THE UNIVERSITY OF CALGARY

SYNTHESIS OF DIGITAL SIGNAL PROCESSING SYSTEMS USING PIPELINED BIT-SERIAL ARITHMETIC

by

Radhakrishna Nagalla

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF

MASTER OF SCIENCE

DEPARTMENT OF

ELECTRICAL AND COMPUTER ENGINEERING

CALGARY, ALBERTA

November, 1991

ISBN 0-315-75140-1

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled, "Synthesis of Digital Signal Processing Systems Using Pipelined Bit-Serial Arithmetic" submitted by Radhakrishna Nagalla in partial fulfillment of the requirements for the degree of Master of Science.

Dr. L. E. Turner
Supervisor,
Department of Electrical and
Computer Engineering

Dr. M. R. Smith
Department of Electrical and
Computer Engineering

Dr. M. Fattouche
Department of Electrical and
Computer Engineering

Dr. G. Birtwistle
Department of Computer Science

Date : November 21, 1991

ABSTRACT

A technique for the architectural synthesis of pipelined bit-serial digital signal processing systems is presented. Architectural (high level) synthesis is the transformation of an abstract behavioral (algorithmic level) specification of a digital system into a register transfer level (RTL) structure that realizes the specified behavior, while satisfying a set of goals and constraints. The RTL structure refers to a set of components such as arithmetic units, registers, multiplexers and their interconnections, as well as the hardware required to control data transfers between them.

Many synthesis tools have been developed for bit-parallel systems, but only a few synthesis tools have been developed for the high level design of pipelined bit-serial systems. A synthesis tool called BITSYN (BIT-Serial SYNthesis) has been developed exclusively for bit-serial digital signal processing systems. The program BITSYN evaluates different resource sharing strategies (multiplexing) and generates a minimum gate design, a high sample rate design and a range of designs between these limits. BITSYN accepts as input a behavioral description in the form of a signal flow graph (SFG) and generates output in the FIRST language.

The BITSYN tool has been tested using different digital filter signal flow graphs. Digital filter designs obtained from BITSYN have been implemented using XILINX field programmable gate arrays.

ACKNOWLEDGEMENTS

I am grateful to Dr. L. E. Turner for his invaluable guidance and encouragement throughout the course of this research.

I appreciate the financial support provided by the Alberta Microelectronics Center and the Department of Electrical and Computer Engineering which allowed me to undertake the presented work.

My sincere thanks to Peter Graumman for helping me with the implementation of different designs in field programmable arrays.

Dedicated

To

my mother Aruna Kumari

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

The process of extracting the useful information from a signal and discarding the extraneous or the process of modifying the contents of the signal is called (loosely) signal processing [1]. The capability of digital systems to achieve a guaranteed accuracy and perfect reproducibility of signals, leads to the increased use of digital signal processing (DSP) over traditional analog signal processing [2]. The advent of very large scale integrated (VLSI) circuit technology has given a boost to the implementation of powerful real-time DSP algorithms that previously have been of only theoretical interest [3]. Digital signal processing systems are usually well suited to VLSI implementation, since they may have highly parallel algorithmic structure, local connectivity and circuit modularity [4].

A DSP algorithm accepts a new data set every sample period T, where T is the reciprocal of the throughput rate (sample rate). Given an input sample and a current state, the DSP algorithm produces a new state and an output sample. Each state is associated with certain latency (or delay), which is defined as the time separating the appearance of a current state value (state output) to the appearance of a corresponding next state value (state input).

In real-time signal processing systems, sample rate requirements can vary from relatively low sample rates, as in speech and telecommunications, to high sample rates, as in radar and in image processing. In general, any signal processing algorithm can be

executed in different implementation styles such as bit-parallel, digit serial [5] [6] and bit-serial. This classification is based on how many signal bits are processed simultaneously. Bit-parallel systems process all bits of a word or a sample in one clock cycle and are used for high speed applications. Digit serial systems process more than one bit at a time and are suitable for medium speed applications [5] [6]. On the other hand, the bit-serial systems which are typically used for lower speed applications, process one bit at a time [3] [7].

The remainder of this chapter provides an overview of bit-serial systems, architectural synthesis and existing synthesis methods.

1.1 Overview of Bit-Serial Systems

Bit-serial architectures have been proposed to create high-performance, low cost, very large scale integrated circuit (VLSI) systems in different digital signal processing (DSP) applications [3] [7]. Bit-serial systems can be implemented as synchronous or asynchronous systems. In synchronous systems, the sequence and the time of operations are controlled by means of system wide clock signals. The functional behavior of a synchronous system depends on the system clock. The maximum clock rate depends on the delays in the elements and wires. That is, the clock rate must be greater than the maximum delay of any path in the system. In asynchronous systems, there is no global clock signal and the elements are driven by a sequence of operations. In this thesis, only synchronous bit-serial systems are used.

Bit-serial architectures are different from bit-parallel systems mainly in their communication and computational strategies. In synchronous bit-serial networks,

communication and computation are executed synchronously one bit at a time under the control of a global bit clock. This serial communication is implemented with single wires between computing elements (in the data path) and for input and output operations (for inter chip communication). In contrast, N-bit parallel systems use buses of N wires, where N is the system word length.

There are inherent advantages due to the use of serial communication such as

(a) the routing of typical bit-serial networks on an integrated circuit chip is much simpler than the routing of parallel buses. The bit-serial networks also consume less area on an integrated circuit chip unlike the parallel buses.

(b) Input and output communications can be performed via single pins. Thus the number of pins required in bit-serial systems can be less than those required in bit parallel systems.

Bit-serial systems can be constructed in hierarchal fashion from the bottom up, by composing operators into functionally equivalent higher-level operators, starting with the low level operators such as adders, multipliers etc. [8]. Fig 1.1 [3] shows a generic bit-serial operator (computing element) or a primitive. An operator performs a signal processing operation on words of bit-serial data. For example, operators could be add, multiply, delay or Fast Fourier Transform (FFT). Each operator is associated with a set of control inputs, on single wires. These control signals indicate the arrival of the least significant bit (LSB) of a word, or the arrival of the first word of a group of words etc. The control signals are generated in a module called the control generator.

The bit-serial operators are pipelined at the bit-level so that their internal components are used at the best possible throughput rate or at the fastest possible bit clock rate.

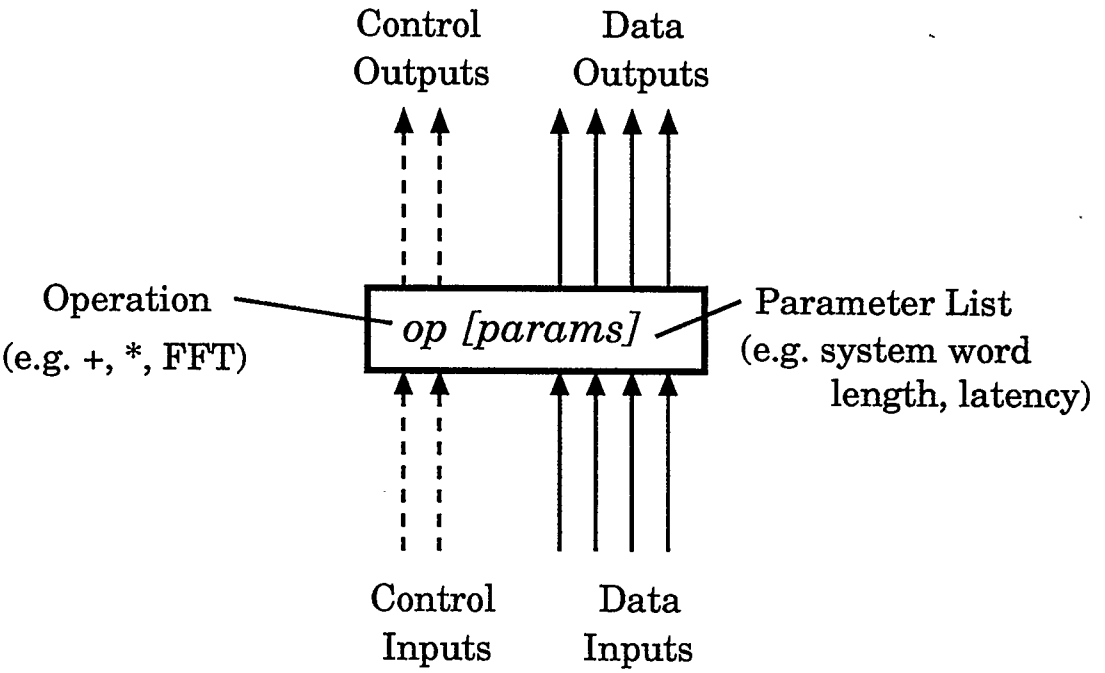


Fig. 1.1 : A generic bit-serial operator [3]

Pipeline operators provide high speeds because their separate stages can operate concurrently. They are similar to a manufacturing assembly line, where different people work concurrently on material passing down the line.

Bit-serial data can be represented with most significant bit (MSB)-first or least significant bit (LSB)-first. Operations such as division and sorting can be performed more easily with MSB-first data because the results can be obtained earlier than with LSB-first data. Common DSP operations such as multiply and add are performed LSB-first because the carry will be propagated from LSB to MSB. A single data format is used to communicate the data between the operators which is independent of the format used in an operator. This convention helps to build arbitrary networks as hierarchies of operators and primitives regardless of a specific data format used in an operator or primitive. The LSB-first format has been chosen in this thesis to communicate the data and to perform the operations. Similarly the bit-serial convention is required only at the interfaces of operators. Internally the operators can adopt signal representations that best suit their operation such as bit-parallel and digit-serial.

With pipelining, there is a finite computation delay or latency associated with each operator. This latency is the time difference between the arrival of input data and the appearance of the output data in synchronous pipelined bit-serial systems. The latency is defined in terms of the bit-times as a number of bits. The latency of an operator may be a constant as in the case of serial adder (latency = 1). The latency of an operator may also be a function of certain parameters such as the coefficient word length (cwl) as in the case of a serial multiplier.

In bit-serial systems, the data is processed bit by bit serially and if necessary the results from the previous computation are stored and used internally. For example in the case of bit-serial carry save adder the previously computed and stored carry is added with the current bits of the two input words as shown in Fig. 1.2 [3]. The control signal required in the case of the serial adder is a pulse synchronized with the arrival of the least significant bits of both the inputs (LSB time). The LSB time signal inhibits the carry resulting from the previous addition.

The hardware required for bit-serial primitives is typically much smaller than bit-parallel primitives at the expense of the data throughput rate. In general, serial structures use 1/Nth of the hardware required by parallel structures but take N times the bit-parallel computation time. Thus, for the same clock rate, the serial and parallel structures will have the same area-time measure. In practice, the (non-pipelined) parallel system clock should be much slower due to the potential for long carry propagation delays in add and multiply operators. These delays are not present in bit-serial systems because of pipelining at the bit-level. Thus bit-serial systems support a faster clock compared to bit-parallel systems and in turn offer implementations which are efficient in area-time measure.

The advantages of a bit-serial architecture can be exploited to implement DSP algorithms in high performance, cost effective VLSI circuits. An integrated circuit design is composed of a number of tasks such as problem specification, logic design, design for testability, placement and routing. The different tasks can be performed separately using a number of design automation tools. A silicon compiler or assembler combines all these tasks into one step and converts the structural description of a design into a set of mask

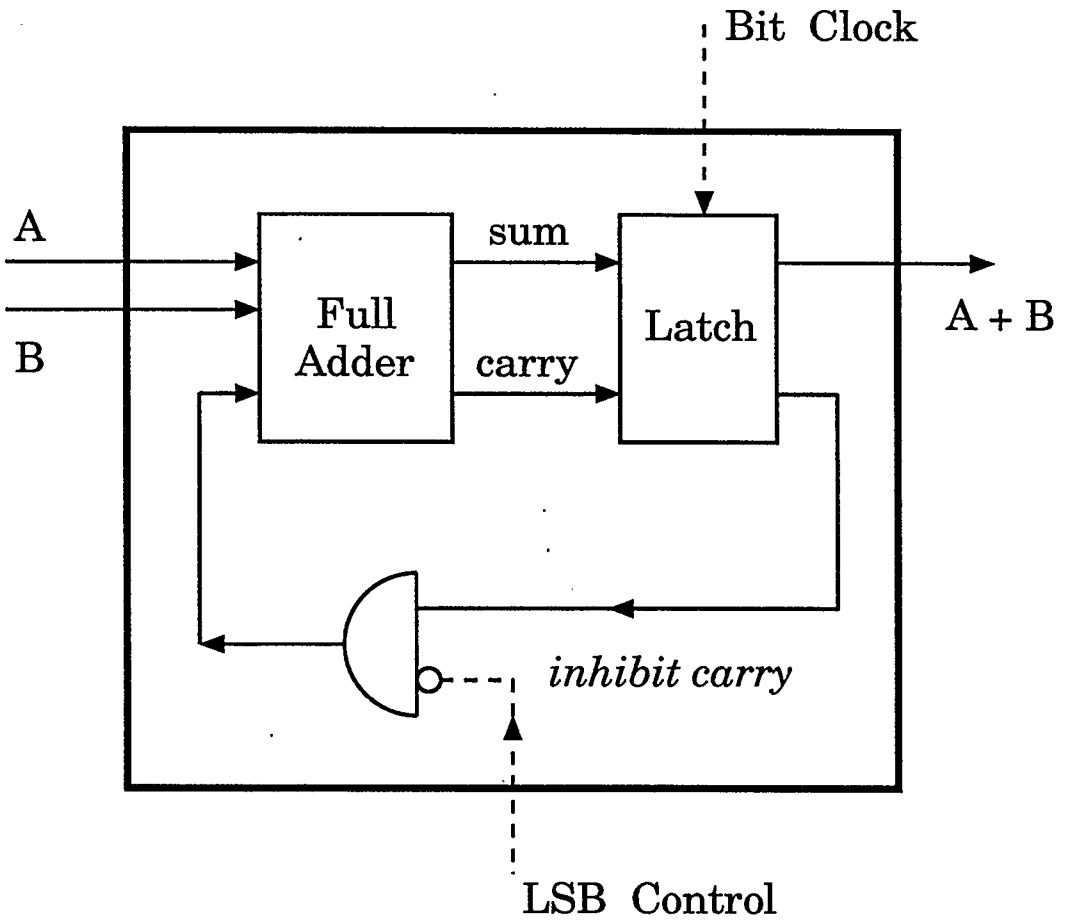


Fig. 1.2 : A bit-serial adder [3]

geometries, in a single step. The FIRST [3] (Fast Implementation of Real-time Signal Transforms) silicon compiler is based on a bit-serial architecture proposed by Jackson, Kaiser, and McDonald [9]. The silicon compiler was built around an architectural methodology for VLSI bit-serial systems proposed initially by Lyon [8] and later extended by Denyer, Renshaw [3] and Bergmann [10]. FIRST accepts the input in the form of hardware description language (see appendix B) at the register transfer level (RTL). A simulator (SIMFIRST) which accepts the same RTL input is also available [3].

1.2 Architectural Synthesis

To use a silicon compiler such as FIRST (probably more correctly a silicon assembler), the designer converts a behavioral specification of the system into a structural level description. For pipelined bit-serial systems, the process of scheduling complicated recursive DSP algorithms and in turn generating structural level description is tedious and error prone. It is very difficult for a designer to choose a structural level solution among many solutions that may be available for the same behavioral specification. Architectural synthesis, an important front end task to a silicon compiler, allows the designer to select a RTL solution that realizes the required behavior.

Architectural synthesis or high level synthesis [11] is the transformation of an abstract behavioral (algorithmic level) specification of a digital system into a register transfer level (RTL) structure that realizes the specified behavior, while satisfying a set of goals and constraints. The behavioral description gives the functional mappings from a group of inputs to a group of outputs and it can be specified in the form of a signal flow graph (SFG) or a

netlist. The RTL structure refers to a set of components such as arithmetic units, registers, multiplexers and their interconnections as well as the hardware required to control data transfers between them. These structures are represented using a hardware description language, such as the input description language for the FIRST silicon compiler. One objective of the synthesis process is to find a structure that best meets the given constraints such as

(a) upper bound on the area or package size,

(b) maximum number of pins,

(c) maximum power consumption

(d) minimum system word length (swl) and

(e) overall delay allowable for completion of the operations

while achieving the goals such as

(a) maximizing the sample rate

(b) minimizing the size or area

(c) minimizing the number of pins and

(d) minimizing the power consumption.

The different steps involved in the design automation process of a DSP system from algorithm to implementation are shown in Fig. 1.3. Hardware Assembly is the process where RTL structural information is converted into different end products of the design process such as process masks used to fabricate full custom and gate array integrated circuits or a data file which defines the configuration of a programmable gate array.

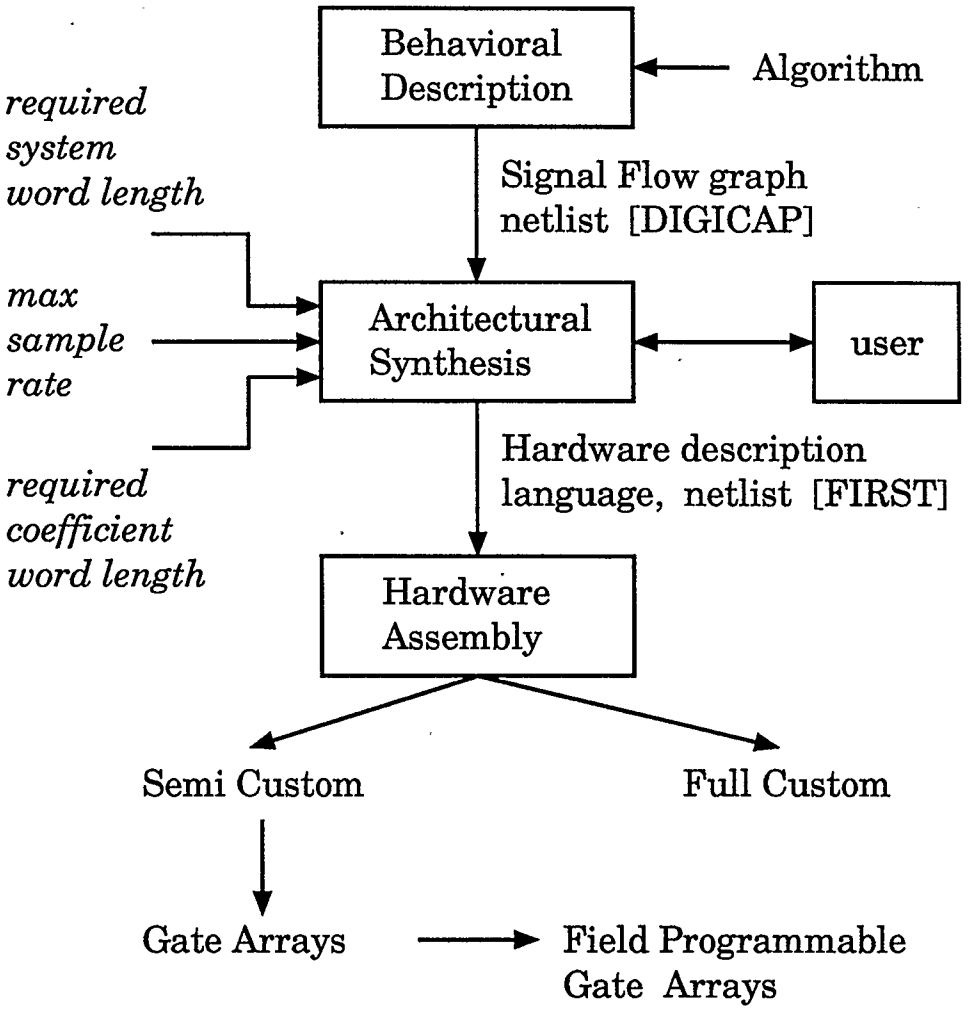


Fig. 1.3 : Design process of a digital signal processing system

Hardware compilation can be performed using a single tool such as the silicon compiler FIRST [3], or commercial design automation (DA) tools such as VTI [12], Cadence [13], XILINX [14], etc. These DA tools accept inputs in the form of different netlists with specific formats. To use any of the existing DA tools, translator programs which can convert hardware description language description into specific netlists have to be developed. Once the design is translated to a specific netlist, the DA tools can be used to generate the data necessary to produce integrated circuit designs.

1.2.1 Advantages of Architectural Synthesis

The use of high-level synthesis tools in the design of large complex systems is appropriate for the following reasons [11].

(a) Shorter design cycle : Without high level synthesis, the designer converts the algorithmic level information into a RTL structure manually. Then the RTL structural information is entered in hierarchical fashion using schematic capture methods or hardware description languages (HDL). High level synthesis automates the process of generating HDL output, which results in a shorter design cycle. Since much of the cost involved in producing integrated circuits is in the design development, automating the design process can lower the cost significantly.

(b) Flexibility : If the specification of a design is changed during the design cycle, changes at the algorithmic level are easily implemented.

(c) To explore the design space : Synthesis systems can be used to produce a number of functionally equivalent designs for the same input specifications in a relatively short time.

Thus the user may search for an optimum design after considering the different performance trade-offs of cost, speed, and power consumption.

(d) Fewer errors : If the automated synthesis process can be verified to be correct, then it will become error-free or correct by construction. This will result in fewer errors and less debugging time for new chips. The synthesis process also reduces the possibility of human errors in the design process.

(e) Simplicity of use: High level synthesis tools can aid designers in developing systems using different unfamiliar design methodologies such as bit-serial or digit-serial.

1.2.2 Existing Synthesis Methods

A number of synthesis tools such as HAL[15], SEHWA[16], BUD[17], MAHA[18], and SPAID[19] have been developed for bit-parallel systems. These systems produce architectures with a minimum number of functional units for a given throughput rate or architectures with the highest possible throughput rate for a given number of functional units. In all the existing synthesis tools, the task of automatically designing a data path from a behavioral description is usually divided into the four sub tasks : translation, scheduling, allocation and binding [11] [15]. In the translation process, the algorithmic description is converted into a graph-based internal representation derived from the data flow. These graphs are called data flow graphs (DFG). The data flow graph shows the correct order in which all the functional operations must be completed. All internal dummy variables used in the behavioral specification are removed in the data flow graphs. Each node in a DFG represents an operation to be performed by a functional unit.

The scheduling operation partitions the DFG into a number of control steps and assigns each operation to a control step. A control step corresponds to a fundamental clock cycle in synchronous circuits. The aim of the scheduling is to minimize the number of control steps, within the limits of the hardware resources. If the design is subjected to a speed constraint, the scheduling algorithm will try to run a sufficient number of operations in parallel to meet the speed constraint. Conversely if there is an area constraint, the scheduler serializes enough operations to meet the size constraint.

Allocation is the assignment of operations to hardware which consists of functional units, memory elements or registers and data transfer components such as multiplexers and buses. During allocation, the aim is to minimize the total hardware required to implement the architecture. The problems of minimizing different types of hardware units are dealt with individually in most of the synthesis systems, as minimizing them together is too complex.

Scheduling and functional unit allocation are interdependent processes [11]. In the case of a single functional unit, two operations cannot be scheduled into one control step unless they can be shared by the same functional unit. The most efficient schedule is possible only by considering the actual delays associated with the real hardware along with the interconnect delays. On the other hand, scheduling helps to determine the mutually exclusive operations (operations which can be done in parallel), which in turn helps to determine the number of functional units needed and how the required operations are distributed among them.

A number of approaches have been used by different synthesis systems to solve this problem. The simplest approach is to set some limit (or no limit) on the number of

functional units and schedule the operations among the functional units. This approach is used in the Facet system [20] and in the Flamel system [21]. A modification of this approach is to change the limit on the number of functional units after each iteration, based on the intermediate scheduling results until a satisfactory design has been found. Another approach is to consider scheduling and resource requirements simultaneously. The MAHA [18] system allocates operations to the functional units as it schedules. It considers the operations on the longest path first. When the operations cannot share the existing functional units, it allocates additional functional units if the cost constraint is not exceeded. If more resources are not available due to the cost constraint, it increases the number of control steps and begins the allocation all over. The HAL [22] system schedules the operations within a timing constraint so as to balance the number of functional units required in each control step.

The final step in the synthesis of a data path is module binding. In this step, the synthesis system decides how each component of the datapath is to be implemented. For binding of the functional units such as multipliers, adders etc., pre-designed gate level structures can be taken from the hardware library.

Finally the control signals necessary to drive the different signal paths as required by the schedule, have to be generated. The controller can be synthesized using finite state machines or microcoded hardware.

There are few tools such as DIGICAP [23] and CATHEDRAL-I [7] available for the high level design of pipelined bit-serial systems. All these tools generate a single structural solution where the number of functional operations is equal to the number of the functional

units. DIGICAP is a special purpose computer aided design (CAD) tool for the analysis and implementation of digital filters. DIGICAP can be used to generate a FIRST silicon compiler netlist from a signal flow graph description. In CATHEDRAL-I, a number of rules for the synthesis of bit-serial architectures have been formulated based on practical design experience. These rules are used in a knowledge based CAD environment. In this tool all constant coefficient multipliers are replaced with canonic signed digit (CSD) equivalents. A CSD equivalent consists of 2-input adders and power of 2 scalers (shifters). Optimization programs are used to reduce the number of delays. Another behavioral to structural translator is developed for bit-serial silicon compiler by Hartley and Jasica [24]. Here linear programming techniques are used to optimally insert the required delay units.

In summary, all these programs available for bit-serial systems translate a behavioral description into a single structural description. Some of these programs uses optimization techniques to reduce the number of delay units.

Different resource sharing (multiplexing) strategies can be used to generate a number of multiplexed pipelined bit-serial solutions for the given behavioral specification.

1.3 Scope of Thesis

This thesis is concerned with the development of a new synthesis tool for pipelined bit-serial DSP systems. A bit-serial synthesis (BITSYN) tool is developed that can generate a number of design solutions with different hardware costs, sample rates and system word lengths.

Chapter 2 describes the different steps involved in the bit-serial synthesis tool BITSYN [25]. BITSYN accepts the behavioral input in the form of SFG description. The details of the SFG description are also given.

Chapter 3 describes the implementation of the overall algorithm for bit-serial synthesis. The overview of the design space that can be generated by BITSYN is explained. The heuristics which can be used to speed up the synthesis process are also discussed.

Chapter 4 discusses the design of control generators compatible with the FIRST and the design of different type of control generators required by BITSYN. In chapter 5, a range of case studies generated by BITSYN are given. Some of the designs generated by BITSYN are implemented in XILINX field programmable gate arrays. In chapter 6, the conclusions and suggestions for future work are presented.

Chapter 2

BIT-SERIAL SYNTHESIS (BITSYN)

This chapter discusses a new approach to the synthesis of pipelined bit-serial digital architectures that implement the algorithms of digital signal processing (DSP) systems. The class of DSP systems considered here are digital approximations to linear shift-invariant discrete-time systems [2]. In a broad sense, the methods of realization of the DSP algorithms can be divided into two classes, recursive and non-recursive [2]. For a recursive realization, the functional relationship between the input sequence {x(n)} and the resulting output sequence {y(n)} can be described as

$$y(n) = F[y(n-1), y(n-2), ..., x(n), x(n-1), ...]$$

Thus in recursive structures, the present output sample y(n) is a function of weighted past output samples as well as present and past weighted input samples. That is the recursive structure contains feedback from its output. For a non-recursive realization, the functional relationship between the output and the input sequence becomes

$$y(n) = F[x(n), x(n-1), ...]$$

In non-recursive structures, the present output sample is a function of only past and present input samples; that is the structure contains no feedback.

In general, a DSP algorithm can be realized in operator-parallel, multiplexed and operator-serial implementations [3]. The different realizations are based on the number of

functional operators in an implementation. In an operator-parallel implementation, each functional operation of a DSP algorithm is executed by a separate functional unit or an operator, resulting in the fastest data throughput rate and large hardware cost. In an operator-serial and multiplexed implementations, where similar operations are executed by fewer functional units than the number of operations, a slower data throughput rate and smaller architecture results. Operators can be implemented in bit-parallel, digit-serial [6] and bit-serial. The bit-serial synthesis [BITSYN] program [25] has been developed to evaluate a range of multiplexed and operator-parallel pipelined bit-serial implementations. The development, implementation and application of BITSYN is discussed in this thesis.

The different properties of bit-serial operators must be known before discussing a synthesis strategy for bit-serial systems. The typical bit-serial primitive operators are add, subtract, multiply, multiplexer, right shift and left shift. These primitive operators can be used to implement the high-level signal processing functions such as the fast Fourier transform (FFT), a digital filter, etc. The bit-serial multiplier is normally the largest operator in size. There is typically an order of magnitude or more difference between the size of all other operators and the bit-serial multiplier. For example, in a particular FIRST [3] implementation, a bit-serial multiplier with a coefficient word length (cwl) of 4 bits requires 260 gates whereas an adder requires only 30 gates. Hardware can be significantly reduced by reducing the number of multiplier operators in bit-serial systems.

In bit-serial systems, the system latency is calculated as sum of the latencies of all the operators on a critical path and is dependent on the system implementation. A critical path is defined as a path of operators with the largest total latency from a state output or a

system input to a state input or system output. In recursive or feedback systems, the system word length (swl) cannot be less than the system latency calculated from state output to the corresponding state input. Thus the recursive structure imposes a minimum bound on the system word length (swl). The *swl* is also limited by the application defined required *swl* constraint, which is determined from an analysis of the nonlinear effects of finite precision arithmetic [23].

The automatic design of a bit-serial system from a functional description is performed in four different phases which are translation, resource sharing, scheduling and allocation. In the translation step, the algorithmic description in the form of signal flow graph (SFG) is converted into data flow graphs (DFG). In resource sharing process, the objective is to reduce the number of multiplier operators implemented, while meeting the sample rate and *swl* constraints. The number of multiplier operators will be reduced when the operators share more than one multiplication operation.

Scheduling takes the latency of each of the operators into account and calculates a time schedule for all operators. The time schedule gives the times at which each functional operation takes place in an operator. The system latency for the scheduled architecture is also determined at this stage. Allocation is the actual mapping of the operators to specific functional units. The registers and data transfer components such as multiplexers are also allocated to realize the resource shared DFG. The different steps involved in the synthesis process will now be described in more detail.

2.1 Translation

BITSYN accepts as input a compact, high-level, signal flow graph (SFG) description which is similar to the SFG description used in DIGICAP [23]. In the translation step, the SFG description is converted into a more detailed internal graphical representation data flow graph (DFG). These DFGs are a modified version of the precedence graphs given in DIGICAP [26]. Each node in the DFG represents a functional operator that performs an operation. When the SFG is converted into a DFG, all the functional operations are replaced by the corresponding functional operators. Thus the DFGs represent the operator-parallel implementation. Any syntax errors in the SFG description will be found and suitable error messages will be generated at this stage.

2.1.1 Signal Flow Graph (SFG) Description

The SFG description is in the form of branches connecting a set of nodes. The nodes are represented by alphanumerical variable names and are connected by different types of branches. The branch types are

(a) delay src dst

defines a unit delay from the output of a source node (src) to a destination node (dst).

(b) mult src1 dst src2 {-c *cwl* -l *lat*}

defines that an input of the node *dst* is equal to the product of the outputs of the source nodes *src1* and *src2*. If *src2* is a numerical variable, then it indicates that the

multiplication is by a constant coefficient. If *src2* is an alphanumerical variable then it indicates that the multiplication is by a signal. The 'mult' branch type can also be used to specify shift operations. If *src2* is specified as 'shift n' (where n is a +ve or -ve integer) then it indicates that the multiplication is by constant coefficient and the coefficient value is 2^n. It also specifies that the multiplication operation should be implemented using bit-shifting operations.

In addition, the 'mult' branch has two optional specifications, coefficient word length (cwl) and latency (lat). The *cwl* option is provided so that multipliers with different 'cwl' can be used in the same design. The default latency of the multiplier used in a FIRST implementation is a function of the *cwl*. The latency is related to *cwl* as

$$latency = \left\lceil \frac{3}{2} cwl \right\rceil + 2 \qquad where\ cwl = multiple\ of\ 2.$$

It is possible to design multipliers with reduced latency. The *lat* option is provided so that multipliers with different latencies can be used.

(c) input dst type

indicates that the node *dst* is a signal input. The *type* specifies the form of the input required by DIGICAP for analysis purposes [ac, sin, pulse, or file]. BITSYN does not need the *type* information.

(d) output src

specifies that the node *src* is the signal output.

The output of any node is equal to the sum of all inputs to that node. Subtraction is implemented by using an appropriate 'mult' branch with its *src2* equal to -1. The branch commands described above are similar to the commands of the DIGICAP [23]. These commands can be used to specify the structure of any uniform sample rate, one-dimensional digital filter. The signal flow graph of a second order wave digital filter [27] is given in Fig. 2.1a and the corresponding BITSYN SFG branch description is provided in Fig. 2.1b.

A generic branch command has been provided to extend the synthesis capability. Six different generic commands x0nnn, x1nnn, x2nnn, x3nnn, x4nnn, x5nnn are introduced. All these commands can be specified with similar syntax. For example, the x0nnn can be specified as

(e) x0nnn src1 dst src2 -l *lat* **-g** *gates*

This command indicates that the input to the *dst* node is some generic function of the outputs of *src1* and *src2* nodes.

The terms x0nnn, x1nnn, ... x5nnn represent six different user specified operations. These commands can be used to represent any new DSP operation if the latency, *lat* of the operator and the cost of the operator in terms of the number of gates, *gates* are known. Note that the function of the generic operator is not needed by BITSYN. These commands can also be used to perform hierarchal synthesis. For example, a system which consists of more than one filter can be synthesized in two phases. In the first phase, a single filter is synthesized from a SFG description implemented using only primitive operations. Once the filter is designed, the latency and the hardware cost in terms of number of gates will be

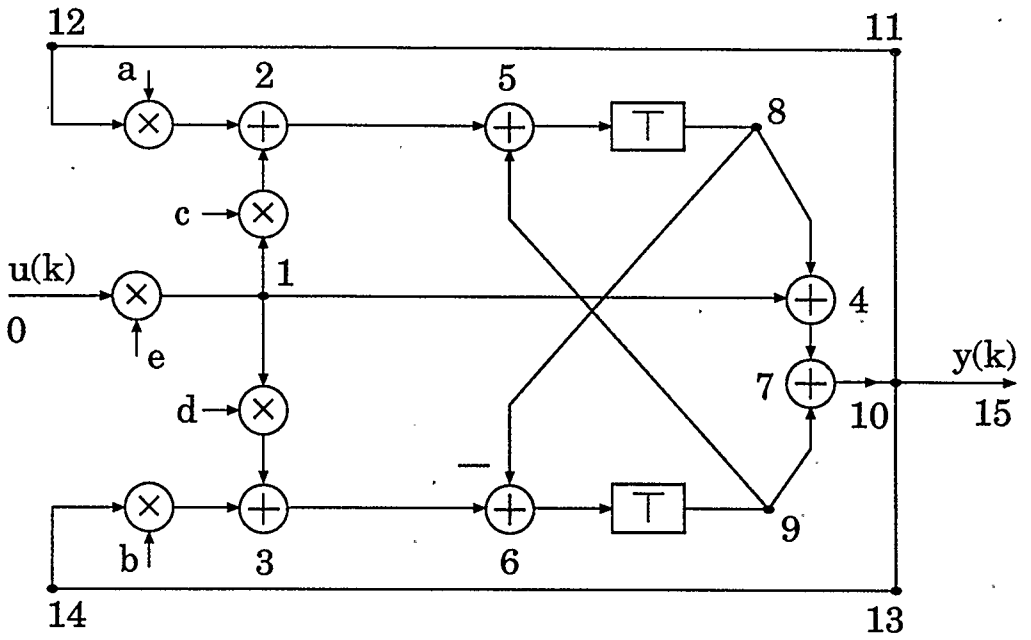


Fig. 2.1a : Second order wave digital filter signal flow graph

input	0	ac		mult	1	2	c
mult	0	1	e	mult	1	3	d
mult	1	4	1.0	mult	2	5	1.0
mult	4	7	1.0	mult	3	6	1.0
mult	7	10	1.0	mult	9	5	1.0
mult	10	15	1.0	mult	8	6	-1.0
mult	10	11	1.0	mult	8	4	1.0
mult	11	12	1.0	mult	9	7	1.0
mult	12	2	a	delay	5	8	
mult	10	13	1.0	delay	6	9	
mult	13	14	1.0	output	15		
mult	14	3	b				

Fig. 2.1b : Second order wave digital filter
- BITSYN branch description

known. In the second phase, the signal flow graph description of the system is specified using one of the six branch commands, to represent the filter operation. Then the BITSYN can design the whole system accepting the new SFG description as input.

2.1.2 SFG to DFG Conversion

The SFG description is first converted into a modified SFG net list before being translated to the DFGs. The nonessential multiplier branches with unity gain, if any, are removed at this stage. The multiplier branch commands are interpreted and the new nodes such as adders, subtracters, right shifts and left shifts are introduced appropriately.

The DIGICAP compatible multiplier branches with shift operations such as

mult *node_s* node_x *shift 2*
mult *node_s* node_x *shift -1*

are redefined as

MSHIFT *node_s* node_x *2*
DSHIFT *node_s* node_x *1*

where for example a two bit MSHIFT operation indicates multiplication of the signal by 2^2 and a one bit DSHIFT operation indicates multiplication by 2^{-1}.

The two multiplier branches associated with *node_x*

mult *left_node* node_x *left_value*
mult *right_node* node_x *right_value*

generate an expression of the form

node_x = left_node * left_value + right_node * right_value.

The two multiplier branches will be replaced by the equivalent branch commands as shown

in Table 2.1. The equivalent branch commands will represent the arithmetic operations such as add, subtract and shift operations, explicitly.

Table 2.1 : Transformations to modify SFG branch commands

left_value	right_value	equivalent branch commands
1	1	**add** *left_node* node_x *right_node*
1	≠ 1	**mult** *right_node* node_x_r *right_value* **add** *left_node* node_x *node_x_r*
≠ 1	1	**mult** *left_node* node_x_l *left_value* **add** *right_node* node_x *node_x_l*
-1	≠ 1	**mult** *right_node* node_x_r *right_value* **sub** *node_x_r* node_x *left_node*
≠ 1	-1	**mult** *left_node* node_x_l *left_value* **sub** *node_x_l* node_x *right_node*
-1	-1	**add** *left_node* node_x *right_node* * **mult** *node_x* node_x_new *-1* *
≠ ±1	≠ ±1	**mult** *left_node* node_x_l *left_value* **mult** *right_node* node_x_r *right_value* **add** *node_x_l* node_x *node_x_r*

* This transformation is possible if and only if the resultant multiplier with negative unity coefficient can be used in other transformations and no negative unity coefficient multiplier exist at the transformation process. Otherwise an error will be reported and the synthesis process will be terminated.

The different rules given in Table 2.1 are used to modify the initial SFG. The modified SFG net list for the filter in Fig. 2.1a is given in Fig. 2.2a. In the modified SFG, each functional operation is represented by a separate branch command.

2.1.3 Data Flow Graphs (DFG)

The first step in the design of a circuit is to construct graphs derived from the data flow of the algorithm. These graphs are directed graphs whose vertices or nodes represent operators or operands. The graphs have directed edges which indicate the direction of flow of the data. The modified SFG description given in Fig. 2.2a is converted to a DFG as shown in Fig. 2.2b. Two groups of identical nodes which could be shared in the resource sharing process are shaded in Fig. 2.2b. The resource sharing process involving the two groups will be discussed in the next section. The DFGs are built using a tree structure in which the leaf nodes represent the data inputs and the current states (state outputs) and the parent nodes at the top of the trees represent the outputs and the next states (state inputs).

In the DFG, the nodes with same names represent a single real instance of the node. The DFG defines the order in which nodes must be evaluated for correct operation. A node in the graph (excluding parent and leaf nodes) denotes an arithmetic functional unit or an operator and it can be evaluated if and only if all its child node values have been previously computed. A node P is recursively defined as child of another node Q if node P is either one of the inputs of node Q or is child of one of the inputs of node Q. Similarly, a node Q is defined as parent to another node P if node P is a child of node Q. For example, it can be observed from the DFG given in Fig. 2.2b that the output node 7 is child to next state nodes 5 and 6. So the next state values (nodes 5 and 6) can be obtained only after evaluating the output value from node 7. For each of the outputs and the state inputs, separate graphs are constructed. A node in these graphs has always two immediate child

input	0	ac		add	1	4	8
mult	0	1	e	add	2	5	9
mult	7	2r	a	sub	3	6	8
mult	7	3r	b	delay	5	8	
mult	1	2l	c	delay	6	9	
mult	1	3l	d	add	4	7	9
add	2l	2	2r	output	7		
add	3l	3	3r				

Fig. 2.2a : Second order wave digital filter
- modified SFG description

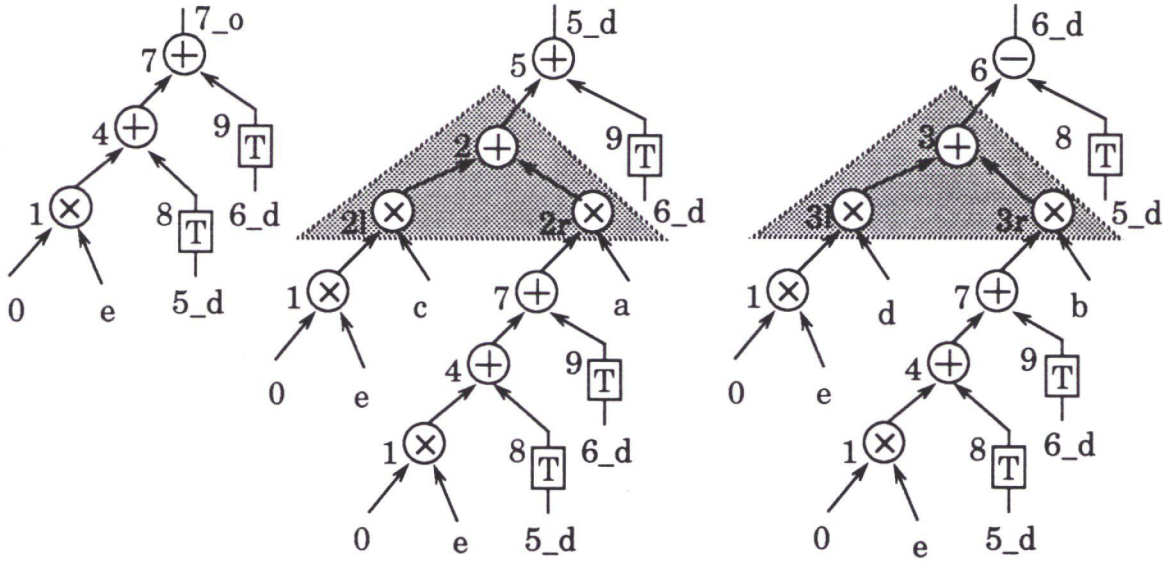


Fig. 2.2b : Second order wave filter data flow graphs (DFG)

nodes, but it can be a child to any number of nodes. That is the output of any node can be used as inputs to different nodes.

The DFG is generated using binary tree structure in BITSYN. A node in the DFG is implemented as shown in Fig. 2.3. Each node in the DFG is represented by a data structure called a *tree_node*. The *tree_node* has a number of fields such as **name** (node name), **type** (operation of the node), etc. which are used to store different characteristics of that node. The **ident** field gives a distinct identification number. All the nodes in a tree are numbered from bottom to top. The field **child** is a string of binary digits in which the positions of the digit '1' (from left side) represent the identification numbers of all the child nodes of the *tree_node*. The 'ident' and 'child' fields will be used to find whether a node is a child of another node. The field **latency** indicates the latency of the bit-serial operator. The field **level** gives the level of multiplexing, if the *tree_node* is a multiplexer or a demultiplexer. In other cases, it is used as conditional variable.

The *tree_node* has three pointer fields **left**, **right** and **prev** which are used to link the child and parent nodes. The **left** and **right** fields are the same as the data structure *tree_node*. The **prev** field is a data structure called *tree_link*. It has two pointer fields, one of which (**node**) is same as the *tree_node* and the other (**next**) is equal to *tree_link*. The data structure *tree_link* can be used to specify multiple previous nodes or multiple immediate parents of a node. Basically the *tree_link* provides a linked list. The DFGs are implemented in the form of binary tree structures with the capability of doubly linked lists as shown in Fig. 2.3. Thus, the DFGs are created such that it is possible to access both parent and child nodes from any node by moving upwards or downwards.

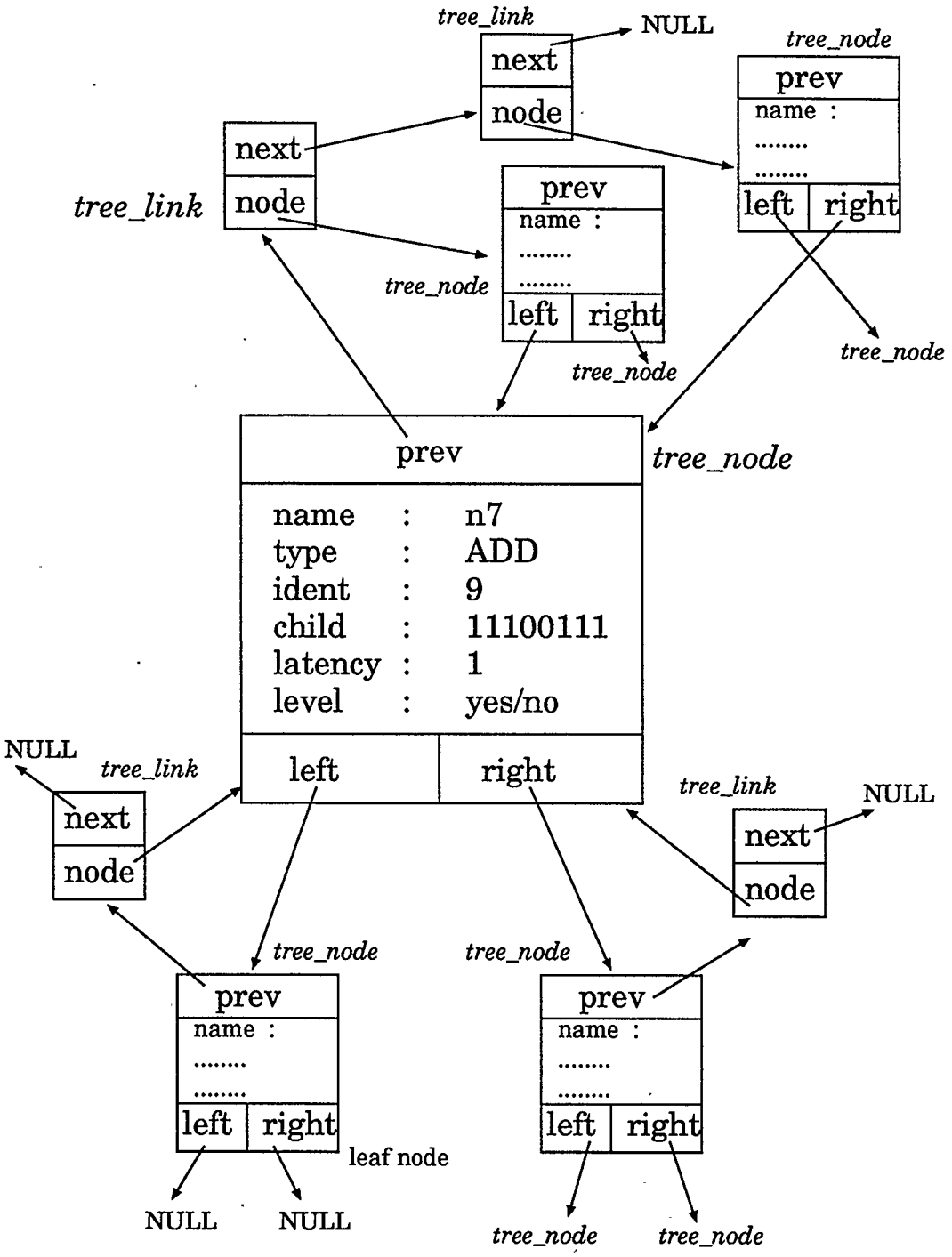


Fig. 2.3 : Implementation of a typical node in DFG

2.2 Resource Sharing

The DFG without any modification provides the data path structure with full operator-parallelism because each node operation in the DFG will be performed by a separate functional unit. The resource sharing process is used to generate multiplexed pipelined bit-serial architectures from the DFG. Whenever a resource or an operator performs two or more operations, it can be referred as the operator shares the operations. Multiplexers and demultiplexers must be added to channel the inputs to and the outputs from the resource whenever a resource shares the operations. ts from the resource. In addition, registers may be needed to store intermediate outputs. Individual add or subtract operations are not shared because the gate cost of a single adder or subtracter is not much more than the gate cost of a multiplexer. Thus the objective is to multiplex the multiplier operators or generic operators which are larger in size.

The first step in the resource sharing process, is to form a list of pairs of nodes. Multiplier nodes and some of the generic operators are used to form the list of pairs. The nodes in each pair should be of similar type and should have the same latency. Two similar nodes P and Q can produce two different pairs (P, Q) and (Q, P). Fig. 2.4a illustrates the difference between the two node pairs. In the figure, P and Q represent multipliers that perform multiplication operations. After the resource sharing process, the two pairs result in different structures. The difference between the two pairs is the order in which the two operations take place. The two multiplication operations are performed in 'ON' and 'OFF' periods of the multiplexing control signal 'c1n'. Note that the node whose operation occurs first, remains in the DFG and the other node is removed from the DFG. That is, the first

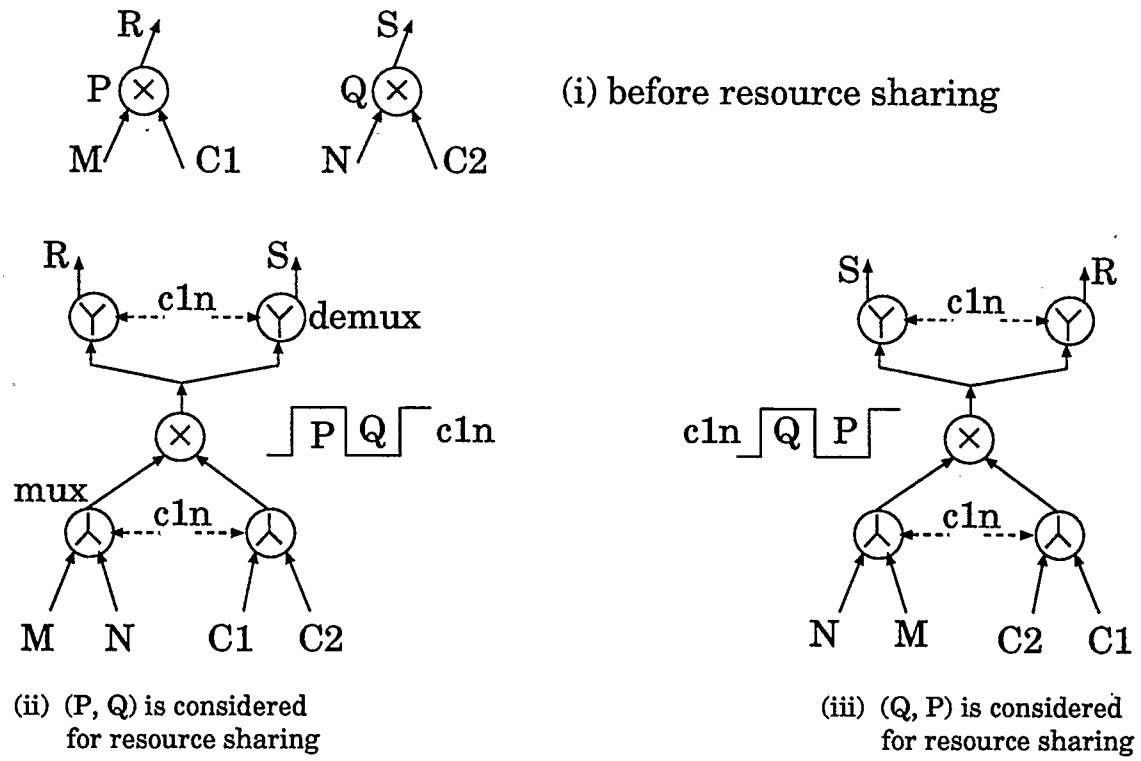


Fig. 2.4a : Illustration of the difference between node pairs (P, Q) and (Q, P) in resource sharing process

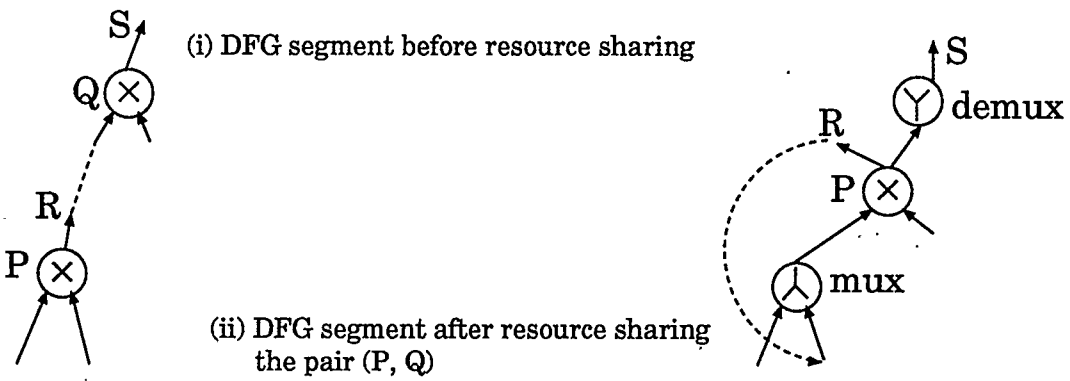


Fig. 2.4b : Illustration of the resource sharing process of (P, Q) when a node 'P' is a child of 'Q'

operator performs both operations in the hardware to be implemented. Thus two different pairs are formed for each pair of nodes.

The next step is to remove the invalid pairs from the list of pairs of multiplier nodes. If a node P is a child of node Q then (P, Q) is valid and (Q, P) is invalid. The output from P is needed to perform the Q's operation. This is illustrated in Fig. 2.4b. But if node P is not a child of node Q and node Q is not a child of node P then both pairs are valid.

Consider a pair of multiplier nodes from the list of pairs of multiplier nodes. All other nodes (adders, subtracters, etc.) which can be grouped with the multipliers are multiplexed along with the multiplier nodes. If two identical nodes are connected to the multiplier nodes through an equal number of identical nodes then they can be grouped with each multiplier. After the resource sharing process, one multiplier with its group remains in the DFG and performs the operations of both groups. The multiplexer nodes and demultiplexers are introduced appropriately. The algorithm processes the data flow graphs, and modifies them as the resource sharing process continues.

2.2.1 Resource Sharing Algorithm

The resource sharing algorithm is implemented in a BITSYN module called **perform_mux**. This module takes a pair of multipliers, e.g. (A, B), as input and performs the multiplexing. The module consists of calls to different functions as shown in Fig. 2.5. The resource sharing algorithm finds identical nodes which can be grouped with each multiplier and performs multiplexing.

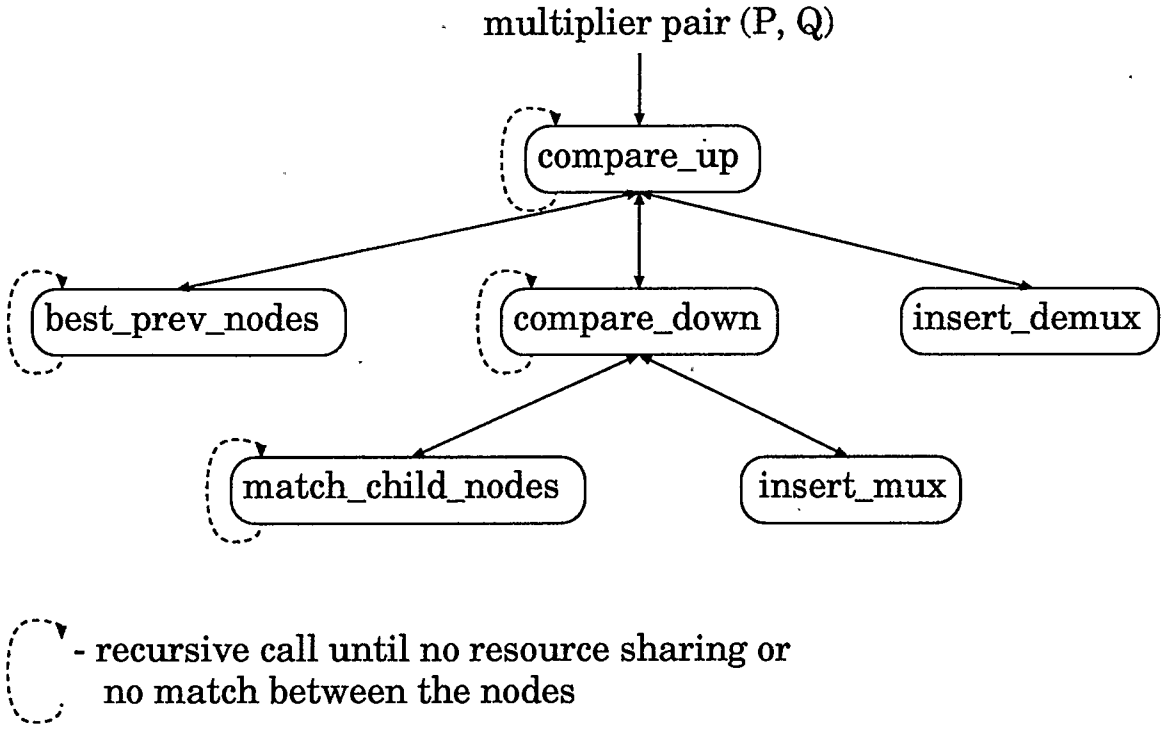


Fig. 2.5 : Resource sharing program structure chart

The function **Compare_down** compares the child nodes of (A, B) and if they are alike, forms pairs of nodes. If the child nodes are not of similar type, the function introduces the multiplexer nodes. If they are alike, the function calls itself with new pairs as input. The function calls itself recursively and moves down the tree until further resource sharing is not possible or the function reaches the leaf nodes.

The function **Match_child_nodes** forms the pairs among the child nodes of (A, B) such that the maximum number of nodes can be multiplexed. If both child nodes of 'A' and 'B' are alike, then two pairs can be formed in two different ways. This function takes each pair as input and recursively finds the number of possible pairs down the tree. Thus this function helps to take a correct decision in making the pairs.

Insert_mux creates and inserts a multiplexer node into the DFG. The multiplexer node will have the first node 'A', as the previous (immediate parent) node and a child of 'A' as the left child node and a child of second node, 'B' as the right child node. These nodes represent a two to one multiplex operation and are identified with the inverted 'Y'. They are named as 'xn....' where **n** denotes the level of multiplexing.

The function **Compare_up** is similar to the **Compare_down** except that it compares the immediate parents of (A, B) and if they are alike, forms pairs of nodes. This function also uses the **Compare_down** to compare the child nodes of each new pair other than 'A' and 'B'. This function calls itself recursively and moves up the tree until further multiplexing is not possible or the function reaches the top of the tree. If the parent nodes are not of the same type, there will be no further multiplexing and the demultiplexer nodes are introduced.

Best_prev_nodes forms the pairs among the parent nodes of A and B in such a way that the maximum number of nodes can be multiplexed. This function takes each pair as input and finds the number of possible pairs up the tree.

The function **insert_demux** creates and inserts a demultiplexer (demux) node into the DFG. The demux nodes are only the conceptual nodes and not the real demultiplex operators. These nodes are identified with **Y**. There are two different types of demultiplexers. The node named 'Ln...' indicate that the connection between child and parents of this node is valid during ON period of the control signal of the multiplexer. The nodes named 'Ln..., will have the parents of A as the previous (parent) nodes. Similarly the node named 'Rn...' indicate that the connection between the child and parents of this node is valid during OFF period of the control signal. The nodes named 'Rn...' will have the parents of B as the previous (parent) nodes. Both demultiplexers will have the node A as their left child and no right child.

In all the shared pairs, the first nodes will be calculated during the ON period of the control signal of the multiplexer. The second nodes will be calculated during the OFF period. Hence, a second node of a shared pair should not be a child to any of the first nodes of all the shared pairs. Otherwise the first node which depends on the output of the second node will be calculated incorrectly during the ON period.

The different steps involved in the resource sharing process are explained with the help of the DFG of the second order wave digital filter shown in Fig. 2.2b. There are five multiplier nodes in the DFG. They can form sixteen valid pairs of multiplier nodes. For the pair of multipliers (2l, 3l) in Fig. 2.2b, the pairs of nodes that can be multiplexed along with

(2l, 3l) are (2, 3) and (2r, 3r). The resultant resource shared DFGs are shown in Fig. 2.6a. The nodes named 'x1...' represent two to one multiplexers. The nodes named *L1a* and *R1a* are the conceptual demultiplexers. Thus after the first level of resource sharing, the number of multiplier operators is reduced from five to three. The resultant DFG can be implemented after scheduling and allocation steps. The multiplier pair (2r, 3r) could have provided the same DFGs obtained from (2l, 3l) after resource sharing. Hence, the pair (2r, 3r) is redundant. Different multiplier pairs provide different structures (hardware solutions) after resource sharing. The DFG generated for each solution (after the first level multiplexing) can be subjected to further resource sharing.

Before considering further sharing, the DFGs obtained after the 1st level of multiplexing are modified. The function named **push_up_demux** is used to push the demux nodes to the top of the trees. This modification allows us to group more nodes along with the multipliers in further resource sharing process. Another purpose of this modification is to reduce the number of registers required, which will be explained in the allocation step. A demux node 'L1..' will be placed at the top of the tree, if and only if the full tree can be computed during the ON period of the multiplexing cycle. Otherwise a demux node 'R1..' will be placed at the top of the tree. A demux node 'R1..' should not be a child of a demux node 'L1..' at any time. Otherwise, the node 'L1..' which depends on the value of the node 'R1..' can not be computed correctly. Fig. 2.6b illustrates the effect of **push_up_demux** on a DFG. The modified DFGs of the second order wave digital filter are given in Fig. 2.7a. The modified DFGs are converted into a new SFG as shown in Fig. 2.7b. The effect of multiplexing on the SFG can be observed by comparing Fig. 2.1a and Fig. 2.7b.

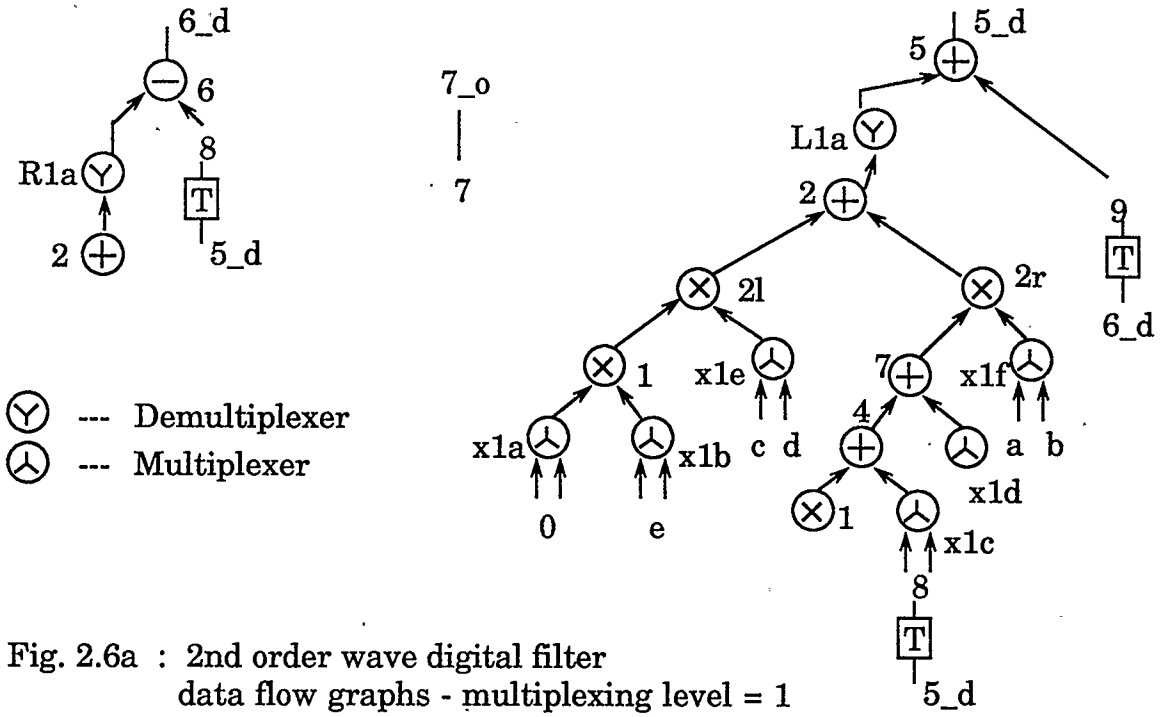


Fig. 2.6a : 2nd order wave digital filter
data flow graphs - multiplexing level = 1

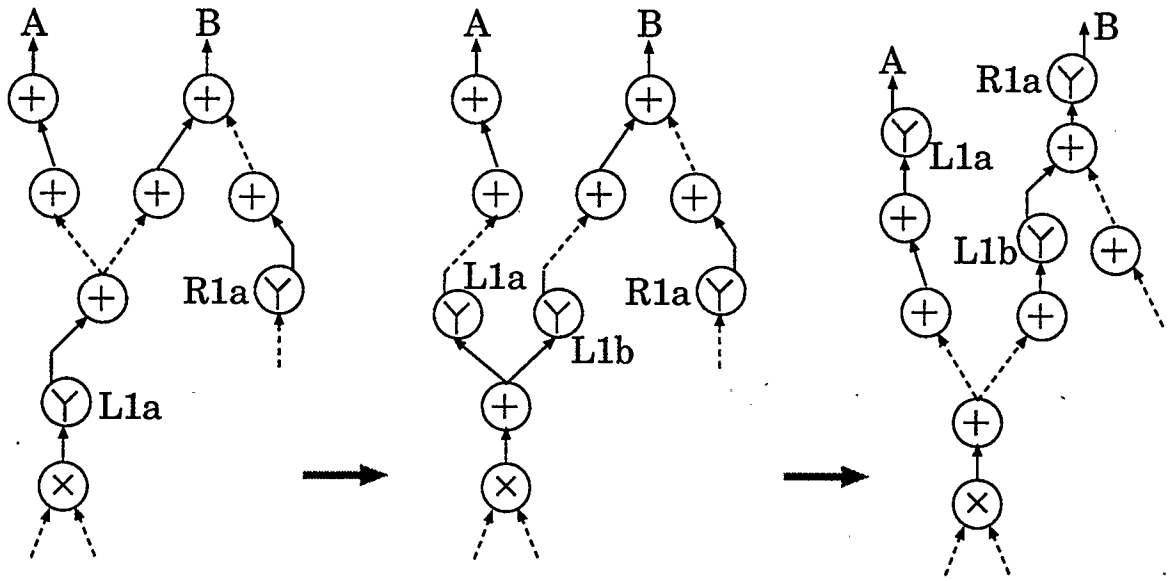


Fig. 2.6b : The effect of function *push_up_demux* on a DFG

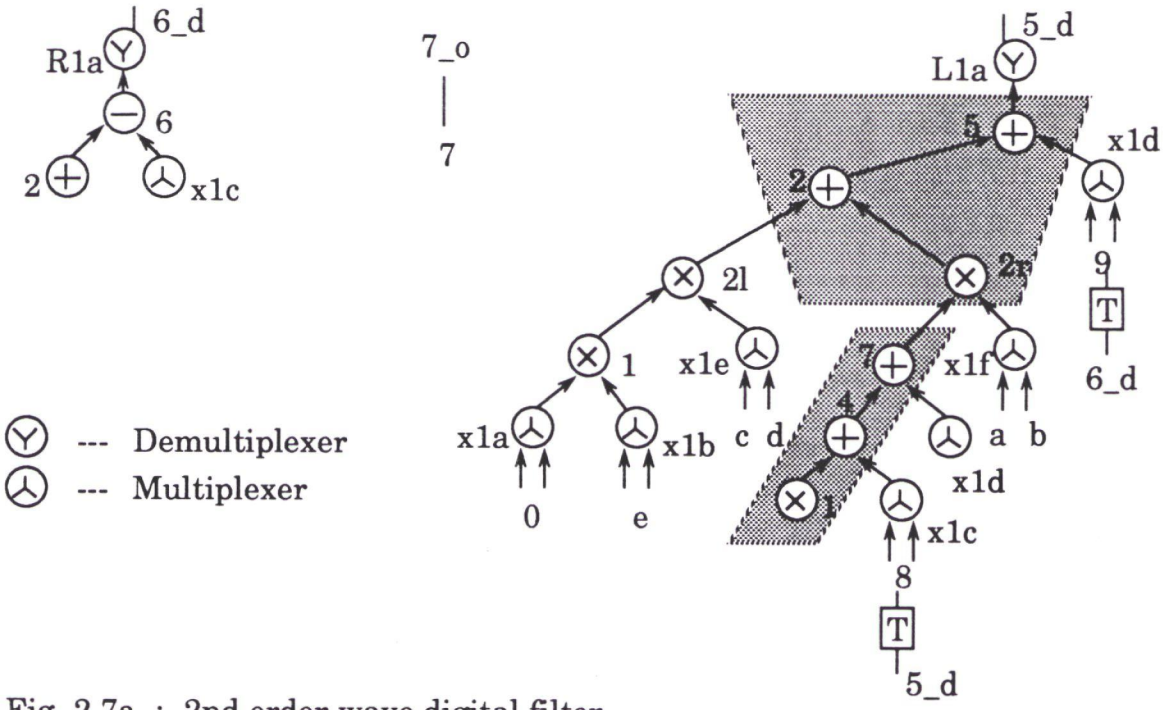


Fig. 2.7a : 2nd order wave digital filter
modified data flow graphs - multiplexing level = 1

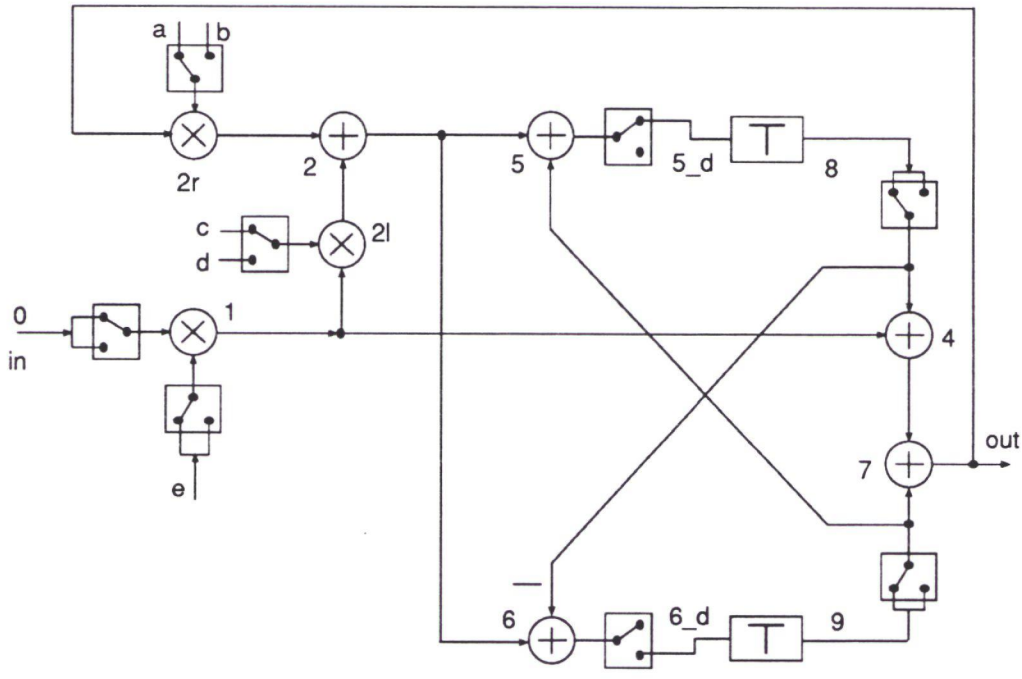


Fig. 2.7b : 2nd order wave digital filter
modified signal flow graph - multiplexing level = 1

There are three multiplier nodes in the DFG obtained after the resource sharing of (2l, 3l). They can form four valid pairs of nodes which can be used for further resource sharing. The resource sharing process continues to reach a minimum number of multipliers provided that the specified constraints (swl, cwl, sample rate) are not violated. For the pair (1, 2r) in Fig. 2.7a, the sets S1 = {1,4,7} and S2 = {2r,2,5} can be multiplexed in the second level of multiplexing. The resultant DFGs are shown in Fig. 2.8a and the resultant SFG is given in Fig. 2.8b. The nodes which could be shared in the third level of multiplexing are shaded in Fig. 2.8a.

2.2.2 Concurrent Multiplexing

It is also possible to multiplex two or more pairs of multiplier nodes in the same level of multiplexing. This phenomenon will be called concurrent multiplexing. Two or more pairs of nodes can be multiplexed simultaneously if they are mutually exclusive. Consider two pairs of multiplier nodes (P, Q) and (R, S) for concurrent multiplexing. The two pairs are selected such that the nodes P, R can perform the operations during the ON period of the multiplexing cycle and the nodes Q, S can perform the operations during the OFF period. The two pairs are not mutually exclusive if any of the following four conditions are true

(a) If S is a child of P

(b) If Q is a child of R

(c) If P is a child of R and S is a child of Q

(d) if R is a child of P and Q is a child of S

The above conditions will apply to all the nodes which can be multiplexed along with each

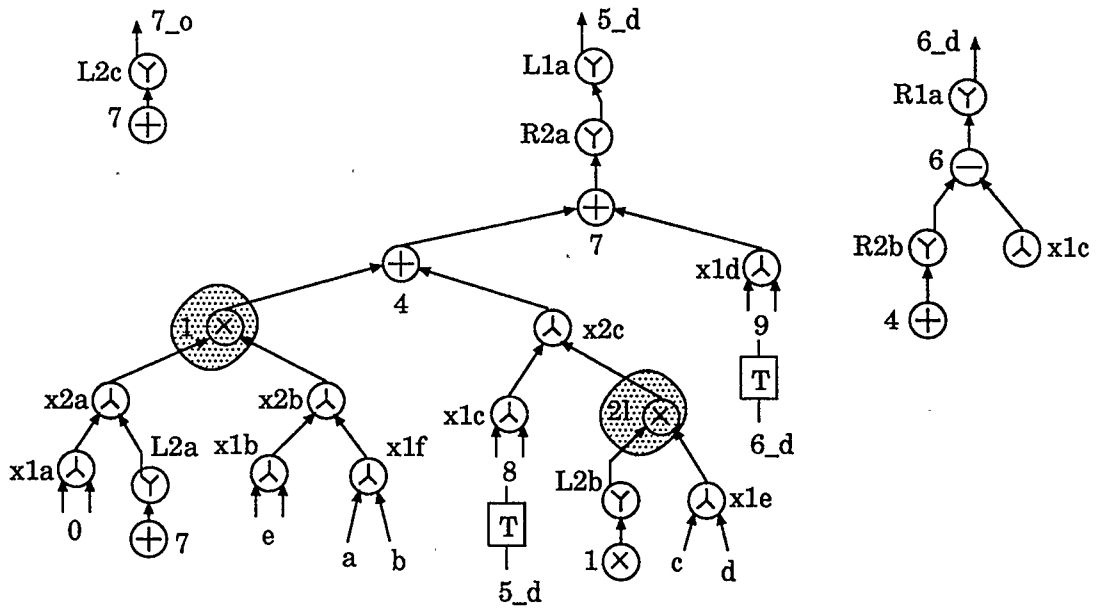


Fig. 2.8a : 2nd order wave digital filter
data flow graphs - multiplexing level = 2

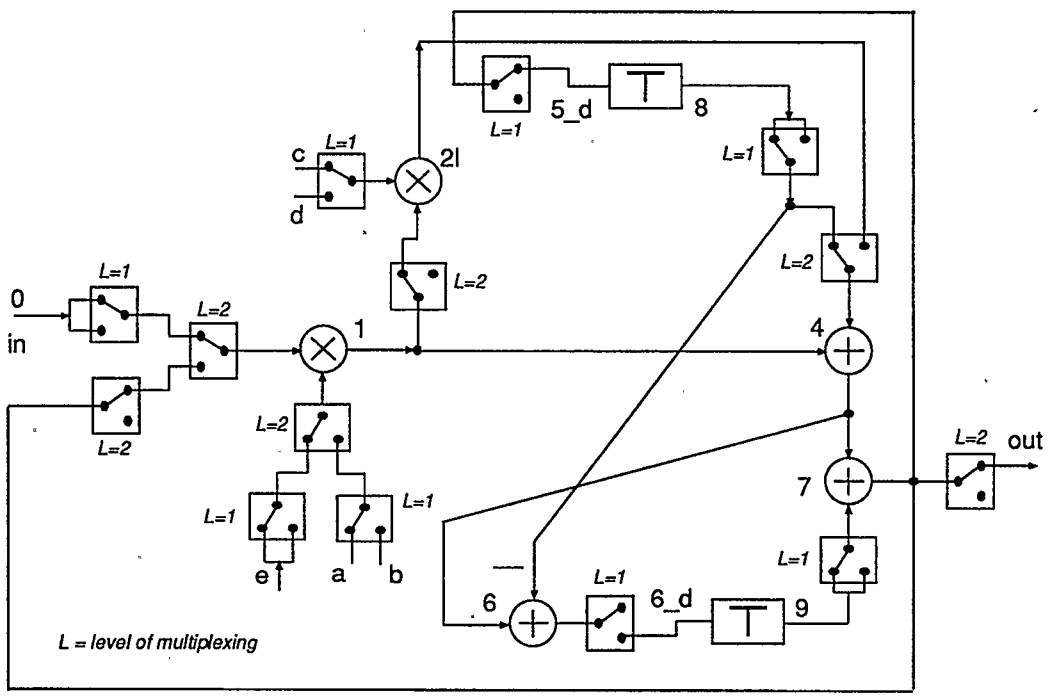


Fig. 2.8b : 2nd order wave digital filter
signal flow graph - multiplexing level = 2

multiplier. Assume that S1, S2, S3, S4 are the sets of nodes which can be multiplexed along with the multiplier nodes P, Q, R, S. Then the condition (a) is modified as

If a node of 'S4' is a child of any node of 'S1' then the two pairs of sets are not mutually exclusive

and is illustrated in Fig. 2.9(a). Fig. 2.9 shows graphically the four different conditions in which the pairs of sets are not exclusive and cannot be multiplexed together. Fig. 2.9(c) illustrates that the two pairs (S1, S2) and (S3, S4) cannot be shared concurrently if a node of 'S1' is a child of any node of 'S3' and a node of 'S4' is a child of any node of 'S2'. Once a pair of multiplier nodes is resource shared, all other pairs from the list of pairs are considered for concurrent multiplexing (CM). The resource sharing algorithm uses all the four conditions to eliminate the pairs which are not mutually exclusive. All the identical nodes around each multiplier must be selected such that they do not violate the mutually exclusive conditions. In the DFGs shown in Fig. 2.2b, the pairs (2l, 2r) and (3l, 3r) are mutually exclusive and they can be concurrently multiplexed.

Sometimes the demux nodes 'Ln...' created in the earlier stages of CM, will become invalid during the later stages of CM. If a demux_node 'Ln...' is equal to left child or child nodes of left child of a multiplexer node 'xn..' then the demux node becomes invalid and it should be removed from the graph. Whenever a demux node is deleted, its immediate child node will be connected to its previous nodes. After concurrent multiplexing is completed, all the invalid demux nodes are removed using the module called *rm_demux_nodes*.

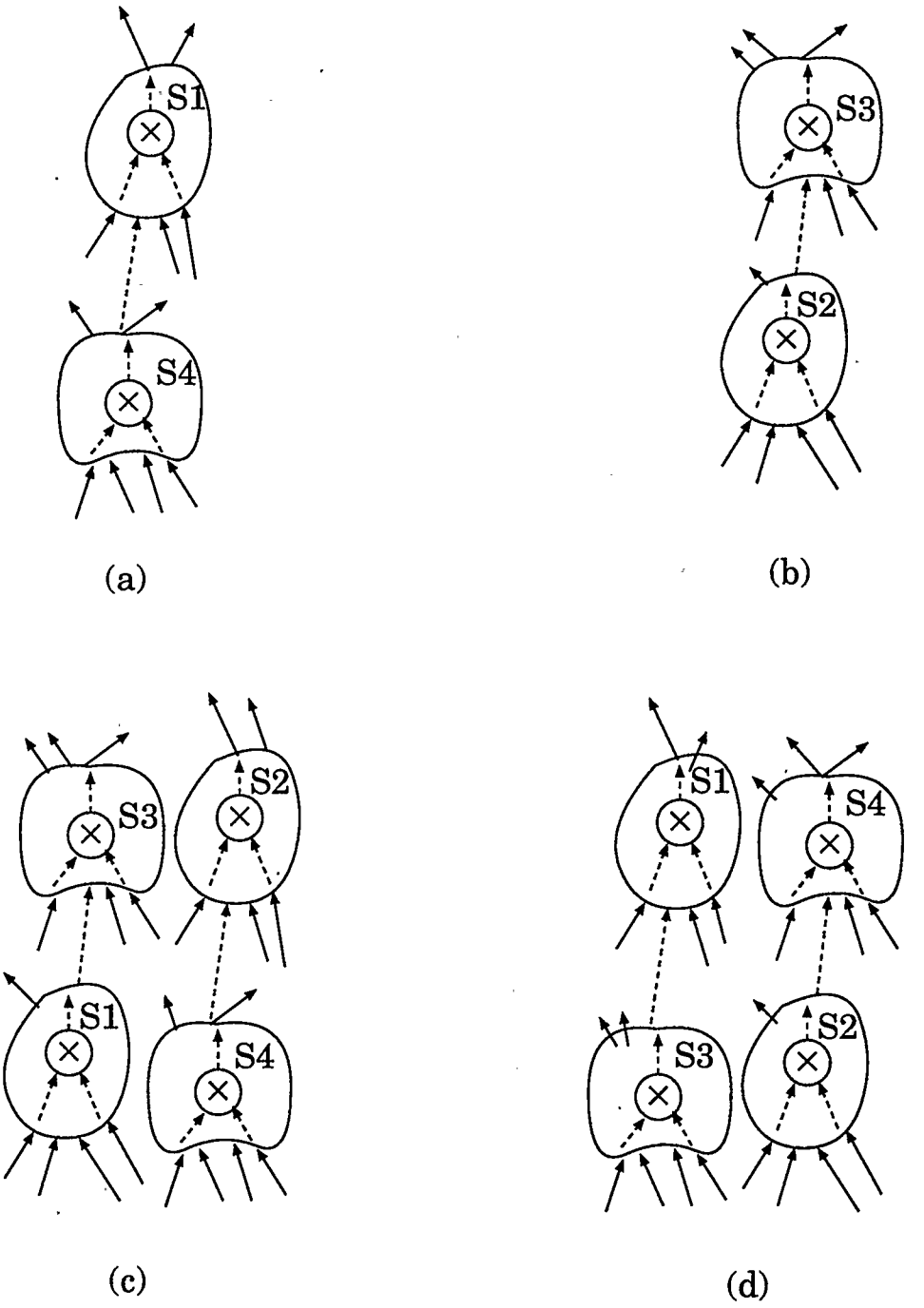


Fig. 2.9 : Four different situations where the two pairs (S1, S2) and (S3, S4) can not be shared concurrently

2.3 Scheduling

Scheduling is the process of determining the time at which each operation occurs. In order for the circuit to operate correctly, the inputs to each operator should be ready by the time they are needed. The timing information found through this process is used to calculate the system latency which is the lower bound on the *swl* in recursive structures. The scheduling process does not change the DFGs. Three different times are computed for each node during the scheduling process. The three times are T_{in}, the time at which node inputs are required, T_{out}, the earliest time at which node outputs are available and T_{needed}, the time at which node outputs are needed. If a node output is connected to many nodes, there may be more than one T_{needed} time for that node. These times are calculated in two phases called forward or as soon as possible (ASAP) scheduling and backward or as late as possible (ALAP) scheduling. The ASAP scheduling calculates the earliest time at which each operation in the DFG can be executed. The ALAP scheduling calculates the latest time at which each operation has to be started.

2.3.1 Forward Scheduling (ASAP)

In the forward scheduling, all the times for each operator are calculated in topological order (bottom to top) by using the precedence relations specified by data dependency. All the inputs at the bottom of the DFGs are initially assigned to have a T_{out} at time zero. In the case of multiplexed structures, the time T_{out} of all the demultiplexer nodes are initially set to zero. The times of the demultiplexer nodes will be modified during the scheduling process. The times are computed by traversing from bottom to top of the graphs such that

for each node :

T_{in} = max { T_{out} (left child), T_{out} (right child) }

T_{out} = T_{in} + latency of the operator

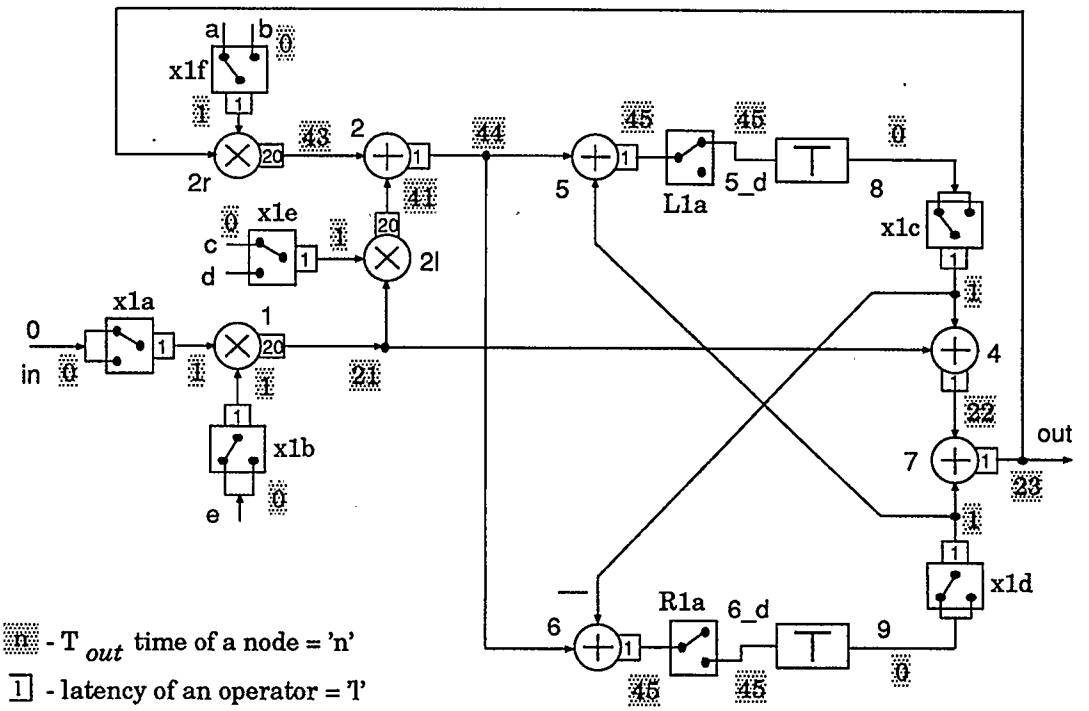
T_{needed} = T_{out}

The demultiplexer (demux) nodes are not the real physical operators. If any closed paths are created due to the multiplexing, the *demux* nodes in that paths are used to break the loop by disconnecting the *demux* node and its previous node or nodes. The T_{needed} time calculated at this stage is not equal to the earliest time at which the output of the node is needed. The T_{needed} time will be modified in backward scheduling.

The longest path length can be found after the forward scheduling. The longest path is equal to the path from a node at the bottom of the DFGs to a node at the top of the DFGs that has maximum delay. If the node at the bottom of the DFGs is a state output (current state) and the node at the top of the DFGs is the corresponding state input (next state), then the longest path is called critical path. The system latency is equal to the critical path length. The backward scheduling is used to find the actual T_{out} times at the bottom of the DFGs. The forward schedule for the second order wave filter after 1st level of multiplexing is given in Fig. 2.10a.

2.3.2 Backward Scheduling (ALAP)

During the backward scheduling the times T_{in}, T_{out} and T_{needed} are updated by traversing from top to the bottom of the graphs. Each node may have more than one T_{needed} if that node is a child of more than one node. The times are modified for each node



node	T_{in}	T_{out}	T_{needed}	child nodes	node	T_{in}	T_{out}	T_{needed}	child nodes
0	-	0	0	-	x1f	0	1	1	a, b
x1a	0	1	1	0	2r	23	43	43	x1f, 7
e	-	0	0	-	c	-	0	0	-
x1b	0	1	1	e	d	-	0	0	-
1	1	21	21	x1a, x1b	x1e	0	1	1	c, d
8	-	0	0	-	2l	21	41	41	x1e, 1
x1c	0	1	1	8	2	43	44	44	2r, 2l
4	21	22	22	x1c, 4	5	44	45	45	2, x1d
9	-	0	0	-	L1a	45	45	45	5
x1d	0	1	1	9	5_d	45	45	-	L1a
7	22	23	23	x1d, 4	6	44	45	45	2, x1c
a	-	0	0	-	R1a	45	45	45	6
b	-	0	0	-	6_d	45	45	-	R1a

Fig. 2.10a : Forward schedule for the 2nd order wave digital filter
- 1st level multiplexing

as follows :

T_{out} = minimum { $T_{needed1}$, $T_{needed2}$, ... }

T_{in} = T_{out} - latency of the operator.

T_{needed} (left child) = T_{needed} (right node) = T_{in}

The second phase of calculations ensures that all the nodes at the bottom of the graphs have correct T_{out} times with respect to the T_{out} times at the top of the graphs. All the *demux* nodes also will have the proper T_{out} times. The times in the longest path or paths do not change during backward scheduling. Such a path or paths represent the critical path or paths. Now the system latency can be calculated accurately as the maximum of the difference between T_{out} times of state inputs to the T_{out} times of the corresponding state outputs. In the case of multiplexed structures if closed loops are created, then the system latency should not be less than the closed loop length. Fig. 2.10b gives the backward scheduling for the second order wave filter which is multiplexed at the 1st level. In Fig. 2.10b,

the system latency = maximum{(T_{out}(6) - T_{out}(9)) , (T_{out}(5) - T_{out}(8)}

= maximum{24, 25} = 25

Since the filter is a recursive structure, the minimum *swl* is equal to the system latency.

The minimum of the T_{out} times of state outputs at the bottom of the graphs is called *base_time*. If *base_time* is not equal to zero, it can be adjusted to zero. The reason for modifying the timings is discussed in the delay reduction section. All other times will be modified with respect to the *base_time*. The modified schedule is shown in Fig. 2.10c. This procedure is used to reduce the number of delay units required. The negative times indicate

node	T_{in}	T_{out}	T_{needed}	child nodes	node	T_{in}	T_{out}	T_{needed}	child nodes
6_d	45	45	-	R1a	b	-	22	22	-
R1a	45	45	45	6	a	-	22	22	-
6	44	45	45	2, x1c	7	22	23	23	x1d, 4
5_d	45	45	-	L1a	x1d	21	22	22, 44	9
L1a	45	45	45	5	9	-	21	21	-
5	44	45	45	2, x1d	4	21	22	22	x1c, 1
2	43	44	44	2r, 2l	x1c	20	21	21, 44	8
2l	23	43	43	x1e, 1	8	-	20	20	-
x1e	22	23	23	c, d	1	1	21	21, 23	x1a, x1b
d	-	22	22	-	x1b	0	1	1	e
c	-	22	22	-	e	-	0	0	-
2r	23	43	43	x1f, 7	x1a	0	1	1	a
x1f	22	23	23	a, b	a	-	0	0	-

Fig. 2.10b : Backward schedule for the 2nd order wave digital filter
- 1st level multiplexing

node	T_{in}	T_{out}	T_{needed}	child nodes	node	T_{in}	T_{out}	T_{needed}	child nodes
6_d	25	25	-	R1a	b	-	2	2	-
R1a	25	25	25	6	a	-	2	2	-
6	24	25	25	2, x1c	7	2	3	3	x1d, 4
5_d	25	25	-	L1a	x1d	1	2	2, 24	9
L1a	25	25	25	5	9	-	1	1	-
5	24	25	25	2, x1d	4	1	2	2	x1c, 1
2	23	24	24	2r, 2l	x1c	0	21	1, 24	8
2l	3	23	23	x1e, 1	8	-	0	0	-
x1e	2	3	3	c, d	1	-6	1	1, 3	x1a, x1b
d	-	2	2	-	x1b	-5	-6	-6	e
c	-	2	2	-	e	-	-5	-5	-
2r	3	23	23	x1f, 7	x1a	-5	-6	-6	a
x1f	2	3	3	a, b	a	-	-5	-5	-

Fig. 2.10c : Final schedule for the 2nd order wave digital filter
- 1st level multiplexing

that the operations take place at times equal to the absolute value of the negative times, but one cycle ahead. For example in Fig. 2.10c, the inputs to the multiplier node '1' are required at minus 6 (at time 6, one cycle ahead). Then, the output of the node '1' is available at 26 (latency of the multiplier = 20). Since the swl is equal to 25, the output of the node is available at time '1' in the current cycle of operation.

2.4 Allocation

The registers or delay units are allocated during this step. All other operators are allocated by replacing the nodes in DFGs with physical bit-serial operators. In a pipelined bit-serial system, registers or delay elements are used to synchronize the arrival of the inputs if there is a mismatch in the timing of the input signals at any operator. If a node has different T_{out} and T_{needed} times then the number of one bit registers needed is equal to the difference of the two times.

When resources are multiplexed, the results computed in the first half period of the multiplexing cycle must be stored or delayed until the other results are computed in the second half period. The conceptual demultiplexers are used for this purpose. The demux nodes with names 'Ln..' are replaced by an appropriate number of one bit registers. The demux nodes with names 'Rn..' are ignored because the input signals to these nodes are valid during the second half period of multiplexing control signal and need no delay elements. In the case of the **n** th level demultiplexer, the number of delay elements required is equal to the number of bit times in the first half period or 2^{n-1} * swl.

All the inputs and state outputs are updated once in each sample period. Circulating buffers are used to store the input and state output data because the data may be required throughout the period (ON and OFF periods) when the resources are multiplexed. These buffers will be removed if they are not required. The delays can be reduced in hardware by combining several delays into one.

2.4.1 Delay Reduction

If a number of delays have to be connected to a single signal, then the delays should be connected sequentially and not in parallel [24]. The sequential connection requires fewer delays. This principle is illustrated in Fig. 2.11a where different delays are connected to a node, x. The total number of delays that need to be connected to any signal is equal to the maximum required delay.

If an input or a state output has a non-zero T_{out} time, then that input or state output can be tapped at an appropriate delay from the circulated buffers without using extra delay elements. During the backward scheduling, the times are adjusted such that the nodes at the bottom of the DFGs have non-zero T_{out} times and all other nodes have the same T_{out} and T_{needed} times. If a node has more than one T_{needed} then the minimum of the T_{needed} times is equal to the T_{out} time. The delay reduction using circulated buffers is illustrated in Fig. 2.11b.

Another delay reduction is at the state output and state input interface. If a state input at the top of the DFGs has a 1st level demux node 'L1..', then the state input signal has to be delayed by *swl* before updating the corresponding state output. Thus the delays

associated with a state input can be merged with the circulated buffer delays associated with a state output. For example in Fig. 2.7a, the demux node 'L1a' at the top of the DFG can be merged with the circulated buffer 'x1c'. For **n** level multiplexing, the number of cycles required to update the new states is equal to 2^n where the period of each cycle is equal to *swl*. The function *push_up_demux* is used to push the demux nodes of 1st level multiplexing to the top of the tree so that the necessary delays can be merged with the circulated buffer delays. This principle is illustrated in Fig. 2.11c. A typical circuit before merging the delays is given in Fig. 2.11c(i). The following pseudocode describes the rules for merging the delays with the circulated buffers.

```
if (m >= swl)
    then merging is not necessary because the delay δ(swl) will
    be part of the delay δm.
else {
    if state output is not used during 'OFF' period of the
    1st level multiplexing cycle
        then merging is possible and the resultant circuit
        is as shown in Fig. 2.11c(ii).
    else {
        if ((m + n) < swl)
            then merging is possible and the resultant
            circuit is as shown in Fig. 2.11c(iii).
        else
            no merging is necessary and the δm will
            be part of the delay δ(swl).
    }
}
```

In all the circuits in Fig. 2.11c, *c10* represents the control signal of a level one multiplexer, *c20* represents the control signal of a level two multiplexer and so on. In the Fig. 2.11c, the control *cs0* is same as *cs1* but delayed by one word time or one *swl*. The different types of control signals and how they are related to *swl* are discussed in the next section.

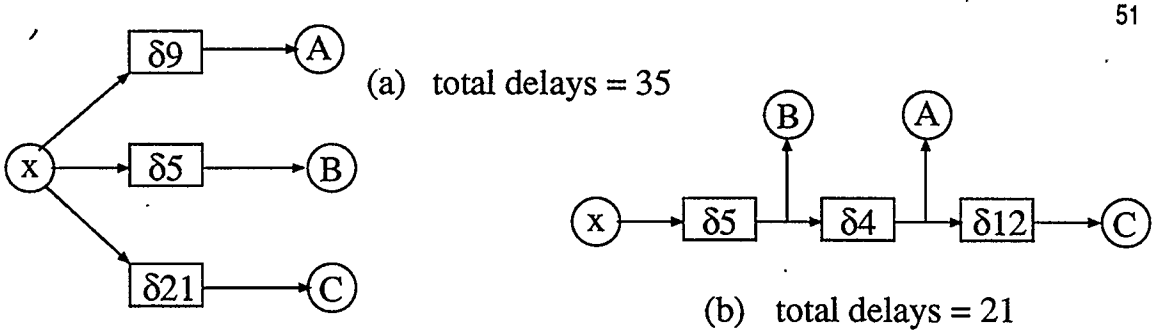


Fig. 2.11a : Comparison of parallel and serial delay implementation

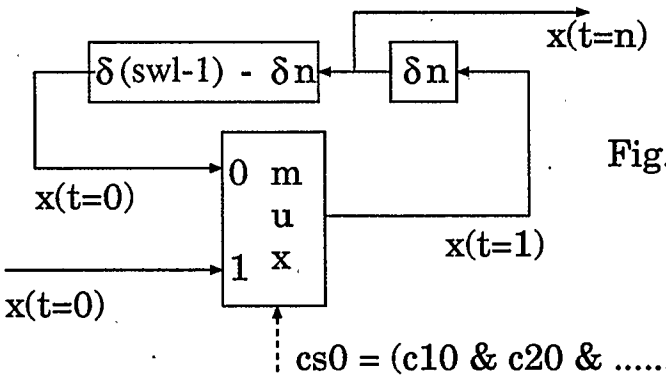


Fig. 2.11b : A typical circulated buffer

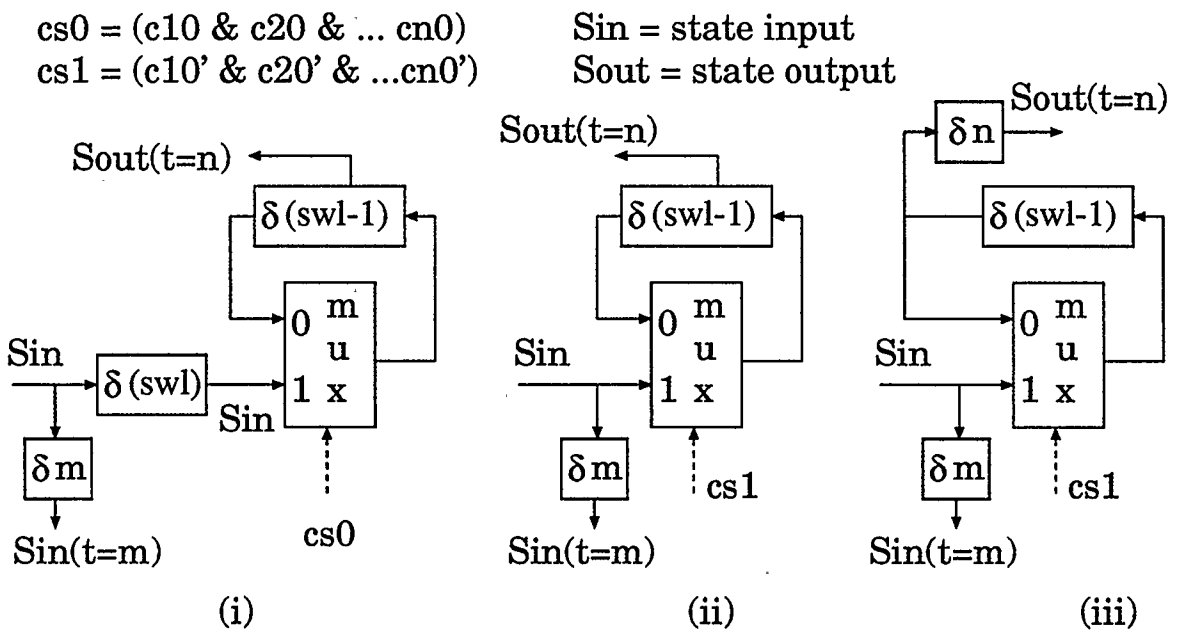


Fig. 2.11c : Delay minimization at the state circulated buffers

All the operators except multiplexers, require the control signal *c00* (LSB time) to identify the arrival of the new word. The *c00* should be given as the control input to each operator at time T_{in}. In the example given in Fig. 2.10b, the *c00* is required at times 1, 21, 22, 23, 43, 44. Thus 44 single bit registers are needed to generate *c00* signal at different times. To reduce the number of delays, the times are modified as shown in Fig. 2.10c. Now the *c00* is required at times 0, 3, 5, 6, 23, 24. Hence, only 24 single bit registers are needed instead of 44. If the *base_time* (minimum of the T_{out} times of state outputs) is greater than half the *swl* then the original schedule should be modified with respect to zero base_time.

Each state input should have a clear multiplexer at its output which can be used to set the state values to zero. The control signal *clear0* which is synchronized with time 0 is used to control the multiplexer. The *clear0* should be given to the clear *mux* at time T_{in} of the state input. In the example given in Fig. 2.10b, the *clear0* is required at time 45. After modifying the times, the *clear0* is required at time 25. Whenever clear *mux* is used, the minimum *swl* is equal to the system latency plus one. Thus in this example the minimum *swl* is equal to 26.

The modified SFG of the second order wave filter after the allocation step is given in Fig. 2.12. In the example given in Fig. 2.12, the *swl* is equal to the minimum *swl* limit. After allocation step, the resulting circuit will be defined in the FIRST [3] language.

We have discussed so far, the basic steps involved in the synthesis process of bit-serial systems. The next step is to find the control signals required for a designed structure. Control signals are used to control the timing or the flow of bits through a network.

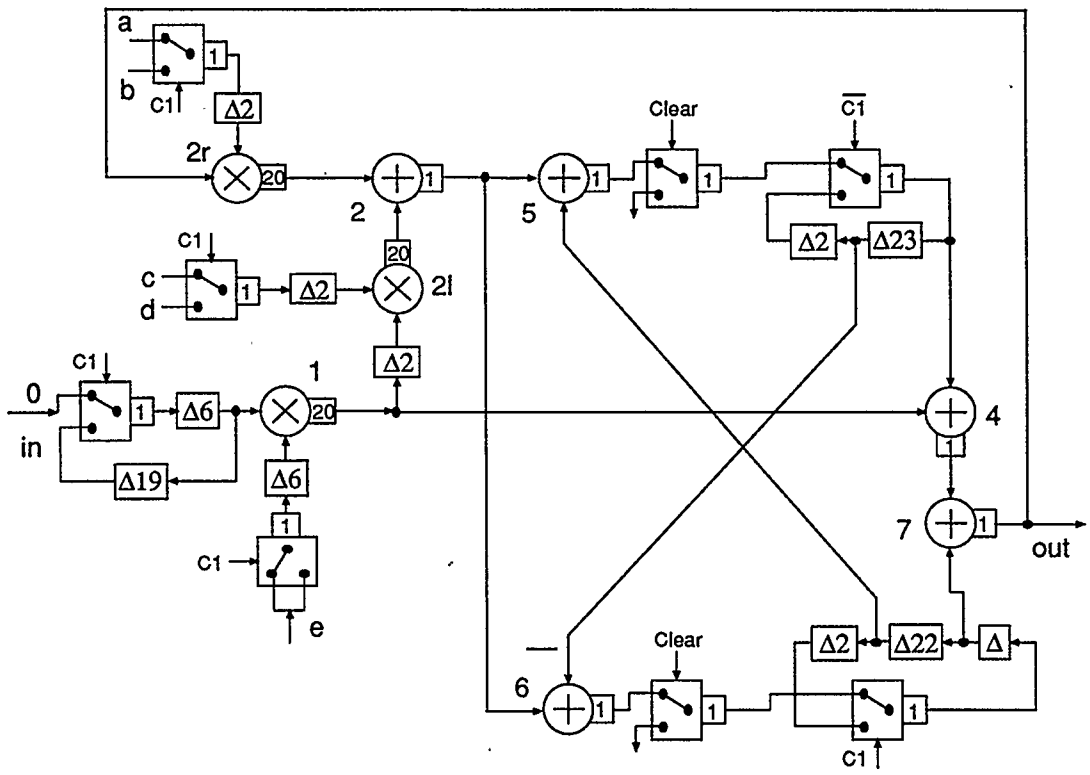


Fig. 2.12 : 2nd order wave digital filter
signal flow graph - after delay allocation

2.5 Control Signals

The control signal used in most of the bit-serial operators is the *LSB time* (c00) and its delayed versions. The *c00* signal is true for the first bit of a word and false for the remaining period. The time period of *c00* is equal to the product of swl and the bit clock period or is equal to swl in bits. This signal is used to define the arrival of a new word. The remaining required signals are the control signals to the multiplexers. The multiplexer of level 1 is controlled by control signal *c10*. The ON and OFF periods of *c10* is equal to the time period of *c00*. That is the *c10* is the same as *c00* divided by 2. Similarly the control signal *c20* used for the multiplexer of level 2, is the same as *c10* divided by 2 or *c00* divided by 4. In general, the control signal *cn0* used for the multiplexer of level **n** is the same as *c00* divided by 2^n.

If **n** level multiplexing is used to design a circuit, the sample period is equal to the product of 2^n and the period of *c00*. Fig. 2.13 illustrates the relation between the system clock, swl and different control signals. Two special control signals *cs0* and *cs1* are needed to update the states and the data input once in a sample period. The control signals *cs0* and *cs1* for 3rd level multiplexing are shown in Fig. 2.13. The control signals will be generated in a module called control generator. The design of control generator is discussed in chapter 3.

The control signal scheme discussed above has some disadvantages. If a design with 5 multipliers can be reduced to a design with 1 multiplier after three levels of multiplexing, then 8 cycles of *c00* are needed to compute one sample. Since the number of multipliers is reduced from 5 to 1, 5 cycles of *c00* may be sufficient to compute one

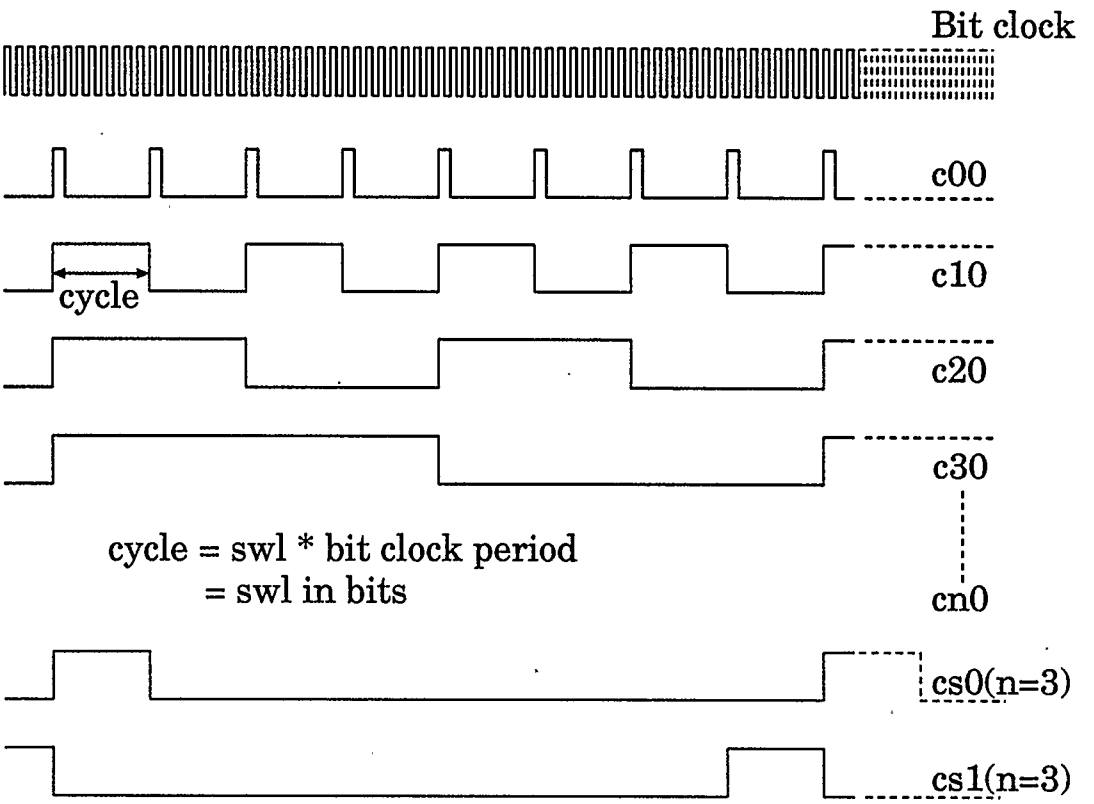


Fig. 2.13 : Control signals for multiplexed architectures

sample. That is some of the 2^n cycles of *c00* (where n is multiplexing level) may be redundant or unused. This timing redundancy can be detected during the resource sharing process and the timing can be improved by modifying the control signals. The number of delay units required to replace demultiplexer node 'Ln...' will be changed if there is a timing redundancy.

2.5.1 Timing Redundancy

When a multiplier node pair is multiplexed in the first level of multiplexing, the two cycles of *c00* are necessary to compute the two multiplier node values. Thus there are no redundant cycles at the first level of multiplexing. Consider a multiplier pair (P, Q) at the second level of multiplexing. Assume that the node P is already multiplexed at the 1st level and the node Q is not multiplexed. That is the node P is used as two multiplier nodes in two cycles. Now if P shares the Q node operation, the node P may perform the Q node operation in one cycle. In that case, three cycles may be needed instead of four to compute a sample. The multiplier pair (P, Q) can not be multiplexed always in three cycles unless P and Q satisfy some conditions. The conditions for the possible improvement in the timing will be explained for a general case taking a multiplier pair (M, N).

Let *n* be the level of multiplexing. Assume that one of the nodes of (M, N) is multiplexed at the level *n-1* and the other at the level *m* (m < n-1). Let 'parent_M' be the top node of the group of nodes that are multiplexed along with M. Similarly let 'parent_N' be the top node of the group of nodes that are multiplexed along with N.

If node 'M' is not a child of node 'parent_N' and node 'N' is not a child of node 'parent_M' then there exists redundant cycles in the control signals. The ON period of the modified control signal *cn0* is equal to the period of control signal associated with node M and the OFF period is equal to the period of control signal associated with node N. If node M is multiplexed at level *n-1* and node N is multiplexed at level **m** then the *cn0* is equal to logical one for the period of *c(n-1)0* and logical zero for the period of *cm0*. During allocation step, the number of delay units required to replace the demultiplexer node 'Ln..' is equal to the period of *cm0* in bits.

If node M is multiplexed at level *m* and node N is multiplexed at level *n-1* then the *cn0* is equal to logical one for the period of *cm0* and logical zero for the period of *c(n-1)0*. During allocation step, the number of delay units required to replace the demultiplexer node 'Ln..' is equal to the period of *c(n-1)0* in bits. In both cases the period of *cn0* is same and is less than (2^n * swl).

The following condition is true only if node 'M' is multiplexed at level 'm' and node 'N' is multiplexed at level 'n-1'

```
If node M is a child of the node 'parent_N' and
node N is not a child of node 'parent_M' {
      if there exists a multiplexer of level n-1 or less
      between node M and 'parent_N'
            then there exists redundant cycles.
      else
            there exists no redundant cycles.
}
```

If there is no multiplexer of level **n-1** or less, it indicates that the node M output is used for the whole time period at *n-1* multiplexing level. Hence there are no redundant cycles. If

node M is a child to the left child of the multiplexer, then the number of delay units required to replace the demultiplexer is equal to period of *cm0* in bits. But if the node M is a child to the right child of the multiplexer, then the number of delay units required to replace the demultiplexer is equal to the period of *c(n-1)0* in bits.

If node M is a not a child of node 'parent_N' and node N is a child of node 'parent_M' then there exists no redundant cycles.

The occurrence of redundant cycles (timing redundancy) can be explained through an example. Consider the 2nd order direct form digital filter whose SFG and data flow graphs are given in Fig. 2.14a and Fig. 2.14b. This filter exhibits timing redundancy at the 3rd level multiplexing.

Fig. 2.14c gives the DFGs after first level of multiplexing. Here, the multiplier pairs (5r, 1r) and (5l, 6l) are multiplexed concurrently. The control signal *c10* is represented as cyclic sequence '10'. The sequence '10' means logical one and logical zero each for one period of 'swl' in bits.

Fig. 2.14d shows the DFGs after 2nd level of multiplexing where the multiplier pair (5r, 5l) is multiplexed. There are no redundant cycles because both the multiplier nodes *5r* and *5l* are multiplexed at 1st level. Thus there is no redundant cycles at the 2nd level. Fig. 2.14e shows the DFGs after 3rd level of multiplexing where the multiplier pair (5r, 6r) is multiplexed. There exists timing redundancy. The multiplier nodes *5r* and *6r* are not a child of one another. Thus there exists three redundant cycles. All the control signals 'c10', 'c20' and 'c30' are given as five digit cyclic sequences. Thus in this example an initial design with 5 multipliers has been converted to a design with 1 multiplier in 5 cycles.

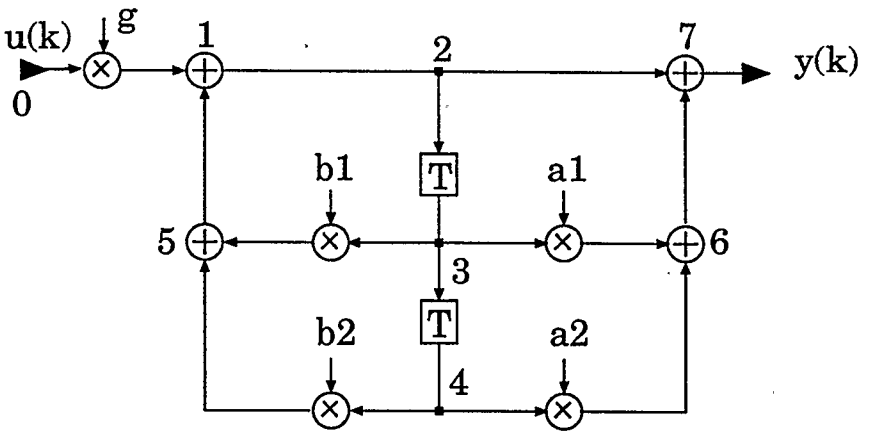


Fig. 2.14a : Signal flow graph of 2nd order direct form digital filter

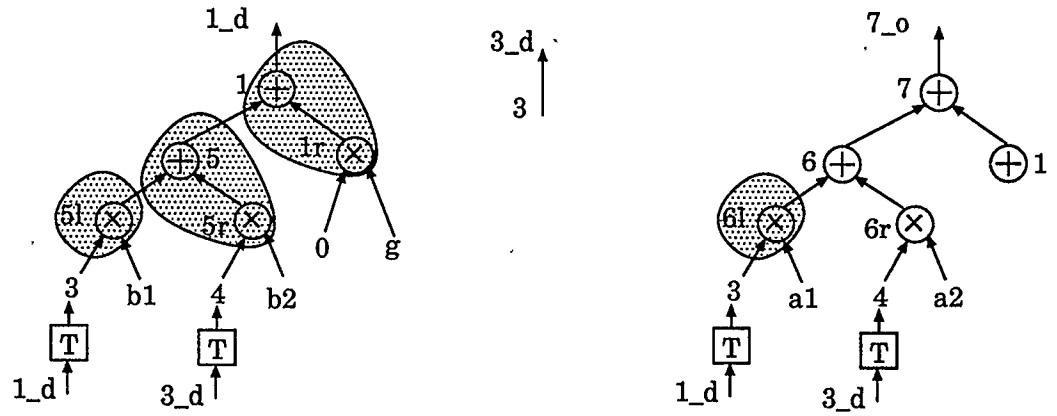


Fig. 2.14b : Data flow graphs of 2nd order direct form digital filter
- no multiplexing

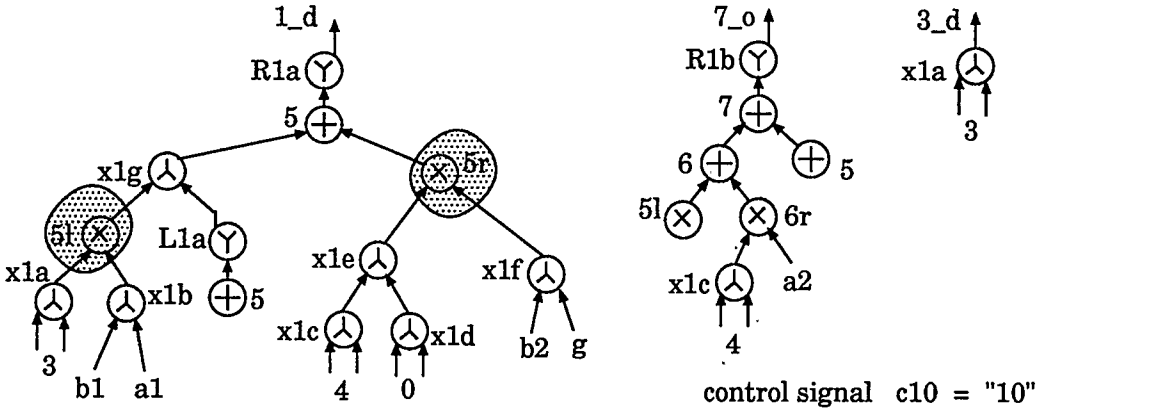


Fig. 2.14c : Data flow graphs of 2nd order direct form digital filter
- 1st level multiplexing

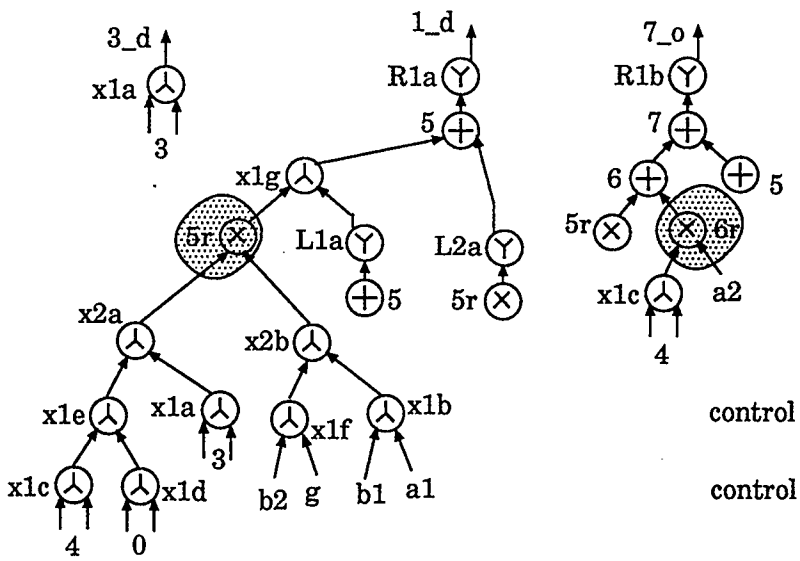


Fig. 2.14d : Data flow graphs of 2nd order direct form digital filter - 2nd level multiplexing

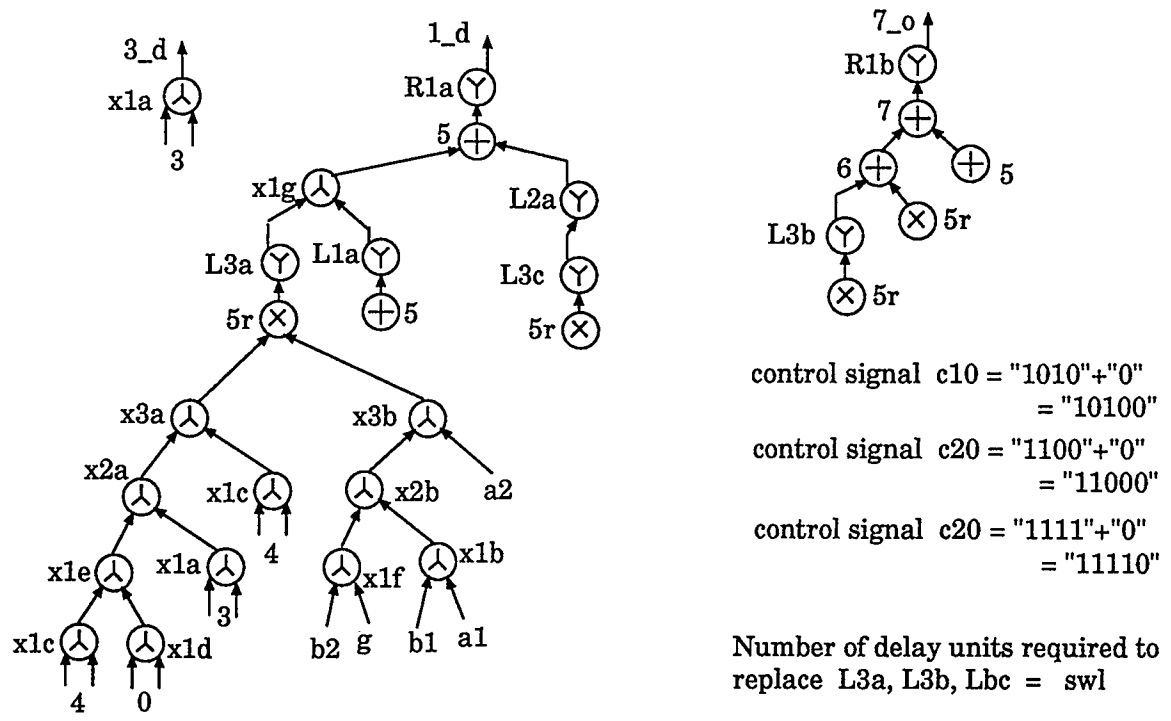


Fig. 2.14e : Data flow graphs of 2nd order direct form digital filter - 3rd level multiplexing

The control signals are represented as finite cyclic sequences when timing redundancy exists. These signals can be generated using a finite state machine (FSM) or linear feedback shift registers (LFSR). Both of these methods are discussed in detail in chapter 4.

2.6 Summary

The basic synthesis steps such as translation, resource sharing, scheduling, and allocation discussed above generate a design solution for a given high level behavioral specification. During the translation step, the given behavioral input in the form of a SFG is converted into an internal graphical representation called data flow graphs (DFG). The DFGs are implemented using binary tree structures in such away that any node in the DFGs can be accessed from any other node by traversing upwards or downwards. The resource sharing process multiplexes the multiplier nodes because the bit-serial multiplier is normally the largest operator in size. If possible, two or more pairs of multiplier nodes are multiplexed concurrently in the same level of multiplexing. The resultant DFGs are scheduled in two different phases. The minimum swl is calculated from the scheduled DFGs. The delay registers are allocated appropriately. Once the delay allocation step is finished, the DFGs with appropriate control signals represent a circuit which can implement the given behavioral specification. A number of design solutions can be obtained for the same behavioral specification by choosing different multiplier pairs and their order of multiplexing in the resource sharing step. The BITSYN algorithm based on the exhaustive search of the solutions is explained in the following chapter.

Chapter 3

BITSYN ALGORITHM

This chapter describes the implementation of the bit-serial synthesis (BITSYN) algorithm. It gives an overview of the design space and the heuristics used to speed up the synthesis process. The BITSYN algorithm evaluates different resource sharing (multiplexing) strategies and generates a minimum gate design, a high sample rate design and a number of designs between these limits. The algorithm exhaustively searches the possible designs that would result due to the multiplexing. BITSYN accepts a signal flow graph (SFG) behavioral description as input and generates the designs in the FIRST [3] netlist format.

3.1 BITSYN Algorithm

The BITSYN algorithm incorporates the major synthesis steps such as translation, resource sharing, scheduling and allocation discussed in chapter 2 to generate multiplexed designs. The implementation of the BITSYN algorithm is outlined as follows :

1. Read the SFG description and the input specifications (e.g : system word length and coefficient word length)
2. Build the precedence graphs or DFGs. (next state and output calculations)
3. Prepare a list of the possible pairs of multiplier nodes. Set the multiplexing level to 1.

4. For each multiplier node pair {

   4a. **Perform the resource sharing operation.** That is, find two groups of nodes which create similar patterns around each multiplier and introduce multiplexers and conceptual demultiplexers into the DFG. Remove the nodes from the DFGs whose operations are shared.

   4b. **Schedule the resultant DFG's and allocate the registers.** Analyze the design structure and find the design characteristics such as number of equivalent gates, system word length (swl) and normalized sample frequency.

   (a) The number of gates is equal to the sum of the gates for each and every operator needed to implement the design in a gate array process.

   (b) swl = maximum of {system latency, required swl}

   (c) If there are no redundant cycles among 2^n (where n is multiplexing level) cycles of c00 (LSB time) then normalized sample frequency = $1/(2^n * swl)$

   If some of the 2^n (where n is multiplexing level) cycles of c00 are redundant then normalized sample frequency = 1/(period of cn0 in bits * swl)

   4c. Store the design. Modify the list of node pairs by removing the pairs containing the shared nodes.

   4d. If the list of multiplier node pairs is not empty then increment the level of multiplexing, go to step 4 and repeat the recursive procedure.

   }

5. Display the designs graphically. Select a design that satisfies the user requirements and generate the design in the FIRST language.

The implementation of the BITSYN algorithm is discussed in detail in appendix A.

3.2 Design Space

BITSYN generates pipelined bit-serial designs in a three dimensional space. The three dimensions are normalized sample frequency (nsf), gate count and the system latency. The normalized sample frequency is defined as the ratio of the sample frequency to the system bit clock frequency. The system latency is the lower bound on the *swl* in recursive structures . In non recursive structures, feedback paths may be created due to multiplexing. In such a case, the system latency is the lower bound on the *swl*. A large memory space is needed to save the designs in FIRST netlist format. Instead to save memory, all the required transformations, that is the multiplier pairs, the order of multiplexing, and the design characteristics are saved. The FIRST netlist of a selected design can be regenerated when required.

It is not necessary to save all the designs generated by BITSYN. If two designs have the same *nsf* and system latency, then the solution with fewer gates is saved. If they have the same sample frequency and different system latencies then both the solutions are saved. Thus a smaller set of designs are saved such that each design at a particular *nsf* has minimum number of gates. A solution is useful (optimal) if there is no other solution at a higher sample frequency having fewer gates. This is illustrated using the design space of the second order wave digital filter (SFG is given in Fig. 2.1a) in Fig. 3.1. All the solutions in the design space may not be useful (optimal).

If the required *swl* of a design is greater than the system latency, then extra delay units are inserted in all the closed loop paths of the design. Since a minimum gate design is saved for each system latency, a design with any required *swl* can be generated from

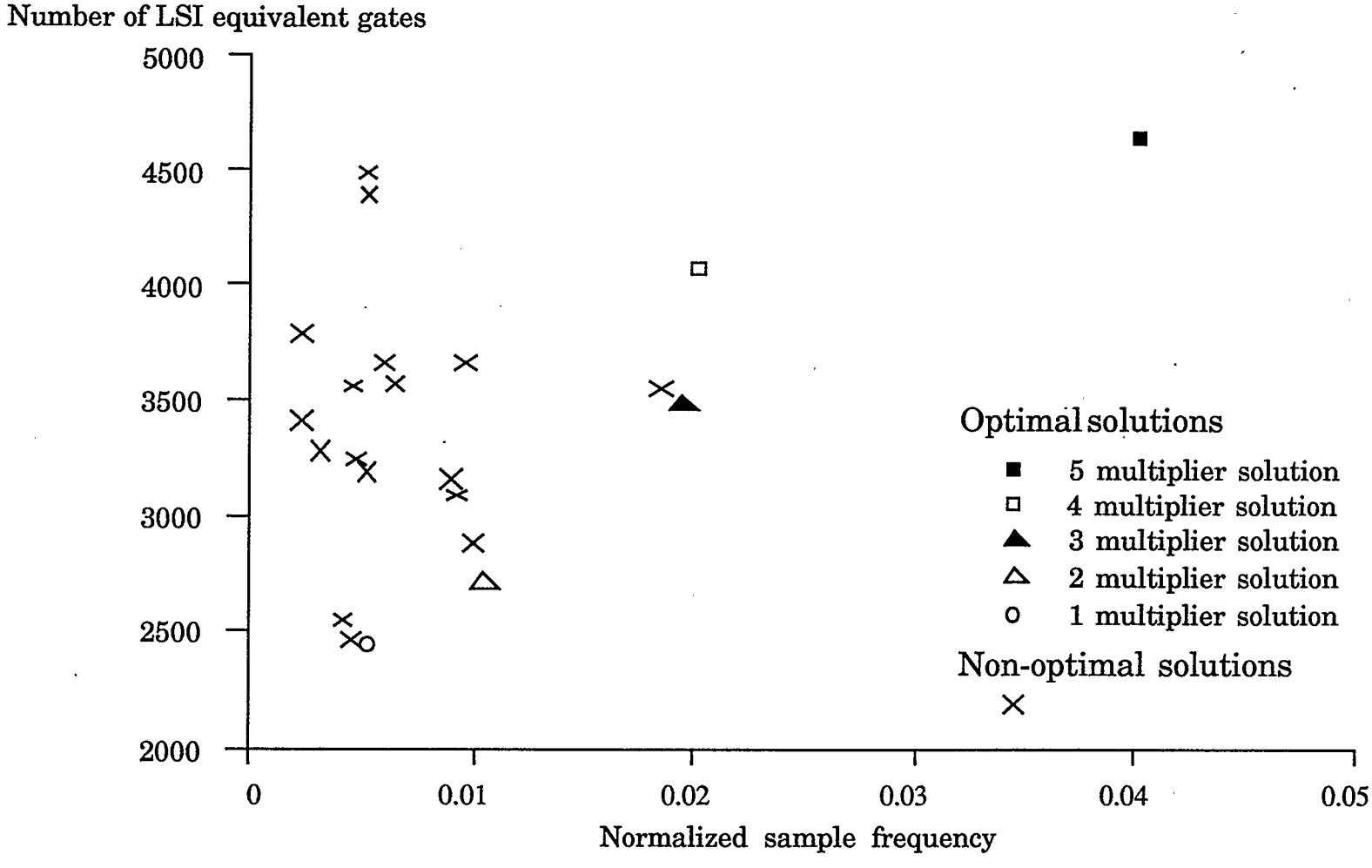


Fig. 3.1 : BITSYN design space for 2nd order wave digital filter

the saved designs by inserting necessary delay units. Thus the user can change the *swl* at the end of the synthesis process by recomputing the minimum gate designs with the new required *swl*. The recomputed designs will be same as if the designs had been generated by synthesizing with the new *swl*. If the *cwl* specification is changed, the recomputed minimum gate designs may not be same as the designs that can be generated by synthesizing with the new *cwl*. But it has been observed in a number of examples (given in chapter 5) that the recomputed designs are same or close to the synthesized designs.

3.3 Heuristics

BITSYN produces a huge number of designs at different levels of multiplexing. The total number of design solutions increases exponentially with the number of multiplier nodes. Let m be the number of multiplier nodes in the initial DFGs. There will be $m*(m-1)$ pairs that can produce multiplexed designs at the first level (assuming that all pairs are valid). Each of the resultant designs can produce $p*(p-1)$ pairs where p is the number of multiplier nodes in the multiplexed design. Thus each of the designs at the first level can produce $p*(p-1)$ designs or multiplexing paths at the second level (assuming all the pairs are valid). This process continues until the designs with a fewest number of multiplier nodes are achieved.

Five heuristic rules are used to reduce the number of design solutions and reduce the computation time. These heuristic rules prune the multiplexing paths which are not likely to produce the useful solutions. The heuristics have been tested using a number of design examples (given in chapter 5). It has been found that they can considerably reduce the

computation time without much distorting the useful solutions. It is observed in most of the examples that the majority of the reduction in gate count will take place at the first few levels of multiplexing due to concurrent multiplexing and fewer delay units. The following five heuristic rules were implemented to speed up the synthesis process.

(1) If a design at any multiplexing level has a lesser number of gates than the design at a higher level of multiplexing, then the multiplexing paths from that higher level multiplexed design are pruned.

(2) If the *nsf* of any design is less than the minimum *nsf*, or the gate count of any design is more than the maximum gate count, then the multiplexing at the higher levels will not be performed. This heuristic rule requires the user to specify the maximum gate count and the minimum sample frequency with the intended clock frequency. The minimum *nsf* will be calculated from the given clock and sample frequencies. If the user does not specify the maximum gate count, the gate count of the operator-parallel design (without multiplexing) is considered as the default maximum gate count.

(3) Assume that pairs (A, B) and (C, D) are part of concurrent multiplexing. In some cases the order of multiplexing may cause minor differences in the group of nodes that are multiplexed around the multiplier nodes. But in general, the order in which the pairs are multiplexed does not effect the resultant DFGs. In the exhaustive search, both the multiplexed designs {(A, B), (C, D)} and {(C, D), (A, B)} are considered. This heuristic rule considers only that design whichever appears first in the search path.

(4) This heuristic prunes the multiplexing paths by comparing each new design with the minimum gate designs generated (saved) until that time. If the *nsf* of a new design is

less than the *nsf* of a saved design and the gate count of the new design is greater than the gate count of the saved design by a certain user specified amount then the further multiplexing will not performed. Even though a design may not appear to be a useful one at some level of multiplexing (at some *nsf*), it may produce a useful solution at further levels of multiplexing (at smaller *nsf*s). It has been observed in a number of design examples (given in chapter 5) that if a design is costlier than a saved design by 10%, it is not likely to produce a useful solution. Thus the default user specified amount is set to 10% of a saved design.

(5) This heuristic prunes the multiplexing paths by comparing two designs at consecutive levels of multiplexing. The design at a higher level of multiplexing is slower than a design at a lower level of multiplexing. If the ratio of the reduction in gate count to the reduction in *nsf* between the two designs is less than user specified ratio then further multiplexing will not be performed. The default user specified ratio is set to a 10% reduction in gates and to a 50% reduction in *nsf*. It has been observed in a number of design examples (given in chapter 5) that a reasonable default user specified ratio considerably reduces the computation time and without much distorting the useful solutions.

All the five heuristics may not be necessary if a design space is small. Different heuristics can be used for designs of different sizes. Thus the heuristics are implemented in five levels. The first level implements only the first heuristic, the second level implements first two heuristic rules and so on. Finally the fifth level implements all the heuristics. The effect of heuristics on the design space and the synthesis time in different examples is given in chapter 5.

Chapter 4

CONTROL GENERATOR DESIGN

4.0 Introduction

This chapter describes the generation of control signals needed in a bit-serial network [3]. The bit-serial control signals are used to control the timing or the flow of data through a network. At the lowest level, a fundamental synchronous bit clock called *cycle 0* (C0) controls the flow of all the data in a system at the bit level. In all the bit-serial operators, the outputs from the operators are latched with this control signal. All the input and output signals to the bit-serial chip are also synchronized with this bit clock.

Various other control signals are needed along with *C0* as shown in Fig. 4.1. The control signal *C1* is needed to define the start of a new data word on all of nodes. This control signal, which is called the least significant bit time (*LSB time*), is true for the first or the least significant bit (LSB) of a word. At the next level, the control signal C2 which is also called as *WORD 0 time* is true during the first word of each frame. Depending upon the complexity of a system, levels of control may be required above this, for example C3 which is true during the first frame of a group of frames, etc.

The control signal C1 is used to terminate the current operation and initiate a new one in all the arithmetic operators. For example, the bit serial adder, as shown in Fig. 1.2, processes successive pairs of input bits through a full adder stage, latches the *sum* as an

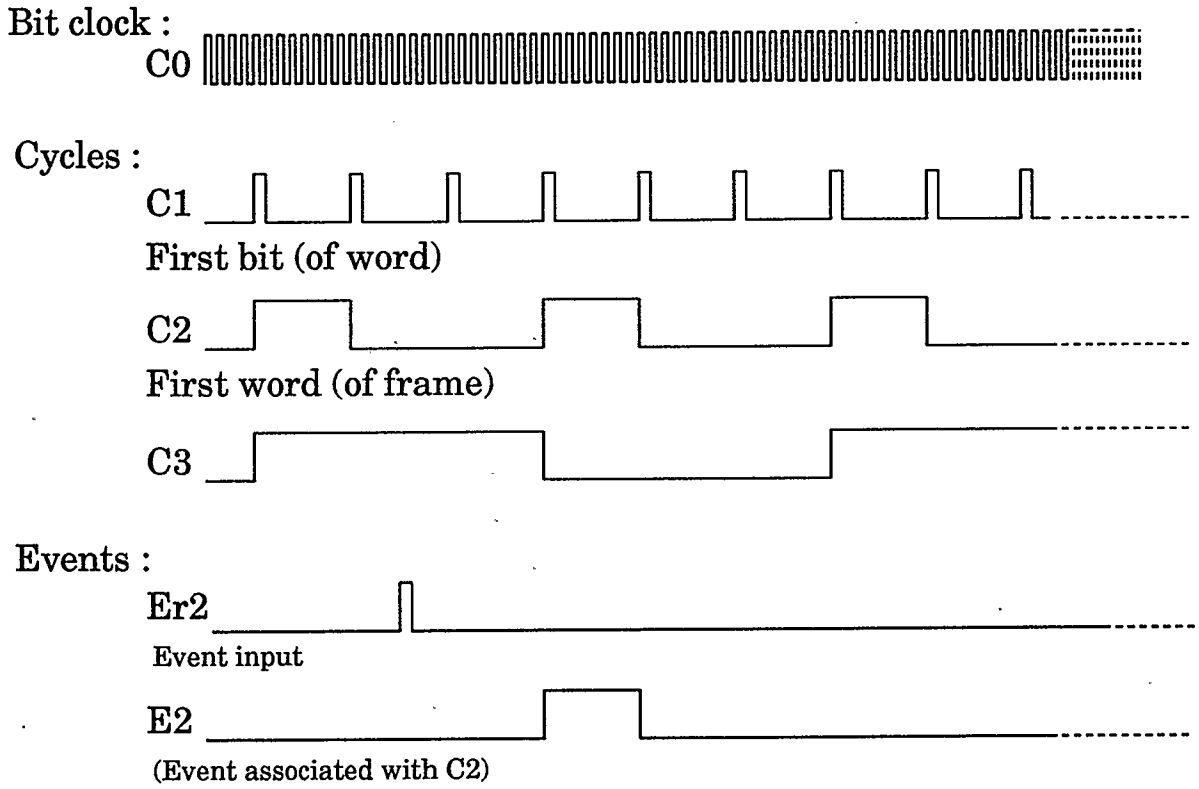


Fig. 4.1 : Format of control signals for bit-serial networks

output and feeds back the *carry* for combination with the next pair of input bits (more significant bits). The control signal C1 is used to prevent any carry resulting from the preceding addition from interfering with the next data word. The higher level control signals C2, C3 are normally used for synchronization and multiplexing.

In the bit-serial design methodology given by Denyer and Renshaw [3] the control signals are generated in a module called the control generator. The control generator module is defined by the FIRST [3] silicon compiler. The FIRST control generator generates all the required control signals. The FIRST control generator does not produce all types of control signals required in the designs generated by BITSYN. The BITSYN control generator is used to generate the signals which cannot be generated from the FIRST control generator. This chapter describes the implementation of FIRST and BITSYN control generator modules.

4.1 FIRST Control Generator

The control generator generates the bit-serial control signals C1, C2, C3, ... which are true for only the first period of the bit-clock or the previous control signal and repeats after an integer number of durations of the bit clock or the previous control signal. All of these control signals are called *cycles* as they repeat with a particular pattern.

Another type of control signals called *events* are required to control occasional or once only operations such as loading or clearing memory. The *events* are similar to *cycles* in principle, except that they do not repeat. Although *events* are not cyclic, they are

synchronous. These occur in response to an external event request to the control generator. Each *event* specified will be associated with a *cycle*. The following two different event signals are used in bit-serial networks.

EVENT [0] : The event signal is *true* for the next occurrence of the *ON* period of its associated *cycle*.

EVENT [1] : The event signal is *true* for the next occurrence of one full period of its associated cycle.

A program was developed to design a control generator which could deliver different types of control signals according to the input specification. A sample FIRST control generator specification is given below:

```
CONTROLGENERATOR (Er2, Er3, .... -> C1, C2, E2, C3, E3, .....)
  CYCLE[8]
  CYCLE[3]
  EVENT[0]
  CYCLE[2]
  EVENT[1]
  CYCLE[2]
  ........
  ........
ENDCONTROLGENERATOR
```

In the above specification, **Er2, Er3** represent the event external inputs, **C1, C2, C3** represent the cycle outputs and **E1, E3** represent the event outputs which are associated with **C1, C3**. Fig. 4.1 gives the control signals C1, C2, C3 and E1 generated according to the above specification.

4.2 BITSYN Control Generator

When a system is synthesized using BITSYN, it requires a set of signals to control the multiplexers in addition to master bit clock, *C0* and LSB time signal, *C1*. The LSB time signal, *C1* is needed in most of the bit-serial operators. For BITSYN, the LSB time signal *C1* is named *c00*. The signal *c00* has the time period equal to the system word length (swl) times the bit clock period. BITSYN uses the *c00* as the basic cycle of operation. The control signals to the multiplexers of levels 1, 2, 3, ... are named as *c10, c20, c30, ...*

The BITSYN control generator uses the FIRST control generator to produce the *c00*. The remaining control signals can be generated by the FIRST control generator when a synthesized design uses all the 2^m cycles (m is the multiplexing level). The *c10, c20, c30, ...* signals are same as the *C2, C3, C4, ...* of length 2 produced from FIRST control generator. The control generator specification for a system with 3 levels of multiplexing and *swl* of 32 is as follows.

```
CONTROLGENERATOR (-> c00, c10, c20, c30)
   CYCLE[32]
   CYCLE[2]
   CYCLE[2]
   CYCLE[2]
ENDCONTROLGENERATOR
```

For the above specification, the control signals are same as the following cyclic finite sequence of digits (0 or 1 of time period equal to *swl*).

c10 = '10101010' c20 = '11001100' c30 = '11110000'

BITSYN requires the additional control signals such as cs0 = '10000000' and cs1 = '00000001' to get an input sample, to update state values and to output the data at

appropriate times. The BITSYN control generator generates cs0 and cs1 using c10, c20, ... as inputs to a combinational logic unit. The signals *cs0* and *cs1* are generated as cs0 = AND(c10, c20, ...) and cs1 = NOR(c10, c20, ...). These signals are latched with the bit clock.

It can be observed from the sequences generated in the FIRST control generator that the ON and OFF periods of cn0 are equal to 2^{n-1} times *swl* or 2^{n-1} cycles (*c00*). In some design examples where a few cycles are redundant (timing redundancy was explained in Chapter 2), the ON and OFF periods may not necessarily be same for control signals c20, c30, ..., cm0. In the above example, when the OFF period of c20 is equal to one *swl* instead of two times the *swl*, the cyclic finite sequences are modified as follows

c10 = '101101' c20 = '110110' c30 = '111000'

This type of control signals can not be produced by the FIRST control generator. Different cyclic finite sequences may be required depending upon where the timing is redundant. The cyclic finite sequences can be generated using two methods (a) linear feedback shift registers or linear sequential circuits and (b) finite state machines.

4.3 Implementation of FIRST Control Generator

The basic cells selected to implement the control generator are NAND, NOR, NOT, and D flipflop with asynchronous clear input. These cells are selected because they will be available in any design libraries such as gate array, standard cell etc. and in any technology such as NMOS, CMOS, etc.

The FIRST control generator can be realized using a set of synchronous counters. Each counter is used to generate a *cycle* signal pulse for every given number of pulses of the fundamental clock or the previous *cycle*. The events can be generated after detecting an event input, with the help of the circuits used to generate *C1*, *C2* etc.

The first step is to design a synchronous counter which will count *L* number of states where *L* is the length of the cycle. A down counter of length *L* is implemented as shown in Fig. 4.2. In all the figures, *load n* = 1 means the counter in level *n* is in zero state and *loadb n* is an inverted signal of *Load n*.

A down counter is needed for each *cycle* to be generated. The fundamental bit clock (master clock) is used as the clock to the counter which generates *C1*. To generate *C2, C3*,... the previous cycle output (*C1, C2*, ...) is used as the clock to the counter. The circuit to generate the *C1* is shown in Fig. 4.3a and the circuit to generate all other *cycles* is same and is shown in Fig. 4.3b. The extra circuitry used in Fig. 4.3b is needed to synchronize all the cycle signals.

The circuit to generate *EVENT[1]*, which occurs for one full period of the *cycle*, is given in Fig. 4.4a. Here the first D flipflop detects the external event input *Er n* whenever it occurs using the master clock. The remaining circuit is used to create the event output in synchronous with the *cycle* after the external input is detected. A similar circuit to generate *EVENT[0]* which occurs for only *ON* period of the *cycle* is given in Fig. 4.4b.

All the cycles and events are delayed by 1 latch delay with respect to master clock.

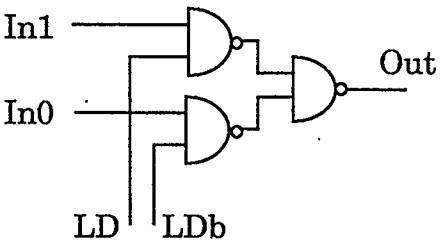


Fig. 4.2a : Schematic diagram of MUX2

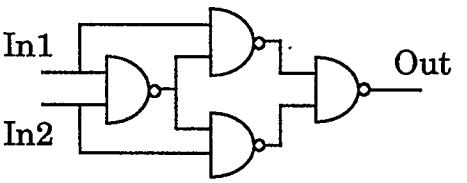


Fig. 4.2b : Schematic diagram of EXOR

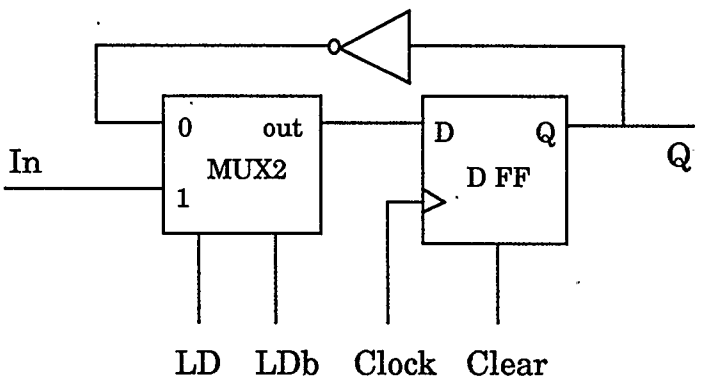


Fig. 4.2c : Schematic diagram of CNT1
(1st stage of the counter -- LSB stage)

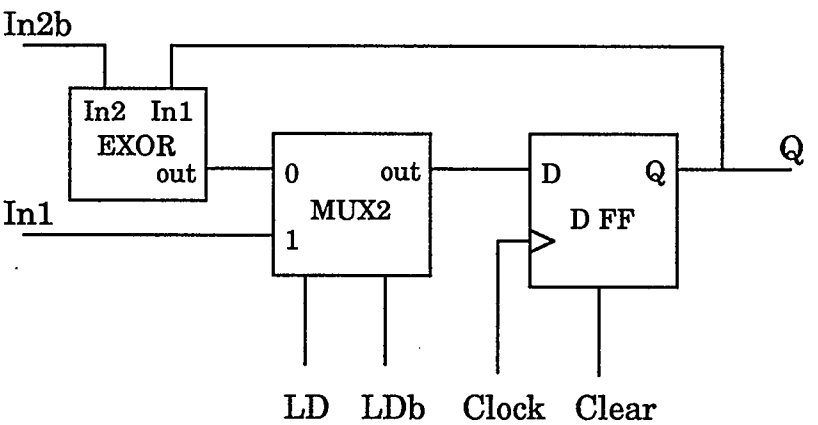


Fig. 4.2d : Schematic diagram of CNTN
(2nd to Nth stage of the counter)

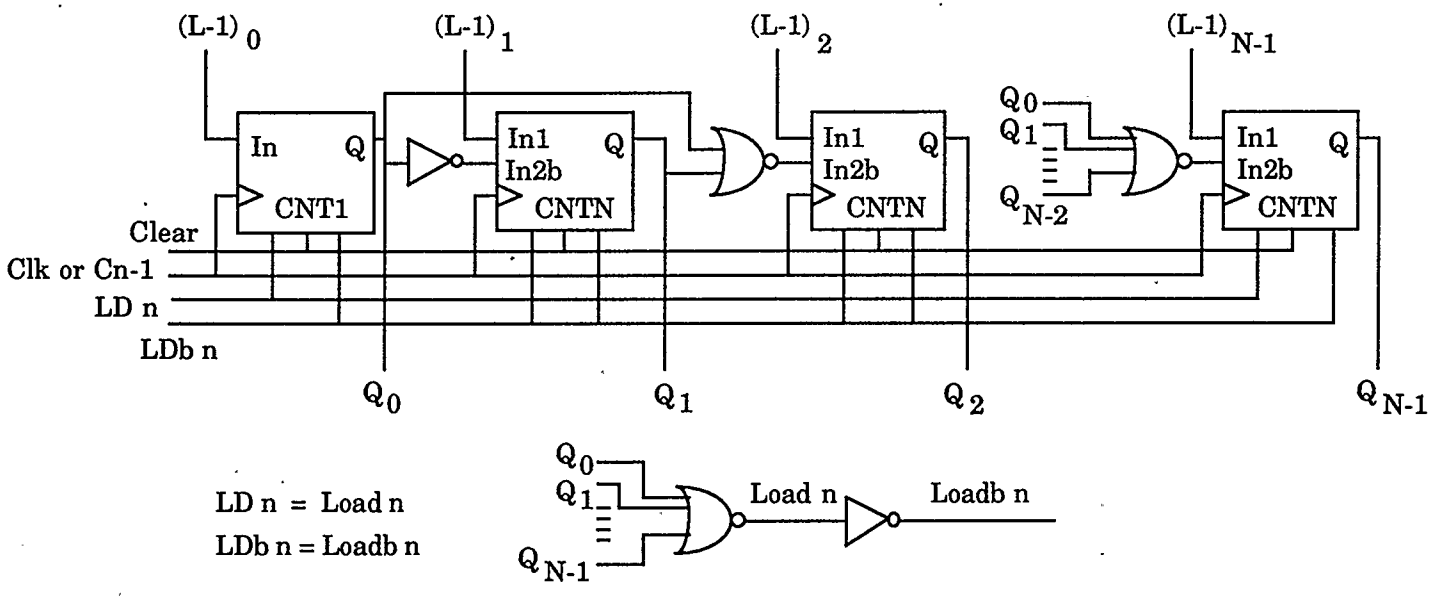
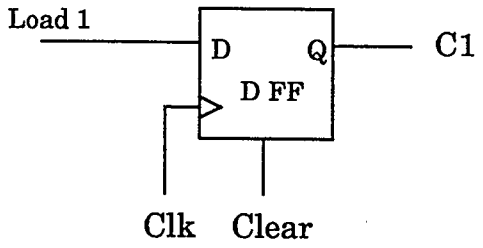
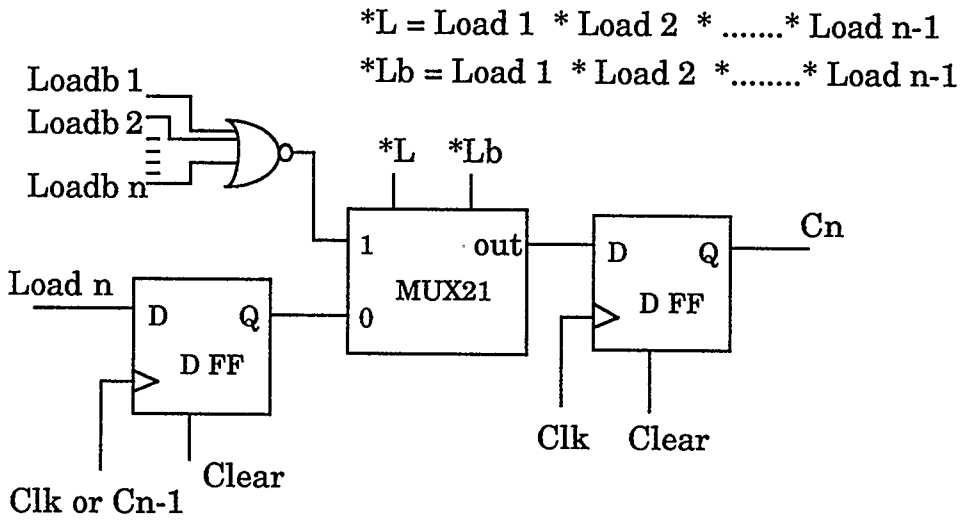


Fig. 4.2e : Schematic diagram of down counter of length L



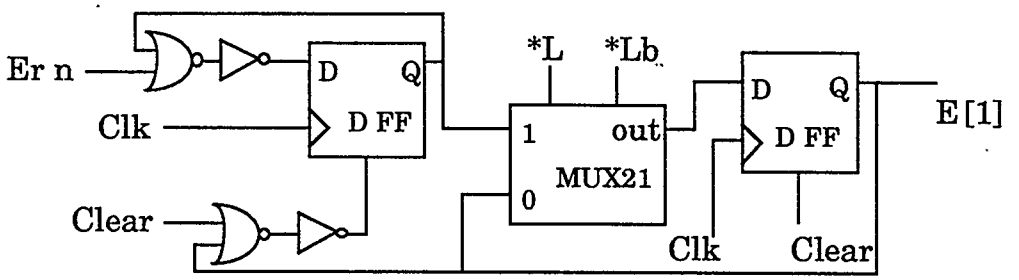
(a) 1st cycle



(b) 2nd to nth cycle

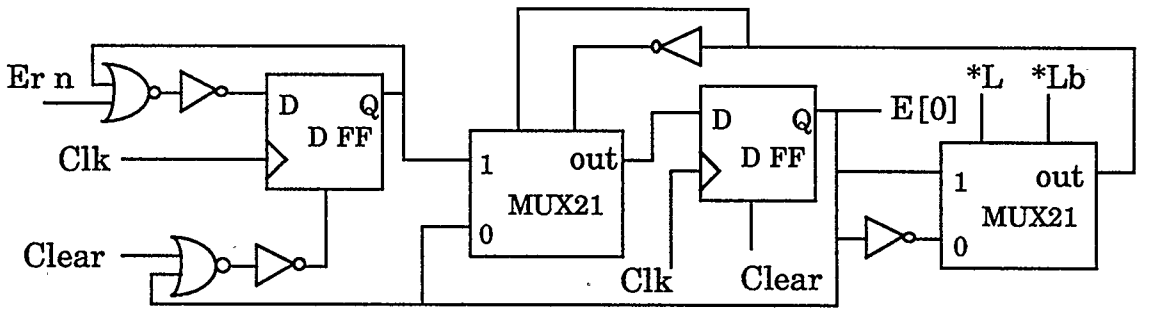
Fig. 4.3 : Schematic diagram of the circuits to generate CYCLEs

*Lb = Load 1 * Load 2 * ..... Load n
*L = Load 1 * Load 2 * ..... Load n



(a) Event type 1 -- True for one full period of the cycle

*L = Load 1 * Load 2 * .....* Load n-1 * Loadb n
*Lb = Load 1 * Load 2 *.....* Load n-1 * Load n



(b) Event type 0 -- True for only the ON period of the cycle

Fig. 4.4 : Schematic diagram of the circuits to generate EVENTs

4.4 Linear Feedback Shift Registers (LFSR)

It is known that shift registers with feedback and/or feed forward connections can be used to multiply and divide polynomials over the binary number field [28] [29]. The circuit that simultaneously multiplies by h(D) and divides by g(D) is shown in Fig. 4.5 [29]. The output b(D) is given by

$$b(D) = \frac{h(D)}{g(D)} a(D)$$

where

$$h(D) = \sum_{j=0}^{r} h_j D^j$$

$$g(D) = \sum_{j=1}^{r} g_j D^j + 1$$

and the input sequence

$$a(D) = \sum_{j=0}^{\infty} a_j D^j$$

In the Fig. 4.5, the boxes represent unit delays or registers, circles labeled with subscripted coefficients represent a connection if the coefficient is a 1 and no connection if the coefficient is a 0. The circles with a "+" inside represent modulo-2 adders or Exclusive-OR gates.

The circuit that can produce a desired cyclic finite sequence can be synthesized if the transfer function of that circuit is known. If the impulse response of the linear sequential circuit is given, then the transfer function can be determined. By analogy to conventional linear circuit, the impulse response of a linear sequential circuit is defined as the sequence

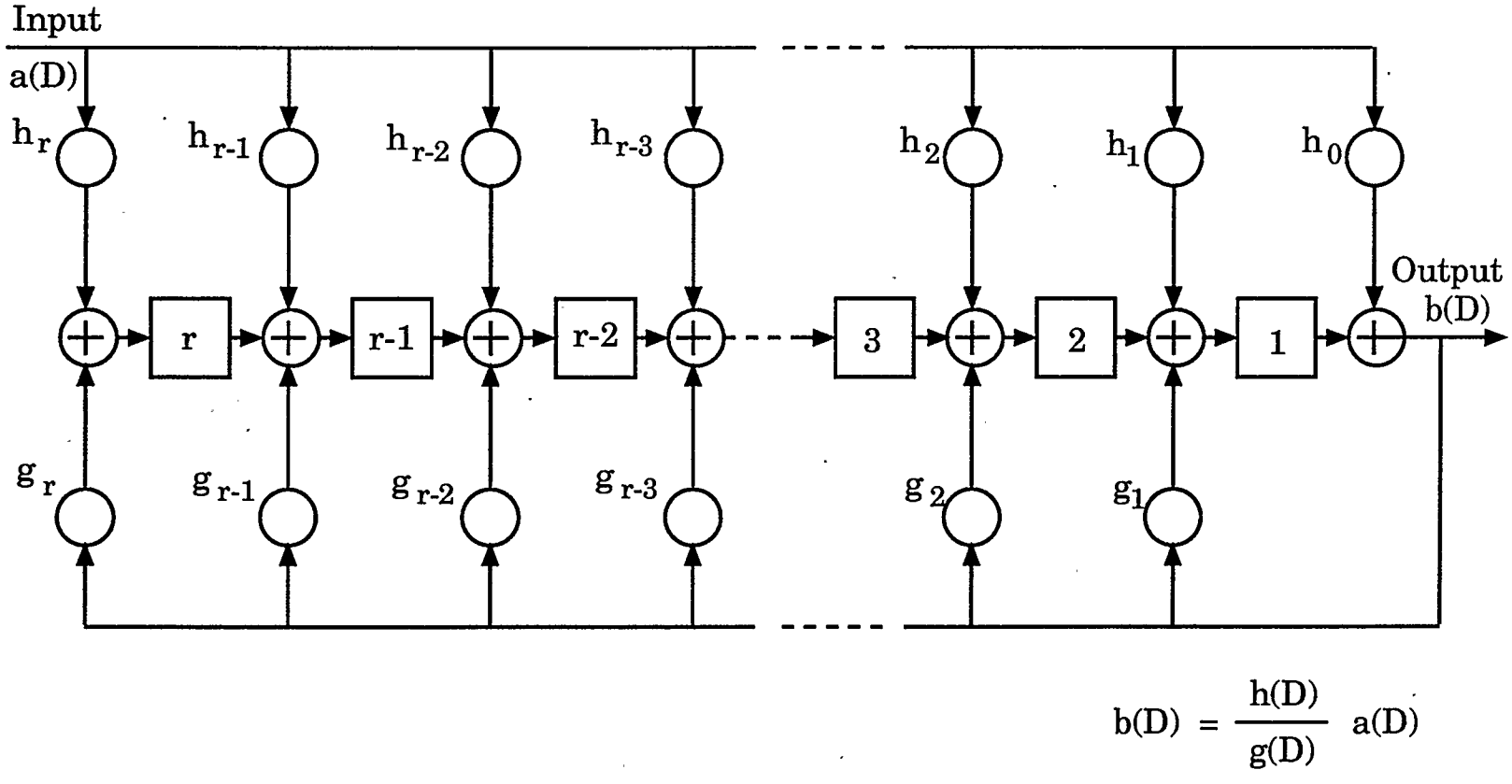


Fig. 4.5 : Circuit that multiplies by h(D) and divides by g(D)

produced by the input sequence 1 0 0 0 0 ..... [28] (All registers are assumed to contain zeros initially). The linear sequential circuit whose impulse response is periodic with period 'r', and whose first period is

$$b_0\, b_1\, b_2 \ldots\ldots b_{r-2}\, b_{r-1}$$

can be synthesized by setting the coefficients in Fig. 4.5 as

$$h_i = b_i \text{ and } g_i = 0 \quad \text{for i = 0 to r-1;} \qquad g_r = 1 \text{ and } h_r = 0$$

The resultant transfer function

$$h(D)=\frac{b_0 + b_1\, D + b_2\, D^2 + \ldots\ldots + b_{r-1}\, D^{r-1}}{1 + D^r}$$

The above transfer function if implemented as it is, would require 'r' unit delays. If the numerator and denominator have a common factor, then the transfer function can be simplified and it will have the same impulse response, but will require the minimum number of register units. If the length of the sequence is 'r', then the number of register units can vary from 'm' to 'r' where $2^{m-1} \leq r \leq 2^m$ and it purely depends on the impulse response.

A similar method is used to find a feedback shift register which can synthesize a given desired sequence. In fact the method synthesizes a circuit whose impulse response is same as the desired sequence and then omits the input to the circuit and initializes the register units appropriately. The initial conditions of registers should be set to match what the input circuit would insert if a 1 is applied. Thus the initial conditions of the registers will be equal to $h_0 h_1 h_2 \ldots\ldots$.

4.5 Finite State Machines (FSM)

A synchronous sequential machine whose past histories can affect their future behavior in only a finite number of ways can be called as finite state machine [30]. A state machine consisting of a finite number of memory devices and combinational logic is represented schematically in Fig. 4.6. The signal value at the output of the memory devices is referred to as the *state variable* s_i. The combination of 'm' state variables $\{s_1, s_2, ..., s_m\}$ represent the present state of the machine. The 'm' state variables can give 2^m different combinations and thus the set of n = 2^m m-tuples constitutes the entire set of states of the machine. The circuit in Fig. 4.6 has a finite number 'k' of input terminals and has a finite number 'l' of output terminals.

The external inputs $x_1, x_2, ..., x_k$ and the values of the state variables $s_1, s_2, ..., s_m$ are given to the combinational logic circuit which in turn produces the outputs $z_1, z_2, ..., z_l$ and the the next state variables $S_1, S_2, ..., S_m$. The relation among the input, present state, output and next state is normally described by a state diagram or a state table. The synthesis of finite state machine involves different steps such as state minimization, state assignment, and implementation of combinational logic using gates, or random access memory. The FSM with no external inputs can be used to synthesize cyclic finite sequences and may be used as part of BITSYN control generator.

4.6 LFSR versus FSM

If a cyclic finite sequence of length 'r' has to be designed, the FSM implementation requires 'm' ($2^{m-1} \leq r \leq 2^m$) memory devices where as LFSR implementation requires the

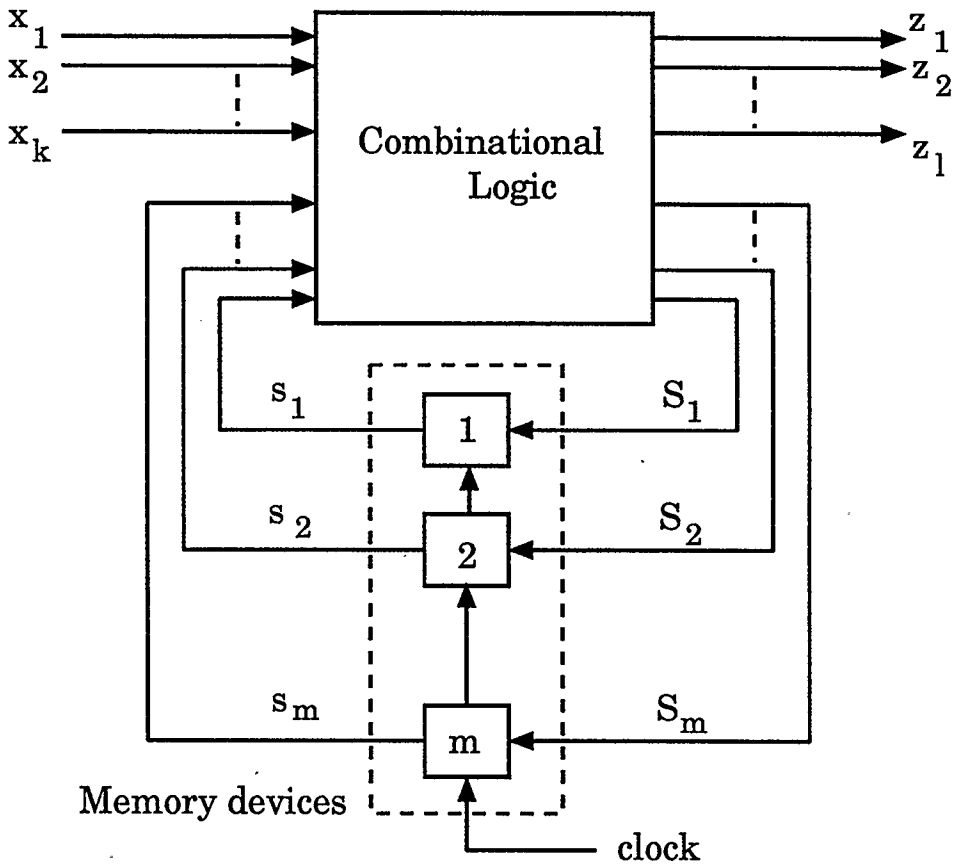


Fig. 4.6 : Schematic representation of a finite state machine

number of memory devices which vary from 'm' to 'r'. The number of registers necessary in an LFSR implementation depends mainly on the characteristics of the sequence. In an LFSR implementation, the required sequence is generated by the proper initialization of the registers in combination with the usage of modulo-2 adders in feedback path. In FSM implementation, the sequence will be generated with the help of the combinational logic and the state variables. In general, if the registers are not considered, the LFSR implementation requires less hardware than the FSM implementation. But the LFSR may require more registers than the FSM, and the difference in the number of registers will increase as the length of the sequence increases.

Example 4.1 : The LFSR circuit whose response is periodic with period 7 and whose first period is 1001011....., has as its transfer function

$$H(D) = \frac{1 + D^3 + D^5 + D^6}{1 + D^7} = \frac{1 + D^2}{1 + D^2 + D^3}.$$

The shift register generator for the above example is as shown in Fig. 4.7 and it can be implemented using 3 registers, one EX-OR and two OR gates. Thus the number of LSI equivalent gates required for the above implementation is equal to 23 [31].

The sequence is also implemented using FSM. The finite state machine compiler, PEG (PLA equation generator) [32], is used to translate the high level language description of a state machine into logic equations needed to implement the state machine design. The logic equations were implemented manually using basic two input gates and it was found that the number of LSI equivalent gates required for the above example is 43.

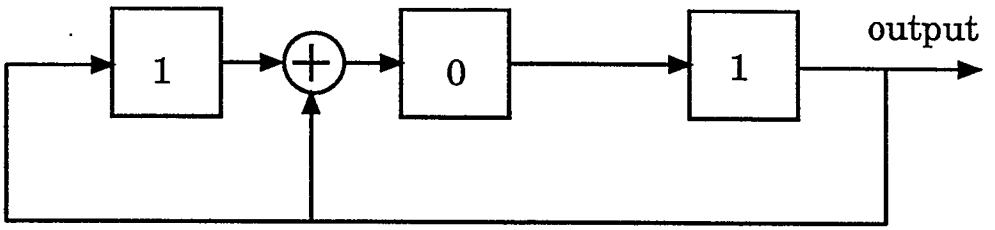


Fig. 4.7a : A shift register generator with output 1001011.....

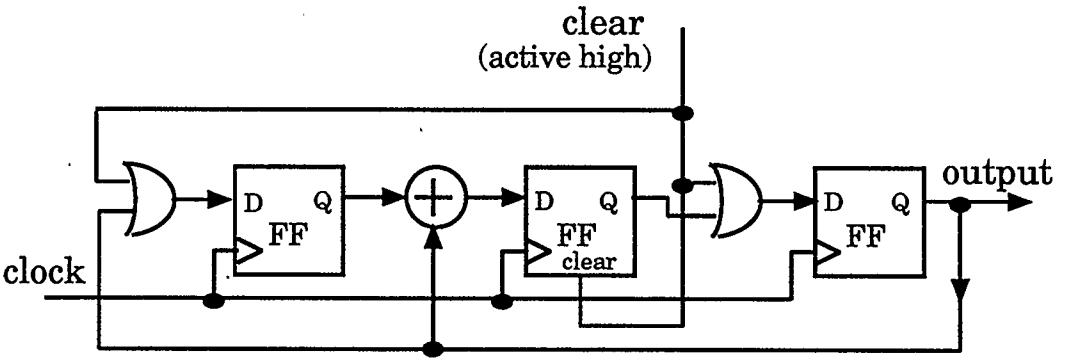


Fig. 4.7b : Circuit implementation of the shift register generator given in Fig. 4.7a

In the above example, even if the transfer function couldn't have been simplified, the LFSR circuit requires 7 registers connected as ring counter out of which 4 registers are initialized to 1. This implementation requires 43 LSI [31] equivalent gates. Thus LFSR can be used to generate cyclic finite sequences. But as the length of the sequence increases, FSM implementation will be cheaper than the LFSR implementation.

In general, more than one finite sequence is required for the control signals in a bit-serial design synthesized by BITSYN. The LFSR design requires separate circuits to generate different sequences and thus requires more registers and gates. The FSM design requires the same number of registers but uses more gates to generate additional outputs. Thus the FSM design may be cheaper than the LFSR design when a larger number of control signals are required. The following example takes the cyclic sequences which are used as control signals in practical digital filter circuits designed by BITSYN.

Example 4.2 : Generate the three control signals C10 with cyclic sequence of '101', C20 with cyclic sequence of '110' and c30 with cyclic sequence of '111000'.

The transfer function for C10,

$$H_1(D) = \frac{1 + D^2}{1 + D^3} = \frac{1 + D}{1 + D + D^2}$$

for c20,

$$H_2(D) = \frac{1 + D}{1 + D^3} = \frac{1}{1 + D + D^2}$$

and for c30,

$$H_3(D) = \frac{1 + D + D^2}{1 + D^6} = \frac{1}{1 + D + D^3 + D^4}$$

The LFSR implementation of C10 uses 17 LSI equivalent gates (2 registers, 2 OR gates and one EX-OR). The LFSR for C20 requires 16 LSI equivalent gates (2 registers, 1 OR gates and one EX-OR) and for c30 requires 31 LSI equivalent gates (4 registers, 1 OR gate and 2 EX-OR gates). Thus LFSR implementation uses 64 LSI equivalent gates.

In the case of the FSM, the state variables remains the same for all three sequences and thus require 3 registers. The combinational logic after the manual minimization requires 26 equivalent gates. Thus the FSM implementation uses 44 equivalent gates.

4.7 Conclusions

The LFSR design can be used to generate a single finite sequence. To generate multiple sequences, the FSM will be cheaper in hardware than the LFSR circuit. Thus it can be concluded that when the FIRST control generator cannot generate the required signals, they can be generated by synthesizing a FSM. Regarding the design process, synthesis of a LFSR circuit is much simpler than a FSM circuit. The synthesis process of LFSR can be easily automated and can be easily adapted into BITSYN program. The synthesis of an optimized FSM is not simple but it can be performed using existing tools such as PEG [32] and logic synthesis tool ESPRESSO [33]. The number of LSI [31] gates required for LFSR implementation can be calculated very easily. The BITSYN package takes LFSR implementation into account to compute over all gate count. The difference in gates between LFSR and FSM implementations is negligible when compared to the over all gate count of a design.

Chapter 5

CASE STUDIES

This chapter presents six DSP filter case studies generated using the BITSYN tool. A variety of practical digital filters are designed using BITSYN. BITSYN supports a set of bit-serial primitives necessary to design digital filters. Generic bit-serial operators are also supported so that BITSYN can synthesize DSP algorithms other than digital filtering algorithms. The case studies presented illustrate the synthesis of recursive digital filters such as biquadratic and wave filters and a non-recursive structure such as a sine/cosine generator. In all the design examples, the gate count (hardware cost) for each design is calculated using the LSI [31] equivalent gates of the FIRST [3] primitives. LSI logic defines "one gate equivalent as equal to two P-channel and two N-channel transistors or enough transistors to create either a two-input NAND gate or a two-input NOR gate" [31]. BITSYN has been implemented in the **C** programming language. For each design example, the time required to obtain the designs is given in *cpu* seconds with the BITSYN program running on a SUN SPARC 1+ computer.

5.1 Fourth order Cascaded Biquad Digital Filter (CasBiq-4)

A low pass filter whose transfer function is given by

$$H(Z) = \frac{g\ (z^2 - 2a_1 z + 1)\ (z^2 + 2z + 1)}{(z^2 - 2b_1 z + b_2)\ (z^2 - 2b_3 z + b_4)}$$

is implemented using two cascaded biquadratic sections. The signal flow graph (SFG) of the filter is given in Fig. 5.1a and its BITSYN branch description is given in Fig. 5.1b. A multiplication of a signal by 2 is implemented by adding the signal to itself in the SFG, as the FIRST silicon compiler supports only fractional coefficient multipliers. The multiplication of a signal by 2 can also be implemented using left shift operation. After the SFG is read by BITSYN, it will be modified as shown in Fig. 5.2a. All the multiplier branch commands in the SFG netlist are interpreted and converted into the equivalent commands using add, subtract or shift operations. The modified SFG netlist can be represented in the form of data flow graphs (DFG). There are four state inputs and one output in this example. Five DFGs are created for each state input and each output as shown in Fig. 5.2b. Two nodes with the same name represent a single instance in the figure. All five DFGs use the same data base.

The DFGs without any modification represent an operator-parallel design solution where separate multipliers are used for each multiplier operation. The DFGs are scheduled and the delays are allocated. The gate count and the normalized sample frequency (nsf) will be found. The next step is to form a list of multiplier node pairs. The six multiplier nodes in this example generates 30 different node pairs. Since the multipliers are independent of each other, each pair can produce a valid multiplexed design and more than one pair can be multiplexed at the same level (concurrent multiplexing).

A design at the 3rd level of multiplexing is chosen to explain the flow of the BITSYN algorithm. This design uses concurrent multiplexing (CM). The groups of nodes that are multiplexed at the first level are shaded in Fig. 5.2b. The three multiplier pairs (5ax, 5az),

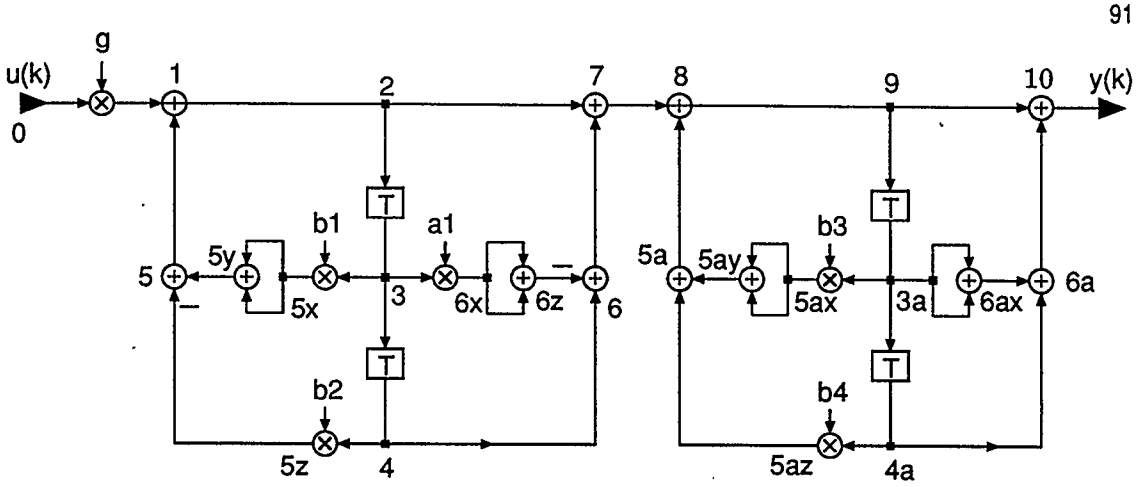


Fig. 5.1a : Signal flow graph of 4th order cascaded biquad digital filter

input	0	ac		mult	7	8	1.0
mult	0	1	g	mult	5a	8	1.0
mult	5	1	1.0	mult	8	9	1.0
mult	1	2	1.0	delay	9	3a	
delay	2	3		delay	3a	4a	
delay	3	4		mult	3a	5ax	b3
mult	3	5x	b1	mult	5ax	5ay	1.0
mult	5x	5y	1.0	mult	5ax	5ay	1.0
mult	5x	5y	1.0	mult	5ay	5a	1.0
mult	5y	5	1.0	mult	4a	5az	b4
mult	4	5z	b2	mult	5az	5a	-1.0
mult	5z	5	-1.0	mult	3a	6ax	1.0
mult	3	6x	a1	mult	3a	6ax	1.0
mult	6x	6z	1.0	mult	6ax	6a	1.0
mult	6x	6z	1.0	mult	4a	6a	1.0
mult	6z	6	-1.0	mult	6a	10	1.0
mult	4	6	1.0	mult	2a	10	1.0
mult	6	7	1.0	mult	10	11	-1.0
mult	2	7	1.0	output	11		

Fig. 5.1b : 4th order cascaded biquad digital filter - BITSYN branch description

input	0	ac		add	6	7	1
mult	0	1r	g	add	7	8	5a
add	5	1	1r	delay	8	3a	
delay	1	3		delay	3a	4a	
delay	3	4		mult	3a	5ax	b3
mult	3	5x	b1	add	5ax	5ay	5ax
add	5x	5y	5x	sub	5ay	5a	5az
sub	5y	5	5z	mult	4a	5az	b4
mult	4	5z	b2	add	3a	6ax	3a
mult	3	6x	a1	add	6ax	6a	4a
add	6x	6z	6x	add	6a	7a	8
sub	4	6	6z	output	7a		

Fig. 5.2a : 4th order cascaded biquad digital filter
- modified SFG description

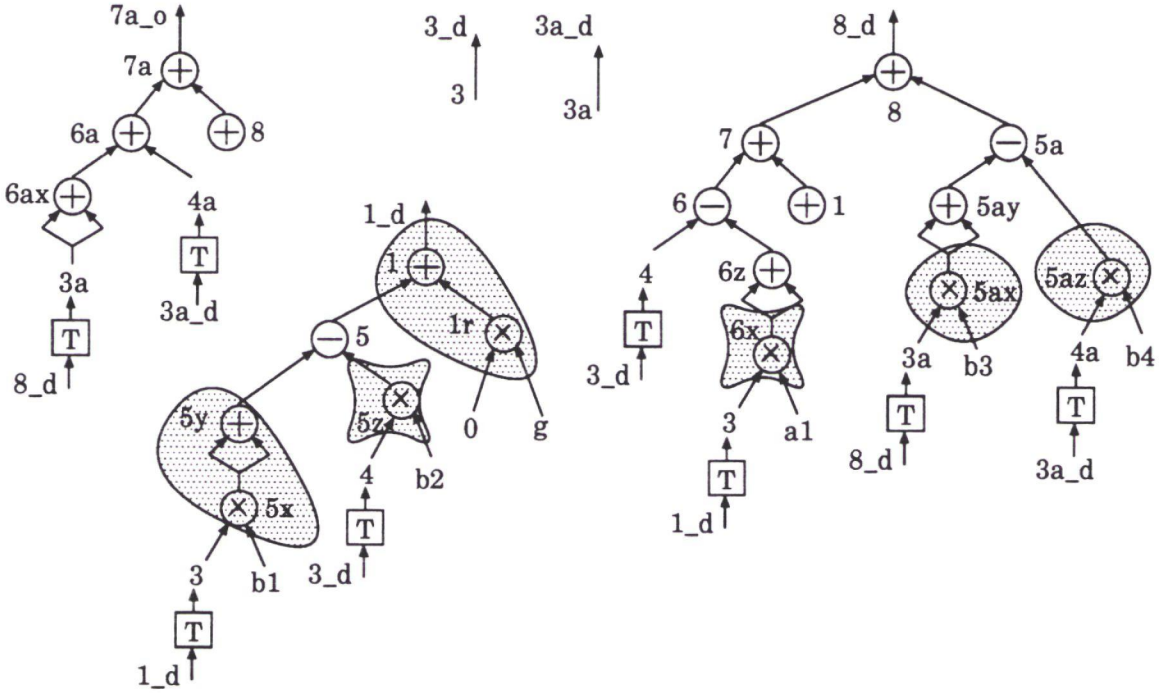


Fig. 5.2b : 4th order cascaded biquad digital filter
- data flow graphs

(5x, 1r) and (6x, 5z) are multiplexed concurrently. An adder pair (5y, 1) is multiplexed along with the pair (5x, 1r). The resultant DFGs after first level of multiplexing are shown in Fig. 5.3. The DFGs will then be scheduled and the delays will be allocated.

Each demultiplexer with a name 'L1...' is replaced with a number of delay units equal to *swl*. The demultiplexer nodes with names 'R1...' are removed. The delays units are not inserted into the DFGs, instead the number of delay units required at the output of any node can be represented by the appropriate T_{needed} time. The demultiplexer node with name 'L1...' is removed and the T_{needed} time of its child node will be modified appropriately. The time schedule for all the operators after the scheduling and allocation steps is given in Table 5.1. In the table, the T_{needed} values identified with "*" are due to the first level demultiplexers 'L1...' The control signal *c10* which can be represented by a cyclic sequence "10" is used to control the multiplexers 'x1...'. The coefficient word length (cwl) of all the multipliers has been set to 10 bits. The minimum *swl* required for the design is equal to 23. The total gate cost of the design in terms of LSI equivalent gates and the *nsf* of the design can then be found.

The design at the first level of multiplexing contains 3 multipliers. It can be seen from Fig. 5.3 that the three multipliers are independent of each other. Therefore six different valid multiplier pairs are available for multiplexing at level 2. The multiplier pair (5x, 6x) is used at the second level of multiplexing. The resultant DFGs are shown in Fig. 5.4 and the time schedule for all the operators after scheduling and allocation steps is given in Table 5.2. The times due to the 2nd level demultiplexers 'L2...' are identified with "#". The minimum

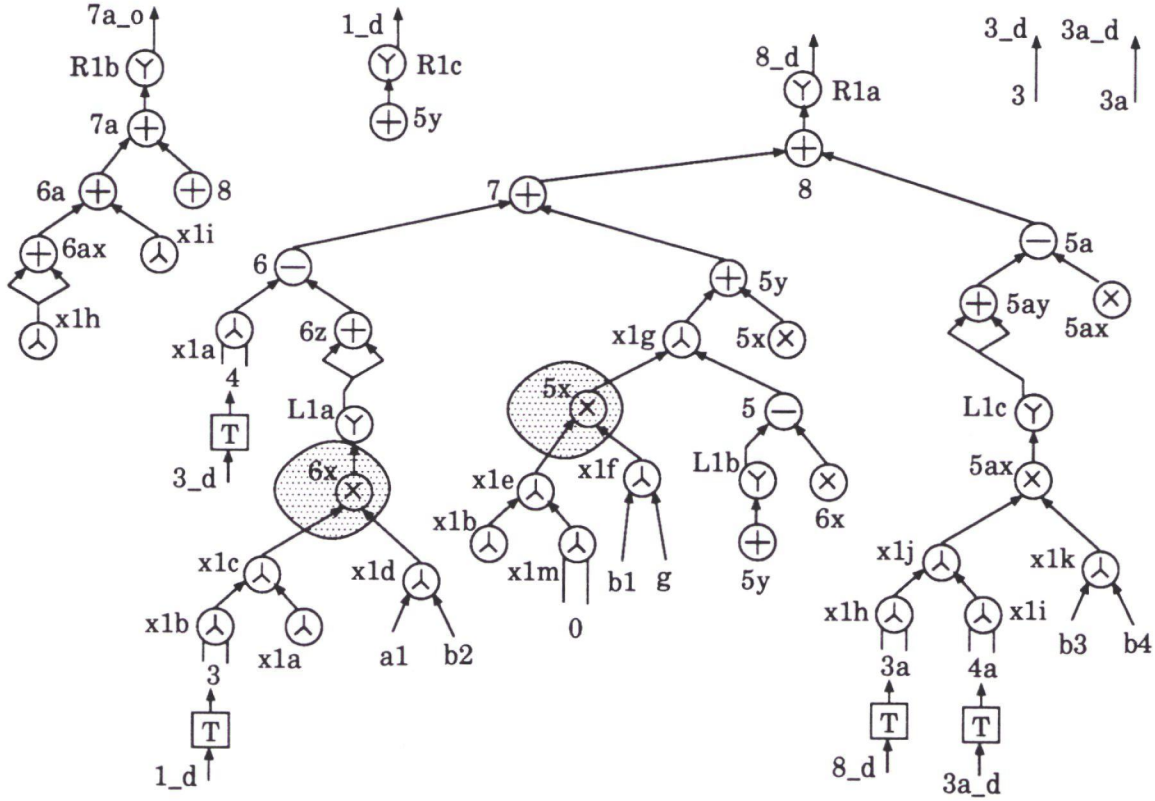


Fig. 5.3 : 4th order cascaded biquad digital filter
- data flow graphs with multiplexing level = 1

Table 5.1 : 4th order cascaded biquad digital filter - time schedule after 1st level multiplexing (cwl = 10 bits; swl = 23 bits)

Node	T_{in}	T_{out}	T_{needed}	Child nodes
3, 4, 0	-	0	0	-
x1b	0	1	1, 2, 22	3, 3
x1a	0	1	1	4, 4
x1c	1	2	2	x1a, x1b
a1, b2	-	1	1	-
x1d	1	2	2	a1, b2
6x	2	19	19, 43*	x1c, x1d
6z	20 (43*)	21	21	6x*, 6x*
6	21	22	22	x1a
x1m	0	1	2	0, 0
x1e	2	3	3	x1b, x1m
b1, g	-	2	2	-
x1f	2	3	3	b1, g
5x	3	20	20, 21	x1e, x1f
5	19, 42*	20	20	6x, 5y*
x1g	20	21	21	5x, 5
5y	21	22	22, 42*	x1g, 5
7	22	23	23	6, 5y
3a	-	2	2, 3	-
x1h	2	3	4, 22	3a, 3a
4a	-	0	0, 3	-
x1i	0	1	4, 23	4a, 4a
x1j	4	5	5	x1i, x1h
b3, b4	-	4	4	-
x1k	4	5	5	b3, b4
5ax	5	22	22, 44*	x1j, x1k
5ay	21 (44*)	22	22	5ax*, 5ax*
5a	22	23	23	5ay, 5ax
8	23	24	24	7, 5a
8_d	24	24	-	8
1_d	22	22	-	5y
3_d	1	1	-	x1b
3a_d	4	4	-	x1h
6ax	22	23	23	x1h, x1h
6a	23	24	24	6ax, x1i
7a	24	25	25	6a, 8
7a_o	25	25	-	7a

* due to 1st level demultiplexer 'L1...'

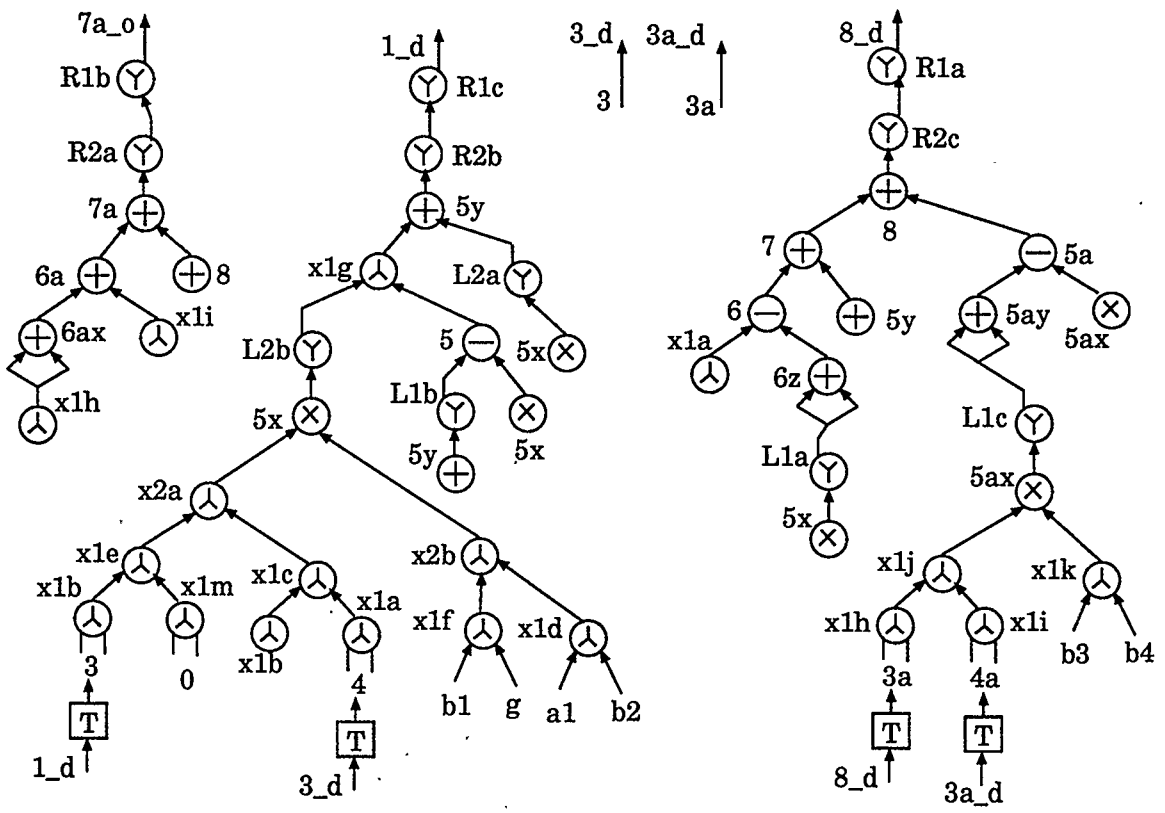


Fig. 5.4 : 4th order cascaded biquad digital filter
- data flow graphs with multiplexing level = 2

Table 5.2 : 4th order cascaded biquad digital filter - time schedule after 2nd level multiplexing (cwl = 10 bits; swl = 24 bits)

Node	T_{in}	T_{out}	T_{needed}	Child nodes
3, 4, 0	-	0	0	-
x1b	0	1	1, 23	3, 3
x1m	0	1	1	0, 0
x1e	1	2	2	x1b, x1m
x1a	0	1	1, 22	4, 4
x1c	1	2	2	x1a, x1b
x2a	2	3	3	x1c, x1e
b1, g	-	1	1	-
x1f	1	2	2	b1, g
a1, b2	-	1	1	-
x1d	1	2	2	a1, b2
x2b	2	3	3	x1f, x1d
5x	3	20	20, 45*, 69#, 70#	x2a, x2b
5	20 (44*)	21	21	5y*, 5x
x1g	21 (69#)	22	22	5, 5x#
5y	22 (70#)	23	23, 44*	x1g, 5x#
1_d	23	23	-	5y
6z	21 (45*)	22	22	5x*, 5x*
6	22	23	23	6z, x1a
7	23	24	24	6, 5y
3a	-	2	2, 4	-
x1h	2	3	5, 23	3a, 3a
4a	-	0	0, 4	-
x1i	0	1	1, 5, 24	4a, 4a
x1j	5	6	6	x1h, x1i
b3, b4	-	5	5	-
x1k	5	6	6	b3, b4
5ax	6	23	23, 46*	x1k, x1j
5ay	22 (46#)	23	23	5ax*, 5ax*
5a	23	24	24	5ay, 5ax
8	24	25	25	7, 5a
8_d	25	25	-	8
3_d	1	1	-	x1b
3a_d	5	5	-	x1h
6ax	23	24	24	x1h, x1h
6a	24	25	25	x1i, 6ax
7a	25	26	26	6a, 8
7a_o	25	25	-	7a

* due to 1st level demultiplexer nodes 'L1...'
\# due to 2nd level demultiplexer nodes 'L2...'

swl of the design is found to be 24. Here each demultiplexer node with a name 'L2...' will be substituted with a number of delay units equal to *2 * swl*. The control signals *c10* (cyclic sequence = "1010") and *c20* (cyclic sequence = "1100") are used to control the multiplexers of first and second level respectively.

The design at the 2nd level of multiplexing contains 2 multipliers. Thus there will be two multiplier pairs that can be multiplexed at the third level. The multiplier pair (5x, 5ax) gives a design at the third level of multiplexing. The multiplier node *5x* is multiplexed twice to perform four multiplication operations. The multiplier node *5ax* is multiplexed once to perform two multiplication operations. Because both multipliers are independent of each other, there exists a timing redundancy. That is only six out of eight cycles of *c00* are required to perform all the multiplication operations and the remaining two cycles are redundant. Since the second node of the pair is *5ax*, the OFF period of *c30* is equal to 2 cycles and ON period is equal to 4 cycles. The control signals *c10* (cyclic sequence = "101010"), *c20* (cyclic sequence = "110000") and *c30* (cyclic sequence = "111100") are used to control the multiplexers of 1st, 2nd and 3rd level multiplexers respectively.

The new DFGs are given in Fig. 5.5 and the time schedule for all the operators after scheduling and allocation steps is given in Table 5.3. The times due to the 3rd level demultiplexers are identified with "@". Here each demultiplexer node with name 'L3...' will be substituted with a number of delay units equal to *2 * swl*. The number of delay units is equal to 2 times the *swl* instead of 4 times the *swl*, because of the two redundant cycles. The gate count of this design is found to be 2762 and its *nsf* will be equal to 0.00725.

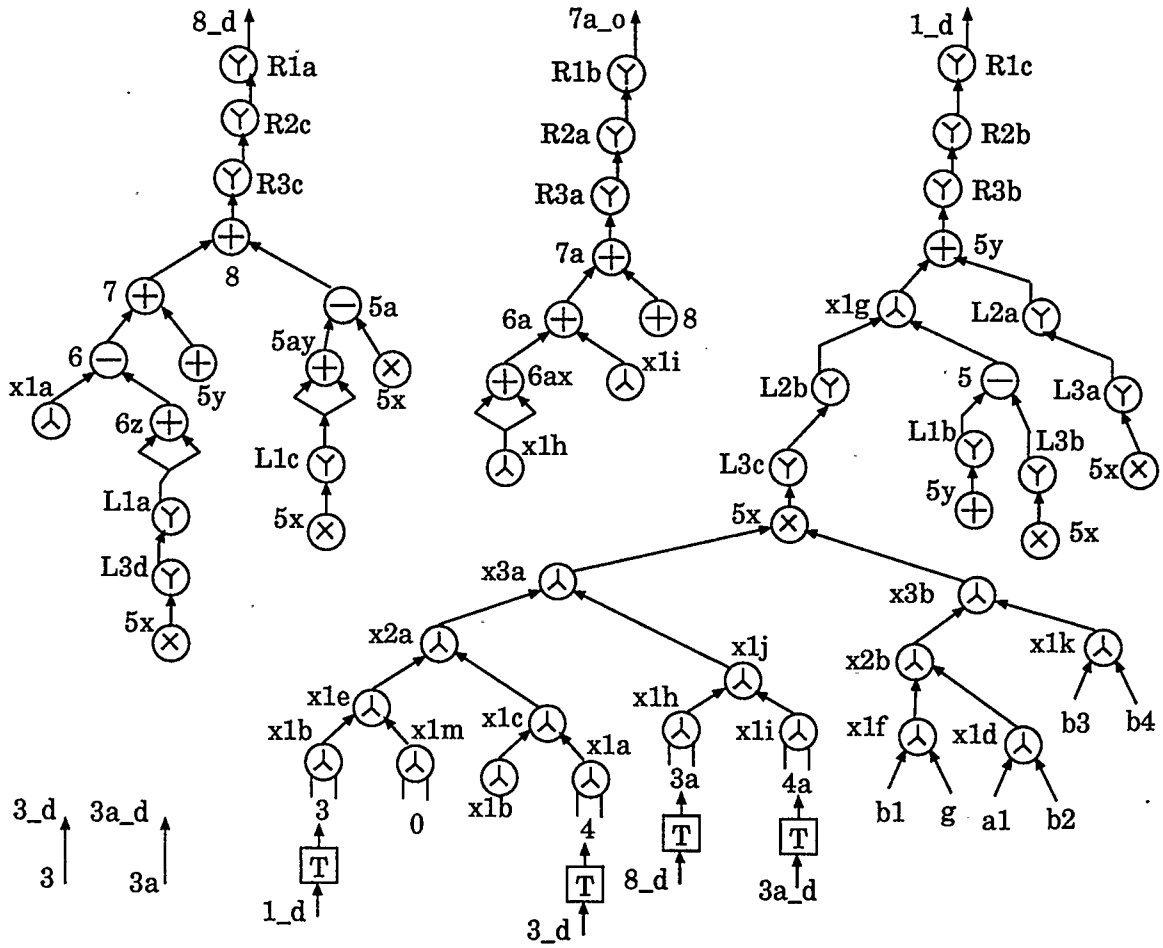


Fig. 5.5 : 4th order cascaded biquad digital filter
- data flow graphs with multiplexing level = 3

Table 5.3 : 4th order cascaded biquad digital filter - time schedule after 3rd level multiplexing (cwl = 10 bits; swl = 23 bits)

Node	T_{in}	T_{out}	T_{needed}	Child nodes
3, 4, 0	-	0	0	-
x1b	0	1	1, 22	3, 3
x1m	0	1	1	0, 0
x1e	1	2	2	x1b, x1m
x1a	0	1	1, 20	4, 4
x1c	1	2	2	x1a, x1b
x2a	2	3	3	x1c, x1e
3a	-	1	1	-
x1h	1	2	2, 21, 22	3a, 3a
4a	-	0	0, 1	-
x1i	0	1	2, 22	4a, 4a
x1j	2	3	3	x1i, x1h
x3a	3	4	4	x2a, x1j
b1, g	-	1	1	-
x1f	1	2	2	b1, g
a1, b2	-	1	1	-
x1d	1	2	2	a1, b2
x2b	2	3	3	x1f, x1d
b3, b4	-	2	2	-
x1k	2	3	3	b3, b4
x3b	3	4	4	x2b, x1k
5x	4	21	21, 43^{*}, $46^{@}$, $88^{@*}$, $93^{@\#}$, $94^{@\#}$	x3a, x3b
5	0 (23^{*}, $46^{@}$)	1	1	$5y^{*}$, $5x^{@}$
x1g	1 ($93^{@\#}$)	2	2	5, $5x^{@\#}$
5y	2 ($94^{@\#}$)	3	3, 21, 22, 23^{*}	x1g, $5x^{@\#}$
1_d	3	3	-	5y
3_d	1	1	-	x1h
6z	19 ($88^{@*}$)	20	20	$5x^{@*}$, $5x^{@*}$
6	20	21	21	x1a, 6z
7	21	22	22	6, 5y
5ay	20 (43^{*})	21	21	$5x^{*}$, $5x^{*}$
5a	21	22	22	5ay, 5x
8	22	23	23	7, 5a
8_d	23	23	-	8
3a_d	2	2	-	x1h
6ax	21	22	22	x1h, x1h
6a	22	23	23	6ax, x1i
7a	23	24	24	6a, 8
7a_o	24	24	-	7a

* due to 1st level demultiplexer nodes 'L1...'
\# due to 2nd level demultiplexer nodes 'L2...'
@ due to 3rd level demultiplexer nodes 'L3...'

All the three designs at different multiplexing levels are possible by selecting the pair (5ax, 5az) initially at the 1st level of multiplexing. There are many pairs other than (5x, 1r) and (6x, 5z) that can be multiplexed concurrently with (5ax, 5az). Similarly there are 29 different pairs with which the multiplexing can be started at the 1st level. The BITSYN algorithm searches for all possible solutions in order to get a minimal set of designs for different sample frequencies. From this set of designs, optimal solutions are selected such that a solution at lower sample frequency is cheaper in gate count than the solution at higher sample frequency. Table 5.4a. gives the optimal solutions obtained for the 4th order cascaded biquadratic digital filter (CasBiq-4) digital filter. Here the *swl* of all the designs is equal to the respective minimum *swl* or system latency. The coefficient word length of all the multipliers is equal to 10 bits.

Table 5.4a : 4th order cascaded biquad digital filter optimal solutions

Design No.	Level of multipl-exing	No. of multi-pliers	System latency (bits)	Gate count	Normalized sample frequency	Sample frequency (KHz)*
1	0	6	21	5017	0.04762	952.4
2	1	3	22	3620	0.02273	454.5
3	1	3	23	3420	0.02174	434.8
4	2	2	22	3265	0.01136	227.3
5	2	2	23	3025	0.01087	217.4
6	3	1	23	2762	0.00725	144.9
7	3	1	23	2746	0.00543	108.7

swl = system latency = minimum swl; cwl = 10 bits
* bit clock frequency = 20 MHz;
cpu seconds on SPARC 1+ = 561.93;
Total number of solutions searched = 11520 (no heuristics);

The BITSYN program evaluated 11520 different design solutions before obtaining the seven optimal designs given in Table 5.4a. The search space is large due to the exhaustive search process. A number of heuristics conditions can be used to cut down the search space and in turn the computation time. The heuristics are implemented in five different levels which are defined in chapter 3. The effect of the different heuristics on the search space and the computation time for the CasBiq-4 digital filter is given in Table 5.4b.

Table 5.4b : 4th order cascaded biquad digital filter solutions : heuristics effect

Heuristic level#	No. of solutions searched	Search time (cpu seconds)	change in the optimal solutions (refer to the solutions in Table 5.4a)
0	11520	561.93	no heuristics are used
1	11520	561.93	no change in optimal solutions
2	11163	548.15	no change in optimal solutions
3	4016	192.99	no change in optimal solutions
4	3170	129.65	no change in optimal solutions
4 (p1=5%)*	2777	84.92	The gate count of the 7th design in Table 5.4a is changed. The new gate count = 2777 (increase of 31 gates).
5	2880	115.00	no change in optimal solutions
5 (p2=5%)*	1208	37.55	The gate counts of 6th and 7th designs in Table 5.4a are changed. The new gate count for the 6th design = 2781 (increase of 19 gates) and for the 7th design = 2751 (increase of 5 gates).

\# The heuristic levels are defined in chapter 3.

* "p1" and "p2" are the percentage limits used in 4th and 5th heuristics.

It was found (Table 5.4b) for this CasBiq-4 filter that the heuristics at the first, second and third levels do not change the optimal solutions. It can be seen that the first two heuristics do not improve the search time in this example. The exhaustive search finds the designs with minimum number of multipliers even though the designs are costly in terms of gates. The first two heuristics are mainly used to prune the search paths leading to the costlier solutions. It was found that the first two conditions do not affect the optimal solutions. In this example, each design is less costlier after multiplexing than the design from which it is derived. Thus the first two heuristics are ineffective. The second heuristic reduces the search time considerably. This heuristic removes the redundant search paths during the concurrent multiplexing. The 4th and 5th heuristics also give optimal or near optimal solutions with a considerable reduction in the search time.

BITSYN saves a set of minimal gate designs at different normalized sample frequencies. If the initial specifications such as the user specified *swl* or *cwl* are changed, the minimal set of designs can be re-computed with the new specifications. It has been observed that the re-evaluated designs will be optimal or near optimal solutions. Table 5.4c gives the designs for the CasBiq-4 filter with different *swl* and *cwl*. The optimal designs with 10 bit *cwl* and with *swl* equal to the system latency is already given in table 5.4a.

The minimal set of saved designs are recomputed to get new designs with the required *swl* of 32 bits. The SFG is resynthesized with user specied *swl* of 32 bits. It has been found that both methods result in same optimal solutions. Similarly the recomputed designs with the *cwl* equal to 12 bits have been found to be the same as the synthesized designs.

Table 5.4c : Modified optimal solutions with swl = 32, and with cwl = 12 for 4th order cascaded biquad digital filter.

Design No.	Level of multipl-exing	No. of multi-pliers	System latency (bits)	Gate count	Normalized sample frequency	Sample frequency (KHz)*
##						
1	0	6	21	5247	0.03125	625.00
2	1	3	24	4030	0.01563	312.50
3	2	2	23	3680	0.00781	156.25
4	3	1	26	3387	0.00521	104.17
5	3	1	26	3351	0.00391	78.13
@@						
1	0	6	24	5887	0.04167	833.33
2	1	3	25	4160	0.02000	400.00
3	1	3	26	3915	0.01923	384.62
4	2	2	25	3705	0.01000	200.00
5	2	2	26	3420	0.00962	192.31
6	3	1	26	3072	0.00641	128.21
7	3	1	26	3050	0.00481	96.15

swl = 32 and cwl = 10;
@@ cwl = 12 and swl = minimum swl = system latency;
* bit clock frequency = 20 MHz;

Thus BITSYN allows the user to change the specifications at the end of the synthesis process. The graphics interface allows the user to compare different designs and to select a design. Once the user has chosen a design, the design is synthesized using the saved design data such as the multiplier pairs and the order of multiplexing. After scheduling and data allocation, the FIRST language output is created. The filter design whose DFGs and the time schedule are given in Fig. 5.5 and Table 5.3 is generated in the FIRST language. The resulting FIRST language output is given in appendix B.

Once the design is generated in the FIRST language, it can be simulated using the simulator SIMFIRST [3]. The netlist translator developed at the Department of Electrical and Computer Engineering, University of Calgary is used to assemble the FIRST language description into the netlist format suitable for field programmable gate array (FPGA), XILINX [14]. The different steps involved in a bit-serial implementation from FIRST code are illustrated in Fig 5.6a. The netlist translator is used to convert FIRST code into a XILINX netlist. The XILINX demonstration setup shown in Fig. 5.6b is used to implement the design. The CasBiq-4 design was successfully implemented and tested in a XILINX FPGA.

5.2 Second order Wave Digital Filter (Wave-2)

A variety of designs generated using BITSYN for the second order wave digital filter (signal flow graph in Fig. 2.1a) have been tabulated in Table 5.5. The cwl has been set to 12 bits (signed) and various values for the swl have been selected. Note that there is a significant reduction in the gate count for the designs where two or fewer multipliers are used. The designs are generated initially without specifying a particular *swl*. These designs marked with "m" have *swl* equal to the system latency because the filter structure is recursive. The 6th design in the table is not optimal for *swl* of 25 bits, because the 5th design is cheaper and has higher sample frequency. The 6th design is part of minimal set of designs that was saved in the design space.

Next the designs are generated with a user specified *swl* of 32 bits. The designs can be generated using two different methods. In the first method, the previously calculated minimum set of designs are recomputed with the new *swl* specification. These designs are

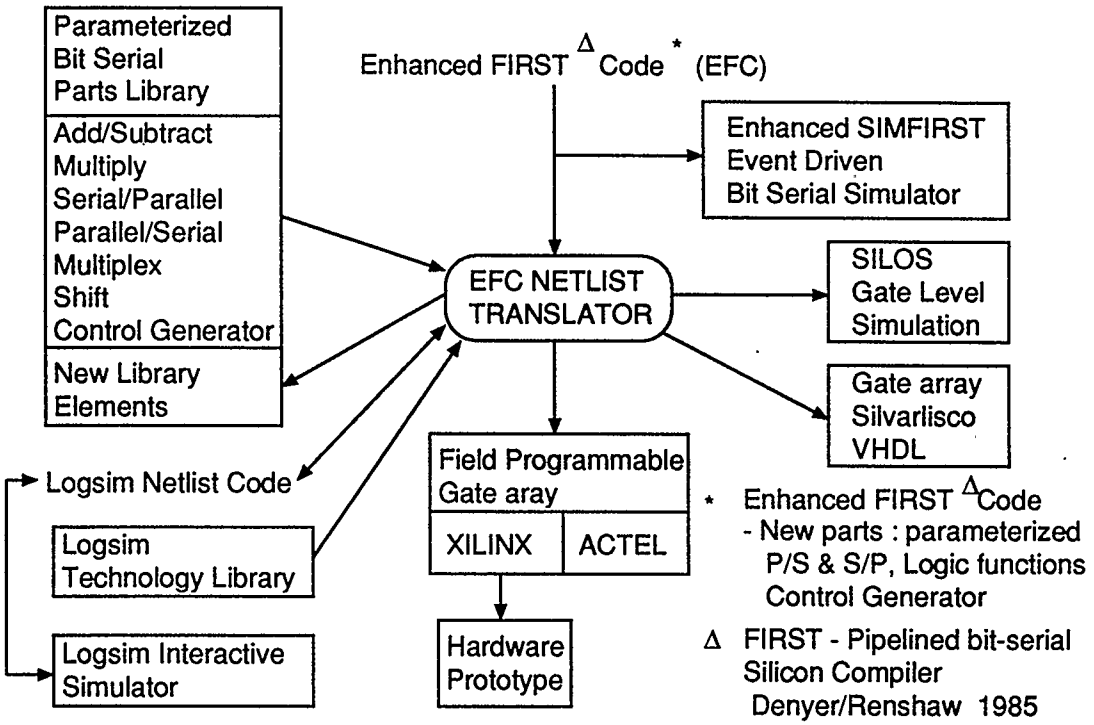


Fig. 5.6a : Different steps involved in the bit-serial implementation

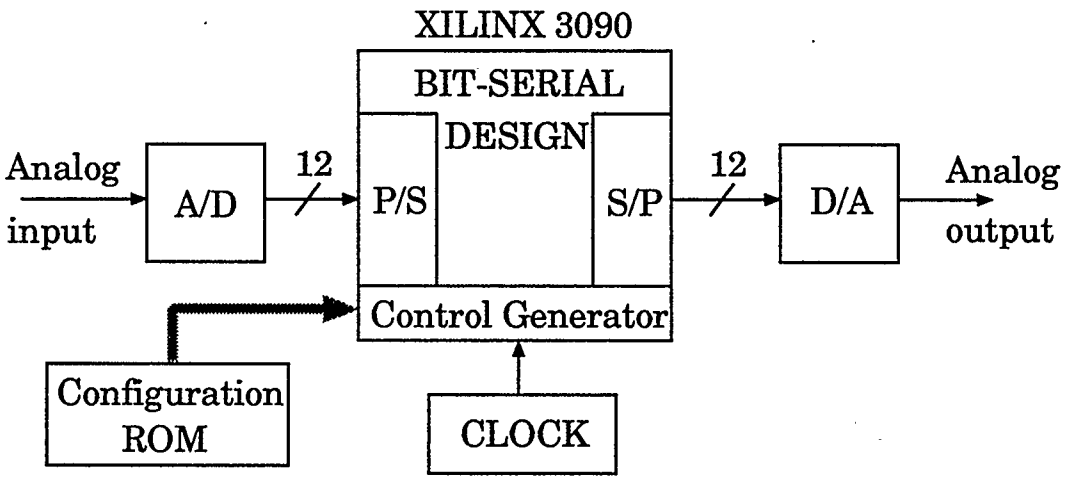


Fig. 5.6b : Block diagram of XILINX demonstration setup

identified with "c". In the second method the SFG is resynthesized with the new specifications. The designs obtained in the second method are identified with "s".

Table 5.5 : 2nd order wave digital filters designed using BITSYN

Design No.	Level of multipl-exing	No. of multi-pliers	System latency (bits)	Gate count	Normalized sample frequency	Sample frequency (KHz)*
1 m	0	5	25	4632	0.0400	800.0
1 c s	0	5	25	4977	0.0313	625.0
2 m	1	4	25	4075	0.0200	400.0
2 c	1	4	25	4355	0.0156	312.5
3 m	1	3	26	3485	0.0192	384.6
3 c s	1	3	26	3990	0.0156	312.5
4 m	2	2	24	2715	0.0104	208.3
4 c s	2	2	24	3185	0.0078	156.3
5 m	3	1	25	2444	0.0050	100.0
5 c	3	1	25	2924	0.0039	78.1
6 m	3	1	28	2458	0.0045	89.3
6 c s	3	1	28	2658	0.0039	78.1

m swl = system latency = minimum swl; cwl = 12 bits;
c swl = 32; the designs are recomputed;
s swl = 32; the designs are resynthesized;
* bit clock frequency = 20 MHz;
cpu seconds on SPARC 1+ = 5.78 (no heuristics);
Total number of solutions searched = 226 (no heuristics);

Note that when the *swl* is changed to 32 bits, the previously non-optimal design (6th design in the table) becomes a useful solution and the previously optimal design (5th design in the table) becomes redundant. The reason is the 6th design has a minimum *swl* of 28 bits where as the 5th design has 25 bits minimum *swl*. To achieve a 32 bit *swl*, a number of delay units equal to the difference of *swl* and minimum *swl*, must be inserted in all the

closed loop signal paths of the designs. Thus the 6th design requires less delay units than the 5th design. The 2nd design becomes redundant when the *swl* is changed to 32 bits. Note that the 2nd and the 5th designs do not appear during the resynthesis process. This demonstrates that recomputation of designs may not give all the solutions when the *swl* specification is decreased.

5.3 Second order Direct Form Digital Filter (Direct-2)

In table 5.6 the design of a second order biquadratic direct form digital filter (signal flow graph in Fig 2.14a) has been evaluated using BITSYN. This filter has a 12 bit (signed) *cwl* and the *swl* is determined by the critical path latency or system latency. Note that this filter can be used to implement the same biquadratic transfer function as the Wave-2 digital filter. For level 0 multiplexing, this direct form design requires fewer gates and has a slightly higher sample frequency as compared to Wave-2 digital filter (due to the lower critical path latency). For multiplexing level three (one multiplier) the wave filter design requires somewhat fewer gates.

Even though both filters have the same number of multiplication operations, the number of solutions searched (with no heuristics) is equal to 968 in the case of Direct-2 as compared to 226 in the case of Wave-2. The size of the search space is dependent on the connectivity between the different multipliers. In the case of Direct-2 digital filter, the multipliers are independent of each other and there exists a larger number of search paths for concurrent multiplexing. It can be observed that the sample frequencies of the 5th and 6th designs are approximately one fifth the sample frequency of the five multiplier solution.

That is, these designs with a single multiplier perform five multiplication operations in 5 cycles to obtain an output sample. Since the designs are multiplexed at the 3rd level, maximum eight (2^n, n = 3) cycles can be used to compute an output sample. Thus there exists timing redundancy with three redundant cycles in these designs. Similarly it can be observed that the 7th and 8th solutions need all eight cycles to compute an output sample.

Table 5.6 : Second order direct-form digital filters designed using BITSYN

Design No.	Level of multipl-exing	No. of multi-pliers	System latency (bits)	Gate count	Normalized sample frequency	Sample frequency (KHz)*
1	0	5	23	4352	0.0435	869.6
2	1	3	24	3320	0.0208	416.7
3	2	2	23	2955	0.0109	217.4
4	2	2	24	2740	0.0104	208.3
5	3	1	25	2356	0.0080	160.0
6	3	1	26	2311	0.0769	153.8
7	3	1	25	2301	0.0500	100.0
8	3	1	26	2291	0.0048	96.2

swl = system latency = minimum swl; cwl = 12 bits;
* bit clock frequency = 20 MHz;
cpu seconds on SPARC 1+ = 23.91 (no heuristics);
Total number of solutions searched = 968 (no heuristics);

5.4 Fifth order LDI Digital filter (LDI-5)

The BITSYN program has been used to analyze designs of a fifth order Bilinear LDI (elliptic) low pass digital filter [26]. The signal flow graph of the LDI-5 digital filter is shown in Fig. 5.7. Designs generated for this low sensitivity ladder filter structure using BITSYN

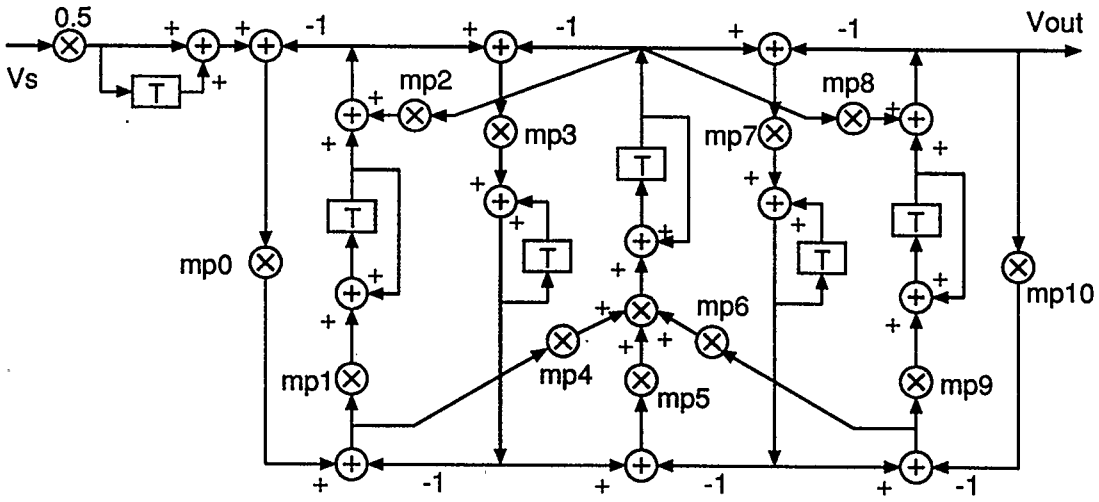


Fig. 5.7 : 5th order bilinear LDI digital filter signal flow graph

Table 5.7 : 5th order bilinear LDI digital filers designed using BITSYN

Design No.	Level of multipl-exing	No. of multi-pliers	System latency (bits)	Gate count	Normalized sample frequency	Sample frequency (KHz)*
1	0	9	68	12043	0.01476	294.12
2	1	6	48	9290	0.01042	208.33
3	1	6	49	9060	0.01020	204.08
4	2	5	26	6876	0.00962	192.31
5	2	5	27	6496	0.00926	185.19
6	3	3	26	6064	0.00481	96.15
7	3	3	27	5539	0.00463	92.59
8	4	2	28	5958	0.00223	44.64

swl = system latency = minimum swl; cwl = 12 bits
* bit clock frequency = 20 MHz;
cpu seconds on SPARC 1+ = 4964 (heuristics at the 2nd level);
Total number of solutions searched = 82987 (heuristics at the 2nd level);

are given in Table 5.7. Note that in this particular filter architecture the implementation which requires the fewest gates is the one in which three multipliers are used. For designs with fewer than three multipliers, the cost of the extra registers required to delay signals is larger than the savings in the gate count due to the sharing of multiplier operators. The design without any multiplexing has a system latency equal to 68. Since the structure is recursive in nature, the *swl* can not be less than 68. In general such a large system word length is not necessary. Thus there is an advantage to multiplex the multipliers in this recursive structure.

The multiplexing strategy reduces the critical path length and in turn the system latency. Thus the designs with considerable reduction in their hardware cost can be achieved without comparable reduction in their sample frequencies. The 5th design in the table is such a design. The hardware cost of this design is reduced by 46% as compared to that of 9 multiplier design where as the sample frequency is reduced only by 37% as compared to the 9 multiplier design. The 2nd level heuristics are used to obtain the designs. As the level of the heuristics increases, some of the designs are no longer optimal. The exhaustive search can not performed in a reasonable amount of time for this design which is more than 2 days of cpu time.

5.5 Fifth order Wave Digital Filter (Wave-5)

The BITSYN program has also been used to evaluate designs for a fifth order wave digital low pass (elliptical) filter (Wave-5) which has been studied extensively in high level synthesis [19] [34]. The signal flow graph of the Wave-5 filter is shown in Fig. 5.8. In

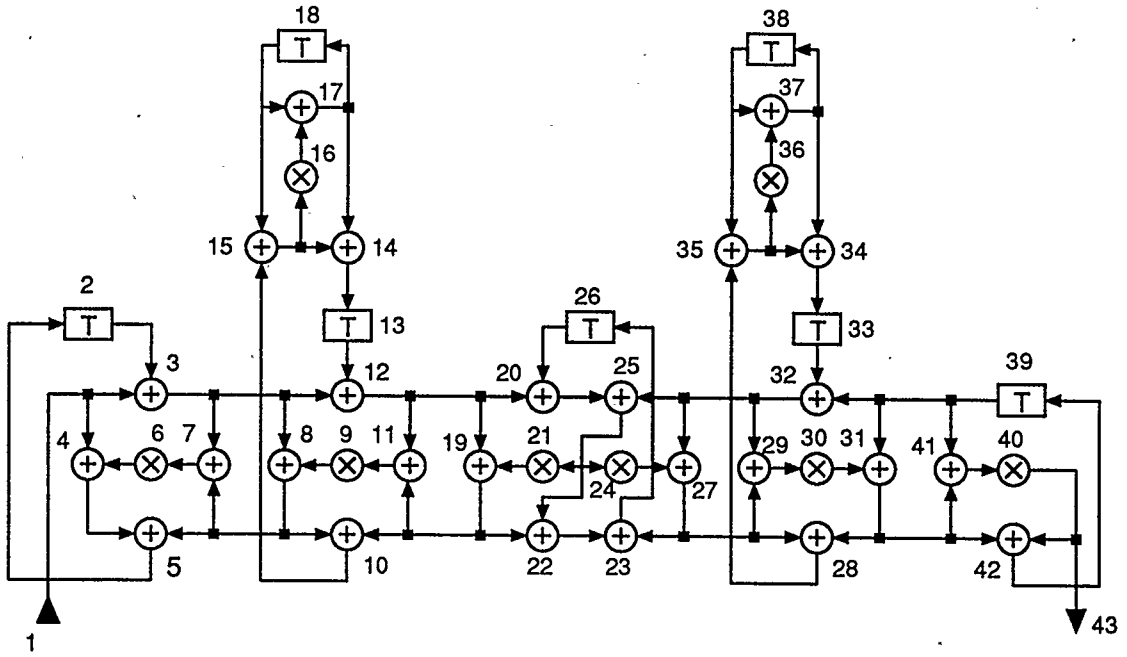


Fig. 5.8 : 5th order elliptic wave digital filter signal flow graph

Table 5.8 : 5th order wave digital filers designed using BITSYN

Design No.	Level of multipl-exing	No. of multi-pliers	System latency (bits)	Gate count	Normalized sample frequency	Sample frequency (KHz)*
1	0	8	71	11563	0.01408	281.69
2	1	7	50	10930	0.01000	200.00
3	1	6	51	10690	0.00980	196.08
4	2	5	52	9935	0.00962	192.31
5	2	5	28	8411	0.00893	178.57
6	2	4	30	6986	0.00833	166.67
7	3	2	29	5883	0.00446	89.29
8	3	2	28	5763	0.00431	86.21
9	4	1	31	7386	0.00201	40.32

swl = system latency = minimum swl; cwl = 12 bits
* bit clock frequency = 20 MHz;
cpu seconds on SPARC 1+ = 291.39 (heuristics at the 2nd level);
Total number of solutions searched = 1985 (heuristics at the 2nd level);

Table 5.8 the design alternatives for this filter are provided. This filter implements the same class of transfer functions as the 5th order LDI filter. Note that the smallest design is larger than that of the LDI but has similar *swl* and sample rate specifications. The designs are synthesized using the heuristics at the 2nd level. Even though this structure has 8 multipliers the number of designs searched are 1985. This is due to the fact that most of the multipliers are interdependent.

5.6 Sine/ Cosine Function Generator

A number of recursive digital filter implementations have been evaluated and contrasted using BITSYN. A sine/cosine function generator is designed using BITSYN. It does not have any states unlike digital filters.

The sine and cosine functions can be expressed in terms of trigonometric series as

$$sin(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - .....$$

$$cos(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - .....$$

The first four terms in each series are used to implement an approximation to the sine and cosine functions. The corresponding SFG is shown in Fig. 5.9. There are eight constant coefficient (fractional coefficient) multipliers and eight signal coefficient multipliers. In this example the coefficient word length of the signal coefficient multipliers will determine the dynamic range instead of the system word length. The coefficient multiplier pairs can not be multiplexed concurrently because the multipliers are dependent on each other.

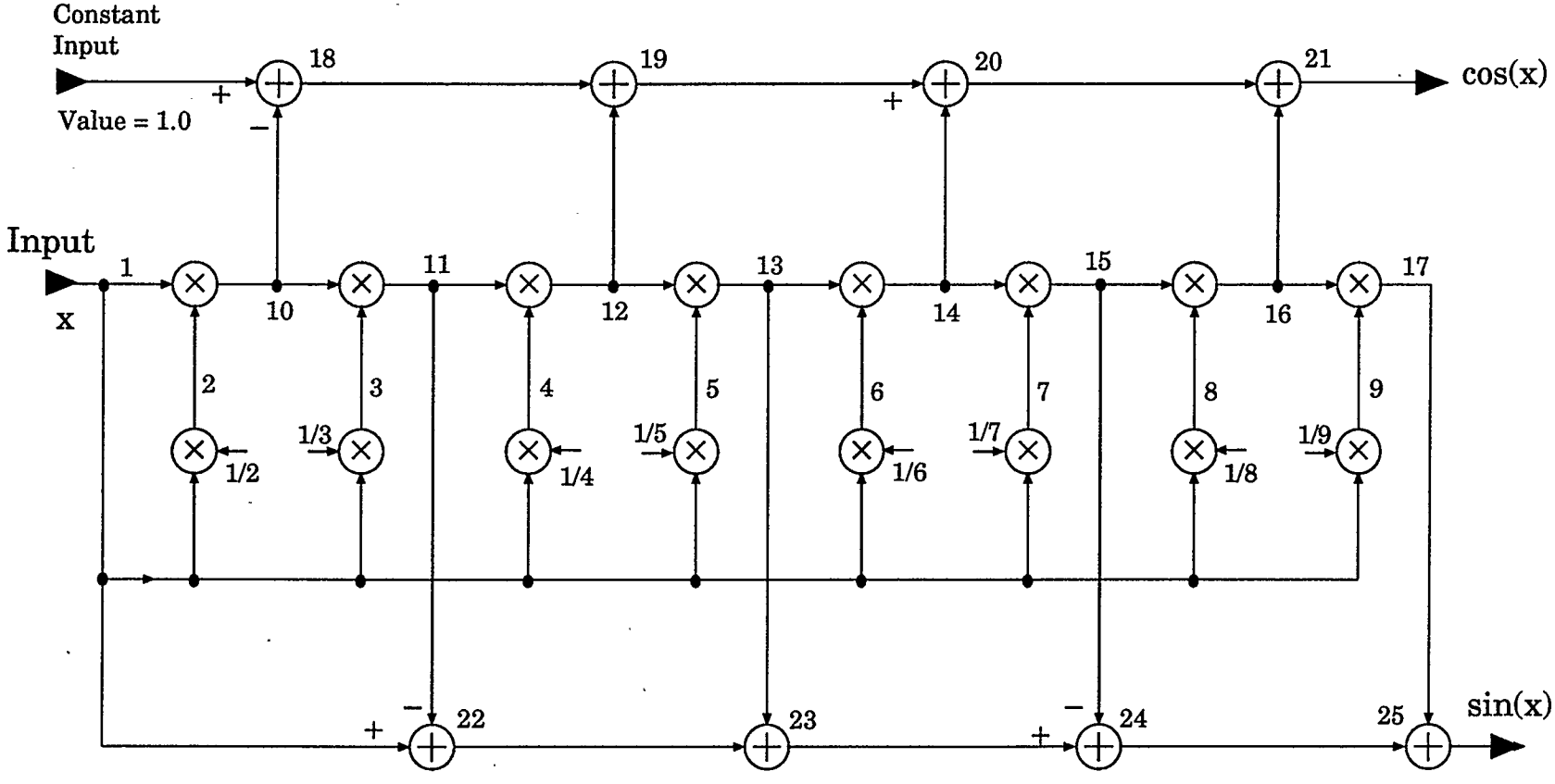


Fig. 5.9 : Signal flow graph of sine/cosine function generator

If the number of multipliers are small, the designs can be obtained in a reasonable amount of time. The different heuristic conditions discussed in chapter 3 perform well, when the number of multiplier pairs that can be multiplexed concurrently are small. The heuristics use the gate count and sample frequency information to prune the multiplexing paths. In this example, the constant coefficient multipliers (eight in number) are independent of each other and form 56 multiplier pairs which can be multiplexed concurrently at the first level. The constant coefficient multipliers can be reduced to four multipliers in 40320 (56*30*12*2) ways at the first level of multiplexing. The heuristics do not perform well if the large number of designs have similar gate counts and sample frequencies.

One way of reducing search space is to select manually a very few (one or two) multiplier pairs at the first one or two levels of multiplexing. The BITSYN program can then be used perform the multiplexing at the higher levels. Fortunately this problem of large search space occurs with the simple structures such as FIR filter and this example, where the user can group the multipliers and specify the multiplexing pairs easily as compared to complicated recursive structures. This manual intervention is avoided by implementing a similar concept in the program itself. Thus a special pruning (heuristic) is introduced for simple structures such as FIR filters where the multipliers are independent. Here the concurrent multiplexing will start always with a pair instead of starting with all the pairs. The position of the multipliers in the tree structure is used to select the pair. The multiplier nodes at bottom most levels of the tree are multiplexed first. The reason is that the multiplier pair does not interfere with any other multipliers. This type of pruning is used at all levels.

This special heuristic is also tested with FIR filters This heuristic generate designs with fewer gates but does not give a range of optimal solutions. Table 5.9 gives the different designs obtained by BITSYN. The design with the fewest gates contains only 2 multipliers and the design is obtained at the 3rd level of multiplexing. The number of gates in the fewest gate design are 3641, where as the operator-parallel design consume 16782 equivalent gates. Since the original SFG is a non-recursive structure, the *swl* of an operator-parallel design is not limited by the system latency. All other designs have a minimum *swl* limit which is equal the system latency or the length of the closed loops inside the design.

Table 5.9 : Sine/ cosine function generator designs obtained using BITSYN

Design No.	Level of multipl-exing	No. of multi-pliers	System latency (bits)	Gate count	Normalized sample frequency	Sample frequency (KHz)*
1	0	16	1	16782	1	20000/required swl
2	1	11	18	11065	0.02277	555.56
3	1	10	34	10440	0.01471	294.12
4	2	8	34	8815	0.00735	147.01
5	3	6	19	7321	0.00657	131.00
6	3	2	37	3641	0.00338	67.567

swl = system latency = minimum swl;
cwl of signal coefficient multipliers = 16 bits; default cwl = 10;
* bit clock frequency = 20 MHz;
cpu seconds on SPARC 1+ = 19.20
Total number of solutions searched = 148 (special pruning and 2nd level heuristics);

Chapter 6

CONCLUSION

The goal of this thesis has been the development of an architectural synthesis tool for the design of pipelined bit-serial digital signal processing (DSP) systems. A synthesis tool named BITSYN (BIT-Serial SYNthesis) has been developed. The BITSYN tool evaluates the resource sharing strategies (multiplexing) and generates a variety of pipelined bit-serial DSP systems. The objective of the synthesis process is to find a bit-serial structure that best meets the constraints of clock rate, area or size, and system word length while achieving goals such as maximizing the sample rate and minimizing the size.

The following sections summarize the contributions presented in this thesis and discuss possible areas for further research.

6.1 Contributions of the Thesis

A very few high level design tools for the bit-serial systems have been reported in the literature [7] [24]. These design tools translate a behavioral description into a single bit-serial design structure and some of these tools use optimization techniques to reduce the number of delay units required in bit-serial designs. In contrast, BITSYN employs a resource sharing or multiplexing strategy and generates different pipelined bit-serial designs in a three dimensional design space. This design space includes the size (gate count), sample rate and minimum system latency (minimum *swl*). BITSYN relieves the designer of

the involved and tedious process of scheduling complicated recursive DSP algorithms for pipelined bit-serial implementations. BITSYN accepts as input a compact, high level, signal flow graph (SFG) description which is similar to the SFG description used in DIGICAP [23] and generates output in the FIRST [3] language.

The major steps involved in the synthesis process are discussed in chapter 2. In BITSYN, the behavioral input is represented in the form of data flow graphs (DFG). The resource sharing process is used to perform multiplexing. The resource sharing algorithm modifies the DFGs by inserting multiplexer and demultiplexer nodes. The resource sharing process starts with multiplier pairs and reduces the number of multipliers as the process continues. The concept of concurrent multiplexing is introduced. During concurrent multiplexing, two or more multiplier node pairs can be shared. The scheduling and delay allocation strategies are developed in such a way that the number of the delay units required is minimized.

Chapter 3 discusses the implementation of the BITSYN algorithm. The algorithm is based on the exhaustive search of all possible multiplexed designs. The total number of search solutions is based on the number of multiplier nodes and not the total number of nodes in the DFGs. As the total number of search solutions increases exponentially with the number of multiplier nodes, five different heuristics have been proposed to obtain optimal or near optimal designs within a reasonable time.

Chapter 4 describes the design of a FIRST control generator and the generation of the different control signals required by the systems produced using BITSYN. The control signals can be generated using finite state machines (FSM) or linear feedback shift

registers (LFSR). After comparing both methods, it was found that a FSM circuit is cheaper than a LFSR circuit while designing a LFSR circuit is much simpler than designing of a FSM.

Digital filter designs have been evaluated and contrasted in chapter 5. The different features of the BITSYN algorithm are discussed using the design examples. Designs obtained from BITSYN have been implemented using XILINX [14] field programmable gate arrays. BITSYN allows the designer to choose different designs with a variety of different hardware costs, sample rates and system word lengths.

6.2 Suggestions for Further Work

BITSYN has been developed for digital filter applications and it is compatible with the filter analysis program DIGICAP [23]. A selective set of functional operators containing two or less inputs and one output, are supported by BITSYN. Generic operators also containing two or less inputs and one output are introduced such that they can simulate any new functional operator. If a functional operator has more than one output, it can be indirectly used in BITSYN. That is, a functional operator with 'm' outputs will be represented in the SFG by 'm' different generic operators each containing one output and the same inputs. After the design is synthesized, all 'm' operators with same data inputs and control inputs can be merged into one operator in the FIRST language output. The problem with this indirect method is that those 'm' operators should not be multiplexed. When a functional operator has more than two inputs the BITSYN does not support that functional operator at all. In future, the synthesis program should support multi-output and multi-input functional

operators so that it can be used in a wide area of DSP applications.

The BITSYN algorithm is based on an exhaustive search. As the number of multipliers increases the search space and search time increases exponentially. The heuristics suggested in chapter 3 can be used to reduce the search space and time to some extent. The search time is also depend on the connectivity between different multipliers. If they are all independent, then the search time increases since the number of combinations for concurrent multiplexing increases. Optimization techniques, such as simulated annealing, should be considered in future to reduce the search time. Simulated annealing may be useful if the user want to find a near-optimal design which meets certain constraints such as gate count and sample frequency.

REFERENCES

[1] James V. Candy, *Signal Processing : The Model-Based Approach,* Reading, New York: McGraw-Hill Inc., 1986.

[2] L.R. Rabiner and Bernard Gold, *Theory and Applications of Digital Signal Processing,* Reading, Englewood Cliffs, NJ: Prentice-Hall, 1975.

[3] P. Denyer and D. Renshaw, *VLSI Signal Processing : A Bit-Serial Approach,* Reading, Massachusetts: Addison-Wesley, 1985.

[4] L.B. Jackson, *Digital Filters and Signal Processing,* Kulwer publishers, 1989.

[5] R. Hartley and P. Corbett, "Digit-Serial Processing Techniques," *IEEE Trans. on Circuits and Systems*, Vol. 37, June 1990.

[6] Keshab K. Parhi, "A Sytematic Approach for Design of Digit-Serial Signal Processing Architectures," *IEEE Trans. on Circuits and Systems*, Vol. 38, no. 4, April 1991.

[7] Rajeev Jain and et al, "Custom Design of a VLSI PCM-FDM Transmultiplexer from System Specifications to Circuit Layout Using a Computer-Aided Design System," *IEEE Journal of Solid-State Circuits*, Vol. 21, no. 1, pp. 73-85, February 1986.

[8] R.F. Lyon, "A Bit-Serial VLSI Architectural Methodology for Signal Processing," *International Conference on VLSI, VLSI 81*, Academic Press, 1981.

[9] L.B. Jackson, J.F. Kaiser, and H.S. McDonald, "An Approach to the Implementation of Digital Filters," *IEEE Trans. on Audio and Electroacoustics*, Vol. vol. AU-16, no. 3, pp. 413-421, 1968.

[10] N.W. Bergmann, "A Case Study of the FIRST Silicon Compiler," *3rd Caltech Conference on VLSI*, pp. 413-430, Computer Science Press, 1983.

[11] M. McFarland, A. C. Parker, and R. Camposano, "Tutorial on High-Level Synthesis," *25th ACM/IEEE Design Automation Conference*, pp. 330-336, 1988.

[12] *VTI tools : Refernce manuals*, VLSI Technology, Inc., 1989.

[13] *Cadence tools : Reference manuals*, Cadence Design Systems, Inc., 1989.

[14] *The programmable Gate Array data book* , XILINX, Inc., 1989.

[15] P. Paulin, J. Knight, and E. Gyrczyc, "HAL: A Multi-Paradigm Approach to Automate Data Path Synthesis," *Proc. 23rd Design Automation Conference*, pp. 263-270, 1986.

[16] N. Park and A. Parker, "SEHWA : A Software Package for Synthesis of Pipelines from Behavioral Specifications," *IEEE Trans. on Computer-Aided Design*, Vol. 7, pp. 356-370, March 1988.

[17] M.McFarland, "Using Bottom Up Design Techniques in Synthesis of Digital Hardware from Abstract Behavioral Descriptions," *Proc. 23rd Design Automation Conference*, pp. 474-480, 1986.

[18] A. Parker, J.Pizarro, and M. Mlinar, "MAHA : A Program for Data Path Synthesis," *Proc. 23rd Design Automation Conference*, pp. 461-466, 1986.

[19] B. S. Haroun and M. I. Elmasry, "Architectural Synthesis for DSP Silicon Compilers," *IEEE Trans. on Computer-Aided Design*, Vol. 8, pp. 431-447, April 1989.

[20] C. Tseng and D.P. Siewiorek, "Automated Synthesis of Data Paths in Digital Systems," *IEEE Trans. on Computer-Aided Design*, Vol. 5, no. 3, pp. 379-395, July 1986.

[21] H. Trickey, "Flamel: A High-Level Hardware Compiler," *IEEE Trans. on Computer-Aided Design*, Vol. 6, no. 2, pp. 259-269, March 1987.

[22] P.G. Paulin and J.P. Knight, "Force-Directed Scheduling in Automatic Data Path Synthesis," *24th ACM/IEEE Design Automation Conference*, pp. 195-202, 1987.

[23] L. E. Turner , D. A. Graham, and P. B. Denyer, "The Analysis and Implementation of Digital Filters using Special Purpose CAD Tool," *IEEE Trans. on Education*, Vol. 32, pp. 287-297, August 1989.

[24] Richard I. Hartley and Jeffrey R. Jasica, "Behavioral to Structural Translation in a Bit-Serial Silicon Compiler," *IEEE Trans. on Computer-Aided Design*, Vol. 7, no. 8, pp. 877-886, August 1988.

[25] R. Nagalla and L.E. Turner, "Pipelined BIT-serial SYNthesis of Digital Filtering Algorithms," *International Conference on VLSI, VLSI 91*, Edinburgh, August 1991.

[26] L. E. Turner and B. K. Ramesh, "Low Sensitivity Digital LDI Ladder Filters with Elliptic Magnitude Response," *IEEE Trans. on Circuits and Systems*, Vol. 33, pp. 697-706, July 1986.

[27] K. Meerkotter and W. Wegener, "A New Second-Order Digital Filter without Parasitic Oscillations," *Arch. Elektronik & Ubertragungstech.*, Vol. 29, pp. 312-314, 1975.

[28] Wesley W. Peterson and E.J. Weldon, Jr, *Error-Correcting Codes,* MIT Press Inc., 1961.

[29] Rodger E. Ziemer and Roger L. Peterson, *Digital Communications and Spread Spectrum Systems,* Reading, New York: Macmillan Inc., 1985.

[30] Zvi Kohavi, *Switching and Finite Automata Theory,* Reading, New York: McGraw-Hill Inc., 1978.

[31] *Databook and Design Manual - HCMOS Macrocell Manual,* LSI Logic Corporation, 1986.

[32] G. Hamachi, *Peg Tutorial,* VLSI Tools, University of California, Berkeley, 1984.

[33] R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis,* Kluwer Academic Publishers, 1984.

[34] R. Potasman, J. Lis, A. Nicolau, and D. Gajski, "Percolation based Synthesis," *27th ACM/IEEE Design Automation Conference,* pp. 444-449, 1990.

[35] M. R. Smith and L. E. Turner, *EPLOT graphical interface language,* Dept. of ECE, University of Calgary, Sept, 1988.

APPENDIX A

IMPLEMENTATION OF THE BITSYN ALGORITHM

This appendix discusses the implementation of the BITSYN algorithm in detail. The BITSYN algorithm has been implemented in a **C** language program running on a Sun SPARC CPU.

Fig. A.1 gives a flow diagram of the BITSYN algorithm. BITSYN starts by reading the input specifications such as coefficient word length (cwl) of a multiplier and user specified system word length (swl). These specifications are optional inputs. If *cwl* is not specified the default *cwl* is equal to 4. The constraints such as maximum number of gates and the minimum sample frequency can be specified optionally. They can be used to limit the range of the exhaustive search. The technology to implement the design is also specified here.

The module *read_SFG_input* is used to read the signal flow graph (SFG) description. If there are any syntax errors in SFG netlist, this module reports the errors to the user. The translation from the SFG netlist to the data flow graphs (DFG) is carried out by the module *make_DFG*. Once the DFGs are created in the form of binary structures, the parameters such as **ident** and **child** of each node are updated.

The module *form_mult_pairs* is used to form a list of pairs of the multiplier nodes or the generic nodes that are large in size. These pairs will be used in resource sharing process. If there are *m* similar type of multiplier nodes, the number of pairs is equal to *m*(m-1)*.

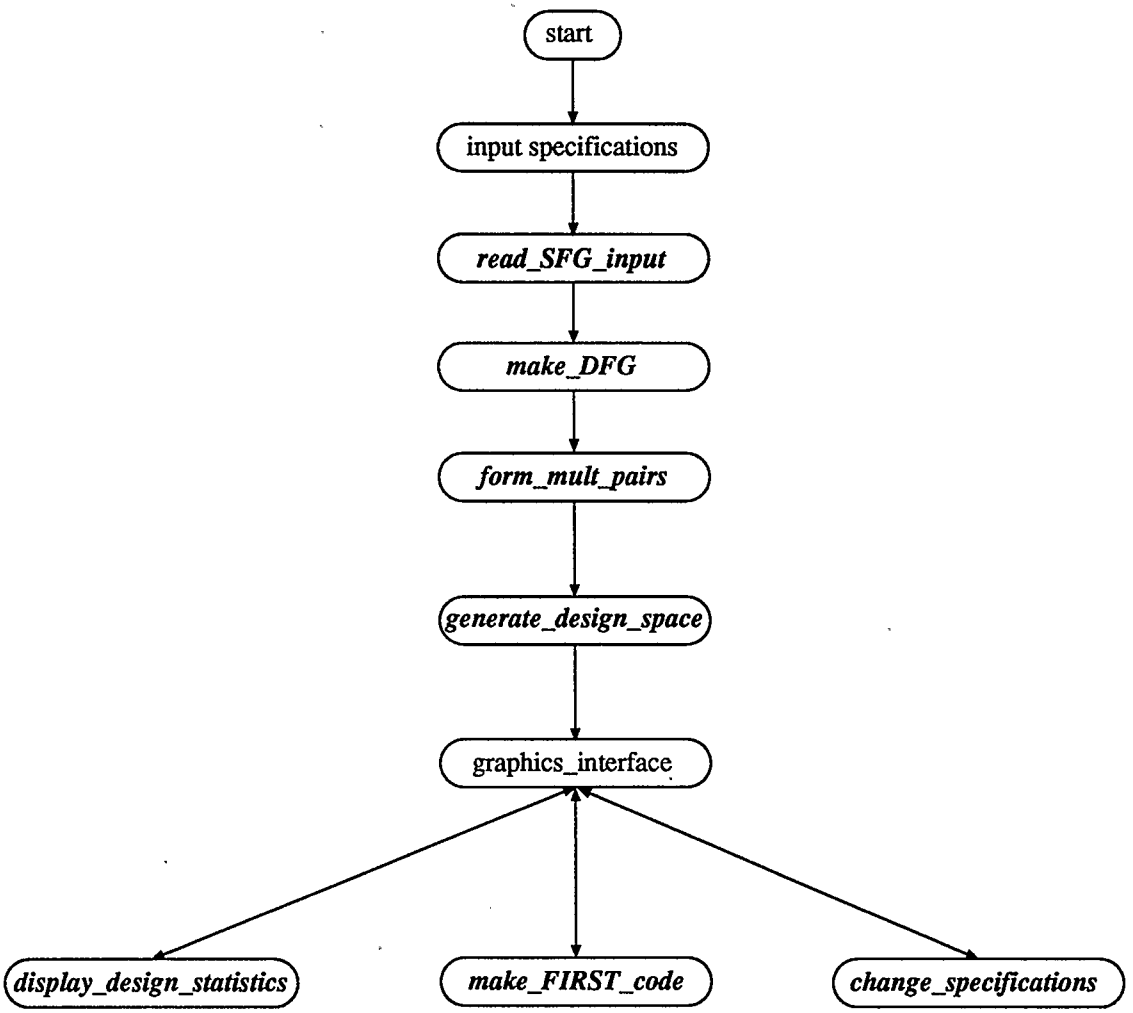


Fig. A.1 : Flow diagram of the BITSYN algorithm

The module *generate_design_space* uses all three steps such as resource sharing, scheduling and allocation and generates a number of designs with different sample frequencies. An operator-parallel solution without any multiplexing is generated using scheduling and allocation steps. Then all the designs are displayed using *graphical_interface*. The functions of the modules *generate_design_space* and *graphical_interface* are explained in the following sections. The first step in *generate_design_space* is to analyze each multiplier pair using the module *analyze_pairs*.

A.1 Analyze multiplier pairs.

The flow diagram illustrating the function of *analyze_pairs* is given in Fig. A.2. This module identifies all the invalid pairs. A pair becomes invalid in two different situations. If the second node is a child of the first node in a pair, then the pair is considered as invalid. If the algorithm is in the middle of concurrent multiplexing (CM) and a pair does not meet the conditions for CM, then the pair is treated as invalid. The principle of the concurrent multiplexing is explained in chapter 2. If a pair is a valid one, resource sharing (multiplexing) explained in chapter 2 is performed using the module called *make_mux*. The resource sharing process modifies the DFGs. If a pair can be concurrently multiplexed (multiplexed at the same level) with any other pair, then that pair will be considered as part of concurrent multiplexing (CM). If the pair is valid and is part of CM, no processing is performed after multiplexing.

If a pair is valid and not part of CM, then the multiplexed DFGs can be scheduled using *scheduling* module. This step consists of both forward scheduling and backward

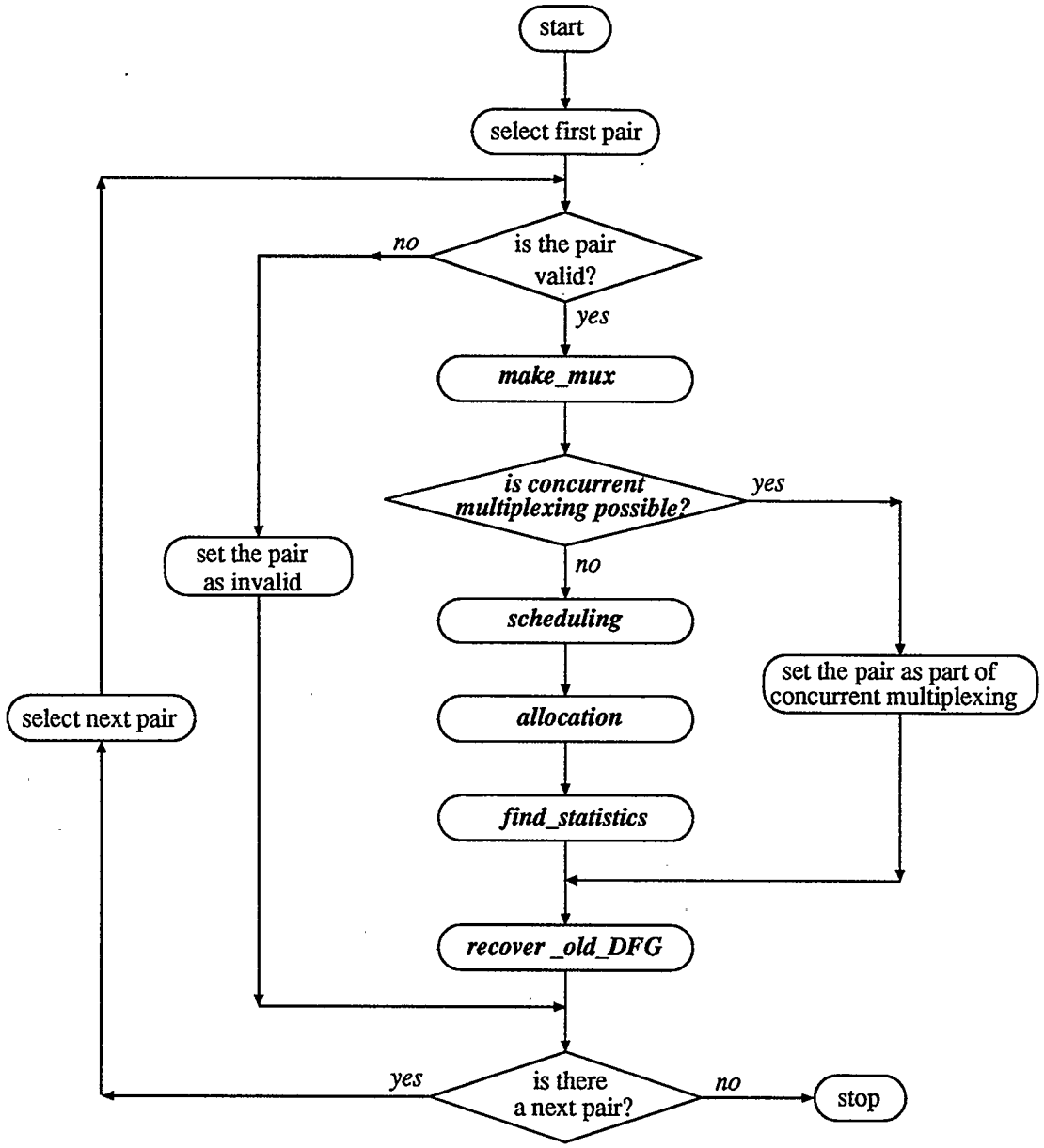


Fig. A.2 : Flow diagram of the module *analyze_pairs*

scheduling. The system latency will be obtained at the end of this step. In recursive or feed back structures, the system word length will be equal to the maximum of the user specified *swl* and the system latency. After the swl is decided, the *allocation* module replaces the nodes with equivalent functional operators and introduces delay units in appropriate places. The scheduling and allocation steps are explained in chapter 2.

The next step is to find the design characteristics using the module *find_statistics*. Here, the number of different types of operators such as multipliers, adders, registers, will be counted. The user specifies the number of equivalent gates required to implement each operator in different technologies of interest. BITSYN uses the LSI [32] logic gate array technology and the XILINX [14] programmable gate array technology.

The total number of equivalent gates (gate count) is calculated as the sum of the gates for each and every operator needed to implement the design in a particular technology of interest. If there are no redundant cycles among 2^n (where n is the multiplexing level) cycles of c00 (LSB time) then the normalized sample frequency (nsf) for the design is equal to $1/(2^n * swl)$. But, if some of the 2^n cycles of c00 are redundant then the *nsf* is equal to 1/(period of cn0 in bits * swl). The sample frequency of a design can be found by multiplying the *nsf* with the bit clock frequency. Thus the size and the speed of a design is found in this step. The design characteristics such as gate count, normalized sample frequency and the number of different types of operators are stored as the parameters of the multiplier pair.

The original DFG should be recovered to analyze another pair. This is done using module *recover_old_DFG*. Before multiplexing, a copy of the DFGs is stored at each level.

The *recover_old_DFG* removes all the multiplexers and demultiplexers that are inserted during the resource sharing process and recovers the old DFGs. Thus a pair is analyzed and is classified as an invalid pair or as a pair that is part of CM or as a pair that can produce a multiplexed design with a given characteristics.

A.2 Generation of design space

The flow diagram illustrating the algorithm of *generate_design_space* module is given in Fig. A.3. This module performs multiplexing at different levels. The first module *analyze_pairs* is used to characterize the multiplier pairs. Then the pairs are sorted in ascending order of their gate count. All the invalid pairs are placed at the end of the list. The pairs which are part of CM are placed at the beginning of the list. This sorting is useful when heuristics are used to cut down the search space. The invalid pairs are kept at the end of the list so that they can be ignored.

The synthesis steps used here are similar to the ones used in module *analyze_pairs*. A pair will be multiplexed using the module *make_mux*. here the multiplexers and the demultiplexers are introduced into the DFGs.

If a pair is not a member of CM, or if a pair is the last member of CM (no further CM), then the design along with its characteristics is saved using the module *save_design_statistics*. The design does not have to be analyzed again. The designs are saved in three dimensional space. The three dimensions are normalized sample frequency (nsf), gate count and the system latency. The function of *save_design_statistics* is explained in the *design space* section of chapter 3. After saving the design, multiplexing at the next

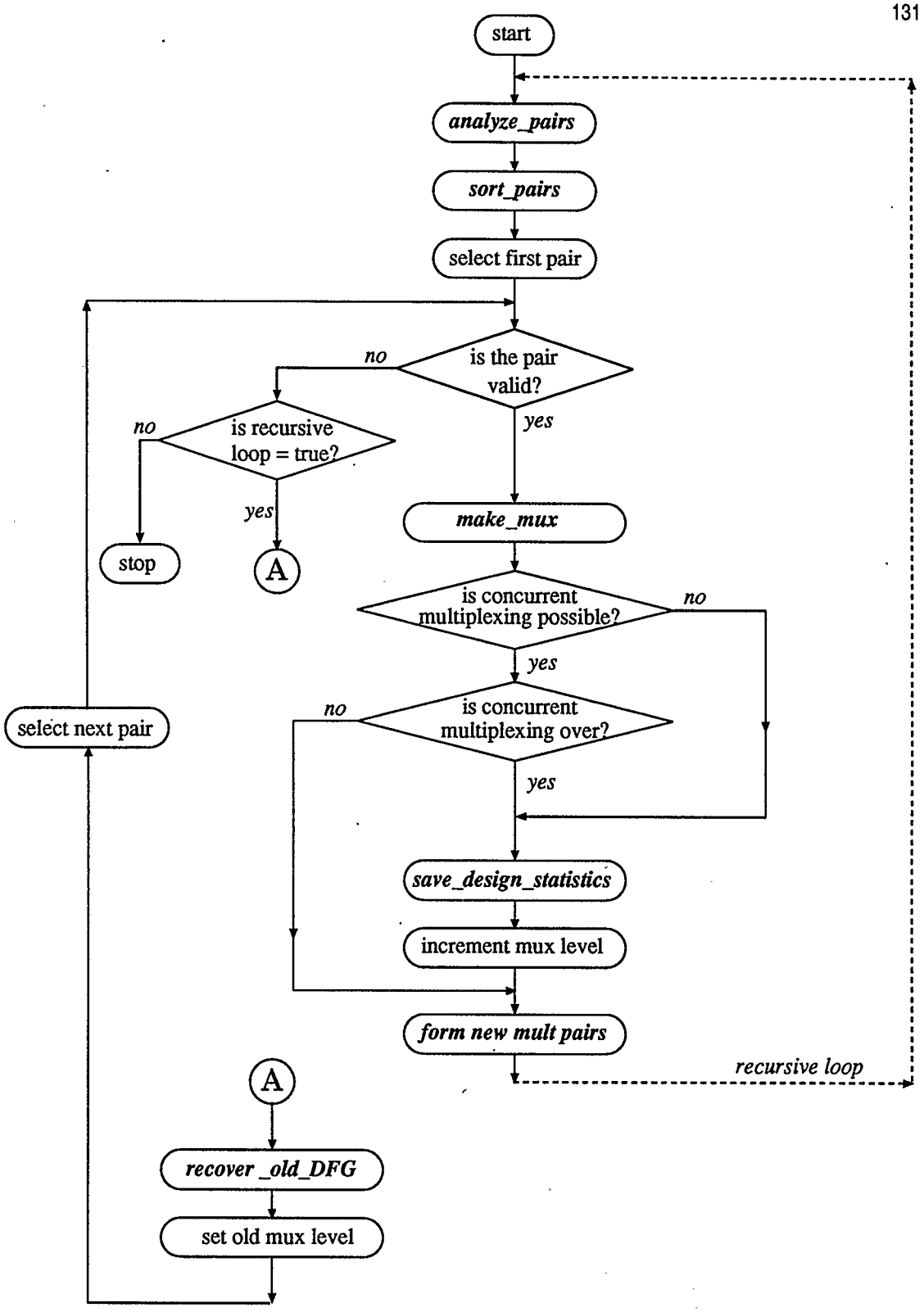


Fig. A.3 : Flow diagram of the module *generate_design_space*

level is considered by a recursive call to the *generate_design_space*. Before the recursive call, the level of multiplexing is incremented.

Once a pair is multiplexed, the second node of that pair does no longer exist in DFGs. So all other pairs containing that second node as a member will become invalid for future multiplexing. The module *form new mult pairs* is used to create a new set of pairs before the recursive call.

If a pair is part of CM, there may be more than one other pair that can be shared along with that pair. So there may be more than one design that can be generated with each pair at the same level of multiplexing. All the possible designs can be generated by calling *generate_design_space*, but without incrementing the level of multiplexing. Suppose a new pair is multiplexed concurrently in the recursive call. If this new pair is again part of CM, it indicates that there is at least one more pair which can be shared along with the first two pairs. The recursive call is used again to multiplex at the same level. If the new pair is not part of CM, it indicates that concurrent multiplexing is over. The characteristics of the resultant design will be calculated in *analyze_pairs*. After saving the design, the recursive call is used to find designs at the higher levels of multiplexing.

Once the algorithm is returned from the recursive call, the old DFGs are recovered. The level of multiplexing is set to the old value. The next valid pair is used to produce new designs. The recursive call is continued until all the pairs in the list are invalid. Thus the recursive module *generate_design_space* generates all the possible designs at different levels of multiplexing.

A.3 Graphical interface

An interactive graphics interface is developed using the EPLOT graphics library [35]. The *graphics interface* module will be used after generating the designs as shown in Fig. A.1. All the designs will be displayed in X-Y plot where the X coordinate represents the normalized sample frequency and the Y coordinate represents the number of LSI equivalent gates (gate count). A selective portion of the graph can be observed in full scale by using the zoom facility. This tool can be used to observe all the characteristics of a particular design using the module *display_design_statistics.*

Two other modules *make_first_code* and *change_specifications* can be used in the graphics environment. The module *make_first_code* generates FIRST [3] language output of a selected design. The second module *change_specifications* recomputes all the designs if the input specifications are changed. If the program is being used in alphanumerical terminal then the two modules can be used interactively in non graphics environment.

A.3.1 Display design statistics

The module *display_design_statistics* is used to display different characteristics of a design such as the number of multipliers, the gate count, the normalized sample frequency, *swl*, minimum *swl* or system latency. in a tabular form. It also compares all the characteristics of two recently selected designs using histograms. All the designs are selected by the mouse. After comparing the characteristics of different designs the user can select a design and can generate the design in the form of the FIRST language.

A.3.2 Make FIRST code

The module *make_first_code* is used to generate the FIRST language output of a design. Once a design is selected from the design space, the major steps in the synthesis process are used. The *make_mux* module performs the resource sharing at different levels using the multiplier pairs associated with the design. After multiplexing, the resultant multiplexed design is scheduled using the module *scheduling*. The appropriate delays are allocated in the DFGs using the module *allocation*. The module *make_first_node* generates the equivalent circuit of the resultant DFGs in the form of FIRST language. Once the design in FIRST language is generated, it can be tested using the SIMFIRST [3] simulator.

A.3.3 Change specifications

The module *change_specifications* is used to modify the specifications such as *swl* and *cwl* of the designs in the design space. All the designs will go through the procedure similar to the one involved in the *make_first_code* except the generation of the FIRST code. The effect of changing the specifications on the design space is shown in different design examples given in chapter 5.

Appendix B

A MULTIPLEXED 4th ORDER CASCADED BIQUAD DIGITAL FILTER

This section contains a multiplexed design of a 4th order cascaded biquad digital filter in the form of FIRST [3] hardware description language. The signal flow graph of the filter is given in Fig. 5.1a. The BITSYN program is used to generate the multiplexed design with three levels of multiplexing. In the following FIRST code, the statements which starts with '!' are comments. This filter has six multiplication operations. The design given here has one multiplier which performs all six multiplication operations in six cycles. The FIRST control generator can not be used to generate the required control signals. Either finite state machine or linear feedback shift registers (LFSR) can be used to generate the required control signals. The gate count given here includes the gate cost of the LFSR based control generator. The necessary information required to design the LFSR based control generator is provided as comments. The design is generated in the form of a high-level bit-serial operator, FILTER as given below.

FIRST code of the 4th order cascaded biquad digital filter

```
! cwl:  10;  swl:  23;  min_swl:  23
! add: 11;  mult: 1;  mux: 21;  dshift: 0;  mshift: 0;  delay: 282
!        signal_delay: 230; c_delay: 30; cl_delay: 22
! Total gates in LSI : 2762  Normalized sample frequency : 0.00724638

! OPERATOR FILTER created by BITSYN

  OPERATOR FILTER [] ( c00, c10, c20, c30, clear0, cs0 )
     n0_d0, m0_d0, m1_d0, m2_d0, m3_d0, m4_d0, m5_d0 ->  n7a_o

! minimum system word length = 23
! system word length = 23
! coefficient word length = 10
! multiplier latency = 17
! add                11
! multiply           1
! multiplex          21
! bitdelay           282

! n7a_o output at time 140  (or at time 25 in ^6^ cycle of period[=23])
! n0_d0 input at time 0
! multiplier coeff. m0_d0 (= b1) input time = 0, internal delay = 1
! multiplier coeff. m1_d0 (= g) input time = 0, internal delay = 1
! multiplier coeff. m2_d0 (= a1) input time = 0, internal delay = 1
! multiplier coeff. m3_d0 (= b2) input time = 0, internal delay = 1
! multiplier coeff. m4_d0 (= b3) input time = 0, internal delay = 2
! multiplier coeff. m5_d0 (= b4) input time = 0, internal delay = 2

  SIGNAL  n6ax_d22, n6a_d23, n6z_d20, n6_d21, n5_d1, nx14_d2,
          n5y_d3, n5y_d21, n5y_d22, n5y_d23, n7_d22, n5ay_d21,
          nx18_d1, nx18_wd, nx12_d2, nx15_d2, nx20_d3,
          nx10_d3, nx30_d4, m0_d1, m1_d1, nx13_d2, m2_d1,
          m3_d1, nx16_d2, nx21_d3, m4_d2, m5_d2, nx11_d3,
          nx31_d4, n5x_d21, n5x_d43, n5x_d46, n5x_d67,
          n5x_d88, n5x_d93, n5x_d94, n5a_d22, n8_d23, n7a_d24,
          n3_d0, nx17_d1, nx17_d22, nx17_wd, n4_d0, nx110_d1,
          nx110_d20, nx110_wd, n3a_d1, nx113_d2, nx113_d21,
          nx113_d22, nx113_wd, n4a_d0, nx114_d1, nx114_d2,
          nx114_d22, nx114_wd
```

```
CONTROL  c02, c04, c019, c020, c021_5x, c022, c023, c11, c12,
                c22, c33, clear22, cs0_1


ADD [1,0,0,0] (c021_5x) nx113_d21, nx113_d21, GND -> n6ax_d22, NC
ADD [1,0,0,0] (c022) n6ax_d22, nx114_d22, GND -> n6a_d23, NC
ADD [1,0,0,0] (c019) n5x_d88, n5x_d88, GND -> n6z_d20, NC
SUBTRACT [1,0,0,0] (c020) nx110_d20, n6z_d20, GND -> n6_d21, NC
SUBTRACT [1,0,0,0] (c00) n5y_d23, n5x_d46, GND -> n5_d1, NC
MULTIPLEX [1,0,0] (c11) n5_d1, n5x_d93 -> nx14_d2
ADD [1,0,0,0] (c02) nx14_d2, n5x_d94, GND -> n5y_d3, NC
BITDELAY [18] n5y_d3 -> n5y_d21
BITDELAY [1] n5y_d21 -> n5y_d22
BITDELAY [1] n5y_d22 -> n5y_d23
ADD [1,0,0,0] (c021_5x) n6_d21, n5y_d21, GND -> n7_d22, NC
ADD [1,0,0,0] (c020) n5x_d43, n5x_d43, GND -> n5ay_d21, NC
MULTIPLEX [1,0,0] (cs0) nx18_wd, n0_d0 -> nx18_d1
BITDELAY [22] nx18_d1 -> nx18_wd
MULTIPLEX [1,0,0] (c11) nx18_d1, nx17_d1 -> nx12_d2
MULTIPLEX [1,0,0] (c11) nx110_d1, nx17_d1 -> nx15_d2
MULTIPLEX [1,0,0] (c22) nx15_d2, nx12_d2 -> nx20_d3
MULTIPLEX [1,0,0] (c12) nx114_d2, nx113_d2 -> nx10_d3
MULTIPLEX [1,0,0] (c33) nx10_d3, nx20_d3 -> nx30_d4
BITDELAY [1] m0_d0 -> m0_d1
BITDELAY [1] m1_d0 -> m1_d1
MULTIPLEX [1,0,0] (c11) m1_d1, m0_d1 -> nx13_d2
BITDELAY [1] m2_d0 -> m2_d1
BITDELAY [1] m3_d0 -> m3_d1
MULTIPLEX [1,0,0] (c11) m3_d1, m2_d1 -> nx16_d2
MULTIPLEX [1,0,0] (c22) nx16_d2, nx13_d2 -> nx21_d3
BITDELAY [2] m4_d0 -> m4_d2
BITDELAY [2] m5_d0 -> m5_d2
MULTIPLEX [1,0,0] (c12) m5_d2, m4_d2 -> nx11_d3
MULTIPLEX [1,0,0] (c33) nx11_d3, nx21_d3 -> nx31_d4
MULTIPLY [1,10,0,0] (c04->c021_5x) nx30_d4,nx31_d4 -> n5x_d21, NC
BITDELAY [22] n5x_d21 -> n5x_d43
BITDELAY [3] n5x_d43 -> n5x_d46
BITDELAY [21] n5x_d46 -> n5x_d67
BITDELAY [21] n5x_d67 -> n5x_d88
BITDELAY [5] n5x_d88 -> n5x_d93
BITDELAY [1] n5x_d93 -> n5x_d94
SUBTRACT [1,0,0,0] (c021_5x) n5ay_d21,n5x_d21,GND -> n5a_d22, NC
ADD [1,0,0,0] (c022) n7_d22, n5a_d22, GND -> n8_d23, NC
ADD [1,0,0,0] (c023) n6a_d23, n8_d23, GND -> n7a_d24, NC
```

```
MULTIPLEX [1,0,0] (cs0) nx17_wd, n3_d0 -> nx17_d1
BITDELAY [21] nx17_d1 -> nx17_d22
BITDELAY [1] nx17_d22 -> nx17_wd
MULTIPLEX [1,0,0] (cs0) nx110_wd, n4_d0 -> nx110_d1
BITDELAY [19] nx110_d1 -> nx110_d20
BITDELAY [3] nx110_d20 -> nx110_wd
MULTIPLEX [1,0,0] (cs0_1) nx113_wd, n3a_d1 -> nx113_d2
BITDELAY [19] nx113_d2 -> nx113_d21
BITDELAY [1] nx113_d21 -> nx113_d22
BITDELAY [2] nx113_d22 -> nx113_wd
MULTIPLEX [1,0,0] (cs0) nx114_wd, n4a_d0 -> nx114_d1
BITDELAY [1] nx114_d1 -> nx114_d2
BITDELAY [20] nx114_d2 -> nx114_d22
BITDELAY [1] nx114_d22 -> nx114_wd

! DELAY and OUTPUT branches
MULTIPLEX [1,0,0] (cs0_1) GND, n7a_d24 -> n7a_o
MULTIPLEX [1,0,0] (clear22) n5y_d22, GND -> n3_d0
MULTIPLEX [1,0,0] (clear22) nx17_d22, GND -> n4_d0
MULTIPLEX [1,0,0] (clear0) n8_d23, GND -> n3a_d1
MULTIPLEX [1,0,0] (clear22) nx113_d22, GND -> n4a_d0

CBITDELAY [2] ( c00 -> c02 )
CBITDELAY [2] ( c02 -> c04 )
CBITDELAY [15] ( c04 -> c019 )
CBITDELAY [1] ( c019 -> c020 )
CBITDELAY [1] ( c021_5x -> c022 )
CBITDELAY [1] ( c022 -> c023 )
CBITDELAY [1] ( c10 -> c11 )
CBITDELAY [1] ( c11 -> c12 )
CBITDELAY [2] ( c20 -> c22 )
CBITDELAY [3] ( c30 -> c33 )
CBITDELAY [22] ( clear0 -> clear22 )
CBITDELAY [1] ( cs0 -> cs0_1 )

! LFSR control ganerator (c00 is the clock to LFSRs)
! Numerator - defines the initial states of flipflops
! Denominator - defines the the tap off/feed back paths
                        and the position of EX-ORs
! c10 = ^101010^  (numerator - ^01^  denominator - ^101^
! c20 = ^110000^  (numerator - ^000011^  denominator - ^1000001^
! c30 = ^111100^  (numerator - ^0011^  denominator - ^10101^
! cs0 = ^100000^ (numerator - ^000001^  denominator - ^1000001^
```

```
! control signal  c00 is used = 2 times
! control signal  c02 is used = 5 times
! control signal  c04 is used = 3 times
! control signal  c019 is used = 3 times
! control signal  c020 is used = 3 times
! control signal  c021_5x is used = 6 times
! control signal  c022 is used = 4 times
! control signal  c023 is used = 2 times
! control signal  c10 is used = 1 times
! control signal  c11 is used = 6 times
! control signal  c12 is used = 2 times
! control signal  c20 is used = 1 times
! control signal  c22 is used = 2 times
! control signal  c30 is used = 1 times
! control signal  c33 is used = 2 times
! control signal  clear0 is used = 2 times
! control signal  clear22 is used = 3 times
! control signal  cs0 is used = 5 times
! control signal  cs0_1 is used = 2 times

 END
```