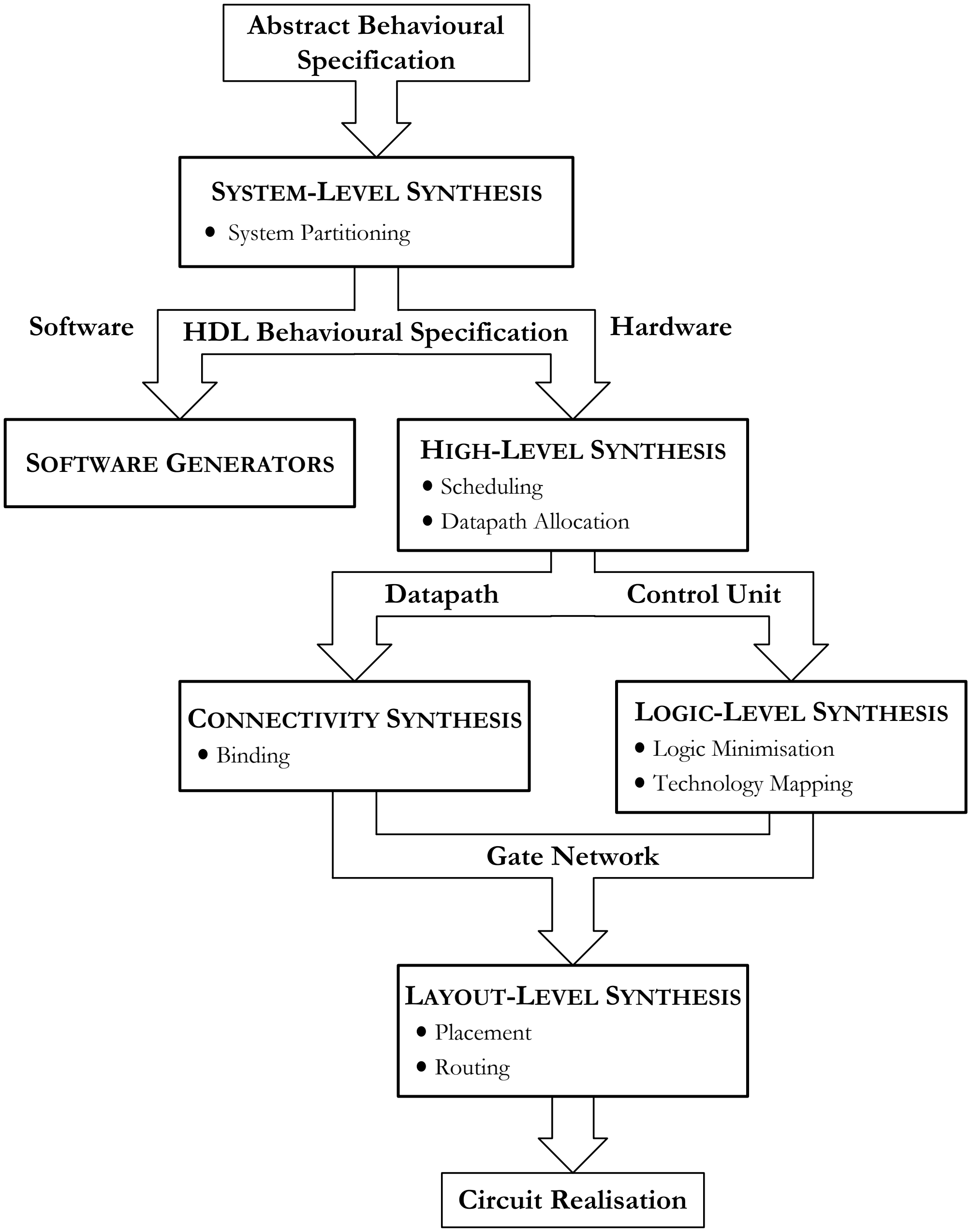


Figure 1.4 – Automatic synthesis of digital circuits.

The second step is the **high-level synthesis** (sometimes called behavioural or architectural synthesis). The tasks to be implemented in hardware are described behaviourally using a hardware description language (HDL) and the result is both a structural view of the datapath and a logic-level specification of the control unit. There are two basic tasks at this step. The **allocation** task determines the type and quantity of resources used in the datapath. The **scheduling** task makes the partition of the behavioural description into control steps (states) so that the allocated resources can compute all the variable assignments in each state. High-level synthesis is well documented in [McFParCam90, CamWol91, Gajski92, MicLauDuz92, GajRam94, Micheli94, WalCha95].

The third step is **logic-level synthesis** and it is divided in two parts. The **datapath synthesis** consists of a complete binding of the datapath, defining the interconnection among the resources, steering logic circuits like multiplexers or busses, registers, input/output ports and the control unit [Micheli94]. The **control unit synthesis** consists in the generation of a state register and the logic that generates the next state and the outputs of the control unit. The logic synthesis tasks are **logic minimisation** and **technology mapping**. Logic minimisation is used to reduce the size or delay of the logic and technology mapping transforms a technology independent logic network generated during the logic minimisation step into a network of standard gates from a particular library.

Since the control unit is modelled as a FSM, the first task of the logic-level synthesis also known as **sequential logic synthesis**, is further divided into the three following tasks normally used in the optimisation of FSMs:

1. **state minimisation** is used to decrease the number of states of the FSM, by replacing equivalent states with a single state. It is a very important task, since the number of states determines the size of the state register and combinational logic;
2. **state encoding** assigns binary codes to the abstract states of the FSM, with the purpose of minimising the next state and output functions;
3. **logic minimisation** is used to reduce the size or delay of the combinational logic that implements the next state and output functions.

Logic-level synthesis is well documented in [AshDevNew92, MicLauDuz92, Micheli94].

Finally, the **layout-level synthesis** step consists in generating the layout of the chip. The major tasks are **placement** of the components and wiring them also known as **routing** [Micheli94].

1.3 Objectives of the Work

With the increasing complexity of control units, many authors have proposed hierarchical specification models. However, the hierarchical implementations proposed for the synthesised control units did not support the versatility of the specification models.

Therefore, the goal of this thesis is the development of a methodology for the synthesis of reprogrammable control units from hierarchical specifications described with HGSs, i.e. to propose hierarchical FSM models based on the HFSM with stack memory, which can provide such facilities as flexibility, extensibility and reusability. And also to propose a FSM model that can combine hierarchy and parallelism.

Another goal is to propose a sequential logic synthesis methodology that can convert the hierarchical specification of control units in the proposed hierarchical

FSM models. Since, the sequential logic synthesis of a FSM is a well known subject, the proposed methodology will consist in the transformation of the hierarchical specification into a state transition table already minimised in terms of states, i.e. to implement automatically the first step of sequential logic synthesis. Since, manual synthesis is very tedious and error prone for all but trivial circuits, another purpose of this thesis is to create a tool that can automatically implement this synthesis methodology.

In order to validate the proposed models and the synthesis methodology, the VHDL simulation for an implementation based on lookup tables will be used.

1.4 Organisation of the Thesis

This thesis is organised as follows:

- Chapter 2 is devoted to the specification of control units. The specification requirements needed to conceptualise embedded reactive control units are described. The chapter presents the most common state-based formal models and the main characteristics of the VHDL hardware specification language.

- Chapter 3 describes in detail the graph-schemes of algorithms and how they can be used to synthesise Moore and Mealy finite state machines. The hierarchical graph-schemes and the parallel hierarchical graph-schemes are presented in detail. The facilities provided by the tool **SIMULHGS** for the verification and simulation of algorithms described by hierarchical graph-schemes are presented.

- Chapter 4 starts by introducing the implementation of a hierarchical algorithm in a finite state machine with stack memory. The first models of the hierarchical and the parallel finite state machines are briefly explained and the new proposed models are fully described. The new facilities provided by them and the concept of a virtual hierarchical finite state machine are presented. Finally a full description of the proposed synchronisation mechanism for the different machines is made.

- Chapter 5 is devoted to the synthesis of the proposed hierarchical and parallel machines. The steps that must be performed in order to transform a hierarchical algorithm to an ordinary state transition table are enumerated and explained in detail. The facilities provided by the tool **SIMULHGS** to perform the automatic synthesis of hierarchical machines are presented.

- Chapter 6 describes the internal decomposition of the machines and the optimisation techniques used for a RAM-based implementation.

- Chapter 7 presents the VHDL simulation results of the hierarchical and parallel machines and explains how to provide flexibility, extensibility and reusability. The comparison between a hierarchical and a non-hierarchical implementation of an algorithm is made.

- Finally, chapter 8 presents the final conclusions and proposes future work.

2 SPECIFICATION OF CONTROL UNITS

Summary

The aim of this chapter is to survey the specification of control units in particular those that can appear in embedded real-time reactive control systems.

The design process of an embedded system begins with the specification of its intended functionality. Since, in most cases embedded systems are very complex and heterogeneous, designers need a precise manner of capturing this functionality in order to ensure correct implementations of a system. The best way to achieve the level of precision required is to use a **formal model**. There are different kinds of formal models, but a state-oriented model is more suitable for describing control-dominated systems.

However, since a model is basically a theoretical concept, designers need to use a **hardware description language** in order to capture these formal models in a concrete form. There are different description languages, **VHDL** being the most widely used by the academic community.

There are a variety of formal models and hardware description languages. Hence, for choosing the more appropriate model and language it is necessary to first understand the **specification requirements** for conceptualising embedded systems.

2.1 Introduction

System design is the implementation of a desired functionality with a set of physical components, and the whole process starts by specifying the desired functionality. Since a natural language description is often ambiguous and incomplete, designers need a more precise way to specify the system functionality. The best way to achieve the level of precision required is to consider the system as a set of simpler objects. There are different methods for decomposing the functionality into simpler objects. They differ in the type of the objects and the rules for assembling the system functionality. Each particular method is called a (**formal**) **model** [Gajski94, Edwards97]. Moreover, in order to master the design complexity and heterogeneity, the use of formal models is recommended to ensure implementations that are correct by construction [Edwards97].

However, to establish the formal model more appropriate for capturing the functionality of control units, in particular those that can appear in embedded real-time reactive control systems, it is necessary to establish a relation between the specification requirements of the embedded systems and the characteristics of the formal models.

2.2 Specification Requirements

The requirements appropriated for conceptualising embedded systems are the following [Gajski94, GupLia97].

2.2.1 State transitions

Embedded systems are best conceptualised as a set of modes or states, where each mode represents a state of being or some arbitrary computation. They are constantly responding to external events computing their outputs as a function of their inputs and their present state. The transitions between states are determined by external events.

2.2.2 Concurrency

In many situations the representation of the system behaviour with only sequential sub-behaviours would result in complex and unnatural descriptions that can be difficult to understand. Therefore, embedded systems are more easily conceptualised as a set of concurrent sub-behaviours that collaborate with each other in order to achieve the desired functionality.

2.2.3 Hierarchy

The "Divide and conquer" principle is the basic way of handling complexity. The hierarchical specification of a system allows it to be described as a set of smaller subsystems and enables the designer to focus on one subsystem at a time. There are two kinds of hierarchy namely, structural and behavioural [Gajski94].

Structural hierarchy is defined as the process of decomposing a system as a set of interconnected components, each one of them can in turn have its own internal decomposition. It allows the designer to generate a new component from a set of already existing components. Structural hierarchy is closely related to concurrency.

Behavioural hierarchy is defined as decomposing the system behaviour into distinct sub-behaviours that can be either sequential or concurrent. It allows the designer to break down the system complexity into manageable parts.

Both structural hierarchy and behavioural hierarchy are required to allow the specification of a complex embedded system and they are essential to manage the complexity of systems in terms of specification size and readability.

2.2.4 Non-determinism

Non-deterministic behaviour is the quality of a system to be unpredictable and yielding different results from the same sequence of events. Although often non-determinism is simply the result of an imprecise eventually incorrect specification, it can be an extremely powerful mechanism to reduce the complexity of a system by abstraction [Edwards97], since it eliminates all details that are not essential to a high-level description.

However, the behaviour of a system should be predictable and even if behaviour may be non-deterministic, when there is not the complete information to predict its exact behaviour, it can be decomposed into deterministic parts [GupLia97].

There are two types of non-deterministic behaviour in conceptual models [Gajski94]: **selection non-determinism** refers to non-deterministic selection of exactly one of several choices; **ordering non-determinism** involves a non-deterministic ordering of several actions that have to be executed.

2.2.5 Behavioural Completion

Behavioural completion is defined as the ability to indicate that the behaviour has completed, i.e. that all the computations in the behaviour have been performed, and that other behaviours can detect this completion [Gajski94].

Behavioural completion is achieved in a state-based specification with the explicit definition of a set of final states, and with the control flowing to one of these final states. When using programming language constructs behavioural completion occurs when the last statement in the program has been executed.

The specification of behavioural completion has two advantages [Gajski94]: it helps conceptualising each hierarchical level of description as an independent module, facilitating its analysis and verification; it allows a natural decomposition of behaviour into sequential sub-behaviours.

2.2.6 Programming Constructs

Certain sub-behaviours of embedded systems can be specified more easily by means of mathematical expressions or an algorithm. There are several notations to describe algorithms, but programming language constructs are more usually used.

These constructs include assignment statements, branching statements (if, case statements), iteration statements (while, repeat and for loops), and subroutines (functions and procedures). The support of structured data types such as records, arrays and linked lists that allow for the modelling of complex data structures is also a very useful feature.

2.2.7 Communication

If the behaviour of a system is described as a set of concurrent sub-behaviours or processes they need to communicate with each order, in order to achieve the desired functionality. This kind of communication between them is usually conceptualised in term of the **shared memory** or the **message passing** paradigms [Gajski94].

In the shared memory model, each sending process writes to a shared medium, such as a global variable or port, which can be read by all receiving processes [Gajski94]. The shared medium can be **persistent** or **non-persistent**.

A persistent shared medium is one that retains the value written by one process, until that value is rewritten by another process, while in a non-persistent shared medium the data is only available at the instance when it is written, since it is not retained by the medium between two successive writes [Gajski94].

In the message-passing model, the data is transferred between processes over an abstract medium called a **channel**, using send-receive primitives [Gajski94].

2.2.8 Synchronisation

When the behaviour of a system is described as a set of concurrent sub-behaviours or processes, each process may generate data and events that need to be recognised by the other processes. In such cases, data exchanged between processes or actions performed by different processes at the same time may need to be synchronised [Gajski94]. There are two synchronisation methods namely, **control-dependent** and **data-dependent**.

In a control-dependent synchronisation mechanism, the control structure of the process is responsible for the synchronisation [Gajski94]. In addition to that, the synchronisation can be achieved by means of the following inter-process communication methods [Gajski94]: **shared-memory based synchronisation**; **synchronisation by common event**; **synchronisation by status detection**; **synchronisation by message passing**.

2.2.9 Exceptions

In some cases, the occurrence of a certain external event, like a reset or an interrupt, demands that a behaviour will be immediately terminated rather than having to wait for the computation to complete, and that a predefined behaviour will be executed instead.

2.2.10 Timing

Since in real-time systems the performance is measured in terms of how well it respects the timing constraints, it is important the notion of timing to reflect real implementations, i.e. by specifying time the simulation results obtained are more realistic. There are two ways of specifying timing information namely, **functional timing** and **timing constrains** [Gajski94].

Functional timing is defined as all timing information that affects the simulation output of the system specification, and therefore adding functionality to the system. Timing constrains are utilised in the specification of a system in order to be used by simulation and synthesis tools.

2.3 Specification models

2.3.1 Introduction

The purpose of a model is to provide an abstract view of a system and in order to be useful should possess the following qualities [Gajski94]: it should be formal to provide no ambiguity; it should be complete to allow describing the entire system; it should be comprehensive and easy to modify to allow future changes in the system functionality; it should be natural enough to help the designer to understand the system.

A model of a design should consist of the following components [Edwards97]: a functional specification; a set of properties that the design must satisfy; a set of performance indexes that evaluate the quality of the design in terms of cost, reliability, speed, size, etc.; a set of constraints.

In general, the models fall into the following five distinct categories: state-oriented; activity-oriented; structure-oriented; data-oriented and heterogeneous.

A state-oriented model describes a system in terms of states and transitions between them provoked by external events. A state-oriented model is more suitable for describing control-dominated systems, such as embedded real-time reactive control systems, where the temporal behaviour of the system is the most important feature of the design. Basically there are the following state-oriented models: finite state machines (FSM); algorithmic state machines (ASM); graph-schemes of algorithms (GS); Petri nets and Statecharts.

2.3.2 Finite State Machine model

Since the behaviour of control-dominated systems is more naturally represented in the form of states and transition between states, the most popular state-oriented model is the **finite state machine** model (**FSM**). A FSM can be represented graphically through a state transition diagram (see Figure 2.1a) or textually through a state transition table (see Figure 2.1b), and it can be formally described as a quintuple,

$$< A, X, Y, \delta : A \times X \rightarrow A, \lambda : A \times X \rightarrow Y >$$

where $A=\{a_0,a_1,\ldots,a_M\}$ is a finite set of **states**, $X=\{x_1,\ldots,x_L\}$ is a finite set of **inputs**, $Y=\{y_1,\ldots,y_N\}$ is a finite set of **outputs**, δ is the **transition function** or the **next state function**, which determines the next state from the present state and the inputs, and λ is the **output function**, which determines the outputs from the present state and the inputs.

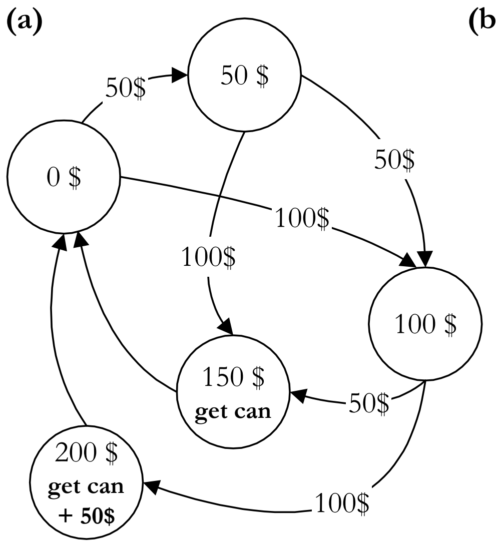
There are two well-known types of FSMs that are, the **transition-based Mealy** FSM and the **state-based Moore** FSM. They differ in the definition of the output function. In Moore the outputs depend only on the present state (λ: $A \rightarrow Y$), while in Mealy the outputs depend on both the present state and the inputs (λ: $A \times X \rightarrow Y$).

In other words, the outputs are associated with states in Moore, while in Mealy they are associated with transitions. In practical terms, the major difference between the two models is that Moore may require more states than Mealy to describe the behaviour of a control system. This is because Mealy can have multiple arcs pointing to a single state with each arc having a different output value, while Moore demands a different state for each different output value.

Let's consider the example of a vending machine that delivers drink cans after it has received one hundred and fifty Portuguese escudos (150$). The machine accepts coins of 50$ and 100$, one at a time. If the consumer supplies three coins of 50$ or one coin of 50$ and one coin of 100$ he receives a can, but if he supplies two coins of 100$ he receives a can and 50$ of change.

Let's assume the following rules in order to keep the example simple: the machine only supplies one kind of can so there is no need for pushing any button to retrieve the can; there is not a cancel button to retrieve the already inserted coins; and the consumer does not insert any extra coins after having inserted enough money and while waiting for the can and the change.

The vending machine state transition diagram (Moore FSM type) depicted in Figure 2.1a only includes transitions that explicitly cause a state transition. The machine remains in a state while a coin is not inserted. On the other hand the outputs *get can* and *50$ change* are represented only in the states where they are asserted. On all others states they are negated. The equivalent and complete state transition table is presented in Figure 2.1b.



(b)

Present State & Outputs		Inputs		Next
(get can 50$ change)		100$	50$	State
0 $	(0 0)	0	0	0 $
		0	1	50 $
		1	0	100 $
		1	1	×
50 $	(0 0)	0	0	50 $
		0	1	100 $
		1	0	150 $
		1	1	×
100 $	(0 0)	0	0	100 $
		0	1	150 $
		1	0	200 $
		1	1	×
150 $	(1 0)	×	×	0 $
200 $	(1 1)	×	×	0 $

Figure 2.1 – (a) Vending machine state diagram. (b) Vending machine state transition table.

In general the FSM is suitable for modelling control dominated systems, but since the FSM model does not explicitly support hierarchy and concurrency, it is not suitable for modelling complex systems due to an explosion in the number of states [Harel90, Gajski94, Edwards97].

2.3.3 Algorithmic State Machine model

State diagrams are not adequate to capture the notation of an algorithm and they are weak in capturing the structure behind complex sequencing [Katz94]. The **algorithmic state machine** model (**ASM**) introduced by Clare in [Clare73] is an alternative way to describe a FSM behaviour that looks like a program flowchart.

The ASM chart is used to design a state machine that implements an algorithm. It is a graphical description of the output and next state functions of the state machine and when completed it becomes part of the design documentation.

The ASM chart consists of one or more interconnected ASM blocks. One **ASM block** (see Figure 2.2) describes the state machine operation during one state time, and represents the present state, the state outputs, the conditional outputs and the next state for a set of inputs. Therefore, the output and next state functions are represented by the ASM chart on a state by state basis, with only one restriction imposed to the ASM blocks interconnection. This restriction is that there must be only one next state for each state and a stable set of inputs [Clare73].

The ASM block (see Figure 2.2) has one entry path and any number of exit paths. It is composed of one state box, and a network of decision boxes and conditional output boxes. This network can have any number (zero is allowed) of decision and conditional output boxes.

A state is represented by a **state box** (see Figure 2.2) and it has the following information: a name encircled on the left or right side of the state box; a code that is probably unknown when first drawing the ASM description; an output list selected from a defined set of operations written inside the state box. The output list mentions the signals that are asserted whenever the state is entered. It is possible to specify if the signal is asserted immediately or if it is delayed until the next clock event. Usually the immediate signals are prefixed with the letter I, while the delayed signals are not prefixed.

The **decision box** (see Figure 2.2) involves the inputs to the state machine and gives the conditions that control the state transitions and the conditional outputs. The box contains a Boolean expression that determines the ASM block to be entered next. The decision box has two exit paths. The True Exit Path usually indicated by 1 or T, is taken when the enclosed condition is true and the False Exit Path usually indicated by 0 or F, is taken when the enclosed condition is false. The order in which the condition boxes are cascaded is irrelevant for the determination of the next ASM block [Katz94].

The **conditional output box** (see Figure 2.2) describes other outputs, which are dependent on input signals in addition to the state of the machine. The output signals written inside the condition box can also have immediate and delay qualifiers.

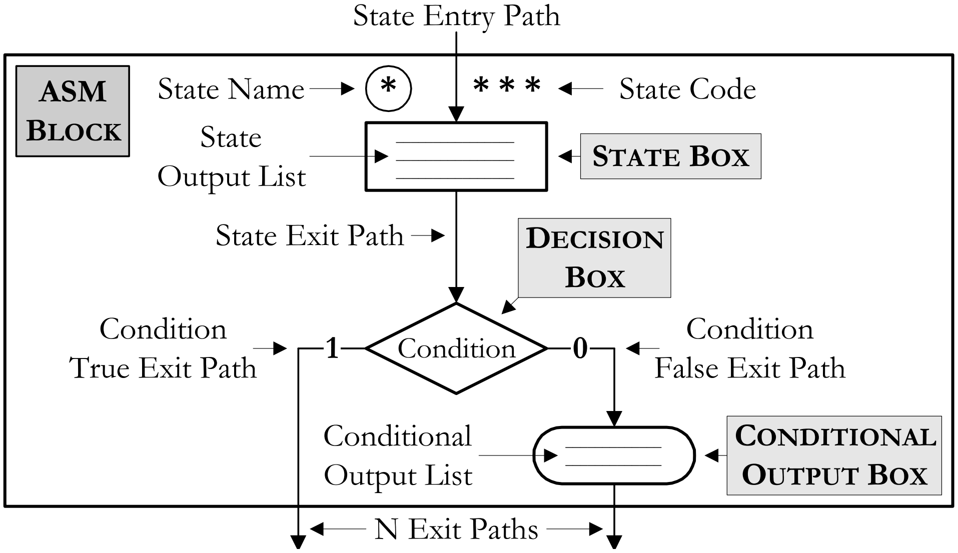


Figure 2.2 – The ASM block.

The ASM chart of the vending machine state transition diagram presented in Figure 2.1a is depicted in Figure 2.3.

In order to simplify the ASM chart drawing, the outline box of the ASM block can usually be omitted, because the block is clearly defined to include all the conditional boxes and conditional output boxes between one state and the next. Moreover, some of the conditional boxes from one state can be shared by another state [Clare73].

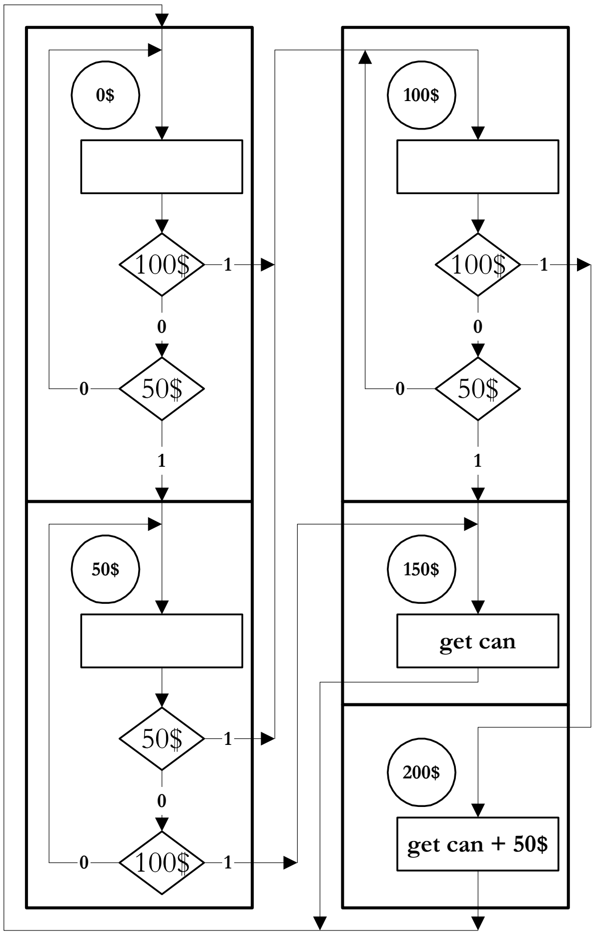


Figure 2.3 – Vending machine ASM chart.

The ASM model is well documented in [WinPro80, Green86]. Like the FSM model the ASM model does not explicitly support concurrency or hierarchy and it is not suitable for modelling complex systems.

2.3.4 Graph-Scheme of Algorithm model

The **graph-scheme of algorithm** model (**GS**) was proposed in [Baranov74]. It is also presented in [Baranov94] and it is described in detail in the next chapter.

A GS is a directed connected graph, which is composed of an initial rectangular node labelled with **Begin**, a final rectangular node labelled with **End** and a finite set of **rectangular** and **rhomboidal** nodes. Each rectangular node, apart from the nodes **Begin** and **End** lists the output signals that are asserted whenever the node is reached. Each rhomboidal node tests one input signal in order to determine the path to follow. The GS of the vending machine state transition diagram presented in Figure 2.1a is depicted in Figure 2.4.

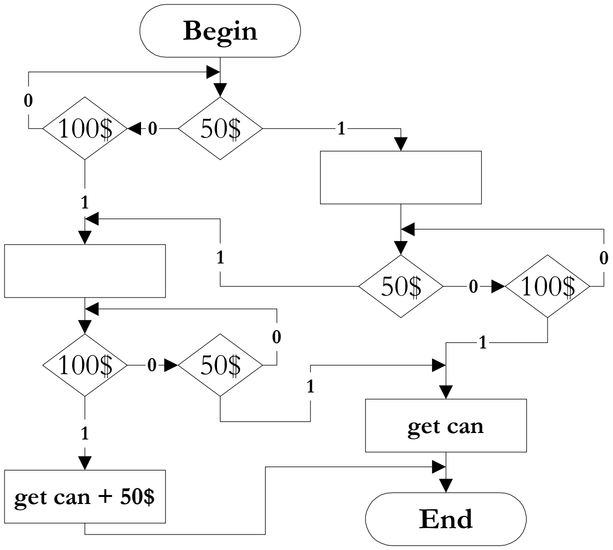


Figure 2.4 – Vending machine GS description.

Like the FSM and the ASM models, the GS model does not explicitly support concurrency or hierarchy and it is not suitable for modelling complex systems. However, the **hierarchical graph-schemes** (**HGS**) introduced in [Sklyarov84] support hierarchical descriptions based on the use of **macrooperations** and **logic functions**. The **parallel hierarchical graph-schemes** (**PHGS**) introduced in [Sklyarov87] in addition to hierarchical descriptions also allow macrooperations invoked in parallel. They are both suitable for modelling complex systems and they are further described in the next chapter.

2.3.5 Petri net model

The **Petri net** model is a state-oriented model for describing and studying information processing systems that are concurrent, asynchronous, distributed, parallel, non-deterministic and stochastic [Murata89].

The Petri net graphical model consists of a set of **places**, a set of **transitions** and a set of **tokens** (see Figure 2.5). Tokens inhabit in places and flow through the net by being consumed and produced whenever a transition fires, and they are used to simulate the dynamic and concurrent activities of the system. A Petri net can be formally described as a quintuple [Murata89],

$$PN = (P, T, F, W, M_0)$$

where $P=\{p_1,\dots,p_M\}$ is a finite set of **places**, $T=\{t_1,\dots,t_N\}$ is a finite set of **transitions** with P and T being disjoint sets. $F \subseteq (P \times T) \cup (T \times P)$ is a set of **arcs** between places and transitions (flow relation), $W: F\rightarrow\{1, 2, 3, \dots\}$ is a **weight function** and $M_0: P\rightarrow\{0, 1, 2, 3, \dots\}$ is the **initial marking**, i.e. the initial number of tokens in each place.

In order to simulate the dynamic behaviour of a system, the Petri net marking changes according to the following transition (firing) rules [Murata89]:

1. a transition t is enabled if each input place p of t is marked with at least w(p, t) tokens, where w(p, t) is the weight of the arc from p to t;
2. an enabled transition may or may not fire, depending on whether or not the event actually takes place;
3. a firing of an enabled transition t removes w(p, t) tokens from each input place p of t, and adds w(t, p) tokens to each output place p of t, where w(t, p) is the weight of the arc from t to p.

A transition without any input place is called a source transition and a transition without any output place is called a sink transition [Murata89].

The Petri net that represents the functionality of the vending machine is depicted in Figure 2.5. There are six places graphically represented as circles (*0$*, *50$*, *100$*, *150$*, *50$ change*, *no change*) and eight transitions graphically represented as bars (three *50$ coin*, three *100$ coin*, *get can*, *get can + 50$*). The marking function assigns one token to the places *0$* and *no change* and zero tokens to the remaining places.

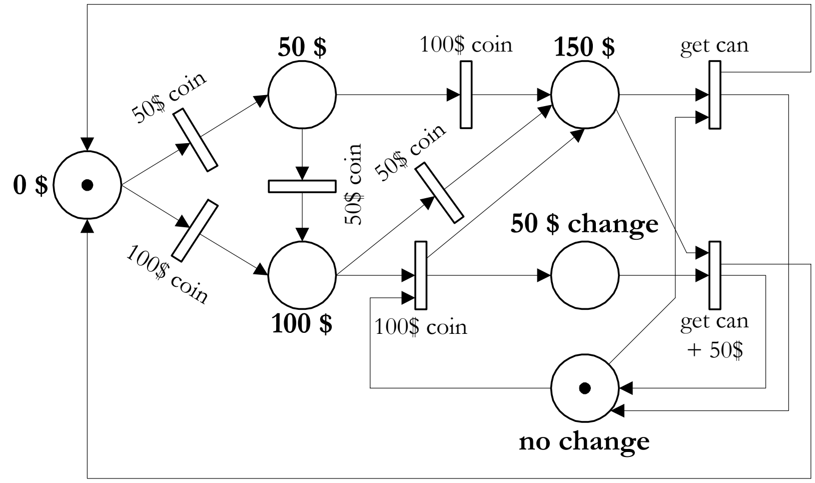


Figure 2.5 – Vending machine Petri net.

Any finite state machine or its state diagram can be modelled by a subclass of Petri nets called state machines. State machines are Petri nets with only one token and where each transition has exactly one incoming arc and exactly one outgoing arc.

The state machine that describes the state diagram presented in Figure 2.1a is depicted in Figure 2.6 and it is equivalent to the Petri net presented in Figure 2.5. The five states of the FSM are represented by the five places (*0$*, *50$*, *100$*, *150$*, *200$*), where the initial state (place *0$*) is indicated by having one token. The transitions between states are shown by the six transitions labelled with input conditions (three *50$ coin*, three *100$ coin*) and the outputs of the state machine are generated in the two transitions *get can* and *get can + 50$*.

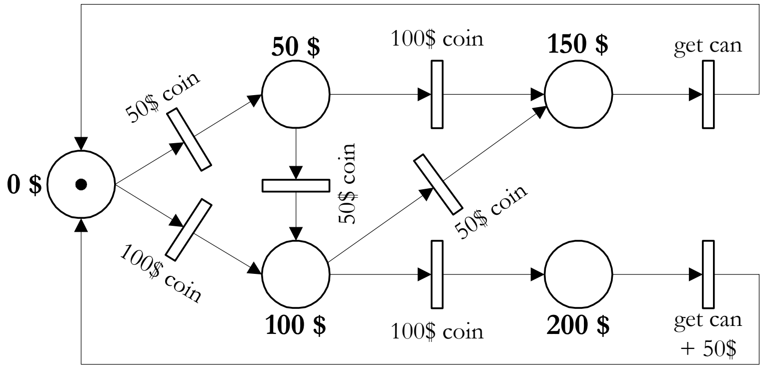


Figure 2.6 – State machine equivalent to the previous Petri net.

The structure of the place *0$* having the two output transitions *50$ coin* and *100$ coin*, is referred to as a conflict, decision or choice depending on applications. State machines allow the representation of conflicts, but not the synchronisation of parallel activities.

Petri net models can be used to check certain system properties, such as safeness and liveness. A Petri net is said to be safe if the number of tokens in each place does not exceed one token. A Petri net is said to be live if, no matter what marking has been reached from M_0, there is always one transition that can fire. For that reason, a live Petri net guarantees deadlock-free operation, no matter what firing sequence is chosen. The Petri nets of Figure 2.5 and Figure 2.6 are both safe and live.

Although a Petri net does have many advantages in modelling concurrent systems it does not support hierarchy, and like the FSM, the ASM and the GS models it is not suitable for modelling complex systems.

2.3.6 Statecharts model

The Statecharts model was introduced in [Harel87] as a visual formalism for specifying the behaviour of complex reactive systems [DruHar89, Harel90]. To demonstrate the formal syntax and semantics of Statecharts the output-free Statechart depicted in Figure 2.7 and presented in [DruHar89] will be used.

Like the FSM model, Statecharts are based on states, events and conditions with the latter two causing transitions between states. States and transitions can be associated in various ways with output events, called actions, which can be triggered either by executing a transition, or by entering, exiting or being in a state.

They combine the Moore and Mealy FSMs extended with hierarchical and concurrent constructions, in order to overcome the limitations of the conventional FSM model to specify complex reactive systems [DruHar89].

Statecharts are a graphical language, where states are represented with rounded rectangles that can be repeatedly combined into higher level states, or alternatively high level states can be detailed into lower level states, using AND and OR clustering modes.

Figure 2.7 shows an AND state A composed of two states B and C separated by a dash line, meaning that when the system is in A it must be in B and in C. In other words B and C are orthogonal states. However B and C are OR states, meaning that when the system is in B it must be in D or E or F, and when the system is in C it must be in G or H. The states (D, E, F) and the states (G, H) are exclusive states. Thus when the system is in the state A there are the following possible state configurations: (D, G); (D, H); (E, G); (E, H); (F, G); (F, H). The arrows beginning with a dot (default arrow) and pointing to the states E and G means that those are the initial states of B and C respectively. Therefore the initial state of A is the configuration (E, G).

Transitions in a Statechart are not restricted to a level and can lead from a state on any level of clustering to any other state [DruHar89]. Some examples are shown in Figure 2.7. The event *a* causes a transition from the state K to the state L. The event *b* causes a transition from the state J, which means from the state L or M, to the state K. The event *c* causes a transition from A, from one of the state configurations listed above, to the state M. In the case of the event *d* the transition is made to the super state J, or in other words to its initial state L that is the state with the default arrow.

Transitions are in general from configurations to configurations with the possibility of orthogonal components in the source and target states. The event *f* in Figure 2.7 causes a transition from the configuration (F, H) to the state P, or from the state P to the configuration (D, H).

Concurrency and independence are both made possible by orthogonality [DruHar89]. The event *m* in Figure 2.7 causes a simultaneous transition from E to F and from G to H if the configuration is (E, G), but the event *p* causes a transition from E to D independently of what is happening in C.

Outputs can be associated with transitions as in a Mealy FSM by writing *a/o* along an arrow triggered by event *a* that will assert the action *o*. Similarly *o* can be associated with entering (*entry o*), exiting (*exit o*) or being (*throughout o*) in a state like in a Moore FSM. In either case *o* can be an external event (action), or an internal one that can be used to synchronise other transitions in some orthogonal states [DruHar89].

Statecharts allow timing specifications in states. However, since those timing constraints can appear in states at any level and in any orthogonal component, it actually allows global timing constraints [DruHar89].

Statecharts provide various synchronisation methods. An event that reaches a state boundary box synchronises the state. For example, event *e* in Figure 2.7 reinitialises state A to its initial configuration (E, G). Another way is asserting an internal variable that will be used as an event to synchronise other transitions.

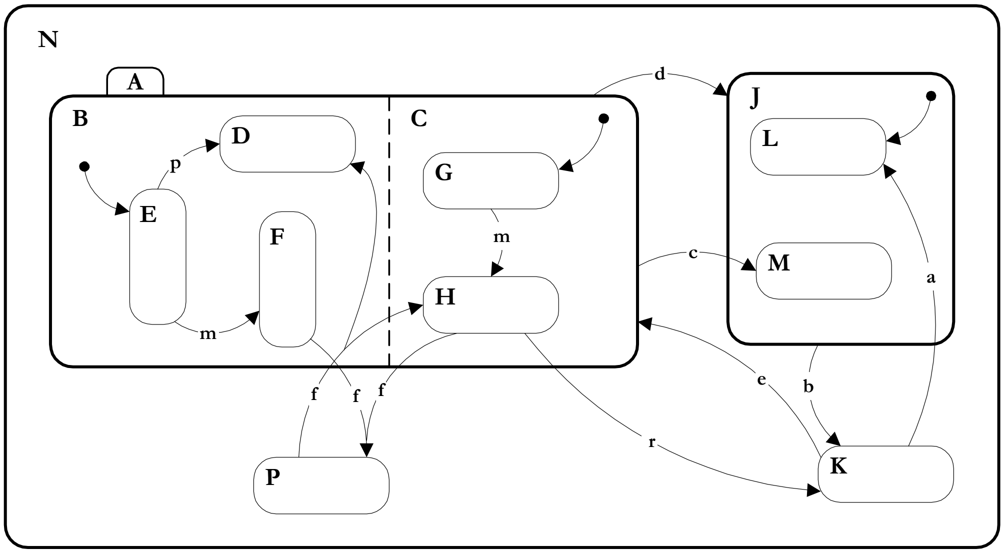


Figure 2.7 – A Statechart example.

Statecharts allow for the description of complex reactive systems, because they support hierarchical and concurrent descriptions, timing specifications and synchronisation methods, but like any other state-oriented model it is tailored for control-dominated systems, with the data associated with activities within states or along transitions [DruHar89]. As a result they are not suitable for modelling complex systems, which may require complex data structures [Gajski94].

The Statechart of the vending machine presented in Figure 2.1a is depicted in Figure 2.8.

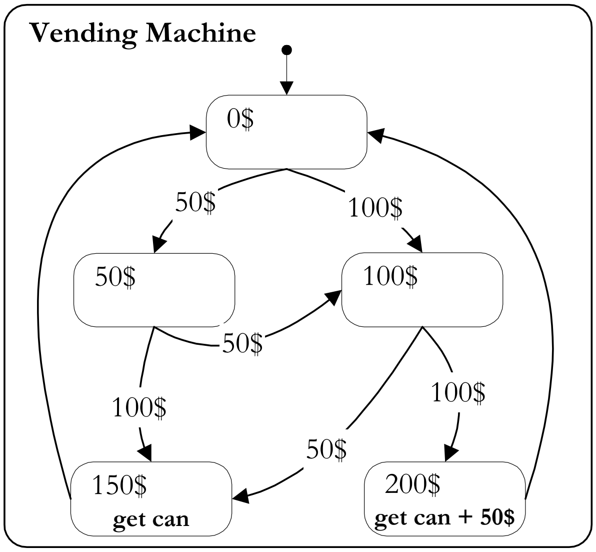


Figure 2.8 – Vending machine Statechart.

2.4 Specification languages

2.4.1 Introduction

The formal models presented in the previous paragraph can be used to understand and describe the system functionality. But, since a model is a theoretical concept, designers need an executable specification language capable of capturing the system functionality in a simulatable form [Gajski94].

Such an approach has the following advantages [Gajski94]: designers can verify the correctness of the intended functionality of a system through simulation; the specification can be used as an input to synthesis tools; the specification can serve as part of the system documentation and to exchange the design information between different designers and tools.

To be useful, a design language must help the designer to meet the following goals [GupLia97]: to model correctly and unambiguously the hardware behaviour at various levels of abstraction; to simulate the hardware model along with the rest of the system that can contain software parts; to synthesise an efficient hardware solution using existing CAD tools.

The two most widely used hardware specification languages are VHDL [IEEE94] and Verilog [ThoMoo91]. The latter is mainly used in industry, while the former is widely accepted by the design community, specially the academic community, as a description, simulation, verification and synthesis language and a large number of tools using graphical environments were developed for it like for example EaseVHDL, VSystem, ViewLogic and Synopsys.

2.4.2 VHDL

In the search for a standard design and documentation tool for the VHSIC (Very High Speed Integrated Circuits) program the United States Department of Defense (DoD) sponsored a workshop on hardware description languages in the summer of 1981. Based on the recommendations of that workshop, the DoD established in 1983 the requirements for a standard VHSIC Hardware Description Language (VHDL) and contracted IBM, Texas Instruments and Intermetrics corporations for its development [Navabi93]. The VHDL language was standardised by the IEEE in 1987.

VHDL borrowed some features from the ADA language and can be used to represent and describe hardware components and systems. Since it was created as a language for specifying large systems, readability was preferred to writability and consequently the language is fairly verbose [Micheli94].

The description of a component consists of an interface specification and an architectural specification. The interface description is identified by the keyword

entity and contains the input and output ports of the component, and other external characteristics such as time.

An architectural description is identified by the keyword **architecture** and describes the component functionality. This functionality depends on the ports and the other parameters specified in the interface specification. It can be described behaviourally using programming constructs, structurally using existing components, in a dataflow manner specifying the flow of data through the registers and buses (register transfer level) or using a combination of the above. An entity can have more than one architectural specification.

VHDL supports concurrent instantiation of components, which is the basic construct for structural hierarchy. A structural description consists of instantiations of already existing components and the interconnections between them are specified using signals.

VHDL supplies the *process* construct in order to describe behaviourally a component. A process is a statement that is active at all times, executing concurrently with other processes and that can be made sensitive to selected signals using a sensitivity list. A *process* is identified by the keyword **process** and it is composed of a declarative part and a statement part. The statement part of a process is sequential, always active, triggered by the signals declared in the sensitivity list and it executes in zero time. The statement part of a process can use functions and procedures and can select and assign values to signals using *if*, *case* and loop (*for* and *while*) statements.

VHDL supports a two-level behaviour hierarchy [Gajski94], the first level being a specification decomposed into a set of concurrent processes, the second level being a sequential decomposition of these processes into procedures.

VHDL has two kinds of objects that can be used for carrying values from one point in the program to another, namely variables and signals. Signals have hardware significance and differ from variables in that they have a time component associated with them. The *after* clause allows signal assignment statements to schedule future value updates. Signals can be used in sequential and concurrent bodies of VHDL, they can be global, but they can only be declared in concurrent bodies of VHDL. Variables on the other hand are mainly used for keeping intermediate values, they can only be declared and used in sequential bodies of VHDL, and they are local to the body in which they are declared.

In addition to the programming constructs already mentioned that can be used inside a process statement, VHDL also offers a wide range of data types suitable for high-level behavioural modelling such as integer, real, enumeration, physical, array, record and pointer types. It also provides logical, relational and arithmetic operators. The latter, however apply only to the integer and real data types. VHDL also allows the overloading of operators.

VHDL also provides a package mechanism to encapsulate declarations and subprograms that can be included in any VHDL program. It allows the construction of libraries of commonly used declarations, procedures and functions into packages enhancing the modularity and reusability of the models [Micheli94].

Exceptions are supported by VHDL using guarded concurrent signal assignments, but there are no constructs for terminating the execution of a process in response to an exception [Gajski94].

In VHDL, communication between processes can be achieved by a shared memory model, based on signals that can be assigned by any process and that are visible to other processes.

In VHDL, synchronisation can be achieved in one of the two following ways. The first is based on the sensitivity list of a process, which ensures that the process will begin to execute when an event occurs on any of the signals mentioned in the sensitivity list. The second employs the *wait* statement, which suspends the process until it detects either the occurrence of an event on any of the specified signals or the occurrence of the specified condition.

In VHDL it is possible to specify functional timing using for example an after clause, while timing constraints can be indirectly specified using attributes [Gajski94].

VHDL does not support state transitions, and true behavioural hierarchy in which concurrency can be specified at any level of the hierarchy [Gajski94], and only through global variables it is possible to make a VHDL description non-deterministic.

The VHDL behavioural description of the vending machine state transition diagram presented in Figure 2.1a is depicted in Figure 2.9.

2.5 Conclusions

Control units such as embedded real-time reactive control systems are intrinsically state-based systems and since in most cases, their specification as a whole will lead to a solution with a huge number of states, they are too complex to be considered in their entirety. To avoid incomprehensive and eventually erroneous descriptions, their functionality is more easily described when hierarchically decomposed into a set of sequential and concurrent behaviours using a top-down decomposition.

Since hierarchy, concurrency and non-determinism can be used in order to reduce the size of the representation of system behaviour when compared to a flat deterministic representation [Edwards97], the use of formal state-based models that can support hierarchical and concurrent specifications is highly recommended by many authors [DruHar89, Gajski94, Micheli94, Edwards97] for modelling the functionality of complex control units.

Among the formal state-based models, Statecharts and HGSs/PHGSs are the only models that can provide hierarchical and concurrent decompositions and therefore they can be used to model the behaviour of complex control units that do not require complex data structures, which is the case of control-dominated systems.

Since VHDL provides features that can support the most important specification requirements of embedded systems, it can be used for capturing the functionality of control units with two advantages: wide acceptability and availability of simulators.

```
entity VENDING_MACHINE is
    port ( CLOCK, COIN_50, COIN_100 : in bit; GET_CAN, GET_CHANGE : out bit );
end VENDING_MACHINE;

architecture BEHAVIOURAL of VENDING_MACHINE is

    type STATE is (STATE_0, STATE_50, STATE_100, STATE_150, STATE_200);
    signal CURRENT_STATE : STATE;

begin

    process( CLOCK )
    begin
      if ( CLOCK = '1' and CLOCK'event ) then
        case CURRENT_STATE is
          when STATE_0 =>   if COIN_50 = '1'
                                then CURRENT_STATE <= STATE_50;
                                elsif COIN_100 = '1'
                                      then CURRENT_STATE <= STATE_100;
                            end if;
          when STATE_50 =>  if COIN_50 = '1'
                                then CURRENT_STATE <= STATE_100;
                                elsif COIN_100 = '1'
                                      then CURRENT_STATE <= STATE_150;
                            end if;
          when STATE_100 => if COIN_50 = '1'
                                then CURRENT_STATE <= STATE_150;
                                elsif COIN_100 = '1'
                                      then CURRENT_STATE <= STATE_200;
                            end if;
          when STATE_150 | STATE_200 => CURRENT_STATE <= STATE_0;
        end case;
      end if;
    end process;

    GET_CAN <= '1' when ( CURRENT_STATE = STATE_150 or
                          CURRENT_STATE = STATE_200 ) else '0';
    GET_CHANGE <= '1' when CURRENT_STATE = STATE_200 else '0';

end BEHAVIOURAL;
```

Figure 2.9 – Vending machine VHDL behavioural description.

3 HIERARCHICAL GRAPH-SCHEMES

Summary

The previous chapter has briefly introduced the **graph-schemes of algorithms** (**GS**) as a state-oriented formal model. The aim of this chapter is to present the basic concepts of GSs and how they can be used to synthesise a Moore and a Mealy **finite state machine** (**FSM**). The presented formal definition and notation closely follows [Baranov94].

This chapter also introduces the basic concepts of **hierarchical graph-schemes** (**HGS**) and **parallel hierarchical graph-schemes** (**PHGS**). Some considerations concerning execution, synchronisation and correctness of a HGS/PHGS are presented.

Finally, the C++ tool **SIMULHGS** is introduced and it is explained how it can be used to construct, verify and simulate a hierarchical algorithm described by a set of HGSs.

3.1 Graph-Schemes of Algorithms

A **graph-scheme of algorithm** (**GS**) is a directed connected graph [Baranov94], which is composed of an initial node **Begin**, a final node **End**, and a finite set of operational nodes (rectangular nodes) and conditional nodes (rhomboidal nodes) (see Figure 3.1).

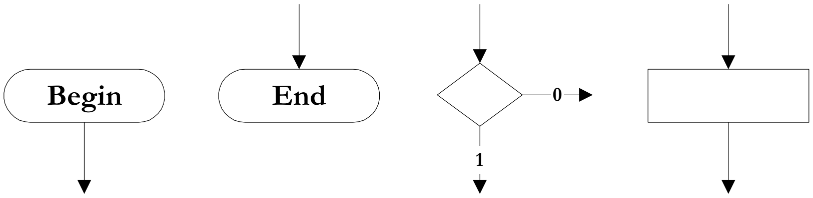


Figure 3.1 – Nodes of GS.

It has the following formal description [Baranov94]:

- each GS has one entry point which is an operational node marked with a **Begin** label, and one exit point which is an operational node marked with an **End** label;
- other operational nodes contain microinstructions from the set $\iota=\{Y_1,Y_2,\ldots,Y_T\}$. Any microinstruction Y_t, includes a subset of microoperations from the set $Y=\{y_1,\ldots,y_N\}$. A microoperation is an output signal, which causes a simple action in the datapath such as setting a register or incrementing a counter. It is possible to write the same microinstruction in different operational nodes;
- each conditional node contains just one element from the set X, where $X=\{x_1,\ldots,x_L\}$ is the set of logic conditions. A logic condition is an input signal, which communicates the result of a test, such as the state of a sensor. It is possible to write the same logic condition in different conditional nodes;
- all nodes, except the node **Begin**, have only one input. The node **Begin** has no inputs. All operational nodes, apart from the node **End**, have only one output. The node **End** has no outputs. A conditional node has two outputs marked with “1” (true) and “0” (false);
- Inputs and outputs of the nodes are connected by directed lines (arcs), which go from the output to the input in such a way that:
    - every output is connected with only one input;
    - every input is connected with at least one output;
    - every node is located on at least one of the paths, which go from the node **Begin** to the node **End**. A GS with sub-graphs containing infinite cycles will not be considered;
    - one of the outputs of a conditional node can be connected with its input. Such conditional node is called a waiting node.

An example of a graph-scheme with 7 logic conditions ($x_1,\ldots,x_7$) and 8 microoperations ($y_1,\ldots,y_8$) is depicted in Figure 3.2.

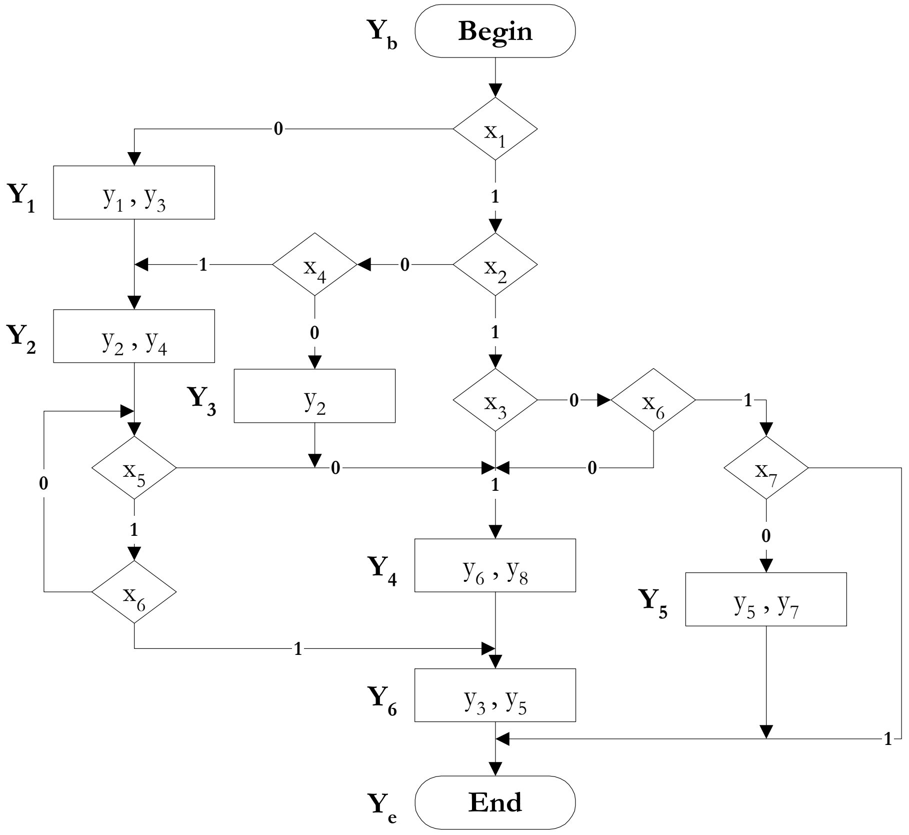


Figure 3.2 – An example of a GS.

3.2 Execution of a GS

3.2.1 GS Traverse Procedure

Denote all possible L-component vectors of the logic conditions $x_1,\ldots,x_L$ by $\Delta_1,\ldots,\Delta_{2^L}$ and suppose that the values of logic conditions can be changed only during microinstruction execution. Define the execution of a GS on any given sequence of vectors $\Delta_{m1},\ldots,\Delta_{mq}$ beginning from the operational node **Begin**, i.e. the initial operational node Y_b. Let's demonstrate this procedure for the GS of Figure 3.2 and the sequence below containing four vectors $\Delta_1,\ldots,\Delta_4$.

	x_1	x_2	x_3	x_4	x_5	x_6	x_7
$\Delta_1 =$	1	0	0	1	0	0	1
$\Delta_2 =$	1	0	0	0	0	1	1
$\Delta_3 =$	0	1	0	1	1	1	0
$\Delta_4 =$	0	1	0	1	0	0	0

The traverse procedure starts from the initial operational node Y_b and consists of the following steps.

Step 1. Write the initial microinstruction Y_b.

$$Y_b$$

Step 2. Exit from the node **Begin** with the first vector Δ_1. If the node following the initial node is the operational node Y_s, write Y_s right to Y_b and change the vector Δ_1 to Δ_2. If the node following the initial node is the conditional node with the logic condition x_p, find the value x_p in Δ_1. If $x_p=1$, then exit the conditional node with x_p through the output “1” (true) else if $x_p=0$ then exit the conditional node with x_p through the output “0” (false). In the example, the conditional node with x_1 follows the node **Begin** and since $x_1=1$ in Δ_1, this conditional node is exited through the output “1”.

Step 3. If the operational node Y_t, follows the conditional node with the logic condition x_p, write Y_t right to Y_b and change the vector Δ_1 to Δ_2. But if the conditional node with the logic condition x_m, follows the conditional node with the logic condition x_p, find the value x_m in the current vector and leave the node with x_m through the corresponding output, etc. until an operational node is reached. In the example, the operational node Y_2 is reached, so it is written right to Y_b.

$$Y_b\,Y_2$$

The traverse procedure continues with the vector Δ_2 and it arrives to the operational node Y_4, then it enters the operational node Y_6 with the vector Δ_3, and it reaches the final operational node Y_e with the vector Δ_4, thus obtaining the following row of microinstructions.

$$Y_b\,Y_2\,Y_4\,Y_6\,Y_e$$

The microinstruction row obtained is the value of the GS for the given sequence of vectors $\Delta_{m1},...,\Delta_{mq}$. There are only two possible results of the traverse procedure:

- it reaches the node **End**. In this case, the number of microinstructions in the microinstruction row (without Y_b and Y_e) is less than the number of vectors;
- the vectors are exhausted but it has not yet reached the node **End**. In this case, the number of microinstructions in the microinstruction row (without Y_b) is equal to the number of vectors.

3.2.2 Paths in GS

Let the GS have a path from the operational node Y_i (i=b,1,2,…,T) to the operational node Y_j (j=1,…,T,e), passing only through conditional nodes with the logic conditions $x_{i1},...,x_{iR}$.

$$Y_i x_{i1}^{e_{i1}} ... x_{iR}^{e_{iR}} Y_j$$

If there are no conditional nodes in the path (R=0), then two operational nodes follow each other, for example Y_1 and Y_2 in Figure 3.2, and the path turns into the form.

$$Y_i Y_j$$

Suppose that $e_{ir} = 1$ if the path proceeds from the conditional node with x_{ir} through the output "1" and $e_{ir} = 0$ if the path proceeds trough the output "0" with r=1,…,R. The notation $\overline{x}_i$ is used instead of x_i^0 and x_i instead of x_i^1.

To find all the paths from an operational node Y_i, the sub-graph with the node Y_i as the root is traversed. This procedure is demonstrated for the operational node Y_b of the GS of Figure 3.2.

Step 1. In the first path, all logic conditions are asserted (without negation). Find such a path from Y_i, that leaves each conditional node through the output "1". In the example, the first path is $Y_b x_1 x_2 x_3 Y_4$.

Step 2. To find the second path leave the last conditional node in the first path through the output "0" and continue the path leaving the following conditional nodes through the output "1". In the example, the second path is $Y_b x_1 x_2 \overline{x}_3 x_6 x_7 Y_e$. Repeat step 2 for the new path.

In the (q-1)-th path $Y_i \widetilde{x}_{i_1} ... \widetilde{x}_{i_{t-1}} x_{i_t} \overline{x}_{i_{t+1}} ... \overline{x}_{i_{t+r}} Y_{i_{q-1}}$, the variables $x_{i_1},...,x_{i_{t-1}}$ are either asserted or negated and all the variables $x_{i_{t+1}},...,x_{i_{t+r}}$ are negated, so x_{i_t} is the last asserted variable in this path.

Step q. Find the path leaving the conditional node with x_{i_t} through the output "0" and continue this path leaving the following conditional nodes through the output "1". The q-th path is $Y_i \widetilde{x}_{i_1} ... \widetilde{x}_{i_{t-1}} \overline{x}_{i_t} x_{i_{p+1}} ... x_{i_{p+s}} Y_{i_q}$.

The procedure ends when all the variables in the path are negated, i.e. when leaving each conditional node through the output "0". For this example there are the following seven paths: $Y_b x_1 x_2 x_3 Y_4$; $Y_b x_1 x_2 \overline{x}_3 x_6 x_7 Y_e$; $Y_b x_1 x_2 \overline{x}_3 x_6 \overline{x}_7 Y_5$; $Y_b x_1 x_2 \overline{x}_3 \overline{x}_6 Y_4$; $Y_b x_1 \overline{x}_2 x_4 Y_2$; $Y_b x_1 \overline{x}_2 \overline{x}_4 Y_3$; $Y_b \overline{x}_1 Y_1$.

When finding all the paths from the operational node Y_2 in Figure 3.2, a cycle containing only conditional nodes is reached. In this case there is an infinite set of paths between the operational nodes Y_2 and Y_4 and between Y_2 and Y_6. The paths that contain the same variable asserted and negated are ignored, since $x_i\overline{x}_i = 0$. Furthermore, the paths that contain the same variable asserted x_i (or negated $\overline{x}_i$) repeated several times, all but one x_i (or $\overline{x}_i$) are removed. In this case there are just the following three paths: $Y_2x_5x_6Y_6$; $Y_2x_5\overline{x}_6Y_2$; $Y_2\overline{x}_5Y_4$.

3.2.3 Matrix Scheme of Algorithm

Let's call $\alpha_{ij} = x_{i1}^{e_{i1}}...x_{iR}^{e_{iR}}$ a transition function from the operational node Y_i to the operational node Y_j. If two operational nodes follow each other then $\alpha_{ij} = 1$. If there exists more than one path (K paths) between two operational nodes Y_i and Y_j through conditional nodes, then the transition function is

$$\alpha_{ij} = \bigvee_{k=1}^{K} \alpha_{ij}^{k} ,$$

where α_{ij}^{k} is the k-th path.

The **matrix scheme of algorithm** (**MSA**) [Baranov94], which is equivalent to a GS is a square matrix with rows Y_b,Y_1,...,Y_T and columns Y_1,...,Y_T,Y_e. At the intersection of the row Y_i and the column Y_j the transition function α_{ij} is written.

The MSA for the GS of Figure 3.2 is presented in Table 3.1, where it can be seen that the logic sum of all transition functions from an operational node Y_t is always equal to 1.

Table 3.1 – Matrix scheme of algorithm.

	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_e
Y_b	$\overline{x}_1$	$x_1\overline{x}_2x_4$	$x_1\overline{x}_2\overline{x}_4$	$x_1x_2x_3$ $x_1x_2\overline{x}_3\overline{x}_6$	$x_1x_2\overline{x}_3x_6\overline{x}_7$		$x_1x_2\overline{x}_3x_6x_7$
Y_1		1					
Y_2		$x_5\overline{x}_6$		$\overline{x}_5$		x_5x_6	
Y_3				1			
Y_4						1	
Y_5							1
Y_6							1

The MSA is the counterpart of the state transition table of a FSM and it is useful to describe large GSs. Moreover, since the logic sum of all transition functions from an operational node Y_t is always equal to 1, the MSA allows detecting missing transitions in a GS.

3.3 Graph-Schemes of Algorithms and Finite State Machines

Graph-schemes of algorithms can be efficiently used in order to describe the behaviour of control units. They can be used to synthesise a Moore or a Mealy **finite state machine** (**FSM**).

The synthesis process is divided in the two following steps [Baranov94]:

- construct a marked graph-scheme;
- construct the state diagram of the FSM, or the state transition table in the case of a GS with a large number of states and transitions.

3.3.1 Synthesis of a Moore Finite State Machine

In order to mark a GS as a Moore machine it is necessary to perform the following actions [Baranov94]:

- the label a_0 is assigned to the node **Begin** and to the node **End** of the GS;
- the labels $a_1,a_2,\ldots,a_M$ are assigned to the operational nodes in the GS;
- apart from a_0, all the labels in the GS must be unique;
- if any node has already been labelled, it must not be labelled again.

After applying these rules to the GS of Figure 3.2, the GS depicted in Figure 3.3 labelled with the states $a_0,\ldots,a_6$ is obtained.

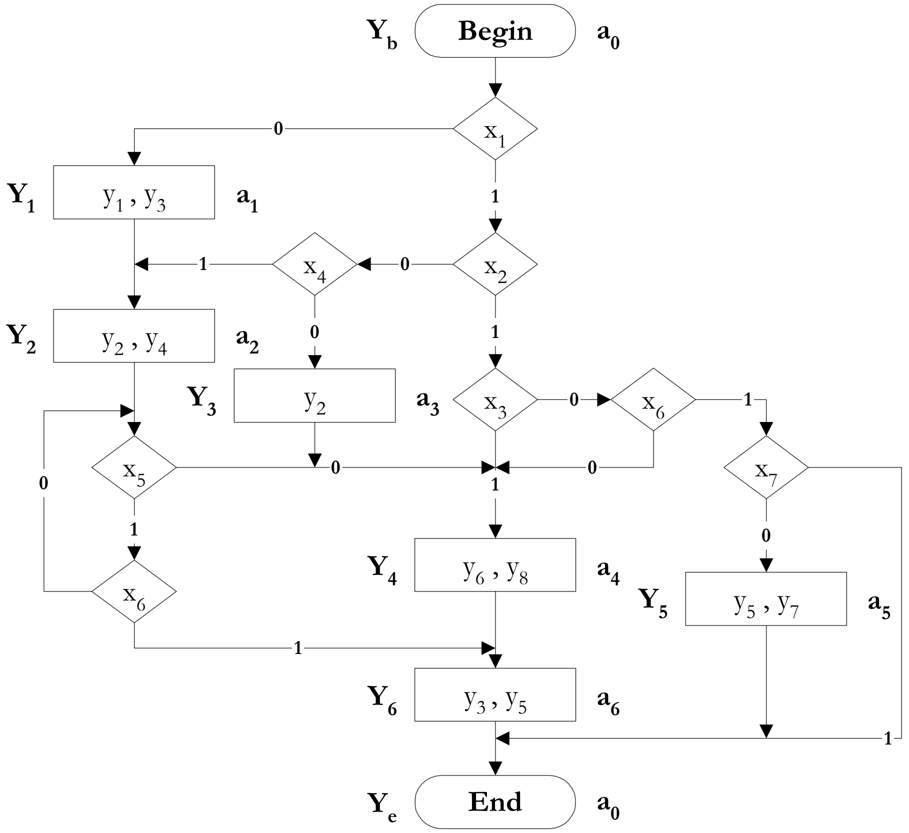


Figure 3.3 – GS marked for Moore synthesis.

In order to build the state diagram of the Moore machine, it is necessary to consider the transition paths $a_m x_{m1}^{e_{m1}} \ldots x_{mR}^{e_{mR}} a_s$ (a_m, $a_s \in \{a_0, \ldots, a_M\}$) in the marked GS, between two operational nodes marked with the states a_m and a_s ($a_m = a_s$ is allowed) and containing R conditional nodes. If there are no conditional nodes in the path (R=0) then the path turns into the form $a_m a_s$.

Defining $X(a_m, a_s) = \bigwedge_{r=1}^{R} x_{mr}^{e_{mr}}$ (if R=0, then $X(a_m,a_s)=1$) then the above path can be rewritten as $a_m X(a_m,a_s) a_s$.

Now let's construct a state diagram with the states $a_0, \ldots, a_M$, where a_0 is the initial state of the FSM. If a_m labels the operational node with the microinstruction Y_t, write the microinstruction Y_t inside the circle with the state a_m ($Y(a_m)=Y_t$). If there is a transition path $a_m X(a_m,a_s) a_s$ in the marked GS, then draw an arrow line from the state a_m to the state a_s, labelled with the transition condition $X(a_m,a_s)$.

When a cycle containing only conditional nodes is reached, there are some paths that loop forever around conditional nodes. Those transition paths $a_m X(a_m,a_m) a_m$ are represented with an arrow line looping around the state a_m, because the microinstruction Y_t inside the state a_m is considered active until a state transition really occurs (see $a_2 x_5 \overline{x}_6 a_2$ in Figure 3.4).

The state diagram of the Moore machine realising the GS of Figure 3.3 is depicted in Figure 3.4, where x_i^1 is used instead of x_i and x_i^0 instead of $\overline{x}_i$. As a result the Moore machine obtained has as many states as the number of labels needed to mark the GS, i.e. the number of operational nodes of the GS plus the initial state.

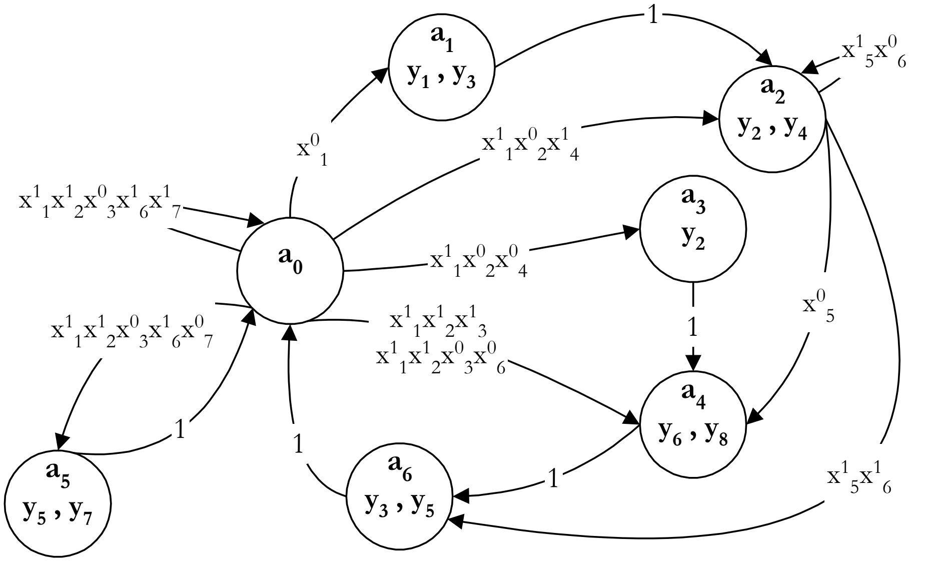


Figure 3.4 – State diagram of the Moore FSM.

3.3.2 Synthesis of a Mealy Finite State Machine

In order to mark a GS as a Mealy machine it is necessary to perform the following actions [Baranov94]:

- the label a_0 is assigned to the node **Begin** and to the node **End** of the GS;
- the labels $a_1,a_2,\ldots,a_M$ are assigned to the inputs which directly follow from output(s) of operational node(s) in the GS;
- apart from a_0, all the labels in the GS must be unique;
- if any input has already been labelled, it must not be labelled again.

After applying these rules to the GS of Figure 3.2, the GS of Figure 3.5 labelled with the states $a_0,\ldots,a_4$ is obtained.

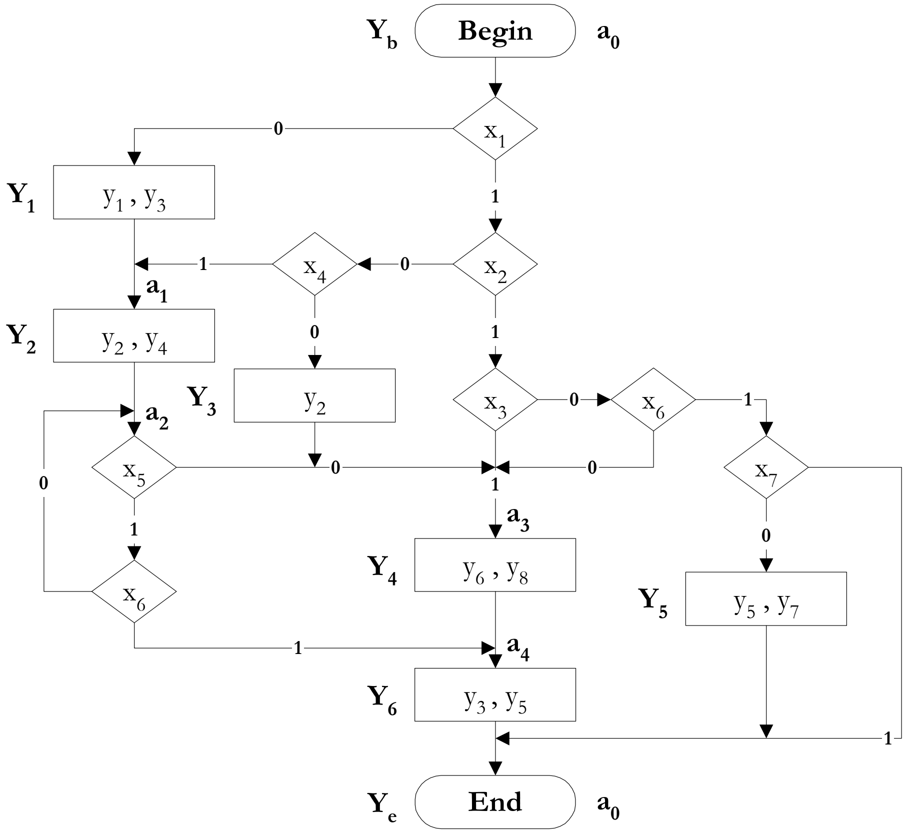


Figure 3.5 – GS marked for Mealy synthesis.

In order to build the state diagram of the Mealy machine, it is necessary to consider the transition paths $a_m x_{m1}^{e_{m1}} \ldots x_{mR}^{e_{mR}} Y_t a_s$ or $a_m x_{m1}^{e_{m1}} \ldots x_{mR}^{e_{mR}} a_s$ (a_m, a_s $\in\{a_0,\ldots,a_M\}$) in the marked GS, between two operational nodes marked with the states a_m and a_s ($a_m=a_s$ is allowed) and containing R conditional nodes. If there are no conditional nodes in the first path (R=0) then the path turns into the form $a_m Y_t a_s$. The first path contains only one operational node at the end of the path, while the second path contains no operational nodes.

In some cases it can only be used the second kind of path, for example $a_0\ x_1x_2\overline{x}_3x_6x_7\ a_0$, but in all other cases it should be used the first kind. Therefore the paths $a_0\ x_1\overline{x}_2x_4\ a_1$, $a_0\ x_1x_2x_3\ a_3$, $a_0\ x_1x_2\overline{x}_3\overline{x}_6\ a_3$, $a_2\ \overline{x}_5\ a_3$ and $a_2\ x_5x_6\ a_4$ are not transition paths, because each one of them can be extended in order to cross one operational node. For example, $a_0\ x_1\overline{x}_2x_4Y_2\ a_2$ instead of $a_0\ x_1\overline{x}_2x_4\ a_1$.

Defining $X(a_m,a_s)=\bigwedge_{r=1}^{R} x_{mr}^{e_{mr}}$ (if R=0, then $X(a_m,a_s)=1$) and $Y(a_m,a_s)=Y_t$ then the above paths can be rewritten as $a_mX(a_m,a_s)Y(a_m,a_s)a_s$ and $a_mX(a_m,a_s)a_s$.

Now let's construct the Mealy state diagram with the states $a_0,\ldots,a_M$, where a_0 is the initial state of the FSM. If there is a transition path $a_mX(a_m,a_s)Y(a_m,a_s)a_s$, then draw an arrow line from the state a_m to the state a_s, labelled with the transition condition $X(a_m,a_s)$ and with the output signals $Y(a_m,a_s)$. If there is a transition path $a_mX(a_m,a_s)a_s$, then draw an arrow line from the state a_m to the state a_s, labelled with the transition condition $X(a_m,a_s)$ and without asserting any output signals.

When a cycle containing only conditional nodes is reached, there are some paths that loop forever around conditional nodes. Those transition paths $a_mX(a_m,a_m)a_m$ are represented with an arrow line looping around the state a_m (see $a_2x_5\overline{x}_6a_2$ in Figure 3.6).

The state diagram of the Mealy machine realising the GS of Figure 3.5 is depicted in Figure 3.6, where x_i^1 is used instead of x_i and x_i^0 instead of $\overline{x}_i$. As a result the Mealy machine obtained has as many states as the number of labels needed to mark the GS, i.e. commonly less than the number of operational nodes.

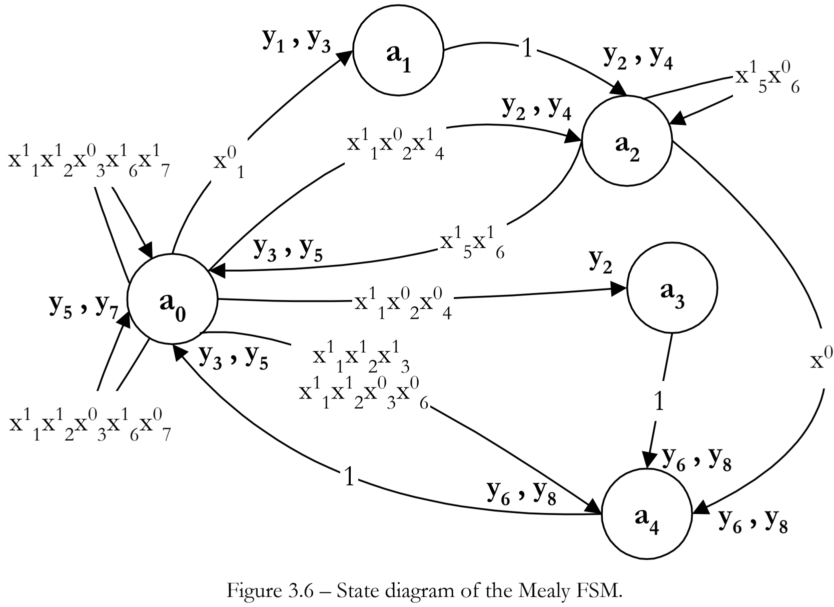


Figure 3.6 – State diagram of the Mealy FSM.

Like it was expected, Figure 3.4 and Figure 3.6 are equivalent state diagrams, if the differences between Moore and Mealy machines are taken in consideration.

3.4 Hierarchical Graph-Schemes

Hierarchical graph-schemes (**HGS**) were introduced in [Sklyarov84] and are graph-schemes of algorithms with the following distinctive features:

- their operational nodes contain either microinstructions from the set $\iota=\{Y_1,Y_2,\ldots,Y_T\}$ or macroinstructions from the set $\varphi=\{Z_1,Z_2,\ldots\}$, or both. Any macroinstruction Z_q, incorporates a subset of macrooperations from the set $Z=\{z_1,\ldots,z_Q\}$. Each macrooperation is described by another HGS of a lower level. For now each macroinstruction is assumed to include just one macrooperation, meaning that only sequential processes are considered;

- their conditional nodes contain just one element from the set $X\cup\Theta$, where $X=\{x_1,\ldots,x_L\}$ is the set of logic conditions, and $\Theta=\{\theta_1,\ldots,\theta_I\}$ is the set of logic functions. Each logic function is calculated by performing a predefined set of sequential steps that are described by a HGS of a lower level.

Consider the set $E=\{\varepsilon_1,\ldots,\varepsilon_V\}$, for which $E=Z\cup\Theta$. Each element $\varepsilon_v\in E$ corresponds to the HGS Γ_v, which describes either an algorithm for performing ε_v (if $\varepsilon_v\in Z$) or an algorithm for calculating ε_v (if $\varepsilon_v\in\Theta$). In both cases an algorithm is being described using a HGS of lower level. Let's assume that $Z(\Gamma_v)$ is the subset of macrooperations and $\Theta(\Gamma_v)$ is the subset of logic functions that belong to the HGS Γ_v. If $Z(\Gamma_v)\cup\Theta(\Gamma_v)=\varnothing$ then the algorithm has only one-level of representation and becomes an ordinary GS.

A **hierarchical algorithm** of a control unit can be specified by a set of HGSs, which describes the main part and all the elements of the set E. The main part is being described by HGS Γ_1 from which the execution of the control algorithm will be started. All other HGSs will be subsequently called either from Γ_1 or from other HGSs that are descendants of Γ_1. Figure 3.7 demonstrates a description of an algorithm with the HGSs Γ_1, Γ_2, Γ_3, Γ_4, Γ_5 and Γ_6, with $Z=\{z_1, z_2, z_3, z_4, z_5\}$ and $\Theta=\{\theta_6\}$.

Some operations in a HGS can be designated as virtual. A macrooperation (a logic function) is called **virtual** if it is not permanently attached during the design of a control unit. Any virtual element (VE), which is either a macrooperation or a logic function, can accept in future different implementations. A VE is described by the appropriate virtual HGS. The virtual HGS can be seen as a variable part of the control algorithm.

A virtual element is called a **pure virtual** element (PVE) if it just has a name and does not have any implementation. A PVE is described by a pure virtual HGS, which is composed of just two nodes following each other: **Begin** and **End** (see the HGS Γ_5 in Figure 3.7). The notions considered above were borrowed from the object-oriented programming [Booch94].

If a HGS has at least one PVE, it is called an incomplete HGS. An incomplete HGS can be executed for the purposes of testing, but ultimately all PVEs have to be replaced with non pure VEs.

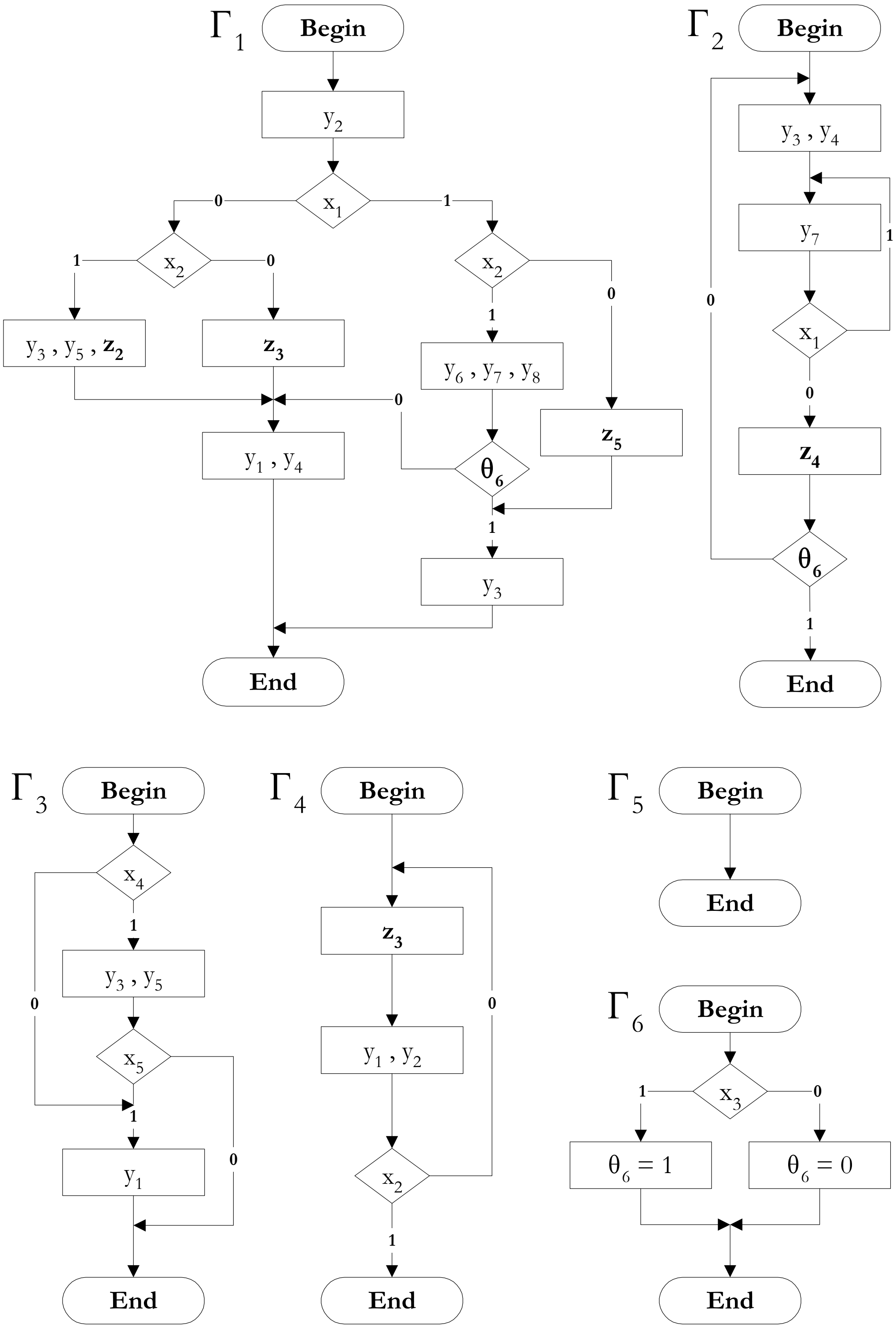


Figure 3.7 – An algorithm described by hierarchical graph-schemes.

3.5 Parallel Hierarchical Graph-Schemes

Parallel hierarchical graph-schemes (**PHGS**) were introduced in [Sklyarov87] and include macroinstructions, which are composed of more than one macrooperation. When a macroinstruction has more than one macrooperation all of them will be executed in parallel. The transition from any active operational node to the next node will be carried out when all its components (a microinstruction and/or a macroinstruction) will be terminated.

Figure 3.8 depicts a description of an algorithm with the PHGSs Γ_1, Γ_2, Γ_3, Γ_4, Γ_5 and Γ_6, with Z={z_1,z_2,z_3,z_4,z_5} and Θ={θ_6}.

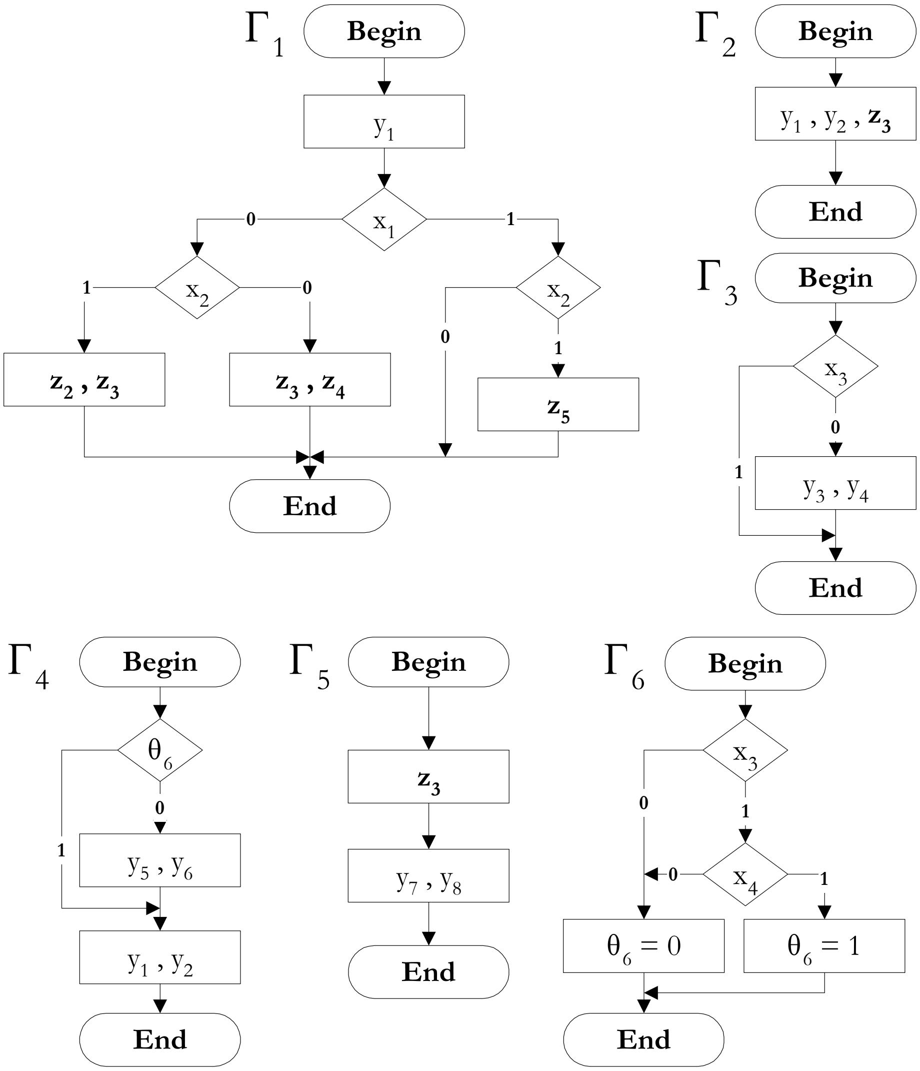


Figure 3.8 – An algorithm described by parallel hierarchical graph-schemes.

3.6 Execution and synchronisation of a HGS/PHGS

The execution of a HGS is based on the GS traversal procedure explained in the paragraph 3.2.1. The execution starts from the node **Begin** of the main part, described by the HGS Γ_1, and it is performed like for an ordinary GS until it arrives to a complex operation ε_v. Each complex operation ε_v such as $\varepsilon_v=z_q\in Z$ and $\varepsilon_v=\theta_i\in\Theta$ ($v\in\{1,...,V\}$) described by a separate HGS Γ_v, initiates the execution of a new HGS in a new hierarchical level. When the execution of the new HGS Γ_v reaches its node **End**, the interrupted HGS will be resumed. The execution continues until the node **End** of the HGS Γ_1 is reached.

The execution of a PHGS is similar to a HGS with the following difference. When a macroinstruction contains more than one macrooperation, each macrooperation invoked in the node initiates the execution of a new PHGS. All PHGSs will run in parallel at the same hierarchical level and the interrupted PHGS will be resumed only after all parallel macrooperations will have been executed. In order to synchronise the parallel execution of macrooperations, the set of PHGSs must be extended and transformed accordingly with the rules defined in [Sklyarov87]. Constructing extended PHGSs depends on the model, Parallel FSM or Parallel Hierarchical FSM, chosen for implementation (see PFSM/PHFSM synthesis).

3.7 Correctness of a HGS/PHGS and Problems with Recursive Calls

Each HGS/PHGS that belongs to a hierarchical algorithm must be expurgated of sub-graphs containing infinite cycles (see Figure 3.9). If the conditional node x_3 will be exited through the "0" output, the HGS execution will enter in the group of four nodes on the right, constituted by two operational nodes (y_3,y_5,y_7 and z_2) and two conditional nodes (x_6 and x_2), and will never reach the node **End**.

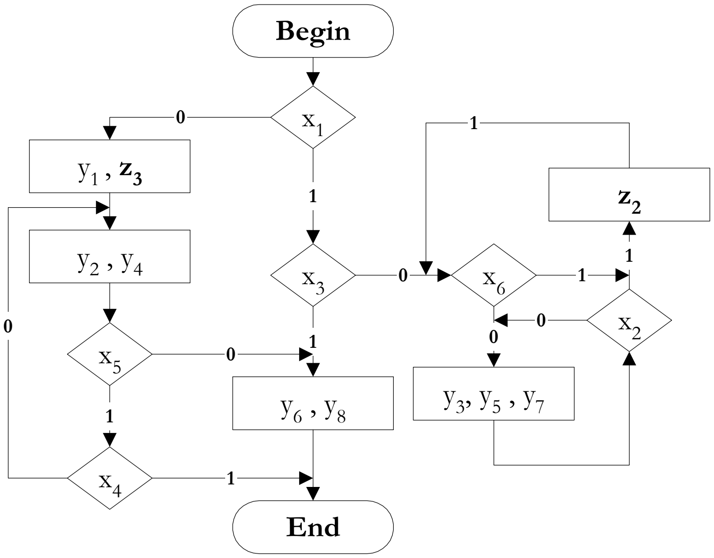


Figure 3.9 – An example of a HGS with an infinite cycle.

In order to prevent infinite recursion[*] in the execution of HGSs they must be checked for correctness using the technique described in [Sklyarov84]. A special graph is constructed, where the nodes represent macrooperations and logic functions, and the arcs between them represent invocations. One arc from the node z_i (θ_i) to the node z_j (θ_j) will mean that the HGS Γ_i invokes the HGS Γ_j in one or more operational nodes. If the graph does not have any group of nodes connected in a loop, the algorithm is free of infinite recursion. Figure 3.10 presents this graph for the hierarchical algorithm of Figure 3.7.

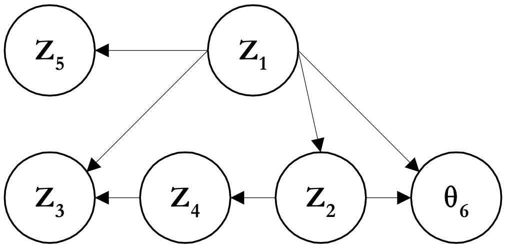


Figure 3.10 – Special graph to detect infinite recursion.

If there are two or more macrooperations or logic functions in a loop (macrooperation looping, logic function looping), it must be ensured that the algorithm has at least one escape to stop the infinite recursion. Figure 3.11 shows an example of macrooperation looping without infinite recursion. The HGS Γ_1 calls the HGS Γ_2 when x_1 is negated ("0"), that calls the HGS Γ_3 that calls back the HGS Γ_1. The loop exists but it can be broken when x_1 is asserted ("1") during the execution of the HGS Γ_1.

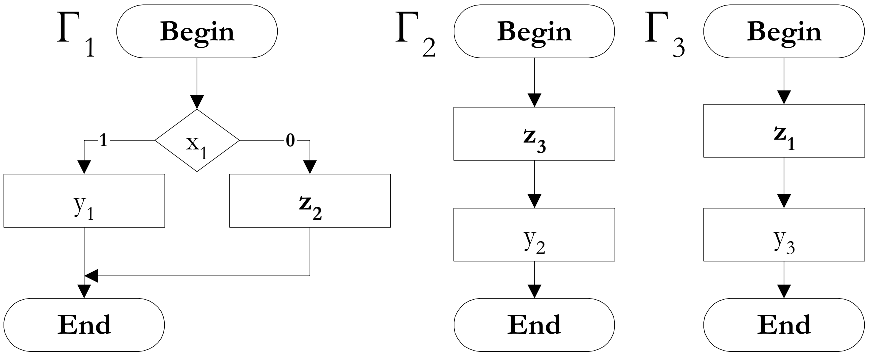


Figure 3.11 – Macrooperation looping without infinite recursion.

If a recursive algorithm (a macrooperation calling itself) is being described, the terminating condition that will stop the recursion and that will let the algorithm to return back from the multiple invocation must be provided, otherwise the algorithm will fall in infinite recursion.

[*] In this case recursion means circular invocation of macrooperations (logic functions).

3.8 An Example

As an example let's consider the fixed-point binary multiplication algorithm with operands represented in sign and module, the left most bit with index 0 being the sign. There are three 16 bits registers, the multiplicand A, the multiplier B and the result C. The multiplication is decomposed in two operations (see Figure 3.12). Macrooperation z_1 implements the multiplication and macrooperation z_2 implements the round and sign calculation.

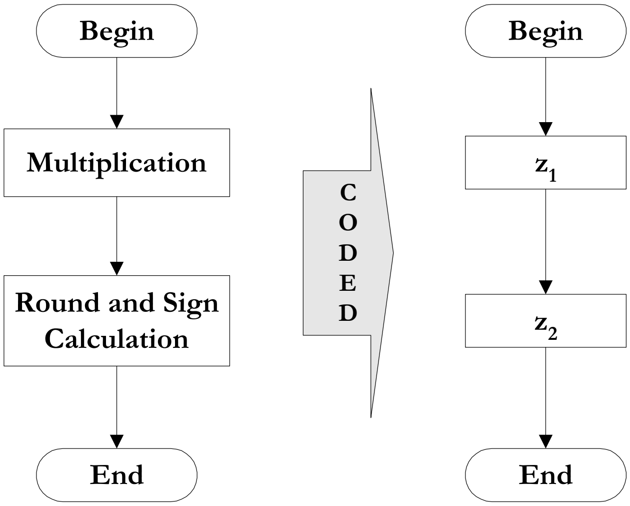


Figure 3.12 – Binary multiplication algorithm.

The multiplication algorithm depicted in Figure 3.13, starts by checking if any of the operands A and B is equal to zero, and if one of them is zero the product C is zero (C:=0) and the multiplication has finished. Otherwise it starts by clearing the result (C:=0) and setting the counter to fifteen (count:=1111_2). If the least significant bit of the multiplier is one (B(15)=1) the multiplicand is added to the result (C:=C+A(1:15)). Then the multiplier and the result are shifted right one bit (B(1:15):=SHR(B(1:15)), C:=SHR(C)), with the shifted out bit of the result written into the most significant bit of the multiplier (B(1):=C(15)) and the counter is decreased one unit (count:=count-1). The multiplication is repeated until all fifteen bits of the multiplier are processed, i. e. until the counter reaches zero (count=0).

The round and sign calculation algorithm depicted in Figure 3.14 rounds the result (C:=C+1) if the most significant bit of the multiplier, i.e. the last shifted out bit of the result, is one (B(1)=1). If the two operands have the same sign (A(0)=B(0)) the result is positive, otherwise it is negative (C(0):=1).

After coding the instructions, there are the following six logic conditions:
x_1 : A=0; x_2 : B=0; x_3 : B(15)=1; x_4 : count=0; x_5 : B(1)=1; x_6 : A(0)=B(0),

and the following nine microoperations:
y_1 : C:=0; y_2 : count:=1111_2; y_3 : C:=C+A(1:15); y_4 : B(1:15):=SHR(B(1:15));
y_5 : C:=SHR(C); y_6 : B(1):=C(15); y_7 : count:=count-1; y_8 : C:=C+1; y_9 : C(0):=1.

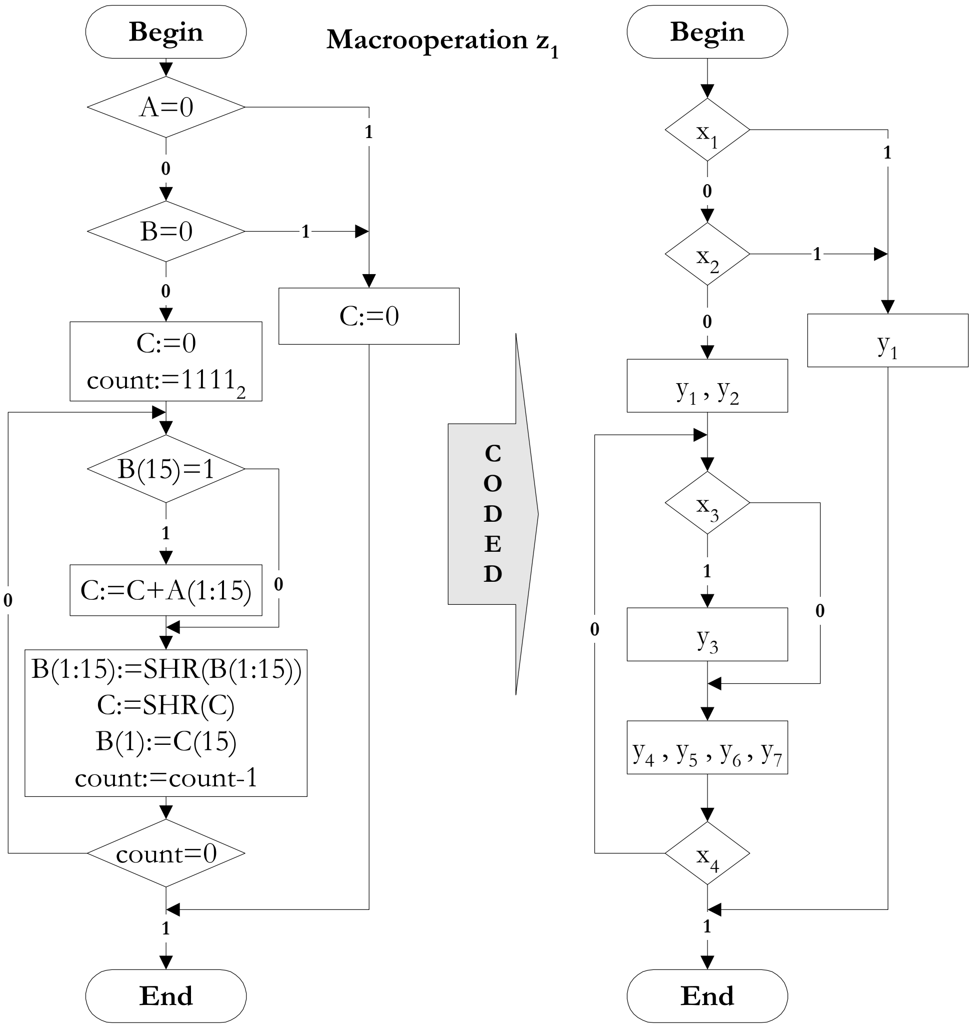


Figure 3.13 – Macrooperation z_1 implementation and codification.

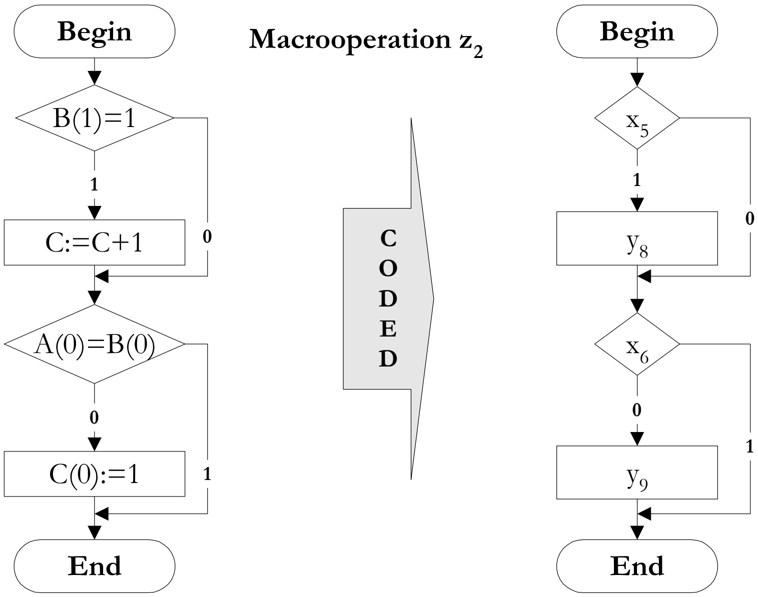


Figure 3.14 – Macrooperation z_2 implementation and codification.

The hierarchical implementation for a small example like this does not offer any advantages. In this case the two macrooperations should be merged in one graph-scheme of algorithm and should be implemented as an ordinary FSM, i.e. the hierarchy should be flattened (see Figure 3.15).

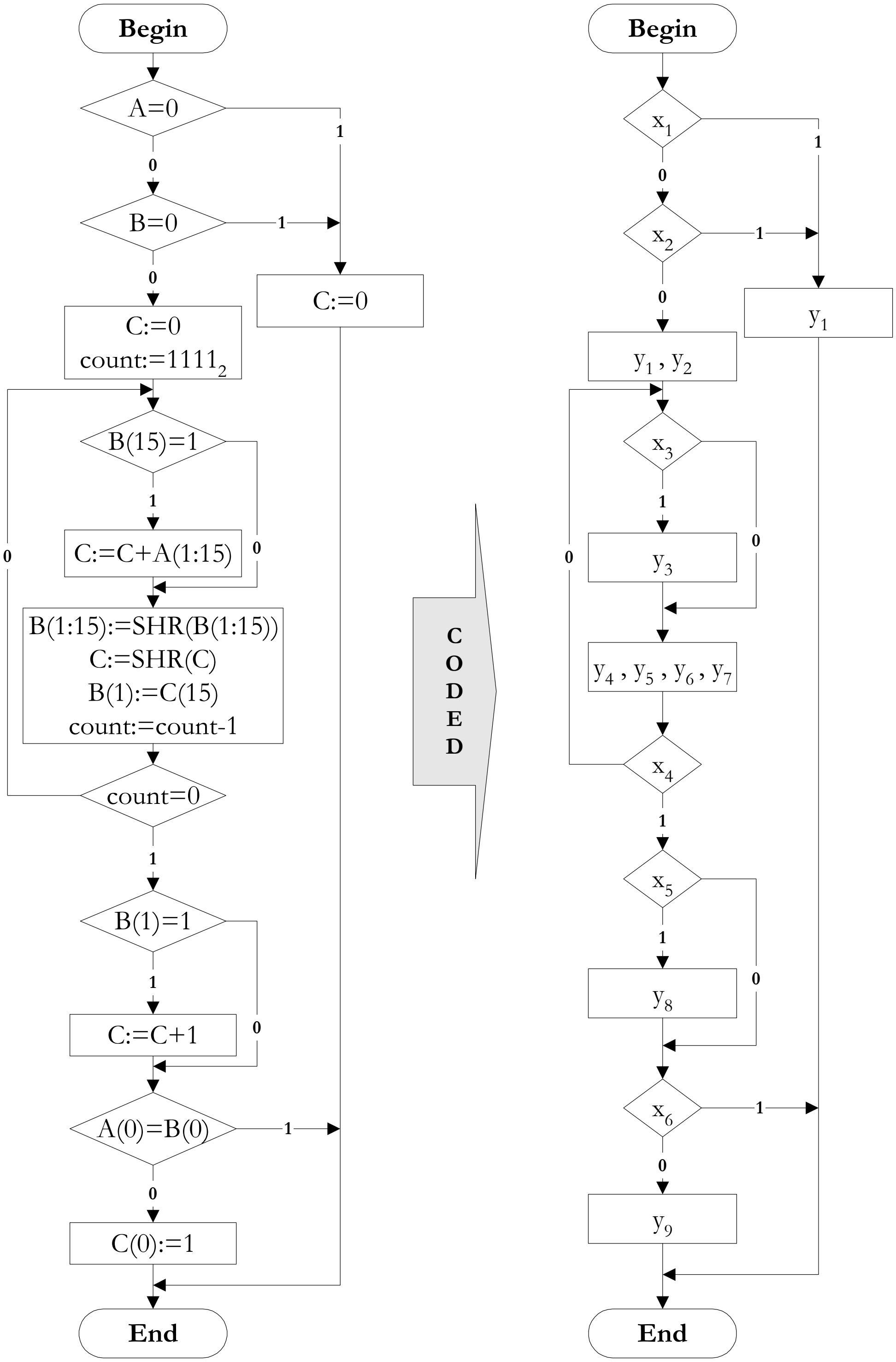


Figure 3.15 – Non-hierarchical implementation and codification of the binary multiplication algorithm.

However, a hierarchical decomposition of an algorithm allows the designer to develop any complex control algorithm part by part concentrating his efforts on different levels of abstraction. Moreover, the macrooperations described can be separately tested and can be used to implement other algorithms developed in the future. For example, the round and sign calculation (macrooperation z_2) can also be used in the algorithm that implements the binary division.

3.9 C++ Simulation of Hierarchical Graph-Schemes

3.9.1 Introduction

A hierarchical algorithm can be expressed by a set of HGSs, describing all its components (macrooperations and logic functions). In order to ensure the correctness of the description, each HGS must be formally correct. The HGS must not have unreachable nodes nor nodes constituting infinite cycles, i. e. every node must be located on at least one of the paths which go from the node **Begin** to the node **End**. It is always possible to check manually the consistency of a small HGS containing a few nodes, but if the HGS is big containing let's say 100 nodes, it is probably very easy to miss some errors. So it is useful to consider **automatic checking** of HGSs.

When executing an algorithm infinite recursion must be prevented, and therefore the algorithm has to be checked for macrooperations (logic functions) invoked in a loop. But, it is possible to have an algorithm with macrooperation looping without having infinite recursion, like it was shown in Figure 3.11. So, in order to ensure the correctness of the algorithm **simulation** is used.

The tool **SIMULHGS** described in this paragraph provides such facilities. The tool was developed in C++ using an object-oriented methodology and its code is presented in Appendix B.

3.9.2 Description of the Class System

Figure 3.16 depicts the class system diagram using the Booch notation [Booch94]. In simple terms a hierarchical algorithm can be considered as a set of nodes grouped in entities called HGSs. So, the base class of the tool is the class **Node**. An object of class Node has a name **nname**, a type **ntype** and a state **nstate**. Certain member methods need to traverse the HGS recursively and to avoid infinite looping the traversed nodes are marked. For that purpose the data member **nmark** is used. The data member **nauto** is used to distinguish between the nodes that are marked automatically (by the marking methods) and those that are marked manually (at user request).

There are seven types of nodes:

- node **BEGIN**;
- node **END**;
- node **ASSIGN** that assigns the return value of a logic function;
- node **MICROOP** that contains a subset of microoperations;
- node **MACROOP** that contains one macrooperation and that can also contain a subset of microoperations;
- node **CONDITION** that contains one input condition;
- node **FUNCTION** that contains one logic function.

These node types are grouped in two classes: **operational nodes** (the first five) and **conditional nodes** (the last two). So, two classes were derived from the base class **Node**. Class **Onode** adds to the base class a pointer to the next node, while class **Cnode** adds two pointers, next node when true and next node when false.

A hierarchical graph-scheme is a directed connected graph composed of a finite set of nodes and the class **Graphscheme** represents it. An object of the class Graphscheme has a name **gsname**, a type **gstype** and the number of nodes **gsnnodes**. There are two kinds of hierarchical graph-schemes: **LFUNCGS** (logic function) and **MACROGS** (macrooperation). They must be distinguished because they have different properties concerning the type of nodes they can have.

In some methods, all the nodes of a HGS need to be processed in a repetitive for statement, so it is worth to have an array of pointers to the nodes (**gslist**) instead of just one pointer to the node **Begin**. In some synthesis steps, a HGS must know if it is the main HGS of the algorithm or not. That information is stored on **gsmain**. The number of states used to mark each HGS, in the case of the HFSM model 3, is stored in **gsnstates**.

The class **Hgraphscheme** represents a hierarchical algorithm. An object of this class has a name **hgsname**, the number of elements of the set of HGSs **hgsngs** and an array of pointers to the HGSs **hgslist**.

The remaining data members store information generated by some methods. If an algorithm is checked and is correct **hgscheck** is equal to 1. After simulating an algorithm the deepest level of hierarchy reached is stored in **hgsdeep**. The string **hgssyn** indicates if it is a Moore or a Mealy HFSM that is being synthesised. If the HFSM model 2 is chosen, one state transition table for the set of HGSs (**merge tables**), then **hgsmark** is equal to 1. Otherwise, in the case of the HFSM model 3, one state transition table per HGS (**split tables**), then **hgsmark** is equal to 0. The number of states needed to mark the algorithm, in the case of the HFSM model 2, is stored in **hgsnstates**.

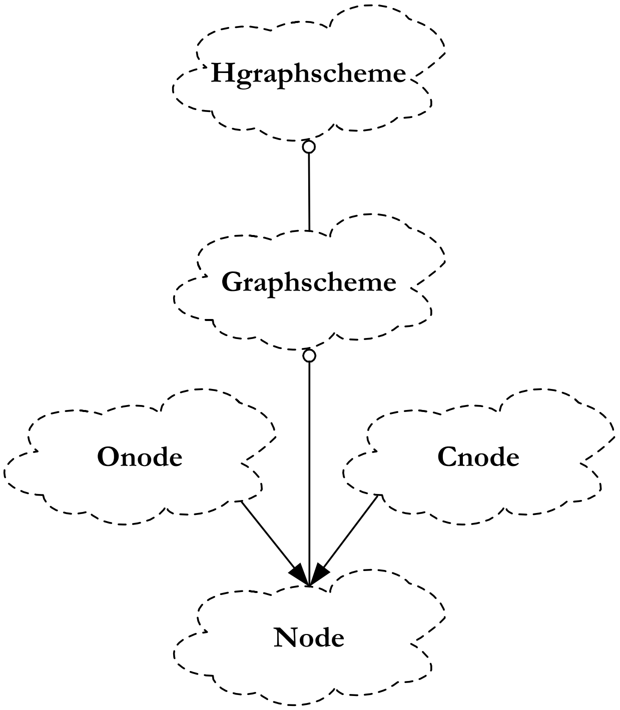


Figure 3.16 – Class system diagram of the tool SIMULHGS.

3.9.3 Acquisition and Construction of a Hierarchical Algorithm

SIMULHGS constructs a hierarchical algorithm from a set of text files. The text files can be prepared manually with a text editor or generated automatically in a graphical editor developed for that purpose and described in [ParCra98]. This graphical editor allows the creation of separated HGSs as well as algorithms composed with already existing HGSs and newly developed HGSs. The editor generates text descriptions of correct HGSs and the textual decomposition of algorithms.

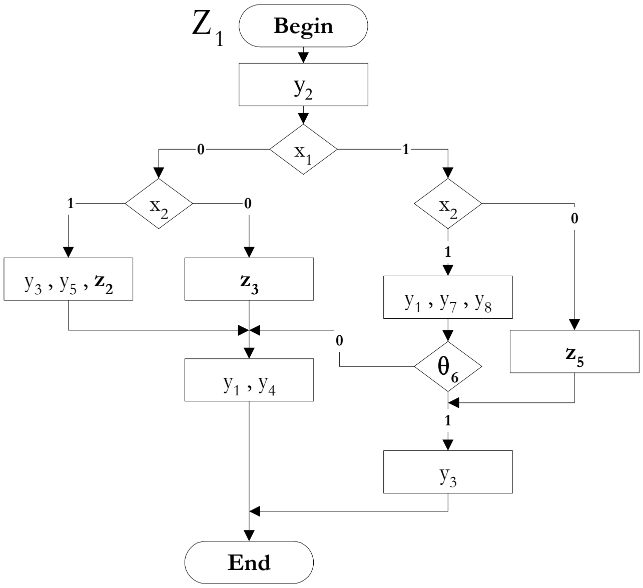
Each HGS is constructed from a text file with the following format (see Figure 3.17 and Figure 3.18). The first line is the macrooperation (logic function) name. The remaining lines have the description of the nodes, in fields separated by the character space, with the following format:

- the first field of the line is the character O for an operational node or C for a conditional node;
- the second field contains the node name and defines its type:
  - BEGIN node type, BEGIN;
  - END node type, END;
  - MICROOP node type, a set of microoperations starting with the character y and without any z character (for example y1,y7,y8);
  - MACROOP node type, a macrooperation starting with the character z (for example z5) or a set of microoperations followed by a macrooperation (for example y3,y5,z2);

  - CONDITION node type, an input condition starting with the character x (for example x2);
  - FUNCTION node type, a logic function starting with the character f (for example f6);
  - ASSIGN node type, a string fi=value where value is 0 or 1 (for example f6=0);

- the third field contains the number of the line where the next node of an operational node is described (this field does not exist for the node END). Or in the case of a conditional node the number of the line where the next node when the condition is true is described;
- the fourth field only exists in the case of a conditional node and contains the number of the line where the next node when the condition is false is described.

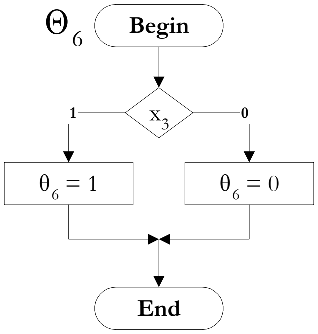
Figure 3.17 and Figure 3.18 present respectively text descriptions of the macrooperation Z_1 and of the logic function θ_6.



```
Z1.txt

Z1
O BEGIN 2
O y2 3
C x1 8 4
C x2 5 6
O y3,y5,z2 7
O z3 7
O y1,y4 13
C x2 9 10
O y1,y7,y8 11
O z5 12
C f6 12 7
O y3 13
O END
```

Figure 3.17 – Macrooperation Z_1 and its text description.



```
F6.txt

F6
O BEGIN 2
C x3 3 4
O f6=1 5
O f6=0 5
O END
```

Figure 3.18 – Logic function Θ_6 and its text description.

The algorithm decomposition is described in a text file where, the first line is the algorithm name and the remaining lines list the HGSs names being the first HGS, second line of the file, the main HGS. The text description of the algorithm of Figure 3.7 will look like the following text.

```
THESIS HGS
Z1
Z2
Z3
Z4
Z5
F6
```

When **SIMULHGS** starts, the user must supply the filename where the set of HGSs is described and the **Hgraphscheme** constructor is invoked. If the file does not exist the program will terminate, because the set of HGSs cannot be constructed. If the file is successfully opened the name of the algorithm is stored in **hgsname**. Then for each name listed in the following non-empty lines a filename with the **txt** extension is created, the **Graphscheme** constructor is invoked and the HGS pointer is stored in the array **hgslist**. When the end of file is detected, the number of HGSs is stored in **hgsngs** and the file is closed.

The first parameter of the **Graphscheme** constructor is the filename where the HGS is described. The second parameter **main** is 1 for the main HGS and 0 for the remaining HGSs. By default this parameter is 0. If the file is successfully opened the name of the HGS is stored in the string **gsname**, otherwise an empty HGS will be constructed. If the name starts with the character Z it describes a macrooperation **MACROGS**. If the name starts with the character F it describes a logic function **LFUNCGS**.

Then for each non-empty line of the file the type of node is detected and the respective constructor **Onode** or **Cnode** is invoked. After all the nodes have been constructed and their pointers stored in the array **gslist**, the nodes must be linked. The information about the next nodes, third and fourth field of the line, is used by **Setnext()** to link the nodes. The number of nodes is stored in **gsnnodes**, the information about the main HGS passed through the second parameter is stored in **gsmain** and the file is closed.

The **Onode** and **Cnode** constructors use the **Node** base class constructors. Because the nodes **Begin** and **End** have their types automatically defined by their names, there are two constructors for operational nodes. The first, used for the nodes **Begin** and **End**, has only one parameter **name**, while the second, used for the remaining operational nodes, has two parameters **name** and **type**.

3.9.4 Checking a Hierarchical Algorithm

The most important feature of **SIMULHGS** is to check if a set of HGSs is correctly described and can be processed. This is performed by the method **Checkhgs()**. If the algorithm is really a hierarchical algorithm, i.e. has more than one HGS, and all HGSs are correct the method returns the value 1, otherwise it returns the value 0. The check value (1/0) is stored in **hgscheck**. In order to avoid abnormal execution, all **Hgraphscheme** methods must inquire **hgscheck** before start executing.

Checkgs() checks a HGS in three steps. The first two steps are only needed for text descriptions generated manually, because they can contain elementary errors that can be easily made in a text editor. On the other hand, the graphical editor does not allow the creation of HGSs with any of the error situations checked in the first two verification steps. Moreover, it eliminates redundant nodes when generating the HGS text description (see [ParCra98] for more details). Let's describe in detail the three verification steps.

In step 1 **Primarycheckgs()** will check the overall consistency of the HGS. Let's see with more detail what means overall consistency. The following situations must be verified. The HGS must have one and only one node **Begin**. The HGS must have one and only one node **End**. A HGS describing a macrooperation cannot have nodes of type **ASSIGN**, on the other hand a HGS describing a logic function can only have operational nodes of type **ASSIGN**. The main HGS must have at least one macrooperation or logic function, otherwise it is not a hierarchical description. If a node invokes a macrooperation or a logic function **Searchgraphhgs()** will see if the respective HGS makes part of the algorithm. If a macrooperation is only composed with the nodes **Begin** and **End**, a warning message of pure virtual macrooperation will be printed on the screen.

If the first step is successful, in step 2 **Markgs()** traverses recursively the HGS and marks all the nodes that are reachable from the node **Begin**. Then **Checknode()** tests all the nodes reporting unreachable and dummy nodes. Unreachable node means a node with its input not connected to any node. Dummy node means a node that is pointing to the node **Begin**, or that is pointing to itself. Nevertheless, a conditional node can have one of the outputs, but not the two, pointing to itself. It is also detected if a node of type **ASSIGN** holds a value different from 0 and 1 and if there are two conditional nodes with the same input condition following each other. If a conditional node has both outputs pointing to the same node, different from the node **Begin**, a warning message is printed, because it is a useless node.

If the second step does not detect any errors, in step 3 **Loopcheckgs()** will look for nodes in infinite cycles. Since all nodes are reachable from the node **Begin**, it is necessary to ensure that starting from each node it is possible to reach the node **End**. **Loopcheckgs()** calls, for every node in the HGS, **Loopnode()** that traverses recursively the HGS starting from the specified node. It stops when it reaches a node already traversed, meaning that it entered a loop, or when it reaches the node **End**. In the latter case returns a flag set to 1. This flag is enquired for each call of **Loopnode()** and if it is not equal to 1 the node is reported to be in an infinite cycle. If there is at least one node in this situation a check failure will be returned. Since it is only necessary to ensure that there is at least one path, if the node **End** is reachable when exiting a conditional node through the true output, the false output is not explored in order to speed up the **Loopnode()** method.

If all three steps are successful, the method **Checkgs()** prints the message that the HGS is correct and returns the check value 1, otherwise it prints the message that the HGS is incorrect and returns the check value 0.

3.9.5 Running a Hierarchical Algorithm

To run a set of HGSs allows to verify if it is free of infinite recursion and to evaluate the deepest hierarchy level reached by the algorithm, in order to define the stack memory size. After ensuring that the set of HGSs is already checked **Runhgs()** starts the execution of the main HGS. A variable will keep track of the hierarchy level reached at any moment in the simulation.

The **Rungs()** method starts the HGS execution by invoking the recursive **Runnode()** to the node **Begin**, whose pointer is obtained through **Begings()**.

Rungs() returns an integer value. This value is meaningless when a HGS describes a macrooperation, but when it describes a logic function this value is returned to the conditional node where the logic function was invoked in order to decide the path to follow. On the other hand, when the execution reaches a conditional node containing an input condition the user must supply a bit value to decide the path to follow.

Every time that a node holding a complex operation is reached, such as a macrooperation or a logic function, the **Searchgraphhgs()** method is invoked to return the pointer to the respective HGS and the execution of the complex operation will start.

Every time that the node **Begin** is detected the hierarchy level is incremented. If it is the deepest level reached so far, it will be stored in **hgsdeep** by the method **Deeplevelhgs()**. When the execution reaches the node **End**, the hierarchy level is decremented, and the execution of the previous hierarchical level HGS is resumed. The algorithm execution will stop when the node **End** of the main HGS is reached.

This execution of a hierarchical algorithm is a functional simulation and in order to be useful the designer must explore all possible paths of the algorithm to ensure that infinite recursion situations are not present and to evaluate correctly the deepest hierarchical level of the algorithm.

3.10 Conclusions

HGSs and PHGSs enable the development of any complex control algorithm part by part in a top-down manner, which can be viewed at various levels of abstraction, one level at a time. They provide for a clear separation of the control unit functionality from its implementation. They embody multilevel representations of control algorithms through the use of macro blocks such as macrooperations and logic functions, and consequently they support hierarchical and parallel specifications. They also allow for the use of virtual operations for the sake of testing an algorithm and the use of logic function nodes allows a more abstract decomposition of the algorithm. Moreover, since HGSs can be seen as relatively autonomous components they can be separately tested and can be reused to implement other algorithms to be developed in the future.

However, HGSs and PHGSs have the following constraints in the transitions between hierarchical levels. When a new component is invoked, the state transition is always done to its first state. When the component finishes executing it always returns to the interrupted state of the previous hierarchical level. The latter cannot really be considered a constraint since in most algorithmic specifications the desired functionality is to resume the execution of the algorithm in the state after the hierarchical invocation. But the former can be considered an annoying constraint in situations where the last part of an already existing macrooperation must be performed without the need to perform the first part.

When developing an algorithm with HGSs/PHGSs, they must be checked for correctness, eliminating macrooperation and logic function looping, in order to prevent infinite recursion and each HGS/PHGS must be expurgated of sub-graphs containing infinite cycles.

The tool **SIMULHGS** used in conjunction with a graphical editor of HGSs, allows the specification of a hierarchical algorithm with a set of HGSs. It provides for the automatic checking of the algorithm and it ensures that it is in fact a correct and consistent hierarchical algorithm. **SIMULHGS** also allows the simulation of the algorithm in order to detect macrooperations (logic functions) invoked in a loop and to evaluate the deepest level in the hierarchy reached by the hierarchical algorithm.

4 HIERARCHICAL FINITE STATE MACHINES

Summary

A hierarchical algorithm described by a set of HGSs can be efficiently implemented with a **hierarchical finite state machine** (**HFSM**) with **stack memory**. This chapter starts by explaining the first model of a HFSM with stack memory proposed in [Sklyarov84]. Then it proposes two new models that can provide such new facilities as **flexibility**, **extensibility** and **reusability**. The concept of a **virtual HFSM** is also presented.

In order to allow the execution of macrooperations in parallel described by PHGSs, a **parallel finite state machine** (**PFSM**) was proposed in [Sklyarov87]. This model has however some limitations and to overcome them a new model of a **parallel hierarchical finite state machine** (**PHFSM**) that combines hierarchy and parallelism is proposed.

Finally, the synchronisation mechanisms for all FSM models are fully described.

4.1 Introduction

A **finite state machine** (**FSM**) (see Figure 4.1a) like it was presented in the second chapter can be formally described as follows:

$$A = \delta[A,X];$$
$$Y_{Mealy} = \lambda_{Mealy}[A,X];$$
$$Y_{Moore} = \lambda_{Moore}[A].$$

Where $A=\{a_0,a_1,\ldots,a_M\}$ is a finite set of **states** being a_0 the FSM initial state, $X=\{x_1,\ldots,x_L\}$ is a finite set of **inputs**, $Y=\{y_1,\ldots,y_N\}$ is a finite set of **outputs**, δ is the **transition function** or the **next state function**, which determines the next state from the present state and the inputs, and λ is the **output function**. Moore outputs are dependent on only the present state, while Mealy outputs are dependent on both the present state and the external inputs.

On the other hand, a **hierarchical finite state machine** (**HFSM**) (see Figure 4.1b) can be formally described as follows [SklFer98]:

$$A = \begin{cases} \delta\left[A,X\right], \text{if } \varepsilon(t) = \varnothing; \\ \xi\left[A,X\right], \text{if } \varepsilon(t) \neq \varnothing; \end{cases}$$
$$Y_{Mealy} = \lambda_{Mealy}[A,X];$$
$$Y_{Moore} = \lambda_{Moore}[A].$$

It is like an ordinary FSM but with two distinctive blocks, each one implementing a different transition function. The transition function between states of the same hierarchical level δ is provided by the **Combinational Scheme**, while the transition function between states of different hierarchical levels ξ is provided by the **Hierarchical Scheme**.

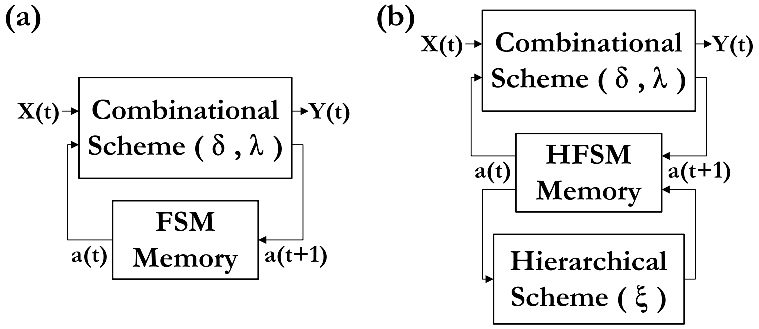

(a)
X(t)
Combinational Scheme (δ , λ)
Y(t)
FSM Memory
a(t)
a(t+1)
(b)
X(t)
Combinational Scheme (δ , λ)
Y(t)
HFSM Memory
a(t)
a(t+1)
Hierarchical Scheme (ξ)

Figure 4.1 – (a) FSM block diagram. (b) HFSM block diagram.

It must be kept in mind that the HFSMs are used to describe control units. Since the control unit is connected with a datapath it must have input and output registers that fix respectively the external inputs received from the datapath and the external outputs generated at the control unit. Moreover, in some models extra storage elements are needed to hold intermediate outputs or to synchronise internal operations. Therefore, as against of an ordinary FSM the **Combinational Scheme** of the HFSM is not a pure combinatorial block.

4.2 FSM with Stack Memory (managing hierarchy)

Let's introduce the graph G_h (see Figure 4.2), which shows the hierarchical levels of the algorithm depicted in Figure 3.7 and that can be considered as a tree. The root $\mathbf{z_1}$ of the tree corresponds to the main HGS Γ_1 of level 1. The leaves of the tree correspond to HGSs, which do not contain elements from the set E (graph-schemes of algorithms).

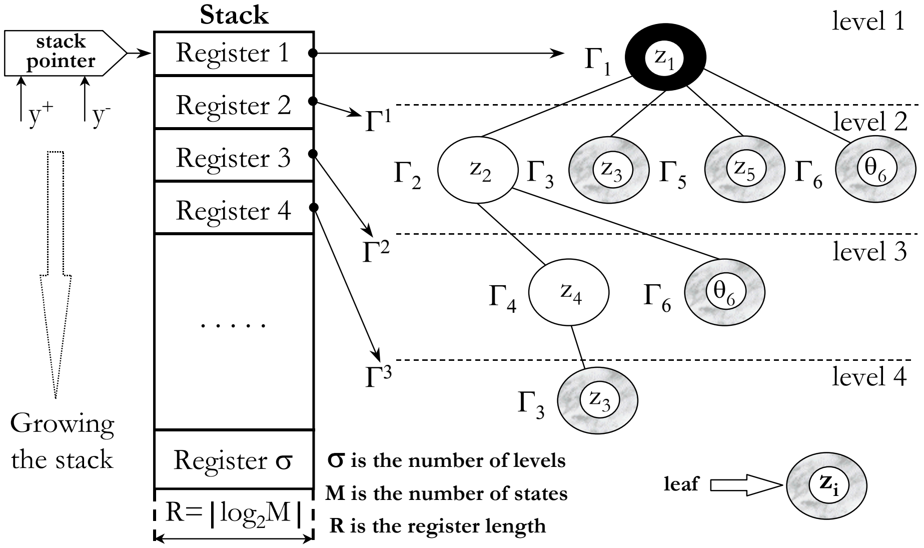


Figure 4.2 – Graph G_h showing hierarchical levels.

Consider the following sequence of HGSs: Γ_1 (level 1) ⇒ Γ^1 (HGSs of level 1) ⇒ Γ^2 (HGSs of level 2) ⇒... , where Γ^1 is the set of HGSs that are used to describe elements from the set $Z(\Gamma_1)\cup\Theta(\Gamma_1)$, Γ^2 is the set of HGSs that are used to describe elements from the sets $\bigcup_{\gamma\in\Gamma^1} Z(\gamma)$ and $\bigcup_{\gamma\in\Gamma^1}\Theta(\gamma)$.

The same way can be used to determine other sets (Γ^3, Γ^4, etc.). The problem is the following: how to perform switching to the various levels? This problem can be efficiently solved using a **hierarchical finite state machine** (**HFSM**) with **stack memory** (see Figure 4.3). This model was proposed in [Sklyarov84] and has been explored in [SklRoc96A, SklRoc96B].

The top of the stack is the register, which is used as the FSM memory for the HGS Γ_1. Suppose it is necessary to perform an algorithm for a component ε_v of Γ_1 and $\varepsilon_v\in Z(\Gamma_1)\cup\Theta(\Gamma_1)$. In such case the stack pointer is **incremented** by activating the output $\mathbf{y^+}$ and set the new register, that is now located on the new top of the stack, with the first state of the HGS Γ_v. As a result the old top of the stack keeps the **interrupted state** of the HGS Γ_1, and the new top of the stack holds the **entry state** of the HGS Γ_v. The same sequence of steps can be applied to other levels.

Therefore the total size of the stack **σ**, i.e. the number of registers, must not be less than the number of various levels for the graph G_h. When the execution of a HGS of a level other than the level one, is being terminated the opposite sequence of steps must be performed in order to return back to the interrupted HGS. In this case the stack pointer is **decremented** by activating the output $\mathbf{y^-}$.

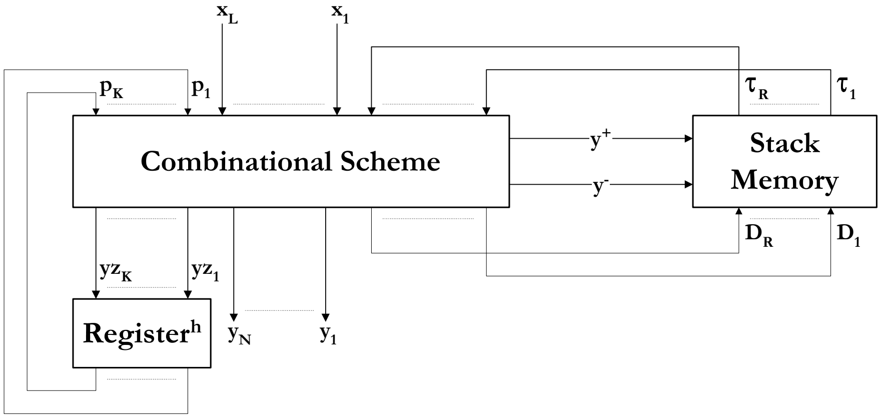


Figure 4.3 – Hierarchical finite state machine structure (model 1).

The code in the **Register^h** indicates which HGS must be executed next. Let's assign to elements of the set $E=\{\varepsilon_1,\ldots,\varepsilon_V\}$ (set of HGSs specifying the algorithm) **binary codes** with the length $K\geq|\log_2(V+1)|$ and the code containing all zeros (00...0) will not be used. Designate $K(\varepsilon_v)=\{e_{vK}...e_{v1}\}$ as the code of the ε_v, where $e_{vk}\in\{0,1,-\}$, k=1,...,K, and "-" is the don't care value of a bit. $K(\varepsilon_v)$ can be considered as the code of the HGS describing the element ε_v.

The hierarchical FSM operates like an ordinary FSM if there are no transitions from one HGS to another HGS. If it is necessary to call a new HGS Γ_v in order to perform either a **macrooperation** or a **logic function** the following sequence of actions will be carried out (see Figure 4.3):

1. the code $K(\varepsilon_v)$ is stored into the **Register^h** through the inputs $yz_K,...,yz_1$;
2. the stack pointer is incremented ($y^+=1$). As a result a new register RG_{new} of the **Stack Memory** will be selected as the current register of the HFSM. The previous register RG_{new-1} keeps the state of the HFSM when it was interrupted (in which the control to the HGS Γ_v was passed). The new register RG_{new} will be automatically set to zero (00...0);
3. the code $K(\varepsilon_v)$ stored into **Register^h** when presented at the extra inputs $p_K,...,p_1$ of the **Combinational Scheme** and in conjunction with the state binary code set to zero (00…0) presented in the inputs $\tau_R,...,\tau_1$, causes a transition to the initial state of the HGS Γ_v. As a result Γ_v will be responsible for the control from this point until it terminates;
4. after the termination of the HGS Γ_v it is necessary to decrement the stack pointer ($y^-=1$) in order to return back to the interrupted state. As a result the control will be passed to the state in which the HGS Γ_v has been called.

If a HGS is used to calculate a **logic function**, the value to be calculated will be stored in any bit of **Register^h**. This is allowed because **Register^h** is not being used while a HGS is being terminated. The interrupted HGS will later test the value stored in order to perform the respective transition.

Based on this first model, the model depicted in Figure 4.4 has been proposed in [RocSkl97A, RocSkl97B], where **Register^h** has been replaced with a **Code Converter** block.

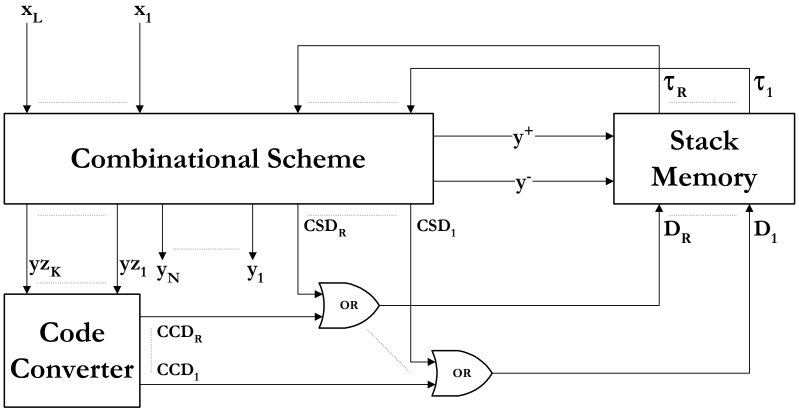


Figure 4.4 – Hierarchical finite state machine structure (model 2).

This model works as follows:

- in all states that do not invoke a macrooperation or a logic function, the outputs CCD$_R$,…,CCD$_1$ of the **Code Converter** are set to 0, because its inputs yz$_K$,...,yz$_1$ are set to 0, and the next state is provided by the outputs CSD$_R$,…,CSD$_1$ of the **Combinational Scheme** like in an ordinary (non-hierarchical) FSM;

- if it is necessary to call a new HGS Γ_v in order to perform either a **macrooperation** or a **logic function** the following sequence of actions will be carried out (see Figure 4.4):
  1. the code K(ε_v) of a state with ε_v is presented at the inputs yz$_K$,...,yz$_1$ of the **Code Converter**;

  2. the stack pointer is incremented (y^+=1). The register which is the new top of the stack is clean (set to zero) if it is the first time that this hierarchy level is reached or holds the code of the returning state of the previous HGS that has run in this hierarchy level;

  3. in both cases the outputs CSD$_R$,…,CSD$_1$ of the **Combinational Scheme** are set to 0. The code K(ε_v) is converted to the code of the first state of the HGS Γ_v, which is generated on the outputs CCD$_R$,…,CCD$_1$ of the **Code Converter**. Now the HGS Γ_v is responsible for the control from this point until it terminates;

  4. after the termination of the HGS Γ_v it is necessary to decrement the stack pointer (y^-=1) in order to return back to the interrupted state. As a result control is passed to the state in which the HGS Γ_v was called.

If a HGS is used to calculate a logic function, the value to be calculated must be stored in an extra 1-bit register of the **Combinational Scheme**.

This model has the following advantages:

- since there are no extra lines used to identify the proper HGS the total number of inputs of the **Combinational Scheme** is smaller;
- many modifications of macrooperations (logic functions) can be done in the **Code Converter**, and there is no need to modify the kernel of the **Combinational Scheme**;
- if the **Combinational Scheme** and the **Code Converter** blocks are reprogrammable components such as RAMs, this model can provide such new facilities as flexibility, extensibility and reuse of an algorithm described by HGSs.

Another model proposed in [SklRocFer98] provides an association between HGSs in a given set and mutually exclusive elements in the **Combinational Scheme** called **Reprogrammable Element** (**RE**) (see Figure 4.5). In this model each HGS Γ_v from the set of HGSs $\Gamma_1,...,\Gamma_V$ is implemented with one autonomous circuit such as RE_v, and a one to one association between the set $\Gamma_1,...,\Gamma_V$ and the set of elements $RE_1,...,RE_V$ is obtained. This model allows optimising control units for algorithms with a large number of microoperations and whose model is the Mealy FSM, and can be seen as a good candidate for reprogrammable and reconfigurable control circuits, such as FPGAs.

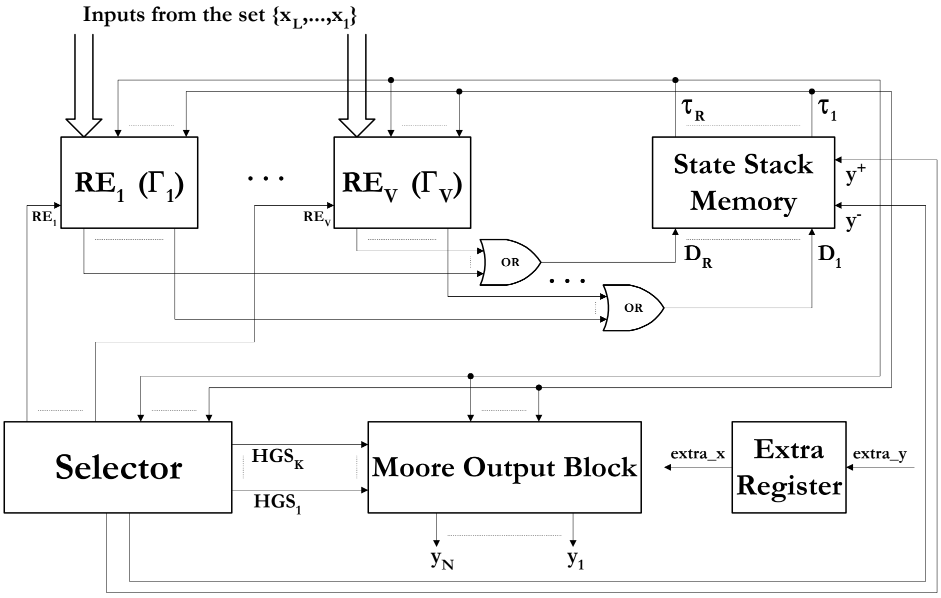


Figure 4.5 – Hierarchical finite state machine structure (model 3).

Let’s describe the functionality of this model with more detail. If this model is used to implement a Moore machine the **Moore Output Block** is necessary to provide the microoperations, that depend on the state binary code and on the HGS binary code. The **State Stack Memory** is used to store the current state of the HFSM, like in the previous models. The 1-bit register **Extra Register** is used to store the calculated value of a logic function and its input and output is connected with some REs, depending on their kind. There are three kinds of REs:

- a RE that implements a macrooperation and that does not invoke any logic function. This RE has one activation input RE_v, some inputs from the set $x_L,...,x_1$, the outputs $D_R,...,D_1$ to provide the next state to the **State Stack Memory** and some outputs from the set $y_N,...,y_1$ if a Mealy machine is being implemented;

- a RE that implements a macrooperation and that invokes at least one logic function. This RE has the inputs and outputs mentioned above and one extra input **extra_x** connected with the **Extra Register** output, in order to use the calculated value of the logic function;

- a RE that implements a logic function. This RE has the inputs and outputs mentioned for the first kind of RE and one extra output **extra_y** connected with the **Extra Register** input, in order to store the calculated value of the logic function, considering that the logic function is implemented as a Mealy machine. Moreover, it can also have an extra input **extra_x** if it invokes at least one logic function. But, if the logic function is implemented as a Moore machine, its calculated value is generated in the **Moore Output Block** and the RE does not have the extra output **extra_y**.

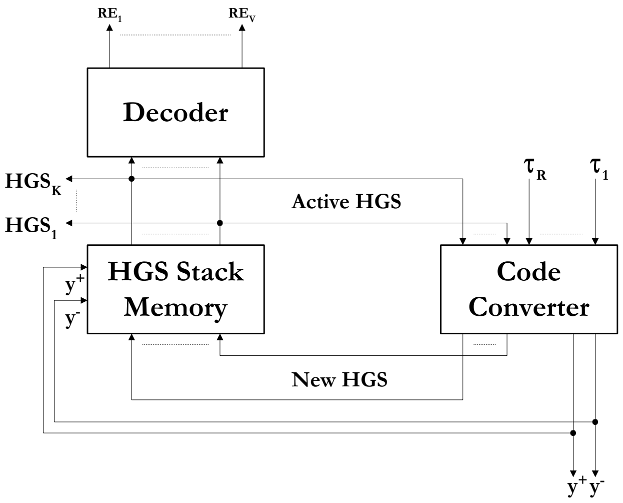


Figure 4.6 – Selector implementation.

The **Selector** (see Figure 4.6) is also based on stack memory (**HGS Stack Memory** block) and it is used for storing the binary code $K(\varepsilon_v)$ of the HGSs in accordance with the hierarchical sequence of the HGSs to be called. It enables the RE of the active HGS through the outputs $RE_1,...,RE_V$ provided by the **Decoder**. And it is also responsible for generating in the **Code Converter** the special signals that increment (y^+) and decrement (y^-) both stack pointers.

This model works as follows:

- when the HGS Γ_v is running its reprogrammable element RE_v connected to the **State Stack Memory** works as an ordinary FSM;
- if it is necessary to call a new HGS Γ_k in order to perform either a **macrooperation** or a **logic function** the following sequence of actions will be carried out:
  1. the **Code Converter** generates the signal y^+ and the code of the new HGS Γ_k that will run next;
  2. both stack pointers are incremented and a new register in each stack is selected. The new top of the **State Stack Memory** is clean (set to zero) if it is the first time that this hierarchy level is reached or holds the code of the returning state of the previous HGS that has run in this hierarchy level;
  3. The code of the HGS Γ_k is stored in the **HGS Stack Memory** and the **Decoder** activates the output RE_k, enabling the reprogrammable element RE_k and disabling the RE_v that was running. When the information stored in the new register of the **State Stack Memory** is presented at the inputs of the RE_k through the lines $\tau_R,...,\tau_1$, the RE_k will generate the entry state of the HGS Γ_k. Now the HGS Γ_k is responsible for the control from this point on until it is terminated;
  4. after the termination of the HGS Γ_k the **Code Converter** generates the signal y^-. Both stack pointers are decremented and control returns back to the interrupted state, stored in the **State Stack Memory**, of the interrupted HGS Γ_v, which code is stored in the **HGS Stack Memory**.

The considered scheme has the following advantages. It is:

- flexible in the sense that the functions of each RE can be easily modified because it implements autonomous and simple transitions for the associated HGS;
- extensible in the sense that the functions of each RE can be completely changed. Besides the scheme can be extended, adding new REs, without modifying the structure;
- virtual in the sense that a control algorithm can be implemented in a scheme with restricted resources, even if the complexity of the scheme is insufficient for implementing the entire algorithm. It is important that all links of REs are known. Each HGS is a relatively independent component, hence any HGS Γ_v can be replaced just by freezing a single link, which is the line RE_v from the **Selector** associated with the replacing RE_v (see Figure 4.5 and Figure 4.6). All other parts of the algorithm will not be suspended;
- the state coding is now of the form (HGS_code, local_state_code). Since state assignment can be done modularly, i.e. local to each HGS, the states in different HGSs will have the same local code and the local state code length will decrease. However, the state code length will increase.

As against the previous model, the binary code K(ε_v) containing all zeros must be assigned to the main HGS. Therefore the HGS binary code length is $K \geq |\log_2 V|$, where V is the number of HGSs used to described the algorithm.

4.3 Parallel HFSM

Now let's consider how to perform various macrooperations in parallel. The **parallel finite state machine** (**PFSM**) depicted in Figure 4.7 was proposed in [Skyarov87] and was explored in [RocSklFer97].

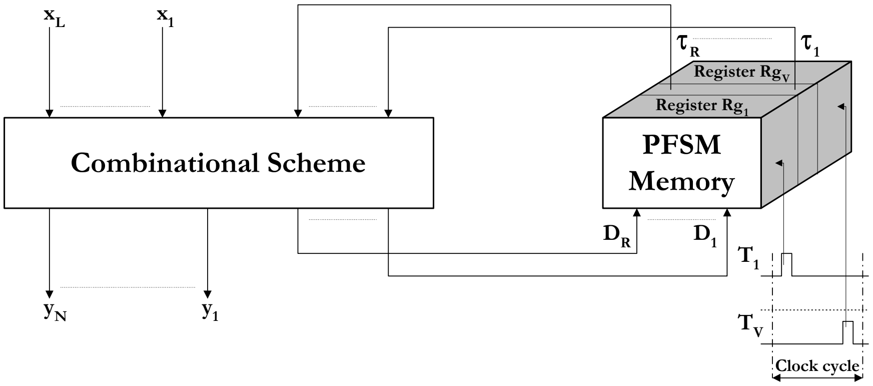


Figure 4.7 – Model of a pseudo-parallel finite state machine.

The memory of the PFSM (**PFSM Memory**) has R inputs D_R,...,D_1 and R outputs τ_R,...,τ_1 connected to the **Combinational Scheme**, and it is composed of V registers, which are sequentially scanned by activating the respective sub-clocks T_1,…,T_V (see Figure 4.7). V is the number of PHGSs used to describe the algorithm and defines the maximum number of macrooperations implemented in parallel. Inputs and outputs of the memory are common to all its registers.

In order to synchronise the parallel execution of macrooperations, the PFSM **Combinational Scheme** has a set of SR flip-flops (Z_i flip-flops), one for each PHGS of the algorithm except for the PHGS Γ_1 (for more details see PFSM synthesis). In order to store the calculated value of logic functions it also has a set of SR flip-flops (Θ_i flip-flops), one for each logic function of the algorithm.

The PFSM outputs can be persistent, i.e. being active during the entire clock cycle or non-persistent meaning that they only will be active during one sub-clock T_j. In order to provide persistent outputs, the PFSM must have a set of SR flip-flops to store the microoperations.

The clock cycle (see Figure 4.7) of the control circuit is divided into V sub-clocks and each such sub-clock is further divided into sub-clocks in order to provide the proper internal synchronisation. Each sub-clock affects the respective register and changes its state, if and only if the respective SR flip-flop is set.

The first pulse of the clock cycle changes the state of the register Rg_1 that is responsible for the main macrooperation. As a result the proper transition in the PHGS Γ_1 is performed. The next transition in the PHGS Γ_1 will be carried out with the first pulse of the next clock cycle. The second pulse causes a similar transition in the PHGS Γ_2, etc.

Suppose it is necessary to perform the macroinstruction Z_i, which is composed of more than one macrooperation. In this case the following sequence of actions takes place:

1. for each macrooperation z_j belonging to the macroinstruction Z_i, the PHGS Γ_j is asserted, i.e. its SR flip-flop is set, in order to activate the register j;
2. each following clock cycle causes the proper transitions in all asserted PHGSs, because if a PHGS Γ_j is passive, i. e. its SR flip-flop is reset, the respective sub-clock does not change its idle state. At the end of the clock cycle all macrooperations of the macroinstruction Z_i, will have been sequentially executed;
3. the interrupted PHGS is in a waiting state, introduced for the purpose of synchronisation, until all macrooperations of the macroinstruction Z_i have terminated and then resumes its execution.

This model has one constraint [Sklyarov87]. If a macrooperation is being executed then it cannot be invoked again, because the respective register is being used. In order that this constraint will be eliminated it is necessary to manage both hierarchy and parallelism in the same machine and the two schemes shown in Figure 4.4 and Figure 4.7 must be combined.

Indeed the **parallel hierarchical finite state machine** (**PHFSM**) presented in Figure 4.8, is visualised as a set of Q individual stacks with common inputs and outputs, where Q is equal to the total number of macrooperations that are invoked in parallel, plus one for the main PHGS. Each stack can be managed independently of the other stacks, i.e. increment and decrement operations are individual for the respective stack.

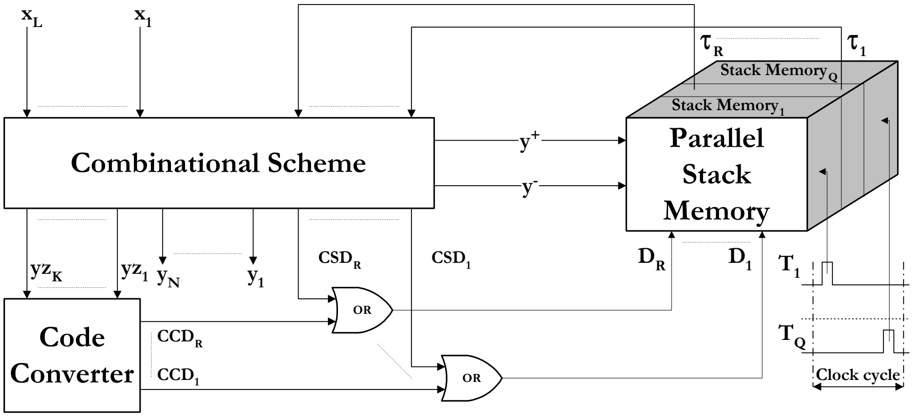


Figure 4.8 – Model of a pseudo-parallel hierarchical finite state machine.

This machine uses the PFSM synchronisation mechanism when performing a macroinstruction, which is composed of more than one macrooperation, but a single macrooperation or a logic function invocation is performed in a new hierarchical level like in the HFSM.

The PHFSM allows that a macrooperation can be executed in two different stacks and each of them has its own register. However, there are still some restrictions. A macrooperation that runs in parallel with others cannot invoke itself. This is because the stack number, given by the sub-clock T_j, is used to distinguish if a macrooperation is running in parallel or in a hierarchical invocation. If a macrooperation is running in parallel and it is recursive, all its invocations will run in the same stack and therefore it will not be possible to distinguish the first run from the following recursive runs (see the PHFSM synthesis for more details).

Moreover, if in a certain period of time, a set of macrooperations is running in parallel, each one of them can just invoke one of the others, eventually in parallel with other macrooperations that are inactive at the moment. Furthermore, two macrooperations running in parallel cannot invoke the same logic function at the same time. Since there is only one flip-flop per logic function, it must be ensured that a value already stored is used, before a new value will be stored.

4.4 Virtual HFSM

The implementation of a set of HGSs in a HFSM with stack memory allows to create a regular structure where all external connections are known and can be fixed for generally speaking an infinite number of applications (see Figure 4.4 and Figure 4.5). In this case the particular customising of the scheme can be achieved just by programming (or reprogramming) its components. The proposed models if implemented with RAM or with field-programmable devices such as FPGAs, can provide such new facilities as flexibility, extensibility and reuse of an algorithm described by HGSs.

Flexibility means to modify a given behaviour in minimal time and with minimal effort. Suppose that it is needed to modify a macrooperation call in an operational node, for example z_j instead of z_i. Or that it is required to change completely the functionality of an operational node, i.e. to change the active microoperations and the macrooperation invoked in the node.

For the HFSM model 2 (Figure 4.4), in the first case it is only necessary to reprogram the **Code Converter**, replacing the initial state addressed with the code of z_i with the initial state of the macrooperation z_j. This way can also be used to provide a flexible conditional node. It is possible to develop several versions of a logic function and then to change, if necessary, from one version to another by just reprogramming the initial state stored in the **Code Converter**.

In the second case it is necessary to reprogram the **Combinational Scheme**. More specifically, to reprogram the component responsible for generating the microoperations associated with the node. In order to change the macrooperation invoked in the node, and alternatively to reprogram the **Code Converter,** the macrooperation binary code generated in the node must be changed. Since the microoperations and the macrooperation binary code are generated in the same component, these two changes can be made altogether.

For the HFSM model 3, the first case can be done by reprogramming the **Code Converter** inside the **Selector** (see Figure 4.6), replacing the HGS binary code $K(\varepsilon_i)$ with the HGS binary code $K(\varepsilon_j)$ and the **Decoder** will activate the output line RE_j instead of the output line RE_i. For the second case the **Code Converter** and the component responsible for the generation of the microoperations must be reprogrammed. In the Moore machine, that means to reprogram the **Moore Output Block**, while for the Mealy machine that means to reprogram the RE where it is implemented the operational node that must be modified (see Figure 4.5).

Extensibility means to define a behaviour and then to extend it in order to improve something. Suppose that it is required to introduce a new operational node or delete an existing one. For model 2 it means to reprogram the **Combinational Scheme**, and for model 3 it is a question of reprogramming the respective RE.

Moreover, extensibility and flexibility can be achieved with virtual control circuits, which are able to replace virtual components of a control algorithm, such as virtual HGSs, and to modify them if necessary. In order to simplify the replacement of different components the HFSM model 3 provides a direct association between HGSs and REs. If it is required to modify the HGS behaviour or add/delete functionality, it is necessary to reprogram only the respective RE or add/delete REs to the scheme.

For the PHFSM, flexibility and extensibility are more difficult to apply, since changes in macrooperations invocation demands to reanalyse the PHGS in order to ensure the parallelism synchronisation.

Reuse means that reusable components such as separate HGSs can be created and they will be used for many different control algorithms which the designer expects to create in the future. This can be done by investing a little extra effort in the design of a library of reusable components that will facilitate the creation of similar products.

Generally speaking any dynamically reconfigurable device is a virtual circuit, which denotes that the same physical scheme can be used to implement different logic circuits. The idea is to combine a flexible and extensible behavioural description, i.e. HGSs, with an implementation based on virtual circuits such as FPGAs.

4.5 HFSM/PHFSM Synchronisation

As against of an ordinary FSM a HFSM/PHFSM has more complicated synchronisation. Indeed it is necessary to synchronise the following events:

- transitions between states;
- fixing the output variables $y_N,...,y_1$ (microoperations);
- fixing the input variables $x_L,...,x_1$ (logic conditions);
- setting the **Code Converter**;
- incrementing or decrementing the stack pointer;
- fixing the calculated value of a logic function, i. e. loading the **Extra Register** in the case of the HFSM, or setting/resetting the Z_i and Θ_i SR flip-flops in the case of the PHFSM.

There are many different kinds of synchronisation, but it should be considered just one where microoperations and macrooperations can be combined at the same operational node.

4.5.1 Synchronisation of a Moore HFSM

The synchronisation of a Moore HFSM, or a mixed Moore/Mealy machine, model 2 is depicted in Figure 4.9 and it is based on the synchronisation presented in [Sklyarov84] for the HFSM model 1.

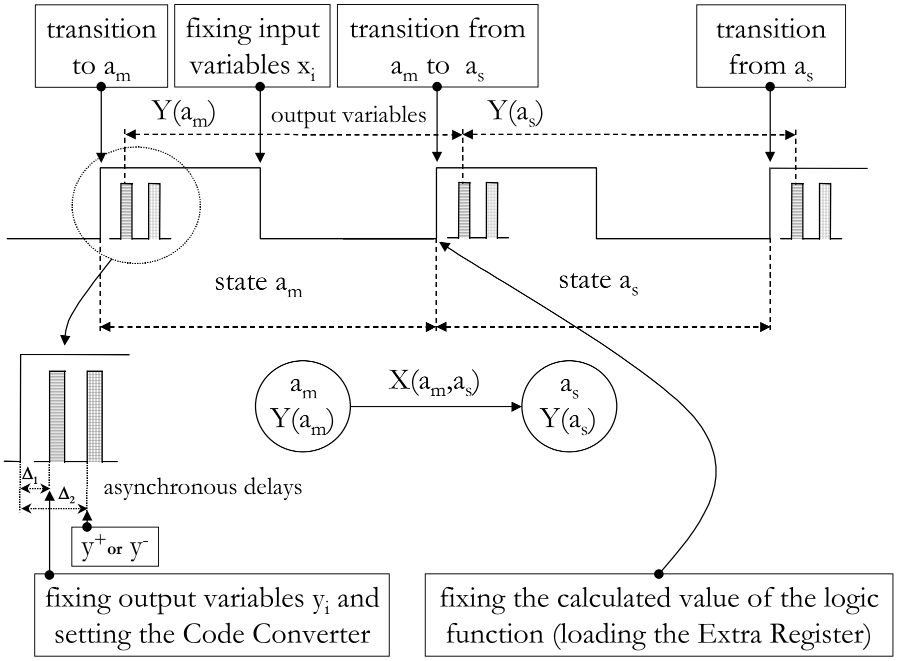

transition to a_m
fixing input variables x_i
transition from a_m to a_s
transition from a_s
$Y(a_m)$
output variables
$Y(a_s)$
state a_m
state a_s
a_m $Y(a_m)$
$X(a_m,a_s)$
a_s $Y(a_s)$
Δ_1
Δ_2
asynchronous delays
y^+ or y^-
fixing output variables y_i and setting the Code Converter
fixing the calculated value of the logic function (loading the Extra Register)

Figure 4.9 – Synchronisation of a Moore HFSM model 2.

The transition between states is synchronised by the low to high front of the **clock** and the next state is stored in the **Stack Memory**, wherever the state a_s is from the same or from a different hierarchical level than the state a_m (see Figure 4.9). After the delay of the **Stack Memory** the new state will be presented to the **Combinational Scheme**. The output signals of the new state, microoperations y_i, **Code Converter** inputs yz_i and the special signals y^+ and y^- are generated.

The first synchronisation pulse **synclk1** sets the output variables y_i and the **Code Converter** inputs yz_i. The time interval Δ_1 must be bigger than the sum of all delays involved in the generation of the signals mentioned above, in order to ensure that the proper values are being fixed, i.e. the values of the actual state and not of the previous state.

The second synchronisation pulse **synclk2** is responsible for controlling the **Stack Memory** pointer. If y^+ or y^- is activated, the stack pointer is incremented or decremented, otherwise it will not be changed. When the stack pointer is incremented the contents of the new top of the stack is output. If that register was already used it holds the state a_1 and the signal y^- is activated. Because this activation occurs after **synclk2** it will be ignored.

On the high to low transition of the **clock** the inputs x_i are fixed and with the new inputs and the present state the **Combinational Scheme** will evaluate the next state that will be stored on the **Stack Memory** at the next transition of the **clock**.

If a HGS is used to calculate a logic function, its returning value must be fixed on the high to low transition of the **clock**, when the transition from the last state of the HGS of the logic function to the interrupted state of the previous HGS occurs. This value must be stable during the next clock cycle where it will be used to decide the next state.

Because the microoperations and macrooperations are fixed at different periods of time, this mechanism allows combining them at the same operational node.

The synchronisation of a HFSM model 3 is more sensitive with the control of the stacks. That is why the signals y^+ and y^- are generated and fixed by the **Code Converter**. It is similar to model 2 with the following differences:

- both stacks, **State Stack Memory** and **HGS Stack Memory**, are controlled simultaneously by the signal **synclk2**;
- when the execution of a HGS is finishing (state a_1) and returning to the previous HGS (y^- active), the code of the interrupted HGS is not known. But this code is stored in the **HGS Stack Memory** and can be used if it is not overwritten with any other HGS code. In order to prevent an unwanted HGS code loading in transitions from the state a_1 to any other state, it is necessary to disable the clock that attacks the **HGS Stack Memory** when y^- is active. This is done by generating a special clock with an AND gate, with the **clock** and the signal **y^-** negated as inputs.

4.5.2 Synchronisation of a Mealy HFSM

The synchronisation of a Mealy HFSM model 2 depicted in Figure 4.10 is different from the Moore machine in the sequence of events, because the Mealy outputs are input dependent. Therefore, the synchronisation order of the three synchronisation events associated with the inputs, the outputs and the stack pointer must be shifted right.

The transition between states is synchronised by the low to high front of the **clock**. The first synchronisation pulse **synclk1** fixes the input signals x_i. Then after a certain amount of time the output signals associated with the state transition produced by these new inputs, microoperations y_i, **Code Converter** inputs yz_i and the special signals y^+ and y^- are generated.

The second synchronisation pulse **synclk2** sets the output signals. The time interval **Δ_2-Δ_1** must be bigger than the sum of delays involved in the generation of the output signals, in order to ensure that the proper values are being fixed.

The **clock**, on its high to low transition, is responsible for controlling the pointer of the **Stack Memory**. As a result the **Stack Memory** of the Mealy machine has only one synchronisation signal, the **clock**, but used in both transitions. The low to high controls the state storage while the high to low controls the stack pointer increment/decrement operation. The returning value of a logic function must be fixed like for a Moore machine.

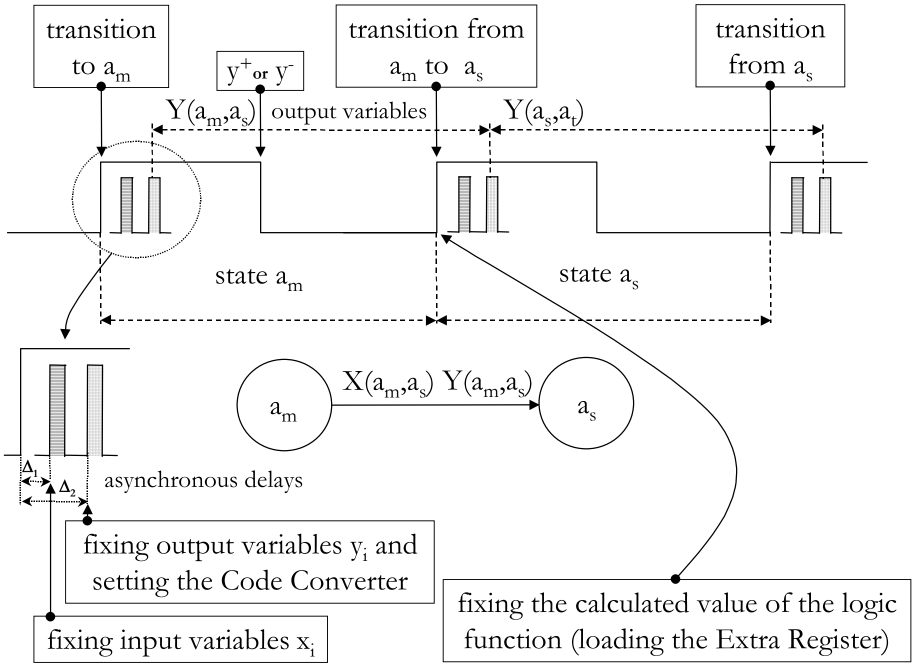

transition to a_m
y^+ or y^-
transition from a_m to a_s
transition from a_s
$Y(a_m,a_s)$
output variables
$Y(a_s,a_t)$
state a_m
state a_s
a_m
$X(a_m,a_s)$ $Y(a_m,a_s)$
a_s
Δ_1
Δ_2
asynchronous delays
fixing output variables y_i and setting the Code Converter
fixing input variables x_i
fixing the calculated value of the logic function (loading the Extra Register)

Figure 4.10 – Synchronisation of a Mealy HFSM model 2.

The Mealy HFSM model 3 synchronisation is similar to the Moore HFSM model 3 with the changes mentioned above.

4.5.3 Synchronisation of a PFSM

The synchronisation of a Moore PFSM is depicted in Figure 4.11. On the low to high transition of the **clock** the machine switches from one register to the next one. During each sub-clock, the output variables generated for the active state are fixed on the low to high front of the first synchronisation pulse **csclk**, that it is also used as the SR flip-flops clock. After fixing the input variables on the high to low transition of the **clock**, the definitive next state is calculated. This state is loaded, on the low to high transition of the second synchronisation pulse **stlclk**, before the termination of the sub-clock.

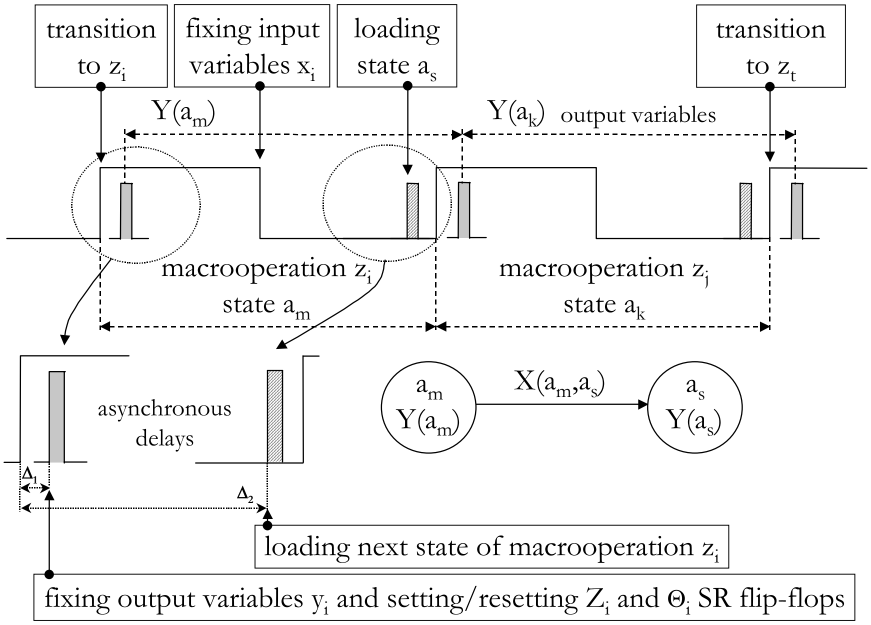


Figure 4.11 – Synchronisation of a Moore PFSM.

In the case of the Mealy PFSM the input variables must be fixed before the output variables, i.e. the sequence of the two events is reversed.

4.5.4 Synchronisation of a PHFSM

The synchronisation of a Moore PHFSM, depicted in Figure 4.12, is based on the PFSM synchronisation described in the previous paragraph (see Figure 4.11), combined with the mechanism used to synchronise the switching between hierarchical levels described for a Moore HFSM (see Figure 4.9).

On the low to high transition of the **clock** the machine switches from one stack to the next one. The first synchronisation pulse **cccsclk** fixes the output variables, the signal y^+, sets the **Code Converter** and sets/resets the Z_i and Θ_i flip-flops.

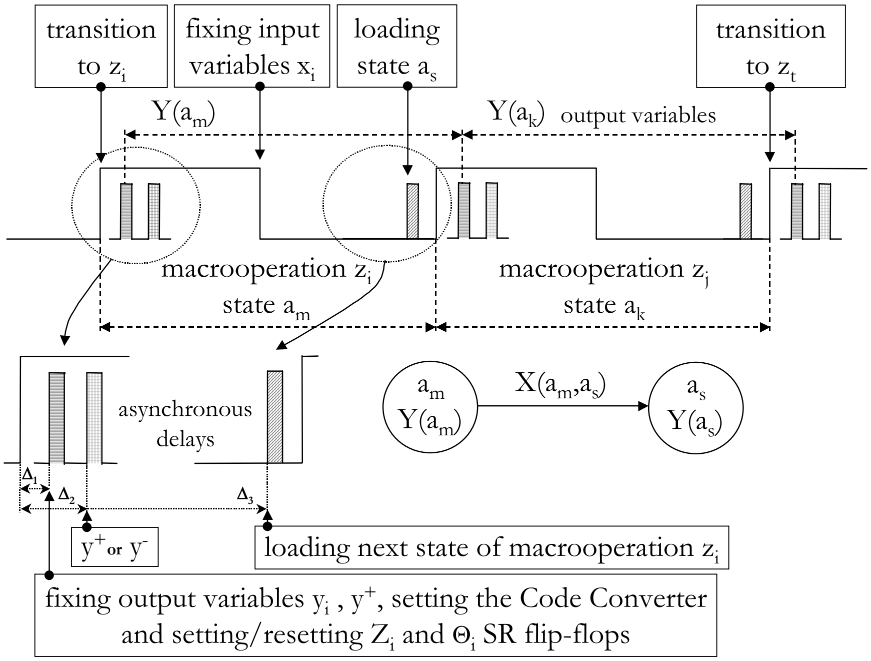


Figure 4.12 – Synchronisation of a Moore PHFSM.

The second synchronisation pulse **spclk** controls the stack pointer of the active stack. On the high to low transition of the **clock** the inputs x_i are fixed. The next state is generated and loaded on the active stack on the low to high transition of the third synchronisation pulse **stlclk**.

The signal y^+ must be fixed for allowing that the hierarchical invocations can work. When, during the sub-clock T_j, a macrooperation z_k (or a logic function θ_k) is invoked and must run in a new hierarchical level, the stack pointer of the stack j is incremented and a new register is selected. This new register is clean (state a_0) and the **Combinational Scheme** will generate the entry state of the macrooperation assigned to the stack j. As a result the signal y^+ will be reset. But, the **Combinational Scheme** must provide the state a_0 in order to let pass the state output by the **Code Converter**.

To obtain the desired performance, it is necessary to insert a multiplexer before the lines $D_R,\ldots,D_1$ exit the **Combinational Scheme**, controlled by the signal y^+ and with the following behaviour: when y^+ is negated the exit state will be the calculated next state, while when y^+ is asserted the output state will be a_0. But, in order to ensure this behaviour the signal y^+ must be fixed during the sub-clock T_j.

In the case of the Mealy PHFSM, the synchronisation sequence of the first three events must change like in a Mealy HFSM.

4.6 Application Field

Sophisticated field-programmable devices such as FPGAs, have gained a wide acceptance because they can be applied to a wide range of applications, they can be used to emulate an entire large hardware system using many interconnected FPGAs and they can also be used as custom computing machines [BroRos96].

Since FPGAs provide a large number of logic gates and flip-flops that can be connected in various ways, they can be tailored to implement any complex digital system. Therefore they are the target technology for implementing a complete system, i.e. the control unit and the datapath.

Besides, since the specification of embedded systems may change continuously and designers must pay attention to such problems as the addition of new functionality in the future [Edwards97], FPGAs should be used because some of them can be reprogrammed as many times as the designer needs.

Moreover, it should be mentioned that the new dynamically reconfigurable FPGA, such as Xilinx XC6200 [Xilinx97] might be customised on the fly to implement a specific software function with very high performance. They can be reprogrammed partially without suspending operation of other parts that do not need to be reconfigured [Xilinx97]. This is achieved by specifying the desired interconnections between logic components (gates, flip-flops, etc.) using SRAM. Therefore, it is possible to use a FPGA with smaller logic resources than those needed to implement the system.

A typical hardware architecture for an embedded system combines custom hardware normally using ASICs with embedded software running on general-purpose microcontrollers. The functionality of an embedded system is a trade-off between performance and cost. Performance is achieved through the implementation of tasks in custom hardware, while the decrease of the manufacturing cost is achieved through the implementation of tasks in software.

Since FPGAs can be tailored to implement any complex system, they can replace ASICs with gains in the manufacturing cost. On the other hand due to the use of the HFSMs, complex control algorithms normally implemented in software can migrate to hardware and therefore FPGAs can replace microcontroller-based implementations with gains in performance.

Since the proposed HFSMs can create hardware systems that implement recursive algorithms they can be used to create hardware engines that will speed up the performance of software algorithms which have recursive solutions that are more intuitive and efficient than iterative solutions, in the mathematical and database fields for example.

4.7 Conclusions

HFSMs with stack memory can efficiently implement a hierarchical algorithm, even a recursive algorithm, provided that the size of the stack is well dimensioned. And if the next state and the output functions are implemented with reprogrammable components such as RAMs or FPGAs, they can provide such new facilities as flexibility, extensibility and reusability.

The introduction of the **Code Converter** in the HFSM model 2 (Figure 4.4), that specifically and efficiently generates the entry state of each hierarchical level, allows to separate the transition function of the machine in two, the transition function between states at the same hierarchical level generated in the **Combinational Scheme** and the transition function between different hierarchical levels generated in the **Code Converter**. Therefore, the complexity of the **Combinational Scheme** decreases, due to a decrease in the number of its inputs and the HFSM becomes more flexible because it is possible to change hierarchical transitions by just reprogramming the **Code Converter**. But, if a HGS requires to be redesigned the **Combinational Scheme** must be reprogrammed.

In order to overcome that, the HFSM model 3 (Figure 4.5 and Figure 4.6) provides an association between the set of HGSs and mutually exclusive **Reprogrammable Elements**. Each RE can be autonomously modified in order to respond to the respective HGS redesign and the HFSM functionality can be easily extended or curtailed with the addition or deletion of REs. The dimension of the stack memory is smaller because the local state code length decreases but this model needs two stacks against one in the previous models.

Moreover, this model can be seen as the HFSM natural interpretation of the hierarchical specification based on HGSs and since the **Combinational Scheme** acquires a regular and modular structure it is more suitable for the implementation of Mealy HFSMs in reprogrammable and reconfigurable control circuits, such as FPGAs.

The proposed PHFSM model (Figure 4.8) eliminates some restrictions of the PFSM (Figure 4.7) and can manage both hierarchy and pseudo-parallelism.

The implementation of HGSs (PHGSs) with HFSMs (PHFSMs) allows that a regular structure, where all external connections are known and can be fixed for generally speaking an infinite number of applications, can be created. With the flexibility, extensibility and reusability provided by the proposed models, the HFSMs (PHFSMs) can be seen as virtual circuits.

The synchronisation mechanism proposed for the HFSM can combine microoperations and macrooperations at the same operational node, and efficiently implement the transition between different levels in the hierarchy. And the synchronisation proposed for the PHFSM allows combining the functionality of the HFSM with the pseudo-parallel execution of macrooperations.

5 SYNTHESIS OF HIERARCHICAL FINITE STATE MACHINES

Summary

Synthesis is the process of transforming a control unit specification with a set of HGSs (PHGSs), into a logic scheme that will implement the HFSM (PHFSM) models that were proposed in the previous chapter.

FSM synthesis involves the following steps: constructing a state transition table; state encoding; combinational logic optimisation and design of the final scheme. In the case of the parallel FSMs the synthesis process has an initial step to provide proper synchronisation of parallel macrooperations. Since, all known methods of logic synthesis can be applied to a state transition table, the purpose of this chapter is to fully explain how to transform a hierarchical algorithm described by a set of HGSs (PHGSs) to an **ordinary state transition table**, accordingly with the HFSM (PHFSM) proposed models.

In order to perform this first step of logic synthesis, it is necessary to mark the set of HGSs (PHGSs) with states, to record all transitions between the states in an extended transition table and then to transform this table to ordinary form. Since all these sub-steps have too many rules and are error prone, it is useful to consider **automatic generation of state transition tables** that are correct by construction. The methods of the tool **SIMULHGS** that provide the automatic synthesis of a hierarchical algorithm described by a set of HGSs are presented.

5.1 Introduction

Synthesis is a process of design refinement where a more abstract specification, such as a set of HGSs (PHGSs), is translated into a less abstract specification, such as a logic scheme that will implement a HFSM (PHFSM).

The synthesis of hierarchical and parallel FSMs involves the following steps:
1. converting a given set of HGSs/PHGSs to a state transition table;
2. state encoding;
3. combinational logic optimisation and design of the final scheme.

In the case of parallel FSMs (hierarchical or not) the given set of PHGSs has to be extended and transformed to a new set of PHGSs providing proper synchronisation of parallel macrooperations, before converting it to a state transition table.

The first step is divided into the three following sub-steps:
1.1. marking the set of HGSs/PHGSs with states;
1.2. recording all transitions between states in the extended state transition table;
1.3. transforming the extended table to ordinary form.

The marking rules for the new proposed models of HFSMs are based on the rules defined for the HFSM model 1 presented in [Sklyarov84].

Let's explain in detail the above sub-steps for all hierarchical and parallel FSMs models proposed in the previous chapter.

5.2 Synthesis of a Moore HFSM

In a Moore machine, the outputs are associated with the states. Therefore, in order to convert a set of HGSs to a Moore machine, the nodes that generate outputs (microoperations and internal signals used to control the switching between hierarchical levels) must be labelled with states. And since a HGS needs an entry state to start its execution, if the first node of a HGS is a conditional node that contains a logic condition, the node **Begin** is also marked with a state.

In order to mark the given set of HGSs (sub-step 1.1) for the Moore HFSM model 2, it is necessary to perform the following actions:

- the label a_0 is assigned to the node **Begin** and to the node **End** of the main HGS (see the HGS Γ_1 in Figure 5.1);
- the label a_1 is assigned to all nodes **End** of HGSs $\Gamma_2,...,\Gamma_V$ (see HGSs $\Gamma_2,...,\Gamma_6$ in Figure 5.1);

- the labels $a_2,a_3,...,a_M$ are assigned to the following nodes and inputs:
  - the operational nodes in HGSs $\Gamma_1,...,\Gamma_V$ (see HGSs $\Gamma_1,...,\Gamma_6$ of Figure 5.1);
  - the inputs of conditional nodes, which contain logic functions, and which directly follow either other conditional nodes, or operational nodes with macrooperations (see state a_{12} in HGS Γ_2 of Figure 5.1);
  - the nodes **Begin** of HGSs $\Gamma_2,...,\Gamma_V$ which have connections to conditional nodes with logic conditions (see state a_{13} in HGS Γ_3 of Figure 5.1);
  - the input of a conditional node with a logic condition that follows a direct connection from the node **Begin** of the main HGS (HGS Γ_1);
- apart from a_1, all the labels in the various HGSs must be unique;
- if any node (input) has already been labelled, it must not be labelled again.

After applying this procedure to the set of HGSs presented in Figure 3.7, the set of HGSs depicted in Figure 5.1 labelled with the states $a_0,\ldots,a_{20}$ (M=21) is obtained.

In order to build the extended state transition table (sub-step 1.2) for the Moore HFSM model 2, it is necessary to perform the actions listed below:

- record all transitions $\mathbf{a_m X(a_m,a_s) a_s}$, where $\mathbf{a_m} \in \{a_2,...,a_M\}$, $\mathbf{a_s} \in \{a_0,...,a_M\}$ and $\mathbf{X(a_m,a_s)}$ is the product of input variables (logic conditions) and logic functions which causes the transition from $\mathbf{a_m}$ to $\mathbf{a_s}$;
- record all sets $\mathbf{YZ(a_m)=Y(a_m) \cup Z(a_m)}$, which are subsets of the microoperations $Y(a_m)$ and the macrooperations $Z(a_m)$ generated in the operational node marked with the label $\mathbf{a_m}$ (see state a_3 in Table 5.1). It is acceptable that $YZ(a_m) \cap Z=\varnothing$ (see for example state a_2 in Table 5.1) or $YZ(a_m) \cap Y=\varnothing$ (see for example state a_4 in Table 5.1). The main macrooperation must be recorded for the state $\mathbf{a_0}$ (see state a_0 in Table 5.1);
- record $\boldsymbol{\theta}\mathbf{(a_m)}$, which is the logic function generated in the input of the conditional node marked with the label $\mathbf{a_m}$ (see state a_{12} in Table 5.1), or if the input of the conditional node containing the logic function $\boldsymbol{\theta_k}$ has not been labelled, record $\boldsymbol{\theta_k}$, in all states $a_{f1},a_{f2},...$ that have been assigned to nodes from which follow direct arrow lines to the considered conditional node (see state a_6 in Table 5.1);
- record $\boldsymbol{\theta}\mathbf{(a_m)=1}$, which is the logic function calculated value generated in the operational node marked with the label $\mathbf{a_m}$ (see state a_{19} in Table 5.1);
- each transition is recorded in one row of the structured table and all transitions from the same state are grouped.

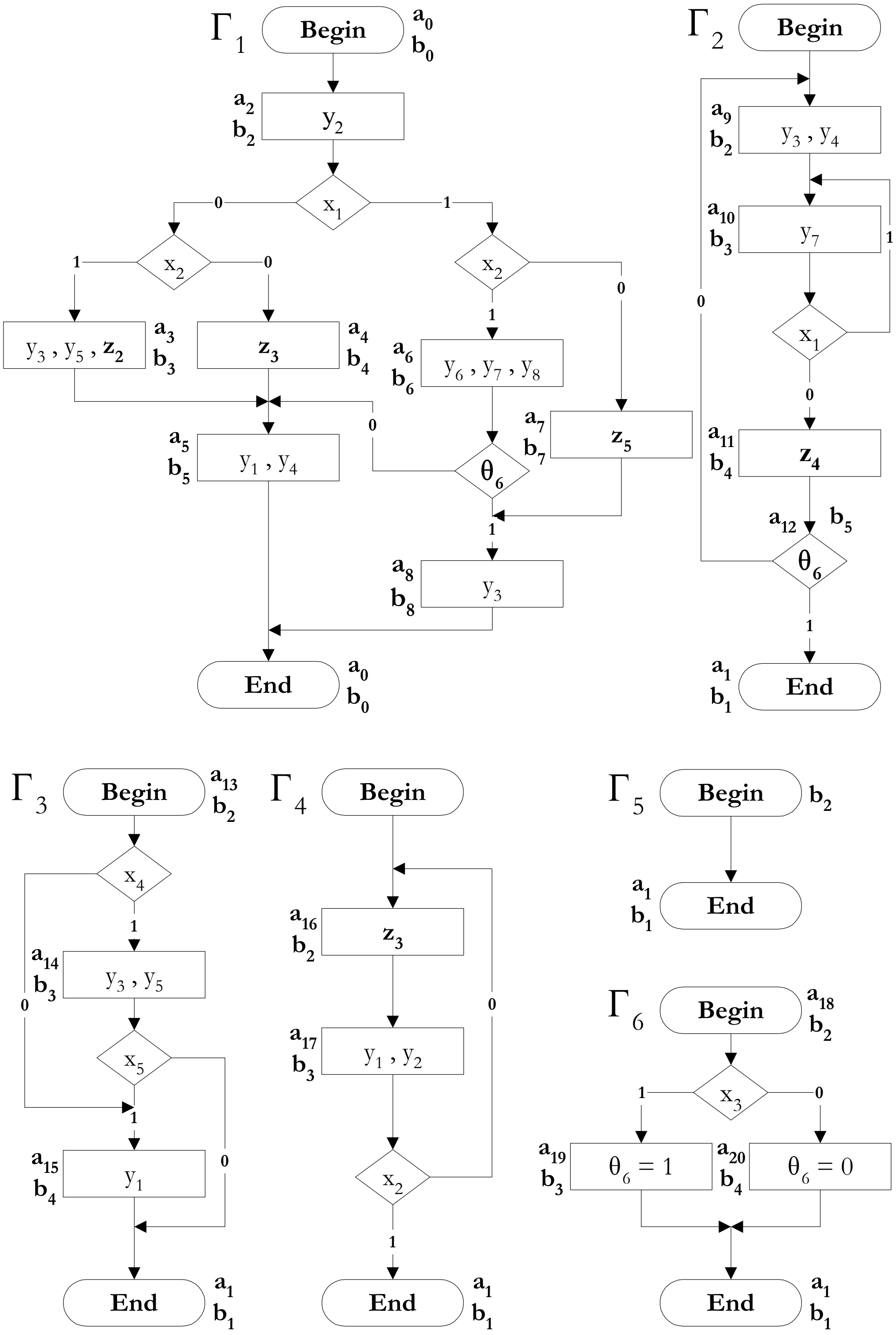


Figure 5.1 – A set of HGSs marked for synthesis as a Moore machine (states a_i for model 2, states b_i for model 3).

After applying this sub-step to the set of HGSs depicted in Figure 5.1, the extended state transition Table 5.1 is built. In order to convert the table from the extended form to ordinary form (sub-step 1.3), it is necessary to perform the following actions:

- generate y^- in the state a_1 (see Table 5.2);
- replace each macrooperation with its binary code $K(\varepsilon_v)$, i. e. the extra outputs yz_i that control the **Code Converter**. Add the new output variable y^+, except for the main macrooperation (see for example state a_3 in Table 5.2);
- replace each logic function θ_k, with its binary code $K(\varepsilon_v)$ and add y^+. If the input of the conditional node containing the logic function θ_k has been labelled with the state a_u, this change has to be made to the set $Y(a_u)$ (see state a_{12} in Table 5.2). If the input of the conditional node containing the logic function θ_k has not been labelled, this change has to be made to the sets $Y(a_{f1})$,$Y(a_{f2})$,... where the states a_{f1},a_{f2},... have been assigned to nodes from which follow direct arrow lines to the considered conditional node (see state a_6 in Table 5.2);
- replace all symbols θ_k in the column $\mathbf{X(a_m,a_s)}$ with the internal input variable **extra_x** which represents the value of θ_k (see states a_6 and a_{12} in Table 5.2);
- replace all symbols $\theta_k=1$ in the column $Y(a_m)$ with the internal output variable **extra_y** (see state a_{19} in Table 5.2). It should be done in order to return the calculated value of θ_k from the called HGS to the calling HGS via the 1-bit **Extra Register** (see the decomposition of the **Combinational Scheme** depicted in Figure 6.2).

After applying this sub-step to the extended state transition Table 5.1, the ordinary state transition Table 5.2 is obtained. Now, all known methods of logic synthesis can be applied to this table to carry out steps 2 and 3.

It must be kept in mind, that the states a_0 and a_1 are used to switch between the hierarchical levels of the HFSM. Therefore, these states do not assert any microoperations and for both of them the **Combinational Scheme** must set to zero the outputs $CSD_R,\dots,CSD_1$, i.e. it must generate the next state a_0 (see Table 5.2).

In order to synthesise a Moore HFSM it is necessary to create the entry state generation table of the **Code Converter**, that provides the entry state of each HGS for the respective HGS binary code $K(\varepsilon_v)$. For the binary code with all bits set to zero, code $K(\varepsilon_v)$ which is used to clear the **Code Converter** outputs, and for all the remaining binary codes not in use, the **Code Converter** will output the state a_0 (see Table 5.3).

Table 5.1 – Moore extended state transition table.

a_m { $Y(a_m)$ }	a_s	$X(a_m,a_s)$
a_0 $\{z_1\}$	a_0	1
a_1	a_0	1
a_2 $\{y_2\}$	a_6 a_7 a_3 a_4	$x_1\ x_2$ $x_1\ \overline{x}_2$ $\overline{x}_1\ x_2$ $\overline{x}_1\ \overline{x}_2$
a_3 $\{y_3,y_5,z_2\}$	a_5	1
a_4 $\{z_3\}$	a_5	1
a_5 $\{y_1,y_4\}$	a_0	1
a_6 $\{y_6,y_7,y_8,\theta_6\}$	a_8 a_5	θ_6 $\overline{\theta}_6$
a_7 $\{z_5\}$	a_8	1
a_8 $\{y_3\}$	a_0	1
a_9 $\{y_3,y_4\}$	a_{10}	1
a_{10} $\{y_7\}$	a_{10} a_{11}	x_1 $\overline{x}_1$
a_{11} $\{z_4\}$	a_{12}	1
a_{12} $\{\theta_6\}$	a_1 a_9	θ_6 $\overline{\theta}_6$
a_{13}	a_{14} a_{15}	x_4 $\overline{x}_4$
a_{14} $\{y_3,y_5\}$	a_{15} a_1	x_5 $\overline{x}_5$
a_{15} $\{y_1\}$	a_1	1
a_{16} $\{z_3\}$	a_{17}	1
a_{17} $\{y_1,y_2\}$	a_1 a_{16}	x_2 $\overline{x}_2$
a_{18}	a_{19} a_{20}	x_3 $\overline{x}_3$
a_{19} $\{\theta_6=1\}$	a_1	1
a_{20}	a_1	1

Table 5.2 – Moore ordinary state transition table.

a_m { $Y(a_m)$ }	a_s	$X(a_m,a_s)$
a_0 $\{yz_1\}$	a_0	1
a_1 $\{y^-\}$	a_0	1
a_2 $\{y_2\}$	a_6 a_7 a_3 a_4	$x_1\ x_2$ $x_1\ \overline{x}_2$ $\overline{x}_1\ x_2$ $\overline{x}_1\ \overline{x}_2$
a_3 $\{y_3,y_5,yz_2,y^+\}$	a_5	1
a_4 $\{yz_2,yz_1,y^+\}$	a_5	1
a_5 $\{y_1,y_4\}$	a_0	1
a_6 $\{y_6,y_7,y_8,yz_3,yz_2,y^+\}$	a_8 a_5	$extra\_x$ $\overline{extra\_x}$
a_7 $\{yz_3,yz_1,y^+\}$	a_8	1
a_8 $\{y_3\}$	a_0	1
a_9 $\{y_3,y_4\}$	a_{10}	1
a_{10} $\{y_7\}$	a_{10} a_{11}	x_1 $\overline{x}_1$
a_{11} $\{yz_3,y^+\}$	a_{12}	1
a_{12} $\{yz_3,yz_2,y^+\}$	a_1 a_9	$extra\_x$ $\overline{extra\_x}$
a_{13}	a_{14} a_{15}	x_4 $\overline{x}_4$
a_{14} $\{y_3,y_5\}$	a_{15} a_1	x_5 $\overline{x}_5$
a_{15} $\{y_1\}$	a_1	1
a_{16} $\{yz_2,yz_1,y^+\}$	a_{17}	1
a_{17} $\{y_1,y_2\}$	a_1 a_{16}	x_2 $\overline{x}_2$
a_{18}	a_{19} a_{20}	x_3 $\overline{x}_3$
a_{19} $\{extra\_y\}$	a_1	1
a_{20}	a_1	1

Table 5.3 – Moore model 2 Code Converter table.

HGS Γ_v	$K(\varepsilon_v)$ [yz_i]	Initial State
	$\overline{yz_3}\ \overline{yz_2}\ \overline{yz_1}$	a_0
z_1	$\overline{yz_3}\ \overline{yz_2}\ yz_1$	a_2
z_2	$\overline{yz_3}\ yz_2\ \overline{yz_1}$	a_9
z_3	$\overline{yz_3}\ yz_2\ yz_1$	a_{13}
z_4	$yz_3\ \overline{yz_2}\ \overline{yz_1}$	a_{16}
z_5	$yz_3\ \overline{yz_2}\ yz_1$	a_1
θ_6	$yz_3\ yz_2\ \overline{yz_1}$	a_{18}
	$yz_3\ yz_2\ yz_1$	a_0

For the HFSM model 3 the given set of HGSs (sub-step 1.1) has to be labelled modularly, using the rules defined for model 2, i.e. the HGS Γ_i labelling starts always with the state b_2. In the case of a pure virtual macrooperation the node **Begin** must be labelled with the state b_2. The set of HGSs depicted in Figure 5.1 labelled with the states b_i is obtained after applying this marking procedure.

The extended table is built (sub-step 1.2), one table for each HGS instead of just one global table for the set of HGSs, in the same manner as for model 2, with the exception that the macrooperations **$Z(a_m)$** and the logic functions **$\theta(a_m)$** are not recorded in the extended table. Only the microoperations **$Y(a_m)$** and the logic functions assignment values are recorded.

The conversion to ordinary form (sub-step 1.3) applies the same rules of model 2 (see Table 5.4, Table 5.5, Table 5.6, Table 5.7, Table 5.8 and Table 5.9).

Like in the HFSM model 2, the states b_0 and b_1 are used to switch between the hierarchical levels of the HFSM. Since the reprogrammable element is responsible for generating the entry state of the respective HGS, the next state for both states is the state b_2 in all HGSs tables.

The **Code Converter** table generates the next HGS and the extra signals used to control the stacks for each state of each HGS. If a state has to perform the macrooperation z_v (or the logic function θ_v) it generates the signal y^+ and the binary code $K(\varepsilon_v)$, represented in the table by the HGS name. In all b_1 states the signal y^- is generated. In this situation and because it is not possible to know which HGS called the HGS that is finishing to run, the HGS binary code generated is indifferent. For the remaining states it is output the binary code $K(\varepsilon_v)$ of the active HGS (see Table 5.10).

Table 5.4 – RE$_1$ Moore ordinary state transition table.

b_m {$Y(b_m)$}	b_s	$X(b_m,b_s)$
b_0	b_2	1
b_1	b_2	1
b_2 {y_2}	b_6	$x_1\, x_2$
	b_7	$x_1\, \bar{x}_2$
	b_3	$\bar{x}_1\, x_2$
	b_4	$\bar{x}_1\, \bar{x}_2$
b_3 {y_3,y_5}	b_5	1
b_4	b_5	1
b_5 {y_1,y_4}	b_0	1
b_6 {y_6,y_7,y_8}	b_8	$extra\_x$
	b_5	$\overline{extra\_x}$
b_7	b_8	1
b_8 {y_3}	b_0	1

Table 5.5 – RE$_2$ Moore ordinary state transition table.

b_m {$Y(b_m)$}	b_s	$X(b_m,b_s)$
b_0	b_2	1
b_1	b_2	1
b_2 {y_3,y_4}	b_3	1
b_3 {y_7}	b_3	x_1
	b_4	$\bar{x}_1$
b_4	b_5	1
b_5	b_1	$extra\_x$
	b_2	$\overline{extra\_x}$

Table 5.6 – RE$_3$ Moore ordinary state transition table.

b_m {$Y(b_m)$}	b_s	$X(b_m,b_s)$
b_0	b_2	1
b_1	b_2	1
b_2	b_3	x_4
	b_4	$\bar{x}_4$
b_3 {y_3,y_5}	b_4	x_5
	b_1	$\bar{x}_5$
b_4 {y_1}	b_1	1

Table 5.7 – RE$_4$ Moore ordinary state transition table.

b_m {$Y(b_m)$}	b_s	$X(b_m,b_s)$
b_0	b_2	1
b_1	b_2	1
b_2	b_3	1
b_3 {y_1,y_2}	b_1	x_2
	b_2	$\bar{x}_2$

Table 5.8 – RE$_5$ Moore ordinary state transition table.

b_m {$Y(b_m)$}	b_s	$X(b_m,b_s)$
b_0	b_2	1
b_1	b_2	1
b_2	b_1	1

Table 5.9 – RE$_6$ Moore ordinary state transition table.

b_m {$Y(b_m)$}	b_s	$X(b_m,b_s)$
b_0	b_2	1
b_1	b_2	1
b_2	b_3	x_3
	b_4	$\bar{x}_3$
b_3 {$extra\_y$}	b_1	1
b_4	b_1	1

Table 5.10 – Moore model 3 Code Converter table.

Active HGS	State	Next HGS	Stack Pointer
z_1	b_3	z_2	y^+
	b_4	z_3	y^+
	b_6	θ_6	y^+
	b_7	z_5	y^+
	all other states	z_1	
z_2	b_1	--	y^-
	b_4	z_4	y^+
	b_5	θ_6	y^+
	all other states	z_2	
z_3	b_1	--	y^-
	all other states	z_3	
z_4	b_1	--	y^-
	b_2	z_3	y^+
	all other states	z_4	
z_5	b_1	--	y^-
	all other states	z_5	
θ_6	b_1	--	y^-
	all other states	θ_6	

5.3 Synthesis of a Mealy HFSM

In a Mealy machine, the outputs are associated with the transition between states. Therefore, in order to convert a set of HGSs to a Mealy machine, the inputs of nodes that follow nodes that generate outputs (microoperations and internal signals used to control the switching between hierarchical levels) must be labelled with states. Moreover, a node that invokes a macrooperation (logic function) must has its input marked with a state in order to generate the appropriate internal signals needed to fire up the macrooperation (logic function) execution. Furthermore, since a HGS needs an entry state to start its execution, the node after the node **Begin** is also marked with a state.

In order to mark the given set of HGSs (sub-step 1.1) for the Mealy HFSM model 2, it is necessary to perform the following actions:

- the label a_0 is assigned to the node **Begin** and to the node **End** of the main HGS (see the HGS Γ_1 in Figure 5.2);
- the label a_1 is assigned to all nodes **End** of HGSs $\Gamma_2,...,\Gamma_V$ (see HGSs $\Gamma_2,...,\Gamma_6$ in Figure 5.2);
- the labels $a_2,a_3,...,a_M$ are assigned to the following inputs:
  - inputs of nodes which directly follow the node **Begin** in all HGSs $\Gamma_1,...,\Gamma_V$ (see for example state a_{10} in HGS Γ_2 of Figure 5.2);
  - inputs of operational nodes with **macrooperations** (designate the subset of respective labels as $\mathbf{A^{mo}}$) (see for example state a_4 in HGS Γ_1 of Figure 5.2);
  - inputs which directly follow from output(s) of operational node(s) in HGS $\Gamma_1,...,\Gamma_V$ (see for example state a_3 in HGS Γ_1 of Figure 5.2);
  - inputs of conditional nodes containing **logic functions** (designate the subset of respective labels as $\mathbf{A^{lf}}$) (see for example state a_7 in HGS Γ_1 of Figure 5.2);
  - inputs which directly follow the true and false outputs of conditional nodes containing **logic functions** (see states a_6 and a_9 in HGS Γ_1 of Figure 5.2);
- apart from a_1, all the labels in the various HGSs must be unique;
- if any input has already been labelled, it must not be labelled again.

The marking rules require more states for marking a set of HGSs as a Mealy machine rather than for a Moore machine. Two states against one are needed for operational nodes containing a macrooperation, but on the other hand, one state, eventually none, against one is needed for operational nodes containing microoperations. And for conditional nodes containing logic function three states against one, eventually none are needed.

After applying this procedure to the set of HGSs presented in Figure 3.7, the set of HGSs depicted in Figure 5.2 labelled with the states $a_0,\ldots,a_{20}$ (M=21) is obtained. For this example the number of states is the same for both machines.

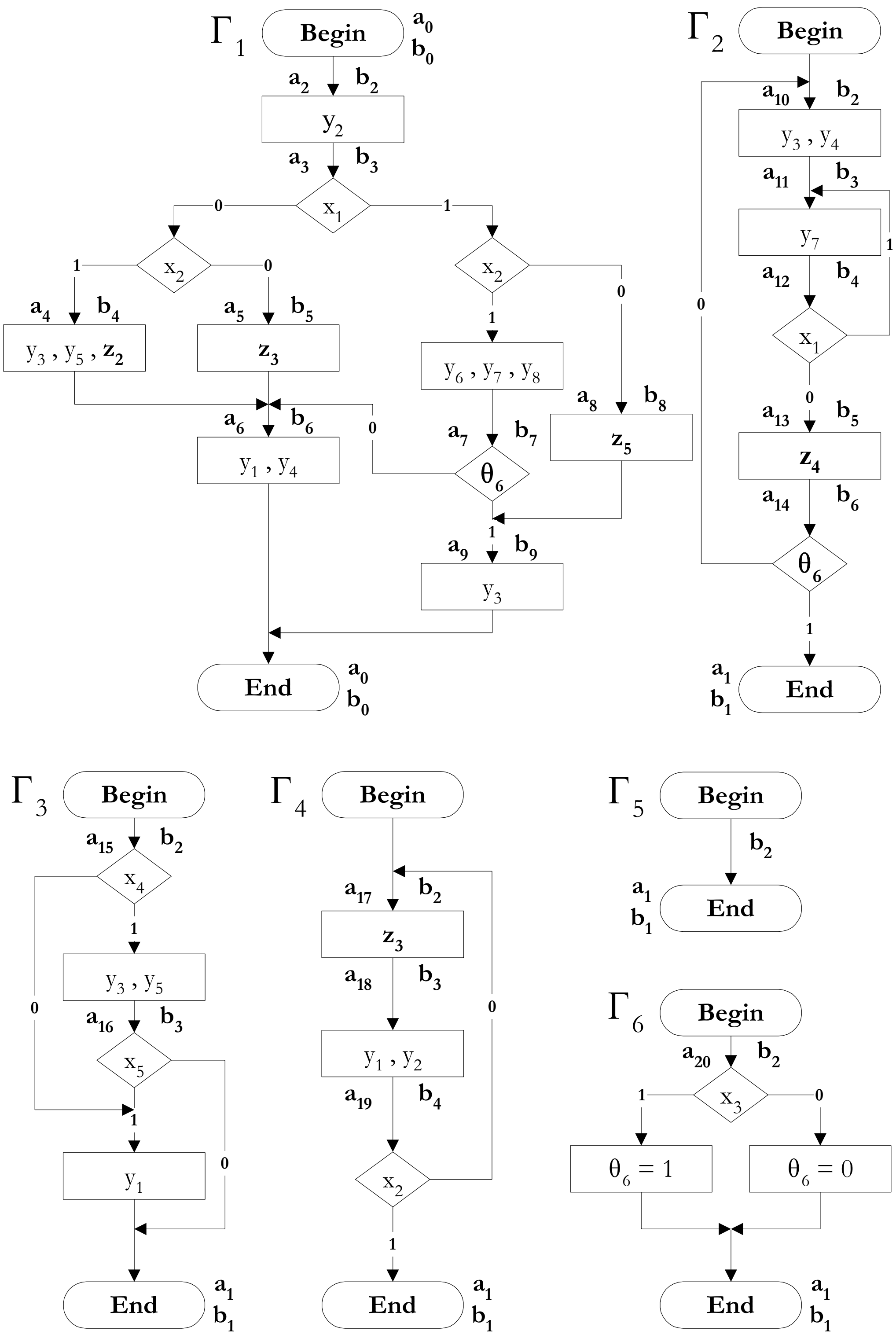


Figure 5.2 – A set of HGSs marked for synthesis as a Mealy machine (states a_i for model 2, states b_i for model 3).

For the second sub-step, which is recording all transitions between the states in the extended state transition table, it is necessary to perform the following actions:

- record all transitions $\mathbf{a_m X(a_m,a_s) YZ(a_m,a_s) a_s}$ and $\mathbf{a_m X(a_m,a_s) a_s}$, where $\mathbf{X(a_m,a_s)}$ is the product of input variables (logic conditions) and logic functions which causes the transition from $\mathbf{a_m}$ to $\mathbf{a_s}$, $\mathbf{YZ(a_m,a_s)=Y(a_m,a_s) \cup Z(a_m)}$ which are subsets of the microoperations $Y(a_m,a_s)$ that have active values after the transition and the macrooperations $Z(a_m)$ invoked in the operational node whose input is marked with the label $\mathbf{a_m}$. It is acceptable that $YZ(a_m,a_s) \cap Z=\varnothing$ (see for example transition a_2 to a_3 in Table 5.12), or $YZ(a_m,a_s) \cap Y=\varnothing$ (see for example transition a_5 to a_6 in Table 5.12). The main macrooperation must be recorded for the state $\mathbf{a_0}$ (see state a_0 in Table 5.12). The transition $\mathbf{a_m X(a_m,a_s) YZ(a_m,a_s) a_s}$ passes exactly through just one operational node ($X(a_m,a_s)=1$ is admissible), while the transition $\mathbf{a_m X(a_m,a_s) a_s}$ passes only through conditional nodes. For some cases only the latter transition can be used. In any other cases it is mandatory to perform the former transition. If the state $a_s \in A^{lf} \cup A^{mo}$ then it is not allowed to pass any transition through the state a_s. It can only either start any transition from the state a_s or finish any transition in the state a_s;

- record all transitions $\mathbf{a_m X(a_m,a_s) [\theta_k=1] a_s}$, which is the logic function θ_k calculated value generated in the operational node traversed by the transition from $\mathbf{a_m}$ to $\mathbf{a_s}$ (see state a_{20} in Table 5.12);

- each transition is being recorded in one **row** of the structural table and all transitions from the same state are grouped.

After applying this sub-step to the HGSs depicted in Figure 5.2, the extended state transition Table 5.12 is built. In order to convert the table from extended form to ordinary form (see Table 5.13) and to create the code converter table (see Table 5.11), all the rules described for a Moore machine apply.

Table 5.11 – Mealy model 2 Code Converter table.

HGS Γ_v	$K(\varepsilon_v)$ [yz_i]	Initial State
	$\overline{yz_3}\ \overline{yz_2}\ \overline{yz_1}$	a_0
z_1	$\overline{yz_3}\ \overline{yz_2}\ yz_1$	a_2
z_2	$\overline{yz_3}\ yz_2\ \overline{yz_1}$	a_{10}
z_3	$\overline{yz_3}\ yz_2\ yz_1$	a_{15}
z_4	$yz_3\ \overline{yz_2}\ \overline{yz_1}$	a_{17}
z_5	$yz_3\ \overline{yz_2}\ yz_1$	a_1
θ_6	$yz_3\ yz_2\ \overline{yz_1}$	a_{20}
	$yz_3\ yz_2\ yz_1$	a_0

Table 5.12 – Mealy extended state transition table.

a_m	a_s	$X(a_m,a_s)$	$Y(a_m,a_s)$
a_0	a_0	1	z_1
a_1	a_0	1	
a_2	a_3	1	y_2
a_3	a_7	$x_1 x_2$	y_6,y_7,y_8
	a_8	$x_1 \overline{x}_2$	
	a_4	$\overline{x}_1 x_2$	
	a_5	$\overline{x}_1 \overline{x}_2$	
a_4	a_6	1	y_3,y_5,z_2
a_5	a_6	1	z_3
a_6	a_0	1	y_1,y_4
a_7	a_9	θ_6	θ_6
	a_6	$\overline{\theta}_6$	θ_6
a_8	a_9	1	z_5
a_9	a_0	1	y_3
a_{10}	a_{11}	1	y_3,y_4
a_{11}	a_{12}	1	y_7
a_{12}	a_{12}	x_1	y_7
	a_{13}	$\overline{x}_1$	
a_{13}	a_{14}	1	z_4
a_{14}	a_1	θ_6	θ_6
	a_{10}	$\overline{\theta}_6$	θ_6
a_{15}	a_{16}	x_4	y_3,y_5
	a_1	$\overline{x}_4$	y_1
a_{16}	a_1	x_5	y_1
	a_1	$\overline{x}_5$	
a_{17}	a_{18}	1	z_3
a_{18}	a_{19}	1	y_1,y_2
a_{19}	a_1	x_2	
	a_{17}	$\overline{x}_2$	
a_{20}	a_1	x_3	$\theta_6=1$
	a_1	$\overline{x}_3$	

Table 5.13 – Mealy ordinary state transition table.

a_m	a_s	$X(a_m,a_s)$	$Y(a_m,a_s)$
a_0	a_0	1	yz_1
a_1	a_0	1	y^-
a_2	a_3	1	y_2
a_3	a_7	$x_1 x_2$	y_6,y_7,y_8
	a_8	$x_1 \overline{x}_2$	
	a_4	$\overline{x}_1 x_2$	
	a_5	$\overline{x}_1 \overline{x}_2$	
a_4	a_6	1	y_3,y_5,yz_2,y^+
a_5	a_6	1	yz_2,yz_1,y^+
a_6	a_0	1	y_1,y_4
a_7	a_9	$extra\_x$	yz_3,yz_2,y^+
	a_6	$\overline{extra\_x}$	yz_3,yz_2,y^+
a_8	a_9	1	yz_3,yz_1,y^+
a_9	a_0	1	y_3
a_{10}	a_{11}	1	y_3,y_4
a_{11}	a_{12}	1	y_7
a_{12}	a_{12}	x_1	y_7
	a_{13}	$\overline{x}_1$	
a_{13}	a_{14}	1	yz_3,y^+
a_{14}	a_1	$extra\_x$	yz_3,yz_2,y^+
	a_{10}	$\overline{extra\_x}$	yz_3,yz_2,y^+
a_{15}	a_{16}	x_4	y_3,y_5
	a_1	$\overline{x}_4$	y_1
a_{16}	a_1	x_5	y_1
	a_1	$\overline{x}_5$	
a_{17}	a_{18}	1	yz_2,yz_1,y^+
a_{18}	a_{19}	1	y_1,y_2
a_{19}	a_1	x_2	
	a_{17}	$\overline{x}_2$	
a_{20}	a_1	x_3	$extra\_y$
	a_1	$\overline{x}_3$	

All the rules defined above for the Mealy HFSM model 2 in conjunction with the particular aspects presented in the Moore HFSM model 3, apply for the Mealy HFSM model 3. After the marking sub-step the set of HGSs depicted in Figure 5.2 labelled with the states b_i is obtained and the set of tables Table 5.14, Table 5.15, Table 5.16, Table 5.17, Table 5.18 and Table 5.19 are obtained after applying the sub-steps 1.2 and 1.3. The Mealy **Code Converter** table is generated as for the Moore machine (see Table 5.20).

Table 5.14 – RE_1 Mealy ordinary state transition table.

b_m	b_s	$X(b_m,b_s)$	$Y(b_m,b_s)$
b_0	b_2	1	
b_1	b_2	1	
b_2	b_3	1	y_2
b_3	b_7	$x_1\,x_2$	y_6,y_7,y_8
	b_8	$x_1\,\bar{x}_2$	
	b_4	$\bar{x}_1\,x_2$	
	b_5	$\bar{x}_1\,\bar{x}_2$	
b_4	b_6	1	y_3,y_5
b_5	b_6	1	
b_6	b_0	1	y_1,y_4
b_7	b_9	$extra\_x$	
	b_6	$\overline{extra\_x}$	
b_8	b_9	1	
b_9	b_0	1	y_3

Table 5.15 – RE_2 Mealy ordinary state transition table.

b_m	b_s	$X(b_m,b_s)$	$Y(b_m,b_s)$
b_0	b_2	1	
b_1	b_2	1	
b_2	b_3	1	y_3,y_4
b_3	b_4	1	y_7
b_4	b_4	x_1	y_7
	b_5	$\bar{x}_1$	
b_5	b_6	1	
b_6	b_1	$extra\_x$	
	b_2	$\overline{extra\_x}$	

Table 5.16 – RE_3 Mealy ordinary state transition table.

b_m	b_s	$X(b_m,b_s)$	$Y(b_m,b_s)$
b_0	b_2	1	
b_1	b_2	1	
b_2	b_3	x_4	y_3,y_5
	b_1	$\bar{x}_4$	y_1
b_3	b_1	x_5	y_1
	b_1	$\bar{x}_5$	

Table 5.17 – RE_5 Mealy ordinary state transition table.

b_m	b_s	$X(b_m,b_s)$	$Y(b_m,b_s)$
b_0	b_2	1	
b_1	b_2	1	
b_2	b_1	1	

Table 5.18 – RE_4 Mealy ordinary state transition table.

b_m	b_s	$X(b_m,b_s)$	$Y(b_m,b_s)$
b_0	b_2	1	
b_1	b_2	1	
b_2	b_3	1	
b_3	b_4	1	y_1,y_2
b_4	b_1	x_2	
	b_2	$\bar{x}_2$	

Table 5.19 – RE_6 Mealy ordinary state transition table.

b_m	b_s	$X(b_m,b_s)$	$Y(b_m,b_s)$
b_0	b_2	1	
b_1	b_2	1	
b_2	b_1	x_3	$extra\_y$
	b_1	$\bar{x}_3$	

Table 5.20 – Mealy model 3 Code Converter table.

Active HGS	State	Next HGS	Stack Pointer
z_1	b_4	z_2	y^+
	b_5	z_3	y^+
	b_7	θ_6	y^+
	b_8	z_5	y^+
	all other states	z_1	
z_2	b_1	--	y^-
	b_5	z_4	y^+
	b_6	θ_6	y^+
	all other states	z_2	
z_3	b_1	--	y^-
	all other states	z_3	
z_4	b_1	--	y^-
	b_2	z_3	y^+
	all other states	z_4	
z_5	b_1	--	y^-
	all other states	z_5	
θ_6	b_1	--	y^-
	all other states	θ_6	

5.4 Synthesis of a Mixed Moore/Mealy HFSM

If a set of HGSs contains many macrooperation and logic function invocations, as against an ordinary FSM, the synthesised Mealy HFSM has more states than Moore. This is due to the fact that in a Moore HFSM, in opposition to Mealy, it is possible to use the same state to generate the microoperations and to fire up the execution of a macrooperation. But on the other hand, it is more convenient to use a Mealy HFSM to implement a logic function that does not invoke any other logic functions, i.e. that is mainly composed with conditional nodes and few assignment nodes, in order to spare states and to speed up its evaluation.

Therefore, a mixed Moore/Mealy HFSM is proposed in order to take the better of the two machines. In this mixed HFSM, the calculated value of a logic function **extra_y** is a Mealy output generated in the appropriate state transition (see transition a_{18} x_3 a_1 in Table 5.21).

This mixed Moore/Mealy model marking is depicted in Figure 5.3. The ordinary state transition table, for the HFSM model 2, is the combination of Table 5.2 and Table 5.13, and it is presented in Table 5.21*. Its **Code Converter** table is the same as for the Moore HFSM presented in Table 5.3.

Table 5.21 – Mixed Moore/Mealy ordinary state transition table.

a_m $\{Y(a_m)/Y(a_m,a_s)\}$	a_s	$X(a_m,a_s)$
a_0 $\{yz_1\}$	a_0	1
a_1 $\{y^-\}$	a_0	1
a_2 $\{y_2\}$	a_6 a_7 a_3 a_4	$x_1\ x_2$ $x_1\ \overline{x}_2$ $\overline{x}_1\ x_2$ $\overline{x}_1\ \overline{x}_2$
a_3 $\{y_3,y_5,yz_2,y^+\}$	a_5	1
a_4 $\{yz_2,yz_1,y^+\}$	a_5	1
a_5 $\{y_1,y_4\}$	a_0	1
a_6 $\{y_6,y_7,y_8,yz_3,yz_2,y^+\}$	a_8 a_5	$extra\_x$ $\overline{extra\_x}$
a_7 $\{yz_3,yz_1,y^+\}$	a_8	1
a_8 $\{y_3\}$	a_0	1
a_9 $\{y_3,y_4\}$	a_{10}	1
a_{10} $\{y_7\}$	a_{10} a_{11}	x_1 $\overline{x}_1$
a_{11} $\{yz_3,y^+\}$	a_{12}	1
a_{12} $\{yz_3,yz_2,y^+\}$	a_1 a_9	$extra\_x$ $\overline{extra\_x}$
a_{13}	a_{14} a_{15}	x_4 $\overline{x}_4$
a_{14} $\{y_3,y_5\}$	a_{15} a_1	x_5 $\overline{x}_5$
a_{15} $\{y_1\}$	a_1	1
a_{16} $\{yz_2,yz_1,y^+\}$	a_{17}	1
a_{17} $\{y_1,y_2\}$	a_1 a_{16}	x_2 $\overline{x}_2$
a_{18} $\{extra\_y\}$	a_1 a_1	x_3 $\overline{x}_3$

The mixed Moore/Mealy HFSM model 3 for this example is implemented with the Z_i Moore tables (Table 5.4, Table 5.5, Table 5.6, Table 5.7, Table 5.8), the Θ_6 Mealy table (Table 5.19), and the Moore **Code Converter** table (Table 5.10).

[*] Due to space limitation Mealy and Moore outputs are presented altogether in the column $\{Y(a_m)/Y(a_m,a_s)\}$.

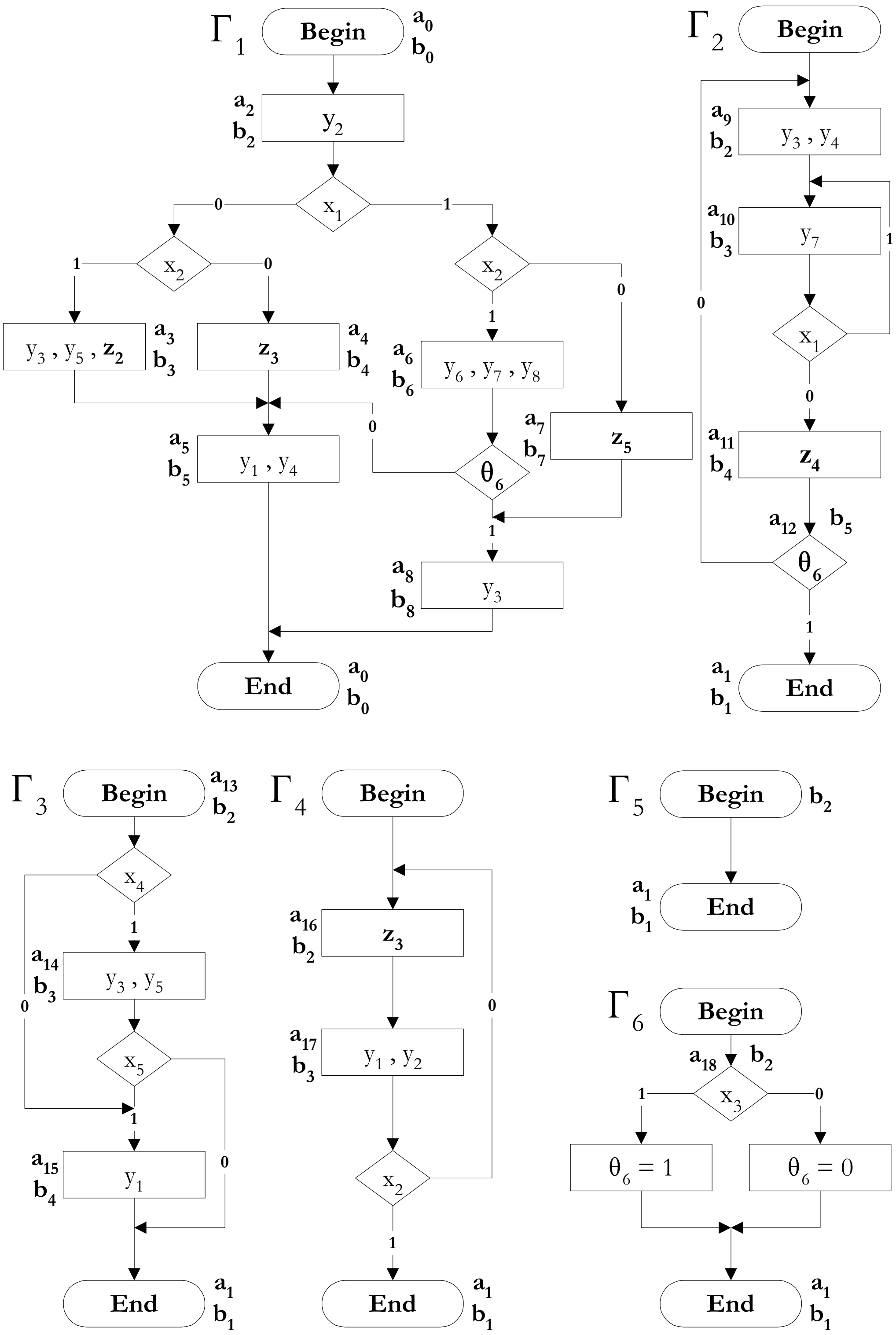


Figure 5.3 – A set of HGSs marked for synthesis as a mixed Moore/Mealy machine (states a_i for model 2, states b_i for model 3).

5.5 Synthesis of a PFSM

Like it was already mentioned, the synthesis of the PFSM must start by providing proper synchronisation of parallel macrooperations. In order to extend and transform the set of PHGSs for a parallel non-hierarchical FSM implementation it is necessary to perform the following steps [Sklyarov87]:

- to add a new group of nodes after any node of a PHGS, containing macrooperations. The first is an empty operational node, i.e. it does not hold any operations. The other nodes are conditional nodes, one for each macrooperation, and each one holding a logic condition Z_i. The logic condition Z_i is asserted ("1") if the respective PHGS Γ_i is running and it is negated ("0") if the PHGS Γ_i has been terminated. This group provides the proper suspension of the PHGS, which is invoking the macrooperations, until all macrooperations will end their execution. If at least one macrooperation is still active the transition from the considered above empty node is not allowed. After all macrooperations are passive the ordinary execution of the PHGS will continue (see PHGS Γ_1 in Figure 5.5);

- to add a new group of nodes composed of two operational and one conditional nodes, before any node of a PHGS containing a logic function (see PHGS Γ_2 in Figure 5.5). The first rectangular nodes enables the logic function PHGS, asserting its logic condition, and it is designated by Θ_k for the logic function θ_k. The latter two nodes are used in order to suspend the executing PHGS until the logic function will be calculated;

- to add after the node **Begin**, in all PHGSs except for the PHGS Γ_1, the conditional node which contains for the PHGS Γ_i the logic condition Z_i (see Figure 5.5). If $Z_i = 1$ the PHGS Γ_i must be activated. If $Z_i = 0$ the PHGS Γ_i must be in passive (idle) state. This node is used in order to synchronise the execution of different PHGSs;

- to generate a special signal, which is designated Z_i^* for the PHGS Γ_i, before the node **End** in all PHGSs apart from the PHGS Γ_1 (see Figure 5.5). Sometimes, an extra operational node must be added for that purpose (see PHGS Γ_2 in Figure 5.5). This signal is used in order to reset the associated logic condition considered in the previous item.

After applying these rules to the set of PHGSs presented in Figure 5.4, the set of extended PHGSs depicted in Figure 5.5 is obtained.

Since the FSM memory is scanned by activating the respective sub-clocks $T_1,\ldots,T_V$ (see Figure 4.7), the parallel machine needs a counter and a decoder to generate the proper sequence of pulses. Assuming that, during the clock T_v the PHGS Γ_v is processed and the counter binary code can be used to identify the macrooperation (or logic function) that is running. Hence, each extended PHGS should be marked separately in order to reduce the complexity of state encoding.

Therefore, the rules defined for the HFSM model 3 can be applied to mark the extended set of PHGSs, with only one difference. The label a_0 is assigned to the nodes **Begin** and **End** of all PHGSs (see Figure 5.5).

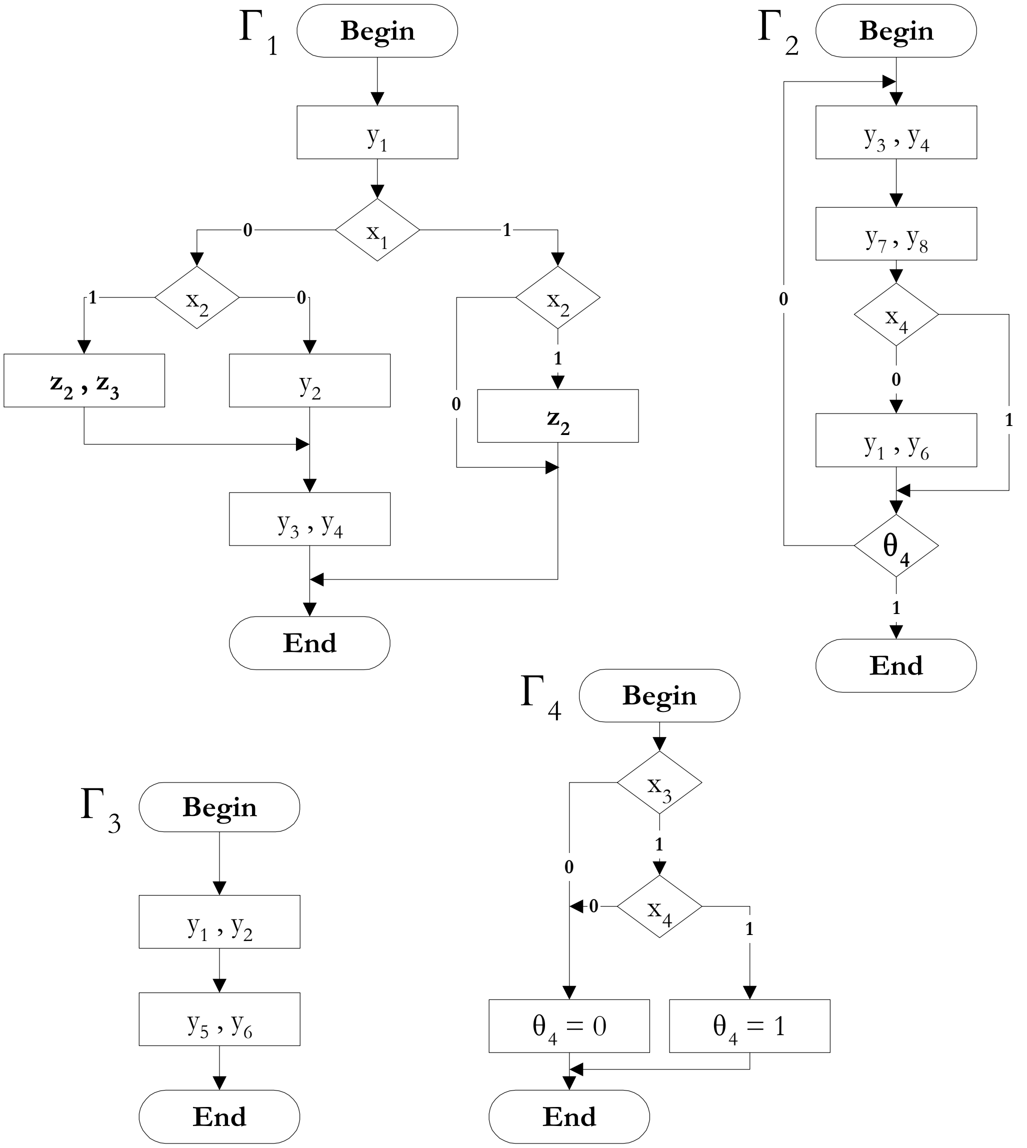


Figure 5.4 – A set of PHGSs for implementing in a PFSM.

To create the extended state transition Table 5.22, it is necessary to apply the same actions as for the HFSM. And to convert it to the ordinary state transition Table 5.23 it is necessary to perform the following actions:

- replace each macrooperation z_i (logic function θ_i) invocation, with the signal Z_i. Each macrooperation (logic function), except the PHGS Γ_1 has a SR flip-flop that holds its state (running or passive). The signal Z_i sets the flip-flop and enables the macrooperation (logic function) to run, while the signal Z_i^* resets the flip-flop and disables the macrooperation (logic function);

- It is also needed a SR flip-flop per logic function to hold its calculated value. Replace all symbols θ_k=1 in the column Y(a_m) with the signal Θ_k (setting the θ_k SR flip-flop), and all symbols θ_k=0 with the signal Θ_k^* (resetting the θ_k SR flip-flop);
- replace all symbols θ_k and $\bar{\theta}_k$ in the column X(a_m,a_s) with the signals Θ_k and $\bar{\Theta}_k$ respectively, that represent the value stored in the θ_k SR flip-flop.

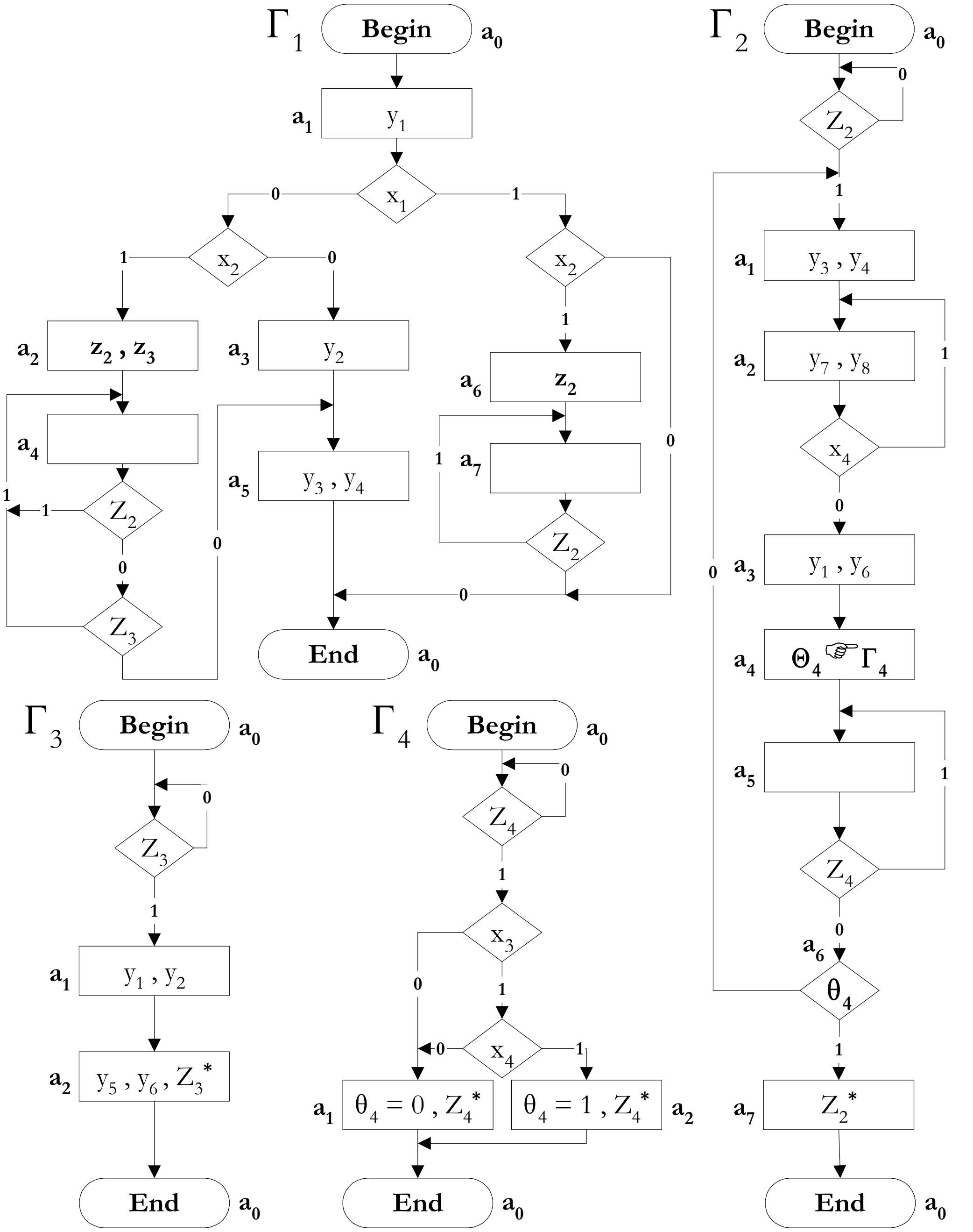


Figure 5.5 – A set of extended PHGSs marked for synthesis as a Moore PFSM.

Table 5.22 – PFSM extended state transition table.

ε_v a_m { $Y(a_m)$ }	a_s	$X(a_m,a_s)$
z_1 a_0	a_1	1
z_1 a_1 $\{y_1\}$	a_6 a_0 a_2 a_3	$x_1\ x_2$ $x_1\ \bar{x}_2$ $\bar{x}_1\ x_2$ $\bar{x}_1\ \bar{x}_2$
z_1 a_2 $\{z_2,z_3\}$	a_4	1
z_1 a_3 $\{y_2\}$	a_5	1
z_1 a_4	a_4 a_4 a_5	Z_2 $\bar{Z}_2\ Z_3$ $\bar{Z}_2\ \bar{Z}_3$
z_1 a_5 $\{y_3,y_4\}$	a_0	1
z_1 a_6 $\{z_2\}$	a_7	1
z_1 a_7	a_7 a_0	Z_2 $\bar{Z}_2$
z_2 a_0	a_0 a_1	$\bar{Z}_2$ Z_2
z_2 a_1 $\{y_3,y_4\}$	a_2	1
z_2 a_2 $\{y_7,y_8\}$	a_2 a_3	x_4 $\bar{x}_4$
z_2 a_3 $\{y_1,y_6\}$	a_4	1
z_2 a_4 $\{\theta_4\}$	a_5	1
z_2 a_5	a_5 a_6	Z_4 $\bar{Z}_4$
z_2 a_6	a_7 a_1	θ_4 $\bar{\theta}_4$
z_2 a_7 $\{Z_2^*\}$	a_0	1
z_3 a_0	a_0 a_1	$\bar{Z}_3$ Z_3
z_3 a_1 $\{y_1,y_2\}$	a_2	1
z_3 a_2 $\{y_5,y_6,Z_3^*\}$	a_0	1
z_4 a_0	a_0 a_1 a_1 a_2	$\bar{Z}_4$ $Z_4\ \bar{x}_3$ $Z_4\ x_3\ \bar{x}_4$ $Z_4\ x_3\ x_4$
z_4 a_1 $\{\theta_4=0,Z_4^*\}$	a_0	1
z_4 a_2 $\{\theta_4=1,Z_4^*\}$	a_0	1

Table 5.23 – PFSM ordinary state transition table.

ε_v a_m { $Y(a_m)$ }	a_s	$X(a_m,a_s)$
z_1 a_0	a_1	1
z_1 a_1 $\{y_1\}$	a_6 a_0 a_2 a_3	$x_1\ x_2$ $x_1\ \bar{x}_2$ $\bar{x}_1\ x_2$ $\bar{x}_1\ \bar{x}_2$
z_1 a_2 $\{Z_2,Z_3\}$	a_4	1
z_1 a_3 $\{y_2\}$	a_5	1
z_1 a_4	a_4 a_4 a_5	Z_2 $\bar{Z}_2\ Z_3$ $\bar{Z}_2\ \bar{Z}_3$
z_1 a_5 $\{y_3,y_4\}$	a_0	1
z_1 a_6 $\{Z_2\}$	a_7	1
z_1 a_7	a_7 a_0	Z_2 $\bar{Z}_2$
z_2 a_0	a_0 a_1	$\bar{Z}_2$ Z_2
z_2 a_1 $\{y_3,y_4\}$	a_2	1
z_2 a_2 $\{y_7,y_8\}$	a_2 a_3	x_4 $\bar{x}_4$
z_2 a_3 $\{y_1,y_6\}$	a_4	1
z_2 a_4 $\{Z_4\}$	a_5	1
z_2 a_5	a_5 a_6	Z_4 $\bar{Z}_4$
z_2 a_6	a_7 a_1	Θ_4 $\bar{\Theta}_4$
z_2 a_7 $\{Z_2^*\}$	a_0	1
z_3 a_0	a_0 a_1	$\bar{Z}_3$ Z_3
z_3 a_1 $\{y_1,y_2\}$	a_2	1
z_3 a_2 $\{y_5,y_6,Z_3^*\}$	a_0	1
z_4 a_0	a_0 a_1 a_1 a_2	$\bar{Z}_4$ $Z_4\ \bar{x}_3$ $Z_4\ x_3\ \bar{x}_4$ $Z_4\ x_3\ x_4$
z_4 a_1 $\{\Theta_4^*,Z_4^*\}$	a_0	1
z_4 a_2 $\{\Theta_4,Z_4^*\}$	a_0	1

5.6 Synthesis of a PHFSM

The synthesis of the PHFSM involves the same steps considered for the PFSM. In order, to transform the set of PHGSs to extended form the parallelism and the hierarchy must be combined. Logic functions and a solo macrooperation invocations should be implemented taking advantage of the hierarchy, while the invocation of a set of macrooperations must be implemented in parallel.

Constructing extended PHGSs for the PHFSM involves the following steps:

- to add a new group of nodes, after any node of the PHGS containing more than one macrooperation, in order to perform the same synchronisation of parallelism of the PFSM. Like it was explained for the PFSM, the first is an empty operational node and the others are conditional nodes holding the logic conditions Z_i (see PHGS Γ_1 in Figure 5.7);
- it is not necessary to add any synchronisation nodes, after a node containing just one macrooperation or before a node containing a logic function (see for example z_5 in PHGS Γ_1 in Figure 5.7);
- to add after the node **Begin**, in all PHGSs implementing macrooperations that run in parallel except for the PHGS Γ_1, the conditional node, which contains for the PHGS Γ_i the logic condition Z_i. This node is used in order to synchronise the execution of different macrooperations in parallel (see PHGS Γ_2, Γ_3 and Γ_4 in Figure 5.7);
- to add before the node **End**, in all PHGSs implementing macrooperations that run in parallel, an operational node which contains for the PHGS Γ_i the special signal Z_i^*. This signal is used in order to reset the associated logic condition considered in the previous item (see PHGS Γ_2 in Figure 5.7). This node is not necessary, if an operational node containing only microoperations is situated right before the node **End** (see PHGS Γ_4 in Figure 5.7);
- to add before the node **End**, in all PHGSs implementing logic functions and macrooperations that never run in parallel, an operational node which contains for the PHGS Γ_i the special signal Z_i^-. This signal is used in order to return back to the previous hierarchical level. This node is not necessary in the two following cases. In the case of a PHGS implementing a macrooperation, the signal can be generated in an operational node containing only microoperations, situated right before the node **End** (see PHGS Γ_5 in Figure 5.7). In the case of a PHGS implementing a logic function, and if all the paths from the node **Begin** to the node **End** pass through an operational assignment node, the signal can be generated in all the assignment nodes (see PHGS Γ_6 in Figure 5.7);

- to add a group of nodes composed of one conditional node and two operational nodes, right before the node **End** in all PHGSs implementing macrooperations that run in parallel and that are invoked alone. The conditional node is to detect if the macrooperation Z_i is running in parallel in the stack i, or alone in any stack but i. The logic condition is given by the counter binary code of the sub-clock T_i. The two operational nodes are needed to generate the two special signals mentioned in the two previous items (see PHGS Γ_3 in Figure 5.7).

After applying these rules to the set of PHGSs presented in Figure 5.6, the extended set of PHGSs depicted in Figure 5.7 is obtained.

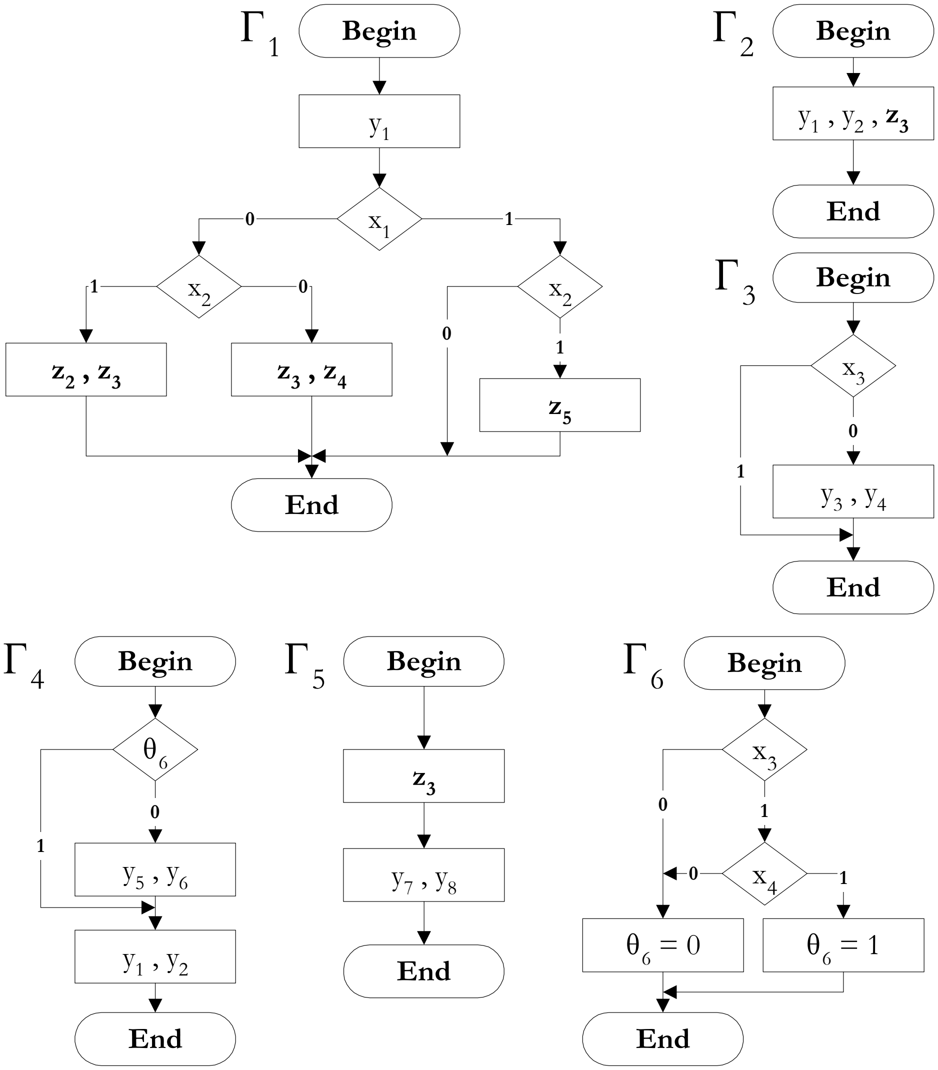


Figure 5.6 – A set of PHGSs for implementing in a PHFSM.

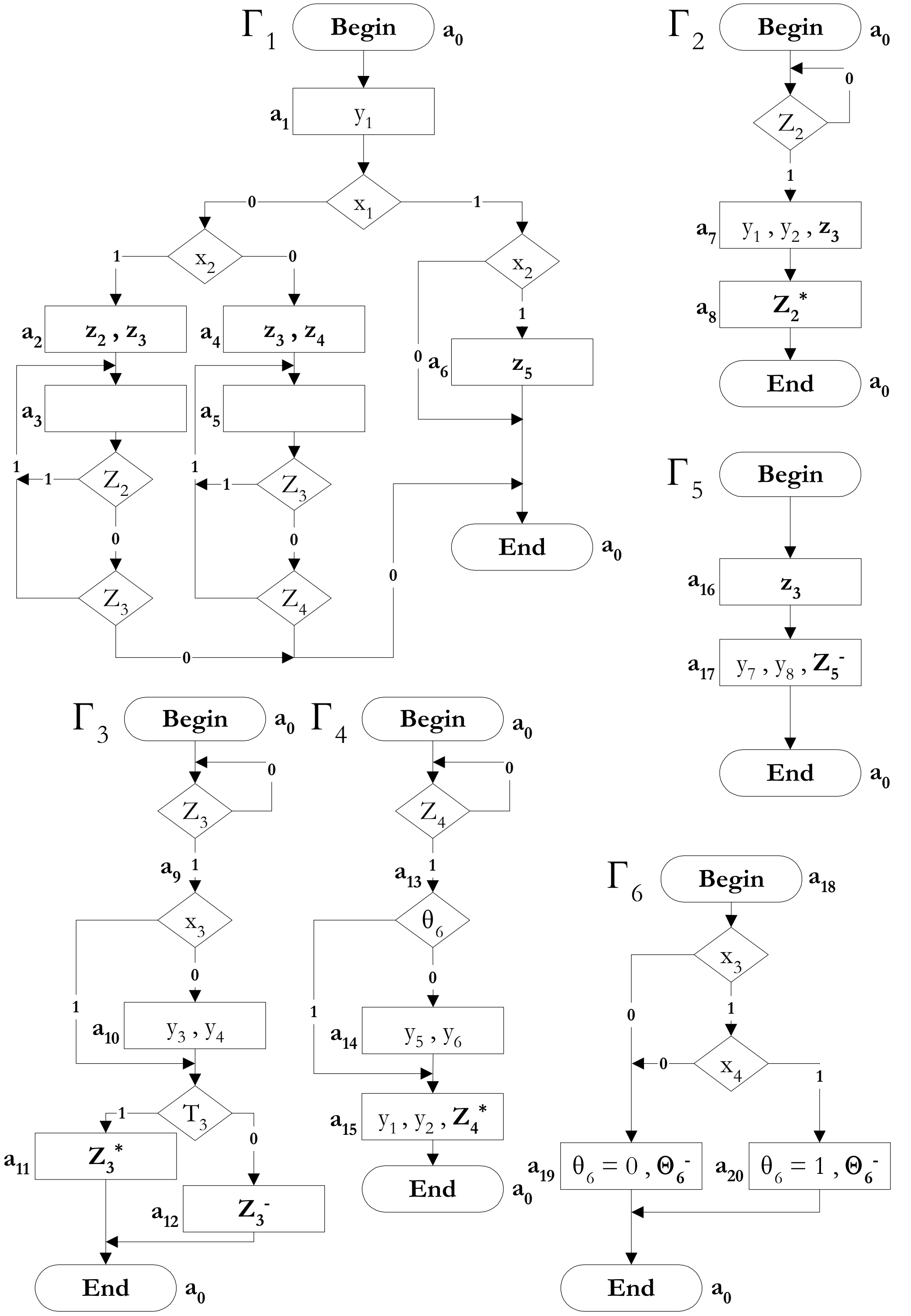


Figure 5.7 – A set of extended PHGSs marked for synthesis as a Moore PHFSM.

When a macrooperation z_i is invoked in parallel it will run in the stack i, during the sub-clock T_i. However, if it is invoked alone it will run in another stack but the stack i. Thus, the counter binary code cannot be used to identify the macrooperation (logic function) that is running.

Therefore, the set of extended PHGSs should be marked globally, applying the rules defined for the HFSM model 2 with the following differences:

- the label a_0 is assigned to the nodes **Begin** and **End** of all PHGSs that implement macrooperations that are invoked in parallel and to the node **End** of all remaining PHGSs (see Figure 5.7);
- a macrooperation z_i that is invoked in parallel and alone needs an entry state after the extra conditional node inserted for synchronisation purposes. If the node or input of the node, after the conditional node Z_i, is not marked yet it needs a label (see state a_9 in PHGS Γ_3 of Figure 5.7).

The construction of the extended state transition table is straightforward applying the rules of the HFSM model 2. But since the transitions from the state a_0 depend on the macrooperation, the state has to be prefixed with the macrooperation name. Table 5.24 represents the Moore extended transition table for the set of PHGSs depicted in Figure 5.7.

In order to transform the extended table to ordinary form, in the case of a Moore machine, the following actions must be performed:

- merge all transitions z_i a_0 in just one transition from the state a_0, by prefixing the logic condition, Z_i or $\overline{Z}_i$ in the column $X(a_m,a_s)$ by the counter binary code of the sub-clock T_i, denoted by T_i (see state a_0 in Table 5.25);
- replace each macrooperation that is invoked alone with its binary code $K(\varepsilon_v)$ and add the signal y^+ (see for example state a_6 in Table 5.25);
- replace each logic function θ_k with its binary code $K(\varepsilon_v)$ and add y^+ (see state a_{13} in Table 5.25). Apply the same rules defined for logic functions in the HFSM;
- replace each macrooperation z_i invoked in parallel, with the signal Z_i (see state a_2 in Table 5.25);
- replace all symbols θ_k=1 in the column $Y(a_m)$ with the signal Θ_k, and all symbols θ_k=0 with the signal Θ_k^* (see states a_{19} and a_{20} respectively in Table 5.25);
- replace all symbols θ_k and $\overline{\theta}_k$ in the column $X(a_m,a_s)$ with the signals Θ_k and $\overline{\Theta}_k$ respectively (see state a_{13} in Table 5.25);
- replace all symbols Z_i^- with the signal y^- (see state a_{12} in Table 5.25).

After applying these rules to Table 5.24, the ordinary state transition Table 5.25 is obtained.

Table 5.24 – PHFSM extended state transition table.

a_m { Y(a_m) }	a_s	X(a_m,a_s)
z_1 a_0	a_1	1
z_2 a_0	a_7 a_0	Z_2 $\overline{Z}_2$
z_3 a_0	a_9 a_0	Z_3 $\overline{Z}_3$
z_4 a_0	a_{13} a_0	Z_4 $\overline{Z}_4$
a_1 {y_1}	a_6 a_0 a_2 a_4	x_1 x_2 x_1 $\overline{x}_2$ $\overline{x}_1$ x_2 $\overline{x}_1$ $\overline{x}_2$
a_2 {z_2,z_3}	a_3	1
a_3	a_3 a_3 a_0	Z_2 $\overline{Z}_2$ Z_3 $\overline{Z}_2$ $\overline{Z}_3$
a_4 {z_3,z_4}	a_5	1
a_5	a_5 a_5 a_0	Z_3 $\overline{Z}_3$ Z_4 $\overline{Z}_3$ $\overline{Z}_4$
a_6 {z_5}	a_0	1
a_7 {y_1,y_2,z_3}	a_8	1
a_8 { Z_2^* }	a_0	1
a_9	a_{11} a_{12} a_{10}	x_3 T_3 x_3 $\overline{T}_3$ $\overline{x}_3$
a_{10} {y_3,y_4}	a_{11} a_{12}	T_3 $\overline{T}_3$
a_{11} { Z_3^* }	a_0	1
a_{12} { Z_3^- }	a_0	1
a_{13} {θ_6}	a_{15} a_{14}	θ_6 $\overline{\theta}_6$
a_{14} {y_5,y_6}	a_{15}	1
a_{15} {y_1,y_2, Z_4^* }	a_0	1
a_{16} {z_3}	a_{17}	1
a_{17} {y_7,y_8, Z_5^- }	a_0	1
a_{18}	a_{19} a_{19} a_{20}	$\overline{x}_3$ x_3 $\overline{x}_4$ x_3 x_4
a_{19} {θ_6=0, Θ_6^- }	a_0	1
a_{20} {θ_6=1, Θ_6^- }	a_0	1

Table 5.25 – PHFSM ordinary state transition table.

a_m { Y(a_m) }	a_s	X(a_m,a_s)
a_0	a_1 a_7 a_0 a_9 a_0 a_{13} a_0	T_1 T_2 Z_2 T_2 $\overline{Z}_2$ T_3 Z_3 T_3 $\overline{Z}_3$ T_4 Z_4 T_4 $\overline{Z}_4$
a_1 {y_1}	a_6 a_0 a_2 a_4	x_1 x_2 x_1 $\overline{x}_2$ $\overline{x}_1$ x_2 $\overline{x}_1$ $\overline{x}_2$
a_2 { Z_2 , Z_3 }	a_3	1
a_3	a_3 a_3 a_0	Z_2 $\overline{Z}_2$ Z_3 $\overline{Z}_2$ $\overline{Z}_3$
a_4 { Z_3 , Z_4 }	a_5	1
a_5	a_5 a_5 a_0	Z_3 $\overline{Z}_3$ Z_4 $\overline{Z}_3$ $\overline{Z}_4$
a_6 {yz_3,yz_1,y^+}	a_0	1
a_7 {y_1,y_2,yz_2,yz_1,y^+}	a_8	1
a_8 { Z_2^* }	a_0	1
a_9	a_{11} a_{12} a_{10}	x_3 T_3 x_3 $\overline{T}_3$ $\overline{x}_3$
a_{10} {y_3,y_4}	a_{11} a_{12}	T_3 $\overline{T}_3$
a_{11} { Z_3^* }	a_0	1
a_{12} {y^-}	a_0	1
a_{13} {yz_3,yz_2,y^+}	a_{15} a_{14}	Θ_6 $\overline{\Theta}_6$
a_{14} {y_5,y_6}	a_{15}	1
a_{15} {y_1,y_2, Z_4^* }	a_0	1
a_{16} {yz_2,yz_1,y^+}	a_{17}	1
a_{17} {y_7,y_8,y^-}	a_0	1
a_{18}	a_{19} a_{19} a_{20}	$\overline{x}_3$ x_3 $\overline{x}_4$ x_3 x_4
a_{19} { Θ_6^* ,y^-}	a_0	1
a_{20} { Θ_6 ,y^-}	a_0	1

In order to synthesise a PHFSM, it is necessary to create the entry state generation table of the **Code Converter** (see Table 5.26), that provides the entry state of each PHGS for the respective PHGS binary code $K(\varepsilon_v)$, in the same manner as for the HFSM model 2. However, it is necessary to keep in mind, that the entry state of the macrooperation z_i that is also invoked in parallel, is the first state after the conditional node Z_i (see states a_7, a_9 and a_{13} in Table 5.26).

Table 5.26 – Moore PHFSM Code Converter table.

PHGS Γ_v	$K(\varepsilon_v)$ $[yz_i]$	Initial State
	$\overline{yz_3}\ \overline{yz_2}\ \overline{yz_1}$	a_0
z_1	$\overline{yz_3}\ \overline{yz_2}\ yz_1$	a_1
z_2	$\overline{yz_3}\ yz_2\ \overline{yz_1}$	a_7
z_3	$\overline{yz_3}\ yz_2\ yz_1$	a_9
z_4	$yz_3\ \overline{yz_2}\ \overline{yz_1}$	a_{13}
z_5	$yz_3\ \overline{yz_2}\ yz_1$	a_{16}
θ_6	$yz_3\ yz_2\ \overline{yz_1}$	a_{18}
	$yz_3\ yz_2\ yz_1$	a_0

5.7 Automatic Synthesis of a HFSM

The first step of logic synthesis has a lot of rules according to the HFSM model chosen for implementation, so it will be useful to provide **automatic generation of state transition tables** that are correct by construction.

After the synthesis, extra states can be inserted in order to split transitions, extra states that can be removed afterwards. In both situations the synthesis methods are invoked automatically to synthesise again the new machine. In this situation the marking step must be skipped. To accomplish it, all synthesis methods have the parameter **statemark**. Its default value **1** forces the marking step to be performed.

The automatic synthesis of a HFSM is divided in 3 steps: marking the set of HGSs; generating the state transition table; generating the **Code Converter** table.

5.7.1 Marking a Hierarchical Algorithm

When marking a set of HGSs for synthesis as a Moore or a Mealy machine, it must be kept in mind that the HFSM model 3 demands marking each HGS separately. This detail however can be handled inside the marking **Graphscheme** method, without the need of two different methods for each machine.

5.7.1.1 Marking for synthesis as a Moore HFSM

Before marking the HGS **Cleanstategs()** cleans any previous marking. **Moorestatehgs()** starts to check if the set of HGS has any logic functions. In affirmative case the user can choose to synthesise them as Moore or as Mealy machines.

If **Moorestategs()** is invoked for marking a HGS for the HFSM model 2, the HGS initial state is the state after the last one used in the previous HGS, except for the main HGS that is 2. And the total number of states is stored in **hgsnstates**. For the HFSM model 3 the HGS initial state is always 2 and the number of states used to mark the HGS is stored in **gsnstates**.

The Moore marking rules depend on the type of the nodes. Exceptionally they can also depend on the following node or the previous nodes. So the nodes can be marked individually without the need to traverse the HGS. **Moorestatenode()** is invoked for all nodes of the HGS and applies the following marking rules:

- it is not necessary to mark a logic function node if the nodes that are in the back of it are nodes that hold only microoperations. But to take that decision is necessary to know the type of all the nodes that are pointing to a logic function node, so these nodes are left to the end. When marking a node that is pointing to a logic function, and according to its type, a flag is placed in the logic function node. The flag can have two values: ignore state **-2** and need state **-3** that has precedence over the ignore state. After marking all the other nodes, the logic function nodes are inspected and only those who have the flag need state are marked;

- if the HGS is the main HGS the nodes **Begin** and **End** are marked with the state 0, otherwise the node **Begin** is only marked if it is pointing to a conditional node with an input condition and the node **End** is marked with the state 1;

- all remaining operational nodes are marked. Any operational node that is pointing to a logic function node calls **Needstatenode()** to flag that the logic function needs a state, except the nodes that hold only microoperations that call **Ignorestatenode()** to flag that the logic function does not need a state;

- conditional nodes are not marked, but they also call **Needstatenode()** when they are pointing to a logic function node.

Setstatenode() marks the node with a state, if it is not marked yet, and increments the state number. It has a second parameter to indicate if the node is marked automatically or manually. The default value **1** means automatic mode.

5.7.1.2 Marking for synthesis as a Mealy HFSM

The Mealy machine method **Mealystatehgs()** is simplified when compared with the Moore machine, because all HGSs are marked for synthesis as Mealy machines using the method **Mealystategs()**.

In the Mealy machine the marking rules do not have exceptions. However, the states are assigned to the inputs of nodes instead of being assigned to the nodes, with the exception of the nodes **Begin** and **End**. Because some of the states are assigned to the inputs of nodes that follow certain types of nodes, those nodes are marked when the node in the back is processed. **Mealystatenode()** is invoked for all the nodes of the HGS and applies the following marking rules:

- if the HGS is the main HGS the nodes **Begin** and **End** are marked with the state 0, otherwise the node **Begin** is not marked and the node **End** is marked with the state 1;
- the input of a node that holds a macrooperation is marked;
- the input of a node following an operational node is marked;
- the input of a node that holds a logic function is marked, and the inputs of the two nodes that follow it are marked.

5.7.2 Constructing a State Transition Table

To generate a state transition table it is necessary to record all transitions between states. Because states a_0 and a_1 have a special meaning they are handled at the HGS level according to their kind, model 2 or model 3, and they are not processed at the node level.

The algorithm is basically the same for both Moore and Mealy machines. However, it is simpler for Moore machines because the output function (microoperations, macrooperations and logic functions) is associated with the state. While, for Mealy machines it is associated with a state transition and it must be taken into account that it is always preferable to record a transition that passes through one operational node. Each node marked with a state, other than a_0 or a_1, is a starting state and must be written in the present state column. Then the HGS is traversed from that node in order to explore all the paths, and to record the respective transition condition, that will lead to another node marked with a state. This arriving state must be written in the next state column. Because a starting state can also be an arriving state, the algorithm must distinguish both situations.

When exploring a path, the method can reach a cycle containing only conditional nodes (see Figure 5.8). In a situation like this, not only the method must prevent infinite recursion, but also to stop the search of an arriving state. The arriving state for paths in loop is the starting state like for example $a_k \; \overline{x}_1 \; a_k$ in Figure 5.8.

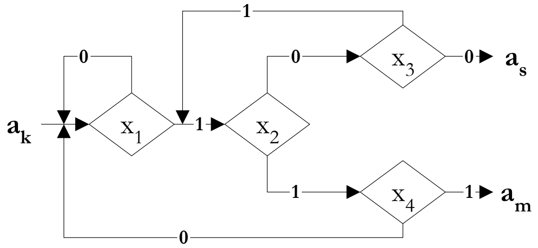


Figure 5.8 – A cycle of conditional nodes.

To detect a loop situation it is sufficient to test every time a conditional node is reached, if the node condition is already in the condition transition. For the above example the state transitions that must be recorded are $a_k\ x_1x_2x_4\ a_m$, $a_k\ x_1x_2\overline{x}_4\ a_k$, $a_k\ x_1\overline{x}_2x_3\ a_k$, $a_k\ x_1\overline{x}_2\overline{x}_3\ a_s$ and $a_k\ \overline{x}_1\ a_k$.

5.7.2.1 Constructing a Moore State Transition Table

Because the HFSM models 2 and 3 have different state transition tables they are generated by different methods.

In the HFSM model 2, **Mooretablemergehgs()** writes the table head and the first two lines. In the first line it is written the state a_0 and the main macrooperation name, and in the second line the state a_1 and the special signal y^-. Then **Mooretablemergegs()** is invoked to construct each HGS part of the table.

In the HFSM model 3, **Mooretablesplithgs()** just writes the table head and invokes **Mooretablesplitgs()** for each HGS. **Mooretablesplitgs()** is responsible for writing the first three lines of the HGS table. The first line is the name of the HGS and the other two are the state transitions for the states a_0 and a_1.

The nodes are handled with **Mooretablenode()** for both models. It is invoked for all the nodes marked with a state, and it must accommodate the difference between the two models. In model 3 the macrooperations, logic functions and the special signals y^+ and y^- are not written in the state transition table, but in the **Code Converter** table instead. Therefore the method has a parameter to distinguish the model in question. The HGS is traversed recursively by the derived classes methods but the table lines are written by the base class method.

The **Onode** method must accommodate the Mealy output when a logic function HGS is marked for synthesis as a Mealy machine. For this purpose, when an unmarked assignment node is crossed the variable **extray** is set. If an operational node is pointing to an unmarked logic function node, in the case of the HFSM model 2, it is necessary to construct the string **logicf**, with the logic function name and the special signal y^+, to be written as an extra output function of this node. But it only will be written in the first column of the next line (see state a_6 in Figure 5.9).

The **Cnode** method must traverse through the conditional node to search for an arriving state. First it makes a copy of the entry transition condition. If the node condition is already listed on the entry transition condition, the node is in a cycle of conditional nodes and the transition is of the type $a_sX(a_s,a_s)a_s$. On the contrary, if it is the first time that this node is reached, the node condition in its asserted form is added to the transition condition and the method exits the node through the true output to search for an arriving state. When the method returns back after exploring the true output path, it recovers the entry transition condition and the condition in its negated form is added to the transition condition, and the false output path is explored. Finally, when the method returns back, it recovers the entry transition condition. To mean a negated condition the character ~ is prefixed to the condition (see Figure 5.9). If the conditional node contains a logic function, the label **xextra** is used instead of the logic function name.

The **Node** base class method writes the table lines in parts by successive invocations. The starting state and the output function are written in the first column of the table. The output function depends on the node type:

- no output function for the node **Begin**, or a conditional node with an input condition, or an assignment node of a logic function HGS marked for synthesis as a Mealy machine;
- **yextra** for an assignment node with an assign value of 1 if the logic function HGS is marked for synthesis as a Moore machine;
- microoperations for operational nodes without macrooperations and **logicf**;
- microoperations if any and in the case of model 2 the macrooperation name and the extra signal y^+ for operational nodes with a macrooperation;
- in the case of model 2 the logic function name and the extra signal y^+ for a marked logic function node.

The arriving state is written in the second column of the table. After the first transition it is necessary to write spaces to pass over the first column, except if **logicf** is not a null string. The transition condition **transition** is written in the third column and if **extray** is set, the Mealy output of a mixed Moore/Mealy machine **yextra** is written in the fourth column (see state a_{18} in Figure 5.9).

The state transition table for the mixed Moore/Mealy HFSM model 2 generated by **SIMULHGS** is depicted in Figure 5.9. It is equivalent to Table 5.21 with the difference that the macrooperation (logic function) name is used instead of the extra outputs yz_i.

The RE_1 state transition table for the mixed Moore/Mealy HFSM model 3 generated by **SIMULHGS** is depicted in Figure 5.10 (compare it with Table 5.4).

```
---------------------------------------------------------
|           HGS Moore State Transition Table            |
---------------------------------------------------------
|      am , Y(am)      | as |     X(am,as)     |Y(am,as)|
---------------------------------------------------------
|a0  , z1              |a0  | 1                | --     |
---------------------------------------------------------
|a1  , y-              |a0  | 1                | --     |
---------------------------------------------------------
|a2  , y2              |a6  | x1x2             | --     |
|                      |a7  | x1~x2            | --     |
|                      |a3  | ~x1x2            | --     |
|                      |a4  | ~x1~x2           | --     |
---------------------------------------------------------
|a3  , y3,y5,z2 , y+   |a5  | 1                | --     |
---------------------------------------------------------
|a4  , z3 , y+         |a5  | 1                | --     |
---------------------------------------------------------
|a5  , y1,y4           |a0  | 1                | --     |
---------------------------------------------------------
|a6  , y6,y7,y8        |a8  | xextra           | --     |
|f6 , y+               |a5  | ~xextra          | --     |
---------------------------------------------------------
|a7  , z5 , y+         |a8  | 1                | --     |
---------------------------------------------------------
|a8  , y3              |a0  | 1                | --     |
---------------------------------------------------------
|a9  , y3,y4           |a10 | 1                | --     |
---------------------------------------------------------
|a10 , y7              |a10 | x1               | --     |
|                      |a11 | ~x1              | --     |
---------------------------------------------------------
|a11 , z4 , y+         |a12 | 1                | --     |
---------------------------------------------------------
|a12 , f6 , y+         |a1  | xextra           | --     |
|                      |a9  | ~xextra          | --     |
---------------------------------------------------------
|a13 , --              |a14 | x4               | --     |
|                      |a15 | ~x4              | --     |
---------------------------------------------------------
|a14 , y3,y5           |a15 | x5               | --     |
|                      |a1  | ~x5              | --     |
---------------------------------------------------------
|a15 , y1              |a1  | 1                | --     |
---------------------------------------------------------
|a16 , z3 , y+         |a17 | 1                | --     |
---------------------------------------------------------
|a17 , y1,y2           |a1  | x2               | --     |
|                      |a16 | ~x2              | --     |
---------------------------------------------------------
|a18 , --              |a1  | x3               | yextra |
|                      |a1  | ~x3              | --     |
---------------------------------------------------------
```

Figure 5.9 – State transition table generated by SIMULHGS for the mixed Moore/Mealy HFSM model 2.

```
----------------------------------------------------------
|           HGS Moore State Transition Table             |
----------------------------------------------------------
|      am , Y(am)       | as |     X(am,as)     |Y(am,as)|
----------------------------------------------------------
|                           z1                           |
----------------------------------------------------------
|a0  , --               |a2  | 1                | --     |
----------------------------------------------------------
|a1  , --               |a2  | 1                | --     |
----------------------------------------------------------
|a2  , y2               |a6  | x1x2             | --     |
|                       |a7  | x1~x2            | --     |
|                       |a3  | ~x1x2            | --     |
|                       |a4  | ~x1~x2           | --     |
----------------------------------------------------------
|a3  , y3,y5            |a5  | 1                | --     |
----------------------------------------------------------
|a4  , --               |a5  | 1                | --     |
----------------------------------------------------------
|a5  , y1,y4            |a0  | 1                | --     |
----------------------------------------------------------
|a6  , y6,y7,y8         |a8  | xextra           | --     |
|                       |a5  | ~xextra          | --     |
----------------------------------------------------------
|a7  , --               |a8  | 1                | --     |
----------------------------------------------------------
|a8  , y3               |a0  | 1                | --     |
----------------------------------------------------------
```

Figure 5.10 – RE_1 state transition table generated by SIMULHGS for the mixed Moore/Mealy HFSM model 3.

5.7.2.2 Constructing a Mealy State Transition Table

The details of the Moore HFSM models 2 and 3 are applicable to the Mealy machine. At the graph-scheme level the differences are related to the table appearance, specifically the column where the output function is displayed. For the HFSM model 2 there are the methods **Mealytablemergehgs()** and **Mealytablemergegs()** and for the HFSM model 3 there are the methods **Mealytablesplithgs()** and **Mealytablesplitgs()**.

At the node level **Mealytablenode()** also accommodates the difference between models 2 and 3, but it works differently from the Moore method. In Mealy the output function is stored in the string **output** when an operational node is crossed and it is written in the last table column. It is always preferable to use one transition of the kind $a_sX(a_s,a_a)Y(a_s,a_a)a_a$ instead of the kind $a_sX(a_s,a_a)a_a$. For that purpose a transition does not stop in the state at the input of an operational node with only microoperations but in the state at the input of the next node. Unless the operational node with the microoperations follows an operational node with a macrooperation or a conditional node with a logic function.

In order to obtain this behaviour the nodes of the types mentioned above are counted and the transition path stops in a state only after one of those nodes have been traversed.

Figure 5.11 presents the situations mentioned above. The transition $a_m\ x_1\ a_n$ is ignored because the state a_n is at the input of the operational node y_6,y_9 and the transition did not crossed any operational node yet. So the transition $a_m\ x_1\ y_6 y_9\ a_o$ is recorded instead. But the transition $a_q\ z_2\ a_n$ stops at the state a_n because it already crossed the operational node z_2 and the transition $a_o\ \theta_3\ a_p$ stops at the state a_p because it crossed the conditional node θ_3.

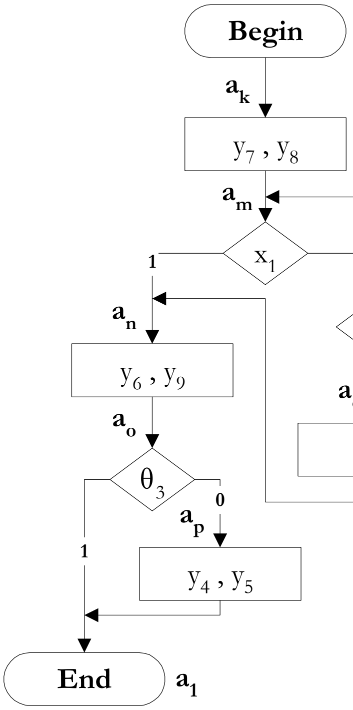


Figure 5.11 – A HGS marked for synthesis as a Mealy machine.

The **Onode** method processes all marked operational nodes. When an operational node is crossed the output function is constructed according to the node type:

- no output function for the node **Begin**;
- **yextra** for an assignment node that assigns the value 1;
- microoperations for operational nodes without macrooperations;
- microoperations if any and in the case of model 2 the macrooperation name and the extra signal y^+ for operational nodes with a macrooperation.

The counter **micro** is incremented if an operational node with microoperations or macrooperations is traversed. The former in order to stop only after it, the latter in order to stop at the input of the node that follows it. The method is then invoked for the next node, to search for an arriving state.

The **Cnode** method works as for the Moore method, but it also constructs the output function of a conditional node that contains a logic function, the function name and the extra signal y^+, in the case of the HFSM model 2.

```
-----------------------------------------------
|      HGS Mealy State Transition Table       |
-----------------------------------------------
| am | as |     X(am,as)      |    Y(am,as)     |
-----------------------------------------------
|a0  |a0  | 1                 | z1              |
-----------------------------------------------
|a1  |a0  | 1                 | y-              |
-----------------------------------------------
|a2  |a3  | 1                 | y2              |
-----------------------------------------------
|a3  |a7  | x1x2              | y6,y7,y8        |
|    |a8  | x1~x2             | --              |
|    |a4  | ~x1x2             | --              |
|    |a6  | ~x1~x2            | --              |
-----------------------------------------------
|a4  |a5  | 1                 | y3,y5,z2 , y+   |
-----------------------------------------------
|a6  |a5  | 1                 | z3 , y+         |
-----------------------------------------------
|a5  |a0  | 1                 | y1,y4           |
-----------------------------------------------
|a8  |a9  | 1                 | z5 , y+         |
-----------------------------------------------
|a7  |a9  | xextra            | f6 , y+         |
|    |a5  | ~xextra           | f6 , y+         |
-----------------------------------------------
|a9  |a0  | 1                 | y3              |
-----------------------------------------------
|a10 |a11 | 1                 | y3,y4           |
-----------------------------------------------
|a11 |a12 | 1                 | y7              |
-----------------------------------------------
|a12 |a12 | x1                | y7              |
|    |a13 | ~x1               | --              |
-----------------------------------------------
|a13 |a14 | 1                 | z4 , y+         |
-----------------------------------------------
|a14 |a1  | xextra            | f6 , y+         |
|    |a10 | ~xextra           | f6 , y+         |
-----------------------------------------------
|a15 |a16 | x4                | y3,y5           |
|    |a1  | ~x4               | y1              |
-----------------------------------------------
|a16 |a1  | x5                | y1              |
|    |a1  | ~x5               | --              |
-----------------------------------------------
|a17 |a18 | 1                 | z3 , y+         |
-----------------------------------------------
|a18 |a19 | 1                 | y1,y2           |
-----------------------------------------------
|a19 |a1  | x2                | --              |
|    |a17 | ~x2               | --              |
-----------------------------------------------
|a20 |a1  | x3                | yextra          |
|    |a1  | ~x3               | --              |
-----------------------------------------------
```

Figure 5.12 – State transition table generated by SIMULHGS for the Mealy HFSM model 2.

The **Node** base class method writes the table lines in parts by successive invocations. The starting state is written in the first column of the table and the arriving state in the second column. After the first transition it is necessary to write spaces to pass over the first column. The transition condition **transition** and the output function **output** are written in the third and the fourth column of the table respectively. The state transition table for the Mealy HFSM model 2 generated by **SIMULHGS** is depicted in Figure 5.12 (compare it with Table 5.13).

5.7.3 Constructing a Code Converter Programming Table

CCmergehgs() builds the **Code Converter** table for the HFSM model 2. It is a table with the entry state of each HGS of the algorithm. The entry state of a HGS is obtained invoking **Nodeinitialstategs()** and it is the first node marked with a state. It also presents the HGS binary code K(ε_v) and its representation through the extra outputs yz$_i$. Instead of replacing each macrooperation (logic function) name in the state transition table with the proper extra outputs yz$_i$, they are presented in this table.

It must be kept in mind that for the binary code with all bits set to zero (code K(ε_v) that is used in order to clean the **Code Converter** outputs), and for all remaining binary codes not in use, the **Code Converter** will output the state a_0.

The **Code Converter** table for the mixed Moore/Mealy HFSM model 2 generated by **SIMULHGS** is depicted in Figure 5.13 (compare it with Table 5.3).

```
---------------------------------------------------------
|               Code Converter Programming              |
---------------------------------------------------------
|  | Zi/Fi | Zi/Fi |   Code Converter   |     Zi/Fi     |
|  | Name  | Code  |     inputs YZi     | Initial state |
---------------------------------------------------------
|0 |       | 000   |                    | a0            |
---------------------------------------------------------
|1 | z1    | 001   | yz1                | a2            |
---------------------------------------------------------
|2 | z2    | 010   | yz2                | a9            |
---------------------------------------------------------
|3 | z3    | 011   | yz2 yz1            | a13           |
---------------------------------------------------------
|4 | z4    | 100   | yz3                | a16           |
---------------------------------------------------------
|5 | z5    | 101   | yz3 yz1            | a1            |
---------------------------------------------------------
|6 | f6    | 110   | yz3 yz2            | a18           |
---------------------------------------------------------
|7 |       | 111   |                    | a0            |
---------------------------------------------------------
```

Figure 5.13 – Code Converter table generated by SIMULHGS for the mixed Moore/Mealy HFSM model 2.

The **Code Converter** table of the HFSM model 3 is a table with the next HGS and the extra signals used to control the stacks, for each state of each HGS. In a state where the macrooperation z_v, or the logic function θ_v, is invoked the table presents the HGS Γ_v name and the extra signal y^+. In all a_1 states it presents the extra signal y^-. For all other states it presents only the active HGS. **CCsplithgs()** builds the head of the table and **CCsplitgs()** builds each HGS part of the table.

If the HGS is not the main HGS, its node **End** is marked with the state a_1 and that will be the first line of the sub-table. Each node that contains a macrooperation or a logic function or a node with microoperations that is pointing to an unmarked logic function is processed. The state and the macrooperation (logic function) name and the extra signal y^+ are written. A final line with the word **other** in the state column and the HGS name is written.

The **Code Converter** table for the Mealy HFSM model 3 generated by **SIMULHGS** is depicted in Figure 5.14 (compare it with Table 5.20).

```
-------------------------------------------
|       Code Converter Programming        |
-------------------------------------------
| Active | HGS  | State | Next |  Stack   |
|   HGS  | Code |       | HGS  | Pointer  |
-------------------------------------------
| z1     | 000  | a4    | z2   | y+       |
|        |      | a6    | z3   | y+       |
|        |      | a8    | z5   | y+       |
|        |      | a7    | f6   | y+       |
|        |      | other | z1   |          |
-------------------------------------------
| z2     | 001  | a1    | --   |       y- |
|        |      | a5    | z4   | y+       |
|        |      | a6    | f6   | y+       |
|        |      | other | z2   |          |
-------------------------------------------
| z3     | 010  | a1    | --   |       y- |
|        |      | other | z3   |          |
-------------------------------------------
| z4     | 011  | a1    | --   |       y- |
|        |      | a2    | z3   | y+       |
|        |      | other | z4   |          |
-------------------------------------------
| z5     | 100  | a1    | --   |       y- |
|        |      | other | z5   |          |
-------------------------------------------
| f6     | 101  | a1    | --   |       y- |
|        |      | other | f6   |          |
-------------------------------------------
```

Figure 5.14 – Code Converter table generated by SIMULHGS for the Mealy HFSM model 3.

5.8 Conclusions

If a hierarchical algorithm is specified with many macrooperations the Moore machine is more efficient in terms of the number of states than the Mealy machine. This is due to the fact that an operational node of a HGS that contains microoperations and a macrooperation needs only one state in the Moore machine, against two states for Mealy. But if a hierarchical algorithm is specified with many logic functions, and even considering that the Moore marking rules only need a state against three states for Mealy, the latter needs less states to mark the logic function HGSs and consequently takes less time to evaluate them.

Therefore, if a hierarchical algorithm is specified with many macrooperations and logic functions, as against an ordinary FSM, the Moore HFSM is more efficient than the Mealy HFSM. Not only, it normally has fewer states but it also has more intuitive marking rules. Moreover, is less erroneous to record the transitions between states. But, on the other hand the Mealy HFSM speeds up the evaluation of logic functions. As a conclusion it can be said that the best model to implement a hierarchical algorithm is the proposed mixed Moore/Mealy HFSM.

Since in a complex parallel hierarchical algorithm it can be very difficult to predict the present state of the machine, the logic synthesis of the proposed PHFSM is very complex and demands a deep knowledge of the algorithm, particularly when it is necessary to construct the extended PHGSs. In an operational node that invokes macrooperations in parallel, it can be hard to decide which of them will run in its own stack and the ones that are candidates for a hierarchical run. The decision depends on the flow of the algorithm and on the present state (running or passive) of each macrooperation. A problem can also arise in the invocation of the same logic function in more than one macrooperation that can eventually run in parallel, since it is necessary to ensure that a logic function is never invoked before the calculated value of a previous execution has already been used. As a conclusion it can be said that the synthesis of a PHFSM is very complex.

The tool **SIMULHGS** implements automatically the first step of logic synthesis, i.e. state marking and state transition recording. It generates the state transition table and the **Code Converter** table for Moore, Mealy and mixed Moore/Mealy HFSMs, for both models 2 and 3.

6 IMPLEMENTATION AND OPTIMISATION OF HIERARCHICAL FINITE STATE MACHINES

Summary

The goal of this chapter is to present an implementation of the HFSM and PHFSM models that can provide such facilities as flexibility, extensibility and reusability and to discuss some optimisation techniques that will improve the performance of the FSMs for the target technology chosen.

The chapter starts by presenting the internal decomposition of the **Combinational Scheme** for a **RAM**-based implementation and then describes some optimisation techniques. The **replacement of input variables** technique proposed in [Baranov79] and a **special state encoding algorithm** proposed in [Sklyarov96] that embodies the transition condition in the state code in order to eliminate the input variables from the next state generation are explained. Furthermore, it is shown how the **state splitting** technique can keep the replacement of the input variables working with a small set of new variables and how it can help to apply the special state encoding algorithm. Finally, the method of the tool **SIMULHGS** that automatically applies the state splitting technique to an already synthesised algorithm is presented.

6.1 Introduction

The **Combinational Scheme** and the **Code Converter** of the HFSM and PHFSM models can be implemented with simple logic gates, a programmable logic device of type PAL/PLA, a ROM or the read/write equivalent device **RAM**, or sophisticated field-programmable devices such as **FPGA**s. Since it is necessary to provide such new facilities as flexibility, extensibility and reusability, i.e. to provide a dynamically reconfigurable device, the choice can be either to use a RAM or a FPGA.

A RAM can be easily simulated in a hardware description language since it is basically a **lookup table** (**LUT**). A RAM (ROM) with m inputs and n outputs can implement n different logic functions of m logic variables, because every minterm is provided. It is useful in the following situations [Bolton90]:

- where the problem is naturally specified with a truth table which is mapped directly into the words of the RAM and when the function needs to be modified word by word;
- when a universal, rewriteable logic block is required and the minterms demand cannot be predicted.

Moreover, since only the number of inputs and outputs determines the complexity of a RAM, the kind of RAM that is needed is the same regardless of the state assignment [Katz94]. However, a RAM (ROM) is suitable to implement functions which are fully specified and involve relatively few variables, such as simple code conversions where one binary pattern acting as the address is converted into another pattern found at the address location [Green86]. Since not every combination of input variables and present state is meaningful and they do not occur in a HGS description, the RAM (ROM) based approach may not be an attractive implementation. Unless an optimisation technique that will select the appropriate input variables for each state will be applied.

Despite all the advantages of a RAM-based implementation, since the appearance of the FPGAs it does not make any sense to implement complex digital systems in any other platform. Therefore, the FPGA should be the target technology for the HFSM and PHFSM models with the extra advantage of allowing the complete implementation of the models.

However, in order to validate the proposed models in a FPGA platform, it is necessary to have a development system for FPGAs. The alternative is to create a complete VHDL model of a particular FPGA architecture and to programme each experimental example manually in the model. Since, this procedure was time consuming and error prone it was not pursued. But, given that some FPGAs are internally implemented with LUTs, a RAM-based approach can be considered as the first step to validate the proposed models for a LUT-based technology.

6.2 Decomposition of the Combinational Scheme

The **Combinational Scheme** depicted in Figure 6.1 is the component where the next state δ and the output λ functions are implemented using RAMs as LUTs.

With the external inputs supplied by the datapath $x_L,...,x_1$ and the present state $\tau_R,...,\tau_1$ stored in the **Stack Memory**, it generates the external outputs (microoperations) $y_N,...,y_1$, the next state $CSD_R,\ldots,CSD_1$, the binary codes of the macrooperations and logic functions $yz_k,...,yz_1$ and the special signals y^+ and y^- that respectively increment and decrement the **Stack Memory** pointer.

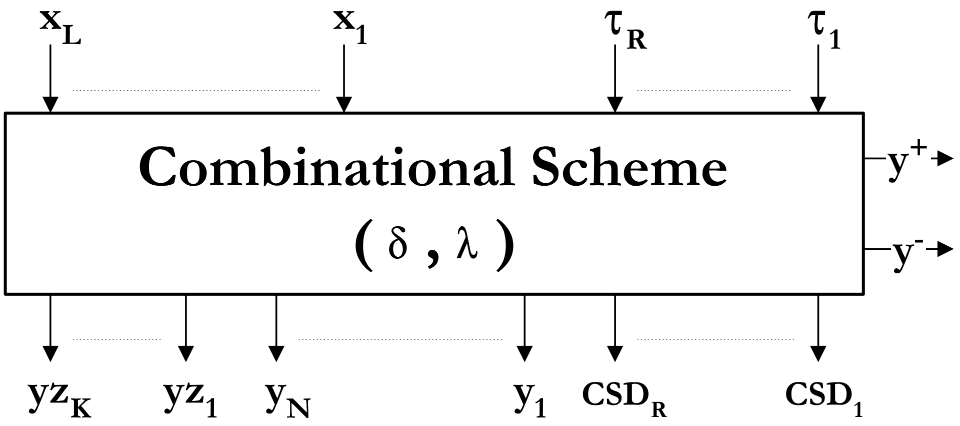


Figure 6.1 – Combinational Scheme.

As it was mentioned before in Chapter 4, the **Combinational Scheme** is not a pure combinatorial block and in accordance with the kind of machine, Moore or Mealy it has respectively one or two blocks to generate the next state and the output functions (see Figure 6.2 and Figure 6.3).

6.2.1 Hierarchical FSM

The **Combinational Scheme** of the Moore HFSM model 2, depicted in Figure 6.2, is composed of two RAMs and three registers.

The **Next State Memory** implements the next state function δ. Since a HGS describing a logic function can be marked for synthesis as a Mealy machine in the case of the mixed Moore/Mealy HFSM, it also generates the value of the extra output variable **extra_y**, used to carry out the calculated value of a logic function. This value is tested through the extra input variable **extra_x**, in order to generate the appropriate next state.

Since Moore outputs ($Y_{Moore}=\lambda_{Moore}[A]$) are dependent only on the present state, all outputs (microoperations, binary codes of macrooperations and logic functions and the special signals that increment and decrement the stack) are generated on the **Output Memory**.

The **Input Register** and the **Output Register** are used to fix respectively the external input variables (logic conditions) and the external output variables (microoperations), i. e. the signals that communicate with the datapath.

The 1-bit **Extra Register** is used to store and fix the calculated value of a logic function. Only one register is needed because the calculated value of the logic function is used at the next clock cycle after being calculated and before another logic function is invoked.

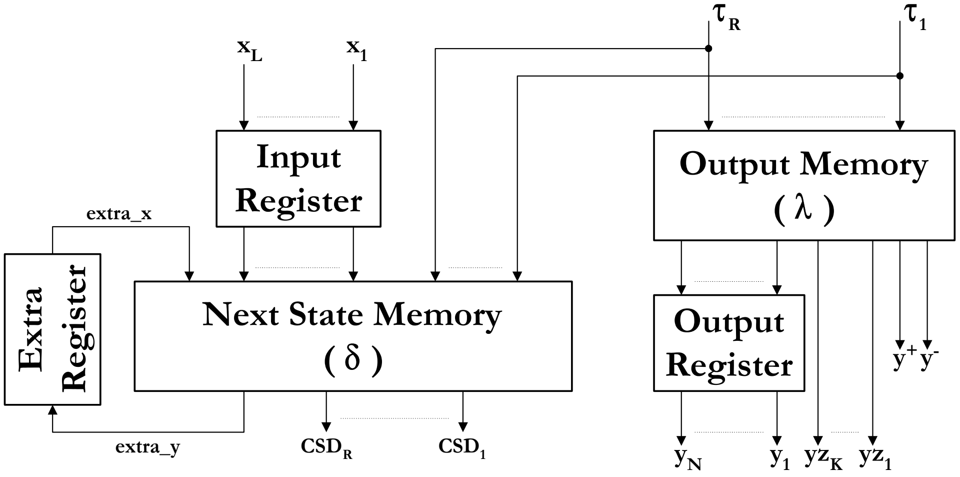


Figure 6.2 – Decomposition of the Moore HFSM Combinational Scheme for a RAM-based implementation.

Since Mealy outputs ($Y_{Mealy}=\lambda_{Mealy}[A,X]$) are dependent on both the present state and the external inputs, they are generated on the same component that generates the next state. Therefore, the Mealy **Combinational Scheme** depicted in Figure 6.3, as only one RAM component, that is the **Next State and Output Memory**. The three registers have the same purpose as in the Moore machine.

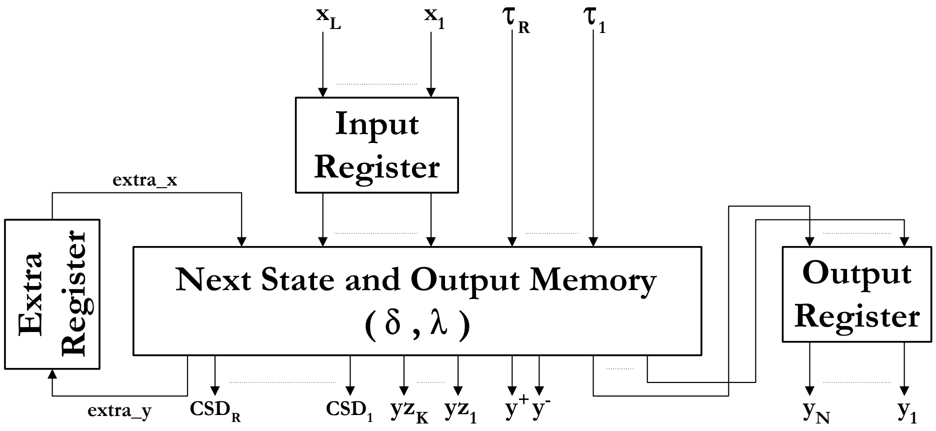


Figure 6.3 – Decomposition of the Mealy HFSM Combinational Scheme for a RAM-based implementation.

6.2.2 Parallel FSM/HFSM

The **Combinational Scheme** of the parallel machines is similar to the hierarchical non-parallel machines. It was mentioned before that a set of SR flip-flops is used to store the logic conditions, that hold the information about the running state of the macrooperations and that are used for synchronisation purposes. There are J-1 flip-flops, where J is the number of memory registers of the PFSM (parameter V in Figure 4.7) or the number of stacks of the PHFSM (parameter Q in Figure 4.8).

Since, it is also possible to have logic functions running in parallel, it is necessary one storage element for each logic function. And since, the calculated value is not used immediately, it must be stored until it is needed, that is until the next logic function call. The easiest way is using a SR flip-flop. The flip-flop is set for returning the logic value 1 and reset for returning the logic value 0.

Figure 6.4 depicts the **Combinational Scheme** of the Moore PHFSM, where the SR flip-flops used for storing the calculated value of logic functions are omitted.

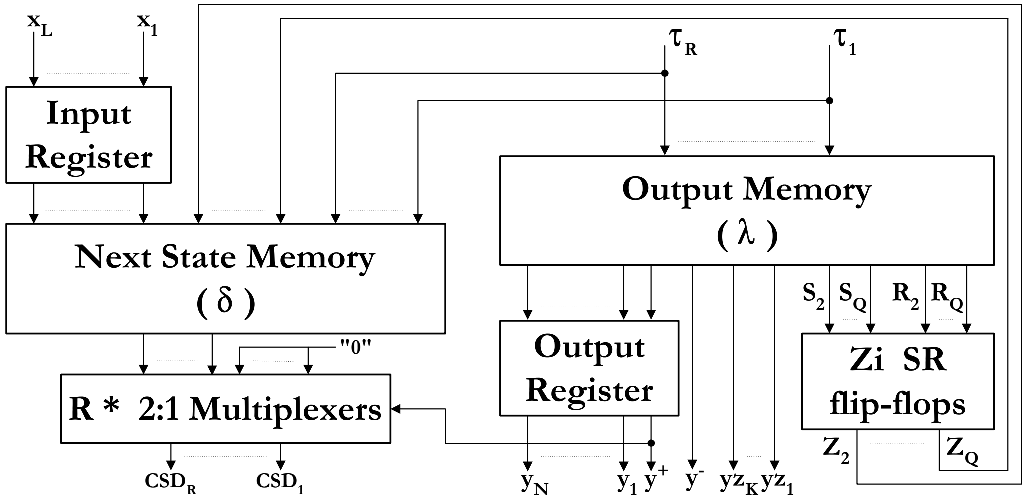


Figure 6.4 – Decomposition of the Moore PHFSM Combinational Scheme for a RAM-based implementation.

Like it was explained before in the synchronisation of the PHFSM paragraph, a multiplexer is required before the lines D_R,...,D_1 exit the **Combinational Scheme** in order to let pass the entry state of the PHGS generated by the **Code Converter** when the signal y+ is active. It has the following behaviour: when y^+ is negated the output state will be the calculated next state, while when y^+ is asserted the output state will be a_0.

6.3 Replacement of Input Variables

In order to decrease the amount of memory needed to implement the FSMs, the number of address lines of the RAM that generates the next state must be minimised. Let's consider the replacement of input variables proposed in [Baranov79]* for the ordinary state transition table of the mixed Moore/Mealy HFSM presented in Table 5.21.

In practice, the set of input variables involved in all transitions from a state a_m $X(a_m)=\{x_{m1},...,x_{mK}\}$ is much smaller than the total set $X=\{x_1,...,x_L\}$. Let's define $G=\max|\#X(a_m)|$ where $m=1,...,M$ as the maximum number of input variables involved in all transitions from a state a_m. Since, commonly $G<<L$ in a HGS description, a transformation of input variables will decrease the number of variables involved in the next state generation. Hence, a new set of variables $P=\{p_1,...,p_G\}$ is built, and for each state a_m a one to one correspondence between the set $X=\{x_1,...,x_L\}$ and the leftmost variables of the new set $P=\{p_1,...,p_G\}$ is defined (see column $P(a_m,a_s)$ in Table 6.1).

In this example G=2 due to state a_2 in Table 6.1. For this state a correspondence is made between $\{x_1,x_2\}$ and $\{p_1,p_2\}$. Now let's consider a state whose transitions depend on only one variable x_i, for example, all transitions from the state a_{13} are caused by x_4. In this case, the left most variable in the set P, that is p_1, is used. The variable x_4 must be replaced with p_1 in such a way that, if the transition is caused by x_4 it is caused by p_1 and if the transition is caused by $\bar{x}_4$ it is caused by $\bar{p}_1$. For the states with the transition condition equal to 1, it is not necessary to generate any variable of the set P (see for example state a_0 in Table 6.1).

Table 6.1 – Ordinary state transition table with the replacement of input variables.

a_m $\{Y(a_m)/Y(a_m,a_s)\}$	a_s	$X(a_m,a_s)$	$P(a_m,a_s)$
a_0 $\{yz_1\}$	a_0	1	
a_1 $\{y^-\}$	a_0	1	
a_2 $\{y_2\}$	a_6	$x_1\ x_2$	$p_1\ p_2$
	a_7	$x_1\ \bar{x}_2$	$p_1\ \bar{p}_2$
	a_3	$\bar{x}_1\ x_2$	$\bar{p}_1\ p_2$
	a_4	$\bar{x}_1\ \bar{x}_2$	$\bar{p}_1\ \bar{p}_2$
a_3 $\{y_3,y_5,yz_2,y^+\}$	a_5	1	
a_4 $\{yz_2,yz_1,y^+\}$	a_5	1	
a_5 $\{y_1,y_4\}$	a_0	1	
a_6 $\{y_6,y_7,y_8,yz_3,yz_2,y^+\}$	a_8	$extra\_x$	p_1
	a_5	$\overline{extra\_x}$	$\bar{p}_1$
a_7 $\{yz_3,yz_1,y^+\}$	a_8	1	
a_8 $\{y_3\}$	a_0	1	
a_9 $\{y_3,y_4\}$	a_{10}	1	
a_{10} $\{y_7\}$	a_{10}	x_1	p_1
	a_{11}	$\bar{x}_1$	$\bar{p}_1$
a_{11} $\{yz_3,y^+\}$	a_{12}	1	
a_{12} $\{yz_3,yz_2,y^+\}$	a_{11}	$extra\_x$	p_1
	a_9	$\overline{extra\_x}$	$\bar{p}_1$
a_{13}	a_{14}	x_4	p_1
	a_{15}	$\bar{x}_4$	$\bar{p}_1$
a_{14} $\{y_3,y_5\}$	a_{15}	x_5	p_1
	a_1	$\bar{x}_5$	$\bar{p}_1$
a_{15} $\{y_1\}$	a_1	1	
a_{16} $\{yz_2,yz_1,y^+\}$	a_{17}	1	
a_{17} $\{y_1,y_2\}$	a_1	x_2	p_1
	a_{16}	$\bar{x}_2$	$\bar{p}_1$
a_{18} $\{extra\_y\}$	a_1	x_3	p_1
	a_1	$\bar{x}_3$	$\bar{p}_1$

* A similar technique is proposed in [Green86] for the ROM-based implementation of ASM designs.

The Boolean functions for the new set of variables $P=\{p_1,p_2\}$, obtained from Table 6.1 are the following:

$p_1 = a_2x_1 \sim a_6extra\_x \sim a_{10}x_1 \sim a_{12}extra\_x \sim a_{13}x_4 \sim a_{14}x_5 \sim a_{17}x_2 \sim a_{18}x_3$;
$p_2 = a_2x_2$.

Here for all a_m, $a_m=1$ if the HFSM is in the state a_m, otherwise $a_m=0$.

This technique allows for a reduction of the number of address lines of the component that generates the next state from R+L to R+G with G<<L.

The replacement of input variables can be implemented with a **Programmable Multiplexer** block (see Figure 6.7). Since all variables from the set X in the above equations are asserted, all inverted lines in the programmable array implementation of the p_i Boolean functions can be skipped. As a result the complexity of the arrays will be reduced by almost a factor of two.

6.4 State Encoding

The next state function $a_s = \delta(a_m,X)$ generates the next state from the present state and the inputs. After the replacement of input variables the next state function is of the form $a_s = \delta'(a_m,P)$ with #P<<#X. In order to reduce even further the number of variables involved in the next state generation it is possible to use a state encoding algorithm with a variable code length. This algorithm will assign to a state a_m, many state codes where the most significant bits will be fixed and the least significant bits will embody the transition condition $P(a_m,a_s)$. Afterwards, the next state function will turn into the form $a_s = \mu(a_m)$.

In order to accomplish that, it is necessary to yield the logic connection between the lines $p_G,...,p_1$ and some of the lines $\tau_R,...,\tau_1$ providing the **or** logic function (see Figure 6.6). But, such connections are admissible if and only if the state code stored in the FSM memory (**base code**) has the bits $\tau_G,...,\tau_1$ equal to zero and if there is no ambiguity between transitions, i.e. if the resulting or-ed binary code will identify correctly the next state to be generated. For this purpose a special state encoding algorithm was presented in [Sklyarov96].

This algorithm assigns a different state binary code for each different transition of every state. As a result, if a state a_m has transitions to N different next states involving H variables from the new set of variables P (with H<=G), N different binary codes that only differ in their H least significant bits, must be assigned to the state a_m. One of them has the H least significant bits set to zero in order to act as the base code, while the remaining binary codes have their H least significant bits in a way that they will match the combinations of the P variables that occur in the transition conditions.

One possible state encoding for the HFSM presented in the state transition Table 6.1 is depicted in Figure 6.5. Since the state a_2 has four transitions depending on the two variables p_2 and p_1, it requires the four following binary codes: 001**00** for the transition a_2 $\bar{p}_2\ \bar{p}_1$ a_4 that will act as the base code; 001**01** for the transition a_2 $\bar{p}_2\ p_1$ a_7; 001**10** for the transition a_2 $p_2\ \bar{p}_1$ a_3; 001**11** for the transition a_2 $p_2\ p_1$ a_6.

$\tau_4\tau_3$ \ $\tau_2\tau_1$	00	01	11	10
00	a_0	a_1	a_3	a_4
01	a_2	a_2	a_2	a_2
11	a_6	a_6	a_5	a_7
10	a_8	a_9	a_{10}	a_{10}

$\tau_5 = 0$

$\tau_4\tau_3$ \ $\tau_2\tau_1$	00	01	11	10
00	a_{12}	a_{12}	a_{11}	a_{15}
01	a_{13}	a_{13}	a_{14}	a_{14}
11	a_{17}	a_{17}	a_{16}	
10	a_{18}	a_{18}		

$\tau_5 = 1$

Figure 6.5 – Karnaugh map for the special state encoding algorithm.

The new decomposition of the Moore HFSM that implements the replacement of input variables and this state encoding algorithm is depicted in Figure 6.6.

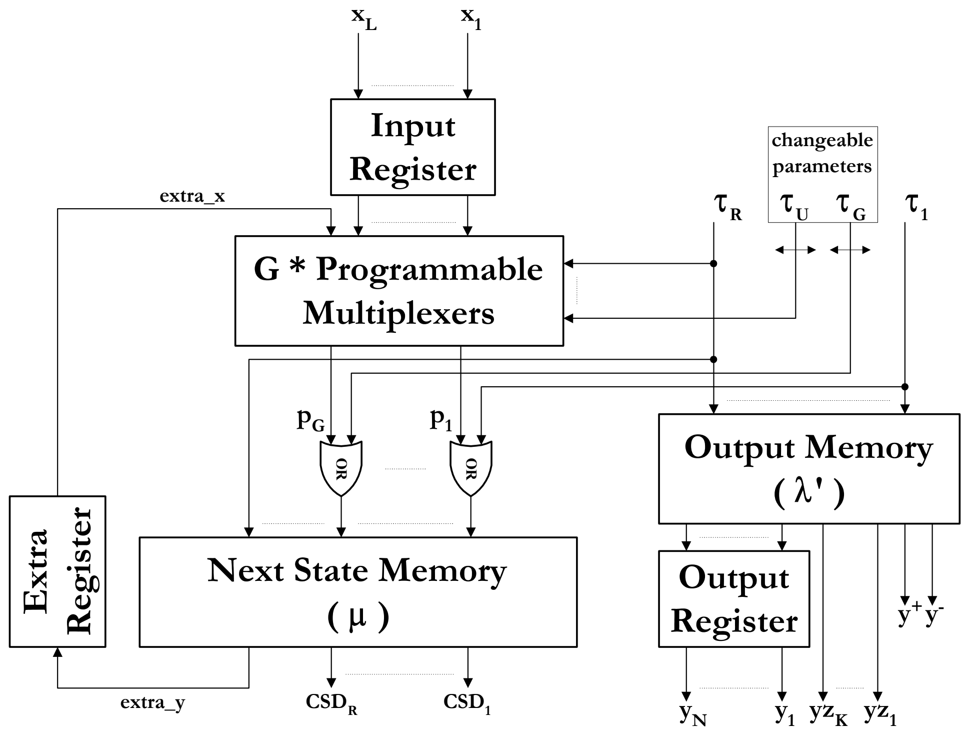


Figure 6.6 – Decomposition of the Moore HFSM combinational scheme using the replacement of input variables and the special state encoding algorithm.

Now the number of address lines of the **Next State Memory** is equal to R_{sc} instead of R_{bc}+G. However, since in most cases this state encoding algorithm will increase the state code length when compared with the ordinary binary algorithm, R_{sc} is generally bigger than R_{bc}.

Furthermore, since now a state a_m has more than one state binary code the output function is not exactly the same as before $Y(a_m) = \lambda'(a_m)$. And, since the **Output Memory** is only attacked by the base code of a state, for all the remaining state binary codes the RAM is wasted.

The **Programmable Multiplexer** block (see Figure 6.7) is controlled by the lines τ_R,...,τ_U. The minimal number of most significant bits of the state binary code should be used, in order to decrease the size of the multiplexers. In the example U=2, because the states in the p_i Boolean functions have at least two different codes, so the least significant bit τ_1 is not needed to distinguish the states.

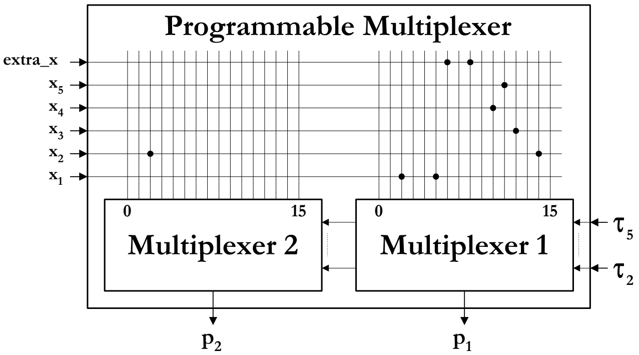


Figure 6.7 – Programmable multiplexer.

6.5 State Splitting

Suppose that the value of G has been fixed in the scheme of Figure 6.6, and there is a transition from the state a_m to the state a_s and $|X(a_m)|>G$, then the replacement of input variables cannot be carried out. In order to meet the constraint G, the transition can be split by inserting an intermediate state a_i in such a way that $|X(a_m)|<=G$ and $|X(a_i)|<=G$, and a_i is a state that does not assert any outputs.

Figure 6.8 shows an example where there are transitions traversing four conditional nodes ($|X(a_m)|=4$) from the state a_m to the states a_q and a_r. To make the replacement of input variables, G must be equal to four (see Table 6.2). But, if the state a_i is inserted at the input of the conditional node x_3, the transitions that were depending on four logic conditions are now split in transitions depending on only two logic conditions. Now it is possible to apply the replacement of input variables for G=2 (see Table 6.3).

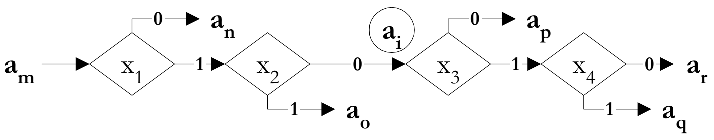


Figure 6.8 – Applying the state splitting technique.

The insertion of the extra state a_i can also reduce the state binary code length in some cases. Table 6.2 presents part of the state transition table of the example shown in Figure 6.8. Because the transitions from the state a_m involve four logic conditions the replacement of input variables will use four variables, and since there are five transitions, five binary codes are needed to encode the state a_m. And one of them must have the last four bits set to zero, to serve as the state base code. Since, the binary code with all bits set to zero is reserved for the state a_0, a binary code with only four bits cannot be used. For this example a binary code of five bits is needed, even if the number of states will not demand it.

Table 6.2 – State transition table without the extra state.

a_m	a_s	$X(a_m,a_s)$	$P(a_m,a_s)$	$K(a_m)$
a_m	a_n	$\bar{x}_1$	$\bar{p}_1$	10000
	a_o	$x_1\ x_2$	$p_1\ p_2$	10011
	a_p	$x_1\ \bar{x}_2\ \bar{x}_3$	$p_1\ \bar{p}_2\ \bar{p}_3$	10001
	a_r	$x_1\ \bar{x}_2\ x_3\ \bar{x}_4$	$p_1\ \bar{p}_2\ p_3\ \bar{p}_4$	10101
	a_q	$x_1\ \bar{x}_2\ x_3\ x_4$	$p_1\ \bar{p}_2\ p_3\ p_4$	11101

However, after inserting the extra state a_i, the replacement of input variables can be made with only two variables and only three binary codes are needed to encode each one of the states a_m and a_i (see Table 6.3). It is spent one more binary code than before, but now a binary code with only four bits is needed, if the number of states will allow it. Moreover, if small groups of binary codes are required, it is certainly easier to make the state encoding.

Table 6.3 – State transition table after inserting the extra state.

a_m	a_s	$X(a_m,a_s)$	$P(a_m,a_s)$	$K(a_s)$
a_m	a_n	$\bar{x}_1$	$\bar{p}_1$	0100
	a_o	$x_1\ x_2$	$p_1\ p_2$	0111
	a_i	$x_1\ \bar{x}_2$	$p_1\ \bar{p}_2$	0101
a_i	a_p	$\bar{x}_3$	$\bar{p}_1$	1100
	a_r	$x_3\ \bar{x}_4$	$p_1\ \bar{p}_2$	1101
	a_q	$x_3\ x_4$	$p_1\ p_2$	1111

Furthermore, the insertion of extra states is a technique used in FSM synthesis when the FSM is too complex to be implemented with programmable logic components of type PAL/PLA at hand and it has to be partitioned [Bolton90, Baranov94, Katz94].

6.6 State Splitting in the Tool SIMULHGS

In order to apply the state splitting technique, the tool **SIMULHGS** allows the introduction of extra states at the user request. This facility is provided by **Insertstatehgs()**, which starts by checking if the set of HGSs is already marked. A HGS name is requested to the user and after checking its existence **Insertstategs()** is called. After the insertion is completed and if any extra states were really inserted the method reports the actual number of states in the case of the HFSM model 2. And the proper synthesis method, with the marking step deactivated, is invoked to construct the new state transition table.

Since, a node can be repeated more than once in a HGS, it is very important to identify correctly the node to be marked. **Insertstategs()** requests a node name and if it is an unmarked conditional node, it is marked with a state in manual mode after the user confirmation.

With **Removestatehgs()** it is possible to remove states, but only those which were inserted at user request with the previous method. After selecting a HGS, **Removestategs()** removes the state assigned to a node after the user confirmation, but only if it was manually marked.

6.7 Quantification of the Optimisation Techniques

It is obvious that the replacement of input variables proposed in [Baranov79] should always be applied, since it allows the reduction of the number of address lines of the next state LUT from R+L to R+G. And if necessary, the introduction of extra states (state splitting) can always keep G<<L.

In the case of the special state encoding algorithm proposed in [Sklyarov96], the optimisation depends on the side effects generated by its application. More specifically the fact that the code length of the proposed special binary code (R_{sc}) in most practical examples will be bigger than the code length of the ordinary binary code (R_{bc}). If the length difference (R_{sc}-R_{bc}) will still be less than G, it should be taken into consideration. But even so, it can have different consequences for the two kinds of FSMs.

In the case of the Mealy FSM, the total amount of memory will automatically decrease, because the size of the **Next State and Output Memory** will decrease $2^{R_{bc}+G} - 2^{R_{sc}}$ words of R+N+K+3 bits.

However, in the case of the Moore FSM, it does not guarantee a decrease of the total amount of memory. On the contrary, the total amount of memory can eventually increase. Because, while the size of the **Next State Memory** will decrease $2^{R_{bc}+G} - 2^{R_{sc}}$ words of R+1 bits, the size of the **Output Memory** will increase $2^{R_{sc}} - 2^{R_{bc}}$ words of N+K+2 bits, and typically N+K+2>>R+1.

6.8 Conclusions

Since the specification of control units may change often, designers have to provide them with such facilities as flexibility, extensibility and reusability. Hence, it is necessary to implement them with dynamically reconfigurable HFSMs (PHFSMs) that can be reprogrammed in minimal time and with minimal effort. In order to achieve that functionality they must be assembled with reprogrammable components such as RAMs or FPGAs.

If a HFSM (PHFSM) is implemented with RAMs, it is convenient to make the replacement of the input variables in order to decrease the size of the next state memory. When this optimisation technique is used in combination with the special state encoding algorithm, the size of the next state memory is further reduced. Nevertheless, most of the times this algorithm can lead to an increase of the state code length and therefore to compromise this goal in particular in the case of Moore machines.

However, the application of some optimisation techniques, like complex state encoding algorithms, in most cases can demand more time to be spent when it is necessary to reprogram the HFSMs (PHFSMs).

Since state splitting is an important technique that allows the replacement of input variables to work with a small set of new variables and can help the special state encoding algorithm, the tool **SIMULHGS** has methods for inserting and removing extra states, in order to apply this technique to an already synthesised algorithm.

7 VHDL Simulation of Hierarchical Finite State Machines

Summary

The aim of this chapter is to present the simulation results of the HFSM and the PHFSM models proposed in chapter three. The VHDL models were created and simulated using the Synopsys tools and waveforms of all simulations are presented.

The practical examples are those that were used in chapter four to explain the synthesis rules and whose ordinary state transition tables have been presented. All examples use the replacement of input variables and there are simulation results for the ordinary binary code and for the special state encoding algorithm. It is also shown how to provide flexibility and extensibility of a hierarchical algorithm with minimal changes in the VHDL models. Finally the advantages of a hierarchical implementation over a non-hierarchical one are discussed.

7.1 Introduction

The different FSM models (Hierarchical, Parallel and Parallel Hierarchical) were simulated using the Synopsys tools. In order to decrease the development time of a new FSM the models are parameterised. Each model has a parameter file that defines the clock cycle, the synchronisation pulse delays, the component delays and the bit vector sizes. The clock cycle used in all simulations was 40 nanoseconds.

The most important parameters are:

- **MAXSTACK** - stack memory size;
- **L** - number of input signals;
- **N** - number of output signals (microoperations);
- **G** - number of new input signals after the replacement of input variables;
- **R** - state binary code length;
- **K** - HGS binary code length;
- **U** - least significant bit that controls the programmable multiplexer when the special state encoding algorithm is used;
- **V** - number of registers of the PFSM memory;
- **Q** - number of stacks of the PHFSM memory;
- **C** - counter binary code length used in the parallel FSMs;
- **NLF** - number of logic functions used in the parallel FSMs.

The FSMs are modelled in VHDL using a structural description. Besides the components already depicted in Figure 4.4, Figure 4.5, Figure 4.6, Figure 4.7 and Figure 4.8, the FSM also includes a **Clock Generator** component that provides the clock and the synchronisation signals required for each model.

The components **Combinational Scheme**, **Selector**, **PFSM Memory** and **Parallel Stack Memory** are also described structurally. All their internal components and the main components **Clock Generator**, **Stack Memory**, **Code Converter** and **Reprogrammable Element** are described behaviourally.

In order to provide reconfigurable FSMs the **Combinational Scheme** blocks that generate the next state and output functions, the **Code Converter** and the **Reprogrammable Elements** of the HFSM model 3 are modelled as LUTs. Their contents are presented in Appendix A and they are refereed in the text as LUT #.

The **Combinational Scheme** of all models includes a **Programmable Multiplexer** to apply the replacement of input variables technique in order to reduce the size of the next state LUT. The multiplexer matrix part (see Figure 7.1) is modelled as bit vectors generated with the concatenation of the input signals and the logic value 0 in the appropriate input positions. This implementation makes it easy to reprogram the multiplexer behaviour.

7.2 Simulation of a Moore HFSM model 2

Let's consider the set of HGSs depicted in Figure 5.3. It has five input signals (L=5) and eight microoperations (N=8) and because it is specified with five macrooperations (z_1, z_2, z_3, z_4, z_5) and one logic function (θ_6), the HGS binary code length is three (K=3). That leaves the **Code Converter** with two free codes, one to output the state a_0 when the next state is generated by the **Combinational Scheme** and one not in use (see Table 5.3). This algorithm can be implemented as a mixed Moore/Mealy HFSM using the ordinary state transition Table 5.21. If the binary code is used for state encoding and since there are eighteen states, the state code length is five bits (R=5).

The replacement of input variables (see Table 6.1) demands two new variables (G=2) and to implement them two 32:1 multiplexers are required (see Figure 7.1).

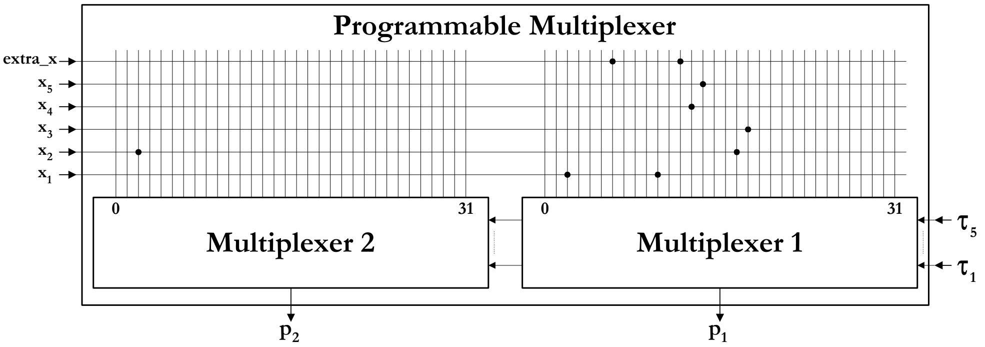


Figure 7.1 – Programmable multiplexer for the mixed Moore/Mealy HFSM model 2 with binary state encoding.

To simulate the mixed Moore/Mealy HFSM, a RAM of 2^K words of R bits to implement the **Code Converter** (see LUT 1), a RAM of 2^{R+G} words of R+1 bits to implement the next state function (see LUT 2) and a RAM of 2^R words of N+K+2 bits to implement the output function (see LUT 3) are required.

The waveform generated during the VHDL simulation of this mixed HFSM is depicted in Figure 7.2. The microoperations, the state binary codes and the HGS binary codes appear in decimal format in order to fit in the clock cycle slot. Moreover, since the binary code is used for state encoding the state number is equal to the state code in decimal format and it is very easy to track down the FSM internal status stored in the **Stack Memory** and represented by the signals STACK. The new set of input signals after the replacement of input variables, i.e. the **Programmable Multiplexer** output, is represented by the signal NEWX. The two synchronisation pulses presented in Figure 4.9 are represented by the signals SYNCLK1 and SYNCLK2.

The entry state of a HGS is provided by the **Code Converter** output lines CCD, in accordance with the HGS binary code given by its input lines YZ, while the **Combinational Scheme** output lines CSD are set to zero. Every time a HGS is invoked the signal INCSTACK (y^+) is activated, except for the main macrooperation z_1, and the HGS entry state is stored in the new top of the stack (see states a_9 stored in STACK(2) at 140 ns, a_{16} stored in STACK(3) at 260 ns, a_{13} stored in STACK(4) at 300 ns, a_{18} stored in STACK(3) at 540 ns in Figure 7.2). The z_1 entry state is stored in the first register of the stack (see state a_2 stored in STACK(1) at 60 ns in Figure 7.2).

The **Combinational Scheme** output lines CSD generate the next state inside each HGS, while the **Code Converter** output lines CCD are set to zero (see for example state a_3 stored in STACK(1) at 100 ns in Figure 7.2). The calculated value of the logic function θ_6 is fixed during the clock cycle after being generated (see EXTRAX activated at 580 ns in Figure 7.2).

Every time the HGS execution reaches the state a_1, the signal DECSTACK (y^-) is activated and the HFSM returns to the interrupted HGS (see DECSTACK at 385 ns in Figure 7.2). The signal INCSTACK and the binary code of the terminated HGS are again generated, but they are ignored because it occurs after the second synchronisation pulse (see INCSTACK and YZ at 395 ns in Figure 7.2). The same happens when the stack is incremented and the selected register holds the state a_1 because it was already used. But, in this case it is the signal DECSTACK that is generated and ignored (see DECSTACK at 515 ns in Figure 7.2).

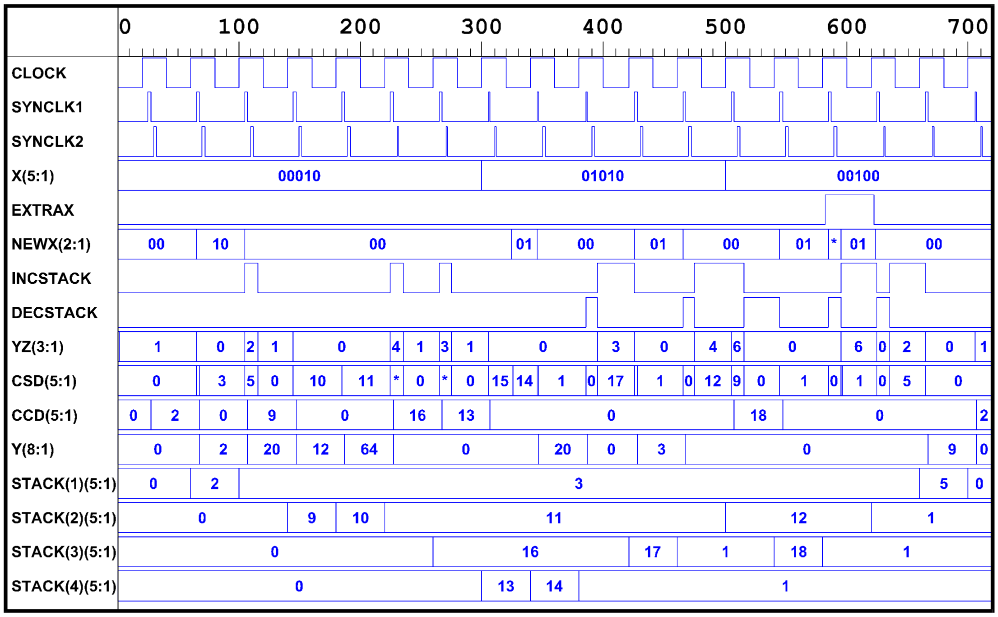


Figure 7.2 – Waveform of the mixed Moore/Mealy HFSM model 2 with binary state encoding.

Now let's consider the special state encoding algorithm proposed in Chapter 6 and the state encoding presented in Figure 6.5. For this example the need of more than one code for some of the states does not increase the state code length of five bits (R=5). Since, with this algorithm some states have more than one binary code, the least significant bit τ_1 is not needed to distinguish between them and in order to apply the replacement of input variables now it is necessary two 16:1 multiplexers (see Figure 7.3).

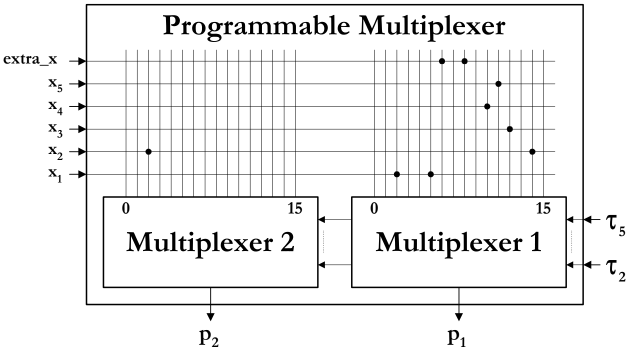


Figure 7.3 – Programmable multiplexer for the mixed Moore/Mealy HFSM model 2 with special state encoding.

To simulate the machine for this state encoding algorithm it is necessary, a RAM of 2^K words of R bits to implement the **Code Converter** (see LUT 4) and a RAM of 2^R words of N+K+2 bits to implement the output function (see LUT 5). But to implement the next state function a RAM of 2^R words of R+1 bits is needed (see LUT 6) instead of a RAM of 2^{R+G} words. For this example, the special state encoding algorithm allows a reduction in the size of the **Next State Memory** by a factor of four and in the size of the **Programmable Multiplexer** by a factor of two.

The waveform generated during the VHDL simulation that is depicted in Figure 7.4 has the same behaviour of the waveform presented in Figure 7.2, i. e. the same microoperation values. However it is more complicate to track down the FSM internal status when compared with the previous waveform, because the state number is most of the times not equal to the state code in decimal format.

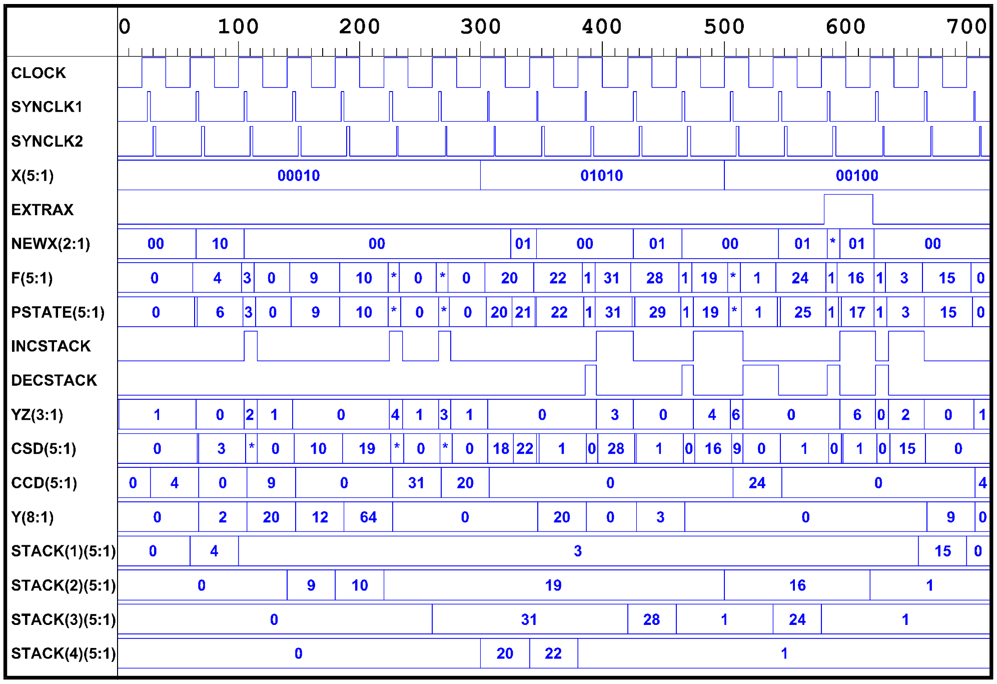


Figure 7.4 – Waveform of the mixed Moore/Mealy HFSM model 2 for the special state encoding.

In order to understand the machine internal status it must be kept in mind that the **Combinational Scheme** and the **Code Converter** always generate the base code of a state that has more than one binary code. The signal PSTATE is the present state, obtained with the logic or connection between the **Programmable Multiplexer** output lines NEWX and some of the **Stack Memory** output lines F. For example, the z_1 entry state a_2 generated by the **Code Converter** has the decimal value four (see F at 60 ns in Figure 7.4). When this binary code is or-ed with NEWX that is equal to "10", the present state binary code gets the decimal value six (see PSTATE at 65 ns in Figure 7.4). For this binary code the **Next State Memory** generates the next state a_3 that has the decimal value three (see CSD at 65 ns in Figure 7.4).

7.3 Simulation of a Mealy HFSM model 2

Let's consider the set of HGSs depicted in Figure 5.2 already marked for synthesis as a Mealy HFSM and the respective ordinary state transition Table 5.13. All parameters have the same values as for the mixed Moore/Mealy HFSM (L=5, N=8, K=3, R=5 and G=2). If the binary code is used, two 32:1 multiplexers are needed to implement the replacement of input variables (see Figure 7.5).

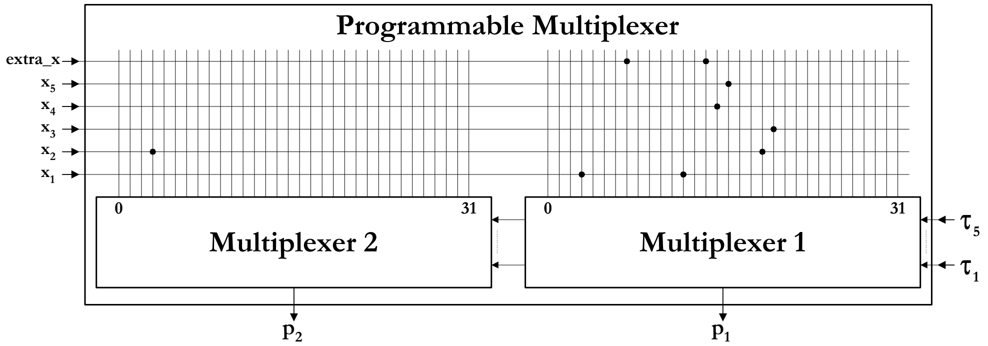


Figure 7.5 – Programmable multiplexer for the Mealy HFSM model 2 with binary state encoding.

The simulation requires, a RAM of 2^K words of R bits to implement the **Code Converter** presented in the Table 5.11 (see LUT 7), and a RAM of 2^{R+G} words of R+N+K+3 bits to implement the **Next State and Output Memory** (see LUT 8).

The waveform generated during the VHDL simulation of this Mealy HFSM is depicted in Figure 7.6 and it is very similar with Figure 7.2 except for the two following differences. The simulation takes three more clock cycles to finish (820 ns against 700 ns). This difference is due to the fact that the set of HGSs is marked with more states when implemented as a Mealy machine. For the path presented in the waveform, the Mealy HFSM has exactly three more states than the mixed Moore/Mealy HFSM. The introduction of these states between operational nodes also changes slightly the microoperation values (signal Y). For example the transition from the state a_3 to the state a_4 in z_1 generates an output with the decimal value zero, which did not occur in the mixed Moore/Mealy HFSM (see Y at 110 ns in Figure 7.6).

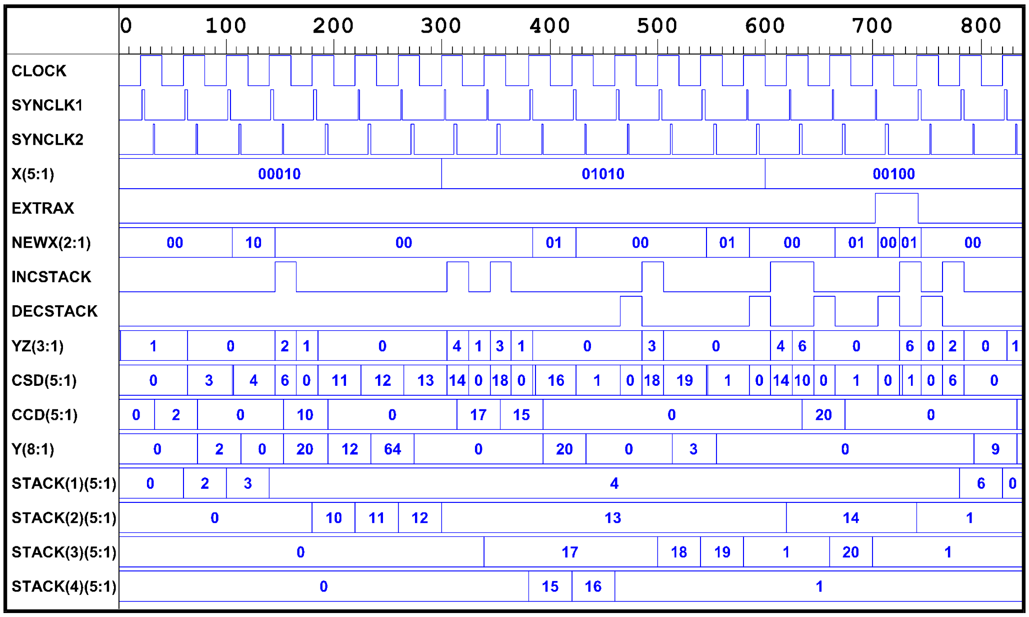


Figure 7.6 – Waveform of the Mealy HFSM model 2 with the binary state encoding.

If the special state encoding algorithm is used, the **Programmable Multiplexer** and the **Next State and Output Memory** sizes will decrease respectively by a factor of two and by a factor of four.

7.4 Simulation of a Moore HFSM model 3

Let's consider the synthesis of the set of HGSs depicted in Figure 5.3 as a mixed Moore/Mealy HFSM model 3 using the state transition Table 5.4, Table 5.5, Table 5.6, Table 5.7, Table 5.8 and Table 5.19 and the binary state encoding. Because macrooperation z_1 has nine states the state code length is four bits (R=4). The remaining parameters are the same as for the HFSM model 2 (L=5, N=8 and K=3). To simulate it is necessary to implement the **Reprogrammable Elements** associated with the elements of the set of HGSs.

The **Reprogrammable Element** associated with the main macrooperation z_1 has three inputs x_1, x_2 and the extra input extra_x that represents the calculated value of the logic function θ_6 invoked in the state b_6, and implements the Table 5.4. It is modelled as a RAM of 2^{R+3} words of R bits (see LUT 9).

The **Reprogrammable Element** associated with the macrooperation z_2 has two inputs x_1 and the extra input extra_x, that represents the calculated value of the logic function θ_6 invoked in the state b_5, and implements the Table 5.5. It is modelled as a RAM of 2^{R+2} words of R bits (see LUT 10).

The **Reprogrammable Element** associated with the macrooperation z_3 has two inputs x_4 and x_5, and implements the Table 5.6. It is modelled as a RAM of 2^{R+2} words of R bits (see LUT 11).

The **Reprogrammable Element** associated with the macrooperation z_4 has one input x_2 and implements the Table 5.7. It is modelled as a RAM of 2^{R+1} words of R bits (see LUT 12).

The **Reprogrammable Element** associated with the pure virtual macrooperation z_5 has no inputs, and implements the Table 5.8. It is modelled as a RAM of 2^R words of R bits (see LUT 13).

The **Reprogrammable Element** associated with the logic function θ_6 has one input x_3. Considering the mixed Moore/Mealy HFSM, the **RE** has the extra output extra_y, which represents the calculated value of the logic function. This output is set to one on the transition from the state b_2 to the state b_1 when x_3 has the logic value '1'. The **RE** implements the Table 5.19 and it is modelled as a RAM of 2^{R+1} words of R+1 bits (see LUT 14).

It is also necessary, a RAM of 2^{K+R} words of N bits to implement the **Output Block Memory** that provides the microoperations (see LUT 15) and a RAM of 2^{K+R} words of K+2 bits to implement the **Code Converter** presented in the Table 5.10 (see LUT 16).

The waveform generated during the VHDL simulation of this mixed HFSM is depicted in Figure 7.7. The microoperations, the HGS binary codes and the state binary codes appear in decimal format in order to fit in the clock cycle slot. Since, the HGS binary code with all bits set to zero is reserved for the main HGS, the decimal value of the macrooperation (logic function) binary code is equal to its index number minus one. In order to track down the FSM internal status, the state stored in the **State Stack Memory** (signals STACKSTATE) and the HGS code stored in the **HGS Stack Memory** (signals STACKHGS) were traced.

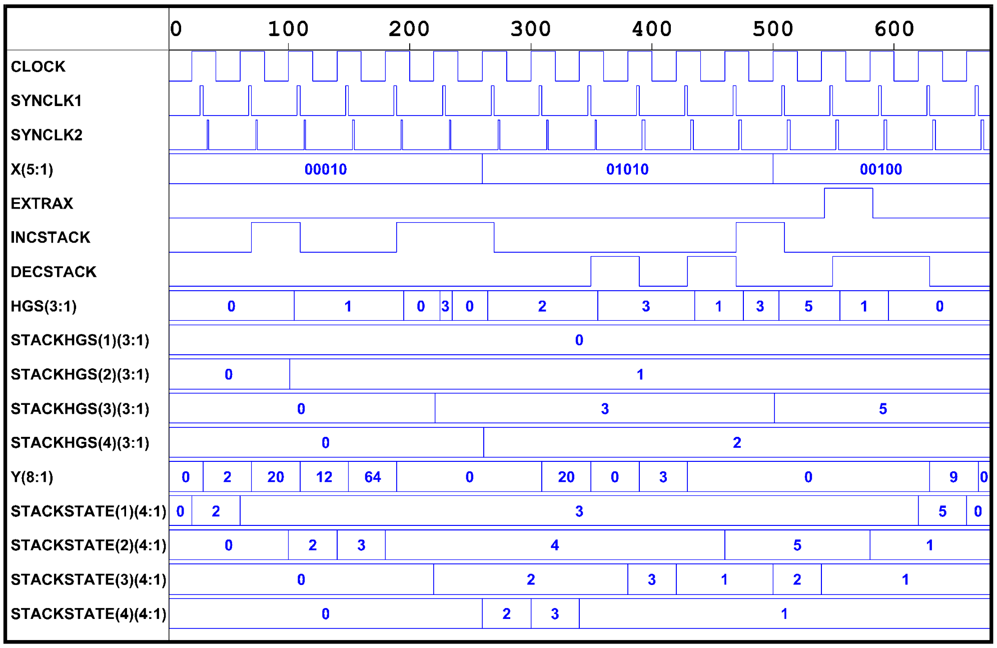


Figure 7.7 – Waveform of the mixed Moore/Mealy HFSM model 3.

Figure 7.7 and Figure 7.2 are practically identical but the HFSM model 3 simulation took one clock cycle less to finish (660 ns against 700 ns), because the HFSM model 3 does not need one clock cycle to start running. When the HFSM model 2 starts executing the starting state a_0 is loaded in the first low to high front of the clock in the **Stack Memory**. And the initial state a_2 of the main macrooperation generated by the **Code Converter** will be loaded in the next low to high front of the clock. But when the HFSM model 3 starts executing, the **Reprogrammable Element** associated with the main macrooperation is activated and generates the initial state b_2 that is loaded in the first low to high front of the clock in the **State Stack Memory** thus economising one clock cycle.

7.5 Simulation of a Mealy HFSM model 3

Let's consider the synthesis of the set of HGSs depicted in Figure 5.2 as a Mealy HFSM model 3 using the ordinary state transition Table 5.14, Table 5.15, Table 5.16, Table 5.17, Table 5.18 and Table 5.19 and the binary state encoding. Since the macrooperation z_1 has ten states, the state code length is four bits (R=4). The remaining parameters are the same as for the previous machine (L=5, N=8 and K=3). To simulate it is necessary to implement the **Reprogrammable Elements** associated with the elements of the set of HGSs.

The **Reprogrammable Element** associated with the main macrooperation z_1 has three inputs x_1, x_2 and the extra input extra_x, and eight outputs y_1, y_2, y_3, y_4, y_5, y_6, y_7 and y_8. The **RE** implements the Table 5.14 and it is modelled as a RAM of 2^{R+3} words of R+8 bits (see LUT 17).

The **Reprogrammable Element** associated with the macrooperation z_2 has two inputs x_1 and the extra input extra_x, and three outputs y_3, y_4, and y_7. The **RE** implements the Table 5.15 and it is modelled as a RAM of 2^{R+2} words of R+3 bits (see LUT 18).

The **Reprogrammable Element** associated with the macrooperation z_3 has two inputs x_4 and x_5, and three outputs y_1, y_3, and y_5. The **RE** implements the Table 5.16 and it is modelled as a RAM of 2^{R+2} words of R+3 bits (see LUT 19).

The **Reprogrammable Element** associated with the macrooperation z_4 has one input x_2, and two outputs y_1 and y_2. The **RE** implements the Table 5.18 and it is modelled as a RAM of 2^{R+1} words of R+2 bits (see LUT 20).

The **Reprogrammable Elements** associated with the pure virtual macrooperation z_5 and the logic function θ_6 are the same as for the mixed Moore/Mealy HFSM (see LUT 13 and LUT 14). To implement the **Code Converter** presented in the Table 5.20 it is necessary a RAM of 2^{K+R} words of K+2 bits (see LUT 21).

Like it was expected the Mealy HFSM model 3 does not need one clock cycle to start running, when comparing with the Mealy HFSM model 2, and the simulation takes one clock cycle less (780 ns against 820 ns). Besides that detail the resulting waveform depicted in Figure 7.8 is identical to Figure 7.6.

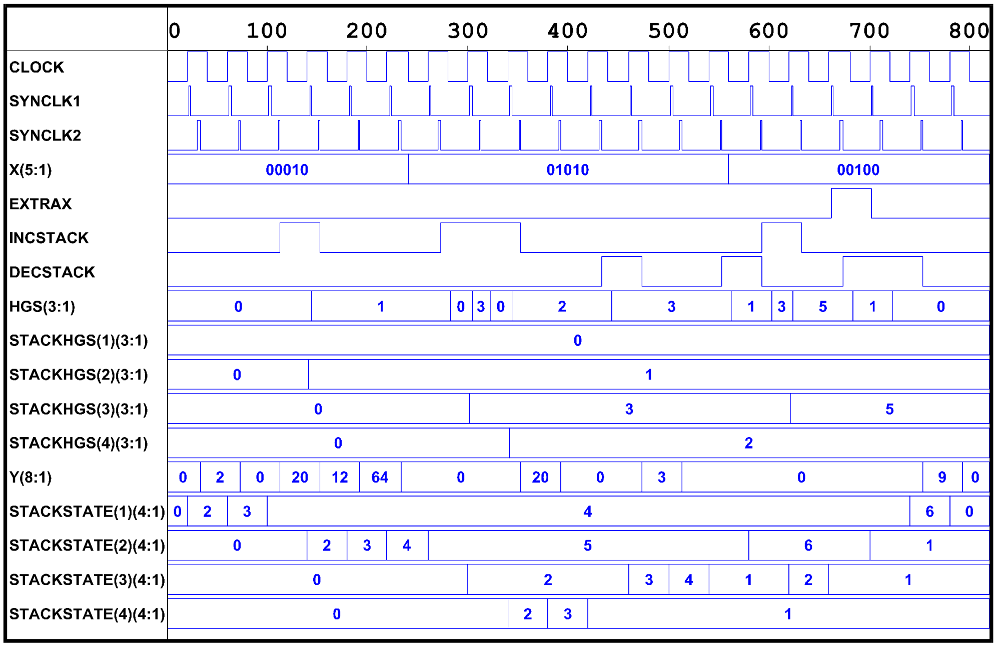


Figure 7.8 – Waveform of the Mealy HFSM model 3.

7.6 Simulation of a Moore PFSM

Let's consider the set of PHGSs depicted in Figure 5.5 already transformed and extended in order to provide proper synchronisation of parallel macrooperations for implementation in a PFSM with non-persistent microoperations. It has four input signals (L=4) and eight microoperations (N=8). Since it is specified with three macrooperations (z_1, z_2, z_3) and one logic function θ_4 (NLF=1), the PFSM memory has four registers (V=4) and a counter of module four is needed to provide the respective sub-clocks to scan the registers. That means a 2-bit binary counter (C=2).

This algorithm can be implemented as a Moore PFSM using the ordinary state transition Table 5.23. If the binary code is used for state encoding and since macrooperations z_1 and z_2 have eight states the state code length is three bits (R=3). The state marking of a PFSM is similar to the HFSM model 3 and the PFSM internal state is identified with the PHGS state and the counter binary code.

The inputs that are responsible for the next state generation, besides the state code, are the four input signals ($x_1,\ldots,x_4$), the calculated value of the logic function θ_4 (Θ_4) and the three extra signals ($Z_2,\ldots,Z_4$) that synchronise the parallel execution of macrooperations (see Figure 5.5 and Table 5.23).

The replacement of input variables demands three new variables (G=3) (see state a_0 of macrooperation z_4 in Table 5.23) and therefore three 32:1 multiplexers are used (see Figure 7.9).

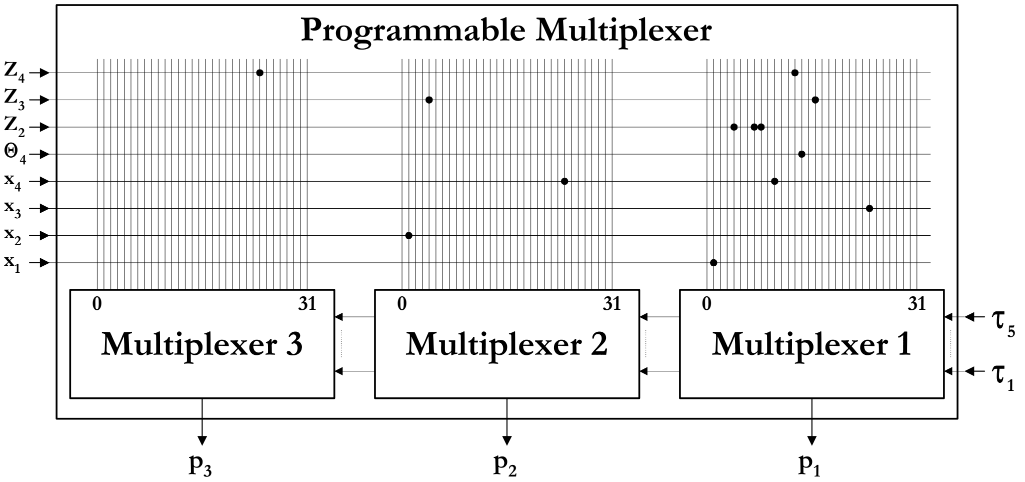


Figure 7.9 – Programmable multiplexer for the Moore PFSM.

To implement the **Next State Memory** a RAM of 2^{R+C+G} words of R bits is required (see LUT 22). The output function generates the microoperations (N), one set and one reset signal for each SR flip-flop that holds a Z_i signal (V-1 flip-flops), and one set and one reset signal for each SR flip-flop that holds the calculated value of a logic function (NLF flip-flops). Therefore, the **Output Memory** RAM must have 2^{R+C} words of N+(V-1)*2+NLF*2 bits (see LUT 23).

The waveform generated during the VHDL simulation of this PFSM is depicted in Figure 7.10. The microoperations and the PFSM state binary code appear in decimal format in order to fit in the clock cycle slot and to be easier to track down the PFSM internal status stored in the **PFSM Memory** (signals PFSMREG). The two synchronisation pulses presented in Figure 4.11 are represented by the signals CSCLK and STLCLK.

Macrooperations z_2 and z_3 are invoked in parallel in the state a_2 of macrooperation z_1. The Z_2 and Z_3 flip-flops are set in order to enable the parallel execution of both macrooperations (see IZ at 350 ns in Figure 7.10), while z_1 stays in the waiting state a_4. Macrooperation z_1 only resumes execution, transition from the state a_4 to the state a_5, after z_2 and z_3 have terminated (see PFSMREG(1) at 1820 ns in Figure 7.10). When macrooperation z_2 invokes logic function θ_4 in the state a_4, Z_4 is set to enable the logic function to run (see IZ at 1020 ns in Figure 7.10), and z_2 waits in the state a_5. The logic function θ_4 ends execution and returns the value 1 (see ILF at 1270 ns in Figure 7.10) and z_2 already in the state a_6 after the termination of θ_4 goes to the state a_7 (see PFSMREG(2) at 1540 ns in Figure 7.10) and finishes execution by resetting Z_2 (see IZ at 1670 ns and PFSMREG(2) at 1700 ns in Figure 7.10).

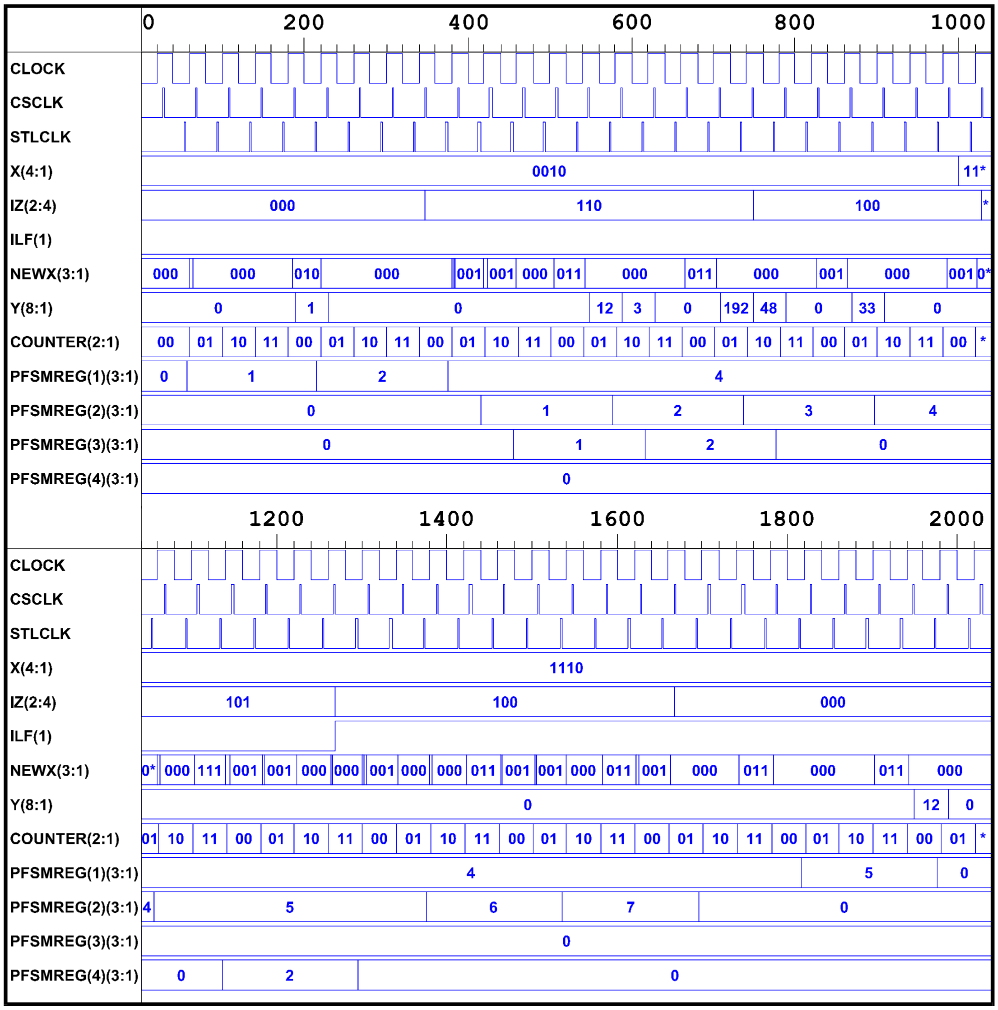


Figure 7.10 – Waveform of the Moore PFSM.

7.7 Simulation of a Moore PHFSM

Let's consider the set of PHGSs depicted in Figure 5.7, that has already been transformed and extended in order to provide proper synchronisation of parallel macrooperations, for implementation in a PHFSM with non-persistent microoperations. It has four input signals (L=4) and eight microoperations (N=8). Since it is specified with five macrooperations (z_1, z_2, z_3, z_4, z_5) and one logic function θ_6, the HGS binary code length is three bits (K=3).

The **Parallel Stack Memory** has only four stacks (Q=4), because the macrooperation z_5 is never invoked in parallel with any other macrooperation and therefore does not need a stack for itself, and logic functions are always invoked in a new hierarchical level. Since a counter of module four is needed to provide the respective sub-clocks to scan the stacks, the counter binary code length is two bits (C=2).

This algorithm can be implemented as Moore PHFSM using the ordinary state transition Table 5.25. If the binary code is used for state encoding and since there are eighteen states the state code length is five bits (R=5).

The replacement of input variables is more complicated to apply because when a macrooperation in invoked in parallel or alone in a new hierarchical level the sub-clock binary code is tested to distinguish both situations (see conditional node T_3 in the z_3 HGS of Figure 5.7).

The inputs that generates the next state, besides the state code, are the four input signals ($x_1,\ldots,x_4$), the calculated value of the logic function θ_6 (Θ_6), the three extra signals ($Z_2,\ldots,Z_4$) that synchronise the parallel execution of macrooperations and the sub-clock binary code T_j. In state a_0 (see Table 5.25) the state transition depends on T_j and all Z_i signals, but in fact for the sub-clock T_m only the respective Z_m must be inquired to decide the next state. Therefore, multiplexing the Z_i values accordingly with the Tj sub-clock, the inputs are reduced to T_m and Z_m. However, because the sub-clock binary code has two bits, the replacement of input variables demands three new variables (G=3) (see for example states a_0 and a_9 in Table 5.25) and therefore three 32:1 multiplexers are used (see Figure 7.11).

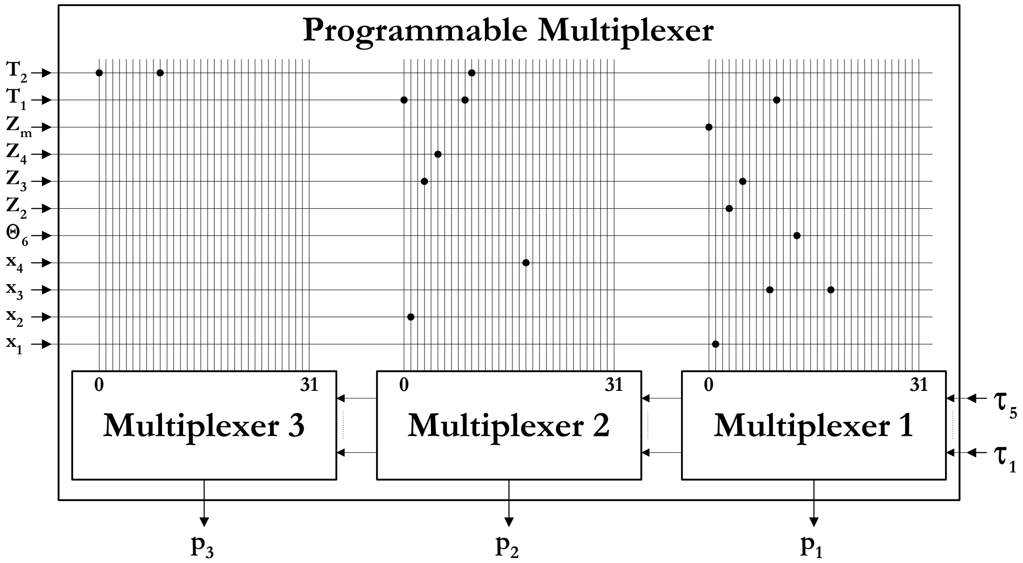


Figure 7.11 – Programmable multiplexer for the Moore PHFSM.

The output function generates, the microoperations (N), the HGS binary code (K), one set and one reset signal for each SR flip-flop that holds a Z_i signal (Q-1 flip-flops), one set and one reset signal for each SR flip-flop that holds the calculated value of a logic function (NLF flip-flops) and the special signals y^+ and y^-. Therefore, this PHFSM demands, a RAM of 2^K words of R bits to implement the **Code Converter** presented in the Table 5.26 (see LUT 24), a RAM of 2^{R+C+G} words of R bits to implement the **Next State Memory** (see LUT 26) and a RAM of 2^{R+C} words of N+K+Q*2+NLF*2 bits to implement the **Output Memory** (see LUT 25).

The VHDL simulation of this PHFSM for two different sequences of input signals produced the two waveforms presented in Figure 7.12 and Figure 7.13. To track down the PHFSM internal status, the state stored in the active register of each stack of the **Parallel Stack Memory** is traced (signals MUXINF).

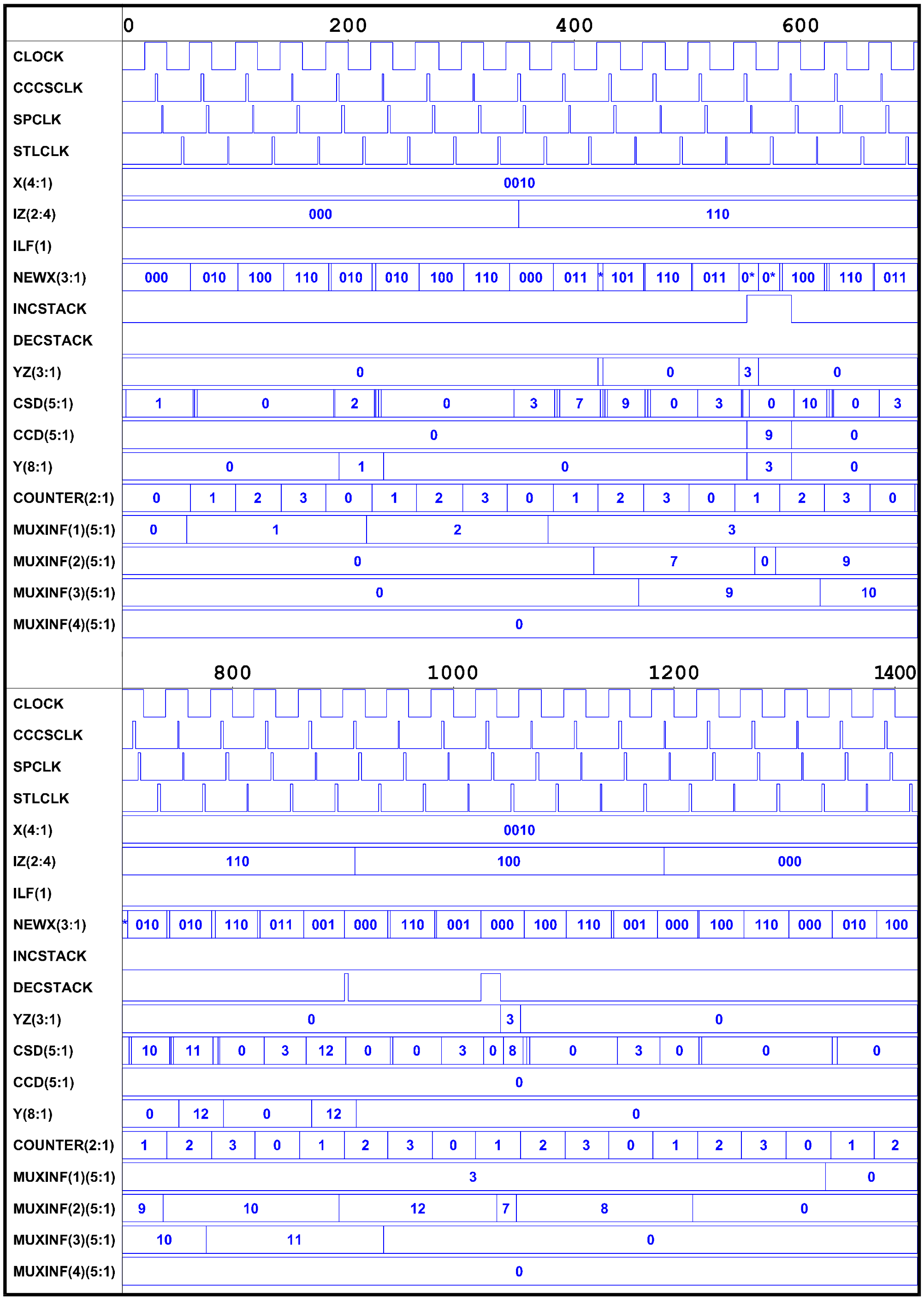


Figure 7.12 – First waveform of the Moore PHFSM.

The three synchronisation pulses presented in Figure 4.12 are represented by the signals CCCSCLK, SPCLK and STLCLK.

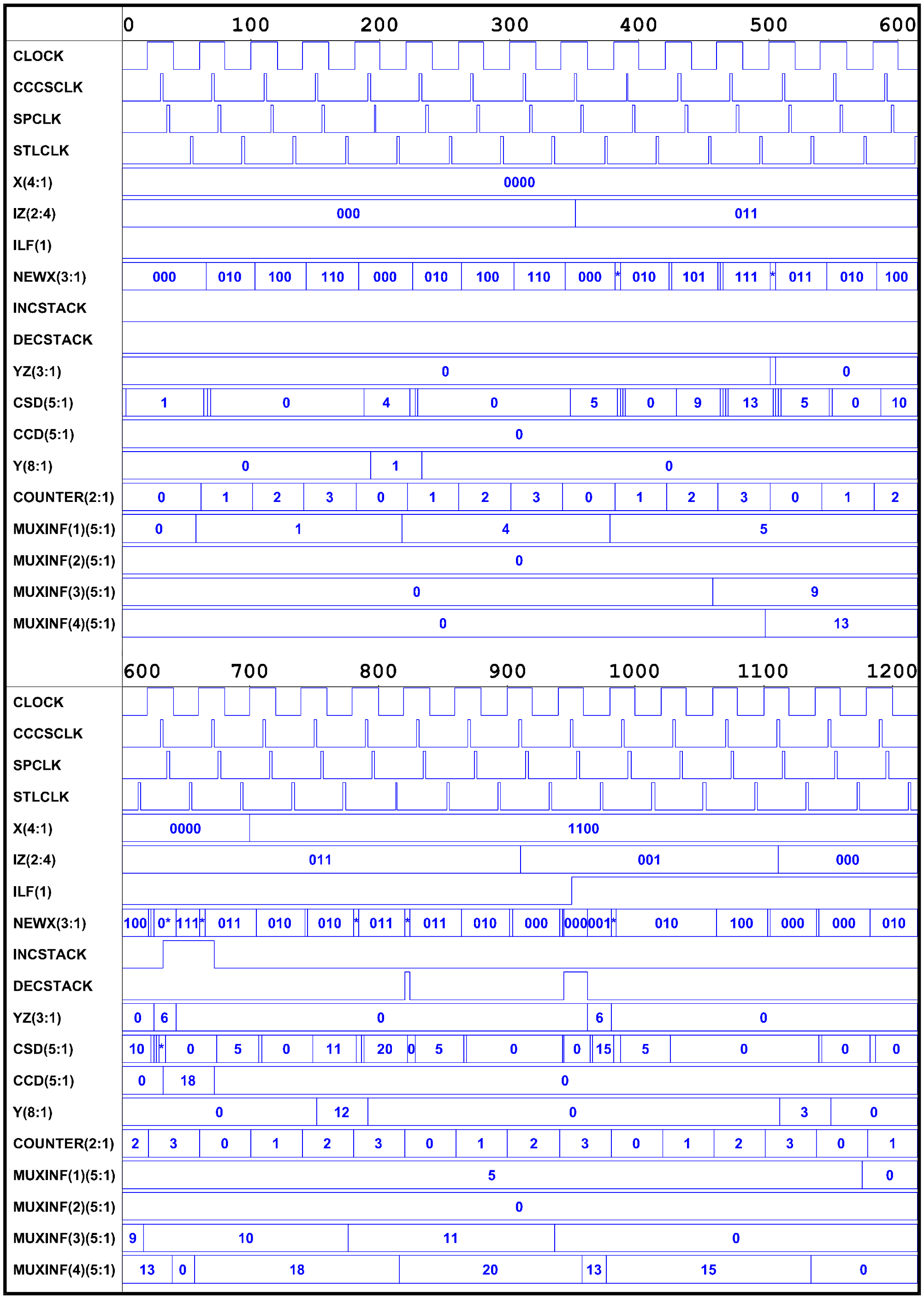


Figure 7.13 – Second waveform of the Moore PHFSM.

In the waveform presented in Figure 7.12 the macrooperations z_2 and z_3 are invoked in parallel at the state a_2 of macrooperation z_1. The Z_2 and Z_3 flip-flops are set in order to enable the parallel execution of both macrooperations (see IZ at 350 ns in Figure 7.12), while z_1 stays in the waiting state a_3. Macrooperation z_1 only resumes execution, transition from the state a_3 to the state a_0, after z_2 and z_3 have been terminated (see MUXINF(1) at 1340 ns in Figure 7.12). Meanwhile, macrooperation z_2 invokes macrooperation z_3 in the state a_7, that will run in a new register of the z_2 stack (see MUXINF(2) at 580 ns in Figure 7.12). At this time z_3 is running in parallel with itself in two different stacks (see states stored in MUXINF(2) and MUXINF(3) between 600 ns and 1000 ns in Figure 7.12). When this z_3 hierarchical invocation ends, z_2 resumes execution and goes from the state a_7 to the state a_8 (see MUXINF(2) at 1060 ns in Figure 7.12).

In the waveform shown in Figure 7.13, the macrooperations z_3 and z_4 are invoked in parallel at the state a_4 of macrooperation z_1 (see IZ at 350 ns in Figure 7.13), while z_1 stays in the waiting state a_5. Macrooperation z_4 invokes the logic function θ_6 at the state a_{13}, that will run in a new register of the z_4 stack (see MUXINF(4) at 660 ns in Figure 7.13). The execution of the logic function θ_6 ends and returns the value 1 (see ILF at 950 ns in Figure 7.13), and z_4 resumes execution at the state a_{13} and goes to the state a_{15} (see MUXINF(4) at 980 ns in Figure 7.13).

7.8 Providing Flexibility

Flexibility means the feasibility to modify a given behaviour in minimal time and with minimal effort. Let's consider the mixed Moore/Mealy HFSM presented in the previous paragraphs. Suppose it is necessary to change the macrooperation z_4 behaviour. For example, the microoperation y_8 must be inserted in the node marked with the state a_{16} and the microoperations y_1, y_2 must be replaced with the microoperations y_4, y_6 and the macrooperation z_5 in the node marked with the state a_{17} (see Figure 7.14a). Moreover, the current version of the logic function θ_6 must be substituted by another version in all conditional nodes where it is invoked (see Figure 7.14b).

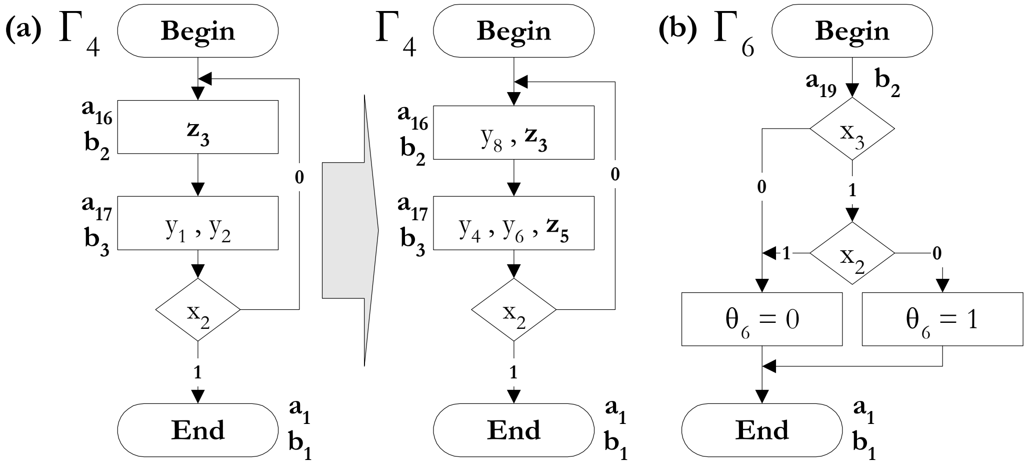


Figure 7.14 – (a) New implementation of the macrooperation z_4. (b) New version of the logic function θ_6.

Let's see how these changes can be provided for the Moore HFSM model 2 with the binary state encoding. To change the functionality of the operational nodes a_{16} and a_{17} it is necessary to reprogram the **Output Memory**. Replacing the 17-th output vector "0000000001110" (associated with the state a_{16}) with the vector "1000000001110" (y_8, z_3), and the 18-th output vector "0000001100000" (associated with the state a_{17}) with "0010100010110" (y_4, y_6, z_5).

In order to use the new version of the logic function, its HGS is marked with free state labels, the state transitions introduced by those new states are recorded and they are programmed in the **Next State Memory**.

In the example, the input of the first conditional node is marked with the state a_{19} and the state transitions presented in Table 7.1 are recorded. The new state transition is programmed by replacing the 20-th group of four empty next state vectors (associated with the state a_{19}) with "000010" "000010" "000011" "000010" (next state a_1 and the extra output extra_y for the transition a_{19} x_3 $\bar{x}_2$ a_1).

Table 7.1 – Ordinary state transition table for the new version of the logic function θ_6.

a_m { $Y(a_m)$ }	a_s	$X(a_m,a_s)$	$Y(a_m,a_s)$
a_{19}	a_1	$\bar{x}_3$	
	a_1	$x_3\ \bar{x}_2$	$extra\_y$
	a_1	$x_3\ x_2$	

Since the replacement of input variables technique is being used the **Reprogrammable Multiplexer** must also be reprogrammed to accommodate the need of the inputs x_3 and x_2 for the new state a_{19} (see Figure 7.15).

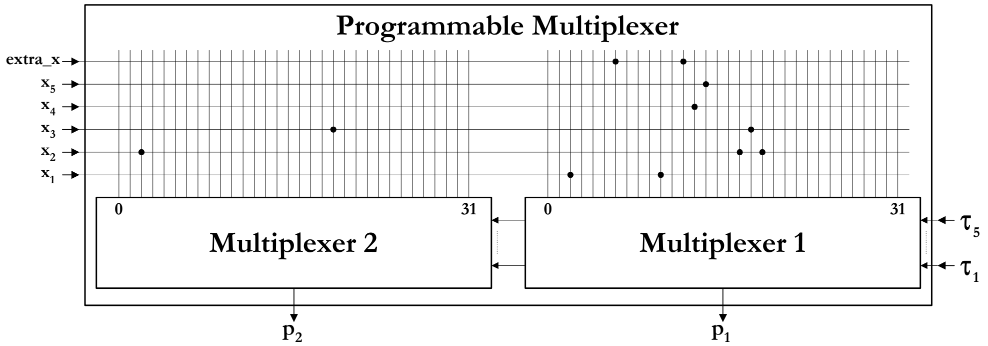


Figure 7.15 – New implementation of the programmable multiplexer for the mixed Moore/Mealy HFSM model 2 with binary state encoding in order to provide flexibility.

Finally to use the new version of the logic function θ_6 in all its invocations, it is necessary to reprogram the θ_6 entry state binary code generated by the **Code Converter**, by replacing the 7-th entry state vector "10010" (state a_{18}) with "10011" (state a_{19}).

However, if it is necessary to invoke the two different versions of the logic function in two different conditional nodes, the following actions must be done:

1. to assign a new HGS binary code to the new version of the logic function, for example the code "111" that is not in use;
2. to reprogram the **Code Converter** for providing the entry state a_{19} for the new HGS binary code by replacing the 8-th entry state vector "00000" (clear state a_0) with "10011" (state a_{19}) and keeping the entry state a_{18} for the old HGS binary code;
3. in the conditional nodes where the first version of θ_6 is invoked the output function must generate the HGS binary code "110", while in the nodes where the second version is invoked the HGS binary code "111" must be generated.

The waveform generated during the VHDL simulation of this mixed HFSM with the changes mentioned above is depicted in Figure 7.16.

Comparably with Figure 7.2, now the state a_{16} generates the microoperation y_8 (128) (see Y at 265 ns in Figure 7.16). And the state a_{17} generates the microoperations y_4, y_6 (40), and since the pure virtual macrooperation z_5 is invoked its HGS binary code (5) is generated (see YZ at 425 ns in Figure 7.16). The pure virtual macrooperation is executed during only one clock cycle and the HFSM returns back to the state a_{17}. When the logic function θ_6 is invoked in the state a_{12} of macrooperation z_2 it is in fact the second version that is running (see state a_{19} stored in STACK(3) and NEWX at 580 ns in Figure 7.16).

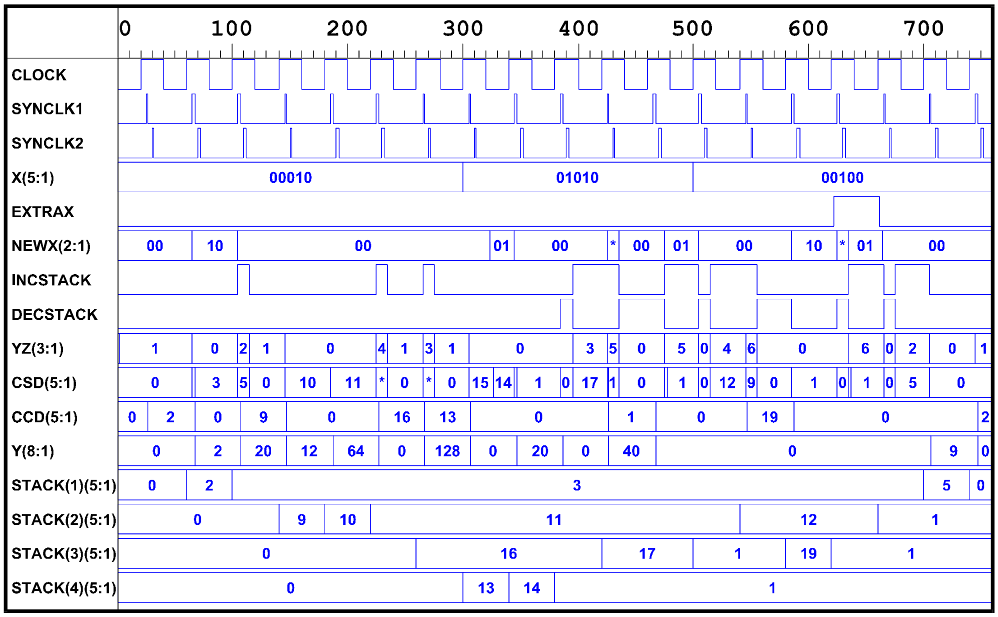
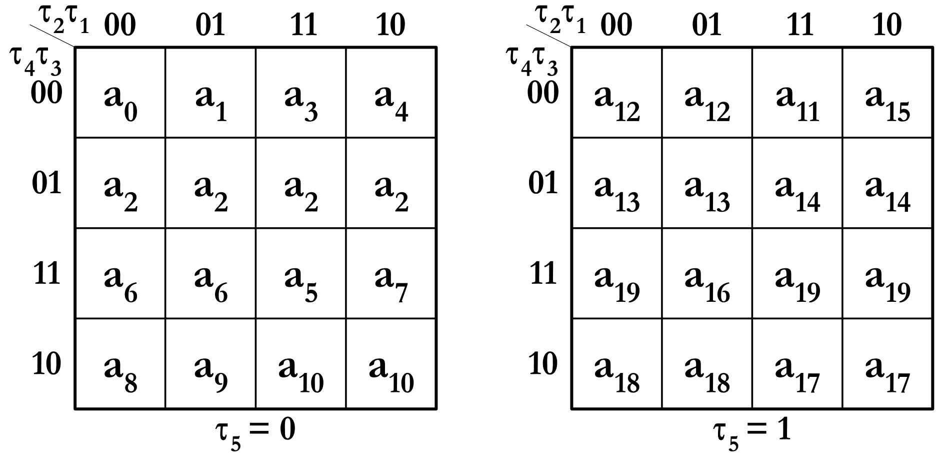


Figure 7.16 – Waveform of the mixed Moore/Mealy HFSM model 2 with the changes that provide flexibility.

If the special state encoding algorithm is used, three binary codes are needed for the state a_{19}. But to accommodate them in the Karnaugh map of Figure 6.5 the state assignments of the states a_{16} and a_{17} have to change (see Figure 7.17 and compare it with Figure 6.5).



$\tau_4\tau_3$ \ $\tau_2\tau_1$	00	01	11	10
00	a_0	a_1	a_3	a_4
01	a_2	a_2	a_2	a_2
11	a_6	a_6	a_5	a_7
10	a_8	a_9	a_{10}	a_{10}

$\tau_5 = 0$

$\tau_4\tau_3$ \ $\tau_2\tau_1$	00	01	11	10
00	a_{12}	a_{12}	a_{11}	a_{15}
01	a_{13}	a_{13}	a_{14}	a_{14}
11	a_{19}	a_{16}	a_{19}	a_{19}
10	a_{18}	a_{18}	a_{17}	a_{17}

$\tau_5 = 1$

Figure 7.17 – Karnaugh map for state encoding.

Now it is necessary to reprogram the **Output Memory** and the **Next State Memory** for the new state a_{19} and for the states a_{16} and a_{17} whose codes have been changed. The **Code Converter** must be reprogrammed like it was explained for the binary state encoding, but the macrooperation z_4 entry state has also to be reprogrammed because the a_{16} code has been changed.

The **Programmable Multiplexer** must be updated for all the states mentioned above that have input signals in their transition functions, i. e. for the states a_{17} and a_{19} (see Figure 7.18 and compare it with Figure 7.3).

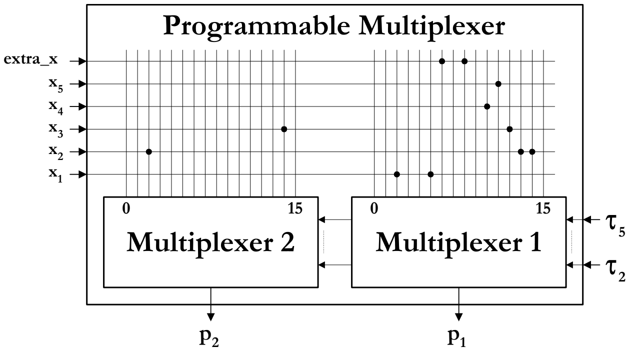


Figure 7.18 – New implementation of the programmable multiplexer for the mixed Moore/Mealy HFSM model 2 with special state encoding in order to provide flexibility.

It should be kept in mind when using the special state encoding algorithm, to leave as much empty rows in the Karnaugh map as possible, in order to allow future state assignments without having to make many changes in the states already assigned. But even doing so, to apply some changes to an already implemented HFSM using this state encoding algorithm, demands more time when compared with the ordinary binary state encoding algorithm.

In the Moore HFSM model 3, to reprogram the **Moore Output Block** changes the operational nodes functionality in terms of the microoperations generation. But in order to add or delete a macrooperation invocation the **Code Converter** has to be reprogrammed.

In order to supply another version of the logic function θ_6 and to keep both versions available it is necessary to insert a new **RE** that needs a RAM of 2^{R+2} words of R+1 bits. But, to substitute the implementation of the logic function θ_6 in all invocations, it is easier to reprogram the **RE$_6$**.

For the PFSM (PHFSM) it is very easy to change the microoperations generation by reprogramming the **Output Memory**. If it is required to add or delete a macrooperation, in a PFSM operational node or in a PHFSM operational node that already contains macrooperations, the transformed PHGS must be reanalysed in order to ensure that the parallel invocation of macrooperations is still correctly synchronised. Then the **Next State Memory** and the **Output Memory** must be reprogrammed accordingly with all the changes that have been made. However, to add a macrooperation in a PHFSM operational node that does not contain yet a macrooperation is done like for the HFSM model 2.

However, it is necessary to ensure that the PHGS of the macrooperation is synchronised to be used in a hierarchical invocation. To replace a logic function with another version or by another logic function in the PFSM is done the same way as inserting a macrooperation invocation, while in the PHFSM is done by reprogramming the **Code Converter** like for the HFSM model 2.

Everything that was said for the Moore machine can be applied to Mealy with one difference. For the model 2 the next state and output vectors are reprogrammed altogether in the same vector of the **Next State and Output Memory**. In the case of the model 3 all the changes in a HGS behaviour are made in the respective **RE**, apart from macrooperation and logic function invocations that are made in the **Code Converter**.

The example of Figure 7.14 has shown how to add the invocation of a macrooperation in an operational node. In order to abolish an existing macrooperation invocation, it is necessary to reprogram the output function in the opposite way, i.e. to delete the macrooperation binary code and the special signal y^+ for the appropriate state.

7.9 Providing Extensibility

Extensibility means to extend the defined behaviour in order to improve something. Let's consider the mixed Moore/Mealy HFSM presented initially. Suppose it is necessary to add a new operational node in the macrooperation z_4 before the node **End** (see Figure 7.19a) invoking the new macrooperation z_7 (see Figure 7.19b).

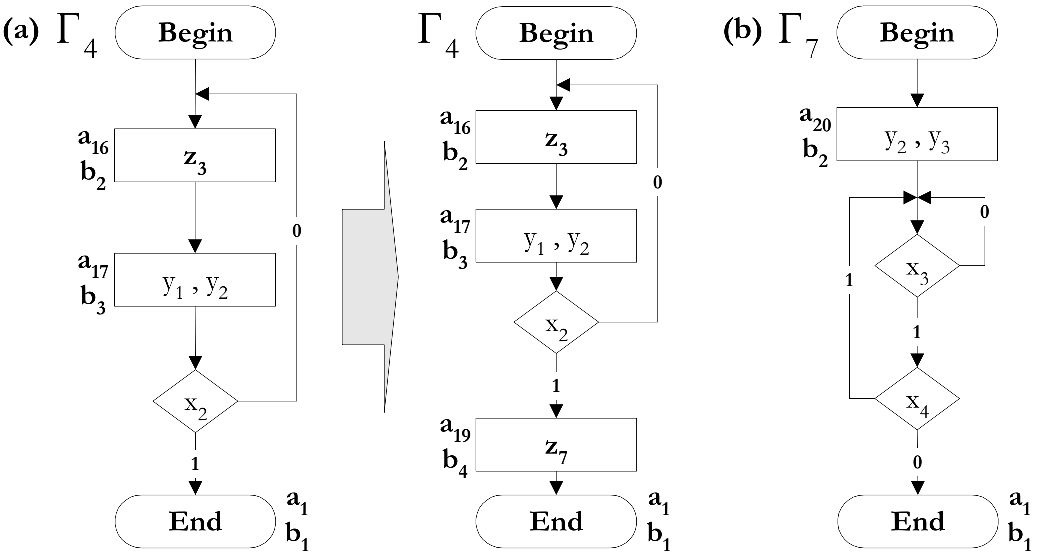


Figure 7.19 – (a) New implementation of the macrooperation z_4. (b) New macrooperation z_7.

In order to execute the changes mentioned above for the Moore HFSM model 2, the following steps must be performed:

1. reprogram the **Next State Memory**;
2. reprogram the **Programmable Multiplexer**;
3. assign a HGS binary code to the new macrooperation z_7 and reprogram the **Code Converter**;
4. reprogram the **Output Memory**.

Let's explain each step in detail for this example assuming binary state encoding.

Step 1. In order to reprogram the **Next State Memory,** the new nodes are marked with free state labels, the new state transitions are recorded and the old transitions that were affected by the insertion of the node are re-evaluated. Then all next state vectors of the states that have been affected must be reprogrammed.

In the example, the operational node inserted in macrooperation z_4 is marked with the state a_{19} and the operational node of macrooperation z_7 with the state a_{20} and the state transitions presented in Table 7.2 are recorded. First the state transition $a_{17}\ x_2\ a_{19}$ that substitutes the transition $a_{17}\ x_2\ a_1$ is reprogrammed by replacing the two vectors "000010" (state a_1) of the 18-th group of next state vectors with "100110" (state a_{19}).

Now, it is necessary to program the new state transitions. For the state transition a_{19} a_1, the 20-th group of four empty next state vectors is replaced with "000010" "000010" "000010" "000010". For the state transitions a_{20} $\overline{x}_3$ a_{20}, a_{20} x_3 x_4 a_{20} and a_{20} x_3 $\overline{x}_4$ a_1, the 21-st group of four empty next state vectors is replaced with "101000" "000010" "101000" "101000".

Table 7.2 – Ordinary state transition table for the new node of the macrooperation z_4 and the new macrooperation z_7.

a_m { $Y(a_m)$ }	a_s	$X(a_m,a_s)$
a_{17} {y_1,y_2}	a_{19}	x_2
	a_{16}	$\overline{x}_2$
a_{19} {z_7,y^+}	a_1	1
a_{20} {y_2,y_3}	a_{20}	$\overline{x}_3$
	a_{20}	x_3 x_4
	a_1	x_3 $\overline{x}_4$

Step 2. The **Programmable Multiplexer** is reprogrammed, if and only if the state transitions of the inserted nodes have input signals in their transition functions. That is the case of the state a_{20} that depends on the inputs x_4 and x_3 (see Figure 7.20).

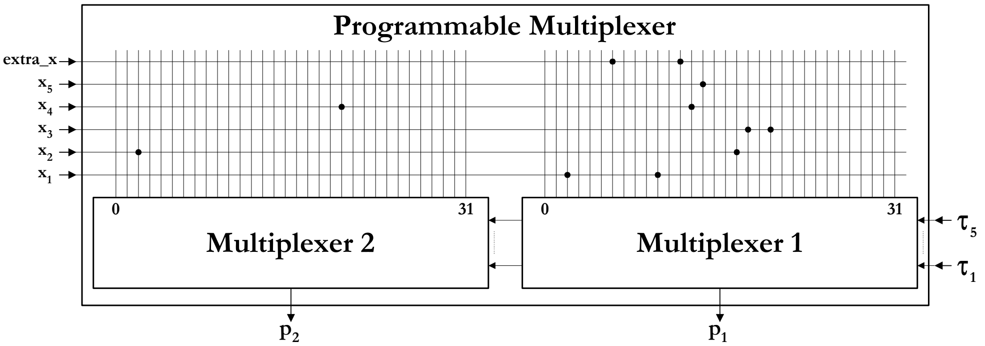


Figure 7.20 – New implementation of the programmable multiplexer for the mixed Moore/Mealy HFSM model 2 with binary state encoding in order to provide extensibility.

Step 3. The HGS binary code “111” can be assigned to the macrooperation z_7, since it is free. The **Code Converter** will generate the entry state a_{20} for z_7, if the 8-th entry state vector "00000" (clear state a_0) is replaced with "10100" (state a_{20}).

Step 4. In order to reprogram the **Output Memory**, the output functions of the new nodes are recorded and their output vectors are loaded. In the example, for the state a_{19}, the 20-th output vector is loaded with "0000000011110" (no microoperations, z_7, y^+) and for the state a_{20}, the 21-st output vector is loaded with "0000011000000" (y_2, y_3).

The waveform for this mixed HFSM with the changes mentioned above is depicted in Figure 7.21. Comparably with Figure 7.2, the next state of the state a_{17} is the state a_{19} (see STACK(3) at 460 ns in Figure 7.21). At the state a_{19} the macrooperation z_7 is invoked and its HGS binary code (7) is generated (see YZ at 465 ns in Figure 7.21). The macrooperation z_7 generates the microoperations y_2

and y_3 (6) in the state a_{20} (see Y at 510 ns in Figure 7.21) and returns back to the previous hierarchical level.

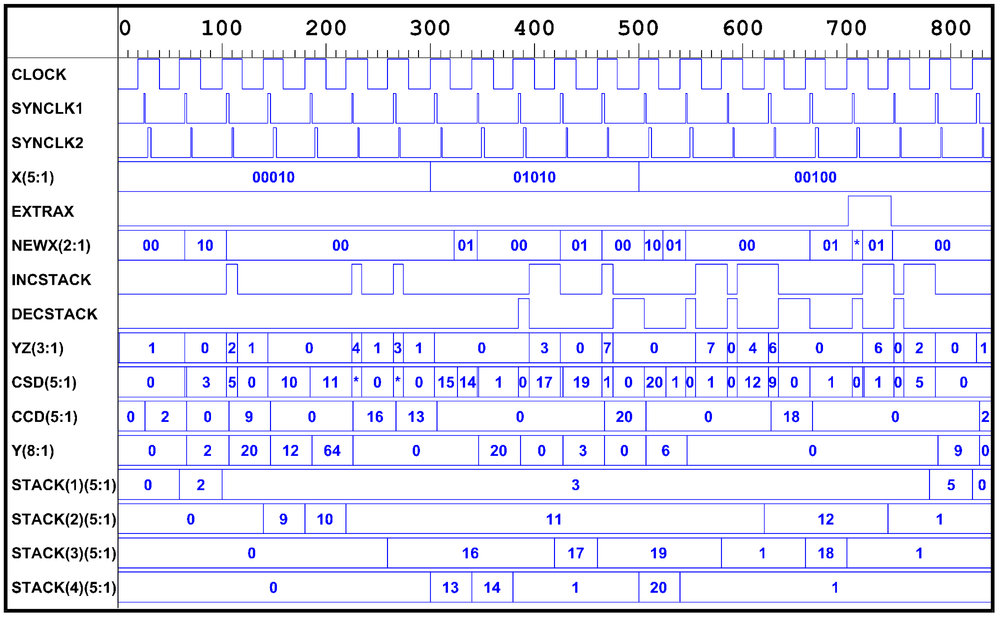


Figure 7.21 – Waveform of the mixed Moore/Mealy HFSM model 2 with the changes that provide extensibility.

If the special state encoding algorithm is used, three binary codes for the state a_{20} and one binary code for the state a_{19} are required, which is not possible to accommodate in the Karnaugh map of Figure 6.5. So the state binary code length will increase to six bits (R=6).

To implement the changes for this state encoding algorithm, the requirements necessary, when compared with the binary state encoding, are the following: two 32:1 **Programmable Multiplexers**; a **Next State Memory** with half of the size; an **Output Memory** with twice the size. Since the **Output Memory** word length is twice of the **Next State Memory** word length the total amount of memory needed is practically the same for both implementations.

In the Moore HFSM, the use of the special state encoding algorithm for an example with only two new variables after applying the replacement of input variables, does not decrease the total amount of memory size if the state code length will increase one bit. In contrast, the total memory size will still decrease for the Mealy HFSM.

In order to execute the changes mentioned above for the Moore HFSM model 3, it is necessary to perform the following steps:

1. reprogram the z_4 **RE**;
2. create the z_7 **RE**;
3. assign a HGS binary code to the new macrooperation z_7 and provide an output decoder line to connect the **RE**$_7$;
4. reprogram the **Code Converter**;
5. reprogram the **Output Memory Block**.

Let's explain in detail each step for this example. Step 1. In order to reprogram the **RE$_4$**, the operational node inserted in the macrooperation z_4 is marked with the first free state that is b_4, the state transitions generated are recorded and the transitions affected by its insertion are re-evaluated. In the example, the state transition b_3 x_2 b_4 that substitutes the transition b_3 x_2 b_1 is reprogrammed by replacing in the 4-th group of next state vectors the vector "0001" (state b_1) with "0100" (state b_4). And replacing the 5-th group of two empty next state vectors with "0001" "0001" (state b_1) programs the state transition b_4 b_1.

Step 2. In order to create the **RE$_7$**, the z_7 HGS is marked with states (see states b_i in Figure 7.19b) and the transitions are recorded. The **RE$_7$** has two inputs x_3 and x_4 and a RAM of 2^{R+2} words of R bits is required (see LUT 27).

Step 3. Since the HGS binary code "110" is free, it can be assigned to the macrooperation z_7 and therefore the **RE$_7$** must be connected with the 7-th output decoder line.

Step 4. To reprogram the **Code Converter** means to provide the z_7 invocation in the state b_4 of the macrooperation z_4, by replacing the 53-rd empty vector "00000" with "11010" (z_7 HGS code, y^+). It is also necessary to program the hierarchical invocations for the z_7 macrooperation states. The 97-th, 98-th and 99-th empty vectors must be replaced with respectively "11000" (z_7 HGS code for the state b_0), "00001" (signal y^- for the state b_1) and "11000" (z_7 HGS code for the state b_2).

Step 5. In order to reprogram the **Output Memory Block**, the microoperations of the new nodes of macrooperations z_4 and z_7 are recorded and loaded in the respective output vectors. In the example, the state b_4 does not have any microoperations, so it is not necessary to perform any action. But for the state b_2 the 99-th empty output vector is loaded with "00000110" (y_2, y_3).

The insertion of new operational nodes in the PFSM/PHFSM proceeds like it was explained before. The transformed PHGS is reanalysed in order to ensure that the invocation of macrooperations is correctly synchronised and then the **Next State Memory** and the **Output Memory** are reprogrammed.

In order to add a new macrooperation to an algorithm described by PHGSs and implemented in a PFSM, a new register must be added to the **PFSM Memory** and the clock generation has to be adapted in order to scan the new register. For the PHFSM two kinds of situations can happen. If the new macrooperation will be used in parallel invocations then it is required to add a new stack to the **Parallel Stack Memory** and proceed like for the PFSM. Otherwise, proceed like when it is necessary to add a new logic function.

Once again the extensibility example was only presented for the Moore machine. For the Mealy machine the differences described in the previous paragraph have to be considered.

7.10 Providing Reusability

It was shown in the two previous paragraphs how easy it is to change the behaviour of the FSMs. With the flexibility and extensibility provided by the models, it can be said that the HFSMs/PHFSMs when implemented with reprogrammable elements such as RAMS are reusable.

Since the VHDL models are parameterised, in order to reuse the HFSMs/PHFSMs, it is only necessary to change the parameters declared in the parameter file, to reprogram the **Programmable Multiplexer** matrix part and the contents of the components implemented as RAMs.

7.11 Using Pure Virtual HGSs

In order to specify an algorithm without a complete description, pure virtual macrooperations can be declared in the description just for the sake of testing it (z_5 in the example). In the end all pure virtual macrooperations must be specified.

A pure virtual macrooperation is executed in only one clock cycle in the HFSM model 2 (see Figure 7.16), because its entry state is in fact the HGS exit state a_1. Since in the case of the HFSM model 3 the entry state of a pure virtual HGS is the state b_2, its execution takes two clock cycles.

7.12 Hierarchical FSMs versus Non-Hierarchical FSMs

A hierarchical decomposition of an algorithm allows the developing of any complex control algorithm part by part concentrating the efforts on different levels of abstraction. Moreover, the macrooperations described can be separately tested and can be used to implement other algorithms developed in future. However, a hierarchical specification does not imply a hierarchical implementation. So why do it?

Let's consider the mixed Moore/Mealy HFSM used as an example in this chapter. If the hierarchical implementation is flattened then the set of HGSs is reduced to the ordinary GS depicted in Figure 7.22 with thirteen operational nodes, that can be implemented as a Moore FSM with a state binary code length of four bits (R=4). This means that the FSM implementation requires less memory to be implemented, because the **Code Converter** is not required and a single register can substitute the **Stack Memory** as the FSM memory.

But this happens because each macrooperation is invoked only once, with the exception of z_3. If the macrooperations were invoked more times, the number of operational nodes of the GS would increase and then the FSM synthesis would also increase in complexity.

It is also possible to provide a flexible, extensible and reusable FSM if the next state and output functions are implemented with reprogrammable elements such as RAMs. But can the non-hierarchical FSM really provide flexibility, extensibility and reusability?

Suppose it is necessary to make some changes in the macrooperation z_3. In order to do that in the GS of Figure 7.22, the changes have to be made in two different places, while in the hierarchical specification the changes are made only once in the HGS Γ_3. And more changes means more memory cells to reprogram.

Suppose it is required to replace all invocations of a macrooperation z_i with the macrooperation z_j. In the GS that forces the replacing of a group of nodes with another group of nodes in all appearances of the macrooperation z_i and therefore to re-synthesise parts of the state transition table and then to reprogram the FSM next state and output functions accordingly. In the HFSM only the **Code Converter** needs to be reprogrammed by replacing the entry state of the macrooperation z_i with the entry state of the macrooperation z_j.

Suppose it is essential to have a flexible conditional node. In the HFSM it is possible to have different versions of a logic function and by reprogramming the **Code Converter** or the output function of the conditional node, to invoke the desired version. In the FSM the condition contained in the conditional node has to be changed, some transitions have to be rewritten in the state transition table, the next state function must be reprogrammed and if the replacement of input variables is being used the **Programmable Multiplexer** also demands to be reprogrammed.

The hierarchical implementation really provides the reuse of an algorithm with simple changes in precise parts of the implementation and without the need to start the design process again from the beginning. The new algorithm inherits the invariable part of the previous algorithm and just deletes parts that are not needed and adds new parts that are different in the new context. The HFSMs with stack memory can be seen as general-purpose architectures capable of implementing hierarchical algorithms directly mapped from hierarchical specifications.

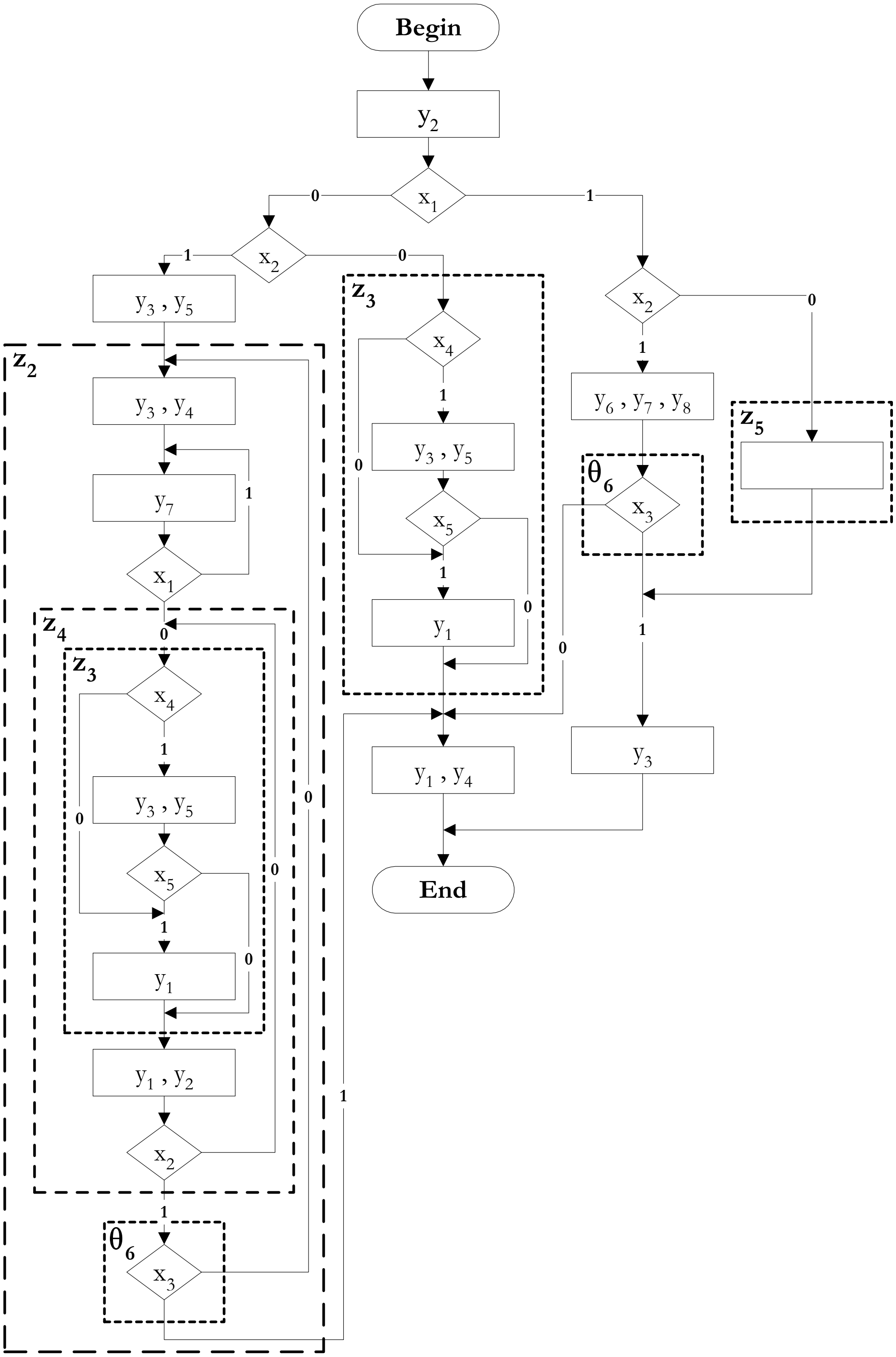


Figure 7.22 – Ordinary GS equivalent to the set of HGSs presented in Figure 5.1.

7.13 Conclusions

The VHDL simulation proved that the proposed HFSM and PHFSM models implemented with the suggested synchronisation mechanism perform correctly the transition between hierarchical levels. Furthermore, the PHFSM can provide pseudo-parallel execution of macrooperations.

The replacement of input variables can in fact reduce the total amount of memory needed to implement the HFSMs.

The special state encoding algorithm must be carefully applied in order to allow assigning binary codes to new states to be inserted in the future without having to make many changes in the states already assigned. If applied without increasing the state code length it effectively reduces the size of the next state LUT. Otherwise, in the case of the Moore HFSMs the size of the output LUT will eventually increase more than the falling size of the next state LUT and therefore will yield the opposite intention.

One way or another, the special state encoding algorithm forces the designer to spend more time when it is necessary to modify the FSMs behaviour, because it is not a straightforward procedure like when using ordinary binary encoding.

It was experimentally proved that flexibility and extensibility could be provided in minimal time and with minimal effort.

8 FINAL CONCLUSIONS AND FUTURE WORK

Summary

This chapter presents the final conclusions and summarises the original contributions of this work, namely the proposed models of hierarchical and parallel hierarchical FSMs, their synthesis from hierarchical specifications and the automatic synthesis tool **SIMULHGS**. The experimental results obtained with the VHDL simulation of the proposed FSM models are also analysed. In addition some future research opened up by this work is discussed.

8.1 Introduction

Formal state-based models that can support hierarchical and concurrent specifications were proposed in [Skyarov84, Sklyarov87, Harel87] and are strongly recommended by many authors [Gajski94, Micheli94, Edwards97] for modelling the functionality of complex control units.

In this thesis HGSs and PHGSs were the formal model chosen to specify the behaviour of control units. They allow a top-down decomposition of control algorithms through the use of macro blocks. Since these macro blocks can be seen as relatively autonomous blocks they can be separately tested and can be reused to implement other algorithms to be developed in the future.

However, since HGSs and PHGSs have the constraints in the transitions between hierarchical levels, they do not allow for example performing the last part of an already existing macrooperation without the need to perform the first part. The designer can solve this problem in two different ways:

- to specify another macrooperation that will implement the desired part of an already existing macrooperation and then invoke this new macrooperation in the required states. But this means to have a partially repeated macrooperation;
- to split the macrooperation implementation into two independent macrooperations, and invoking each part accordingly to the needs of the algorithm. But when it is necessary to execute the complete original macrooperation it demands two invocations to be performed sequentially and therefore to waste more time in the complete execution due to the extra hierarchical transitions.

Both solutions have advantages and disadvantages, but can lead to unnatural specifications of a hierarchical algorithm. Hence, the solution must pass by allowing hierarchical transitions to any state of any hierarchical level like in the Statecharts model. This transition flexibility can and must be accommodated in the HFSM models.

Besides the introduction of the concept of the HCFSM in [Gajski94] and the suggestion for the implementation of a hierarchical FSM through the interconnection of independent FSMs in [Micheli94] or a tree of interconnected FSMs in [DruHar89], there are not many proposals for a HFSM model that can be seen as a complete FSM implementing internally in an efficient way the switching between the different hierarchical levels of the machine.

The first model of a HFSM and its synthesis directly mapped from a hierarchical specification was presented in [Sklyarov84]. A HFSM with stack memory can efficiently implement a hierarchical control algorithm and can even perform a recursive algorithm providing that the size of the stack is well dimensioned.

8.2 Contributions

The contributions of this work to the synthesis of control units from hierarchical specifications covers two different directions. First, it proposes new models of HFSMs that are more efficient and flexible than the first models presented in [Sklyarov84, Sklyarov87] and that can provide such new facilities as flexibility, extensibility and reusability when implemented with reprogrammable components. Second, it proposes a synthesis methodology from hierarchical specifications based on HGSs/PHGSs and it presents an automatic synthesis tool for HFSMs specified by HGSs.

8.2.1 HFSM and PHFSM models

With the introduction of the **Code Converter** in the HFSM model 2, the HFSM becomes less complex and more versatile when compared with the first model. Moreover, this model can implement the hierarchical transition flexibility mentioned above, if the **Code Converter** is used to generate the arriving state of each hierarchical transition instead of the entry state of each macrooperation or logic function. In order to achieve this functionality, the binary codes that address the **Code Converter** and that are associated with the macrooperations and logic functions must be associated with the hierarchical transitions.

The HFSM model 3 provides an association between the set of HGSs and mutually exclusive **Reprogrammable Elements**. This model has acquired a regular structure that can be easily modified with the addition or deletion of REs and can in fact provide a flexible and extensible hierarchical implementation. With its modular structure it becomes suitable for the FPGA implementation of Mealy HFSMs. However, this model cannot implement the hierarchical transition flexibility mentioned above.

Based on the parallel FSM model proposed in [Sklyarov87] that implements the pseudo-parallel execution of FSMs, a parallel hierarchical FSM model and its synthesis from a specification based on PHGSs are also proposed. The PHFSM can manage both hierarchy and pseudo-parallelism, but it is not a true PHFSM and more work should be done in order to achieve it.

8.2.2 Synthesis of HFSMs

The synthesis methodology proposed has proven that as against an ordinary FSM the Moore HFSM is more efficient than the Mealy HFSM for implementing an algorithm with many hierarchical invocations. But, since it is more convenient to use a Mealy HFSM to implement logic functions, in order to speed up their evaluation, the mixed Moore/Mealy HFSM can take the better of the two machines and it should be used in the case of an algorithm with logic functions.

In addition the tool **SIMULHGS** that automatically synthesises HFSMs specified by HGSs was implemented.

According to [Harel88], a variety of computer-related systems and situations can and should be represented by visual formalisms: visual, because they are to be generated, comprehended and communicated by humans; and formal because they are to be manipulated, maintained and analysed by computers.

For that very reason, the work of this thesis was complemented with the development of the graphical editor of HGSs [ParCra98], which allows the creation of separated HGSs as well as algorithms composed with already existing and newly developed HGSs. This graphical editor generates text descriptions of correct HGSs and the textual decomposition of a hierarchical algorithm that can be used as input to the tool **SIMULHGS**.

8.2.3 Experimental Results

The VHDL simulation work done has demonstrated that the proposed models could in fact implement a hierarchical algorithm efficiently, and that they allow the use of virtual components in the description of the algorithm for the sake of testing. Moreover, they provide flexibility, extensibility in minimal time and with minimal effort. Therefore, it can be said that the proposed models are suitable for implementing complex control algorithms and that they are reusable.

The tool **SIMULHGS** used in conjunction with a graphical editor of HGSs provides an environment for the specification, verification, simulation and automatic synthesis of a hierarchical algorithm described by a set of HGSs. But, it is not yet a complete graphical environment and for the moment it only generates text files containing the state transition table and the **Code Converter** table.

8.3 Future Work

Since the tool **SIMULHGS** has not a user-friendly interface to the user, it should be redesigned in order to accommodate a complete graphical environment. It should incorporate a graphical editor and to provide graphically all the facilities already implemented namely, verification, simulation, automatic generation of state transition and **Code Converter** tables and the optimisation technique of state splitting. It should be extended to include the replacement of input variables technique and it should have the possibility of applying different state encoding algorithms.

The implementation of the HFSM model 2 in the Xilinx XC6200 dynamically reconfigurable FPGAs is already being pursued with promising results that are presented in [Sklyarov98]. Since FPGAs can be seen as the ideal target for the implementation of the HFSM model 3, the next step of research should be its implementation in reconfigurable FPGAs and to study how it is possible to take advantage of the dynamically reconfigurable capability in order to provide a flexible, extensible and reusable HFSM.

9 APPENDIX A - LUTS

This appendix presents the contents of the lookup tables of the FSMs presented in Chapter 7.

Moore HFSM model 2 with the binary state encoding algorithm

Code Converter

```
"00000" "00010" "01001" "01101" "10000" "00001" "10010" "00000"
```

LUT 1 - Code Converter.

Next State Memory

```
"000000" "000000" "000000" "000000" "000000" "000000" "000000" "000000"
"001000" "001110" "000110" "001100" "001010" "001010" "001010" "001010"
"001010" "001010" "001010" "001010" "000000" "000000" "000000" "000000"
"001010" "010000" "001010" "010000" "010000" "010000" "010000" "010000"
"000000" "000000" "000000" "000000" "010100" "010100" "010100" "010100"
"010110" "010100" "010110" "010100" "011000" "011000" "011000" "011000"
"010010" "000010" "010010" "000010" "011110" "011100" "011110" "011100"
"000010" "011110" "000010" "011110" "000010" "000010" "000010" "000010"
"100010" "100010" "100010" "100010" "100000" "000010" "100000" "000010"
"000010" "000011" "000010" "000011" "000000" "000000" "000000" "000000"
"000000" "000000" "000000" "000000" "000000" "000000" "000000" "000000"
"000000" "000000" "000000" "000000" "000000" "000000" "000000" "000000"
"000000" "000000" "000000" "000000" "000000" "000000" "000000" "000000"
"000000" "000000" "000000" "000000" "000000" "000000" "000000" "000000"
"000000" "000000" "000000" "000000" "000000" "000000" "000000" "000000"
"000000" "000000" "000000" "000000" "000000" "000000" "000000" "000000"
```

LUT 2 – Next State Memory.

Output Memory

```
"0000000000100" "0000000000001" "0000001000000" "0001010001010"
"0000000001110" "0000100100000" "1110000011010" "0000000010110"
"0000010000000" "0000110000000" "0100000000000" "0000000010010"
"0000000011010" "0000000000000" "0001010000000" "0000001100000"
"0000000001110" "0000001100000" "0000000000000" "0000000000000"
"0000000000000" "0000000000000" "0000000000000" "0000000000000"
"0000000000000" "0000000000000" "0000000000000" "0000000000000"
"0000000000000" "0000000000000" "0000000000000" "0000000000000"
```

LUT 3 - Output Memory.

Moore HFSM model 2 with the special state encoding algorithm

Code Converter

```
"00000" "00010" "01001" "01101" "10000" "00001" "10010" "00000"
```

LUT 4 – Code Converter.

Next State Memory

```
"000000" "000000" "011110" "011110" "000100" "011100" "000110" "011000"
"000000" "010100" "100110" "010100" "011110" "010000" "010000" "000000"
"010010" "000010" "000010" "100000" "100100" "101100" "000010" "100100"
"000010" "000011" "000000" "000000" "111110" "000010" "000000" "111000"
```

LUT 5 – Next State Memory.

Output Memory

```
"0000000000100" "0000000000001" "0000000001110" "0001010001010"
"0000001000000" "0000001000000" "0000001000000" "0000001000000"
"0000010000000" "0000110000000" "0100000000000" "0100000000000"
"1110000011010" "1110000011010" "0000000010110" "0000100100000"
"0000000011010" "0000000011010" "0000000100000" "0000000010010"
"0000000000000" "0000000000000" "0001010000000" "0001010000000"
"0000000000000" "0000000000000" "0000000000000" "0000000000000"
"0000001100000" "0000001100000" "0000000000000" "0000000001110"
```

LUT 6 – Output Memory.

Mealy HFSM model 2 with the binary state encoding algorithm

Code Converter

```
"00000" "00010" "01010" "01111" "10001" "00001" "10100" "00000"
```

LUT 7 – Code Converter.

Next State Output Memory

```
"0000000000000000100" "0000000000000000100" "0000000000000000100" "0000000000000000100"
"0000000000000000001" "0000000000000000001" "0000000000000000001" "0000000000000000001"
"0001100000001000000" "0001100000001000000" "0001100000001000000" "0001100000001000000"
"0010100000000000000" "0100000000000000000" "0010000000000000000" "0011101110000000000"
"0011000001010001010" "0011000001010001010" "0011000001010001010" "0011000001010001010"
"0011000000000001110" "0011000000000001110" "0011000000000001110" "0011000000000001110"
"0000000000100100000" "0000000000100100000" "0000000000100100000" "0000000000100100000"
"0011000000000011010" "0100100000000011010" "0011000000000011010" "0100100000000011010"
"0100100000000010110" "0100100000000010110" "0100100000000010110" "0100100000000010110"
"0000000000010000000" "0000000000010000000" "0000000000010000000" "0000000000010000000"
"0101100000110000000" "0101100000110000000" "0101100000110000000" "0101100000110000000"
"0110000100000000000" "0110000100000000000" "0110000100000000000" "0110000100000000000"
"0110100000000000000" "0110000100000000000" "0110100000000000000" "0110000100000000000"
"0111000000000010010" "0111000000000010010" "0111000000000010010" "0111000000000010010"
"0101000000000011010" "0000100000000011010" "0101000000000011010" "0000100000000011010"
"0000100000000100000" "1000000001010000000" "0000100000000100000" "1000000001010000000"
"0000100000000000000" "0000100000000100000" "0000100000000000000" "0000100000000100000"
"1001000000000001110" "1001000000000001110" "1001000000000001110" "1001000000000001110"
"1001100000001100000" "1001100000001100000" "1001100000001100000" "1001100000001100000"
"1000100000000000000" "0000100000000000000" "1000100000000000000" "0000100000000000000"
"0000100000000000000" "0000110000000000000" "0000100000000000000" "0000110000000000000"
"0000000000000000000" "0000000000000000000" "0000000000000000000" "0000000000000000000"
"0000000000000000000" "0000000000000000000" "0000000000000000000" "0000000000000000000"
"0000000000000000000" "0000000000000000000" "0000000000000000000" "0000000000000000000"
"0000000000000000000" "0000000000000000000" "0000000000000000000" "0000000000000000000"
"0000000000000000000" "0000000000000000000" "0000000000000000000" "0000000000000000000"
"0000000000000000000" "0000000000000000000" "0000000000000000000" "0000000000000000000"
"0000000000000000000" "0000000000000000000" "0000000000000000000" "0000000000000000000"
"0000000000000000000" "0000000000000000000" "0000000000000000000" "0000000000000000000"
"0000000000000000000" "0000000000000000000" "0000000000000000000" "0000000000000000000"
"0000000000000000000" "0000000000000000000" "0000000000000000000" "0000000000000000000"
"0000000000000000000" "0000000000000000000" "0000000000000000000" "0000000000000000000"
```

LUT 8 – Next State Output Memory.

Moore HFSM model 3 with the binary state encoding algorithm

Macrooperation z_1 RE

```
"0010" "0010" "0010" "0010" "0010" "0010" "0010" "0010"
"0010" "0010" "0010" "0010" "0010" "0010" "0010" "0010"
"0100" "0100" "0111" "0111" "0011" "0011" "0110" "0110"
"0101" "0101" "0101" "0101" "0101" "0101" "0101" "0101"
"0101" "0101" "0101" "0101" "0101" "0101" "0101" "0101"
"0000" "0000" "0000" "0000" "0000" "0000" "0000" "0000"
"0101" "0100" "0101" "0100" "0101" "0100" "0101" "0100"
"1000" "1000" "1000" "1000" "1000" "1000" "1000" "1000"
"0000" "0000" "0000" "0000" "0000" "0000" "0000" "0000"
"0000" "0000" "0000" "0000" "0000" "0000" "0000" "0000"
"0000" "0000" "0000" "0000" "0000" "0000" "0000" "0000"
"0000" "0000" "0000" "0000" "0000" "0000" "0000" "0000"
"0000" "0000" "0000" "0000" "0000" "0000" "0000" "0000"
"0000" "0000" "0000" "0000" "0000" "0000" "0000" "0000"
"0000" "0000" "0000" "0000" "0000" "0000" "0000" "0000"
"0000" "0000" "0000" "0000" "0000" "0000" "0000" "0000"
```

LUT 9 – Macrooperation z_1 RE Memory.

Macrooperation z_2 RE

```
"0010" "0010" "0010" "0010" "0010" "0010" "0010" "0010"
"0011" "0011" "0011" "0011" "0100" "0100" "0011" "0011"
"0101" "0101" "0101" "0101" "0010" "0001" "0010" "0001"
"0000" "0000" "0000" "0000" "0000" "0000" "0000" "0000"
"0000" "0000" "0000" "0000" "0000" "0000" "0000" "0000"
"0000" "0000" "0000" "0000" "0000" "0000" "0000" "0000"
"0000" "0000" "0000" "0000" "0000" "0000" "0000" "0000"
"0000" "0000" "0000" "0000" "0000" "0000" "0000" "0000"
```

LUT 10 – Macrooperation z_2 RE Memory.

Macrooperation z_3 RE

```
"0010" "0010" "0010" "0010" "0010" "0010" "0010" "0010"
"0100" "0011" "0100" "0011" "0001" "0001" "0100" "0100"
"0001" "0001" "0001" "0001" "0000" "0000" "0000" "0000"
"0000" "0000" "0000" "0000" "0000" "0000" "0000" "0000"
"0000" "0000" "0000" "0000" "0000" "0000" "0000" "0000"
"0000" "0000" "0000" "0000" "0000" "0000" "0000" "0000"
"0000" "0000" "0000" "0000" "0000" "0000" "0000" "0000"
"0000" "0000" "0000" "0000" "0000" "0000" "0000" "0000"
```

LUT 11 – Macrooperation z_3 RE Memory.

Macrooperation z_4 RE

```
"0010" "0010" "0010" "0010" "0011" "0011" "0010" "0001"
"0000" "0000" "0000" "0000" "0000" "0000" "0000" "0000"
"0000" "0000" "0000" "0000" "0000" "0000" "0000" "0000"
"0000" "0000" "0000" "0000" "0000" "0000" "0000" "0000"
```

LUT 12 – Macrooperation z_4 RE Memory.

Pure Virtual Macrooperation z_5 RE

```
"0010" "0010" "0001" "0000" "0000" "0000" "0000" "0000"
"0000" "0000" "0000" "0000" "0000" "0000" "0000" "0000"
```

LUT 13 – Pure Virtual Macrooperation z_5 RE Memory.

Logic Function θ_6 RE

```
"00100" "00100" "00100" "00100" "00010" "00011" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
```

LUT 14 – Logic Function θ_6 RE Memory.

Output Block Memory

```
"00000000" "00000000" "00000010" "00010100" "00000000" "00001001" "11100000" "00000000"
"00000100" "00000000" "00000000" "00000000" "00000100" "00000000" "00000000" "00000000"
"00000000" "00000000" "00001100" "01000000" "00000000" "00000000" "00000000" "00000000"
"00000000" "00000000" "00000000" "00000000" "00000000" "00000000" "00000000" "00000000"
"00000000" "00000000" "00000000" "00010100" "00000001" "00000000" "00000000" "00000000"
"00000000" "00000000" "00000000" "00000000" "00000000" "00000000" "00000000" "00000000"
"00000000" "00000000" "00000000" "00000011" "00000000" "00000000" "00000000" "00000000"
"00000000" "00000000" "00000000" "00000000" "00000000" "00000000" "00000000" "00000000"
"00000000" "00000000" "00000000" "00000000" "00000000" "00000000" "00000000" "00000000"
"00000000" "00000000" "00000000" "00000000" "00000000" "00000000" "00000000" "00000000"
"00000000" "00000000" "00000000" "00000000" "00000000" "00000000" "00000000" "00000000"
"00000000" "00000000" "00000000" "00000000" "00000000" "00000000" "00000000" "00000000"
"00000000" "00000000" "00000000" "00000000" "00000000" "00000000" "00000000" "00000000"
"00000000" "00000000" "00000000" "00000000" "00000000" "00000000" "00000000" "00000000"
"00000000" "00000000" "00000000" "00000000" "00000000" "00000000" "00000000" "00000000"
"00000000" "00000000" "00000000" "00000000" "00000000" "00000000" "00000000" "00000000"
```

LUT 15 – Output Block Memory.

Code Converter

```
"00000" "00000" "00000" "00110" "01010" "00000" "10110" "10010"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"00100" "00001" "00100" "00100" "01110" "10110" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"01000" "00001" "01000" "01000" "01000" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"01100" "00001" "01010" "01100" "00000" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"10000" "00001" "10000" "00000" "00000" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"10100" "00001" "10100" "00000" "00000" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
```

LUT 16 – Code Converter Memory.

Mealy HFSM model 3 with the binary state encoding algorithm

Macrooperation z_1 RE

```
"001000000000" "001000000000" "001000000000" "001000000000"
"001000000000" "001000000000" "001000000000" "001000000000"
"001000000000" "001000000000" "001000000000" "001000000000"
"001000000000" "001000000000" "001000000000" "001000000000"
"001100000010" "001100000010" "001100000010" "001100000010"
"001100000010" "001100000010" "001100000010" "001100000010"
"010100000000" "010100000000" "100000000000" "100000000000"
"010000000000" "010000000000" "011111100000" "011111100000"
"011000010100" "011000010100" "011000010100" "011000010100"
"011000010100" "011000010100" "011000010100" "011000010100"
"011000000000" "011000000000" "011000000000" "011000000000"
"011000000000" "011000000000" "011000000000" "011000000000"
"000000001001" "000000001001" "000000001001" "000000001001"
"000000001001" "000000001001" "000000001001" "000000001001"
"011000000000" "100100000000" "011000000000" "100100000000"
"011000000000" "100100000000" "011000000000" "100100000000"
"100100000000" "100100000000" "100100000000" "100100000000"
"100100000000" "100100000000" "100100000000" "100100000000"
"000000000000" "000000000000" "000000000000" "000000000000"
"000000000000" "000000000000" "000000000000" "000000000000"
"000000000000" "000000000000" "000000000000" "000000000000"
"000000000000" "000000000000" "000000000000" "000000000000"
"000000000000" "000000000000" "000000000000" "000000000000"
"000000000000" "000000000000" "000000000000" "000000000000"
"000000000000" "000000000000" "000000000000" "000000000000"
"000000000000" "000000000000" "000000000000" "000000000000"
"000000000000" "000000000000" "000000000000" "000000000000"
"000000000000" "000000000000" "000000000000" "000000000000"
"000000000000" "000000000000" "000000000000" "000000000000"
"000000000000" "000000000000" "000000000000" "000000000000"
"000000000000" "000000000000" "000000000000" "000000000000"
"000000000000" "000000000000" "000000000000" "000000000000"
```

LUT 17 – Macrooperation z_1 RE Memory.

Macrooperation z_2 RE

```
"0010000" "0010000" "0010000" "0010000" "0010000" "0010000" "0010000" "0010000"
"0011011" "0011011" "0011011" "0011011" "0100100" "0100100" "0100100" "0100100"
"0101000" "0101000" "0100100" "0100100" "0110000" "0110000" "0110000" "0110000"
"0010000" "0001000" "0010000" "0001000" "0000000" "0000000" "0000000" "0000000"
"0000000" "0000000" "0000000" "0000000" "0000000" "0000000" "0000000" "0000000"
"0000000" "0000000" "0000000" "0000000" "0000000" "0000000" "0000000" "0000000"
"0000000" "0000000" "0000000" "0000000" "0000000" "0000000" "0000000" "0000000"
"0000000" "0000000" "0000000" "0000000" "0000000" "0000000" "0000000" "0000000"
```

LUT 18 – Macrooperation z_2 RE Memory.

Macrooperation z_3 RE

```
"0010000" "0010000" "0010000" "0010000" "0010000" "0010000" "0010000" "0010000"
"0001001" "0011110" "0001001" "0011110" "0001000" "0001000" "0001001" "0001001"
"0000000" "0000000" "0000000" "0000000" "0000000" "0000000" "0000000" "0000000"
"0000000" "0000000" "0000000" "0000000" "0000000" "0000000" "0000000" "0000000"
"0000000" "0000000" "0000000" "0000000" "0000000" "0000000" "0000000" "0000000"
"0000000" "0000000" "0000000" "0000000" "0000000" "0000000" "0000000" "0000000"
"0000000" "0000000" "0000000" "0000000" "0000000" "0000000" "0000000" "0000000"
"0000000" "0000000" "0000000" "0000000" "0000000" "0000000" "0000000" "0000000"
```

LUT 19 – Macrooperation z_3 RE Memory.

Macrooperation z_4 RE

```
"001000" "001000" "001000" "001000" "001100" "001100" "010011" "010011"
"001000" "000100" "000000" "000000" "000000" "000000" "000000" "000000"
"000000" "000000" "000000" "000000" "000000" "000000" "000000" "000000"
"000000" "000000" "000000" "000000" "000000" "000000" "000000" "000000"
```

LUT 20 – Macrooperation z_4 RE Memory.

Code Converter

```
"00000" "00000" "00000" "00000" "00110" "01010" "00000" "10110"
"10010" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"00100" "00001" "00100" "00100" "00100" "01110" "10110" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"01000" "00001" "01000" "01000" "00000" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"01100" "00001" "01010" "01100" "01100" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"10000" "00001" "10000" "00000" "00000" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"10100" "00001" "10100" "00000" "00000" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
```

LUT 21 – Code Converter Memory.

Moore PFSM with the binary state encoding algorithm

Next State Memory

```
"001" "001" "001" "001" "001" "001" "001" "001" "011" "000" "010" "110" "011" "000" "010" "110"
"100" "100" "100" "100" "100" "100" "100" "100" "101" "101" "101" "101" "101" "101" "101" "101"
"101" "100" "100" "100" "101" "100" "100" "100" "000" "000" "000" "000" "000" "000" "000" "000"
"111" "111" "111" "111" "111" "111" "111" "111" "000" "111" "000" "111" "000" "111" "000" "111"
"000" "001" "000" "001" "000" "001" "000" "001" "010" "010" "010" "010" "010" "010" "010" "010"
"011" "010" "011" "010" "011" "010" "011" "010" "100" "100" "100" "100" "100" "100" "100" "100"
"101" "101" "101" "101" "101" "101" "101" "101" "110" "101" "110" "101" "110" "101" "110" "101"
"001" "111" "001" "111" "001" "111" "001" "111" "000" "000" "000" "000" "000" "000" "000" "000"
"000" "001" "000" "001" "000" "001" "000" "001" "010" "010" "010" "010" "010" "010" "010" "010"
"000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000"
"000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000"
"000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000"
"000" "000" "000" "000" "001" "001" "001" "010" "000" "000" "000" "000" "000" "000" "000" "000"
"000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000"
"000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000"
"000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000"
```

LUT 22 – Next State Memory.

Moore PFSM Output Memory

```
"0000000000000000" "0000000100000000" "0000000011000000" "0000001000000000"
"0000000000000000" "0000110000000000" "0000000010000000" "0000000000000000"
"0000000000000000" "0000110000000000" "1100000000000000" "0010000100000000"
"0000000000100000" "0000000000000000" "0000000000000000" "0000000000010000"
"0000000000000000" "0000001100000000" "0011000000001000" "0000000000000000"
"0000000000000000" "0000000000000000" "0000000000000000" "0000000000000000"
"0000000000000000" "0000000000000101" "0000000000000110" "0000000000000000"
"0000000000000000" "0000000000000000" "0000000000000000" "0000000000000000"
```

LUT 23 – Output Memory.

Moore PHFSM with the binary state encoding algorithm

Code Converter

```
"00000" "00001" "00111" "01001" "01101" "10000" "10010" "00000"
```

LUT 24 – Code Converter Memory.

Output Memory

```
"000000000000000000000" "000000010000000000000" "000000000000011000000" "000000000000000000000"
"000000000000001100000" "000000000000000000000" "000000000101100000000" "000000110111000000000"
"000000000000000010000" "000000000000000000000" "000011000000000000000" "000000000000000001000"
"000000000001000000000" "000000001101000000000" "001100000000000000000" "000000110000000000100"
"000000000111000000000" "110000000000100000000" "000000000000000000000" "000000000000100000001"
"000000000000100000010" "000000000000000000000" "000000000000000000000" "000000000000000000000"
"000000000000000000000" "000000000000000000000" "000000000000000000000" "000000000000000000000"
"000000000000000000000" "000000000000000000000" "000000000000000000000" "000000000000000000000"
```

LUT 25 – Output Memory.

Next State Memory

```
"00001" "00001" "00000" "00111" "00000" "01001" "00000" "01101"
"00100" "00000" "00010" "00110" "00000" "00000" "00000" "00000"
"00011" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"00000" "00011" "00011" "00011" "00000" "00000" "00000" "00000"
"00101" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"00000" "00101" "00101" "00101" "00000" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"01000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"01010" "01100" "01010" "01100" "01010" "01011" "01010" "01100"
"01100" "01100" "01011" "01100" "00000" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"01110" "01111" "00000" "00000" "00000" "00000" "00000" "00000"
"01111" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"10001" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"10011" "10011" "10011" "10100" "00000" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
"00000" "00000" "00000" "00000" "00000" "00000" "00000" "00000"
```

LUT 26 – Next State Memory.

Moore HFSM model 3 - Providing Extensibility

Macrooperation z_7 RE

```
"0010" "0010" "0010" "0010" "0010" "0010" "0010" "0010"
"0010" "0001" "0010" "0010" "0000" "0000" "0000" "0000"
"0000" "0000" "0000" "0000" "0000" "0000" "0000" "0000"
"0000" "0000" "0000" "0000" "0000" "0000" "0000" "0000"
"0000" "0000" "0000" "0000" "0000" "0000" "0000" "0000"
"0000" "0000" "0000" "0000" "0000" "0000" "0000" "0000"
"0000" "0000" "0000" "0000" "0000" "0000" "0000" "0000"
"0000" "0000" "0000" "0000" "0000" "0000" "0000" "0000"
```

LUT 27 – Macrooperation z_7 RE Memory.

10 APPENDIX B - SIMULHGS

This appendix lists the C++ code of the tool **SIMULHGS**.

```
//SIMULHGS - Verification, Simulation and Automatic Synthesis of HGSs

#include <fstream.h>
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <conio.h>

//********************** Definition of Constants, Types and String Functions **********************

#define MAXNODES 100
#define MAXHGS 10

enum TYPENODE {BEGIN, END, ASSIGN, MICROOP, MACROOP, CONDITION, FUNCTION};
enum TYPEGRAPH {LFUNCGS, MACROGS};

char* binarycode(int n, int G)
{
    static char* bincode=new char[6]; strcpy(bincode,"");
    for (int i=G, bit=1<<G; i>0; i--) { bit>>=1; if (n&bit) strcat(bincode,"1"); else strcat(bincode,"0"); }
    return bincode;
}

char* yzstring(int n, int G)
{
    static char* yz=new char[20];     strcpy(yz,"");
    for (int i=G, bit=1<<G; i>0; i--) {
        bit>>=1; if (n&bit) { strcat(yz,"yz"); strcat(yz,ecvt(i,1,0,0)); strcat(yz," "); }
    }
    return yz;
}

char* linechar(int n)
{
    static char* line=new char[3]; strcpy(line,"");
    if (n<9) { line=ecvt(n,1,0,0); strcat(line," "); } else line=ecvt(n,2,0,0);
    return line;
}

class Node; class Graphscheme; class Hgraphscheme;

//**************************** Definition of the Class Graph-Scheme ****************************

class Graphscheme
{
    char* gsname;
    TYPEGRAPH gstype;
    unsigned int gsnnodes;
    Node* gslist[MAXNODES];
    unsigned int gsmain;
    unsigned int gsnstates;

public:
    Graphscheme(char* fname, unsigned int main=0);
    ~Graphscheme();
    char*   Graphname(void) {return gsname;}
    TYPEGRAPH Graphtype(void) const { return gstype; }
    unsigned int Graphnnodes(void) const { return gsnnodes; }
    unsigned int Graphmain(void) const { return gsmain; }
    unsigned int Graphnstates(void) const { return gsnstates; }
    Node* Begings(void) const ;
    Node* Endgs(void) const ;
    void Cleanmarkgs(void);
    void Markgs(void);
    int Primarycheckgs(Hgraphscheme*);
    int Checkgs(Hgraphscheme*);
    int Loopcheckgs(void);
    void Listgs(void);
    void Printgs(char*);
    int Rungs(Hgraphscheme*,int&);
    void Cleanstategs(void);
    void Moorestategs(int&, int);
    void Mealystategs(int&, int);
    Node* Nodeinitialstategs(void);
    void Mooretablemergegs(char*);
    void Mooretablesplitgs(char*);
    void Mealytablemergegs(char*);
    void Mealytablesplitgs(char*);
    void CCsplitgs(char*,char*);
    void Insertstategs(int&, int, int&);
    void Removestategs(int&, int, int&);
};

//********************** Definition of the Class Hierarchical Graph-Scheme **********************

class Hgraphscheme
{
    char* hgsname;
    unsigned int hgsngs;
    Graphscheme* hgslist[MAXHGS];
    unsigned int hgscheck;
    unsigned int hgsdeep;
    char* hgssyn;
    int hgsmark;
    unsigned int hgsnstates;

public:
    Hgraphscheme(char* fname);
    ~Hgraphscheme();
    char*   Hgsname(void) { return hgsname; }
    unsigned int Hgsngs(void) const { return hgsngs; }
    char* Ngraphname(int n) { return hgslist[n]->Graphname(); }
    Graphscheme* Searchgraphhgs(char* name);
    int Checkhgs(void);
    void Listhgs(void);
    void Printhgs(void);
    void Deeplevelhgs(int level) { if (level>hgsdeep) hgsdeep=level; }
    void Runhgs(void);
    void Moorestatehgs(void);
    void Mealystatehgs(void);
    void CCmergehgs(char*);
    void CCsplithgs(char*);
    void Mooretablemergehgs(char*);
    void Mooretablesplithgs(char*);
    void Mealytablemergehgs(char*);
    void Mealytablesplithgs(char*);
    void Mooresynmergehgs(int statemark=1);
    void Mooresynsplithgs(int statemark=1);
    void Mealysynmergehgs(int statemark=1);
    void Mealysynsplithgs(int statemark=1);
    void Insertstatehgs(void);
    void Removestatehgs(void);
};
```

```
//************************ Definition and Implementation of the Class Node ************************

class Node
{
protected:
    char*   nname;
    TYPENODE ntype;
    int nmark;
    int nstate;
    int nauto;

public:
    Node(char* name);
    Node(char* name, TYPENODE type);
    Node(const Node&);
    virtual ~Node() { delete [] nname; }
    const Node& operator=(const Node&);
    int operator==(const Node&) const;
    char*   Nodename(void) { return nname; }
    TYPENODE Nodetype(void) { return ntype; }
    int Nodemark(void) { return nmark; }
    int Nodestate(void) { return nstate; }
    int Nodeauto(void) { return nauto; }
    void Cleanmarknode(void) { nmark=0; }
    void Setmarknode(void) { nmark=1; }
    void Cleanstatenode(void) { nstate=-1; }
    void Setstatenode(int&, int autman=1);
    void Needstatenode(void) { nstate=-3; }
    void Ignorestatenode(void) { if (nstate==-1) nstate=-2; }
    char*   Macroname(void);
    char*   Microname(void);
    int Assignvalue(void);
    char* Statechar(void);
    virtual Node* Nextnode(int out=1)=0;
    virtual void Setnext(Node* tpt, Node* fpt=NULL)=0;
    virtual void Marknode(void)=0;
    virtual void Checknode(int&)=0;
    virtual void Loopnode(int&)=0;
    virtual void Listnode(void);
    virtual void Printnode(char*);
    virtual void Runnode(int&, Hgraphscheme*, int&)=0;
    int Startingstatenode(void) { if (nstate>1) return 1; else return 0; }
    virtual void Moorestatenode(int&, Graphscheme* gs, int)=0;
    virtual void Mealystatenode(int&, Graphscheme* gs, int)=0;
    virtual void Mooretablenode(char*, char*, int&, int&, char*, int, Node*);
    virtual void Mealytablenode(char*, char*, char*, int&, int&, int, Node*);
};

Node::Node(char* name)
    :nmark(0), nstate(-1), nauto(0)
{
    nname=new char[15]; strcpy(nname,name);
    if (! strcmp(name,"BEGIN")) ntype=BEGIN;
    else if (! strcmp(name,"END")) ntype=END;
}

Node::Node(char* name, TYPENODE type)
    :nmark(0), nstate(-1), nauto(0)
{ nname=new char[15]; strcpy(nname,name); ntype=type; }
```

```
Node::Node(const Node& node)
{
    nname=new char[15]; strcpy(nname,node.nname);
    ntype=node.ntype; nmark=node.nmark; nstate=node.nstate;
}

const Node& Node::operator=(const Node& node)
{
    if (this != &node ) {
        delete [] nname; nname=new char[15]; strcpy(nname,node.nname);
        ntype=node.ntype; nmark=node.nmark; nstate=node.nstate;
    }
    return *this;
}

int Node::operator==(const Node& node) const
{
    return (! strcmp(this->nname,node.nname) && (this->ntype==node.ntype)
            && (this->nmark==node.nmark) && (this->nstate==node.nstate) );
}

void Node::Setstatenode(int& state, int autman)
{ if (ntype!=END && (nstate==-1 || nstate==-3)) { nstate=state; state++; nauto=autman; } }

char*   Node::Macroname(void)
{
    static char* macroname=new char[15];

    strcpy(macroname,"");
    if (ntype == FUNCTION) strcpy(macroname,nname);
    else if (ntype == MACROOP) { strcpy(macroname,nname); macroname=strstr(macroname,"z"); }
    return macroname;
}

char*   Node::Microname(void)
{
    static char* microname=new char[15];

    strcpy(microname,"");
    if (ntype == MICROOP) strcpy(microname,nname);
    else if (ntype == MACROOP) { int n=strcspn(nname,"z"); if (n) strncat(microname,nname,n-1); }
    return microname;
}

int Node::Assignvalue(void)
{
    static char* assignvalue=new char[15];
    strcpy(assignvalue,nname); assignvalue=strstr(assignvalue,"=");
    strnset(assignvalue,'0',1); return atoi(assignvalue);
}

char* Node::Statechar(void)
{
    static char* stch=new char[4];
    if (nstate<10) { stch=ecvt(nstate,1,0,0); strcat(stch,"  "); }
    else if (nstate<100) { stch=ecvt(nstate,2,0,0); strcat(stch," "); }
         else if (nstate<1000) stch=ecvt(nstate,3,0,0);
    return stch;
}
```

```
void Node::Listnode(void)
{
    switch (ntype) {
        case BEGIN          : cout <<"BEGIN Node" ; break;
        case END            : cout <<"END Node" ; break;
        case CONDITION      : cout <<"Conditional Node " <<nname; break;
        case FUNCTION       : cout <<"Logic Function Node " <<nname; break;
        case ASSIGN         : cout <<"Assignment Node " <<nname; break;
        case MICROOP        :
        case MACROOP        : cout <<"Operational Node " <<nname; break;
    }
    if (nstate!=-1) cout <<" State a" <<nstate; cout <<"\n";
}

void Node::Printnode(char* fname)
{
    fstream pf; pf.open(fname,ios::app);
    switch (ntype) {
        case BEGIN          : pf.write("BEGIN Node",10); break;
        case END            : pf.write("END Node",8); break;
        case CONDITION      : pf.write("Conditional Node ",17);
                              pf.write(nname,strlen(nname)); break;
        case FUNCTION       : pf.write("Logic Function Node ",20);
                              pf.write(nname,strlen(nname)); break;
        case ASSIGN         : pf.write("Assignment Node ",16);
                              pf.write(nname,strlen(nname)); break;
        case MICROOP        :
        case MACROOP        : pf.write("Operational Node ",17);
                              pf.write(nname,strlen(nname)); break;
    }
    if (nstate!=-1) { pf.write(" State a",8); pf.write(Statechar(),3);}
    pf.write("\n",1); pf.close();
}

void Node::Mealytablenode(char* fname, char* transition, char* output, int& nt, int& micro, int, Node*
start)
{
    int i, nsp;
    fstream pf; pf.open(fname,ios::app);
    if (nmark) {
        pf.write("\n------------------------------------------------",49);
        pf.write("\n|a",3); pf.write(Statechar(),3); nmark=0;
    }
    else
        if (ntype!=MICROOP || micro) {
            if (nt>1) pf.write("\n|    ",6); nt++; pf.write("|a",2);
            if (nstate!=-1) pf.write(Statechar(),3); else pf.write(start->Statechar(),3); pf.write("|",1);
            if (strcmp(transition,"")) {
                pf.write(" ",1); nsp=strlen(transition); pf.write(transition,nsp);
                nsp=17-nsp; for (i=0; i<nsp; i++) pf.write(" ",1);
            }
            else pf.write(" 1                ",18);
            if (strcmp(output,"")) {
                pf.write("| ",2); nsp=strlen(output); pf.write(output,nsp); nsp=15-nsp;
                for (i=0; i<nsp; i++) pf.write(" ",1); pf.write("|",1); strcpy(output,"");
            }
            else pf.write("| --             |",18);
            micro=0; if (nstate!=-1) nmark=1;
        }
    pf.close();
}
```

```
void Node::Mooretablenode(char* fname, char* transition, int& extray, int& nt, char* logicf, int
mergesplit, Node* start)
{
    int i, nsp;
    fstream pf; pf.open(fname,ios::app);
    if (nmark) {
        pf.write("\n----------------------------------------------------------",59);
        pf.write("\n|a",3); pf.write(Statechar(),3);
        switch (ntype) {
            case BEGIN          :
            case CONDITION      : pf.write(", --              ",18);
                                  break;
            case ASSIGN         : if (Assignvalue()) pf.write(", yextra          ",18);
                                  else for (i=0; i<18; i++) pf.write(" ",1);
                                  break;
            case MICROOP        : if (strcmp(nname,"y0")) {
                                     pf.write(", ",2); nsp=strlen(nname); pf.write(nname,nsp); nsp=16-nsp;
                                     for (i=0; i<nsp; i++) pf.write(" ",1);
                                  }
                                  else pf.write(", --              ",18);
                                  break;
            case FUNCTION       : if (!mergesplit) { pf.write(", --              ",18); break; }
            case MACROOP        : if (!mergesplit) {
                                     nsp=strlen(Microname());
                                     if (nsp) {
                                         pf.write(", ",2); pf.write(Microname(),nsp);
                                         nsp=16-nsp; for (i=0; i<nsp; i++) pf.write(" ",1);
                                     }
                                     else pf.write(", --              ",18);
                                  }
                                  else {
                                     pf.write(", ",2); nsp=strlen(nname); pf.write(nname,nsp);
                                     pf.write(" , y+",5); nsp=11-nsp; for (i=0; i<nsp; i++) pf.write(" ",1);
                                  }
            case END            : break;
        }
        nmark=0;
    }
    else {
        if (nt>1)
            if (strcmp(logicf,"")) {
                pf.write("\n|",2); nsp=strlen(logicf); pf.write(logicf,nsp);
                nsp=22-nsp; for (i=0; i<nsp; i++) pf.write(" ",1); strcpy(logicf,"");
            }
            else pf.write("\n|                      ",24);
        nt++; pf.write("|a",2);
        if (nstate!=-1) pf.write(Statechar(),3); else pf.write(start->Statechar(),3);
        pf.write("|",1);
        if (strcmp(transition,"")) {
            pf.write(" ",1); nsp=strlen(transition); pf.write(transition,nsp);
            nsp=17-nsp; for (i=0; i<nsp; i++) pf.write(" ",1);
        }
        else pf.write(" 1                ",18);
        if (extray) { pf.write("| yextra |",10); extray=0; } else pf.write("| --     |",10);
        if (nstate!=-1) nmark=1;
    }
    pf.close();
}
```

```
//********************** Definition and Implementation of the Class Onode **********************

class Onode : virtual public Node
{
    Node* p;

public:
    Onode(char* name);
    Onode(char* name, TYPENODE type);
    Onode(const Onode&);
    ~Onode() {};
    const Onode& operator=(const Onode&);
    int operator==(const Onode&) const;
    Node* Nextnode(int out=1) { return this->p; }
    void Setnext(Node* tpt, Node* fpt=NULL) { p=tpt; }
    void Marknode(void);
    void Checknode(int&);
    void Loopnode(int&);
    void Listnode(void);
    void Printnode(char*);
    void Runnode(int&, Hgraphscheme*, int&);
    void Moorestatenode(int&, Graphscheme*, int);
    void Mealystatenode(int&, Graphscheme*, int);
    void Mooretablenode(char*, char*, int&, int&, char*, int, Node*);
    void Mealytablenode(char*, char*, char*, int&, int&, int, Node*);
};

Onode::Onode(char* name)
    :Node(name),p(NULL) {}

Onode::Onode(char* name, TYPENODE type)
    :Node(name,type),p(NULL) {}

Onode::Onode(const Onode& onode)
    :Node(onode) {p=onode.p;}

const Onode& Onode::operator=(const Onode& onode)
{
    if (this != &onode ) {
        delete [] nname; nname=new char[15]; strcpy(nname,onode.nname);
        ntype=onode.ntype; nmark=onode.nmark; nstate=onode.nstate; p=onode.p;
    }
    return *this;
}

int Onode::operator==(const Onode& onode) const
{
    return (! strcmp(this->nname,onode.nname)&& (this->ntype==onode.ntype)
            && (this->nmark==onode.nmark) && (this->nstate==onode.nstate)
            && (this->p==onode.p) );
}

void Onode::Marknode(void)
{ if (!nmark) { nmark=1; if (ntype!=END) p->Marknode(); } }

void Onode::Listnode(void)
{
    Node::Listnode();
    if (!nmark) { nmark=1; if (ntype != END) { cout << nname <<" Next Node -> "; p->Listnode(); } }
}
```

```
void Onode::Checknode(int& ck)
{
    if (!nmark) { cout <<"\nOperational node " <<nname <<" not reachable"; ck=0; }
    if (this==p) { cout <<"\nOperational node " <<nname <<" in loop"; ck=0; }
    else if (p->Nodetype()==BEGIN && ntype!=END)
                { cout <<"\nOperational node " <<nname <<" pointing to BEGIN node"; ck=0; }
    if (ntype==ASSIGN) {
        int value=Assignvalue();
        if (value!=0 && value!=1)
            { cout <<"\nAssignment node " <<nname <<" with a wrong value"; ck=0; }
    }
}

void Onode::Loopnode(int& noloop)
{ if (!nmark) { nmark=1; if (ntype != END) p->Loopnode(noloop); else noloop=1; } }

void Onode::Printnode(char* fname)
{
    fstream pf;
    Node::Printnode(fname);
    if (!nmark) {
        nmark=1;
        if (ntype != END) {
            pf.open(fname,ios::app); pf.write(nname,strlen(nname));
            pf.write(" Next Node -> ",14); pf.close(); p->Printnode(fname);
        }
    }
}

void Onode::Runnode(int& retval, Hgraphscheme* hgs, int& deep)
{
    switch (ntype) {
        case BEGIN   : cout <<"BEGIN Graph-Scheme Execution\n";
                       deep++; hgs->Deeplevelhgs(deep); break;
        case END     : cout <<"END Graph-Scheme Execution\n" ; deep--; break;
        case ASSIGN  : cout <<"Assignment Node " <<nname <<"\n";
                       retval=Assignvalue(); break;
        case MICROOP : cout <<"Activate output signal(s) " <<nname <<"\n"; break;
        case MACROOP : if (strcmp(Microname(),""))
                              cout <<"Activate output signal(s) " <<Microname() <<" & ";
                       cout <<"Execute macro operation " <<Macroname() <<"\n";
                       Graphscheme* ngraph=hgs->Searchgraphhgs(Macroname());
                       ngraph->Rungs(hgs,deep); break;
    }
    if (ntype != END) p->Runnode(retval,hgs,deep);
}

void Onode::Moorestatenode(int& state, Graphscheme* gs, int mergesplit)
{
    switch (ntype) {
        case BEGIN   : if (gs->Graphmain())
                             { nstate=0; if (p->Nodetype()==CONDITION) p->Setstatenode(state); }
                       else if (p->Nodetype()==CONDITION) Setstatenode(state);
                       if (p->Nodetype()==END && !mergesplit) nstate=2;
                       if (p->Nodetype()==FUNCTION) p->Needstatenode(); break;
        case END     : if (gs->Graphmain()) nstate=0; else nstate=1; break;
        case MICROOP : if (p->Nodetype()==FUNCTION) p->Ignorestatenode();
                       Setstatenode(state); break;
        case MACROOP : if (p->Nodetype()==FUNCTION) p->Needstatenode();
        case ASSIGN  : Setstatenode(state); break;
    }
}
```

```
void Onode::Mealystatenode(int& state, Graphscheme* gs, int mergesplit)
{
    switch (ntype) {
        case BEGIN       : if (gs->Graphmain()) nstate=0;
                            else if (p->Nodetype()==END && !mergesplit) nstate=2;
                            p->Setstatenode(state); break;
        case END         : if (gs->Graphmain()) nstate=0; else nstate=1; break;
        case MACROOP : Setstatenode(state);
        case MICROOP   :
        case ASSIGN      : p->Setstatenode(state); break;
    }
}

void Onode::Mooretablenode(char* fname, char* transition, int& extray, int& nt, char* logicf, int
mergesplit, Node* start)
{
    if (nstate==-1 && ntype==ASSIGN) extray=Assignvalue();
    else {  Node::Mooretablenode(fname,transition,extray,nt,logicf,mergesplit,start);
            if (ntype==MICROOP && !nmark && p->Nodetype()==FUNCTION
                 && p->Nodestate()==-1 && mergesplit) {
                strcpy(logicf,p->Nodename()); strcat(logicf," , y+");
            }
    }
    if (!nmark) p->Mooretablenode(fname,transition,extray,nt,logicf,mergesplit,start); else nmark=0;
}

void Onode::Mealytablenode(char* fname, char* transition, char* output, int& nt, int& micro, int
mergesplit, Node* start)
{
    if (nstate>-1) Node::Mealytablenode(fname,transition,output,nt,micro,mergesplit,start);
    if (!nmark) {
        switch (ntype) {
            case BEGIN       :
            case END         : break;
            case ASSIGN      : if (Assignvalue()) strcpy(output,"yextra"); break;
            case MICROOP   : if (strcmp(nname,"y0")) strcpy(output,nname); else strcpy(output,"");
                              micro++; break;
            case MACROOP : if (!mergesplit) strcpy(output,Microname());
                              else  { strcpy(output,nname); strcat(output," , y+"); }
                              micro++; break;
        }
        p->Mealytablenode(fname,transition,output,nt,micro,mergesplit,start);
    }
    else nmark=0;
}

//********************** Definition and Implementation of the Class Cnode **********************

class Cnode : virtual public Node
{
    Node* truep; Node* falsep;

public:
    Cnode(char* name, TYPENODE type);
    Cnode(const Cnode&);
    ~Cnode() {};
    const Cnode& operator=(const Cnode&);
    int operator==(const Cnode&) const;
    Node* Nextnode(int out=1);
    void Setnext(Node* tpt, Node* fpt) { truep=tpt; falsep=fpt; }
    void Marknode(void);
    void Checknode(int&);
    void Loopnode(int&);
    void Listnode(void);
    void Printnode(char*);
    void Runnode(int&, Hgraphscheme*, int&);
    void Moorestatenode(int&, Graphscheme* gs, int);
    void Mealystatenode(int&, Graphscheme* gs, int);
    void Mooretablenode(char*, char*, int&, int&, char*, int, Node*);
    void Mealytablenode(char*, char*, char*, int&, int&, int, Node*);
};

Cnode::Cnode(char* name, TYPENODE type)
    :Node(name,type),truep(NULL),falsep(NULL) {}

Cnode::Cnode(const Cnode& cnode)
    :Node(cnode) {truep=cnode.truep; falsep=cnode.falsep;}

const Cnode& Cnode::operator=(const Cnode& cnode)
{
    if (this != &cnode ) {
        delete [] nname; nname=new char[15]; strcpy(nname,cnode.nname); ntype=cnode.ntype;
        nmark=cnode.nmark; nstate=cnode.nstate; truep=cnode.truep; falsep=cnode.falsep;
    }
    return *this;
}

int Cnode::operator==(const Cnode& cnode) const
{
    return (! strcmp(this->nname,cnode.nname) && (this->ntype==cnode.ntype)
            && (this->nmark==cnode.nmark) && ( this->nstate==cnode.nstate)
            && (this->truep==cnode.truep) && (this->falsep==cnode.falsep) );
}

Node* Cnode::Nextnode(int out)
{ if (out) return this->truep; else return this->falsep; }

void Cnode::Marknode(void)
{ if (!nmark) { nmark=1; truep->Marknode(); falsep->Marknode(); } }

void Cnode::Checknode(int& ck)
{
    if (!nmark) { cout <<"\nConditional node " <<nname <<" not reachable"; ck=0; }
    if (this==truep && this==falsep) { cout <<"\nConditional node " <<nname <<" in loop"; ck=0; }
    else if (truep==falsep && truep->Nodetype()!=BEGIN)
                cout <<"\nWARNING useless conditional node " <<nname <<"   ";
        else if (truep->Nodetype()==BEGIN || falsep->Nodetype()==BEGIN)
                { cout <<"\nConditional node " <<nname <<" pointing to BEGIN node"; ck=0; }
    if (! strcmp(truep->Nodename(),nname) && truep!=this ||
        ! strcmp(falsep->Nodename(),nname) && falsep!=this)
        { cout <<"\nTwo conditional nodes " <<nname <<" follow each other"; ck=0; }
}

void Cnode::Loopnode(int& noloop)
{ if (!nmark) { nmark=1; truep->Loopnode(noloop); if (!noloop) falsep->Loopnode(noloop); } }

void Cnode::Listnode(void)
{
    Node::Listnode();
    if (!nmark) {
        nmark=1; cout <<nname <<" Next Node when true  -> "; truep->Listnode();
        cout <<nname <<" Next Node when false -> "; falsep->Listnode();
    }
}
```

```
void Cnode::Printnode(char* fname)
{
    fstream pf;
    Node::Printnode(fname);
    if (!nmark) {
        nmark=1; pf.open(fname,ios::app); pf.write(nname,strlen(nname));
        pf.write(" Next Node when true  -> ",25); pf.close(); truep->Printnode(fname);
        pf.open(fname,ios::app); pf.write(nname,strlen(nname));
        pf.write(" Next Node when false -> ",25); pf.close(); falsep->Printnode(fname);
    }
}

void Cnode::Runnode(int& retval, Hgraphscheme* hgs, int& deep)
{
    int answer;
    switch (ntype) {
        case CONDITION   : cout <<"Read input condition " <<nname;
                           do { cout <<" [0/1]="; cin >> answer; }
                           while (answer != 0 && answer != 1); break;
        case FUNCTION    : cout <<"Execute logic function " <<nname <<"\n";
                           Graphscheme* ngraph=hgs->Searchgraphhgs(nname);
                           answer=ngraph->Rungs(hgs,deep); break;
    }
    if (answer) truep->Runnode(retval,hgs,deep); else falsep->Runnode(retval,hgs,deep);
}

void Cnode::Moorestatenode(int&, Graphscheme*, int)
{
    if (truep->Nodetype()==FUNCTION) truep->Needstatenode();
    if (falsep->Nodetype()==FUNCTION) falsep->Needstatenode();
}

void Cnode::Mealystatenode(int& state, Graphscheme*, int)
{
    switch (ntype) {
        case FUNCTION    : Setstatenode(state); truep->Setstatenode(state); falsep->Setstatenode(state);
        case CONDITION   : break;
    }
}

void Cnode::Mooretablenode(char* fname, char* transition, int& extray, int& nt, char* logicf, int
mergesplit, Node* start)
{   char* localt=new char[15];
    if (nstate>-1) Node::Mooretablenode(fname,transition,extray,nt,logicf,mergesplit,start);
    if (!nmark) {
        strcpy(localt,transition);
        if (strstr(localt,nname)) Node::Mooretablenode(fname,transition,extray,nt,logicf,mergesplit,start);
        else {
            switch (ntype) {
                case CONDITION   : strcat(transition,nname); break;
                case FUNCTION    : strcat(transition,"xextra"); break; }
            truep->Mooretablenode(fname,transition,extray,nt,logicf,mergesplit,start);
            strcpy(transition,localt);
            switch (ntype) {
                case CONDITION   : strcat(transition,"~");strcat(transition,nname); break;
                case FUNCTION    : strcat(transition,"~xextra"); break; }
            falsep->Mooretablenode(fname,transition,extray,nt,logicf,mergesplit,start);
            strcpy(transition,localt);
        }
    }
    else nmark=0; delete [] localt;
}
```

```
void Cnode::Mealytablenode(char* fname, char* transition, char* output, int& nt, int& micro, int
mergesplit, Node* start)
{
    char* localt=new char[15];
    if (nstate>-1) Node::Mealytablenode(fname,transition,output,nt,micro,mergesplit,start);
    if (!nmark) {
        strcpy(localt,transition);
        if (strstr(localt,nname)) Node::Mealytablenode(fname,transition,output,nt,micro,mergesplit,start);
        else {
            switch (ntype) {
                case CONDITION   : strcat(transition,nname); break;
                case FUNCTION    : strcat(transition,"xextra");
                                   if (mergesplit) { strcpy(output,nname); strcat(output," , y+"); }
                                   micro++; break;
            }
            truep->Mealytablenode(fname,transition,output,nt,micro,mergesplit,start);
            strcpy(transition,localt);
            switch (ntype) {
                case CONDITION   : strcat(transition,"~");strcat(transition,nname); break;
                case FUNCTION    : strcat(transition,"~xextra");
                                   if (mergesplit) { strcpy(output,nname); strcat(output," , y+"); }
                                   micro++; break;
            }
            falsep->Mealytablenode(fname,transition,output,nt,micro,mergesplit,start);
            strcpy(transition,localt);
        }
    }
    else nmark=0; delete [] localt;
}

//************************* Implementation of the Class Graph-Scheme *************************

Graphscheme::Graphscheme(char* filename, unsigned int main)
{   gsname=new char[5]; ifstream gsfile(filename);
    if (!gsfile) { cout <<"\n*** Cannot open Graph-Scheme " <<filename <<" ***\n"; gsnnodes=0; }
    else {
        int i=0,next[MAXNODES][2]; char opercond[MAXNODES],extra; TYPENODE type;
        char *string=new char[30], *name=new char[15], *number=new char[5];
        gsfile.get(string,30); gsfile.get(extra); string=strupr(string); strcpy(gsname,string);
        if (strstr(string,"F")) gstype=LFUNCGS; else gstype=MACROGS;
        while (!gsfile.eof()) {
            gsfile.get(string,30); gsfile.get(extra); string=strupr(string);
            if ( strcmp(string,"")) {
                opercond[i]=string[0]; name=strtok(string," "); name=strtok(NULL," ");
                if (opercond[i]=='O') {
                    number=strtok(NULL," "); next[i][0]=atoi(number); next[i][1]= 0;
                    if (next[i][0]) next[i][0]--;
                    if (! strcmp(name,"BEGIN") || ! strcmp(name,"END")) gslist[i]=new Onode(name);
                    else {
                            if (strstr(name,"Z")) type=MACROOP;
                            else if (name[0]=='Y') type=MICROOP;
                                else if ( (name[0]=='F') && strstr(name,"=")) type=ASSIGN;
                            name=strlwr(name); gslist[i]= new Onode(name,type);
                      }
                }
                else {
                        number=strtok(NULL," "); next[i][0]= atoi(number);
                        number=strtok(NULL," "); next[i][1]= atoi(number);
                        if (next[i][0]) next[i][0]--; if (next[i][1]) next[i][1]--;
                        if (name[0]=='X') type=CONDITION; else if (name[0]=='F') type=FUNCTION;
                        name=strlwr(name); gslist[i]= new Cnode(name,type);
                }
                i++;
            }
        }
        gsnnodes= i; gsmain=main;
        for (i=0; i<gsnnodes; i++)
            switch (opercond[i]) {
                case 'O': gslist[i]->Setnext(gslist[next[i][0]]); break;
                case 'C': gslist[i]->Setnext(gslist[next[i][0]],gslist[next[i][1]]); break;
            }
        delete [] string, name, number;
    }
    gsfile.close();
}

Graphscheme::~Graphscheme(void)
{ delete [] gsname; for (int i=0; i<gsnnodes; i++) delete gslist[i]; }

void Graphscheme::Cleanmarkgs(void)
{ for (int i=0; i<gsnnodes; i++) gslist[i]->Cleanmarknode(); }

void Graphscheme::Markgs(void)
{ if (gsnnodes) { Cleanmarkgs(); Node* begin=Begings(); begin->Marknode(); } }

int Graphscheme::Primarycheckgs(Hgraphscheme* hgs)
{
    int check=1, nbegin=0, nend=0, nassign=0, nmic=0, ncond=0, nmacrologic =0; Graphscheme* pt;
    for (int i=0; i<gsnnodes; i++)
        switch (gslist[i]->Nodetype()) {
            case BEGIN          : nbegin++; break;
            case END            : nend++; break;
            case ASSIGN         : nassign++; break;
            case MICROOP        : nmic++; break;
            case MACROOP        : nmic++; nmacrologic++;
                                  pt=hgs->Searchgraphhgs(gslist[i]->Macroname());
                                  if (pt==NULL) {
                                        cout <<"\nGraph-Scheme with Macro Operation "
                                        <<gslist[i]->Macroname() <<" unavailable"; check=0;
                                  } break;
            case FUNCTION       : ncond++; nmacrologic++;
                                  pt=hgs->Searchgraphhgs(gslist[i]->Nodename());
                                  if (pt==NULL) {
                                        cout <<"\nGraph-Scheme with Logic Function "
                                        <<gslist[i]->Nodename() <<" unavailable"; check=0;
                                  } break;
            case CONDITION      : ncond++; break;
        }
    if (!nbegin) { cout <<"\nGraph-Scheme without BEGIN node"; check=0;}
    else  if (nbegin>1) { cout <<"\nGraph-Scheme with more than one BEGIN node"; check=0; }
    if (!nend) { cout <<"\nGraph-Scheme without END node"; check=0;}
    else  if (nend>1) { cout <<"\nGraph-Scheme with more than one END node"; check=0; }
    if (Graphtype()==MACROGS)
        if (nassign) { cout <<"\nMacro operation with assignment nodes"; check=0; }
        else if (!(nmic+ncond)) cout <<"\nWARNING Virtual Macro operation           ";
             else if (gsmain && !nmacrologic) {
                    cout <<"\nMain Graph-scheme without macrooperations and logic functions";
                    check=0;
                  } else;
    else if (Graphtype()==LFUNCGS)
            if (!nassign) { cout <<"\nLogic Function without assignment nodes"; check=0; }
            else if (nmic) { cout <<"\nLogic Function with operational nodes"; check=0; }
    return check;
}
```

```
Node* Graphscheme::Begings(void) const
{
    int begin=-1;
    for (int i=0; i<gsnnodes; i++) if (gslist[i]->Nodetype()==BEGIN) begin=i;
    if (begin!=-1) return gslist[begin]; else return NULL;
}

Node* Graphscheme::Endgs(void) const
{
    int end=-1;
    for (int i=0; i<gsnnodes; i++) if (gslist[i]->Nodetype()==END) end=i;
    if (end!=-1) return gslist[end]; else return NULL;
}

int Graphscheme::Loopcheckgs(void)
{
    int noloop, loop=1;
    for (int i=0; i<gsnnodes; i++) {
        Cleanmarkgs(); noloop=0; gslist[i]->Loopnode(noloop);
        if (!noloop) cout <<"\nNode " <<gslist[i]->Nodename() <<" in loop";
        loop*=noloop;
    }
    return loop;
}

int Graphscheme::Checkgs(Hgraphscheme* hgs)
{
    int check=1;
    if (gsnnodes) {
        cout <<"\n****** Checking Graph-Scheme " <<gsname <<" *******\n\n";
        cout <<"Step 1 - Graph-scheme overall consistency "; check=Primarycheckgs(hgs);
        if (check) {
            cout <<"-> OK\nStep 2 - Unreacheable and dummy nodes ";
            Markgs(); for (int i=0; i<gsnnodes; i++) gslist[i]->Checknode(check);
        }
        if (check) { cout <<"-----> OK\nStep 3 - Nodes in infinite cycles "; check*=Loopcheckgs(); }
        if (check) cout <<"---------> OK\n\n**************** " <<gsname <<" OK ****************";
        else cout <<"\n\n**************** " <<gsname <<" KO ****************";
    }
    else { cout <<"\n***** Checking NULL Graph-Scheme *****"; check=0; }
    cout <<" Hit any key to continue "; getch();
    return check;
}

void Graphscheme::Printgs(char* fname)
{
    fstream pf;
    if (gsnnodes) {
        Cleanmarkgs(); pf.open(fname,ios::app); pf.write("\n********* Printing Graph-Scheme ",33);
        pf.write(gsname,strlen(gsname)); pf.write(" *********\n",11); pf.close();
        Node* begin=Begings(); begin->Printnode(fname);
        Node* end=Endgs(); end->Node::Printnode(fname);
        pf.open(fname,ios::app); pf.write("**************** ",17);
        pf.write(gsname,strlen(gsname)); pf.write(" Printed ****************\n",26); pf.close();
    }
    else cout <<"\n*** Printing NULL Graph-Scheme ***\n";
}
```

```
void Graphscheme::Listgs(void)
{
    if (gsnnodes) {
        Cleanmarkgs(); cout <<"\n   ********* Listing Graph-Scheme " <<gsname <<" *********\n";
        Node* begin=Begings(); begin->Listnode(); Node* end=Endgs(); end->Node::Listnode();
        cout <<"   **************** " <<gsname <<" Listed ****************";
    }
    else cout <<"\n   *** Listing NULL Graph-Scheme ***";
    cout <<" Hit any key to continue "; getch();
}

int Graphscheme::Rungs(Hgraphscheme* hgs, int & deep)
{
    int run=0;
    if (gsnnodes) {
        Node* begin=Begings();
        cout <<"\n********* Executing Graph-Scheme " <<gsname <<" *********\n";
        begin->Runnode(run,hgs,deep);
        cout <<"**************** " <<gsname <<" Executed ****************\n\n";
    }
    else cout <<"\n*** Running NULL Graph-Scheme ***\n\n";
    return run;
}

void Graphscheme::Cleanstategs(void)
{ for (int i=0; i<gsnnodes; i++) gslist[i]->Cleanstatenode(); }

void Graphscheme::Moorestategs(int& state, int mergesplit)
{
    if (gsnnodes) {
        if (!mergesplit) state=2;
        Cleanstategs(); for (int i=0; i<gsnnodes; i++) gslist[i]->Moorestatenode(state,this,mergesplit);
        for (int j=0; j<gsnnodes; j++)
            if (gslist[j]->Nodetype()==FUNCTION)
                if (gslist[j]->Nodestate()==-3) gslist[j]->Setstatenode(state); else gslist[j]->Cleanstatenode();
        if (!mergesplit) gsnstates=state;
    }
    else cout <<"\n*** Moore state marking NULL Graph-Scheme ***\n\n";
}

void Graphscheme::Mealystategs(int& state, int mergesplit)
{
    if (gsnnodes) {
        if (!mergesplit) state=2;
        Cleanstategs(); for (int i=0; i<gsnnodes; i++) gslist[i]->Mealystatenode(state,this,mergesplit);
        if (!mergesplit) gsnstates=state;
    }
    else cout <<"\n*** Mealy state marking NULL Graph-Scheme ***\n\n";
}

Node* Graphscheme::Nodeinitialstategs(void)
{
    Node* initialnode;
    if (gsnnodes) {
        initialnode=Begings();
        if (!initialnode->Startingstatenode()) initialnode=initialnode->Nextnode();
    }
    return initialnode;
}
```

```
void Graphscheme::Mooretablemergegs(char* fname)
{
    char* transition=new char[20]; strcpy(transition,"");
    char* logicf=new char[21]; strcpy(logicf,"");
    fstream pf; int extray=0, nt;
    if (gsnnodes) {
        Cleanmarkgs();
        for (int i=0; i<gsnnodes; i++)
            if (gslist[i]->Startingstatenode()) {
                gslist[i]->Setmarknode(); nt=1;
                gslist[i]->Mooretablenode(fname,transition,extray,nt,logicf,1,gslist[i]);
            }
    }
    else cout <<"\n*** Processing NULL Graph-Scheme ***\n";
    delete [] transition, logicf;
}

void Graphscheme::Mooretablesplitgs(char* fname)
{
    char* transition=new char[20]; strcpy(transition,""); char* logicf=new char[21]; strcpy(logicf,"");
    char* graphname=new char[5]; fstream pf; int extray=0, i, n, nspace, nt;
    if (gsnnodes) {
        Cleanmarkgs(); pf.open(fname,ios::app);
        pf.write("\n----------------------------------------------------------",59);
        strcpy(graphname,gsname); graphname=strlwr(graphname); n=strlen(graphname);
        pf.write("\n|",2); nspace=55-n; nspace/=2;
        for (i=1; i<=nspace; i++) pf.write(" ",1); pf.write(graphname,n); nspace=55-nspace-n;
        for (i=1; i<=nspace; i++) pf.write(" ",1); pf.write("|",1);
        pf.write("\n----------------------------------------------------------",59);
        pf.write("\n|a0  , --               |a2  | 1                 | --    |",59);
        pf.write("\n----------------------------------------------------------",59);
        pf.write("\n|a1  , --               |a2  | 1                 | --    |",59); pf.close();
        for (int i=0; i<gsnnodes; i++)
            if (gslist[i]->Startingstatenode()) {
                gslist[i]->Setmarknode(); nt=1;
                gslist[i]->Mooretablenode(fname,transition,extray,nt,logicf,0,gslist[i]);
            }
    }
    else cout <<"\n*** Processing NULL Graph-Scheme ***\n";
    delete [] transition, logicf, graphname;
}

void Graphscheme::Mealytablemergegs(char* fname)
{
    char *transition=new char[20]; strcpy(transition,""); char *output=new char[15]; strcpy(output,"");
    fstream pf; int nt, micro;
    if (gsnnodes) {
        Cleanmarkgs();
        for (int i=0; i<gsnnodes; i++)
            if (gslist[i]->Startingstatenode()) {
                gslist[i]->Setmarknode(); nt=1; micro=0;
                gslist[i]->Mealytablenode(fname,transition,output,nt,micro,1,gslist[i]);
            }
    }
    else cout <<"\n*** Processing NULL Graph-Scheme ***\n";
    delete [] transition, output;
}
```

```
void Graphscheme::Mealytablesplitgs(char* fname)
{
    char *transition=new char[20]; strcpy(transition,""); char *output=new char[15]; strcpy(output,"");
    char* graphname=new char[5]; fstream pf; int i, n, nspace, nt, micro;
    if (gsnnodes) {
        Cleanmarkgs(); pf.open(fname,ios::app);
        pf.write("\n------------------------------------------------",49);
        strcpy(graphname,gsname); graphname=strlwr(graphname); n=strlen(graphname);
        pf.write("\n|",2); nspace=45-n; nspace/=2;
        for (i=1; i<=nspace; i++) pf.write(" ",1); pf.write(graphname,n); nspace=45-nspace-n;
        for (i=1; i<=nspace; i++) pf.write(" ",1); pf.write("|",1);
        pf.write("\n------------------------------------------------",49);
        pf.write("\n|a0  |a2  | 1                 | --            |",49);
        pf.write("\n------------------------------------------------",49);
        pf.write("\n|a1  |a2  | 1                 | --            |",49); pf.close();
        for (i=0; i<gsnnodes; i++)
            if (gslist[i]->Startingstatenode()) {
                gslist[i]->Setmarknode(); nt=1; micro=0;
                gslist[i]->Mealytablenode(fname,transition,output,nt,micro,0,gslist[i]);
            }
    }
    else cout <<"\n*** Processing NULL Graph-Scheme ***\n";
    delete [] transition, output, graphname;
}

void Graphscheme::CCsplitgs(char* fname, char* bcode)
{
    fstream pf; int nodeline=0,stz,nsp; char *nodename=new char [15];
    if (gsnnodes) {
        pf.open(fname,ios::app); pf.write("\n| ",3); strcpy(nodename,gsname);
        nodename=strlwr(nodename); stz=strlen(nodename);
        pf.write(nodename,stz); stz=7-stz; for (nsp=0; nsp<stz; nsp++) pf.write(" ",1); pf.write("| ",2);
        stz=strlen(bcode); pf.write(bcode,stz); stz=5-stz;
        for (nsp=0; nsp<stz; nsp++) pf.write(" ",1); pf.write("| ",2);
        if (!gsmain) { pf.write("a1    | --   |      y- |",24); nodeline++; }
        for (int i=0; i<gsnnodes; i++)
            if (gslist[i]->Nodetype()==MACROOP || gslist[i]->Nodetype()==FUNCTION &&
                gslist[i]->Nodestate()!=-1 || gslist[i]->Nodetype()==MICROOP &&
                 gslist[i]->Nextnode()->Nodetype()==FUNCTION
                 && gslist[i]->Nextnode()->Nodestate()==-1) {
                if (nodeline) pf.write("\n|        |      | ",19);
                pf.write("a",1); pf.write(gslist[i]->Statechar(),3); pf.write("  | ",4);
                if (gslist[i]->Nodetype()==MICROOP )
                    strcpy(nodename,gslist[i]->Nextnode()->Nodename());
                else strcpy(nodename,gslist[i]->Macroname());
                stz=strlen(nodename); pf.write(nodename,stz); stz=5-stz;
                for (nsp=0; nsp<stz; nsp++) pf.write(" ",1); pf.write("| y+      |",11); nodeline++;
            }
        strcpy(nodename,gsname); nodename=strlwr(nodename);
        pf.write("\n|        |      | other | ",27); stz=strlen(nodename);
        pf.write(nodename,stz); stz=5-stz; for (nsp=0; nsp<stz; nsp++) pf.write(" ",1);
        pf.write("|         |",11); pf.write("\n------------------------------------------",43); pf.close();
    }
    else cout <<"\n*** Processing NULL Graph-Scheme ***\n";
    delete [] nodename;
}
```

```
void Graphscheme::Insertstategs(int& state, int mergesplit, int& insert)
{
    char* nodename=new char[15]; int answer=1;
    if (!mergesplit) state=gsnstates;
    while (answer) {
        strcpy(nodename,"");
        cout <<"Insert state in the graph-scheme " <<gsname <<" Node Name="; cin >>nodename;
        for (int i=0; i<gsnnodes; i++)
            if (!strcmp(gslist[i]->Nodename(),nodename))
                if (gslist[i]->Nodetype()==CONDITION || gslist[i]->Nodetype()==FUNCTION)
                    if (gslist[i]->Nodestate()==-1 ) {
                        cout <<"Insert state in the node " <<gslist[i]->Nodename()
                        <<" that is pointing to the node(s) " <<gslist[i]->Nextnode()->Nodename()
                        <<" & " <<gslist[i]->Nextnode(0)->Nodename() <<"\n";
                        do {
                            cout <<"Yes[1]/No[0] ? "; cin >> answer;
                        } while (answer != 0 && answer != 1);
                        if (answer) { gslist[i]->Setstatenode(state,0); insert++; }
                    }
                    else cout <<"Node " <<gslist[i]->Nodename() <<" pointing to the node(s) "
                                 <<gslist[i]->Nextnode()->Nodename() <<" & "
                                 <<gslist[i]->Nextnode(0)->Nodename() <<" already marked\n";
                else cout <<"This is not a condititional node\n";
        do {
            cout <<"Insert another state in the graph-scheme " <<gsname <<"  Yes[1]/No[0] ? ";
            cin >>answer;
        } while (answer != 0 && answer != 1);
    }
    if (!mergesplit) gsnstates=state; delete [] nodename;
}

void Graphscheme::Removestategs(int& state, int mergesplit, int& remove)
{
    char* nodename=new char[15]; int answer=1;
    if (!mergesplit) state=gsnstates;
    while (answer) {
        strcpy(nodename,"");
        cout <<"Remove state in the graph-scheme " <<gsname <<" Node Name="; cin >>nodename;
        for (int i=0; i<gsnnodes; i++)
            if (!strcmp(gslist[i]->Nodename(),nodename))
                if (!gslist[i]->Nodeauto() && gslist[i]->Nodestate()!=-1) {
                    cout <<"Remove state in the node " <<gslist[i]->Nodename()
                         <<" that is pointing to the node(s) " <<gslist[i]->Nextnode()->Nodename()
                         <<" & " <<gslist[i]->Nextnode(0)->Nodename() <<"\n";
                    do {
                        cout <<"Yes[1]/No[0] ? "; cin >> answer;
                    } while (answer != 0 && answer != 1);
                    if (answer) {
                        if (gslist[i]->Nodestate()==state-1) state--;
                        gslist[i]->Cleanstatenode(); remove++;
                    }
                }
                else cout <<"This node was not manually marked or it is not marked yet\n";
        do {
            cout <<"Remove another state in the graph-scheme " <<gsname <<"  Yes[1]/No[0] ? ";
            cin >>answer;
        } while (answer != 0 && answer != 1);
    }
    if (!mergesplit) gsnstates=state; delete [] nodename;
}
```

```
//******************** Implementation of the Class Hierarchical Graph-Scheme ********************

Hgraphscheme::Hgraphscheme(char* fname)
{
    hgsname=new char[20]; hgssyn=new char[6]; ifstream hgsfile(fname);
    hgscheck=0; hgsdeep=0; hgsmark=-1; hgsnstates=0; hgsngs=0;
    if (!hgsfile) cout <<"\n*** Cannot open Hierarchical Graph-Scheme " <<fname <<" ***\n";
    else {
        int i=0; char extra, *gsfile=new char[20];
        hgsfile.get(gsfile,20); hgsfile.get(extra); strcpy(hgsname,gsfile);
        while (!hgsfile.eof()) {
            hgsfile.get(gsfile,20); hgsfile.get(extra);
            if (strcmp(gsfile,"")) {
                strcat(gsfile,".txt");
                if (!i) hgslist[i]=new Graphscheme(gsfile,1);
                else hgslist[i]=new Graphscheme(gsfile);
                i++;
            }
        }
        hgsngs=i;
        delete [] gsfile;
    }
    hgsfile.close();
}
Hgraphscheme::~Hgraphscheme(void)
{ delete [] hgsname, hgssyn; for (int i=0; i<hgsngs; i++) delete hgslist[i]; }

void Hgraphscheme::Listhgs(void)
{
    if (hgscheck) {
        cout <<"\n****** Listing Hierarchical Graph-Scheme " <<hgsname <<" ******\n\n";
        for (int i=0; i<hgsngs; i++) hgslist[i]->Listgs();
        cout <<"\n******************* " <<hgsname <<"  Listed *******************\n\n";
    }
    else cout <<"*** Cannot list an unchecked Hierarchical Graph-Scheme ***\n\n";
}

void Hgraphscheme::Printhgs(void)
{
    fstream printfile; char* filename=new char[20];
    if (hgscheck) {
        cout <<"\n****** Printing Hierarchical Graph-Scheme ******\n";
        cout <<"\nFilename -> "; cin >> filename; strcat(filename,".txt");
        printfile.open(filename,ios::out|ios::noreplace);
        if (!printfile) cout <<"\n*** File " <<filename <<" already exists ***\n";
        else {
            cout <<"\n****** Printing Hierarchical Graph-Scheme " <<hgsname <<" ******\n";
            printfile.write("****** Printing Hierarchical Graph-Scheme ",42);
            printfile.write(hgsname,strlen(hgsname)); printfile.write(" ******\n",8); printfile.close();
            for (int i=0; i<hgsngs; i++) hgslist[i]->Printgs(filename);
            printfile.open(filename,ios::app); printfile.write("\n******************* ",21);
            printfile.write(hgsname,strlen(hgsname));
            printfile.write(" Printed ********************\n",30); printfile.close();
            cout <<"\n******************* " <<hgsname <<"  Printed *******************\n\n";
        }
    }
    else cout <<"*** Cannot print an unchecked Hierarchical Graph-Scheme ***\n\n";
    delete [] filename;
}
```

```
int Hgraphscheme::Checkhgs(void)
{
    int check=1;
    if (!hgscheck) {
        cout <<"\n****** Checking Hierarchical Graph-Scheme " <<hgsname <<" ******\n";
        if (hgsngs<2) { cout <<"\nThis is not an Hierarchical Graph-Scheme\n"; check=0; }
        if (check) for (int i=0; i<hgsngs; i++) check*=hgslist[i]->Checkgs(this);
        if (check)
             cout <<"\n****************** " <<hgsname <<"  Checked OK *****************\n\n";
        else cout <<"\n****************** " <<hgsname <<"  Checked KO *****************\n\n";
    }
    else cout <<"*** Hierarchical Graph-Scheme already checked ***\n\n";
    hgscheck=check; return check;
}

Graphscheme* Hgraphscheme::Searchgraphhgs(char* name)
{
    char* searchname=new char[15]; strcpy(searchname,name);
    searchname=strupr(searchname); int search=-1;
    for (int i=0; i<hgsngs; i++) if (!strcmp(hgslist[i]->Graphname(),searchname)) search=i;
    delete [] searchname; if (search!=-1) return hgslist[search]; else return NULL;
}

void Hgraphscheme::Runhgs(void)
{
    int deep=0;
    if (hgscheck) {
        cout <<"\n****** Executing Hierarchical Graph-Scheme " <<hgsname <<" ******\n";
        hgsdeep=0; hgslist[0]->Rungs(this,deep);
        cout <<"********* " <<hgsname <<" Executed [Deepest Level=" <<hgsdeep
             <<"] *********\n\n";
    }
    else cout <<"*** Cannot run an unchecked Hierarchical Graph-Scheme ***\n\n";
}

void Hgraphscheme::Moorestatehgs(void)
{
    int state=2, mooremealy=0, logic=0, i;
    if (hgscheck) {
        for (i=0; i<hgsngs; i++) if (hgslist[i]->Graphtype()==LFUNCGS) logic++;
        if (logic) {
            cout <<"\n Marking " <<logic <<" Logic Function(s) as Moore [1] or Mealy [0]";
            do { cout <<"\n Your Choice -> "; cin >> mooremealy;
            } while (mooremealy != 0 && mooremealy != 1);
        }
        for (i=0; i<hgsngs; i++)
            if (hgslist[i]->Graphtype()==MACROGS) hgslist[i]->Moorestategs(state,hgsmark);
            else if (mooremealy) hgslist[i]->Moorestategs(state,hgsmark);
                 else hgslist[i]->Mealystategs(state,hgsmark);
        if (hgsmark) {
            hgsnstates=state;
            cout <<"\n *** HGS " <<hgsname <<" is marked with " <<state <<" states ***\n";
        }
        else {
            state=0;
            for (i=0; i<hgsngs; i++) if (hgslist[i]->Graphnstates()>state) state=hgslist[i]->Graphnstates();
            cout <<"\n *** The bigger GS of " <<hgsname <<" is marked with " <<state
                  <<" states ***\n";
        }
    }
    else cout <<"*** Cannot mark an unchecked Hierarchical Graph-Scheme ***\n\n";
}
```

```
void Hgraphscheme::Mealystatehgs(void)
{
    int state=2, i;
    if (hgscheck) {
        for (i=0; i<hgsngs; i++) hgslist[i]->Mealystategs(state,hgsmark);
        if (hgsmark) {
            hgsnstates=state;
            cout <<"\n *** HGS " <<hgsname <<" is marked with " <<state <<" states ***\n";
        }
        else {
            state=0;
            for (i=0; i<hgsngs; i++) if (hgslist[i]->Graphnstates()>state) state=hgslist[i]->Graphnstates();
            cout <<"\n *** The bigger GS of " <<hgsname <<" is marked with " <<state
                 <<" states ***\n";
        }
    }
    else cout <<"*** Cannot mark an unchecked Hierarchical Graph-Scheme ***\n\n";
}

void Hgraphscheme::CCmergehgs(char* fname)
{
    fstream pf; int i, G, N, stz, nsp; char* graphname=new char[5];
    if (hgscheck) {
        G=ceil(log(hgsngs+1)/M_LN2); N=pow(2,G); pf.open(fname,ios::app);
        pf.write("\n--------------------------------------------------------",58);
        pf.write("\n|              Code Converter Programming              |",58);
        pf.write("\n--------------------------------------------------------",58);
        pf.write("\n|  | Zi/Fi | Zi/Fi |   Code Converter   |     Zi/Fi     |",58);
        pf.write("\n|  | Name  | Code  |     inputs YZi     | Initial state |",58);
        pf.write("\n--------------------------------------------------------",58);
        pf.write("\n|0 |       | ",14); pf.write(binarycode(0,G),G);
        for (nsp=0; nsp<6-G; nsp++) pf.write(" ",1);
        pf.write("|                    | a0            |",38);
        pf.write("\n--------------------------------------------------------",58);
        for (i=0; i<hgsngs; i++) {
            pf.write("\n|",2); pf.write(linechar(i+1),2); pf.write("| ",2);
            strcpy(graphname,Ngraphname(i)); graphname=strlwr(graphname);
            stz=strlen(graphname); pf.write(graphname,stz);
            for (nsp=0; nsp<6-stz; nsp++) pf.write(" ",1); pf.write("| ",2);
            pf.write(binarycode(i+1,G),G);
            for (nsp=0; nsp<6-G; nsp++) pf.write(" ",1); pf.write("| ",2);
            stz=strlen(yzstring(i+1,G)); pf.write(yzstring(i+1,G),stz);
            for (nsp=0; nsp<19-stz; nsp++) pf.write(" ",1); pf.write("| a",3);
            Node* initialnode=hgslist[i]->Nodeinitialstategs();
            pf.write(initialnode->Statechar(),3);
            for (nsp=0; nsp<10; nsp++) pf.write(" ",1); pf.write("|",1);
            pf.write("\n--------------------------------------------------------",58);
        }
        for (i=hgsngs; i<N-1; i++) {
            pf.write("\n|",2); pf.write(linechar(i+1),2); pf.write("|       | ",10);
            pf.write(binarycode(i+1,G),G); for (nsp=0; nsp<6-G; nsp++) pf.write(" ",1);
            pf.write("|                    | a0            |",38);
            pf.write("\n--------------------------------------------------------",58);
        }
        pf.write("\n\n",2); pf.close();
    }
    else cout <<"*** Cannot process an unchecked Hierarchical Graph-Scheme ***\n\n";
    delete [] graphname;
}
```

```
void Hgraphscheme::CCsplithgs(char* fname)
{
    fstream pf; int G;
    if (hgscheck) {
        G=ceil(log(hgsngs)/M_LN2); pf.open(fname,ios::app);
        pf.write("\n------------------------------------------",43);
        pf.write("\n|       Code Converter Programming       |",43);
        pf.write("\n------------------------------------------",43);
        pf.write("\n| Active | HGS  | State | Next |  Stack  |",43);
        pf.write("\n|   HGS  | Code |       | HGS  | Pointer |",43);
        pf.write("\n------------------------------------------",43); pf.close();
        for (int i=0; i<hgsngs; i++) hgslist[i]->CCsplitgs(fname,binarycode(i,G));
        pf.open(fname,ios::app); pf.write("\n\n",2); pf.close();
    }
    else cout <<"*** Cannot process an unchecked Hierarchical Graph-Scheme ***\n\n";
}

void Hgraphscheme::Mooretablemergehgs(char* fname)
{
    fstream pf;
    if (hgscheck) {
        pf.open(fname,ios::app);
        pf.write("\n----------------------------------------------------------",59);
        pf.write("\n|           HGS Moore State Transition Table             |",59);
        pf.write("\n----------------------------------------------------------",59);
        pf.write("\n|      am , Y(am)       | as |     X(am,as)     |Y(am,as)|",59);
        pf.write("\n----------------------------------------------------------",59);
        pf.write("\n|a0  , ",8); char* maingsname=new char[5];
        strcpy(maingsname,hgslist[0]->Graphname()); maingsname=strlwr(maingsname);
        int n=strlen(maingsname); pf.write(maingsname,n); delete [] maingsname;
        for (int i=n+1; i<17; i++) pf.write(" ",1);
        pf.write("|a0  | 1                | --     |",35);
        pf.write("\n----------------------------------------------------------",59);
        pf.write("\n|a1  , y-              |a0  | 1                | --     |",59);
        pf.close();
        for (int j=0; j<hgsngs;j++) hgslist[j]->Mooretablemergegs(fname);
        pf.open(fname,ios::app);
        pf.write("\n----------------------------------------------------------",59);
        pf.close();
    }
    else cout <<"*** Cannot process an unchecked Hierarchical Graph-Scheme ***\n\n";
}

void Hgraphscheme::Mooretablesplithgs(char* fname)
{
    fstream pf;
    if (hgscheck) {
        pf.open(fname,ios::app);
        pf.write("\n----------------------------------------------------------",59);
        pf.write("\n|           HGS Moore State Transition Table             |",59);
        pf.write("\n----------------------------------------------------------",59);
        pf.write("\n|      am , Y(am)       | as |     X(am,as)     |Y(am,as)|",59);
        pf.close();
        for (int j=0; j<hgsngs;j++) hgslist[j]->Mooretablesplitgs(fname);
        pf.open(fname,ios::app);
        pf.write("\n----------------------------------------------------------",59);
        pf.close();
    }
    else cout <<"*** Cannot process an unchecked Hierarchical Graph-Scheme ***\n\n";
}
```

```
void Hgraphscheme::Mealytablemergehgs(char* fname)
{
    fstream pf;
    if (hgscheck) {
        pf.open(fname,ios::app); pf.write("\n------------------------------------------------",49);
        pf.write("\n|      HGS Mealy State Transition Table       |",49);
        pf.write("\n------------------------------------------------",49);
        pf.write("\n| am | as |     X(am,as)     |    Y(am,as)    |",49);
        pf.write("\n------------------------------------------------",49); pf.write("\n|a0  |a0  | 1                | ",32);
        char* maingsname=new char[5]; strcpy(maingsname,hgslist[0]->Graphname());
        maingsname=strlwr(maingsname); int n=strlen(maingsname); pf.write(maingsname,n);
        delete [] maingsname; for (int i=n+1; i<16; i++) pf.write(" ",1); pf.write("|",1);
        pf.write("\n------------------------------------------------",49);
        pf.write("\n|a1  |a0  | 1                | y-             |",49); pf.close();
        for (int j=0; j<hgsngs; j++) hgslist[j]->Mealytablemergegs(fname);
        pf.open(fname,ios::app); pf.write("\n------------------------------------------------",49); pf.close();
    }
    else cout <<"*** Cannot process an unchecked Hierarchical Graph-Scheme ***\n\n";
}

void Hgraphscheme::Mealytablesplithgs(char* fname)
{
    fstream pf;
    if (hgscheck) {
        pf.open(fname,ios::app); pf.write("\n------------------------------------------------",49);
        pf.write("\n|      HGS Mealy State Transition Table       |",49);
        pf.write("\n------------------------------------------------",49);
        pf.write("\n| am | as |     X(am,as)     |    Y(am,as)    |",49); pf.close();
        for (int j=0; j<hgsngs; j++) hgslist[j]->Mealytablesplitgs(fname);
        pf.open(fname,ios::app); pf.write("\n------------------------------------------------",49); pf.close();
    }
    else cout <<"*** Cannot process an unchecked Hierarchical Graph-Scheme ***\n\n";
}

void Hgraphscheme::Mooresynmergehgs(int statemark)
{
    fstream pf;
    if (hgscheck) {
        cout <<"\n**** Moore (model 2) Synthesis of Hierarchical Graph-Scheme "
             <<hgsname <<" ****\n";
        char* filename=new char[20]; cout <<"\n Filename -> "; cin >> filename; strcat(filename,".txt");
        pf.open(filename,ios::out|ios::noreplace);
        if (!pf) cout <<"\n*** File " <<filename <<" already exists ***\n";
        else {
            pf.write("Moore Synthesis of Hierarchical Graph-Scheme ",45);
            pf.write(hgsname,strlen(hgsname)); pf.write(" [MERGE TABLES]",15);
            pf.write("\n\n",2); pf.close();
            if (statemark) {
                strcpy(hgssyn,"Moore"); hgsmark=1;
                cout <<"\n Step 1 - Marking Hierarchical Graph-Scheme\n"; Moorestatehgs();
            }
            else cout <<"\n Step 1 - Skiping Hierarchical Graph-Scheme Marking\n";
            cout <<"\n Step 2-  Code Converter Programming\n"; CCmergehgs(filename);
            cout <<"\n Step 3 - Generating HGS Moore State Transition Table\n";
            Mooretablemergehgs(filename);
            cout <<"\n**************     " <<hgsname
                 <<" Moore Synthesis Finished     **************\n";
        }
        delete [] filename;
    }
    else cout <<"*** Cannot process an unchecked Hierarchical Graph-Scheme ***\n\n";
}
```

```
void Hgraphscheme::Mooresynsplithgs(int statemark)
{
    fstream pf;
    if (hgscheck) {
        cout <<"\n**** Moore (model 3) Synthesis of Hierarchical Graph-Scheme "
             <<hgsname <<" ****\n";
        char* filename=new char[20]; cout <<"\n Filename -> ";
        cin >> filename; strcat(filename,".txt");
        pf.open(filename,ios::out|ios::noreplace);
        if (!pf) cout <<"\n*** File " <<filename <<" already exists ***\n";
        else {
            pf.write("Moore Synthesis of Hierarchical Graph-Scheme ",45);
            pf.write(hgsname,strlen(hgsname)); pf.write(" [SPLIT TABLES]",15);
            pf.write("\n\n",2); pf.close();
            if (statemark) {
                strcpy(hgssyn,"Moore"); hgsmark=0;
                cout <<"\n Step 1 - Marking Hierarchical Graph-Scheme\n"; Moorestatehgs();
            }
            else cout <<"\n Step 1 - Skiping Hierarchical Graph-Scheme Marking\n";
            cout <<"\n Step 2-  Code Converter Programming\n"; CCsplithgs(filename);
            cout <<"\n Step 3 - Generating HGS Moore State Transition Table\n";
            Mooretablesplithgs(filename);
            cout <<"\n***************     " <<hgsname
                 <<" Moore Synthesis Finished     ***************\n";
        }
        delete [] filename;
    }
    else cout <<"*** Cannot process an unchecked Hierarchical Graph-Scheme ***\n\n";
}

void Hgraphscheme::Mealysynmergehgs(int statemark)
{
    fstream pf;
    if (hgscheck) {
        cout <<"\n**** Mealy (model 2) Synthesis of Hierarchical Graph-Scheme "
             <<hgsname <<" ****\n";
        char* filename=new char[20];
        cout <<"\n Filename -> "; cin >> filename; strcat(filename,".txt");
        pf.open(filename,ios::out|ios::noreplace);
        if (!pf) cout <<"\n*** File " <<filename <<" already exists ***\n";
        else {
            pf.write("Mealy Synthesis of Hierarchical Graph-Scheme ",45);
            pf.write(hgsname,strlen(hgsname)); pf.write(" [MERGE TABLES]",15);
            pf.write("\n\n",2); pf.close();
            if (statemark) {
                strcpy(hgssyn,"Mealy"); hgsmark=1;
                cout <<"\n Step 1 - Marking Hierarchical Graph-Scheme\n"; Mealystatehgs();
            }
            else cout <<"\n Step 1 - Skiping Hierarchical Graph-Scheme Marking\n";
            cout <<"\n Step 2 - Code Converter Programming\n"; CCmergehgs(filename);
            cout <<"\n Step 3 - Generating HGS Mealy State Transition Table\n";
            Mealytablemergehgs(filename);
            cout <<"\n***************     " <<hgsname
                 <<" Mealy Synthesis Finished     ***************\n";
        }
        delete [] filename;
    }
    else cout <<"*** Cannot process an unchecked Hierarchical Graph-Scheme ***\n\n";
}
```

```
void Hgraphscheme::Mealysynsplithgs(int statemark)
{
    fstream pf;
    if (hgscheck) {
        cout <<"\n**** Mealy (model 3) Synthesis of Hierarchical Graph-Scheme "
             <<hgsname <<" ****\n";
        char* filename=new char[20]; cout <<"\n Filename -> "; cin >> filename; strcat(filename,".txt");
        pf.open(filename,ios::out|ios::noreplace);
        if (!pf) cout <<"\n*** File " <<filename <<" already exists ***\n";
        else {
            pf.write("Mealy Synthesis of Hierarchical Graph-Scheme ",45);
            pf.write(hgsname,strlen(hgsname)); pf.write(" [SPLIT TABLES]",15);
            pf.write("\n\n",2); pf.close();
            if (statemark) {
                strcpy(hgssyn,"Mealy"); hgsmark=0;
                cout <<"\n Step 1 - Marking Hierarchical Graph-Scheme\n"; Mealystatehgs();
            }
            else cout <<"\n Step 1 - Skiping Hierarchical Graph-Scheme Marking\n";
            cout <<"\n Step 2 - Code Converter Programming\n"; CCsplithgs(filename);
            cout <<"\n Step 3 - Generating HGS Mealy State Transition Table\n";
            Mealytablesplithgs(filename);
            cout <<"\n***************     " <<hgsname
                 <<" Mealy Synthesis Finished     ***************\n";
        }
        delete [] filename;
    }
    else cout <<"*** Cannot process an unchecked Hierarchical Graph-Scheme ***\n\n";
}

void Hgraphscheme::Insertstatehgs(void)
{
    char* graphname=new char[5]; int nstate, answer, insert=0;
    if (hgscheck)
        if (hgsmark!=-1) {
            nstate=hgsnstates; strcpy(graphname,"");
            cout <<"\n******** Insert extra states in the already marked HGS "
                 <<hgsname <<" ********\n";
            do {
                cout <<"Graph-Scheme Name="; cin >>graphname;
                Graphscheme* gs=Searchgraphhgs(graphname);
                if (gs!=NULL) gs->Insertstategs(nstate,hgsmark,insert);
                else cout <<graphname <<" Graph-Scheme does not belong to HGS "
                          <<hgsname <<"\n";
                do { cout <<"Insert states in another graph-scheme Yes[1]/No[0] ? "; cin >>answer; }
                while (answer != 0 && answer != 1);
            } while (answer);
            if (insert) {
                cout <<"\n *** " <<insert <<" extra states inserted ***\n";
                if (hgsmark) {
                    hgsnstates=nstate;
                    cout <<" *** Now HGS " <<hgsname <<" is marked with "
                         <<hgsnstates <<" states ***\n";
                }
                if (!strcmp(hgssyn,"Moore")) if (hgsmark) Mooresynmergehgs(0); else Mooresynsplithgs(0);
                else if (hgsmark) Mealysynmergehgs(0); else Mealysynsplithgs(0);
            }
            else cout <<"WARNING No extra states were inserted\n";
        }
        else cout <<"WARNING " <<hgsname <<" is not marked yet\n";
    else cout <<"*** Cannot process an unchecked Hierarchical Graph-Scheme ***\n\n";
    delete [] graphname;
}
```

```
void Hgraphscheme::Removestatehgs(void)
{
    char* graphname=new char[5]; int nstate, answer, remove=0;
    if (hgscheck)
        if (hgsmark!=-1) {
            nstate=hgsnstates; strcpy(graphname,"");
            cout <<"\n******** Remove extra states in the already marked HGS "
                  <<hgsname <<" *********\n";
            do {
                cout <<"Graph-Scheme Name="; cin >>graphname;
                Graphscheme* gs=Searchgraphhgs(graphname);
                if (gs!=NULL) gs->Removestategs(nstate,hgsmark,remove);
                else cout <<graphname <<" Graph-Scheme does not belong to HGS "
                          <<hgsname <<"\n";
                do {
                    cout <<"Remove states in another graph-scheme Yes[1]/No[0] ? "; cin >>answer;
                } while (answer != 0 && answer != 1);
            } while (answer);
            if (remove) {
                cout <<"\n *** " <<remove <<" extra states removed ***\n";
                if (hgsmark) {
                    hgsnstates=nstate;
                    cout <<" *** Now HGS " <<hgsname <<" is marked with "
                          <<hgsnstates <<" states ***\n";
                }
                if (!strcmp(hgssyn,"Moore")) if (hgsmark) Mooresynmergehgs(0); else Mooresynsplithgs(0);
                else if (hgsmark) Mealysynmergehgs(0); else Mealysynsplithgs(0);
            }
            else cout <<"WARNING No extra states were removed\n";
        }
        else cout <<"WARNING " <<hgsname <<" is not marked yet\n";
    else cout <<"*** Cannot process an unchecked Hierarchical Graph-Scheme ***\n\n";
    delete [] graphname;
}

//************************************ SIMULHGS Menu ************************************

void main(void)
{
    char* filename=new char[20]; Hgraphscheme *hgs; int choice, exit=1;

    cout <<"\n Graph-Scheme Filename -> "; cin >> filename;
    strcat(filename,".txt"); hgs=new Hgraphscheme(filename);

    if (hgs->Hgsngs() != 0)
        do {
            clrscr();
            cout <<"          ----------------------------------\n";
            cout <<"          |  1 - Check Graph-Scheme        |\n";
            cout <<"          |  2 - List Graph-Scheme         |\n";
            cout <<"          |  3 - Print Graph-Scheme        |\n";
            cout <<"          |  4 - Run Graph-Scheme          |\n";
            cout <<"          |  5 - Moore Synthesis (model 2) |\n";
            cout <<"          |  6 - Mealy Synthesis (model 2) |\n";
            cout <<"          |  7 - Moore Synthesis (model 3) |\n";
            cout <<"          |  8 - Mealy Synthesis (model 3) |\n";
            cout <<"          |  9 - Insert Extra States       |\n";
            cout <<"          | 10 - Remove Extra States       |\n";
            cout <<"          | 11 - Exit Program              |\n";
            cout <<"          ----------------------------------\n";
            cout <<"\n                Your Option -> "; cin >>choice;
            clrscr();
            switch (choice) {
                case  1 : exit=hgs->Checkhgs();
                          if (exit) { cout <<"\nHit any key to continue "; getch(); }
                          else {
                                cout <<"\nCannot processe this hierarchical graph-scheme”;
                                cout <<"\nPROGRAM TERMINATED";
                          }
                          break;
                case  2 : hgs->Listhgs();
                          cout <<"\nHit any key to continue "; getch(); break;
                case  3 : hgs->Printhgs();
                          cout <<"\nHit any key to continue "; getch(); break;
                case  4 : hgs->Runhgs();
                          cout <<"\nHit any key to continue "; getch(); break;
                case  5 : hgs->Mooresynmergehgs();
                          cout <<"\nHit any key to continue "; getch(); break;
                case  6 : hgs->Mealysynmergehgs();
                          cout <<"\nHit any key to continue "; getch(); break;
                case  7 : hgs->Mooresynsplithgs();
                          cout <<"\nHit any key to continue "; getch(); break;
                case  8 : hgs->Mealysynsplithgs();
                          cout <<"\nHit any key to continue "; getch(); break;
                case  9 : hgs->Insertstatehgs();
                          cout <<"\nHit any key to continue "; getch(); break;
                case 10: hgs->Removestatehgs();
                          cout <<"\nHit any key to continue "; getch(); break;
                case 11: exit=0; break;
                default : cout <<"Wrong choice - Hit any key to continue "; getch(); break;
            }
        } while (exit);

    delete hgs; delete [] filename;
}
```

11 REFERENCES

[AleHan75] Igor Aleksander, F. Keith Hanna, “Automata Theory: An Engineering Approach”, Crane, Russak & Company, Inc., 1975.

[AshDevNew92] Pranav Ashar, Srinivas Devadas, A. Richard Newton, “Sequential Logic Synthesis”, Kluwer Academic Publishers, 1992.

[Baranov74] Samary Baranov, ”Synthesis of Microprogrammed Automata”, Energy Publishing Company, 1974 (in Russian).

[Baranov79] Samary Baranov, ”Synthesis of Microprogrammed Automata”, 2nd Edition, Energy Publishing Company, 1979 (in Russian). French Version “Synthèse des Automates Microprogrammés”, Editions Mir, 1983.

[Baranov94] Samary Baranov, ”Logic Synthesis for Control Automata”, Kluwer Academic Publishers, 1994.

[Bolton90] Martin Bolton, “Digital Systems Design with Programmable Logic”, Addison-Wesley Publishing Company, 1990.

[Booch94] Grady Booch, “Object-Oriented Analysis and Design with Applications”, Second Edition, Addison-Wesley Publishing Company, 1994.

[Booth67] Taylor L. Booth, “Sequential Machines and Automata Theory”, John Wiley & Sons, Inc., 1967.

[BroRos96] Stephen Brown, Jonathan Rose, "FPGA and CPLD Architectures: A Tutorial", IEEE Design & Test of Computers, Vol. 13, Nº 2, pp. 42-57, Summer 1996.

[CamWol91] Raul Camposano and Wayne Wolf, Editors, "High-Level VLSI Synthesis", Kluwer Academic Publishers, 1991.

[Clare73] Christopher R. Clare, "Designing Logic Systems Using State Machines", McGraw-Hill, Inc., 1973.

[DruHar89] Doron Drusinsky, David Harel, "Using Statecharts for Hardware Description and Synthesis", IEEE Transactions on Computer-Aided Design, Vol. 8, Nº 7, pp. 798-807, July 1989.

[Edwards97] Stephen Edwards, Luciano Lavagno, Edward A.Lee, Alberto Sangiovanni-Vincentelli, "Design of Embedded Systems: Formal Models, Validation, and Synthesis", Proceedings of the IEEE, Vol. 85, Nº 3, pp. 366-390, March 1997.

[GajRam94] Daniel D. Gajski, Loganath Ramachandran, "Introduction to High-Level Synthesis", IEEE Design & Test of Computers, Vol. 11, Nº 4, pp. 44-54, Winter 1994.

[Gajski92] Daniel D. Gajski, Nikil D. Dutt, Allen Wu, Steve Lin, "High-Level Synthesis Introduction to Chip and System Design", Kluwer Academic Publishers, 1992.

[Gajski94] Daniel D. Gajski, Frank Vahid, Sanjiv Narayan, Jie Gong, "Specification and Design of Embedded Systems", Prentice Hall, Inc., 1994.

[Gajski97] Daniel D. Gajski, "Principles of Digital Design", Prentice Hall, Inc., 1997.

[Green86] David Green, "Modern Logic Design", Addison-Wesley Publishing Company, 1986.

[GupLia97] Rajesh K. Gupta, Stan Y. Liao, "Using a Programming Language for Digital System Design", IEEE Design & Test of Computers, Vol. 14, Nº 2, pp. 72-80, April-June 1997.

[GupMic93] Rajesh K. Gupta, Giovanni De Micheli, "Hardware-Software Cosynthesis for Digital Systems", IEEE Design & Test of Computers, Vol. 10, Nº 3, pp. 29-41, September 1993.

[Harel87] David Harel, "Statecharts: A Visual Formalism for Complex Systems", Science of Computer Programming, Nº 8, pp. 231-274, 1987.

[Harel88] David Harel, "On Visual Formalisms", Communications of the ACM, Vol. 31, Nº 5, pp. 514-530, May 1988.

[Harel90] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, Mark Trakhtenbrot, "Statemate: A Working Environment for the Development of Complex Reactive Systems", IEEE Transactions on Software Engineering, Vol. 16, Nº 4, pp. 403-414, April 1990.

[HarSte66] J. Hartmanis, R. E. Stearns, "Algebraic Structure Theory of Sequential Machines", Prentice Hall, Inc., 1966.

[HopUll79] John E. Hopcroft, Jeffrey D. Ullman, "Introduction to Automata Theory, Languages and Computation", Addison-Wesley Publishing Company, 1979.

[IEEE94] "IEEE Standard VHDL Language Reference Manual", IEEE Std 1076-1993, Institute of Electrical and Electronics Engineers, Inc., 1994.

[IsmJer95] Tarek Ben Ismail, Ahmed Amine Jerraya, "Synthesis Steps and Design Models for Codesign", IEEE Computer, Vol. 28, Nº 2, pp. 44-52, February 1995.

[Katz94] Randy H. Katz, "Contemporary Logic Design", The Benjamin/Cummings Publishing Company, Inc., 1994.

[Kohavi70] Zvi Kohavi, "Switching and Finite Automata Theory", McGraw-Hill, Inc., 1970.

[McFKow90] Michael McFarland, Thaddeus J. Kowalski, "Incorporating Bottom-Up Design into Hardware Synthesis", IEEE Transactions on Computer-Aided Design, Vol. 9, Nº 9, pp. 938-950, September 1990.

[McFParCam90] Michael C. McFarland, Alice C. Parker, Raul Camposano, "The High-Level Synthesis of Digital Systems", Proceedings of the IEEE, Vol. 78, Nº 2, pp. 301-318, February 1990.

[MicGup97] Giovanni de Micheli, Rajesh K. Gupta, "Hardware/Software Co-Design", Proceedings of the IEEE, Vol. 85, Nº 3, pp. 349-365, March 1997.

[Micheli94] Giovanni De Micheli, "Synthesis and Optimization of Digital Circuits", McGraw-Hill, Inc., 1994.

[MicLauDuz92] Petra Michel, Ulrich Lauther and Peter Duzy, Editors, "The Synthesis Approach to Digital System Design", Kluwer Academic Publishers, 1992.

[Murata89] Tadao Murata, "Petri Nets: Properties, Analysis and Applications", Proceedings of the IEEE, Vol. 77, Nº 4, pp. 541-590, April 1989.

[Navabi93] Zainalabedin Navabi, "VHDL Analysis and Modelling of Digital Systems", McGraw-Hill, Inc., 1993.

[ParCra98] José A. L. Parente, Carlos M. M. S. Cravo, "Editor Gráfico de Grafos Hierárquicos", Final Year Project Report, Departamento de Electrónica e Telecomunicações, Universidade de Aveiro, 1998 (in Portuguese).

[RocSkl97A] António Adrego da Rocha, Valery Sklyarov, "VHDL Modeling of Hierarchical Finite State Machines", Proceedings of the Fifth BELSIGN Workshop, Dresden, April 1997.

[RocSkl97B] António Adrego da Rocha, Valery Sklyarov, "Simulação em VHDL de Máquinas de Estados Finitas Hierárquicas", Electrónica e Telecomunicações, Vol. 2, Nº 1, pp. 83-94, September 1997 (in Portuguese).

[RocSklFer97] António Adrego da Rocha, Valery Sklyarov, António Ferrari, "Hierarchical Description and Design of Control Circuits Based on Reconfigurable and Reprogrammable Elements", Proceedings of the International Workshop on Logic and Architectural Synthesis - IWLAS'97, pp. 73-82, Grenoble, December 1997.

[SklFer98] Valery Sklyarov, António Ferrari, "Synthesis of Control Devices Described by Hierarchical Graph-Schemes", Proceedings of the 3rd Australasian Computer Architecture Conference - ACAC'98, pp. 181-191, Perth, February 1998.

[SklRoc96A] Valery Sklyarov, António Adrego da Rocha, "Sintese de Unidades de Controlo Descritas por Grafos dum Esquema Hierárquicos", Electrónica e Telecomunicações, Vol. 1, Nº 6, pp. 577-588, September 1996 (in Portuguese).

[SklRoc96B] Valery Sklyarov, António Adrego da Rocha, “Synthesis of Control Units Described by Hierarchical Graph-Schemes”, Proceedings of the Fourth BELSIGN Workshop, Santander, November 1996.

[SklRocFer98] Valery Sklyarov, António Adrego da Rocha, António de Brito Ferrari, “Synthesis of Reconfigurable Control Devices Based on Object-Oriented Specifications”, in the book “Advanced Techniques for Embedded Systems Design and Test”, Kluwer Academic Publishers, pp. 151-177, 1998.

[Sklyarov84] Valery Sklyarov, “Hierarchical Graph-Schemes”, Latvian Academy of Science, Automatics and Computers, Nº 2, pp. 82-87, Riga, 1984 (in Russian).

[Sklyarov87] Valery Sklyarov, “Parallel Graph-Schemes and Finite State Machines Synthesis”, Latvian Academy of Science, Automatics and Computers, Nº 5, pp. 68-76, Riga, 1987 (in Russian).

[Sklyarov96] Valery Sklyarov, “Applying Finite State Machine Theory and Object-Oriented Programming to the Logic Synthesis of Control Devices”, Electrónica e Telecomunicações, Vol. 1, Nº 6, pp. 515-529, September 1996.

[Sklyarov98] V. Sklyarov, N. Lau, A. Oliveira, A. Melo, K. Kondratjuk, A. Ferrari, R. Monteiro, I. Sklyarova, “Synthesis Tools and Design Environment for Dynamically Reconfigurable FPGAs”, Proceedings of the XI Brazilian Symposium on Integrated Circuit Design – SBCCI98, pp. 46-49, Rio de Janeiro, September 1998.

[StaWol97] Jørgen Staunstrup and Wayne Wolf, Editors, “Hardware/Software Co-Design: Principles and Practice”, Kluwer Academic Publishers, 1997.

[ThoAdaSch93] Donald E. Thomas, Jay K. Adams, Herman Schmit, “A Model and Methodology for Hardware-Software Codesign”, IEEE Design & Test of Computers, Vol. 10 Nº 3, pp. 6-15, September 1993.

[ThoMoo91] Donald E. Thomas, Philip Moorby, “The Verilog Hardware Description Language”, Kluwer Academic Publishers, 1991.

[WalCha95] Robert A. Walker, Samit Chaudhuri, “Introduction to the Scheduling Problem”, IEEE Design & Test of Computers, Vol. 12, Nº 2, pp. 60-69, Summer 1995.

[Wilkes51] M. V. Wilkes, "The Best Way to Design An Automatic Calculating Machine", Manchester University Computer Inaugural Conference, July 1951.

[WinPro80] David Winkel, Franklin Prosser, "The Art of Digital Design An Introduction To Top-Down Design", Prentice Hall, Inc., 1980.

[Wolf94] Wayne H. Wolf, "Hardware-Software Co-Design of Embedded Systems", Proceedings of the IEEE, Vol. 82, Nº 7, pp. 967-989, July 1994.

[Xilinx97] Xilinx, "XC6200 Field Programmable Gate Arrays", Xilinx Product Description (Version 1.10), 1997.

12 GLOSSARY

ALU	Arithmetic logic unit
ASIC	Application-specific integrated circuit
ASM	Algorithmic state machine
CAD	Computer-aided design
DSP	Digital signal processor
FPGA	Field-programmable gate array
FSM	Finite state machine
GS	Graph-scheme of algorithm
HCFSM	Hierarchical concurrent finite state machine
HDL	Hardware description language
HFSM	Hierarchical finite state machine
HGS	Hierarchical graph-scheme
LUT	Lookup table
MSA	Matrix scheme of algorithm
PAL	Programmable array logic
PFSM	Parallel finite state machine
PHFSM	Parallel hierarchical finite state machine
PHGS	Parallel hierarchical graph-scheme
PLA	Programmable logic array
PLD	Programmable logic device
RAM	Random access memory
RE	Reprogrammable element
ROM	Read only memory
SRAM	Static RAM
VHDL	VHSIC hardware description language
VHSIC	Very high speed integrated circuits