7-3-1996

Intelligent dynamic space management systems

Margaret-Rose Madeleine Dabdoub
Florida International University

DOI: 10.25148/etd.FI14061581

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

INTELLIGENT DYNAMIC SPACE MANAGEMENT SYSTEM

A thesis submitted in partial satisfaction of the

requirements for the degree of

MASTER OF SCIENCE

IN

ELECTRICAL ENGINEERING

by

Margaret-Rose Madeleine Dabdoub

1996

To: Dr. Gordon Hopkins
College of Engineering and Design

This thesis, written by Margaret-Rose Dabdoub, and entitled Intelligent Dynamic Space Management System, having been approved in respect to style and intellectual content, is refered to you for judgement.

We have read this thesis and recommend that it be approved.

Amando Barretto

Grover Larkins

Malcolm Heimer

Subbarao Wunnava, Major Professor

Date of Defense: July 3, 1996

The thesis of Margaret-Rose Dabdoub is approved.

Dean Gordon R. Hopkins
College of Engineering

Dr. Richard L. Campbell
Dean of Graduate Studies

Florida International University, 1996

This thesis is dedicated to my parents.

To my mother,

Carmen Marguerite,

who gave us life, love and strength in spirit

and to my father,

Hanna Abraham,

a man of principle who taught us strength in character.

You will never be far from our hearts.

ACKNOWLEDGEMENTS

For all your moral support and more, thank you,

My brothers and sisters ,
Marc-Anthony, Marie-Madeleine, Elizabeth-Ann, John-Peter, Graciella-Gloria.

Thomas Christopher Gilbar,

Bruce, Timon, Ramana, Krishna, Hamid, Viji, DAISY!!, Miriam, Injun, Miguel, Kishore, Isidro, PABLO!, Carlos, Irma, Laura...

My friends in Boston, Yoly, David, Carol, Barbe, Andrea, Kathleen, Mary Lou, Paulette, Robin, Kim, Ross, Louise (...5 years later)

My committee members, especially for their patience and understanding.
The Department of Electrical and Computer Engineering, in particular, Dr. James Story, Pat Brammer, Dr. Malek Adjouadi, Mike Uricinitz.

This work was supported by the National Science Foundation Grant No. CDA-9313624 with the Center for Advanced Technology and Education-CATE with the Department of Electrical and Computer Engineering at FIU.

ViewLogic, Inc.

Xilinx Corporation

Parking Systems and Analysis

ABSTRACT OF THE THESIS

INTELLIGENT DYNAMIC SPACE MANAGEMENT SYSTEM

by

Margaret-Rose Dabdoub

Florida International University, 1996

Professor Subbarao Wunnava, Major Professor

The objective of this work is to design, simulate and synthesize a dynamic space controller system. The concept of space allocation and management can be applied to more than physical space. It may also be taken in the contexts of memory, network or bus management. The management and allocation of any space depends mostly on the twin factors of demand and availability. Time, the size and amount of space available, as well as the number of requests for the spaces, must also be considered. The proposed design has the capability to monitor multiple spaces for vacancies, length of time occupied and to dynamically track the number and location of available spaces. This system is easily programmed to adapt to the user's application and is modular in design in order to facilitate expansion. The specific problem addressed in this work is a parking lot controller.

TABLE OF CONTENTS

LIST OF FIGURES AND TABLES

CHAPTER 1

INTRODUCTION

1.1 Service Queues

Throughout the course of a day one encounters many situations in which it is required to wait for some type of service. Some examples are waiting in line at a checkout counter, or waiting on a train, an elevator or parking space. How long the wait is, is determined by the availability of the item or service in demand. One area of study, queueing theory, deals with the mathematical analysis and models of queues or waiting lines [9]. Questions such as "How many servers are available to clients ?", "How many clients are waiting for each server ?", "How long does it take each server to process a client?" are addressed in this field. The service, whatever it may be, implies a combination of time and space allocation per customer. Each customer has to enter the system at some point in time and in some area of the system, usually the input to the system. It then takes a certain amount of time to handle/serve each customer and make room and time for the next. The idea is to as efficiently and accurately as possible provide enough servers per client to minimize wait time. At the same time, the idea of cost efficiency must be weighed against the necessity of improving the efficiency of the service.

1.1.1 Space Allocation

Space plays a major part in many aspects of today's world. This is evident in areas such as urban development and industrial expansion. There are many situations in which a limited amount of space is in demand by a number of clients. Hotel room reservations and parking spaces make good examples. The problem of space allocation falls within the scope of queueing theory.

The basic characteristics of a queueing system are identified as follows:

(i) The input or arrival pattern of customers;

(ii) The pattern of service;

(iii) The number of servers or service channels;

(iv) The capacity of the system; and

(v) The queue discipline [18].

These characteristics may be applied to the space allocation and management system. The arrival pattern in this field can vary. In some cases it may be random or it may be a fixed pattern. For instance, if the space in question is a retail store shelf, the arrival of the new stock will probably depend on a fixed time from the date ordered to delivery of the order. The amount of stock and the frequency with which it is ordered is directly related to the amount sold, which could, in turn, vary on a daily or weekly basis. This would provide a random pattern of service of the stock. In most cases the amount of spaces available will be limited. The total number of servers (spaces in this case) will be fixed but the number available at any one time can vary. In the case of a fixed number of servers and one server per customer, the capacity of the system will be limited and equal

to the number of servers. The capacity of the system may be fixed if the number of servers are fixed. For this system the queue discipline is First come First served [4].

1.2 Approach to the Problem

The approach to this problem will try to involve the main principles of queueing theory without the benefit of any mathematical models. It is not the objective of this thesis to develop or to apply any queueing system model/algorithm other than a simple intuitive approach to this application. The objective of this work is to design a controller that will monitor the use of designated spaces. The system will be modeled, however, through the hardware design of the controller module. This will be done through hardware description language (HDL) and then simulated to verify its functionality. The design of this space allocation controller will be aided with the use of the Very High Speed Integrated Circuits Hardware Description Language (VHDL) language.

1.3 What is VHDL

VHDL (Very High Speed Integrated Circuit Hardware Description Language) is a hardware description language (HDL). HDLs describe hardware and use that description to simulate, model, test, design and document a digital system. Some languages use primitive constructs or symbols to replace schematics of digital circuits, while others are more structured and can represent the circuit at different levels of abstraction. VHDL is a highly sophisticated HDL which supports hierarchical descriptions and timing constraints of digital systems [5].

1.3.1 The Design Cycle

The design cycle covers the various levels of abstractions.There are two categories of abstraction, behavioral and structural. Behavioral abstraction deals with the function of the system in terms of its inputs and outputs. The structural abstraction describes the system in terms of its interconnections to more primitive components. In Figure 1.1 the different levels of the design cycle are illustrated. The System, Chip and Register levels fall under the behavioral abstraction. The System level describes the VHDL or English description of the design; Chip, Integrated Circuits (ICs) such as a microprocessor and the Register level applies to complex logic circuits. The Gate, Circuit and Layout levels belong under the structural abstraction. The Gate level describes the fundamental logic gates and flip-flops, while the Circuit level presents transistors and passive elements. Finally the Layout describes the silicon masking and manufacturing of the design at its lowest level.

1.4 History of VHDL

The process of designing, testing, producing and maintaining digital systems can be very tedious and costly. The U.S. Department of Defense (DoD), being involved in a variety of military projects awards a number of contracts to many different manufacturers. Generally, each manufacturer goes about the design and documentation of a system in different ways, which affect the cost and on-going maintenance of the system. As a result, the DoD found it necessary to establish a standard to adhere to when evaluating proposals on the basis of the digital system design documentation, production and maintenance

costs. With such a standard in place, the DoD hoped to cut costs during the on-going life cycle of the system.

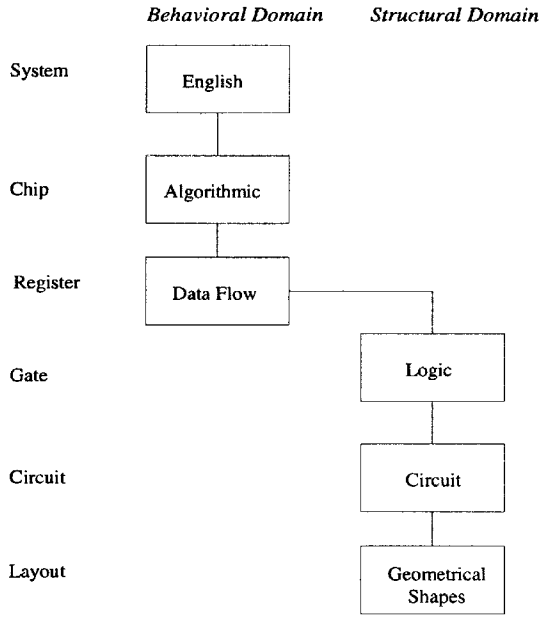

Figure 1.1 The Design Cycle

In an effort to formalize documentation and design in the Very High Speed Integrated Circuit (VHSIC) program the DoD held a workshop in summer of 1981, which investigated the requirements necessary for HDLs to be considered a standard. Representatives from the United States Air Force (USAF) and from industry, including Dr. John Hines, Roger Lipsett, Ron Waxman, Moe Shahdad and Dave Barton met in June 1981 [27]. A document entitled the "Department of Defense Requirements for Hardware Description Languages" was produced at the end of the workshop [17]. In 1983 the DoD contracted with IBM, Texas Instrument and Intermetrics corporations to further develop

the VHDL language based on the requirements established at the 1981 workshop. Within six months a second version of VHDL was released but with several shortcomings. However, by the end of 1984, significant improvements were made to VHDL. These improvements were then incorporated in version six. Quite a few of the people that participated in the Woods Hole workshop also participated in this collaboration. From IBM came Larry Saunders, Ron Waxman; from TI came Don Newman, Dave Ackley and from Intermetrics, Moe Shahdad, Victor Berman, Bill Bail and Bill Carlson [24].

In 1985 the VHDL copyright was handed over to the Institute of Electrical and Electronic Engineers (IEEE) in order to continue with the industrial standardization and development of the hardware description language. The work done by the IEEE produced what is now formally known as VHDL 1076-1987, the standard HDL from the IEEE. The latest version of the IEEE standard is VHDL-1993.

Several members of the 1983 team later published papers reflecting the development of VHDL. The article entitled "Where VHDL Fits Within the CAD Environment" was presented by John Hines at the 24th ACM/IEEE Design Automation Conference Proceedings in 1987 [3]. Another team member, Larry Saunders also presented "The IBM VHDL Design System" at the same conference.

1.5 Why VHDL

VHDL was the result of a government effort to standardize the VHSIC program. This forced most of the industry interested in DoD contracts to conform to the standard required by using the VHDL language.

VHDL can support different design methodologies such as top-down or library based [15]. It also supports different design technologies, for example, Application Specific Integrated Circuit (ASIC) or random logic. This property of VHDL makes it a suitable design tool for designers of different companies with a variety of design needs, whether it be a gate-level or system level approach. VHDL also supports generic modeling which allows a component to be represented by an actual manufacturer's device without changing the component model.

It is important to understand that VHDL is technology and process independent. In other words, a high-level description can be written for a digital system and then translated into gate-level using the technology desired, for example CMOS. VHDL does not need to "know" the technology the design is based on or the process that will be followed for implementation in order to describe and simulate the design [5].

VHDL can simulate the operation of a digital system through its behavioral descriptions. It can model the behavior of the digital system at the black box down to the gate level. This is an important feature of VHDL, whereby components of a system which are represented at different behavioral levels can be integrated into the description of the complete system, regardless of the level used to describe the system. This also enhances the maintenance procedures for existing digital systems in that a redesign or modification in the system can be checked by re-simulating the system with the new VHDL defined component. VHDL surpasses other HDLs in this respect, as they operate best at the logic and gate level and not at the system level [15].

Another advantage of this wide descriptive capability of VHDL is the fact that any VHDL standard system should be able to run any VHDL standard component. This leads to the sharing or exchange of common VHDL modules or components between different design teams. These common modules can be included in libraries which can then be accessed by the different designers. Designers are able to use high-level descriptions of sub-modules of a complete digital system and integrate them into their design description. This allows each sub-system to be developed and tested independently of other sub-systems. Ultimately this reduces the time and effort needed for the development of the complete system.

VHDL also has the capability to model hardware timing. It supports minimum and maximum timing, setup times, holds and spike detection. Overall, VHDL exhibits most of the requirements set out as standard by the DoD requirements. To summarize, it documents; it is capable of high-level design; it supports simulation, synthesis and testing of hardware and can be used as a driver for a physical design tool. It supports hierarchical descriptions, libraries, generic models, sequential and concurrent simulation statements, timing control, sub-programs (functions and procedures) and type declarations. As a result of all this VHDL has emerged as one of the most important HDLs over the last twenty years and has been strongly supported by the industry.

1.6 VHDL Compatibility

One feature of VHDL is its versatility with different technologies. Viewlogic's

VHDL design process allows the use of different manufacturers' device libraries. At the same time, many other computer-aided design tool manufacturers do use products of Viewlogics's design package. In particular the Xilinx corporation, one of the leaders in Field Programmable Gate Array (FPGA) technology, markets XACT Design Manager which supports wirelist files produced with Viewlogic's design synthesizer. To use the wirelist file XACT provides for the conversion of the wirelist file into its own netlist file which it then manipulates into FPGA configuration information. The design process can therefore start with the VHDL description of the system then synthesize according to the target technology, convert into Xilinx format and, finally, configure the FPGA.

CHAPTER 2

VHDL: A POWERFUL DIGITAL DESIGN AND SIMULATION TOOL

2.1 The Need for Modeling Capabilities

Most of the present day electronic, communication, and control systems tend to be heavily digital and computer oriented. The complexity of the digital circuits has grown to fantastic proportions. With several digital modules in a system, interfaces in the forward and feedback paths, logical, functional, interface and timing accuracy is of utmost importance. It is neither practical nor feasible to build several of the digital modules in a laboratory environment, test each independently, and interface them together for a system level test. Often such tests are highly subjective, cumbersome and may not give a real picture of the system under consideration.

An attractive alternative is to simulate the modules, sub-systems, and integrated systems and study the functional, interface and timing response. There have been several digital systems simulation software and hardware description language (HDL) packages in use such as the Instruction Set Processor Specification (ISPS) and A Hardware Programming Language (AHPL) [20]. These do not follow set standards and some of them are incremental. In the last ten years, there has been growing interest in the VHDL (Very High Speed Integrated Circuit HDL) design platform with capabilities to design and simulate the sub-modules, then integrate them and study the response of the system as a whole. In addition VHDL has been following the standards set by the IEEE and leading industries.

At Florida International University, the VHDL package in use is Viewlogic's Powerview. It adheres to most IEEE standards with a few modifications. The package provides the designer with several tools. The tools most used in the design environment are ViewDraw, a schematic capture program; VHDL Analyzer, the program that "compiles" and transforms the VHDL description file; ViewSIM, the simulation program, and ViewSynthesis, which is the tool that creates the schematic from a VHDL description.To understand how these tools work one should have some knowledge of the simulation process. The following sections discusses the techniques involved in design modeling and give an example of how a design is accomplished using these tools.

2.2 VHDL System Design

A design is typically presented in a top-down approach. Its' specifications are given as an outline of the complete functional system. A break-down of the system facilitates the design by incorporating smaller modules which in turn can be broken down into further detail. In this way the overall design is structured and the individual components may be created more easily. A VHDL description is made up of several sections: an entity declaration, an architectural description, a configuration and a package. The latter two are optional and may not be required, however the first two are mandatory. To model wire connections VHDL provides objects called *Signals* that take on digital logic values known as a BIT or *vlbit* in VIEWlogic's version. Other objects and type declarations are permitted, including some user-defined types.

2.2.1 The VHDL System Block Description.

The first step in describing a digital system in VHDL, is the declaration of an *entity*. The entity declaration details the interface, that is, the inputs and outputs of the device. See Figure 2.1. The VHDL keyword *Entity* begins this section and is followed by the design name and the keyword *IS*. The *PORT* declaration follows this detailing the specific inputs and outputs of the design. The system inputs cannot be assigned a value within the entity (i.e. used as outputs), nor can the outputs be used as inputs (i.e. to assign values to) other signals or components in the entity. Essentially, the entity describes a black box view of the system. What makes up this black box or how it operates is described in its architecture. Therefore, the next step is to specify the *architecture* of the system.

```
Entity entity_name IS
        PORT ( Signal input_signal :IN vlbit; Signal ouput_signal:OUT vlbit);
End entity_name;
```

Figure 2.1. Format of an Entity Description

2.2.2 The VHDL Functional Description of a System

The architecture of a design gives the description of it's functions. This section starts with the VHDL keyword *Architecture* followed by an identifier and the design name. An identifier specifies the type of description that will be given. Any components or signals that will be used in the description can be declared next. To begin the actual description the keyword *BEGIN* is used and the architectural body is completed with the keyword *END* followed by the identifier.

The architecture can be a structural, behavioral or data flow description of the design entity. A structural architecture describes the operation through the interconnections to other entities and is used when timing is a factor. Behavioral descriptions are used when the functionality of the system needs to be tested. The data flow description specifies the path the data follows in the actual hardware implementation.

In a purely structural environment, concurrent statements are issued to declare the use of or *instantiate* the component. The input and output connections of the component are the focus of this description. The use of internal signals may be necessary to establish the interconnection of the component to the system's inputs and outputs to any other component. The data flow level uses concurrent signal assignments rather than instantiations. It presents the actual path the data signal may take in an implementation. It can also benefit from the use of internal signals. The behavioral description is most often presented as a process. This is a block of sequential statements, that is, statements that are executed in the order stated. The *Process* statement, although concurrent, implies that the statements to follow will be executed sequentially. The IF-THEN-ELSE and WAIT statements are examples of sequential statements.

An entity can have more than one type of architectural description; meanwhile, an architectural body can be a combination of different architectures [3]. Normally, a configuration is declared to specify which of the architectures is to be used in the simulation or synthesis. In the case of Viewlogic only one architecture is allowed per entity within a file [33].

The three descriptions are depicted in figures 2.2 (a), (b) and (c), using a full subtractor as a design example. Note that the terms structural, dataflow or behavioral is used here as illustrations. They are not VHDL keywords, they are user-defined; however, the identifier is mandatory.

2.2.3 The VHDL Analyzer

The VHDL listing is analyzed using Viewlogic's *VHDL Analyzer*. If there are no syntactical, semantical or lexical errors upon analysis then the design is ready for testing. The VHDL Analyzer creates intermediate files that are placed in a design library. With the Viewlogic *VHDL Analyzer* these files have the extensions of *.vli* and *.vsm*. Both these files are necessary to perform a simulation using the ViewSIM tool. To create the black box schematic, another tool *VHDL=>sym* is used. This depends only on the entity description. The architectural description does not have to be written to accomplish this task. The resulting black box representation is seen in Figure 2.3.

```
ENTITY Full_Subtractor IS          -- Declaration of the Entity
      PORT(A, B, Bin : IN vlbit; S, Bout : OUT vlbit);    -- Interface Information
END Full_Subtractor;               -- End of Entity Declaration

ARCHITECTURE dataflow OF Full_Subtractor IS
                                   -- Data Flow Description of Full_Subtractor
Signal o_1, a_1, a_2, n_1: vlbit;  -- Internal Signals

BEGIN                              -- Begin Data Flow description
   o_1 <= B OR Bin;                -- Assign temporary internal signals
   n_1 <= NOT A;
   a_1 <= B AND Bin;
   a_2 <= o_1 AND n_1;
   Bout <= a_1 OR a_2;             -- Assign Output Bout final value
   S  <= A XOR B XOR Bin; -- Assign Output S final value
END dataflow;                      -- End Data Flow description
```

Figure 2.2(a) Data Flow Description of Full Subtractor

```
ARCHITECTURE structural OF Full_Subtractor IS
                                        -- Structural Description of  Full_Subtractor
Component OR2                   -- Declare Components to be used
        Port (x,y:IN vlbit; z:OUT vlbit);
end Component;

Component NOT1
        Port (x:IN vlbit; z:OUT vlbit);
end component;

Component AND2
        Port (x,y:IN vlbit; z:OUT vlbit);
end Component;

Component XOR3
        Port (x,y,w:IN vlbit; z:OUT vlbit);
end component;

Signal o_1, a_1,a_2, n_1:vlbit;                 -- Declare Internal Signals

BEGIN                                           -- Begin Structural description
        G1: OR2  Port Map (B,Bin,o_1);          -- Instantiations of compnents.
        G2: NOT1 Port Map (A,n_1);              -- Signals are used in order declared in components
        G3: AND2 Port Map (B,Bin,a_1);
        G4: AND2 Port Map (o_1,n_1,a_2);
        G5: OR2  Port Map (a_1,a_2,Bout);               --Final output Bout is out put of 2input OR gate
        G6: XOR3 Port Map (A,B,Bin,S);          -- Final output S is output of 3 input exclusive or gate
END structural;                                 -- End structural description
```

Figure 2.2(b) Structural Description of Full Subtractor

```
ARCHITECTURE behavior OF Full_Subtractor IS
BEGIN

SUBTRACT: Process (A,B,Bin)
BEGIN
        IF (A='0' AND B='0'THEN
                IF ( Bin = '0' )THEN
                        S<= '0';
                        Bout <= '0';
                ELSE
                        s<= '1';
                        Bout <= '1';
                END IF;
        ELSIF (A='0' AND B='1'THEN
                IF ( Bin = '0' )THEN
                        S<= '1';
                        Bout <= '1';
                ELSE
                        s<= '0';
                        Bout <= '1';
                END IF;
        ELSIF (A='1' AND B='0'THEN
                IF ( Bin = '0' )THEN
                        S<= '1';
                        Bout <= '0';
                ELSE
                        s<= '0';
                        Bout <= '0';
                END IF;
        ELSE
                IF (Bin = '0')THEN
                        S<= '0';
                        Bout <= '0';
                ELSE
                        S<= '1';
                        Bout <= '1';
                END IF;
        END IF;
END Process Subtract;

END behavior;
```

Figure 2.2(c) Behavioral Description of Full Subtractor

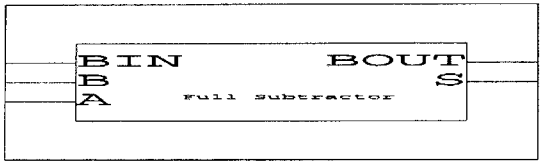

Figure 2.3. Symbol Representation of Full Subtractor

2.3 Simulation

A brief explanation of simulation and how signals are handled is given here before describing the complete process. Simulation is used to verify that the system's functional requirements are met and also to check the system response for faults in certain situations. Simulators in VHDL act on the entity and architectural description of a design. Two main steps are taken before simulation is finally carried out; elaboration and initialization. In the elaboration phase, the entity is expanded and linked, storage for signals and other objects is allocated, constants and variables initial values are assigned. In the initialization phase, the values are assigned to declared signals, any processes are executed once and the simulation time is forced to zero nanoseconds [3].

It is always important to know just how well a system performs before it is put into production. Simulation offers ways not only to test a specific module but also it's

behavior when interfaced with other systems simply by modeling the other systems. The different systems can be simulated together even if they are not at the same level of abstraction.

Simulators can be classified into two types, oblivious and event driven. The former evaluates the outputs of all components at fixed points in time. The latter, as the name suggests, evaluates only that part of a circuit that is affected by an event or change on any of its inputs or related signals[20]. The oblivious method of simulation performs a sequential evaluation of the entire circuit until its output is stable. If at any point in the circuit, any value changes then there must be another simulation pass to determine if there is stability in the outputs. For this reason the oblivious method has proved to be inefficient. The event driven simulation is more efficient in that it does not evaluate the whole circuit but only the nodes affected by the changing inputs. This method lends itself very well to concurrent signal assignments.

2.3.1 Timing Considerations

Timing is, of course, a very important factor in hardware designs. The signals in a hardware component arrive at an output after a certain propagation time. This time may depend on the complexity of the circuitry that provides the signal. If no propagation time is specified then the propagation time is referred to as delta time. Delta time is a non-zero time that is greater than zero but less than any standard unit of time [1]. Digital circuits can incorporate signal delays essential to the function of the system. On the other hand, the circuit may require an immediate change in the output to reflect any

change in the input. VHDL allows the *AFTER* clause to impose a delay on a signal assignment. Delays can be identified as inertial or transport. Inertial delay implies that the signal will only be assigned if it remains at the same level for, at least, as long as the delay time specified in the *AFTER* clause. With transport delays any change on the input signals will propagate to the output after the delay specified, regardless of the time it stays at the new level [1]. Figures 2.4(a) and (b) illustrates these. Unless otherwise specified all delayed signal assignments are assumed to be of the inertial delay type as that is what most represents the properties of capacitive networks in digital systems.

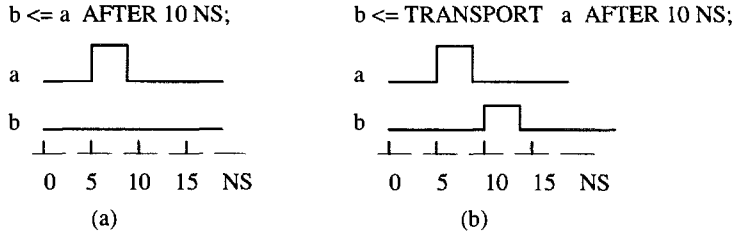

Figure 2.4. Inertial and Transport delays for signal assignments.

The use of concurrent and sequential constructs in VHDL enables a designer to model the design as close as possible. The concurrent signal assignment implies an immediate effect on the output signal. The statement *a <= x OR y;*, depicts a logical OR operation on inputs *x* and *y* that drives the output *a*. Any changes on the right-hand side (RHS) of the statement changes the value of the output on the left-hand side.

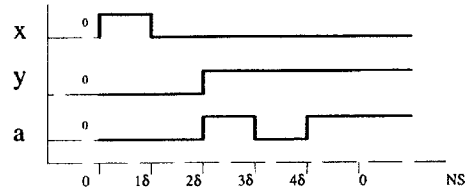

Figure 2.5. Concurrent Signal Assignment Waveforms

Concurrent statements can be regarded as simultaneous occurrences; a parallel effect. The signals may be forced to have delays but the evaluation takes place simultaneously. The output values do not take effect until the delay time. If delta delay time is taken into consideration each signal transaction will be evaluated every 1δ time period resulting in the waveforms of Figure 2.5. A simulation example of concurrency is shown in Figure 2.6(a) for the data flow description of the full subtractor of Figure 2.2(b). Compare this to the resulting waveforms of the same description with delays, Figure 2.6(b) . Each signal assignment is delayed 10 nanoseconds, therefore the total propagation delay for output *Bout* is 30 nanoseconds. This is because the listing describes a three-stage circuit.

Sequential statements on the other hand, are executed one at a time. This forces a delay in signal assignments because the statements are carried out in the order they are written. Again, delays may also be specified for these assignments using the *AFTER* clause. Signal assignments are evaluated with consideration of the existing transactions on that driver. The new transactions on the same signal may take precedence over the old or be added to it, depending on the type of assignment or the timing. Think of sequential

statements as a way of scheduling the signal assignments. The rules of thumb of sequential signal assignments are outlined in Table 2.1

	TRANSPORT	INERTIAL
New Transaction BEFORE Already Existing	Overwrite Existing Transaction.	Overwrite Existing Transaction.
New Transaction AFTER Already Existing	APPEND the new transaction to the driver.	Overwrite the existing value if different, otherwise keep both.

Table 2.1 Guide to evaluate transactions on driver of signal with multiple assignments.

Sequential statements are found in behavioral descriptions such as the one in Figure 2.2(c). The simulation of that entity is the same as the structural or data flow description without delays. To illustrate the difference, delays of 10 ns and 20 ns were added to the signal assignments of S and Bout respectively, and the design simulated again. The results can be seen in Figure 2.7 (a) and (b). Note the difference between the waveforms of the concurrent and sequential descriptions. The delay here is a total of 20 ns and not 60 ns as it might have been if the description was concurrent.

2.3.2 Simulating the Design

In order to test the design the simulator, ViewSIM, is invoked. A waveform for each input and output was generated by cycling through the eight combinations of A, B and Bin as inputs to the Full Subtractor. The batch process is accomplished through the use of a command file, which details the logic levels of the signals at particular times in

the simulation cycle. The waveforms of the dataflow and behavioral descriptions of the Full Subtractor have already been seen in Figures 2.5(a) and 2.7(a). After verifying that each output was correct for each input combination the next step is to synthesize the model.

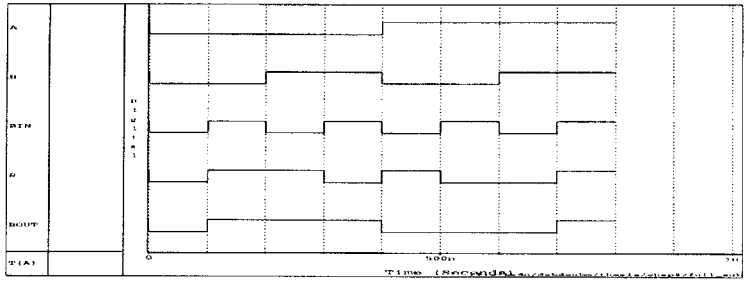

Figure 2.6(a) . Simulation of Data flow description of full subtractor

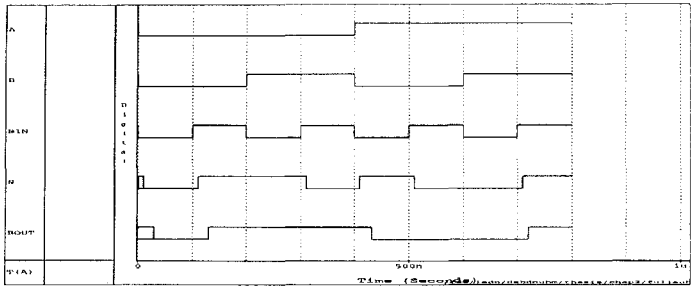

Figure 2.6(b). Simulation of Data flow description of full subtractor with delays

2.4 Synthesizing the VHDL Description

Viewlogic provides the designer with a synthesis tool called *ViewSynthesis*. Within ViewSynthesis the target technology can be specified from a library database. The design may be targeted to any technology desired. This feature of VHDL makes digital designs portable; remember the design is not technology dependent. In this case, the XC4000 library was selected. ViewSynthesis analyzes the VHDL listing before it tries to synthesize. The synthesis process attempts to translate the digital design into an optimized gate-level representation [32]. The result of this process is a wirelist file detailing the interconnections of the design's elements.

Timing is not a factor in synthesis. This does not mean that it is not important. It does not translate into the digital design. Timing specifications are more useful for

simulation than they are for synthesis. This is illustrated in the schematics generated for the descriptions of the full subtractor with and without delays. See Figures 2.8(a) and (b).

The VHDL descriptions of Figures 2.2(a) (b) and (c) were synthesized. The synthesized schematics are shown in Figures 2.8(a)-(d). Notice the difference between the different models. Comparing the language description of the Full Subtractor in Figure 2.2(a) and the final schematic of Figure 2.8(a) it is clear that the final design is slightly different to the original description. This is because the synthesizer tries to find the most optimum design in terms of the number of gates, the timing, loads and the area used in the design at the semi-conductor level.

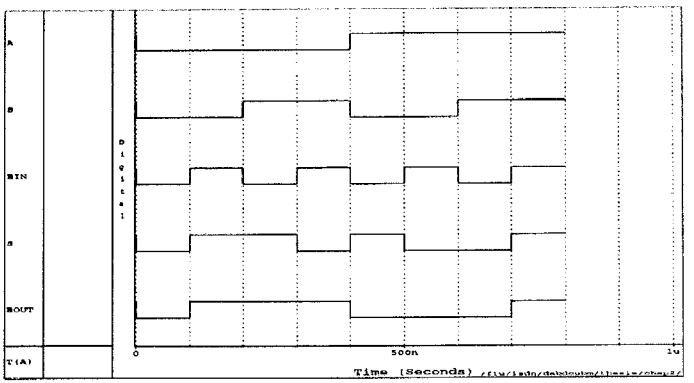

Figure 2.7(a). Simulation of Behavioral Description of the Full Subtractor

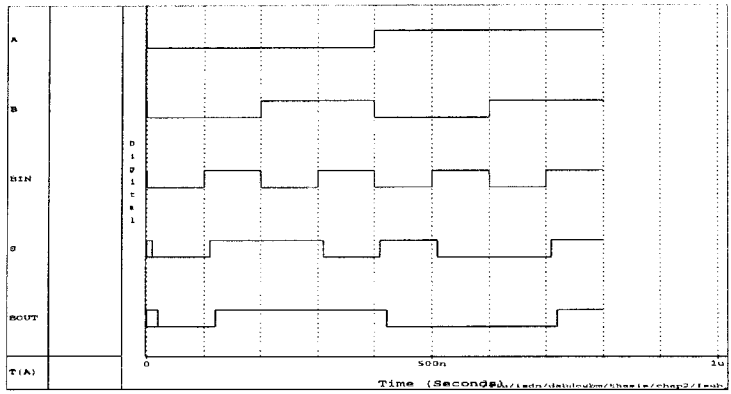

Figure 2.7(b). Simulation of Behavioral Description of the Full Subtractor with Delays

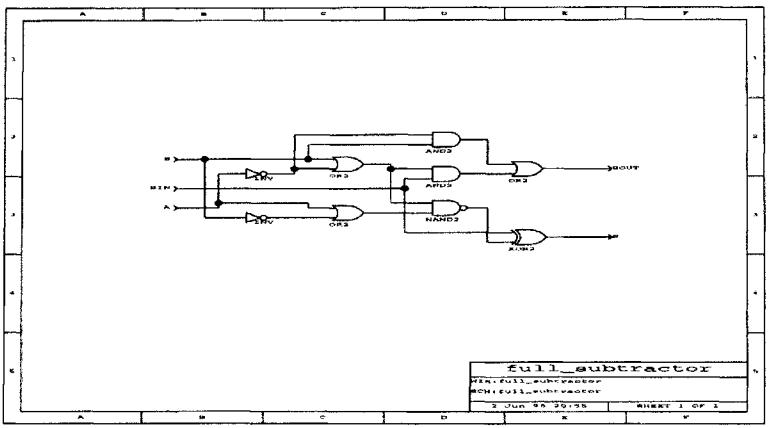

Figure 2.8(a). Synthesized Schematic of Full Subtractor

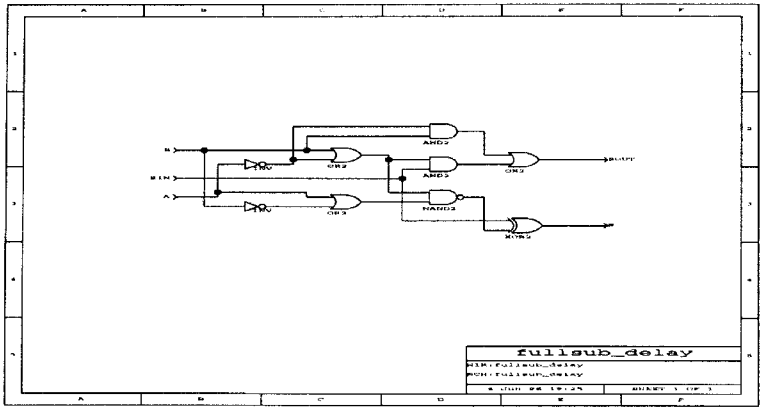

Figure 2.8(b). Synthesized Schematic of Data Flow Description of Full Subtractor

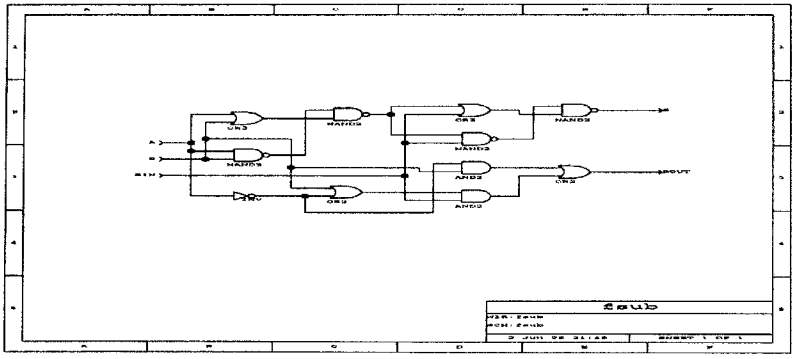

Figure 2.8(c). Synthesized Schematic of Behavioral Description of Full Subtractor

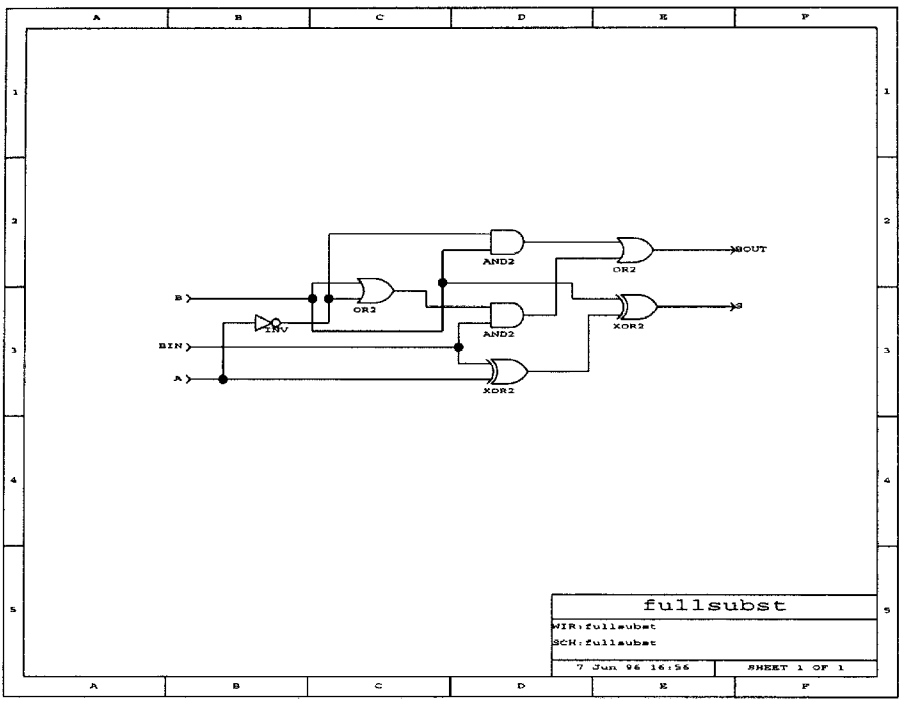

Figure 2.8(d). Synthesized Schematic of Structural Description of Full Subtractor

CHAPTER 3

CONCEPTS OF SPACE ALLOCATION AND MANAGEMENT

3.1 Memory Management

Computers use memory as a means of temporarily storing programs and related data for execution. How a computer manages memory is dependent on the features of the microprocessor and the operating system being used for that computer [28].

In memory management, the necessity arises to allot specified amounts of Random Access Memory (RAM) or space in memory, for use by certain programs. Some programs utilize more memory than others, and in a limited RAM configuration there needs to be some control over which programs are guaranteed memory to run, and the minimum amount of memory they need to execute. In earlier technology, only one program could effectively be loaded and executed in memory at one time.

As processors developed, they brought with them the ability to access larger areas of memory and better techniques to manage it. The memory can be partitioned or segmented. In the partition approach, main memory is cut up into blocks that are allotted to a particular application. If the application is too big for an available block then it cannot run until a large enough block becomes available. If the space is bigger than the application then some portion of the block is unused and wasted. Improvements on the partition idea resulted in a technique known as Paging.

Instead of large partitions, memory would be divided into relatively small equal blocks of memory known as page frames. Each application would also be organized into

small blocks referred to a pages. When the application starts it is loaded into the page frames of main memory. They need not be loaded consecutively. A page table is created for each process loaded into memory. The page table contains the number of the frame where each page of a process is located in memory. This frame number serves as a base address for the physical address. The physical address is the actual location in memory.

Within each page of the process, a logical address is used to refer to data or instructions belonging to the program. The logical address indicates the page number and offset relative to the beginning of the page instead of the program as was the case with larger partitioning schemes. The physical address is determined from the page table holding the frame number and the offset given by the logical address. See Figure 3.1.

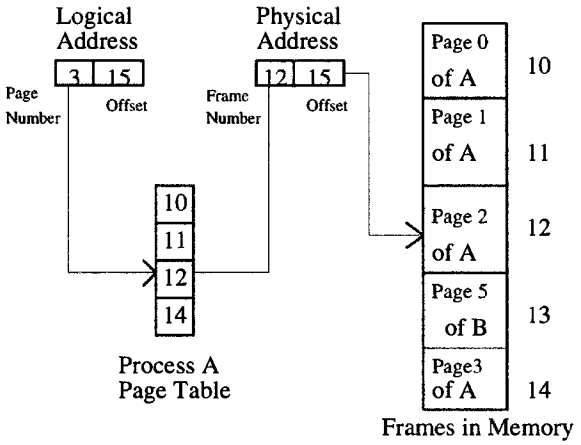

Figure 3.1 Logical and physical addresses in Paging

Paging allows for a multiprogramming environment. Taken a step further, paging may be implemented on demand, where only certain pages of a program may be loaded

into memory when requested instead of loading the entire program. In this way, even more programs may be loaded into memory, a page or more at a time depending on the execution of the program. A program that is larger than main memory can be executed in this way. The pages of the program are stored on the disk creating the illusion of a larger or virtual memory.

Segmentation is another way of organizing memory into large sections. The programmer using this memory needs to be aware of the segment usage and privilege. The segments can be of varied sizes and can be changed by the operating system. Program instructions are referenced using the segment number and an offset to make up the address. A segment may be used to run programs or to store data that can be available to other programs. Segmentation also works with the concept of virtual memory. In the case of the Pentium processor, the non-segmented memory is 2^{32} =4 Gigabytes of memory (32 address lines); however, with its 14-bit segment reference address the processor can sees 2^{46} = 64 terabytes of memory [28]. This is far more than the hard disk space available. The earlier Intel 8086μp also uses this segmentation scheme with four 16-bit segment address registers and 16-bit pointers to index each segment. The segments include a Code, Data, Stack and Extra segments each with its respective index register. The physical address is generated by shifting the segment registers four bits to the left and adding the 16-bit offset provided by the index/pointer register to create a 20-bit physical address. The segments may overlap or be completely isolated from each other [31].

For personal computers running in the Disk Operating System (DOS) environment main memory is partitioned into conventional, upper and extended memory. Conventional memory denotes the first 640 Kbytes of RAM, while the upper memory block is the next 384 Kbytes. Memory addressed over 1 Mbyte is referred to as extended memory. Memory can also be added to a system by the addition of expanded-memory boards, in which case it is considered separate from the conventional and extended memories and is called expanded memory [18]. Again, virtual memory implementation is done mainly through the use of the hard disk.

There are software packages that manage the use of these memory areas. Popular names include HIMEM.SYS, which is used so that the area above 1 Mbyte may be shared between applications. The expanded memory manager, EMM386.SYS, on the other hand only allows an application access to a limited amount of memory at a time. This scheme proved to be slower because the number and size of processes that could be brought into memory for execution is reduced.

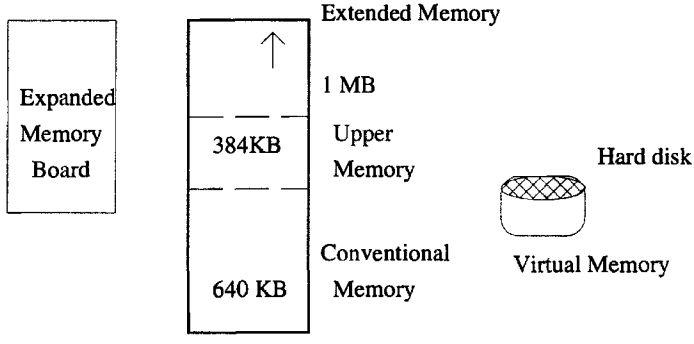

Figure 3.2 Memory organization in PCs.

3.2 Bus Management

In microprocessor based systems such as computers, the bus is a path connecting two or more devices [28]. It is a shared transmission medium, where every device attached to the bus may receive the information, however only one device may transmit at a time. If more than one device tries to transmit simultaneously then contention arises; the signal will be distorted and the information corrupted.

Bus management is based on the simple principle of one client using the server at a time. The server being the bus and the client any device connected to the bus.
The bus may be functionally or physically dedicated. The phrase "functional dedication" means that the bus is used for a particular purpose such as addressing only. The 8086 based microcomputers use a set of address and data lines that are time multiplexed [31]. An address is placed on the bus and the Address Latch Enable (ALE) signal is issued to indicate that there is a valid address on the bus. The 16 data lines are shared with lower 16 address lines. After latching, the address bus is separated from the data bus. Physical dedication refers to the use of several buses to connect small groups of modules. This minimizes the chances of bus contention and yields a higher throughput as it isolates the buses through the use of an adapter to the main bus.

A bus can be managed by an arbiter, also known as a bus controller, which allocates time on the bus for other devices. Since one hardware device is responsible for managing the bus, this is referred to as a centralized approach. Intel's Peripheral Component Interconnect (PCI) bus is an example of a bus architecture incorporating the centralized scheme [28]. On the other hand, each module may have particular control

logic that allows access to the bus. The modules then share the bus. When it needs to use the bus, the control logic designates the module "master", and the receiving or transmitting device as the slave. Therefore only one module controls the bus just as if it were centralized. Whether it is a centralized or distributed arbitration scheme, still, only one module may use the bus at any one time.

Bus management considerations are also a factor in network management which is discussed in the next section.

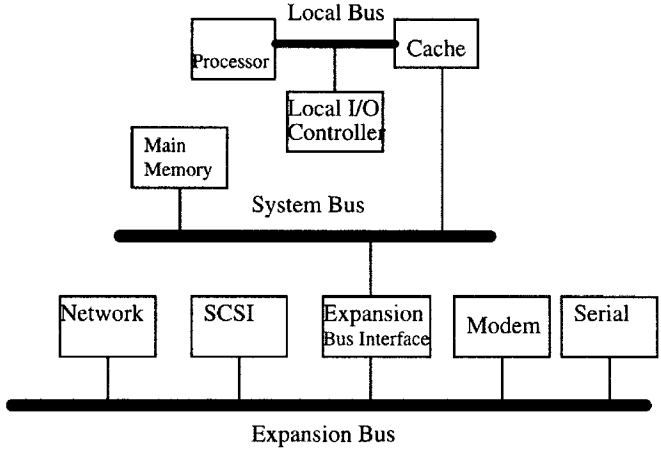

Figure 3.3 Traditional Bus Hierarchy

3.3 Network Management

In the case of networking, users are simultaneously accessing servers and thus the possibility arises for collisions on the network. A good network management system or controller will prevent such scenarios by monitoring the traffic and allocating users to different servers depending on the level of congestion. Networks use protocols to

provide control over network traffic. One widely known protocol is called Carrier Sense Multiple Access with Collision Detect (CMDA/CD), which runs on Ethernet [29] When a station wants to transmit it first tests to see if the line is idle. If it is, it will transmit and wait for an acknowledgement of its transmission. If an acknowledgement is not received in a pre-determined amount of time it will wait a random amount of time and then repeat the transmission if the line is idle. A collision is assumed to have occurred when no acknowledgement is received. The station will cease transmission on detection of a collision and it will send a "jamming" signal to all stations to indicate this. It will then wait a random amount of time before it tries to re-transmit.

This is just one of many protocols that are used to manage traffic on networks. In a data communications network such as a telephone network, circuits can be routed and reconnected through the use of switches. These switches are known as Space-division switches as they allow separate circuits to use the connections of the switch to route to lines of the telephone network [7]. The switches are usually set up in a matrix and allow the inputs to be routed to the outputs of the switches in the next column. The inter-connections continue until the final output connections are established to the outside lines. Any input can reach any output and thus provide full accessibility. See Figure 3.4. This example can be viewed as the management of space - the telephone lines. Of course the outputs cannot be used if they are being held by another connection. The rules governing the priority on connections is another aspect of the space management issue.

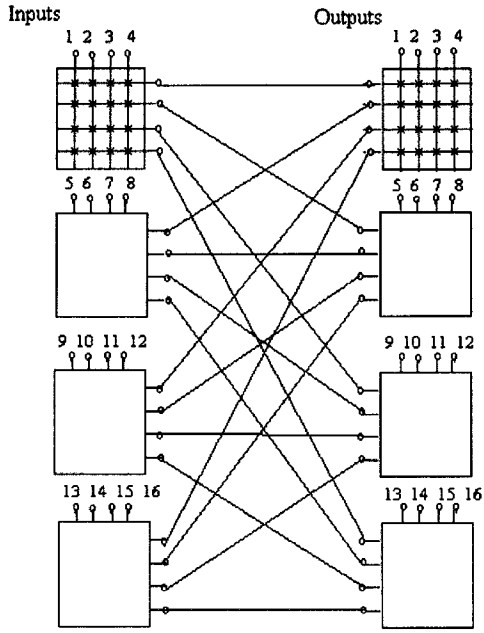

Figure 3.4 Space Division Switch Matrix for telephone network.

3.4 Physical Space Management

There are many situations that depict some type of physical space management from zoning real estate to arranging items on a store shelf. The underlying principle of space management can apply here too, in that if the space is occupied then it is usually not available for use. The method of management is dependent on the situation.

Managing space or seating in a restaurant may seem to be a random process, however there are techniques used to estimate the next available seating. Usually, the wait time is based on the progress and the amount of current customers being served. Obviously if every customer in the restaurant is seated simultaneously, then the wait time

for the next available seat depends partly on the quickest orders. The situation gets more complicated when trying to arrange seating for a group of people. The availability of the required number of seats, whether the arrangement is fixed or seating has to be set up to accommodate the number of customers, is another factor that greatly affects customer service. For example a group of four may arrive at a restaurant where the only two tables left are tables for two at separate ends of the restaurant. It would be too awkward to be carrying tables from one end of a restaurant to another to accommodate seating. The waiting time may now depend on any of the other tables with a higher seating capacity, or even another two-seater close to the ones available.

To help with this problem of seating arrangement, some restaurants have tried implementing a software system that is programmed with the seating arrangement of the restaurant. The tables are numbered as before and the seating capacity of each table is also recorded. The innovation here is the addition of a switch or transmitter, to be more exact, attached to each table. When the table is free, the transmitter is turned on by the waiter; a signal is sent to the computer running the software to indicate its status and of course the table number. The table can then be assigned to any party that might be waiting. The software keeps track of the wait time for seating and helps the host in the overall seating arrangement of the restaurant. A popular restaurant chain, TGI FRIDAYS, is responsible for the implementation of this system; however, due to copyrights the researcher was unable to uncover anymore information than what was stated herein [30]. It is believed, that the transmitter is wireless.

As another example of a physical space management, the parking garage system is studied. There are many parking facilities in place with different management techniques. Some have simple entrance and exit gates with no parking fee included. Those that charge a parking fee may incorporate a manned ticket booth at both entrance and exit gates. Some replace the manual labor at the entrance gate with an automatic gate that issues a ticket upon arrival. This ticket which has the entrance time stamped and punched in on it is then presented at the exit gate to determine time spent and fee owed.

Some techniques use magnetic stripe cards to allow entry and exit into the lot while it monitors time and charge of the car parked. Fee calculation can be done in a few different ways here. One way is, the customer may have a personal or employee parking card with customer billing address or employee information on it. The customer can gain access to the parking area by inserting the card into the card reader/writer at the entrance gate. The arrival time is written onto the card and then the fee calculated at the exit gate by inserting the card into a card reader. The charge can then be billed to the customer's address. These magnetic stripe cards can be used as debit cards also. Instead of billing, money can be deducted from the cards on the way out the exit. If the charge is more than the amount left on the card then a bill for the remainder can be sent out to the customer.

Another may be through the use of a vending machine type pay booth, before returning to the car. Upon entry the card is issued with the time written on it and the customer parks the car. When the customer is leaving the garage, the card is inserted into the pay booth that calculates the fee and writes a code to enable the customer to pass

through the exit gate within a specified time. At the exit gate the customer inserts the card to open the gate and leaves the parking lot. If the customer does not make it out within the time allotted, another fee will have to be paid through the machine [25]. This method is found is Europe in places such as Heathrow airport in London, in the city of Geneva and here in the USA, in Indianapolis, IN.

Once inside the parking garages, it is left to the customer to find a space to park. In small garages it may be easy to identify an available space. However, for much larger garages and at peak service/traffic periods it may take quite some time to find a space. One of the objectives of this work is to aid the customer to find a space without spending too much time to do it.

Chapter 4

Design of Intelligent Space Management System

4. 1 System Design

The space management system designed by the author incorporates basic features that are applicable to a variety of situations. Although some systems may direct users to the general area where space is available not many, if any, detail the exact location. One of the objectives of this space management system is to facilitate the identification of an available space. To accomplish this, sensors are used to monitor the designated space. Certain other features and requirements were laid out in the design of the space management system. They are as follows:

- To reflect the status of each space or the contents of each space.
- To be able to handle/service the object entering the system.
- To provide some measure of security or supervision of the items entering the system.
- To provide overall system status information, such as information about the length of time each space is occupied, or how long one item has remained in same space (flow of items in and out of the system)

The system is made up of different sub-modules. A Main module, a 5-bit counter, a Space Counter module, a Display module, thirty-two Signal_Time modules, one clock generators and ten Mux_scan modules that are the multiplexer-counter combinations. The system receives a Master clock of 273Hz, thirty-two sensor signals and outputs a FULL signal and a serial output. The serial output contains a 16-bit stream of

information reflecting the status of a space. There are three pieces of information embedded into the serial bit-stream, the number of the space, the status of the space and the time, in minutes, that the space has been occupied.

4.1.1 The Display Module

The display module gives the number of the space nearest the entrance that is vacant. It will keep displaying the same number until that space becomes occupied. The sensors are checked by multiplexing the lines through a 32 to 1 multiplexer driven by a five-bit counter. The module is clocked at one-sixteenth the master input clock; however, as a means of providing a more fault tolerant design the counter is driven at half that frequency (See Figure 4.2 and 4.3). This yields more time with which to clock in the sensor signal, latch it and then latch in the space number from the counter output and then reset the counter so that it always checks spaces from the entrance or space number one. Therefore if there are any intermediate signals or glitches from the sensors or the multiplexer output there will be a delay whereby such errors will not be clocked into the outputs.

The display module is an independent of the entire system, except that its clock is taken from the clock generator. It also detects when the system is full by simple ANDing of the sensor inputs. The Full Signal can be combined with the Display and also be shown outside the system to notify other patrons.

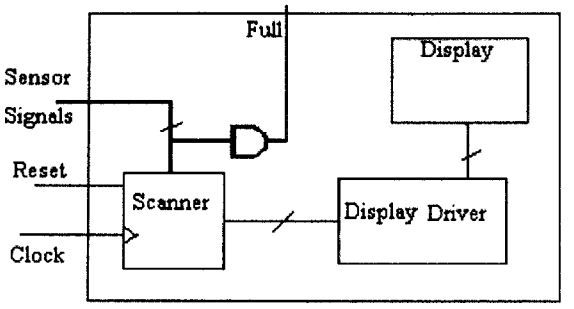

Figure 4.1 Block Diagram of Display Module

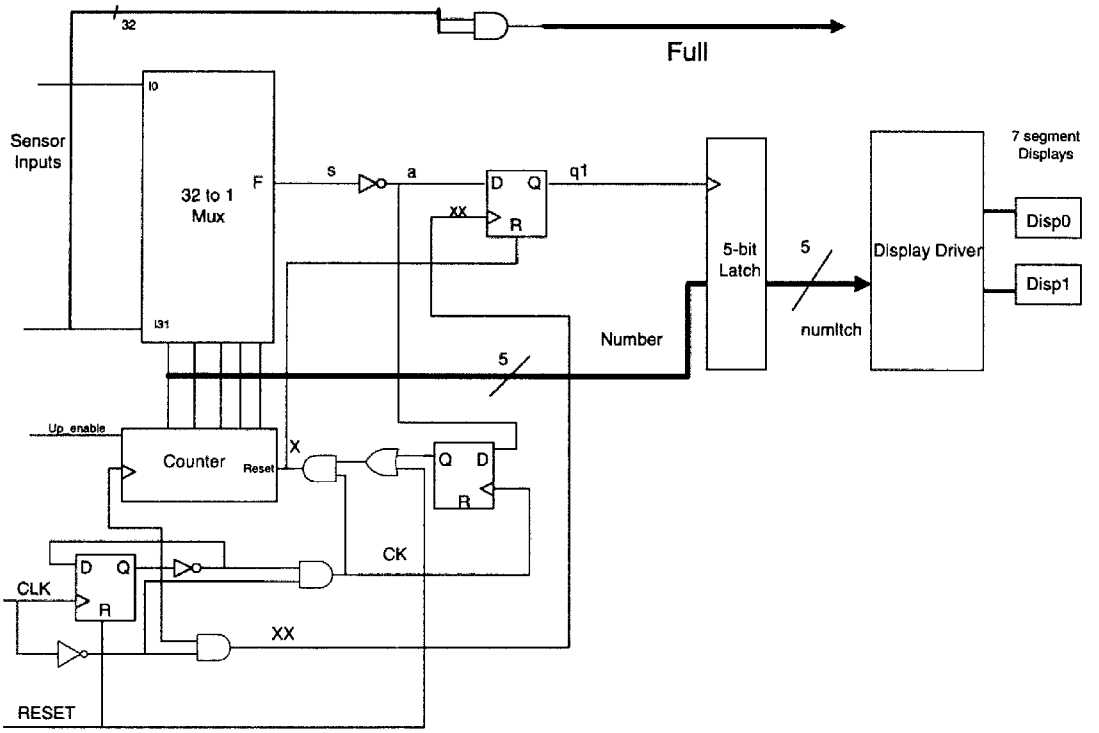

Figure 4.2 Layout of Display Module with Fault Tolerance

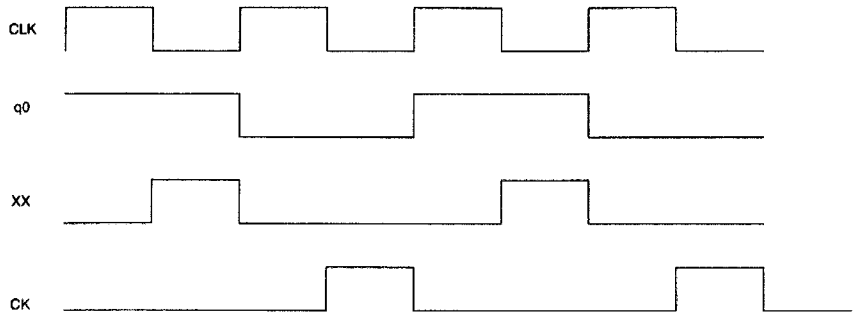

Figure 4.3 Clocking Scheme in Display module to achieve fault tolerance.

4.1.2 Main Module

The other modules are designed so that their outputs are multiplexed into the main serial output. The Main module consists of a 16 to 1 multiplexer that accepts inputs from a five bit counter on inputs I0 through I4, the status bit of that space is connected to I5 and the signal_time outputs for each space is sent to the remaining inputs, least significant first. The counter should be clocked at one sixteenth the master clock so that a complete scan of the multiplexer inputs is done before the counter changes value.

The main module is the coordinator in a sense of all the information passing through the system. It drives a serial output that can be passed into eight bit shift register for serial transmission to a PC, where the information can be stored. This interface was not designed but could also be done by clocking in the output at least the same frequency of the master clock or double try and account for any jitter on the line.

4.1.3 Space Counter

The Space Counter module provides the status bit information to the Main module. It does this by use of the multiplexer- counter combination . The counter in this module is clocked at one sixteenth the master clock to provide synchronized outputs with the counter values that are going into the Main module.

4.1.4 Signal _Time Module

Each space has a signal_timer module associated with it. When a space becomes occupied, the sensor signal becomes high, the timer is triggered and begins to count. It counts in steps of one minute. The clock input is derived from a clock generator that provides a one minute period by dividing the master clock by sixteen and then by one thousand and twenty-four. Notice that the timer outputs are ten bits long. This means that each space can be monitored up to $2^{10} = 1024$ count. Now if the timer is clocked every minute it can yield a maximum of 1024/60 hours. This will be enough time in a parking lot that has daily hours and is closed at night. Also if the parking is limited to a certain amount of time the space can be identified and the car ticketed for overtime charges, thereby increasing revenue.

In order to obtain this one minute clock period the Master clock is then obligated to be about 273Hz (ie. 273Hz = 16*1024/60secs). Each timer is reset at the end of the main multiplexer scan cycle. The count driving the main multiplexer is NORed to obtain a '1' everytime it goes to zero. This signal is ORed with the Master Reset and tied to the

timer's reset. In this way, the time being read is not reset by an event on the sensor before it has a chance to be transmitted properly through the Main Module.

The task still remains to properly send these timer outputs into the Main module. This is done by again multiplexing the respective output bits of each timer into ten multiplexers. One multiplexer for every bit of the timer. Thus, the first multiplexer will accept bit zero from timer of space number one, then bit zero from timer of space number two, etc and pass its output to input I6 of the multiplexer in Main module. The second multiplexer will accept bit one from all timers and pass that into input I7 of the main module multiplexer and so on.

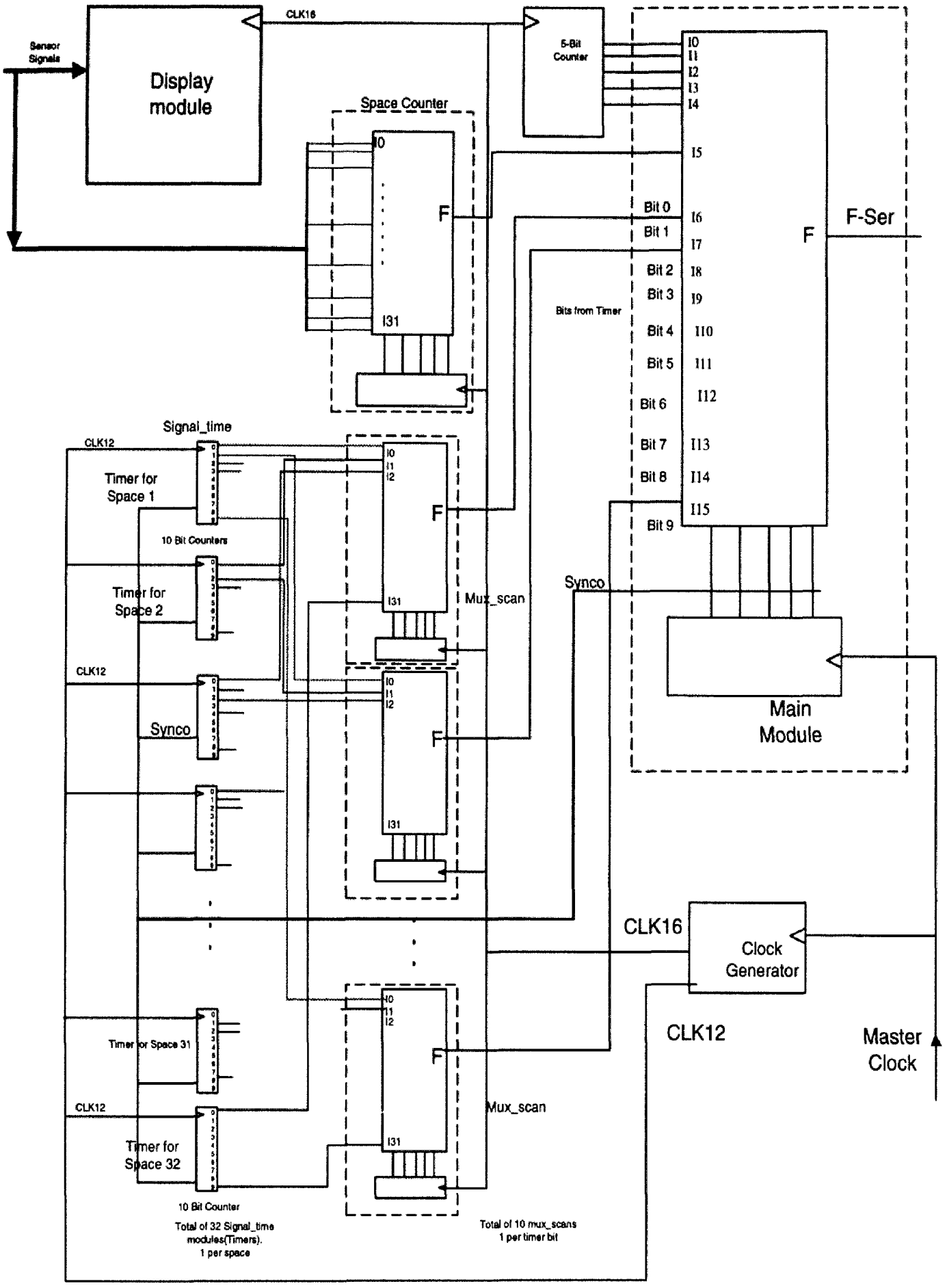

Figure 4.4 Diagram of Sub-module Interconnections in prototype.

4.2 Sensors

As stated before, the monitoring of each space is done by sensors. There are many different types of sensors that could be used for monitoring objects in physical space. Pressure sensitive devices can be useful in cases where heavy objects are being monitored. Pressure sensors are used to detect passing cars and the traffic flow on the road. Loop detectors are used on roadways to detect passing cars as they approach traffic signals or entrance and exit gates of parking lots. Pressure sensors are also used at these entrance and exit gates. Proximity sensors that use sonar are useful in detecting how close an object is or can be used to identify distance between walls.

In this design, the sensors are expected to provide a digital (DC) output voltage signal to reflect the status of the space. Specifically, if the space is occupied, the ouptut voltage level is high or 5Volts, otherwise if the space is free, the voltage level is low or 0Volts. So the transducer should produce a DC voltage from whichever type of sensor is chosen. For the application of a parking lot controller the sensor chosen was a diffuse reflective infra-red sensor. The detection of an object in space was accomplished by transmitting an infra-red beam in the direction of the object and detecting the reflected beam as shown in Figure 4.5. If no reflection is detected then it is assumed that no object is present and the receiver/sensor output is zero Volts. However, upon detecting the infra-red signal the output of the sensor would be 5 Volts. The main reason for choosing this infra-red sensor is cost. It is relatively cheap with an estimated retail price of thirty dollars. Besides the cost, the sensor's performance is also a factor. This transmitter-receiver pair gives fairly reliable output as far as detecting an object within several

inches. If need be it can be made to detect a signal from a maximum of eight feet away. Most cars are about fifteen feet long and standard parking spaces are about eighteen feet long, therefore the car may be, at the most, only a few feet away form the sensor [4]. This is still within sensing range of the sensor. Furthermore, once the object is detected, any other object passing momentarily between the sensor and the car, would still reflect the infra-red signal back to the receiver. In this way, the signal is not interrupted and still indicates an object in the space.

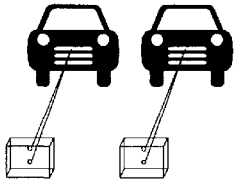

Figure 4.5 Diffuse Reflection of the Infra-red Sensor.

4.3 Implementing with VHDL

The design was coded using the Viewlogic's version of the VHDL language. This does not adhere one hundred percent to the IEEE standard but still encompasses most of the constructs of the language [33].

The multiplexer entity was described using the behavioral constructs of the language. The code described a 32 -input multiplexer by using IF statements to decide what the output should be depending on the select lines. The structural or dataflow description of this multiplexer would entail large amounts of code and possibly a much

more complex a design.

The counter was created using a feature of Viewlogic's standard, the pla_table. This allows the designer to create a present and next state table which can be used with latches or flip-flops. Also provided in the procedural library were the descriptions of D flip-flops which could easily be incorporated into the design..

The five bit counter was reused in creating the ten bit counter. the five bit counter became a component of the ten bit counter entity *timers3*. Each were described in the dataflow abstraction.

A five input nor gate, mnor5 was created using the concurrent signal assignment. The code describing these and other enties used can be found in Appendix C.

4.4 Hardware Implementation

The space management system could be implemented with standard logic gates. The size and quantity of the necessary devices would create a physically large system that could be more difficult to decode because of the complexity of the wiring. Remember, the design called for a 32-input mux and a 5- bit counter, each of which would be used approximately twelve times in the case of the mux and about fifty-five times for the 5-bit counter. Nowadays, digital systems design is moving towards more small scale and low power devices to handle complex circuitry. Custom-made devices such as PLDs and FPGAs would be the more likely route to implement this design.

4.5 Functional system using FPGA Technology

Field Programmable Gate Arrays (FPGAs) are fast becoming a useful part of the design and manufacturing process. This is part of the effort by industry to pursue more efficient means of designing systems. With the old method of system design most of the implementation was done through the use of standard logic Integrated Circuits (ICs). See Figure 4.6. Some designs incorporated custom or semi-custom made ICs that performed particular functions. Usually all system designs start off with specifications and a system block diagram. The features and requirements of the system are presented within the specifications. Then the design was partitioned and each section was described in as much detail as necessary, usually down to the logic level. Once the sections were correctly designed they would be integrated. If any software was required it would be written at this stage.

A prototype would be constructed using breadboards and it would be tested along with the software and revised if necessary. Difficulties could arise at this point especially if there was any integration with a larger system. The compatibility of the interface, speed and software can influence the integration. It could take anywhere from six to twelve months to get to this testing stage. The next stage would be to create the printed circuit boards (PCB). Testing would continue during the manufacturing of the boards which could sometimes lead to design changes and as a result a number of PCB fabrication attempts would take place. The time-to-market could be anywhere from two to three years when this route is taken.

In creating a PCB, many considerations must be taken into account. These include the size of the board, the maintainability of the design and board, and the cost. Earlier PCBs were manufactured using standard or customized logic ICs. This method, however, proved to have some disadvantages due to the technology used to implement it. Some of the disadvantages of standard logic ICs are

- large power consumption
- large physical space needed
- special ventilation and cooling required
- a high failure rate due to the large number of individual ICs [8].

In an effort to steer away from these disadvantages, other forms of technology were developed and eventually brought to the consumer, namely, the designer. Programmable Logic Devices such as ROMs, EEPROMs, PLAs and Gate Arrays opened the door for more customer specific system designs, giving engineers an edge in PCB production. Gate Arrays are semi-custom ICs with functional cells. Although the designers may target their designs towards a certain Gate Array, it is the manufacturer that fabricates the final design product. FPGAs go a step further in the design process as they give the designer almost total control throughout the design and manufacturing stages.

The new approach in system design can be described as follows: As is customary, the system specifications and diagram is presented. (See Figure 4.7.) The next stage is similar to the old process in that it partitions the complete system into blocks, but differs in that the blocks are mainly functional modules, such as microprocessors,

PLDs, FPGAs and some interface logic. A high level description is done with the aid of schematic capture software or some abstract design language such as VHDL. Simulation of the system follows once the description is complete. When the simulation presents an accurate functional picture of the design a netlist may be extracted. This netlist is used in creating the PCB. As before, during the PCB production stage, revisions can be made to the design. However, with this method any change that arises could be effected through the system software, the PLDs or the FPGAs and would not affect the PCB design. The complete process time could take anywhere from three to six months; a dramatic reduction compared to the old method.

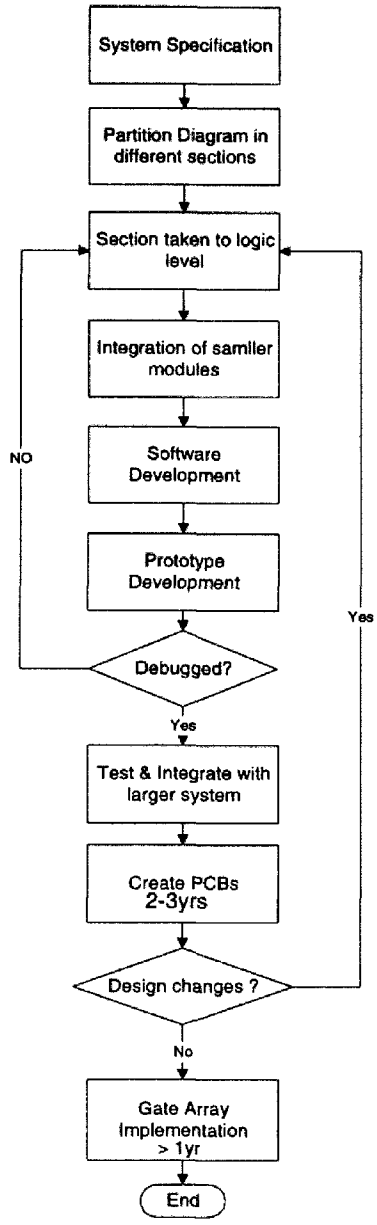

Figure 1 . Design Flow of Previous System Design Process

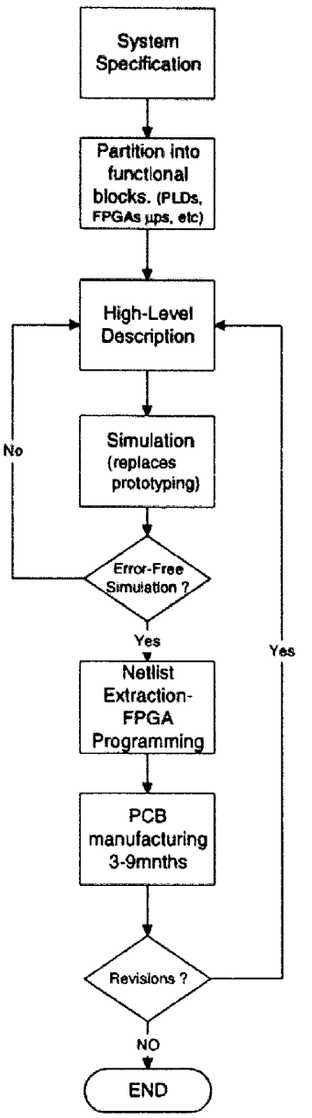

Figure 2 Design Flow of New System Design Process

4.5.1 Contents of an FPGA

The term FPGA was coined after the first Xilinx® Logic Cell Array (LCA) was produced; industry did not accept the term LCA [13]. FPGAs are comprised of building blocks that are universal functions. That is, the functions can be used to make any logic function. The building blocks are called Configurable Logic Blocks (CLBs). In addition to the CLBs are I/O blocks (IOBs) and interconnect network. (See Figure 4.8.) The CLB is made up of flip-flops, multiplexers and a combinatorial logic block. (See Figure 4.9.) The CLBs are laid out in a matrix and can be connected horizontally and vertically throughout the LCA by what are known as Programmable Interconnect Points (PIP). There are general purpose lines that run short lengths of metal lines vertically and horizontally, through the PIPs, forming a grid known as a switching matrix. The switching matrix allows many different interconnections to be possible between the CLBs and other elements. By the particular connection of these PIPs and the switch matrix, a certain routing is obtained between the elements and the CLBs, thereby programming the FPGA. Typically, a CLB output may be routed through a PIP to a switch matrix, out to another element. Throughout the LCA, there are also long lines that traverse the length and width of the die. However, these do not intersect with the switch matrix. The Input/Output block provides the interface between the external device pins and the internal signals, again via the interconnection network.

With the aid of modern design software FPGAs take on a quality similar to a Gate Array by which the design can be repeatedly programmed into the device. The design software must perform two important tasks. It must first translate the design. Translation

is a conversion of the functions of the design into the cells (functions) of the particular FPGA. Its second task is to verify that this translated design is correct.

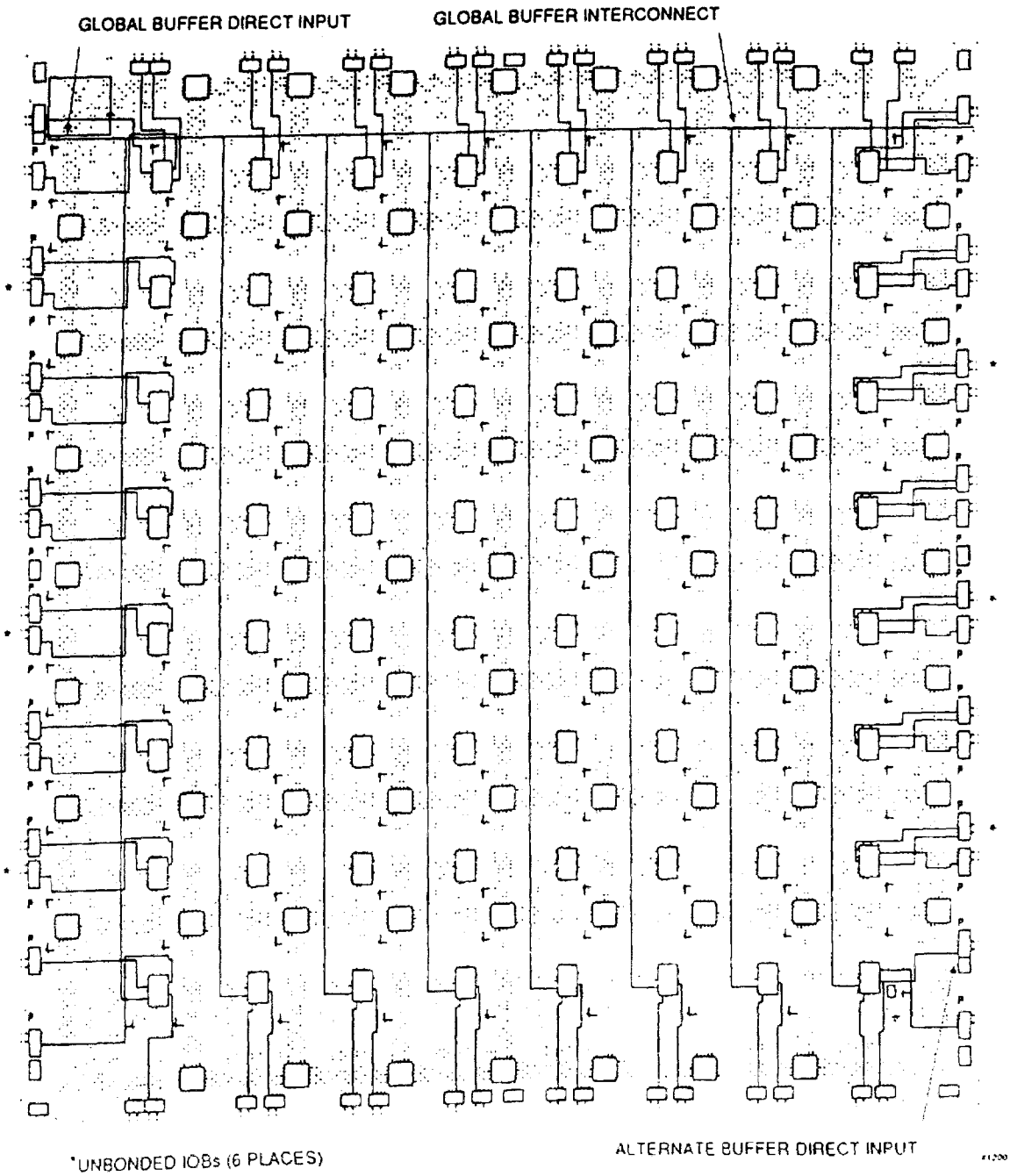

Figure 4.8. Schematic of Layout for LCA. (Courtesy Xilinx . [13])

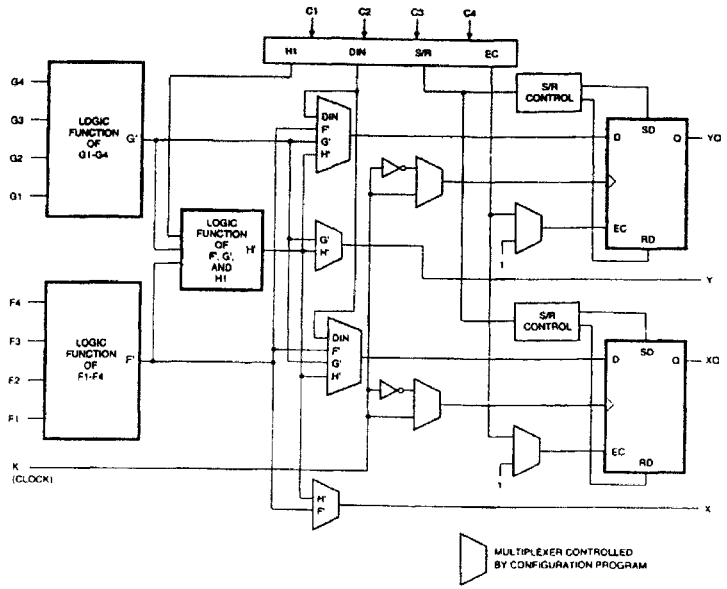

Figure 4.9. Schematic of CLB architecture for the XC4003A FPGA

4.5.2 Advantages of FPGAs

There are several advantages to be gained in the design process by using FPGAs. Unlike some other technologies, FPGAs cut back on the time and expense of redesigning incorrectly printed circuit boards. The fact that the user programs the FPGA, saves time that would be spent in the manufacturing process. The time-to-market cycle is greatly influenced by using FPGAs. Shortened time-to-market cycles are now between three and nine months. Since the programming process is software driven they can be reprogrammed with new design changes very quickly. Although each FPGA may be relatively expensive, they do provide for cost efficiency by eliminating the overhead

costs, such as manufacturing the redesigned boards; the overtime cost of labor in redesigning boards. Also, designers' efforts can be applied in other areas instead of the redesign process.

The example given by Jenkins [13] clearly shows the advantages that FPGAs bring to the design process. The author gives an account of how three manufacturers that had produced disk controller cards targeted for the yet unreleased SCSI bus specifications, dealt with the revised version that was finally and officially released.

Only one company, Company C was spared the agony of returning to "square one". The other two companies, call them Company A and B, had to make major design changes to their product. Company A had 90% of its design embedded in gate array technology and the rest in TTL bus interface logic. It had to take its product off the market and redesign the board completely to account for the changes. Company B had to used a microprocessor- based design with standard logic ICs. It tried to depend mostly on software (90%) to support the design. The drawback in this case was that the software could not adapt to the changes in speed that the new specification had defined. Therefore, Company B had to also redesign its board.

The company that held up best to the changes had a different approach. It used an FPGA to accomplish the speed-sensitive protocol logic necessary for the SCSI bus. The remainder of the design was implemented with a micro-controller. Since the changes applied to the FPGA portion of the design, it would be easy to make amends to it and simply reprogram the FPGA with the new version of the design. And that is exactly what Company C did. For boards that were already in the customers' hands, the new

parts were sent out to them. Company C saved on the time and costs of manufacturing new boards. The final result is that instead of throwing away boards that don't work, the FPGA is simply reprogrammed with the new design.

Summary of Advantages of FPGAs

1. Reduces the design cycle
2. Reduces cost of manufacturing boards
3. Reduces time-to-market cycle
4. Reduces number of board rejections
5. Makes it easier to implement design changes

4.5.3 FPGA Implementation

In the final implementation of this work the target technology will be the Xilinx XC4003A series FPGA. This part was chosen on the basis of availability and also because of its more advanced features. The XC4000A series is a third generation LCA with enhancements to the logic and I/O block functions as well as the interconnection options. The XC4003A contains one hundred CLBs, and with thirty-two storage locations per CLB this device can support a 3,200 bit RAM. It has eighty IOBs with sixteen user-defined I/O pins, an additional eight switch inputs and sixteen LED outputs. The layout of the CLB and I/Os are made more symmetrical in the form of a matrix or grid appearance to facilitate the interconnections to the switch matrix. See Figure 5 for

the arrangement of the switch matrix. The total gate count is approximately 3,000 enabling the LCA to handle larger designs than previous series.

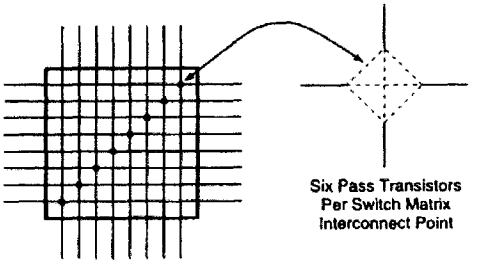

Figure 5. Switch Matrix architecture for XC4003A

CHAPTER 5

RESULTS

5.1 Simulation

The VHDL designs were simulated using Viewlogic's ViewSim Simulation software. The resulting waveforms can be found in Appendix B. The simulations were accomplished by running a batch file specifying the varying inputs and the clock signal. The simulated clock pulse for each module was modeled after the expected clock input frequency. The designs proved true where simulation results are concerned.

One thing that deserves mentioning, as far as the simulation process is concerned, is the apparent glitch that occurs when more than one bit at a time changes in a vector. There is an instantaneous glitch between such transitions and this can be seen even with the vector waveforms generated through the "wfm" command of the simulator package.

This may be a benign property; however, if such a vector is used to clock or drive another, there may be a glitch on the output where both vectors simultaneously have a Hamming distance of at least one between their current and next values. A good example of this was seen in the case of the multiplexer-counter combination module, mux_scann. The author notes that the behavioral description of the module may have also contributed to the glitches. When the multiplexer input was changing from 00000H to 00001H and the value of the counter was also changing from 00000H to 00001H, the output of the multiplexer had the spike exactly at the mid-points of the transition on the rising edge of the counter's clock. See Figure 5.1. What seems to have happened here is that the

outputs of the counter affect the output of the mux slower than the direct multiplexer inputs, because the select lines of the mux has a longer propagation delay than the mux inputs. Thus, for an instant, the multiplexer output was a '1', but then the counter values changed and settled and the multiplexer passed the 'new' value, '0', of the input selected by the new counter value.

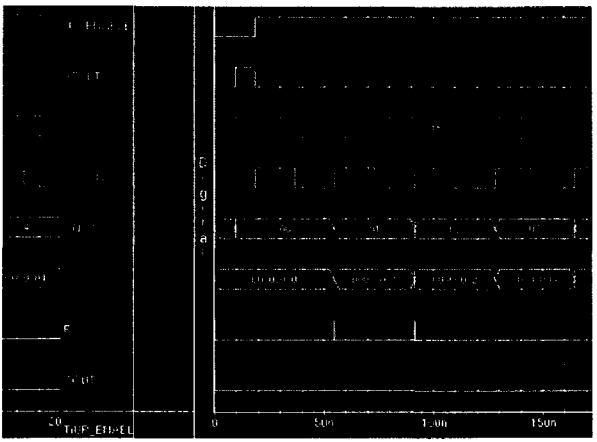

Figure 5.1 Waveforms with Glitches in Mux_scan

Different methods of implementing a more glitch-free module were tried and tested using the WAIT statements. The wait statements delayed the function of the mux by 1 ns and simulation was without error. However, these designs were not synthesized because Viewlogic's synthesis software does not support the use of WAIT statements and thus no netlist could be created to incorporate into the FPGA implementation [33]. The propagation delay and other attributes of the gates and other devices in the synthesized schematic may be changed to achieve the desired timing results. This is a more involved approach and was not pursued for this work. The propagation delays may

be set as generic also in the VHDL code listing, but here again timing does not translate to the synthesized schematic.

Furthermore, during simulation, the correct periods and pulse widths should be calculated and applied to reflect the real-time use of the design. The functionality of the system can be best verified in this manner. Of course, it is always good practice to investigate the behavior of the system with unexpected inputs. Using information gathered in these tests can be used to design a more fault tolerant system.

5.2 Synthesis

The synthesized models were generated with the XC4000 library as its target technology. For the most part synthesis was error free, except for a minor problem in the Display module. In the description of the display module, the inverted output of the multiplexer was originally used to reset the counter. The attempted synthesis of this module revealed a network cycle within the design. It was not clear how this cycle manifested, except that the counter being reset was the same counter driving the multiplexer select lines. The thinking here is that this may have been interpreted as a loop of some sort by the synthesis algorithm in its attempt to detail the netlist.

The solution to this network cycle came in the guise of a D-Flip Flop. If there was a loop along the path from the multiplexer output to the counter reset, the flip flop would then be the best way to buffer or interrupt that loop. Thus Display module was modified accordingly, where the input to the D flip flop was the output of the multiplexer; the output of the flip flop drove the reset of the counter and the flip flop was clocked by the

negative edge of the clock driving the counter. Using the negative edge of the clock gave the counter and the multiplexer outputs more than enough time to settle. Once this was implemented the network cycle error was not encountered again and the synthesis of the display module was completed. Please see the synthesized schematics in Appendix C.

5.3 Implementation

As part of the synthesis process, the IOPAD program was invoked to produce input and output pins for the design. The iopad feature allows these designated pins to be allocated on the target device, in this case the FPGA. These pins can then be assigned to preferred I/O pins on the device. This can be done through the schematic capture program ViewDraw. The pin assignments were done for the Display module design and translated to the Xilinx netlist file of type .xnf.

The Xilinx netlist file then needs to be prepared into a bit stream which is downloaded into the XC4003A FPGA on the demo board. Once this was accomplished the proper external connections were made; however, the system seemed to work only for a portion of the sensor inputs. The Display module had thirty-two sensor inputs and a 5-bit maximum count input number as well as reset, clock and an up_enable counter control line. When the system found space number 8 or higher empty, the display would exhibit incorrect behavior. It should continuously display only the empty space it has encountered, instead it jumped between certain values, all still within the maximum count range. The counter outputs were also checked and found to be counting correctly.

The output of the multiplexer then seemed to be the next possible suspect. This output was connected to an oscilloscope and a glitch was indeed occurring on the output at the very same time that correlated to the latched value being displayed. Thus it seemed that for some unknown reason, although the sensors (DIP switches) were not changing, the output of the multiplexer did change.

Even more sophisticated latching and clocking was implemented to try and settle the output of the mux. Unfortunately the results were not rewarding. By latching the multiplexer output, the system was able to count up to space number 14 without errors. Upon finding space number 15 empty the display showed that value then almost immediately went to "1". This suggests another trigger of sorts that latched in a zero from the counter. Note: a zero on the counter translates to space #1 , etc. It is more important to note that when the additional latching features were implemented the display module design could not be contained in one FPGA. The design had to be split into two separate modules and interconnected to each other as necessary. This implies that the design may have already been close to capacity of the FPGA. Indeed the report listed 100% utilization of all 61 input/output pins. This could have caused loading on the inputs and outputs of the device. About 78% of the CLB function generators were used on the FPGA and only 6% of the CLB flip flops.

The complete space management system was not implemented with the FPGA. Judging from the utilization report above it would be safe to say that it could not have fit into that particular model. The entire design repeatedly used the multiplexer-counter combination to describe its functionality and would need approximately eleven more

similar FPGAs to accommodate the eleven mux_scan modules, and several more yet to accommodate the thirty-two 10-bit timer modules.

CHAPTER 6

CONCLUSIONS AND RECOMMENDATIONS

6.1 Summary of Work

This work demonstrated the capabilities of VHDL in designing a digital system. VHDL behavioral descriptions were written for a 32-to-1 multiplexer, a 5-bit counter with programmable maximum count, a 10-bit counter, a BCD to 7-Segment Display Driver, and a 5-bit Binary to BCD converter. These semi-complex logic circuit descriptions became the building blocks for the entire system. Code for one module could be used and adapted in different parts of the system. This follows along the new industrial trend of re-using code to reduce the design time of the process.

Synthesis of these modules in the Xilinx XC4000 Library was fairly straightforward and uncomplicated. The resulting netlists' reports showed no violations in neither the input loadings nor the output drive calculations per net. The time it takes from the design is described to the actual hardware is sped up significantly, because of the ability to generate the netlist and a working prototype. This reduced time is one of the main advantages of the ASIC and digital circuit design process.

6.2 VHDL in Industry and Education

It took almost ten years from the first workshop on standardization, but VHDL has finally matured into a formalized and practical design tool. VHDL is fast becoming the common tool for circuit and ASIC designs in industry. The portability of the VHDL

netlist to an ASIC configuration software is another key feature that incites this new design process to success. Many engineering companies have customized their in-house design software to work with third-party vendors' VHDL packages. In light of this migration towards a more standardized design process, companies are actively seeking engineers with VHDL and ASIC experience. In order to give its graduates a fighting chance in the ever-changing field of engineering, educational institutions will have to introduce this new design technology into the curriculum. Today's engineer should not only know the theory behind digital hardware design but also the current and state-of-the-art tools to implement that theory.

6.3 Future Work

The system was designed with the intention of having it transmit serial data to a main computer, the monitor. This monitor-system interface was not developed due to time constraints and the complexity of the task. The monitor would be able to connect to the space management system by an RS-232C or similar personal computer interface. The software driving this communication would be able to accept continuous input from the system and store the information on a daily basis. The program would also allow the programming of the system in terms of how many spaces are to be monitored. Any attendant wanting information about the status of the spaces being monitored would simply initiate a display process that would give a flat, on-screen graphical representation of the spatial arrangement. This display would also give the user access to the status and

time of occupation for each space. The software may be written using the C+/C++ Languages.

The serial output of the space management system may be shifted into the monitor by a clock one-half times the Master clock. This ensures that for every transmission clock one bit is sampled in the middle of its pulse width to ensure no change in the values are occurring.. Each 16-bit stream holds information about one particular space.

Another enhancement that could be attempted is the configuration of several of these systems as modules in a token ring network. This would lend to the expansion of the system in an environment where more than just thirty-two spaces exist.

Also, in the future, with more sophisticated synthesis tools and increased capacity FPGAs, the system may be able to fit completely into just one device.

6.4 Other Applications

The system design was applied to a parking lot scenario; however, it is not limited to that application alone. The space management system could be applied in large warehouses, for example, furniture storage areas. It could be applied to movie theater seating, where by the empty seat number would help patrons to find a seat in dark and crowded cinemas. The diffuse-reflective sensor might not be useful in this environment but pressure sensors for the seats or imaging sensors may be more suitable. There could also be "triggers" on the seats when they are pulled down, signifying that someone is seated in that chair.

REFERENCES

[1] Armstrong, James R., and Gail F. Gray. *Structurred Logic Design with VHDL.* New Jersey: Prentice-Hall, Inc., 1993.

[2] Armstrong, James R. *Chip-Level Modeling with VHDL* . New Jersey: Prentice-Hall, Inc., 1989.

[3] Bhasker, J. *A VHDL Primer.* New Jersey: Prentice Hall, 1994.

[4] Chrest, Anthony P., Mary S. Smith and Sam Bhuyan, *Parking Structures: Planning, Design, Construction, Maintenance, and Repair.* New York: Van Nostrand Reinhold, 1989.

[5] Coelho, David R. *The VHDL Handbook.* Boston: Kluwer Academic Publishers, 1989

[6] Collin, Serge. *Computers, interfaces and communication networks.* Trans. John C. C. Nelson. Paris: Prentice Hall International Ltd., 1990.

[7] Davies, Donald W. and Derek L.A. Barber. *Communication Networks for Computers.* London: John Wiley & Sons, 1973.

[8] Garrod, Susan A. R. and Robert J. Borns. *Digital Logic: Analysis, Application & Design.* Philadelphia: Saunders College Publishing, 1991.

[9] Gross, Donald and Carl M. Harris. *Fundamentals of Queuing Theory.* New York: John Wiley & Sons, 1985.

[10] Hord, R. Michael. *Digital Image Processing of Remotely Sensed Data.* New York: Academic Press, 1982.

[11] Hutchinson, David. *Local Area Network Architectures.* Workingham, England: Addison Wesley. 1988.

[12] *IEEE Standard VHDL Language Reference Manual Std. 1076-1987.* IEEE Publications, 1987.

[13] Jenkins, Jesse H. *Designing with FPGAs And CPLDs.* Englewoods Cliffs, NJ: Prentice Hall, 1994.

[14] Kashyap, B. R. K. and M. L. Chaudry. *An introduction to Queueing Theory.* Kingston, Ont. Canada: A & A Publications, 1988.

[15] Lipsett, Roger. *VHDL: Hardware Description and Design.* Boston: Kluwer Academic Publishers, 1989.

[16] Lyons, Jerry L., *The Designer's Handbook of Pressure-Sensing Devices.* New York: Van Nostrand Reinhold Co., 1980.

[17] Marschner, F.E. *Evolutionary Processes in Languages, Software and Design.* In *VHDL for Simulation, Synthesis and Formal Proofs of Hardware.* Ed. Jean Mermet. Netherlands: 1992. 1-13.

[18] Microsoft Inc. *Microsoft Windows 3.1 Users Manual.* Microsoft Publications, 1992.

[19] Medhi, J. *Stochastic Models In Queueing Theory.* San Diego: Academic Press, Inc., 1991.

[20] Navabi, Zainalabedin. *VHDL: Analysis and Modeling of Digital Systems.* New York: McGraw-Hill, 1993.

[21] Newell, G. F. Applications of Queuing Theory. London: Chapman and Hall, Ltd., 1975.

[22] Panico, Joseph. *Queuing Theory: A Study of Waiting Lines for Business, Economics, and Science.* Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1969.

[23] Pickholtz, Raymond L., ed. *Local Area & Multiple Access Networks.* Rockville, Maryland: Computer Science Press, Inc., 1986.

[24] Saunders, Larry. Telephone Interview. 1996.

[25] Schwarz, Harvey. Telephone Interview & literature. OH: Parking Systems and Analysis. 1995

[26] Shahdad, Moe. Telephone Interview. 1996

[27] Seippel, Robert G. *Transducers, Sensors, & Detectors.* Reston, Virginia: Reston Publishing Company, Inc., 1983

[28] Stallings, William. *Computer Organization Architecture-Designing for Performance.* New Jersey: Prentice Hall, 1996 4th ed.

[29] Stallings, William. *A Tutorial on the IEEE 802 Local Network Standard.*1986

[30] Unidentified manager of TGIF. Telephone Interview. Miami: 1995

[31] Wunnava, Subbarao. *The 8086/8088 Family of Microprocessor Software, Hardware and System Applications*. New York: Delmar Publishers, 1992.

[32] Viewlogic Systems, Inc. *Viewlogic Powerview® Tutorials*. Boston: Viewlogic Systems Inc., 1992.

[33] Viewlogic Systems, Inc. *VHDL Reference Manual*.Viewlogic Systems, Inc., 1991.

APPENDIX A

VHDL CODE LISTINGS

FOR THE SPACE MANAGEMENT SYSTEM

```
ENTITY mnor5 IS
        PORT (s3,s2,s1,s0,b3 :IN vlbit; norout :OUT vlbit);
END mnor5;

ARCHITECTURE dataflow OF mnor5 IS

SIGNAL x: vlbit;

BEGIN

        x  <= s3 OR s2 OR s1 OR s0;
  norout <= b3 NOR x;

END dataflow;
```

Figure A.1 VHDL code for 5-bit NOR gate.

```
--      cnt_in.vhd
--      Complete 5-bit counter with Programmable PLC as Counter limit
--      Concurrency; with reset.
--      Used in mux5_scan.vhd
--      rev. Nov. 7, 1995
library synth;
use synth.stdsynth.all;

ENTITY cnt_in IS
PORT (SIGNAL RESET, UP_ENABLE,CLK:IN vlbit;
        PLC :IN vlbit_1d (4 downto 0);
        signal c_out                :out vlbit;
        signal bcd          :out vlbit_1d(4 downto 0));
end cnt_in;

Architecture first of cnt_in Is

constant tbl: vlbit_2d(0 to 31, 9 downto 0):= (
        B"00000_00001",
        B"00001_00010",
        B"00010_00011",
        B"00011_00100",
        B"00100_00101",
        B"00101_00110",
        B"00110_00111",
        B"00111_01000",
        B"01000_01001",
        B"01001_01010",
        B"01010_01011",
        B"01011_01100",
        B"01100_01101",
        B"01101_01110",
        B"01110_01111",
        B"01111_10000",
        B"10000_10001",
        B"10001_10010",
        B"10010_10011",
        B"10011_10100",
        B"10100_10101",
        B"10101_10110",
        B"10110_10111",
        B"10111_11000",
        B"11000_11001",
        B"11001_11010",
        B"11010_11011",
        B"11011_11100",
        B"11100_11101",
        B"11101_11110",
        B"11110_11111",
        B"11111_00000"
          );
signal incz,bcd1,muxz :vlbit_1d (4  downto 0);
signal int :vlbit;
```

```
begin
        pla_table(bcd1, incz,tbl);
        int     <= '0'              when bcd1 = plc else '1';
        muxz    <= B"00000"         when bcd1=plc else
                        incz        when('1' =up_enable) else
                        bcd1;
        c_out   <= '1'  when ('1' =up_enable AND int='0')else '0';

        dffc_v(muxz,reset,clk,bcd1);

        bcd<=bcd1  ;

end first;
```

Figure A.2 VHDL code for 5-bit counter.

```
library synth;
use synth.stdsynth.all;

entity div16 IS
 PORT (SIGNAL reset,Mclk:IN vlbit;SIGNAL clkd16:out vlbit);
end div16;

ARCHITECTURE structural OF div16 IS

component cnt_in
        PORT (SIGNAL RESET, UP_ENABLE,CLK:IN vlbit;
        PLC :IN vlbit_1d (4 downto 0);
        signal c_out                    :out vlbit;
        signal bcd          :out vlbit_1d(4 downto 0));
end component;


SIGNAL qc0,qc1,c4,c16,c32,c64 ,up:vlbit;
Signal plc ,bcd:vlbit_1d(4 downto 0);


BEGIN
        up<= '1';
        plc <= ('0','1','1','1','1');
 d16:  cnt_in Port MAP(reset,up,MCLK,plc,c4,bcd);
        clkd16 <= NOT BCD(3) ;

end structural;
```

Figure A.3 VHDL code for Divide by 16 counter.

```
 library synth;
use synth.stdsynth.all;

entity div1024 IS
        Port (Reset,CLK:IN vlbit; clkd1024:out vlbit);
end  div1024 ;

Architecture structural of div1024 Is

component timers3
        port (reset,up_enable,Clk:In vlbit; c_out,c_out2 :out vlbit;
        final_count:out vlbit_1d(9 downto 0));
end component;


Signal up,c1,c2,q,x:vlbit;
Signal times: vlbit_1d(9 downto 0);

Begin

        up <= '1';
d1024:  timers3 Port MAP(reset,up ,CLK,c1,c2,times);
        clkd1024 <= NOT times(9) ;

end structural;
```

Figure A.4 VHDL code for a Divide by 1024 counter.

```
library synth;
use synth.stdsynth.all;

entity bin2bcda IS
  Port  (number:in vlbit_1d(4 downto 0); bcd1,bcd2:Out vlbit_1d(3 downto 0));
end bin2bcda;

-- this is modified to show 1 for zero input -> don't count from zero up
Architecture behavior of bin2bcda IS
constant tbl: vlbit_2d(0 to 31, 12 downto 0 ) := (
          B"00000_00000001",
          B"00001_00000010",
          B"00010_00000011",
          B"00011_00000100",
          B"00100_00000101",
          B"00101_00000110",
          B"00110_00000111",
          B"00111_00001000",
          B"01000_00001001",
          B"01001_00010000",
          B"01010_00010001",
          B"01011_00010010",
          B"01100_00010011",
          B"01101_00010100",
          B"01110_00010101",
          B"01111_00010110",
          B"10000_00010111",
          B"10001_00011000",
          B"10010_00011001",
          B"10011_00100000",
          B"10100_00100001",
          B"10101_00100010",
          B"10110_00100011",
          B"10111_00100100",
          B"11000_00100101",
          B"11001_00100110",
          B"11010_00100111",
          B"11011_00101000",
          B"11100_00101001",
          B"11101_00110000",
          B"11110_00110001",
          B"11111_00110010"  );

signal bcda,bcdb:vlbit_1d(7 downto 0);
signal num:vlbit_1d(4 downto 0);

begin
     pla_table(number,bcda,tbl);
          bcd1 <= bcda(3 downto 0);
          bcd2 <= bcda(7 downto 4);
end behavior;
```

Figure A.5 VHDL code for Binary to BCD conversion.

```
--      BCD27SEG.VHD
--      Binary to 7 segment display decoder
--      Process -sequential IF statements
--      rev. OCt. 25, 1995

entity bcd27seg is
        port(bcd: in vlbit_1d(3 downto 0);  segment: out vlbit_1d(6 downto 0));
end bcd27seg;

architecture display of bcd27seg is
begin
        process (bcd)
        begin
                if bcd = B"0000" then
                        segment <= B"1000000";
                elsif bcd = B"0001" then
                        segment <= B"1111001";
                elsif bcd = B"0010" then
                        segment <= B"0100100";
                elsif bcd = B"0011" then
                        segment <= B"0110000";
                elsif bcd = B"0100" then
                        segment <= B"0011001";
                elsif bcd = B"0101" then
                        segment <= B"0010010";
                elsif bcd = B"0110" then
                        segment <= B"0000010";
                elsif bcd = B"0111" then
                        segment <= B"1111000";
                elsif bcd = B"1000" then
                        segment <= B"0000000";
                elsif bcd = B"1001" then
                        segment <= B"0010000";
                else
                        segment <= B"XXXXXXX";
                end if;
        end process;
end display;
```

Figure A.6 VHDL code for BCD to Seven Segment Display Converter.

```
entity clkdrvs IS
        PORT (MCLK , reset:IN vlbit;clkd16,clkd1024:out vlbit);
end clkdrvs;


ARCHITECTURE structural OF clkdrvs IS

COMPONENT div16
        PORT (SIGNAL reset,Mclk:IN vlbit;SIGNAL clkd16:out vlbit);
END COMPONENT;


Component div1024
        PORT (SIGNAL reset,clk:IN vlbit;SIGNAL clkd1024:out vlbit);
END COMPONENT;

SIGNAL c16 :vlbit;

BEGIN

  aa: div16         Port MAp  (reset,Mclk, c16);
        clkd16 <= c16 ;
  bb: div1024  Port MAp  (reset,c16, clkd1024);


END structural;
```

Figure A.7 VHDL code for the Clock Generator Module.

```
--      mux16.vhd
--      Complete Thirty-two input Multiplexer using
--      Process and sequential IF statements

ENTITY MUX16 IS
        Port(I:IN vlbit_1d (15 downto 0); Sel:IN vlbit_1d (4 downto 0); F:OUT vlbit);
end MUX16;

Architecture behavior of MUX16 IS

BEGIN

Scans: Process(sel,I)
        Begin
                IF (SEL=B"00000") THEN
                        F<=I(0);
                ELSIF (SEL=B"00001") THEN
                        F<=I(1);
                ELSIF (SEL=B"00010") THEN
                        F<=I(2);
                ELSIF (SEL=B"00011") THEN
                        F<=I(3);
                ELSIF (SEL=B"00100") THEN
                        F<=I(4);
                ELSIF (SEL=B"00101") THEN
                        F<=I(5);
                ELSIF (SEL=B"00110") THEN
                        F<=I(6);
                ELSIF (SEL=B"00111") THEN
                        F<=I(7);
                ELSIF (SEL=B"01000") THEN
                        F<=I(8);
                ELSIF (SEL=B"01001") THEN
                        F<=I(9);
                ELSIF (SEL=B"01010") THEN
                        F<=I(10);
                ELSIF (SEL=B"01011") THEN
                        F<=I(11);
                ELSIF (SEL=B"01100") THEN
                        F<=I(12);
                ELSIF (SEL=B"01101") THEN
                        F<=I(13);
                ELSIF (SEL=B"01110") THEN
                        F<=I(14);
                ELSIF (SEL=B"01111") THEN
                        F<=I(15);
                ELSE
                        F<='0';
                END IF;
   END PROCESS SCANS;
END behavior;
```

Figure A.8 VHDL Code for a 16 to 1 Multiplexer.

```
--      mux32.vhd
--      Complete Thirty-two input Multiplexer using
--      Process and sequential IF statements

ENTITY MUX32 IS
        Port(I:IN vlbit_1d (31 downto 0); Sel:IN vlbit_1d (4 downto 0); F:OUT vlbit);
end MUX32;

Architecture behavior of MUX32 IS

BEGIN


Scans: Process
        Begin
        IF (SEL=B"00000") THEN
                F<=I(0);
        ELSIF (SEL=B"00001") THEN
                F<=I(1);
        ELSIF (SEL=B"00010") THEN
                F<=I(2);
        ELSIF (SEL=B"00011") THEN
                F<=I(3);
        ELSIF (SEL=B"00100") THEN
                F<=I(4);
        ELSIF (SEL=B"00101") THEN
                F<=I(5);
        ELSIF (SEL=B"00110") THEN
                F<=I(6);
        ELSIF (SEL=B"00111") THEN
                F<=I(7);
        ELSIF (SEL=B"01000") THEN
                F<=I(8);
        ELSIF (SEL=B"01001") THEN
                F<=I(9);
        ELSIF (SEL=B"01010") THEN
                F<=I(10);
        ELSIF (SEL=B"01011") THEN
                F<=I(11);
        ELSIF (SEL=B"01100") THEN
                F<=I(12);
        ELSIF (SEL=B"01101") THEN
                F<=I(13);
        ELSIF (SEL=B"01110") THEN
                F<=I(14);
        ELSIF (SEL=B"01111") THEN
                F<=I(15);
        ELSIF (SEL=B"10000") THEN
                F<=I(16);
        ELSIF (SEL=B"10001") THEN
                F<=I(17);
        ELSIF (SEL=B"10010") THEN
                F<=I(18);
```

```
        ELSIF (SEL=B"10011") THEN
                F<=I(19);
        ELSIF (SEL=B"10100") THEN
                F<=I(20);
        ELSIF (SEL=B"10101") THEN
                F<=I(21);
        ELSIF (SEL=B"10110") THEN
                F<=I(22);
        ELSIF (SEL<=B"10111") THEN
                F<=I(23);
        ELSIF (SEL<=B"11000") THEN
                F<=I(24);
        ELSIF (SEL<=B"11001") THEN
                F<=I(25);
        ELSIF (SEL<=B"11010") THEN
                F<=I(26);
        ELSIF (SEL<=B"11011") THEN
                F<=I(27);
        ELSIF (SEL<=B"11100") THEN
                F<=I(28);
        ELSIF (SEL<=B"11101")  THEN
                F<=I(29);
        ELSIF (SEL<=B"11110") THEN
                F<=I(30);
        ELSIF (SEL<=B"11111") THEN
                F<=I(31);
        ELSE
                F<='X';
        END IF;

    END PROCESS SCANS;
END behavior;
```

Figure A.9 VHDL code for 32 to 1 Multiplexer.

```
-- This Timers3.vhd. It is the 10-bit time counter for each space
-- It uses the cnt_in.vhd design but PLC is set to maximum count.

library synth;
use synth.stdsynth.all;

entity timers3 IS
        port (reset,up_enable,Clk:In vlbit; c_out,c_out2 :out vlbit;
        final_count:out vlbit_1d(9 downto 0));
end timers3;
--      cnt_in.vhd
--      Complete 5-bit counter with Programmable PLC as Counter limit
--      Concurrency; with reset.
--      Used in space_cnt.vhd
--      rev. Nov. 7, 1995

Architecture two_cntin of timers3 Is

component cnt_in
PORT (SIGNAL RESET, UP_ENABLE,CLK:IN vlbit;
        PLC :IN vlbit_1d (4 downto 0);
        signal c_out                :out vlbit;
        signal bcd          :out vlbit_1d(4 downto 0));
end component;

signal bcd1,bcd2,plcf:vlbit_1d (4  downto 0);
signal cout,cout2,ncout:vlbit;

begin
        plcf <= ('1','1','1','1','1');
cnt1:   cnt_in Port Map (Reset, Up_enable, CLk,plcf,cout,bcd1);
        ncout <= Not cout;
 cnt2:  cnt_in Port map (Reset, Up_enable, ncout,plcf,cout2,bcd2);

final_count <= (bcd2(4),bcd2(3),bcd2(2),bcd2(1),bcd2(0),bcd1(4),bcd1(3), bcd1(2),bcd1(1),bcd1(0));
        c_out <= cout;
        c_out2 <= cout2;
end two_cntin;
```

Figure A.10 VHDL code for the 10-bit Timer module.

```
library synth;
use synth.stdsynth.all;


entity display_driver IS
        Port  (number:in vlbit_1d(4 downto 0); Disp1,Disp0:Out vlbit_1d(6 downto 0));
end display_driver;


Architecture structure of display_driver Is

component  bin2bcda
        Port  (number:in vlbit_1d(4 downto 0); bcd1,bcd2:Out vlbit_1d(3 downto 0));
end component;

component  bcd27seg
        port(bcd: in vlbit_1d(3 downto 0);   segment: out vlbit_1d(6 downto 0));
end component;

signal set1,set2 :vlbit_1d (3 downto 0);

begin

 o1: bin2bcda POrt MAp (number,set1,set2);
 o2: bcd27seg POrt MAp (set1,Disp0);
 o3: bcd27seg POrt MAp (set2,Disp1);

end structure;
```

Figure A.11 VHDL code for Display Driver module.

```
-- New display driver with latched number
-- 6/26/96
library synth;
use synth.stdsynth.all;


entity ndispld IS
        POrt ( number:in vlbit_1d(4 downto 0); full,reset,clk:in vlbit; num0:out vlbit_1d(4 downto 0);
Disp1,Disp0:Out vlbit_1d(6 downto 0));

end ndispld;


Architecture structure of ndispld Is

component display_driver
        POrt (number:in vlbit_1d(4 downto 0);
        Disp1,Disp0:Out vlbit_1d(6 downto 0));
end component;

signal numltch:vlbit_1d(4 downto 0);
signal D1, D0:vlbit_1d(6 DOWNTO 0);

begin
--
    dffc_v(number,reset,clk, numltch);

od2: Display_driver POrt MAp (numltch, D1,D0);
        num0 <=numltch;

        Disp1 <= B"1111111" WHEN FULL = '1' ELSE D1;
        Disp0 <= B"1111111" WHEN FULL = '1' ELSE D0;


end structure;
```

FIGURE A.12 VHDL code for the Display driver.

```
-- Create reset device for timers3
-- june 16 1996
-- signal_time.vhd
library synth;
use synth.stdsynth.all;

entity signal_time IS
        Port (Reset,Up_enable,sens_sig,CLK:IN vlbit;
        Synchr:IN vlbit_1d(3 downto 0) ;
        Times:out vlbit_1d(9 downto 0));
end signal_time;

Architecture structural of signal_time Is

component timers3
        port (reset,up_enable,Clk:In vlbit; c_out,c_out2 :out vlbit;
        final_count:out vlbit_1d(9 downto 0));
end component;

component Nor5
   PORT (s3,s2,s1,s0,b3 :IN vlbit; norout :OUT vlbit);
END component;

Signal a,b,rst,c1,c2,q,x:vlbit;

Begin

nclk:   nor5 Port MAP(Synchr(0),Synchr(0),Synchr(1),Synchr(2),Synchr(3),b);
        a<= Not sens_sig;
        rst <= q OR RESET;
        dffc(a,reset,b,q);

        x<= Sens_sig and CLK;
time_it: timers3 Port MAP(rst,up_enable,x,c1,c2,times);

end structural;
```

Figure A.13 VHDL code for the Signal_Time module.

```
library synth;
use synth.stdsynth.all;

ENTITY main_moda IS
  PORT (Muxin:IN vlbit_1d(15 downto 0);up_enable, Mclk, reset : in vlbit;
        MMC:Out vlbit_1d(4 downto 0); F:OUT vlbit);
end main_moda;

Architecture structural of main_moda Is

component mux16
        Port(I:IN vlbit_1d (15 downto 0); Sel:IN vlbit_1d (4 downto 0);
    F:OUT vlbit);
end component;

component cnt_in
        PORT (SIGNAL RESET, UP_ENABLE,CLK:IN vlbit;
        PLC :IN vlbit_1d (4 downto 0);
        signal c_out                :out vlbit;
        signal bcd          :out vlbit_1d(4 downto 0));
end component;


signal c_out:vlbit;
signal PLC,count:vlbit_1d(4 downto 0);


begin
  plc <= ('0','1','1','1','1');

   b: cnt_in Port MAp (reset,up_enable,Mclk,PLC,c_out,count);
        MMC <= count;
  a: mux16 Port MAp (Muxin,count,F);

end structural;
```

Figure A.14 VHDL code for the main module.

```
--      mux32_scan.vhd
--      Multiplexer Counter combination to Produce Scanning effect of
--      all 32 inputs. mux32-if statements; cnt_in-
--      rev. Oct. 25, 1995
ENTITY mux32_scan IS
        Port(plc:IN vlbit_1d(4 downto 0); I:IN vlbit_1d(31 downto 0); up_enable, clk_in, reset : in vlbit;
             cout, F:OUT vlbit);
END mux32_scan;

Architecture behavior of mux32_scan IS

COMPONENT
        mux32 Port(I:IN vlbit_1d (31 downto 0); Sel:IN vlbit_1d (4 downto 0);F:OUT vlbit);
END COMPONENT;

COMPONENT
         cnt_in port(reset, up_enable,clk:in vlbit;PLC:in vlbit_1d(4 downto 0); c_out: out vlbit;
                     bcd:out  vlbit_1d(4 downto 0));
END COMPONENT;

SIGNAL SEL:vlbit_1d(4 downto 0);

BEGIN
        cnt: cnt_in PORT MAP(reset,up_enable,clk_in,PLC,cout,sel);

        m1: mux32 PORT MAP(I,sel,F);
END Behavior;
```

Figure A.15 VHDL code for Multiplexer -counter combination .

```
-- 6/26/96
library synth;
use synth.stdsynth.all;


entity muxcnt2 IS
        POrt  (sens:in vlbit_1d(31 downto 0) ; PLC:in vlbit_1d(4 downto 0);
up_enable,reset,clk:in vlbit; Full, f :out vlbit;num0:out vlbit_1d(4 downto 0));

end muxcnt2;


Architecture structure of muxcnt2 Is

COMPONENT  mux32 Port(I:IN vlbit_1d (31 downto 0);
Sel:IN vlbit_1d (4 downto 0);F:OUT vlbit);
END COMPONENT;


component cnt_in
        PORT (SIGNAL RESET, UP_ENABLE,CLK:IN vlbit;
        PLC :IN vlbit_1d (4 downto 0);
        signal c_out                :out vlbit;
        signal bcd          :out vlbit_1d(4 downto 0));
end component;


signal  fl,a,x,s,y,ck,s2,q0,d3,q1,xx,nrst:vlbit;
signal  number:vlbit_1d(4 downto 0);

begin

FULL<= Sens(0) And Sens(1) and Sens(2) and sens(3) and Sens(4) And
Sens(5) and  Sens(6) and sens(7) and  Sens(8) And Sens(9) and Sens(10) and
sens(11) and Sens(12) And Sens(13) and Sens(14) and sens(15) and Sens(0) And
Sens(1) and  Sens(18) and sens(19) and Sens(20) And Sens(21) and Sens(22) and
sens(23) and Sens(24) And Sens(25) and Sens(26) and sens(27) and Sens(28) And
Sens(29) and Sens(30) and sens(31);


        a <= Not s;                     -- create input for mux latch
        x <=  CK and (reset OR nrst);       -- two ways to reset
        d3 <= not q0;
        dffc(d3,reset,clk,q0);  -- divied clock by 2
--      xx <= d3 and clk;       -- create pulse for latching mux output
        dffc(a,x,xx,q1);   -- latch to hold mux output.
        ck <= (not q0) and (not clk) ;                  -- negative edge to clock for reset counter
    xx <= (not clk) and q0;
od0:  cnt_in Port MAp (x,up_enable,q0,PLC,y,number);

od1:  mux32 Port MAp (sens,number,s);

    dffc(a,reset,ck,nrst); -- reset with delay eliminate feedback;
```

```
        f <=q1;
    num0 <= number;
end structure;
```

Figure A.16 VHDL code for the Muxcnt2 used to display space number.

```
library synth;
use synth.stdsynth.all;

ENTITY space_cntr IS
  PORT (I:IN vlbit_1d(31 downto 0);up_enable, clk, reset : in vlbit;
        PLC:In vlbit_1d(4 downto 0); F:OUT vlbit);
end space_cntr;

Architecture structural of space_cntr Is

component mux_scann
        Port(I:IN vlbit_1d(31 downto 0);up_enable, clk_in,reset :in vlbit;
  PLC:In vlbit_1d(4 downto 0);cout, F:OUT vlbit);
end component;

signal c_out:vlbit;

begin

 a: mux_scann Port MAp (I,reset,up_enable,clk,plc,c_out,F);

end structural;
```

Figure A.17 VHDL code for the Space_cntr Module.

```
-- Complete combination of system
--  july 7 1996
-- compsya.vhd renamed and changed to latch f_ser output
library synth;
use synth.stdsynth.all;

entity compsyb IS

        Port (Reset,Up_enable,MCLK:IN vlbit;PLC :IN vlbit_1d(4 downto 0);
Sens:IN vlbit_1d(31 downto 0); Disp1,Disp0:Out vlbit_1d(6 downto 0);
 tess,just :Out vlbit_1d(4 downto 0);Tim0:OUT vlbit_1d(9 downto 0);str0:out
vlbit_1d(31 downto 0);clk16,clk12,FULL,f_ser:out vlbit);

end compsyb;

Architecture structural of compsyb Is

-----------------------------------------
component main_moda
 PORT (Muxin:IN vlbit_1d(15 downto 0);up_enable, Mclk, reset : in vlbit;
 MMC:Out vlbit_1d(4 downto 0); F:OUT vlbit);
end component;

-- clock dividers!!!----------------------------
component clkdrvs
        PORT (MCLK , reset:IN vlbit;clkd16,clkd1024:out vlbit);
end component;



-------------------------------
--component timers3
--        PORT (SIGNAL RESET, UP_ENABLE,CLK:IN vlbit;
--        signal c_out                    :out vlbit;
--        signal bcd          :out vlbit_1d(4 downto 0));
--end component;
-- don't need here 6/20/96

-------------------------

component signal_time

        Port (Reset,Up_enable,sens_sig,CLK:IN vlbit;
     Synchr:IN vlbit_1d(3 downto 0) ;
    Times:out vlbit_1d(9 downto 0));

end component;

-----------------------------------
component space_cntr
 PORT (I:IN vlbit_1d(31 downto 0);up_enable,clk,reset :in vlbit;
    PLC:In vlbit_1d(4 downto 0); F:OUT vlbit);
end component;
```

```
---------------------------------
component cnt_in
        PORT (SIGNAL RESET, UP_ENABLE,CLK:IN vlbit;
        PLC :IN vlbit_1d (4 downto 0);
        signal c_out        :out vlbit;
        signal bcd          :out vlbit_1d(4 downto 0));
end component;

---------------------------------
component mux_scann
        Port(I:IN vlbit_1d(31 downto 0);
up_enable, clk_in, reset : in vlbit;
PLC:In vlbit_1d(4 downto 0);cout, F:OUT vlbit);

end component;

-----------------------------------
--component displaymod2
--      POrt  (sens:in vlbit_1d(31 downto 0) ;
--PLC:in vlbit_1d(4 downto 0);
--up_enable,reset,clk:in vlbit;
--Disp1,Disp0:Out vlbit_1d(6 downto 0));
--
--end component;

component  ndispld
        POrt  ( number:in vlbit_1d(4 downto 0); full,reset,clk:in vlbit;
        num0:out vlbit_1d(4 downto 0); Disp1,Disp0:Out vlbit_1d(6 downto 0));

end component;

component muxcnt2
        POrt  (sens:in vlbit_1d(31 downto 0) ; PLC:in vlbit_1d(4 downto 0);
 up_enable,reset,clk:in vlbit; Full, f :out vlbit;
 num0:out vlbit_1d(4 downto 0));

end component;


---------------------------------------
Signal clkd16,clkd1024,d0,d1,d2,d3,d4,d5,d6,d7,d8,d9 : vlbit;
Signal t,fll,f,f0,f1,nmclk:vlbit;

Signal dum:vlbit_1d(12 downto 0);
Signal plc2 ,bcd,synchr,number,numltch:vlbit_1d(4 downto 0);
Signal synco :vlbit_1d(3 downto 0);
Signal stream0,stream1,stream2,stream3,stream4,stream5,
stream6,stream7,stream8,stream9: vlbit_1d(31 downto 0);

Signal I0,I1,I2,I3,I4,I5,I6,I7,I8,I9,I10,I11,I12,I13,I14,I15 : vlbit;
Signal I: vlbit_1d(15 downto 0);
```

```
Signal Times31, Times30, Times29, Times28, Times27: vlbit_1d(9 downto 0);
Signal Times26, Times25, Times24, Times23, Times22: vlbit_1d(9 downto 0);
Signal Times21, Times20,Times19, Times18, Times17,Times16: vlbit_1d(9 downto 0);
Signal Times15, Times14,Times13, Times12, Times11,Times10: vlbit_1d(9 downto 0);
Signal Times9, Times8, Times7, Times6, Times5, Times4: vlbit_1d(9 downto 0);
Signal Times3,Times2, Times1, Times0: vlbit_1d(9 downto 0);


signal clkdelay:vlbit;
---------------------------------------------------------------------------

begin

 clks : clkdrvs Port MAp (MCLK,reset,clkd16,clkd1024);
 clk16<=clkd16;

 clk12<=clkd1024;

  synco <= synchr(3 downto 0);

  Mod1: space_cntr PORT Map (sens,UP_ENABLE,CLKd16,RESET,PLC,I5);
-- Count through the spaces to give bcd to big multiplexer.
   tess <= bcd;
 clkdelay <= clkd16 ;
  Mod2: cnt_in  Port MAp(reset,up_enable,clkdelay,PLC,dum(10),bcd);
--  I(4 downto 0) <=bcd;
I0 <= bcd(0);
I1 <= bcd(1);
I2 <= bcd(2);
I3 <= bcd(3);
I4 <= bcd(4);
--
--Declare counters for timers for space signals
 s0: signal_time Port MAP (reset,up_enable,Sens(0),CLKd1024,synco,Times0);
 s1: signal_time Port MAP (reset,up_enable,Sens(1),CLKd1024,synco,Times1);
 s2: signal_time Port MAP (reset,up_enable,Sens(2),CLKd1024,synco,Times2);
 s3: signal_time Port MAP (reset,up_enable,Sens(3),CLKd1024,synco,Times3);
 s4: signal_time Port MAP (reset,up_enable,Sens(4),CLKd1024,synco,Times4);
 s5: signal_time Port MAP (reset,up_enable,Sens(5),CLKd1024,synco,Times5);
 s6: signal_time Port MAP (reset,up_enable,Sens(6),CLKd1024,synco,Times6);
 s7: signal_time Port MAP (reset,up_enable,Sens(7),CLKd1024,synco,Times7);
 s8: signal_time Port MAP (reset,up_enable,Sens(8),CLKd1024,synco,Times8);
 s9: signal_time Port MAP (reset,up_enable,Sens(9),CLKd1024,synco,Times9);
s10: signal_time Port MAP (reset,up_enable,Sens(10),CLKd1024,synco,Times10);
s11: signal_time Port MAP (reset,up_enable,Sens(11),CLKd1024,synco,Times11);
s12: signal_time Port MAP (reset,up_enable,Sens(12),CLKd1024,synco,Times12);
s13: signal_time Port MAP (reset,up_enable,Sens(13),CLKd1024,synco,Times13);
s14: signal_time Port MAP (reset,up_enable,Sens(14),CLKd1024,synco,Times14);
s15: signal_time Port MAP (reset,up_enable,Sens(15),CLKd1024,synco,Times15);
s16: signal_time Port MAP (reset,up_enable,Sens(16),CLKd1024,synco,Times16);
s17: signal_time Port MAP (reset,up_enable,Sens(17),CLKd1024,synco,Times17);
s18: signal_time Port MAP (reset,up_enable,Sens(18),CLKd1024,synco,Times18);
s19: signal_time Port MAP (reset,up_enable,Sens(19),CLKd1024,synco,Times19);
```

```
s20: signal_time Port MAP (reset,up_enable,Sens(20),CLKd1024,synco,Times20);
s21: signal_time Port MAP (reset,up_enable,Sens(21),CLKd1024,synco,Times21);
s22: signal_time Port MAP (reset,up_enable,Sens(22),CLKd1024,synco,Times22);
s23: signal_time Port MAP (reset,up_enable,Sens(23),CLKd1024,synco,Times23);
s24: signal_time Port MAP (reset,up_enable,Sens(24),CLKd1024,synco,Times24);
s25: signal_time Port MAP (reset,up_enable,Sens(25),CLKd1024,synco,Times25);
s26: signal_time Port MAP (reset,up_enable,Sens(26),CLKd1024,synco,Times26);
s27: signal_time Port MAP (reset,up_enable,Sens(27),CLKd1024,synco,Times27);
s28: signal_time Port MAP (reset,up_enable,Sens(28),CLKd1024,synco,Times28);
s29: signal_time Port MAP (reset,up_enable,Sens(29),CLKd1024,synco,Times29);
s30: signal_time Port MAP (reset,up_enable,Sens(30),CLKd1024,synco,Times30);
s31: signal_time Port MAP (reset,up_enable,Sens(31),CLKd1024,synco,Times31);

Tim0 <= TImes0;

stream0 <= (Times31(0),Times30(0),times29(0),times28(0),times27(0),
times26(0),times25(0),times24(0),times23(0),times22(0),times21(0),
times20(0),times19(0),times18(0),times17(0),times16(0),times15(0),
times14(0),times13(0),times12(0),times11(0),times10(0),times9(0),
times8(0),times7(0),times6(0),times5(0),times4(0),times3(0),
times2(0),times1(0),times0(0));

 Str0 <= Stream0;

stream1 <= (Times31(1),Times30(1),times29(1),times28(1),times27(1),
times26(1),times25(1),times24(1),times23(1),times22(1),times21(1),
times20(1),times19(1),times18(1),times17(1),times16(1),times15(1),
times14(1),times13(1),times12(1),times11(1),times10(1),times9(1),
times8(1),times7(1),times6(1),times5(1),times4(1),times3(1),
times2(1),times1(1),times0(1));

stream2 <= (Times31(2),Times30(2),times29(2),times28(2),times27(2),
times26(2),times25(2),times24(2),times23(2),times22(2),times21(2),
times20(2),times19(2),times18(2),times17(2),times16(2),times15(2),
times14(2),times13(2),times12(2),times11(2),times10(2),times9(2),
times8(2),times7(2),times6(2),times5(2),times4(2),times3(2),
times2(2),times1(2),times0(2));


stream3 <= (Times31(3),Times30(3),times29(3),times28(3),times27(3),
times26(3),times25(3),times24(3),times23(3),times22(3),times21(3),
times20(3),times19(3),times18(3),times17(3),times16(3),times15(3),
times14(3),times13(3),times12(3),times11(3),times10(3),times9(3),
times8(3),times7(3),times6(3),times5(3),times4(3),times3(3),
times2(3),times1(3),times0(3));


stream4 <= (Times31(4),Times30(4),times29(4),times28(4),times27(4),
times26(4),times25(4),times24(4),times23(4),times22(4),times21(4),
times20(4),times19(4),times18(4),times17(4),times16(4),times15(4),
times14(4),times13(4),times12(4),times11(4),times10(4),times9(4),
times8(4),times7(4),times6(4),times5(4),times4(4),times3(4),
times2(4),times1(4),times0(4));
```

```
stream5 <= (Times31(5),Times30(5),times29(5),times28(5),times27(5),
times26(5),times25(5),times24(5),times23(5),times22(5),times21(5),
times20(5),times19(5),times18(5),times17(5),times16(5),times15(5),
times14(5),times13(5),times12(5),times11(5),times10(5),times9(5),
times8(5),times7(5),times6(5),times5(5),times4(5),times3(5),
times2(5),times1(5),times0(5));


stream6 <= (Times31(6),Times30(6),times29(6),times28(6),times27(6),
times26(6),times25(6),times24(6),times23(6),times22(6),times21(6),
times20(6),times19(6),times18(6),times17(6),times16(6),times15(6),
times14(6),times13(6),times12(6),times11(6),times10(6),times9(6),
times8(6),times7(6),times6(6),times5(6),times4(6),times3(6),
times2(6),times1(6),times0(6));


stream7 <= (Times31(7),Times30(7),times29(7),times28(7),times27(7),
times26(7),times25(7),times24(7),times23(7),times22(7),times21(7),
times20(7),times19(7),times18(7),times17(7),times16(7),times15(7),
times14(7),times13(7),times12(7),times11(7),times10(7),times9(7),
times8(7),times7(7),times6(7),times5(7),times4(7),times3(7),
times2(7),times1(7),times0(7));

stream8 <= (Times31(8),Times30(8),times29(8),times28(8),times27(8),
times26(8),times25(8),times24(8),times23(8),times22(8),times21(8),
times20(8),times19(8),times18(8),times17(8),times16(8),times15(8),
times14(8),times13(8),times12(8),times11(8),times10(8),times9(8),
times8(8),times7(8),times6(8),times5(8),times4(8),times3(8),
times2(8),times1(8),times0(8));

stream9 <= (Times31(9),Times30(9),times29(9),times28(9),times27(9),
times26(9),times25(9),times24(9),times23(9),times22(9),times21(9),
times20(9),times19(9),times18(9),times17(9),times16(9),times15(9),
times14(9),times13(9),times12(9),times11(9),times10(9),times9(9),
times8(9),times7(9),times6(9),times5(9),times4(9),times3(9),
times2(9),times1(9),times0(9));



timux0:  mux_scann POrt Map (stream0,up_enable,clkd16,reset,PLC,dum(0),I6);
timux1:  mux_scann POrt Map (stream1,up_enable,clkd16,reset,PLC,dum(1),I7);
timux2:  mux_scann POrt Map (stream2,up_enable,clkd16,reset,PLC,dum(2),I8);
timux3:  mux_scann POrt Map (stream3,up_enable,clkd16,reset,PLC,dum(3),I9);
timux4:  mux_scann POrt Map (stream4,up_enable,clkd16,reset,PLC,dum(4),I10);
timux5:  mux_scann POrt Map (stream5,up_enable,clkd16,reset,PLC,dum(5),I11);
timux6:  mux_scann POrt Map (stream6,up_enable,clkd16,reset,PLC,dum(6),I12);
timux7:  mux_scann POrt Map (stream7,up_enable,clkd16,reset,PLC,dum(7),I13);
timux8:  mux_scann POrt Map (stream8,up_enable,clkd16,reset,PLC,dum(8),I14);
timux9:  mux_scann POrt Map (stream9,up_enable,clkd16,reset,PLC,dum(9),I15);
```

```
 I <= (I15,I14,I13,I12,I11,I10,I9,I8,I7,I6,I5,I4,I3,I2,I1,I0);


 U1 : main_moda Port Map (I,up_enable,Mclk,reset,synchr,f0);
-- Big Multiplexer -- .

  nmclk <= not Mclk;
        dffc(f0,reset,nmclk,f1);
        f_ser <=f1;

 U2: muxcnt2 Port Map (sens,plc,up_enable,reset,clkd16,fll,f,number);

 U3: ndispld port map (number,fll,reset,f,numltch,DISP1,DISP0);
        just <= numltch;

FULL <= fll;

end structural;
```

Figure A.18 VHDL code for the entire space management system.

APPENDIX B

SIMULATION RESULTS

OF THE SPACE MANAGEMENT SYSTEM

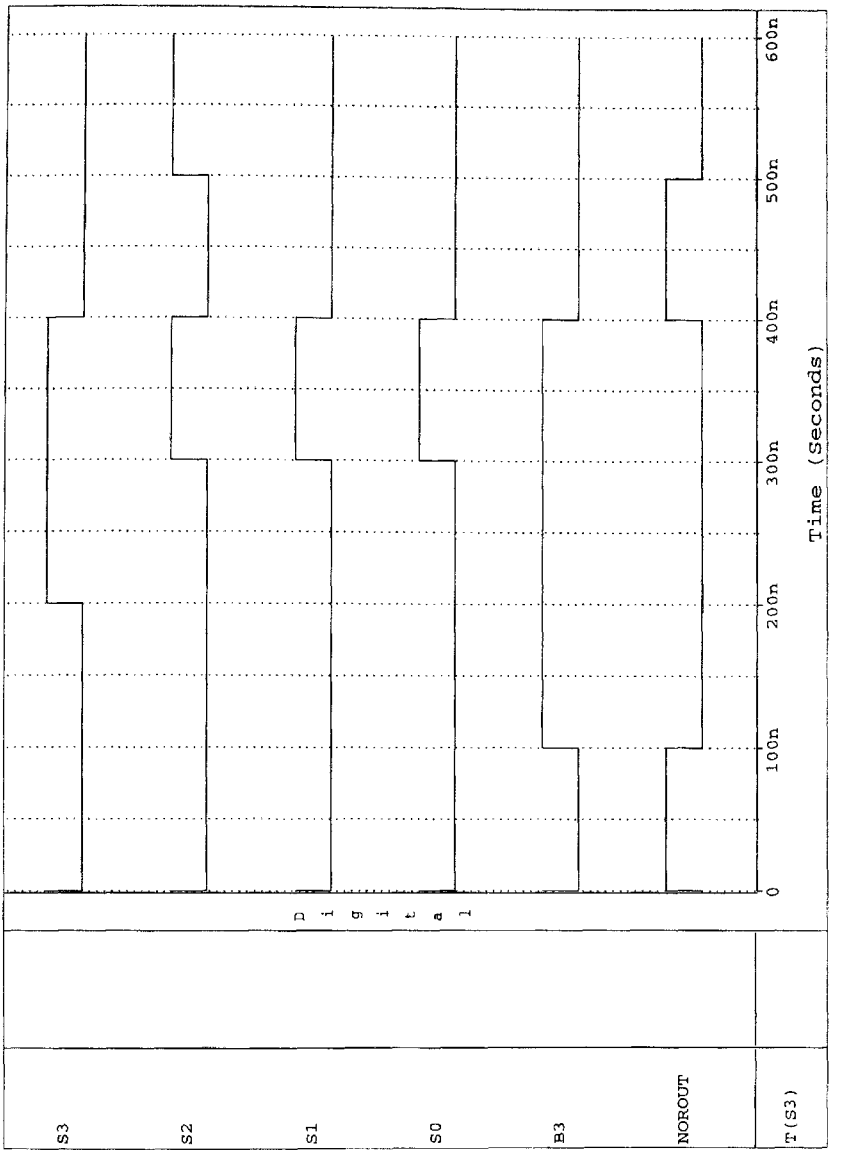

Figure B.1 Simulation Results of the 5-input NOR gate.

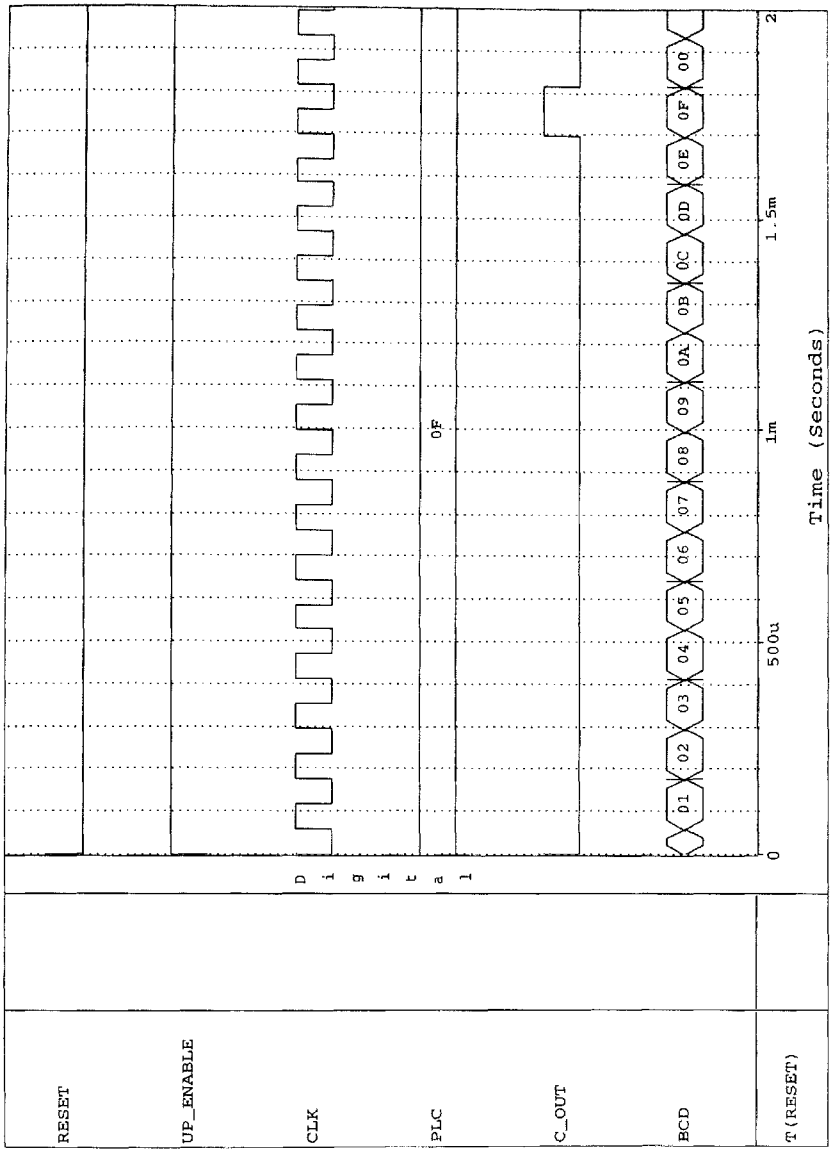

Figure B.2 Simulation Results of the 5-Bit Counter.

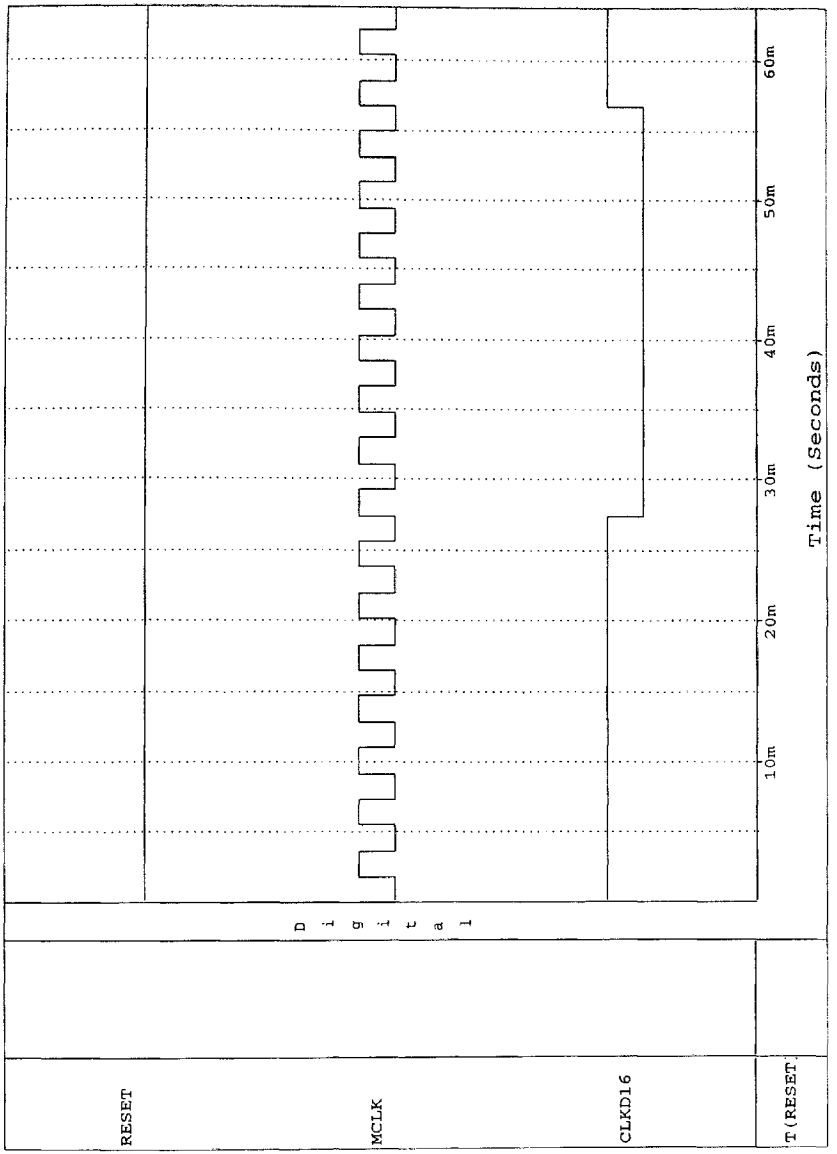

Figure B.3 Simulation of the Divide by 16 Counter.

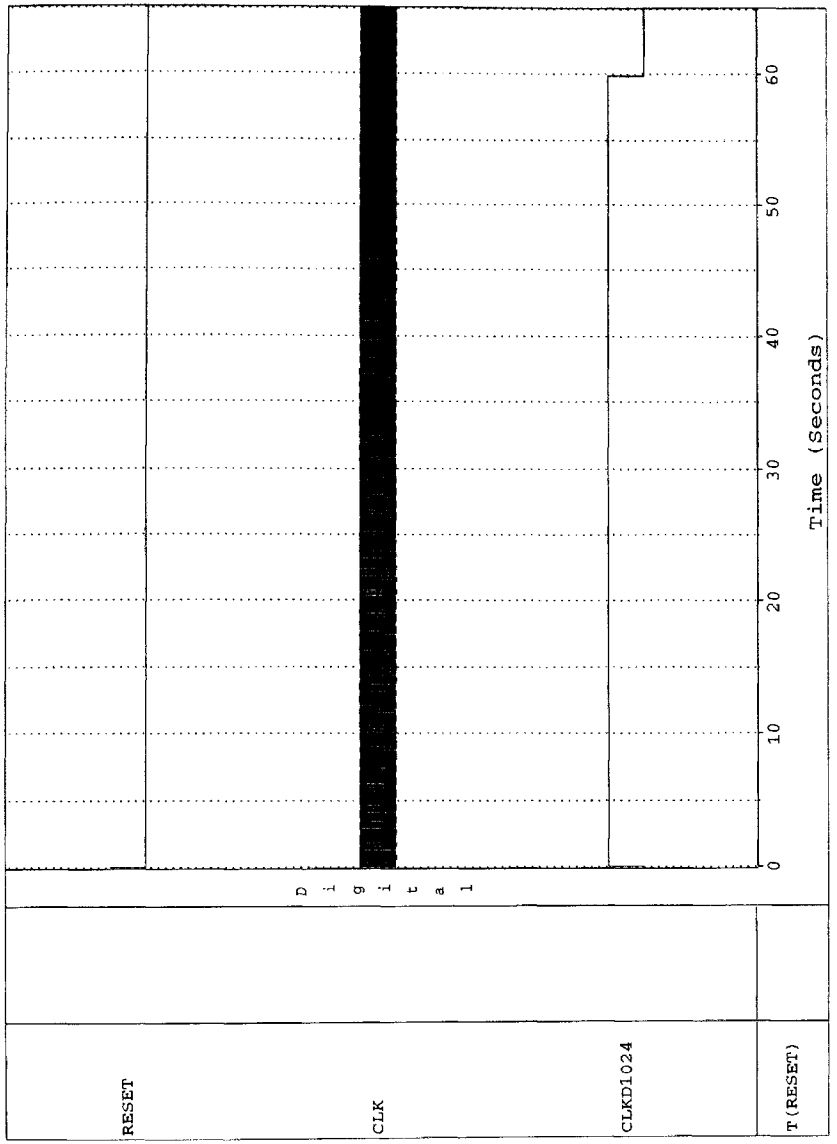

Figure B.4 Simulation Results of the Divide by 1024 Counter.

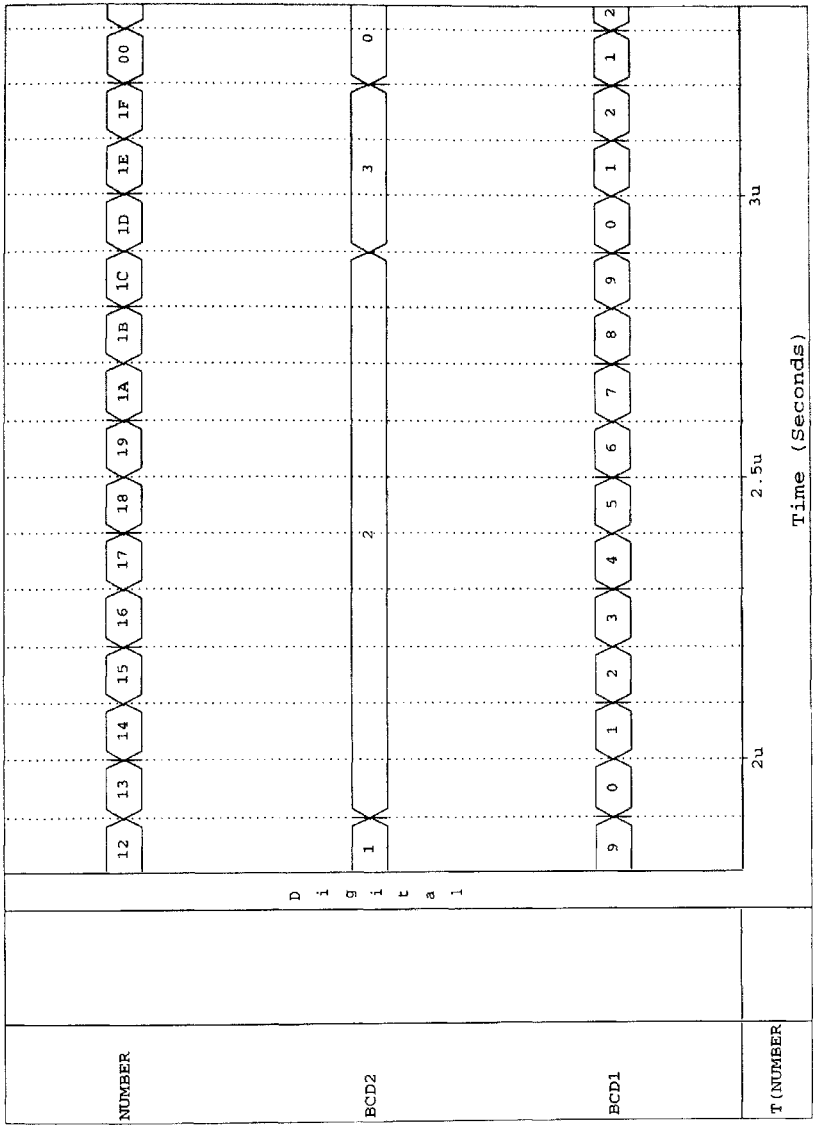

Figure B.5(a) Simulation Results of the Binary to BCD converter. (Zoomed view)

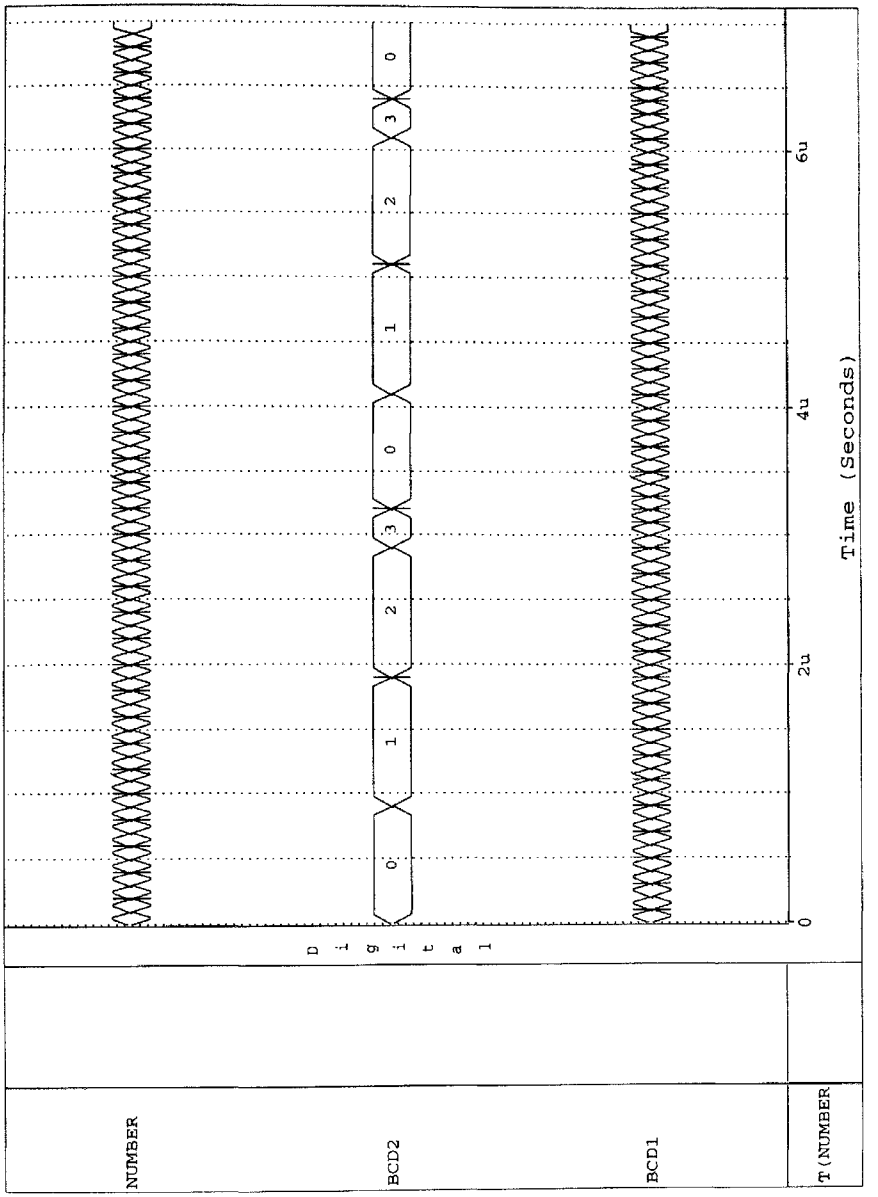

Figure B.5(b) Simulation Results of the Binary to BCD converter.

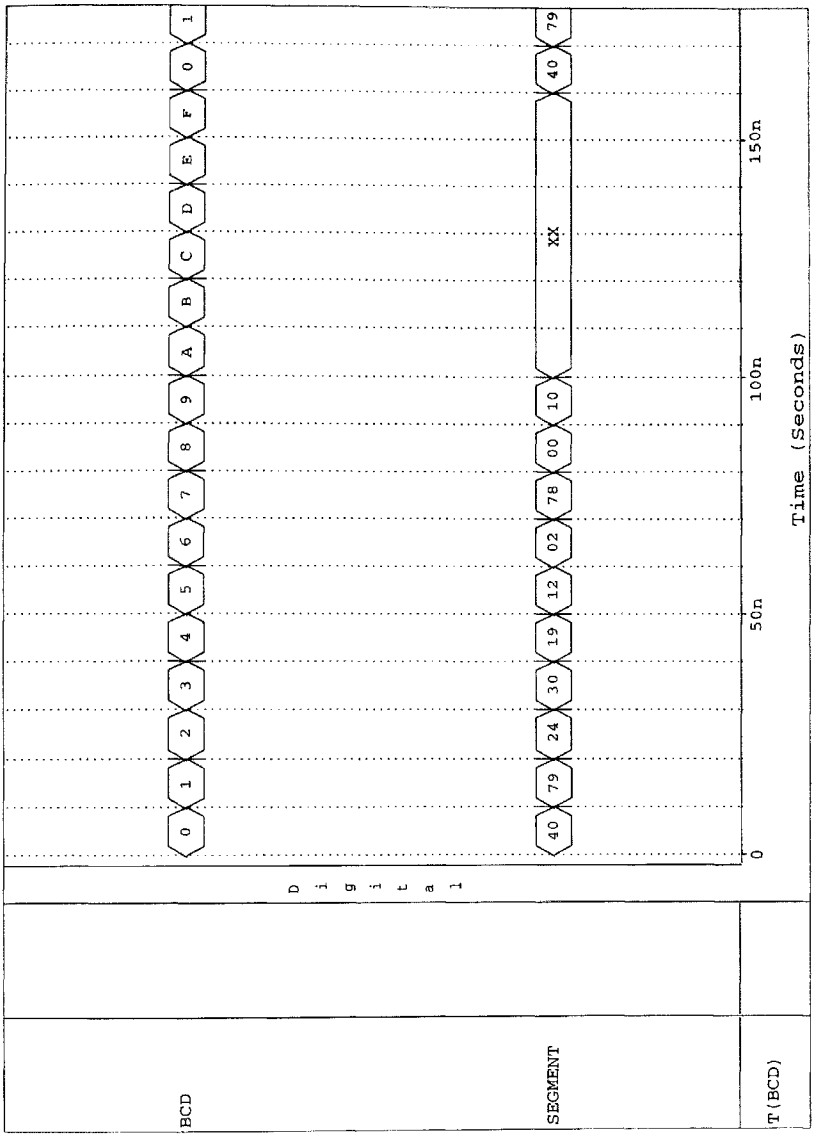

Figure B.6 Simulation Results of the BCD to Seven Segment Converter.

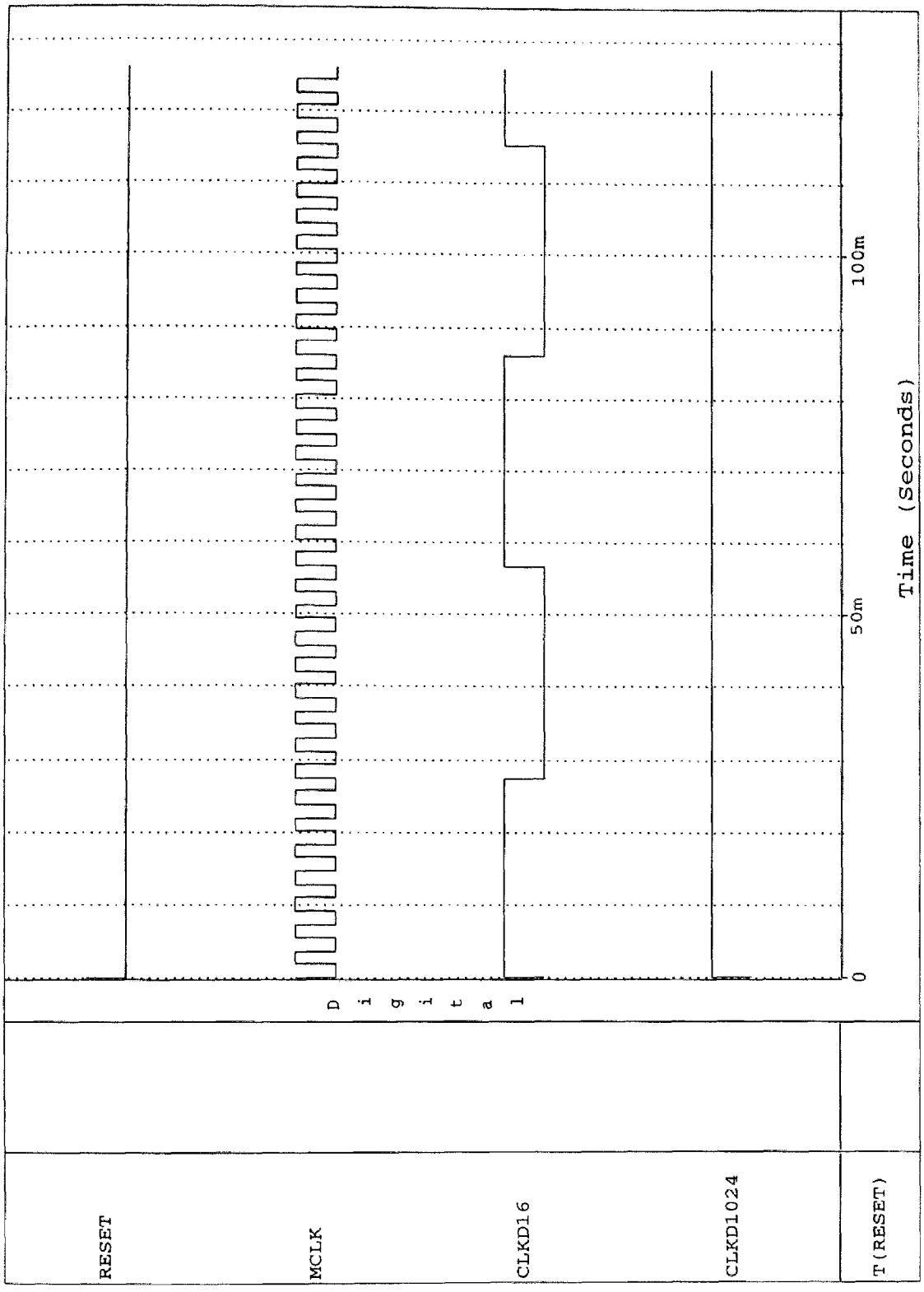

Figure B.7 Simulation Results of the Clock Generator.

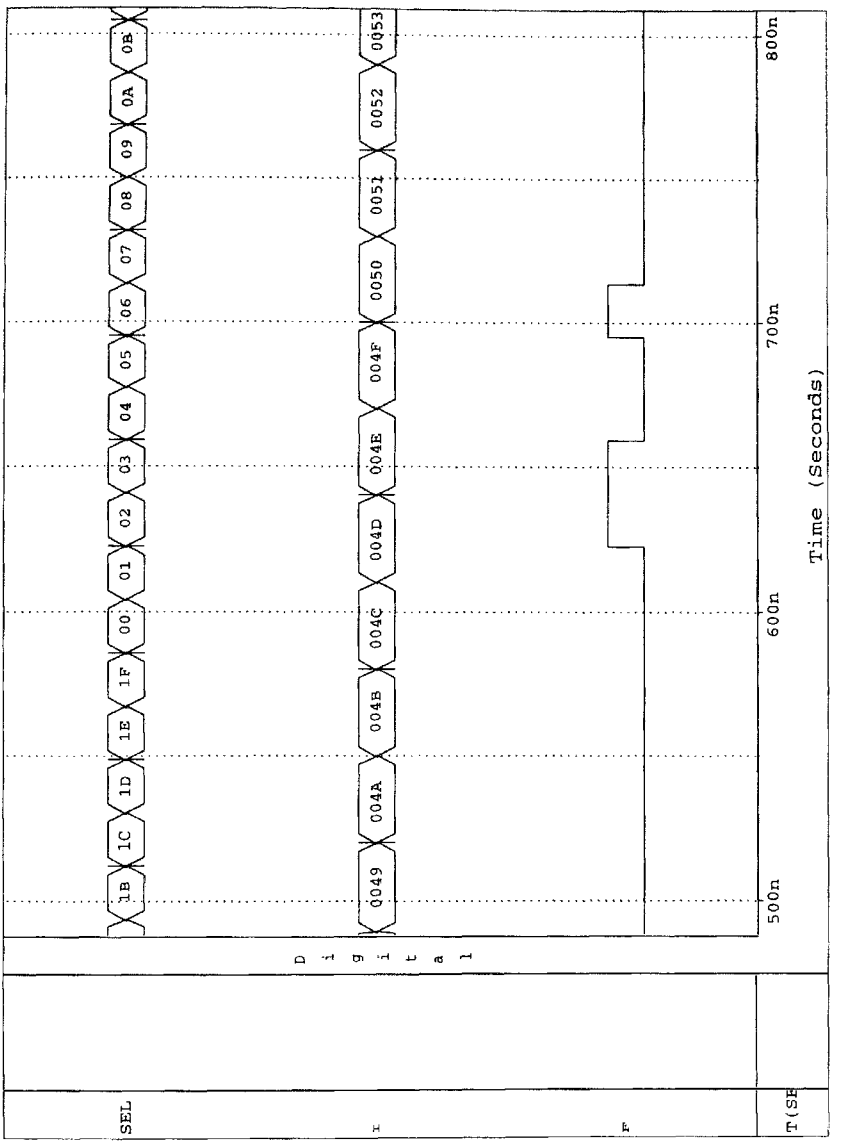

Figure B.8 Simulation Results of the 16-input Multiplexer.

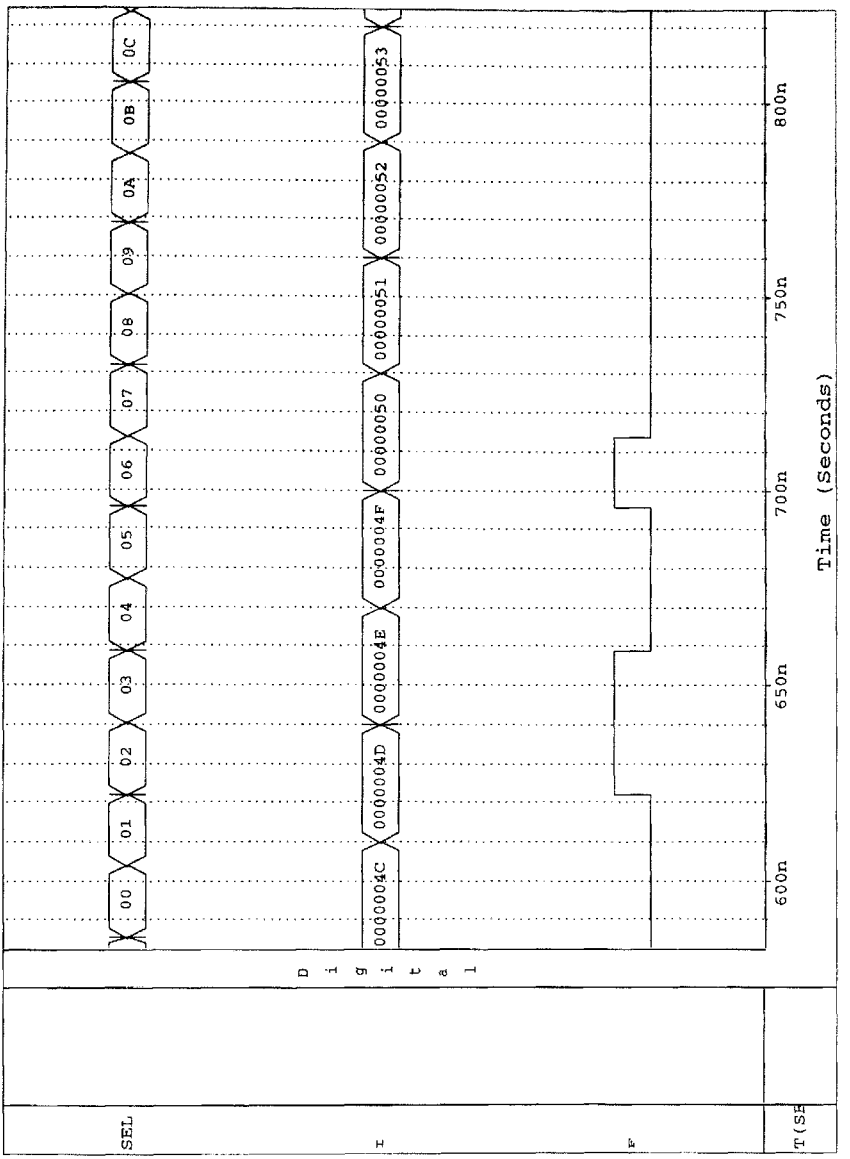

Figure B.9 Simulation Results of the 32-input Multiplexer.

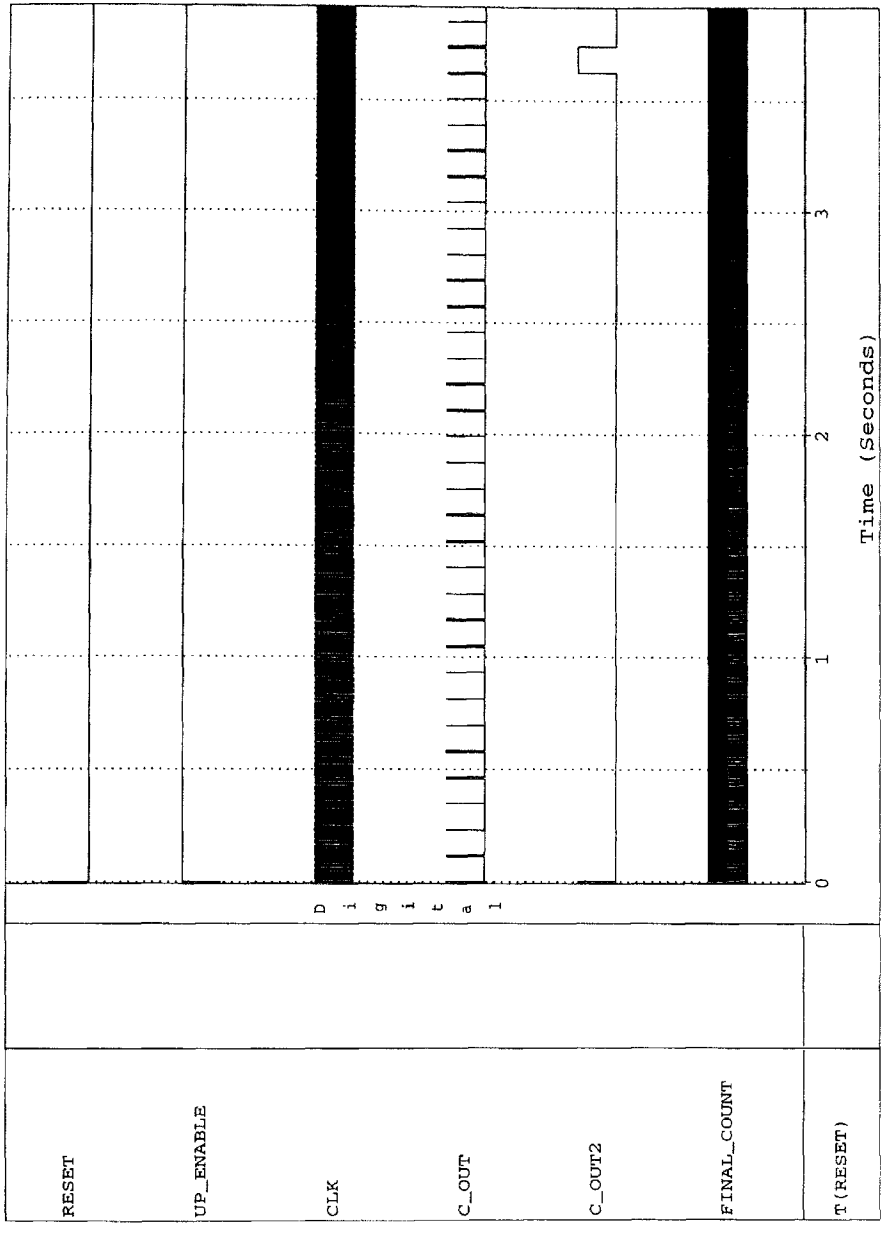

Figure B.10(a) Simulation Results of the 10-bit timer.

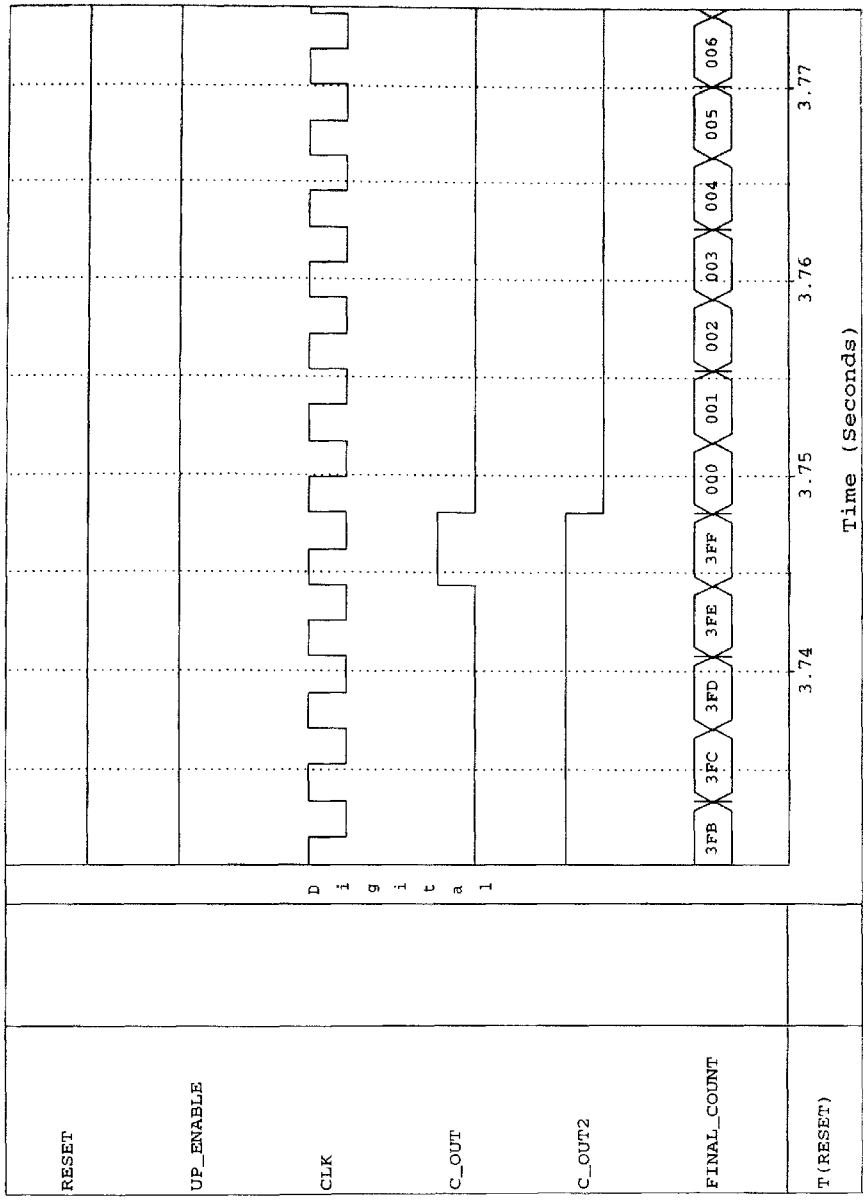

Figure B.10(b) Simulation Results of the 10-bit timer. (Zoomed View)

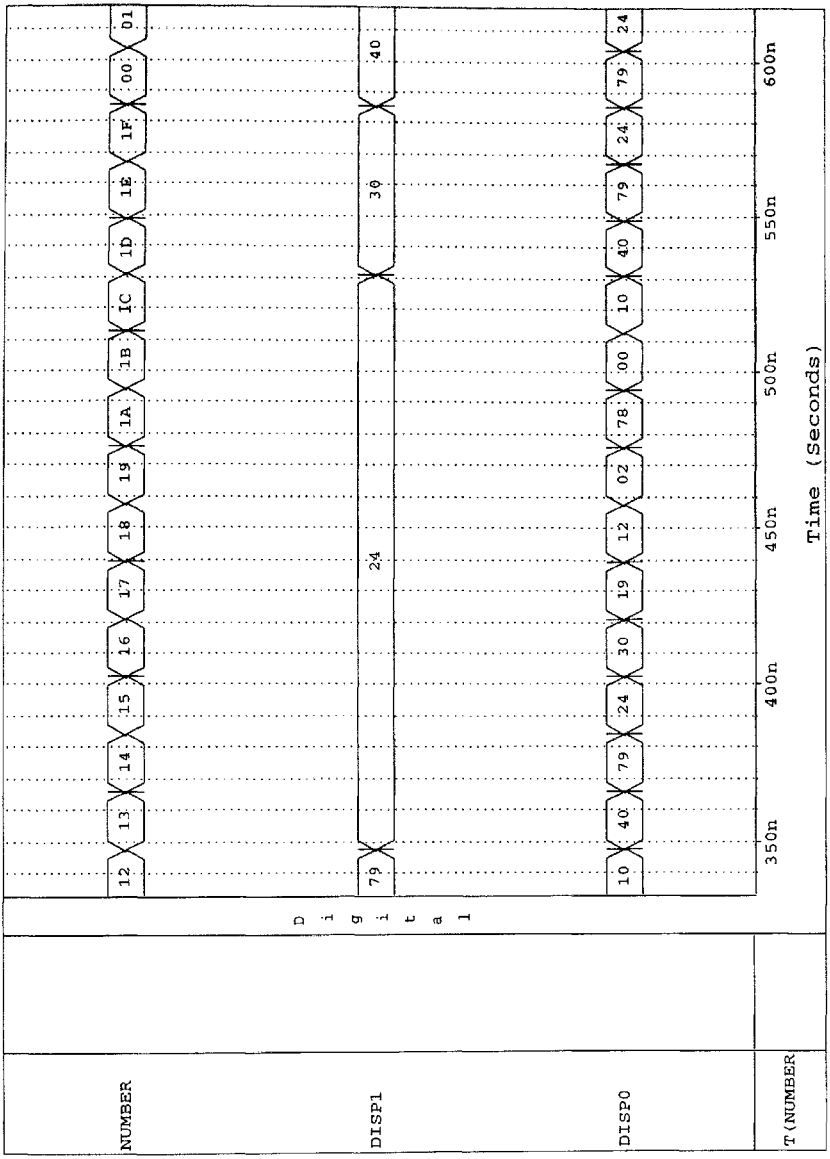

Figure B.11 Simulation Results of the Display Driver Module.

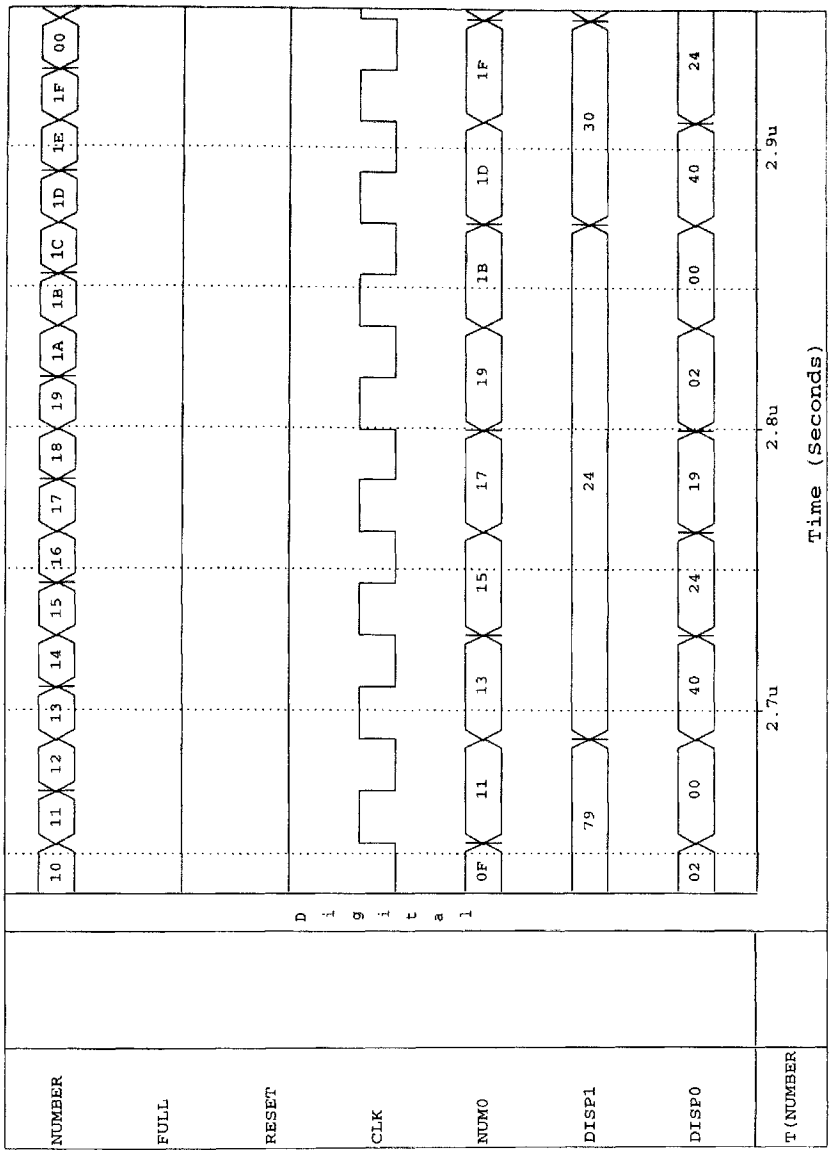

Figure B.12 Simulation results of the Display Module

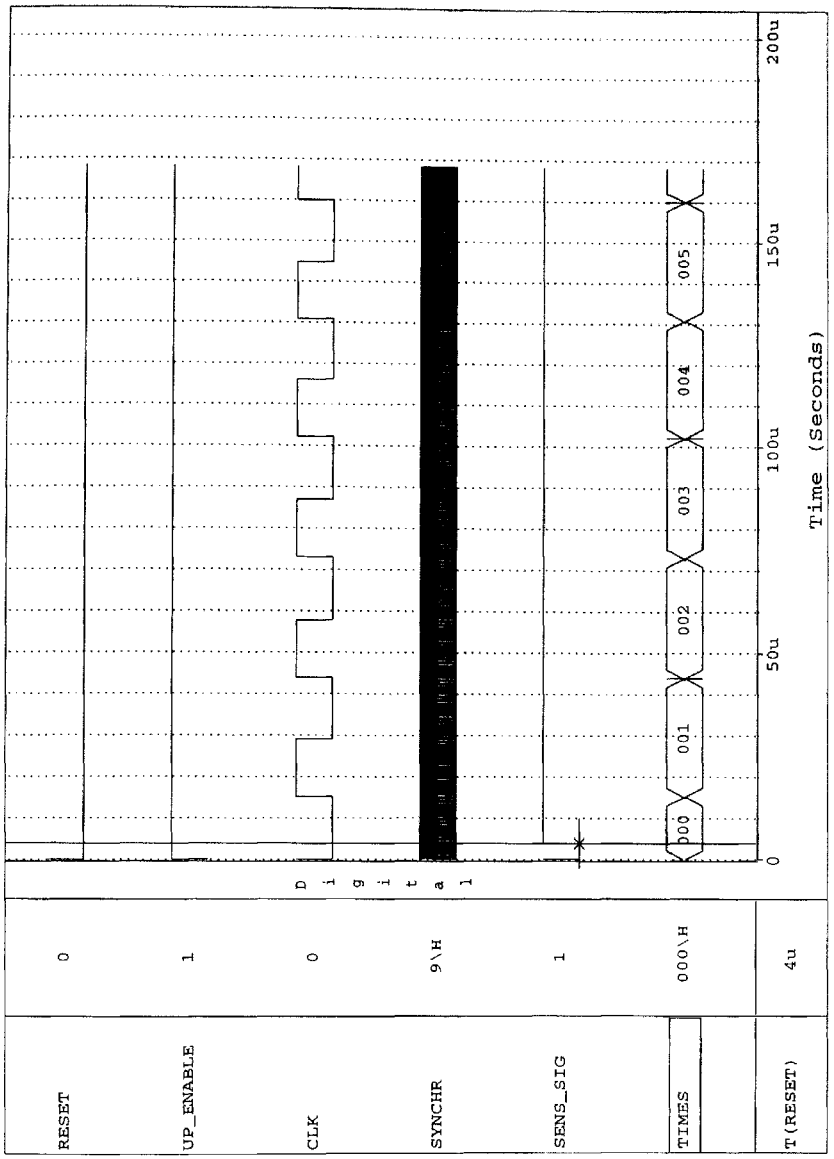

Figure B.13 Simulation results of the Signal_Time Module

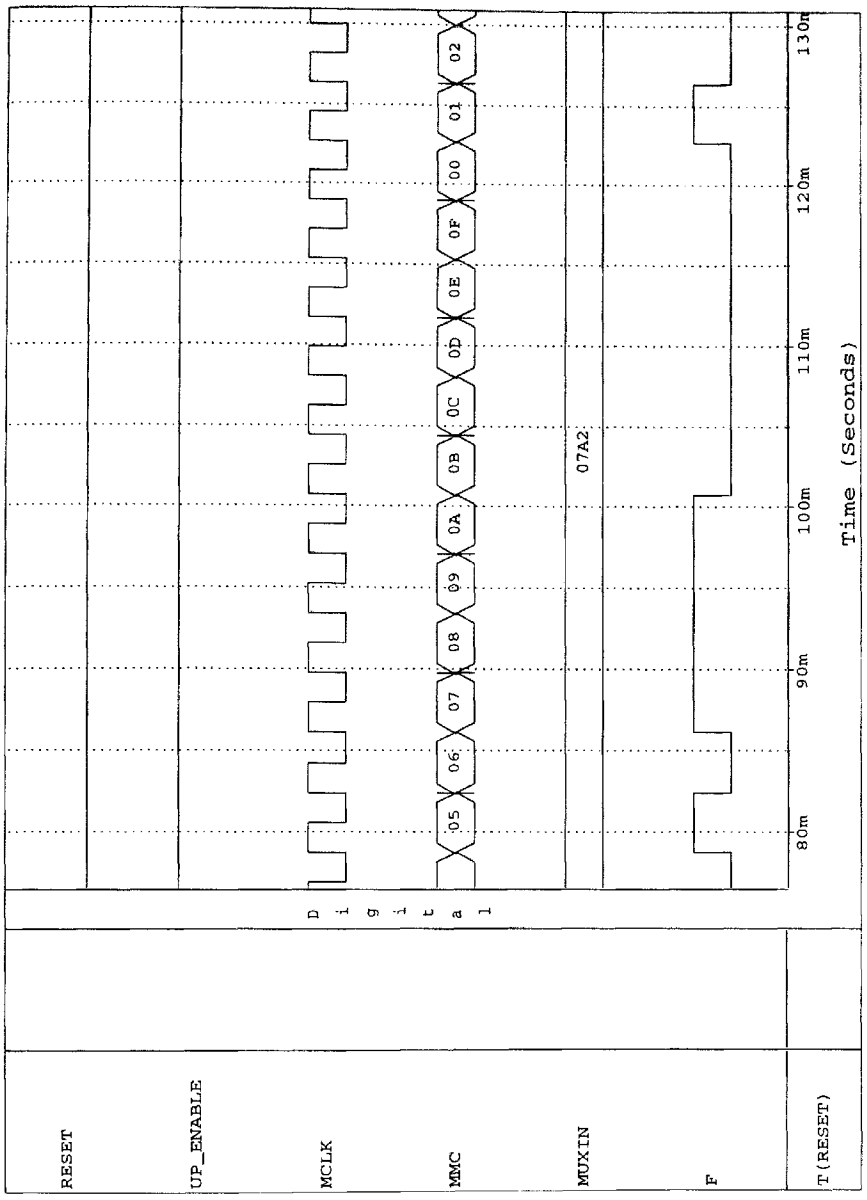

Figure B.14 Simulation results of the main module.

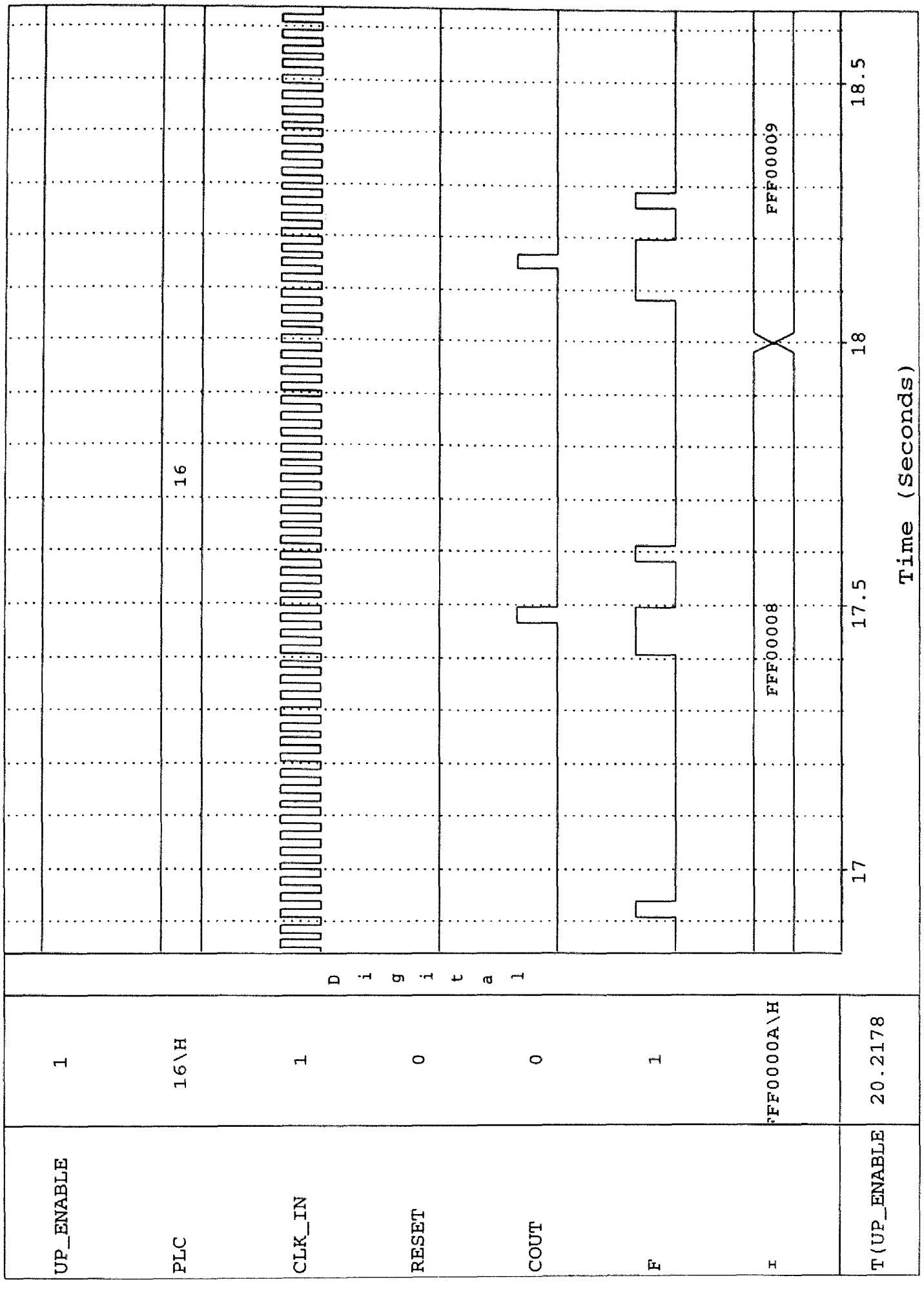

Figure B.15 Simulation Results of the Mux-Scann module.

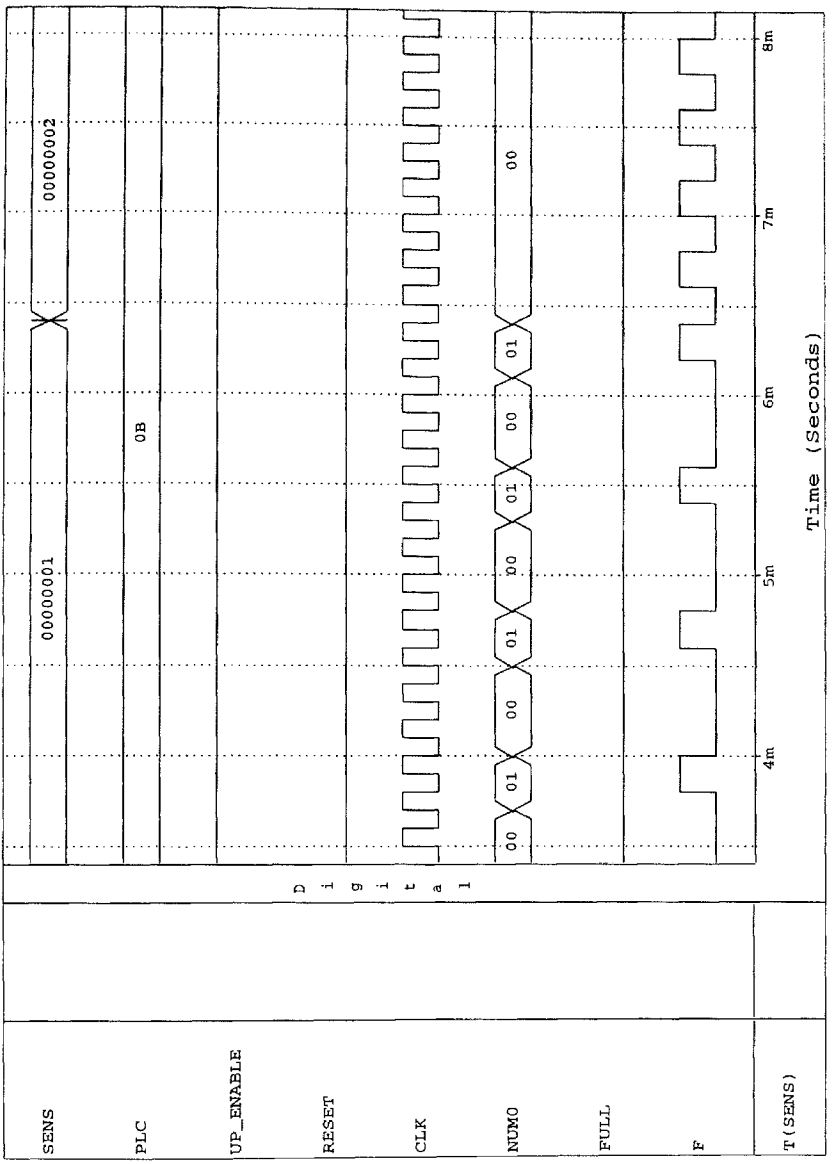

Figure B.16(a) Simulation Results of Muxcnt2 Module.

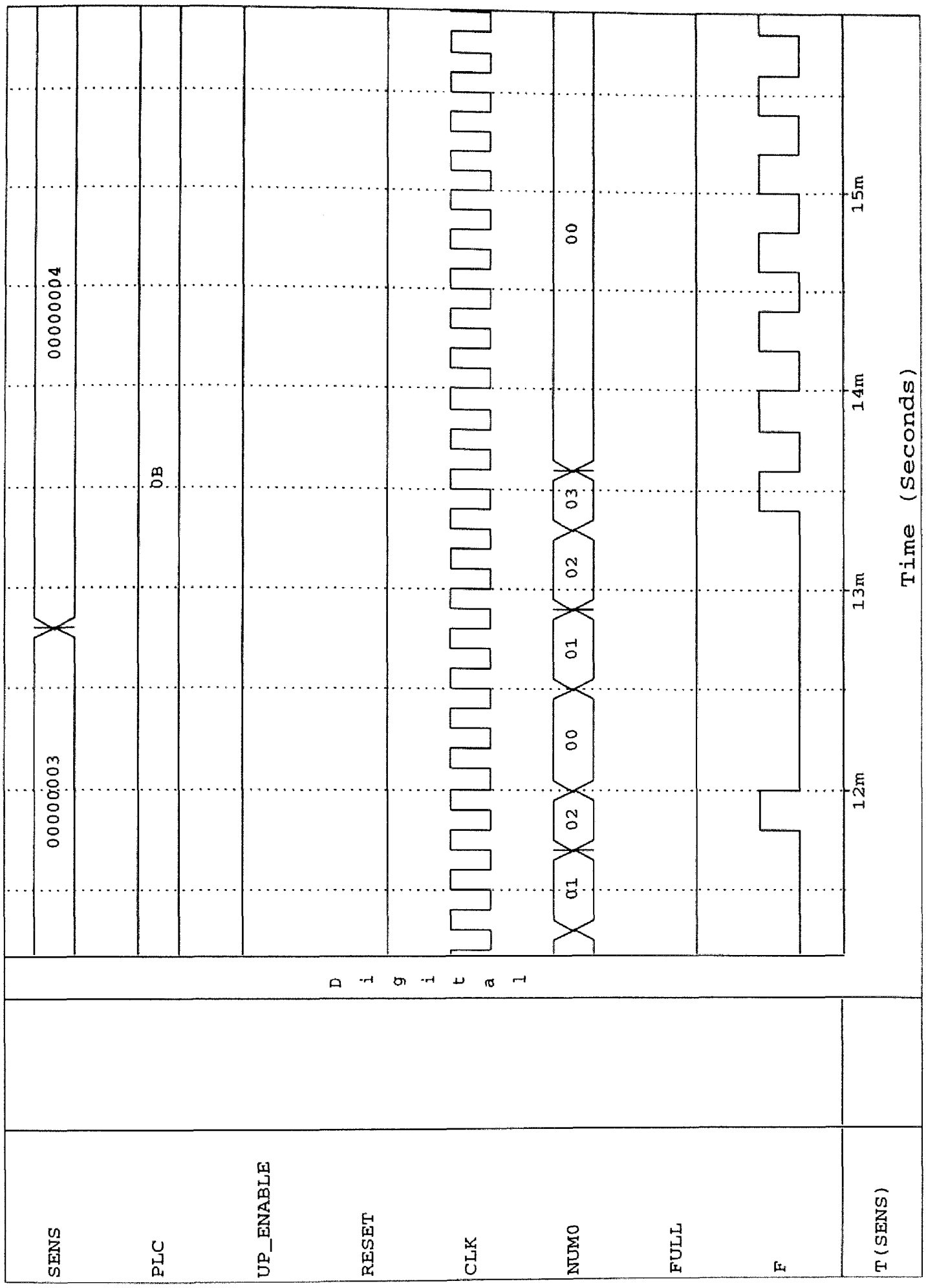

Figure B.16(b) Simulation of Muxcnt2 Module. (Zoomed View)

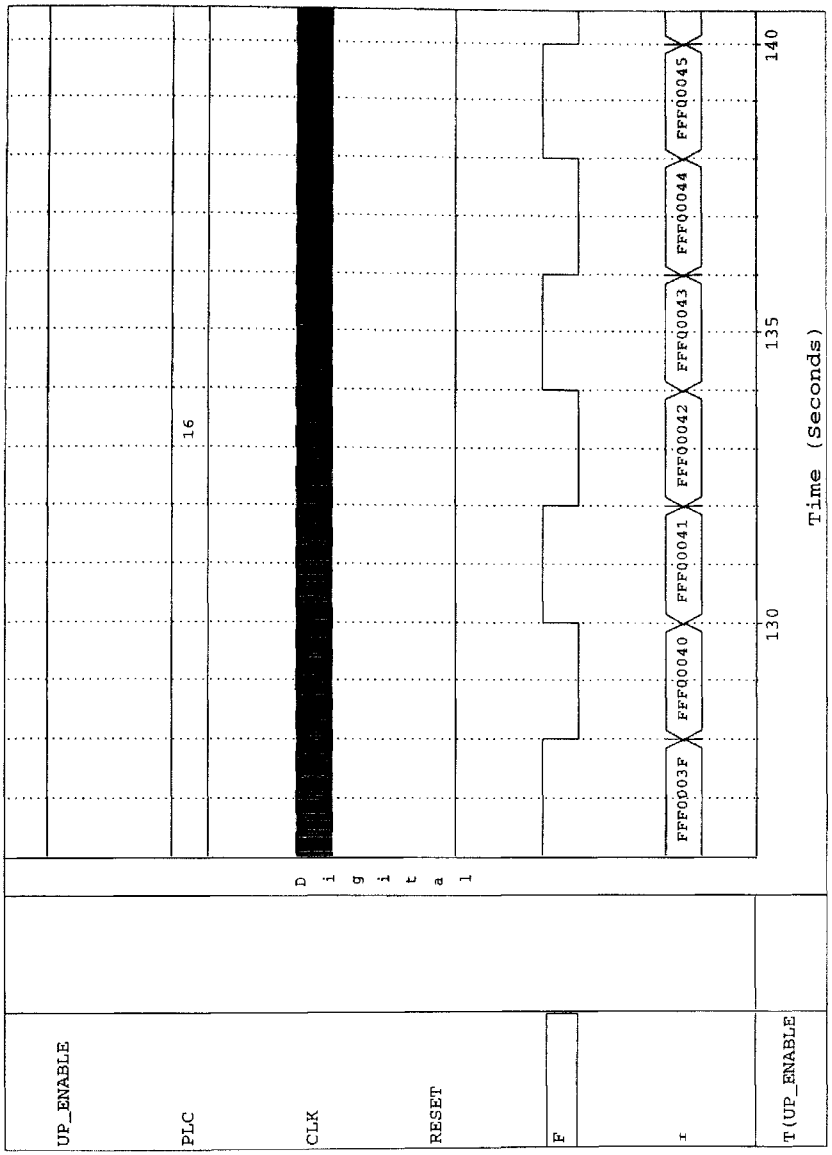

Figure B.17(a) Simulation Results of the Space_Cntr module.

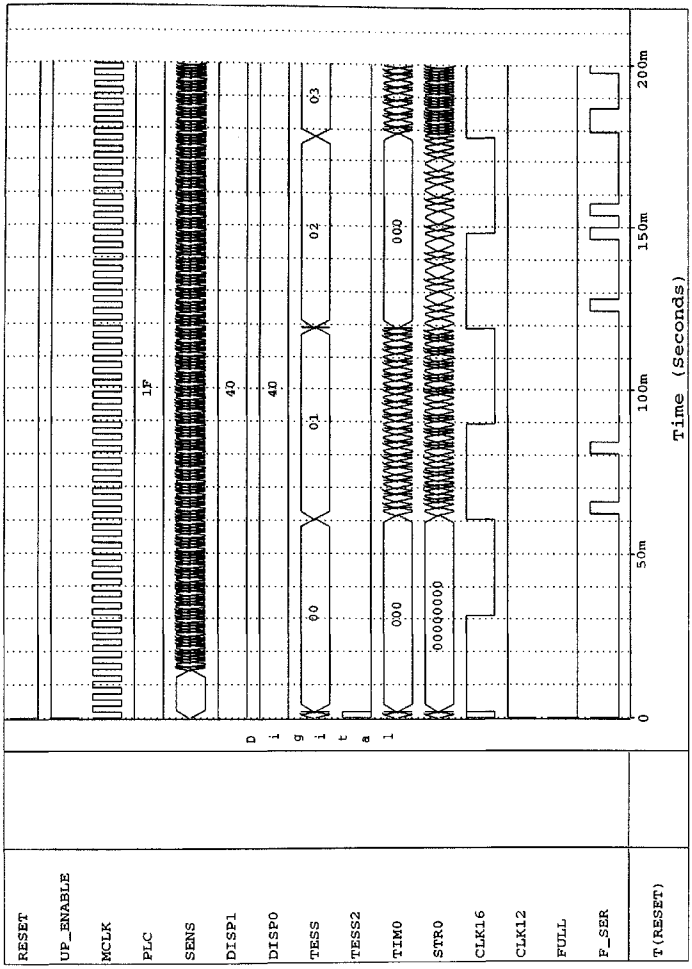

Figure B.18 Results of System Simulation

APPENDIX C

SYNTHESIZED SCHEMATICS

OF THE SPACE MANAGEMENT SYSTEM

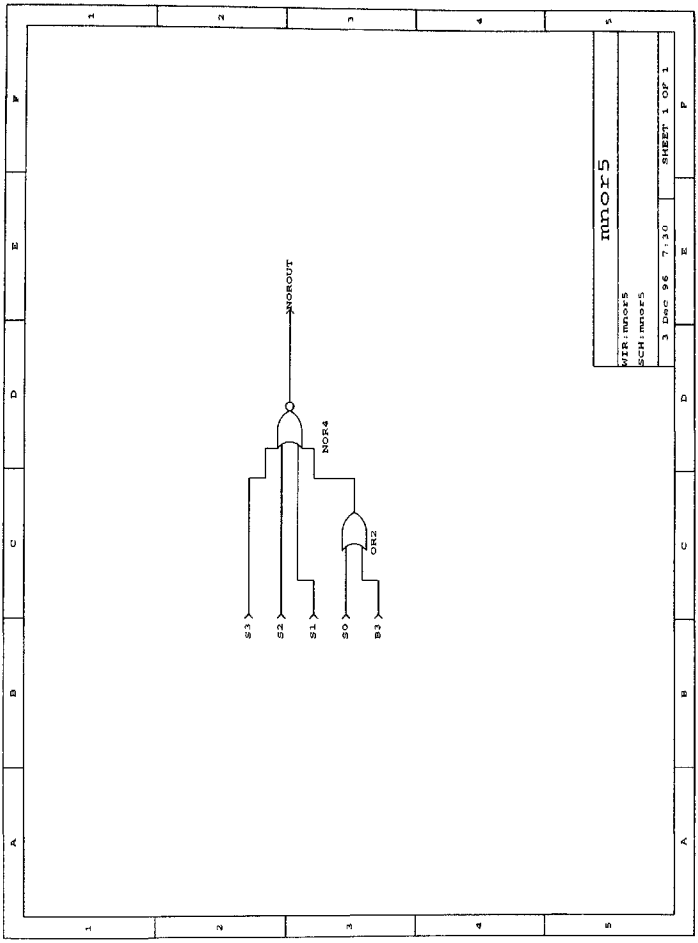

Figure C.1 Synthesized Schematic of a 5-input NOR gate.

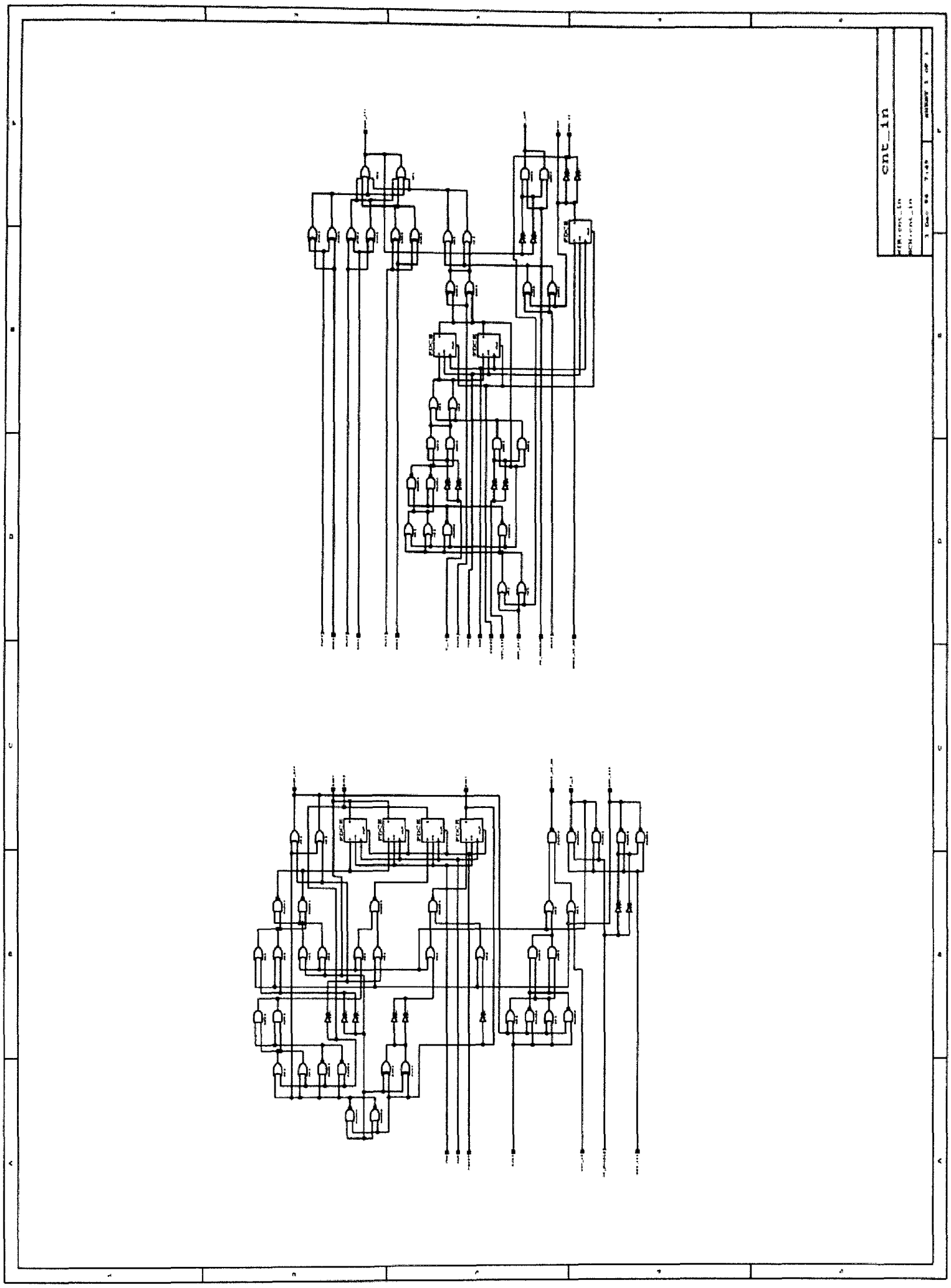

Figure C.2 Synthesized Schematic of 5-Bit Counter

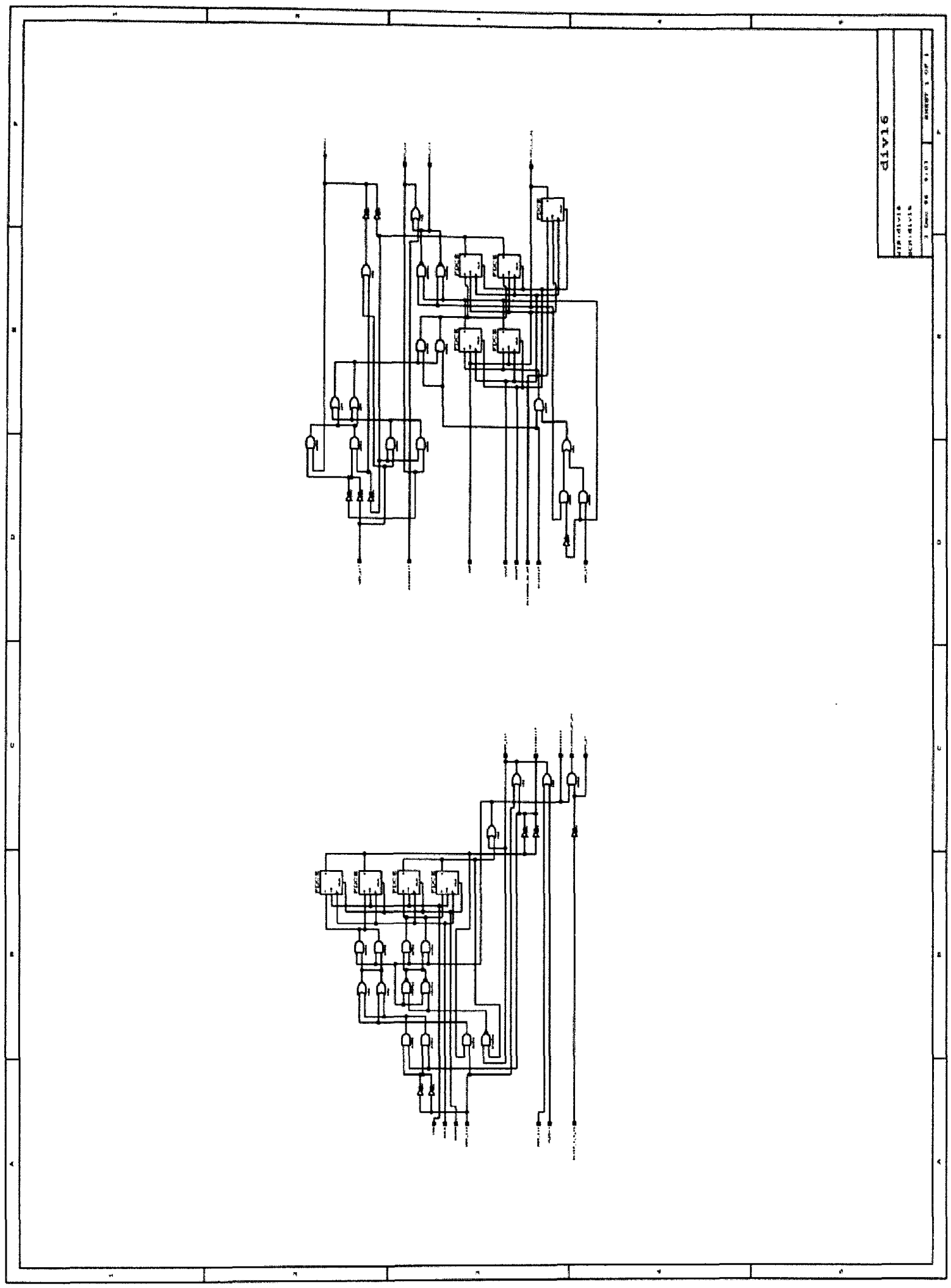

Figure C.3 Synthesized Schematic of Divide by 16 counter.

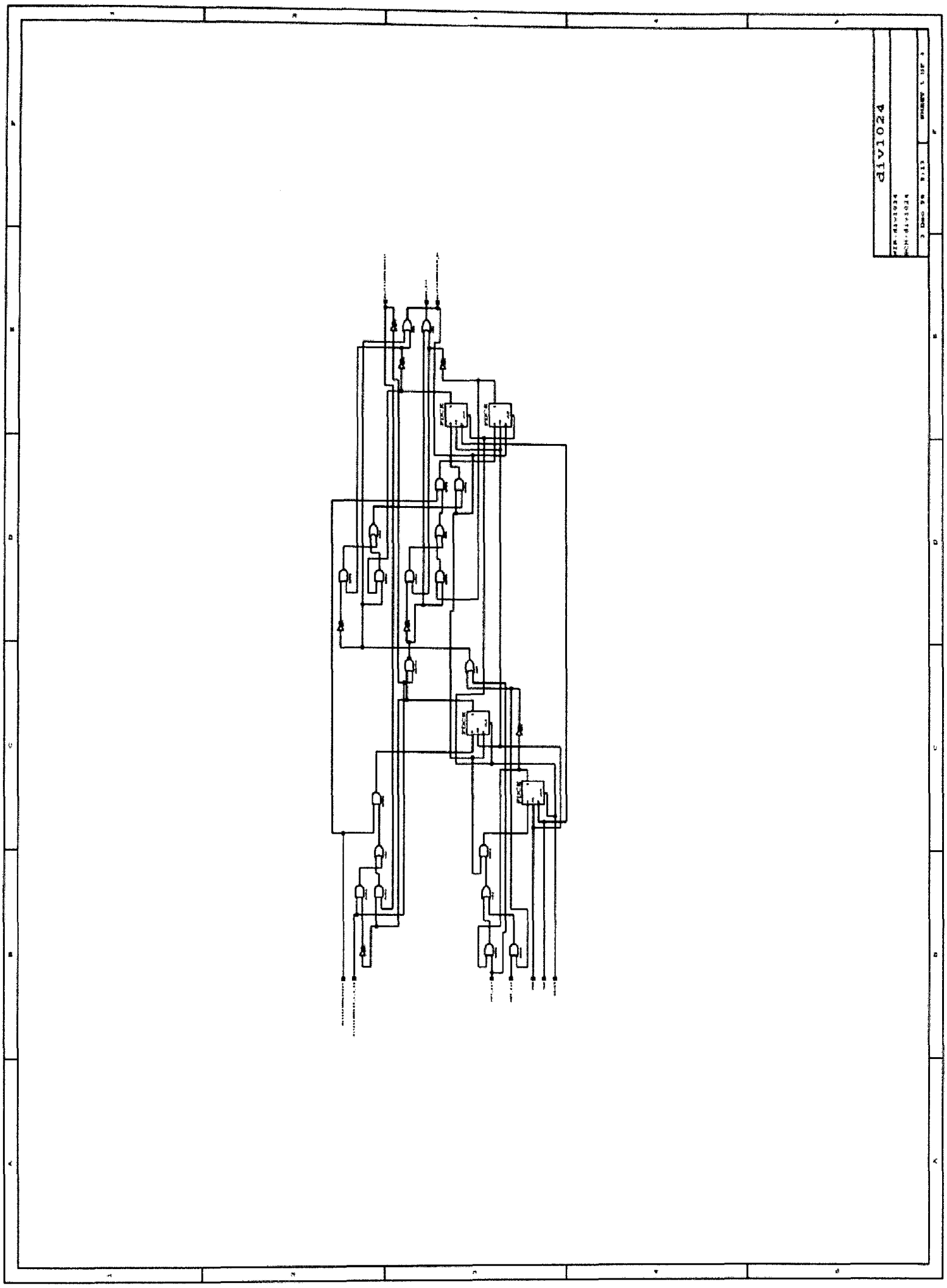

Figure C.4 Synthesized Schematic of Divide by 1024 counter.

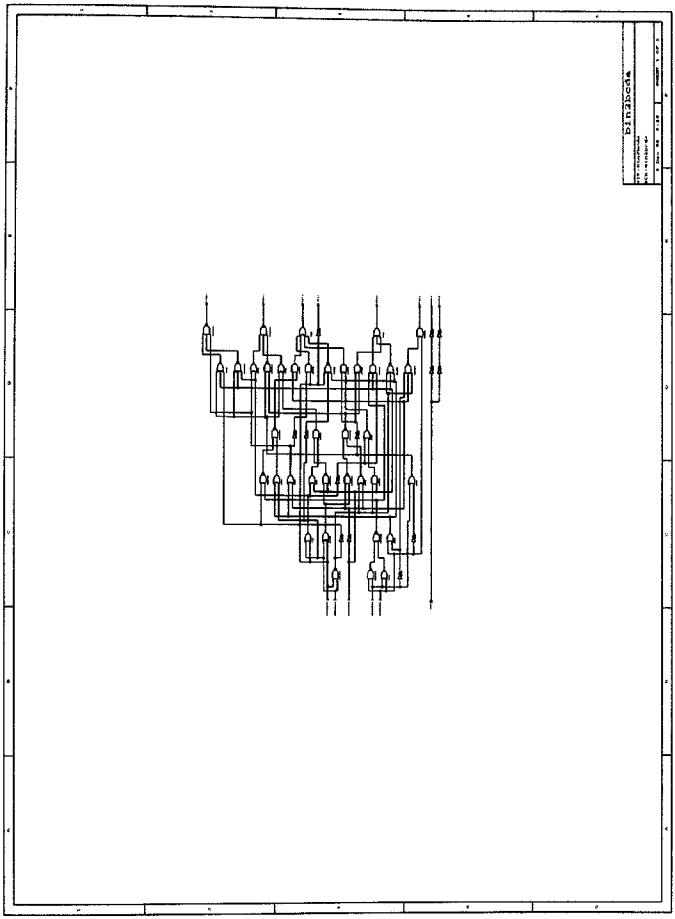

Figure C.5 Synthesized Schematic of Binary to BCD Converter.

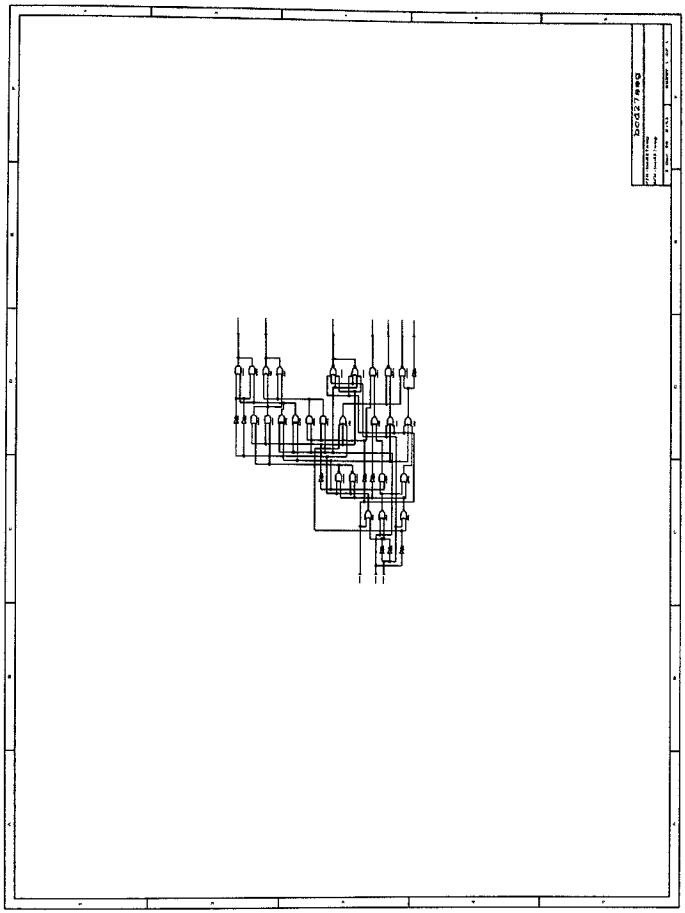

Figure C.6 Synthesized Schematic of BCD to Seven Segment Converter.

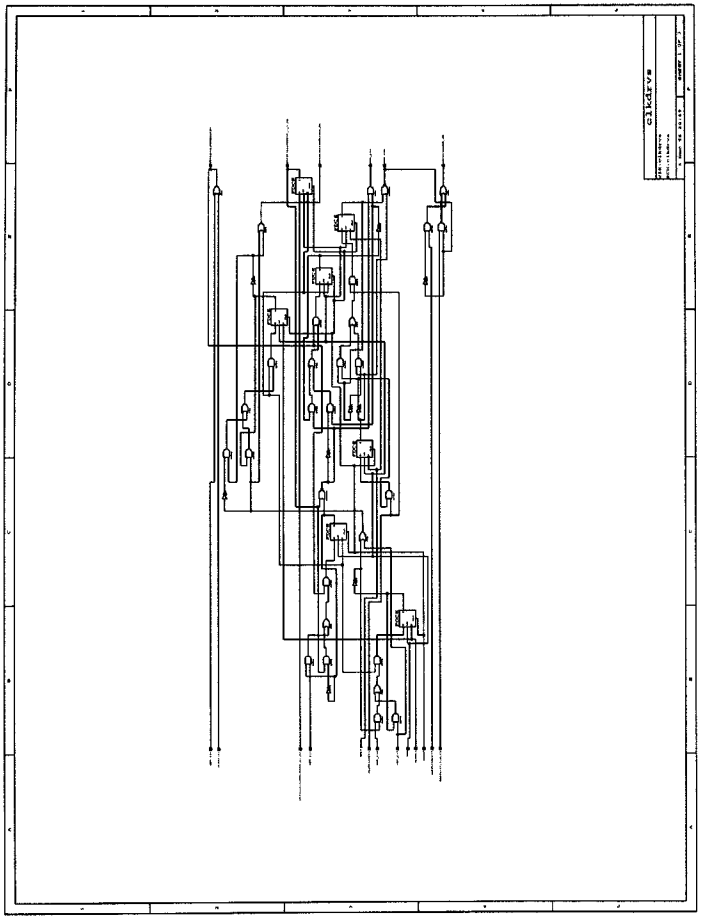

Figure C.7(a) Synthesized Schematic of Clock Generator.

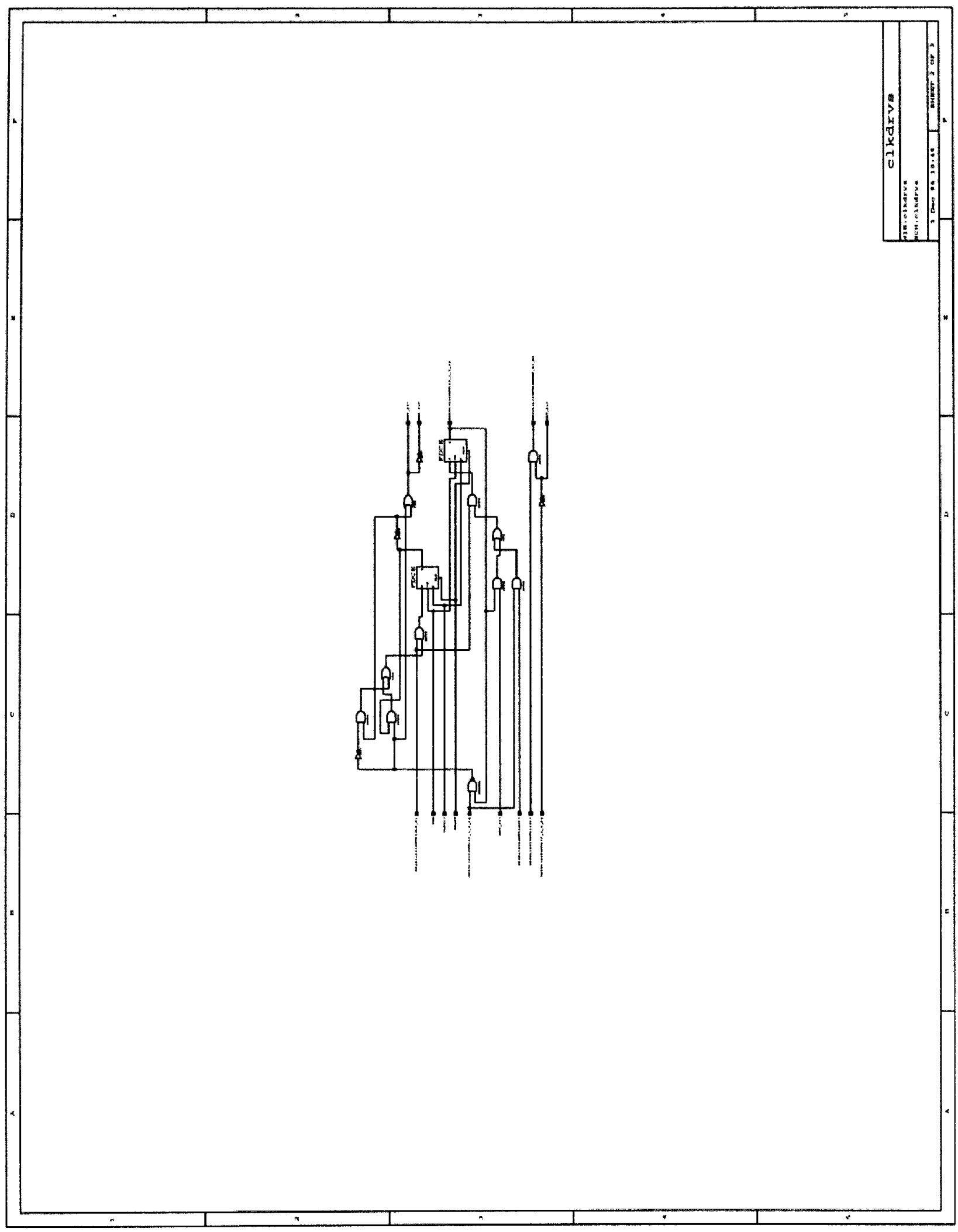

Figure C.7(b) Synthesized Schematic of Clock Generator

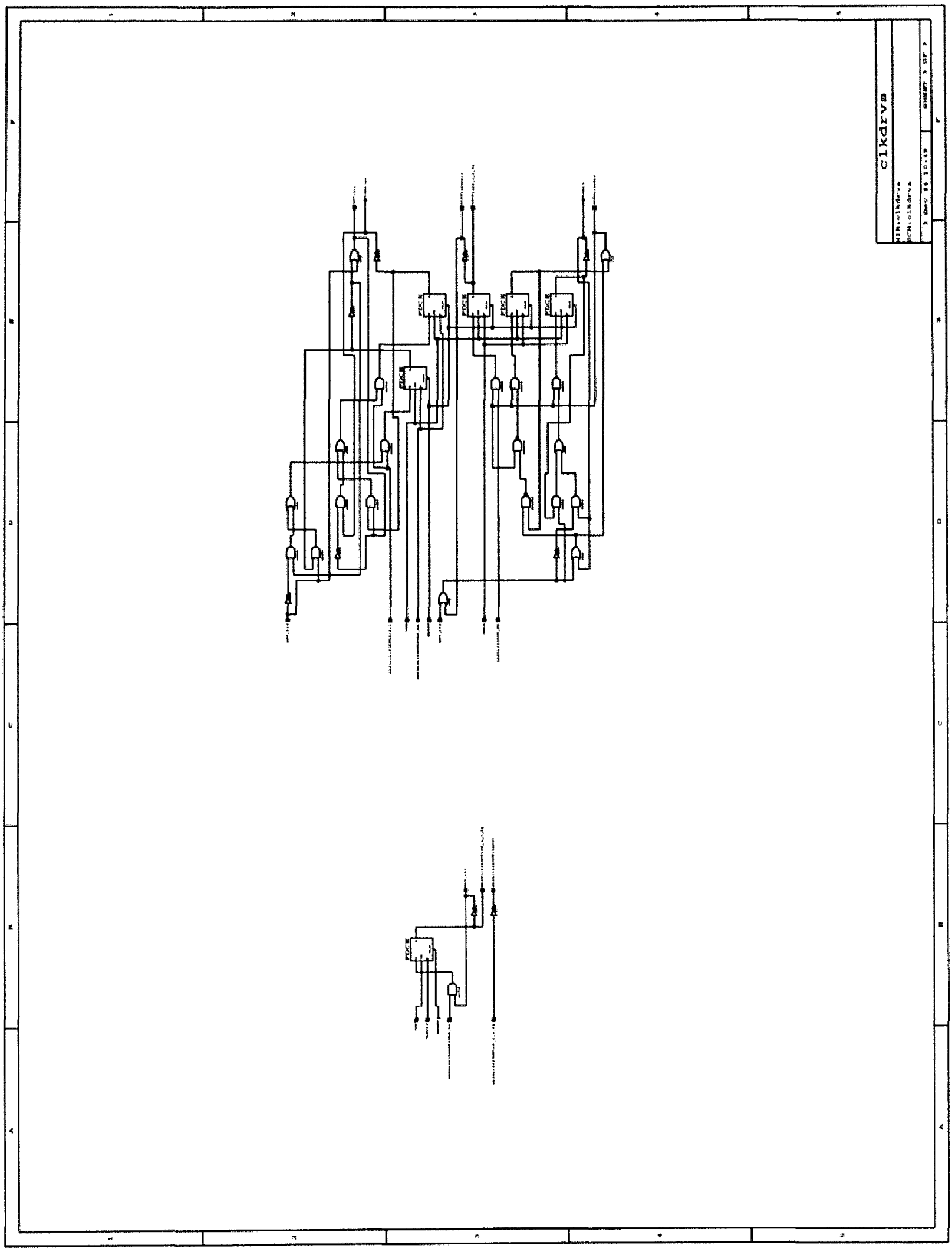

Figure C.7(c) Synthesized Schematic of Clock Generator

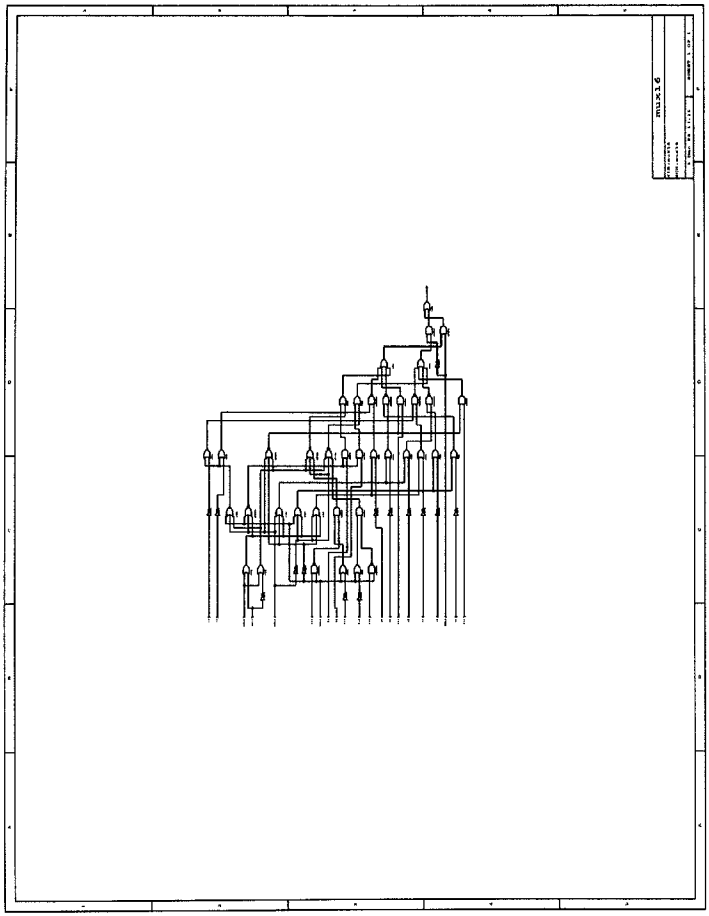

Figure C.8 Synthesized Schematic of 16 to 1 Multiplexer

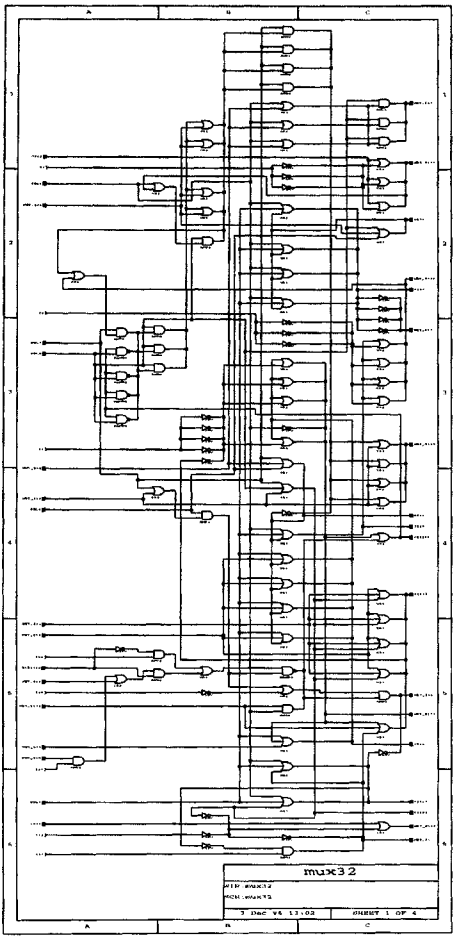

Figure C.9(a) Synthesized Schematic of 32 to 1 Multiplexer

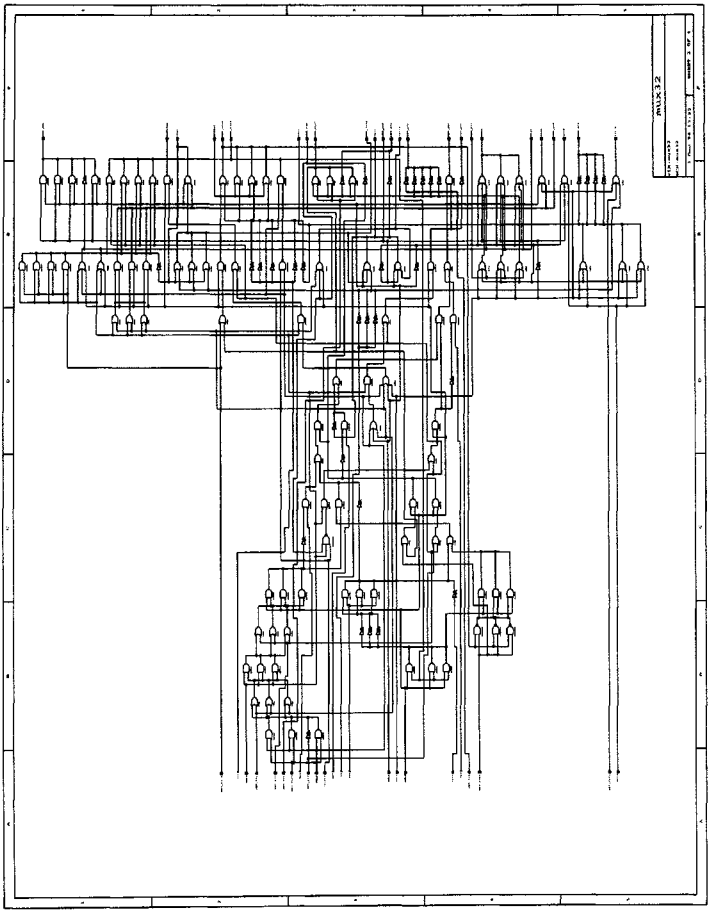

Figure C.9(b) Synthesized Schematic of 32 to 1 Multiplexer

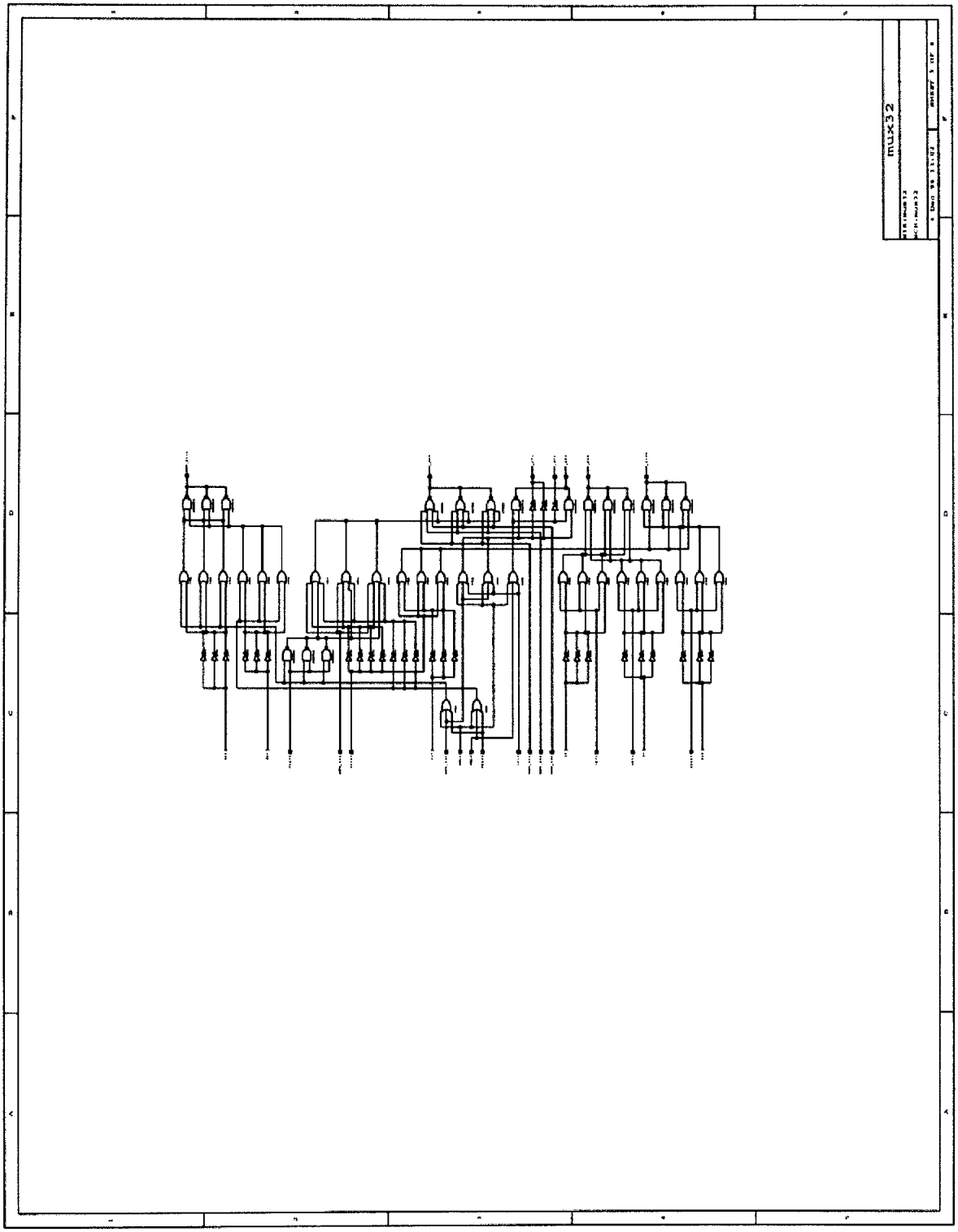

Figure C.9(c) Synthesized Schematic of 32 to 1 Multiplexer

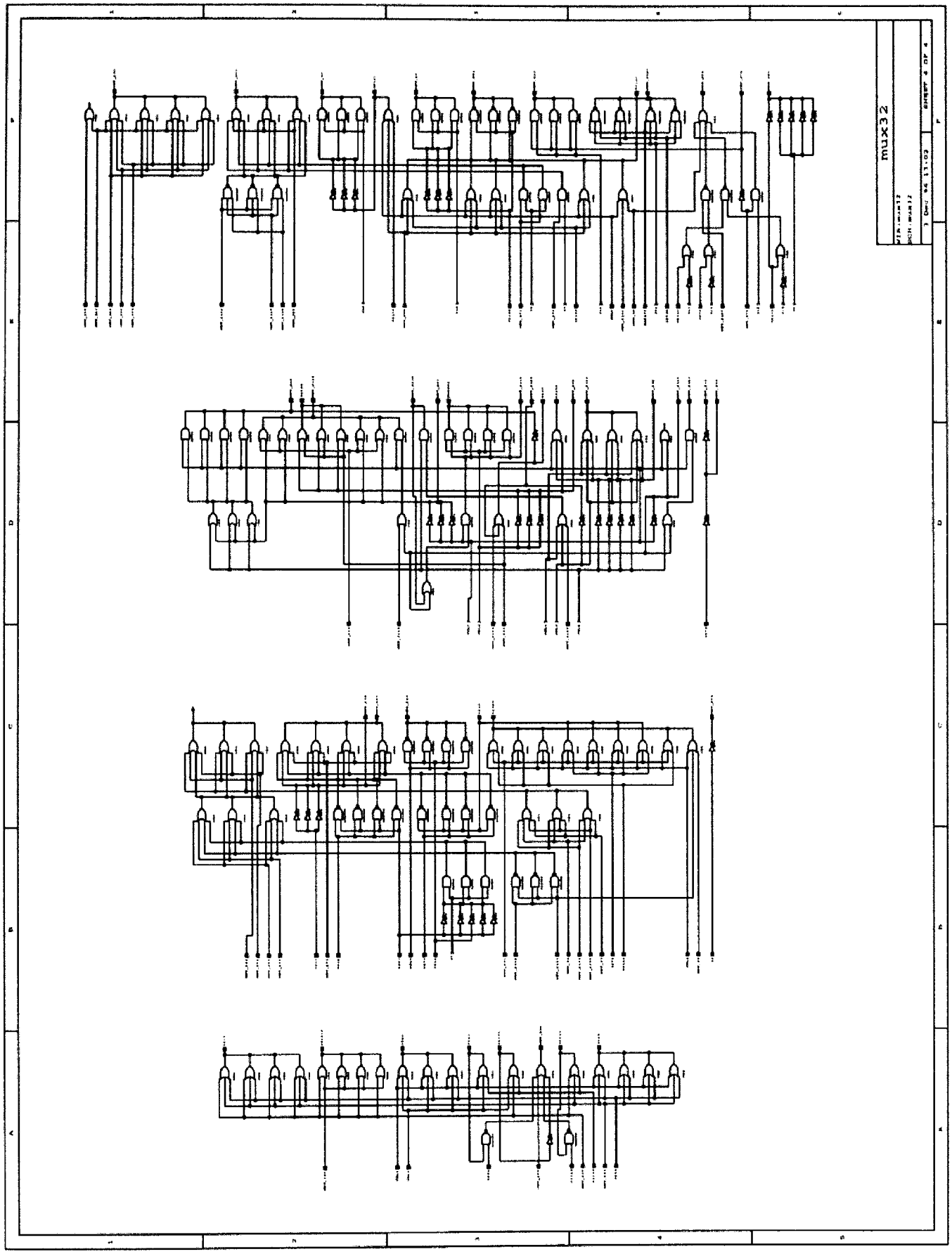

Figure C.9(d) Synthesized Schematic of 32 to 1 Multiplexer

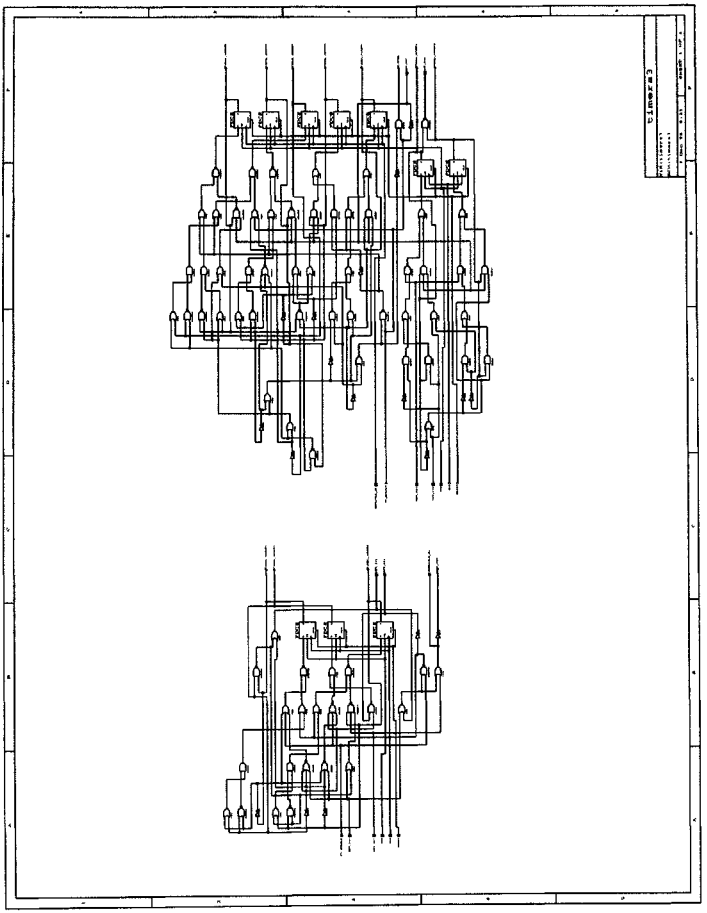

Figure C.10 Synthesized Schematic of Timer Module

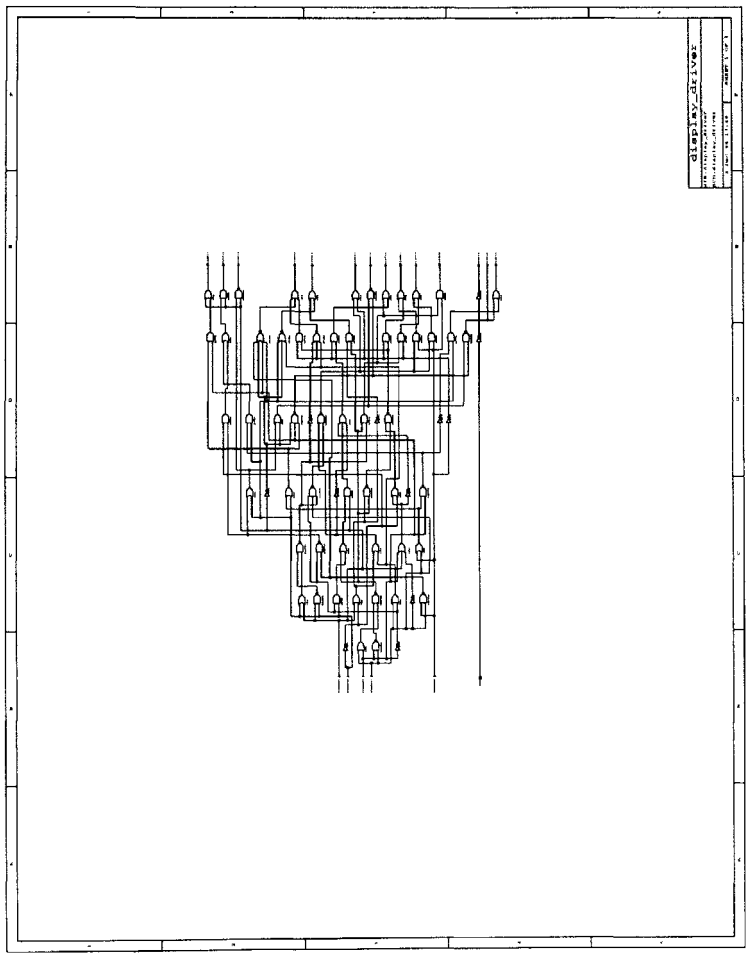

Figure C.11 Synthesized Schematic of Display Driver

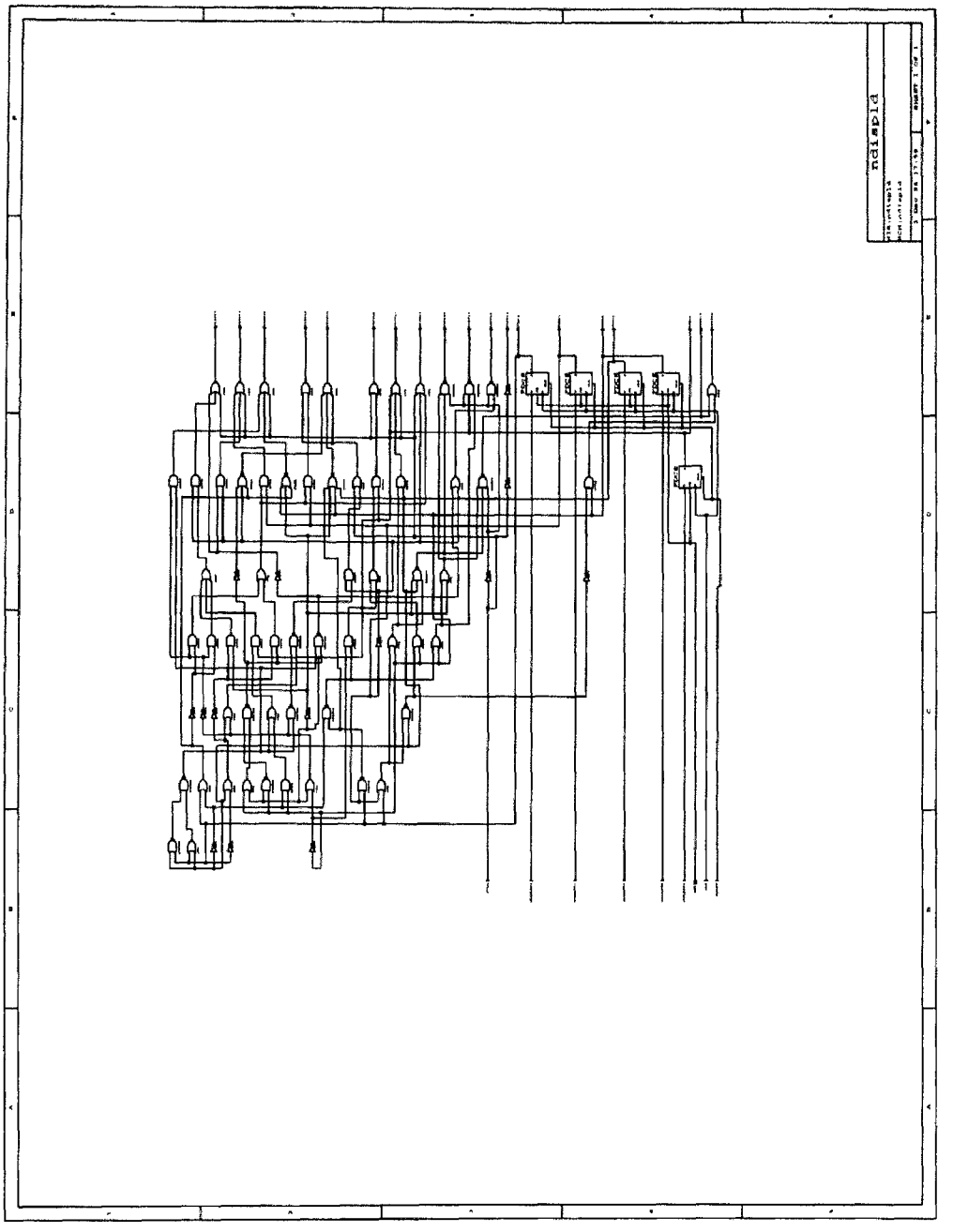

Figure C.12 Synthesized Schematic of Display Module

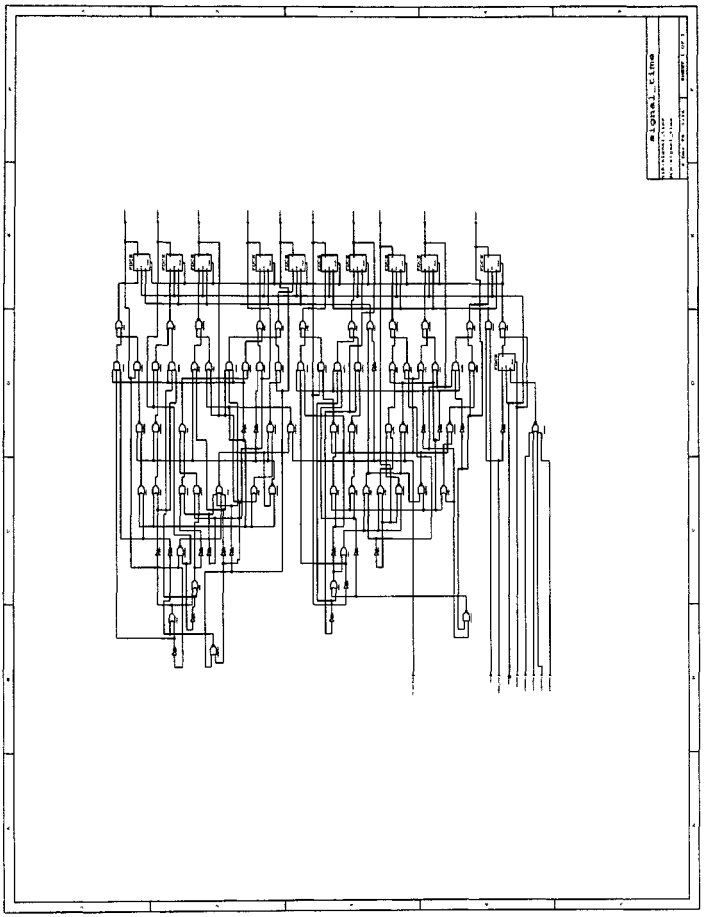

Figure C.13 Synthesized Schematic of Signal Timer Module

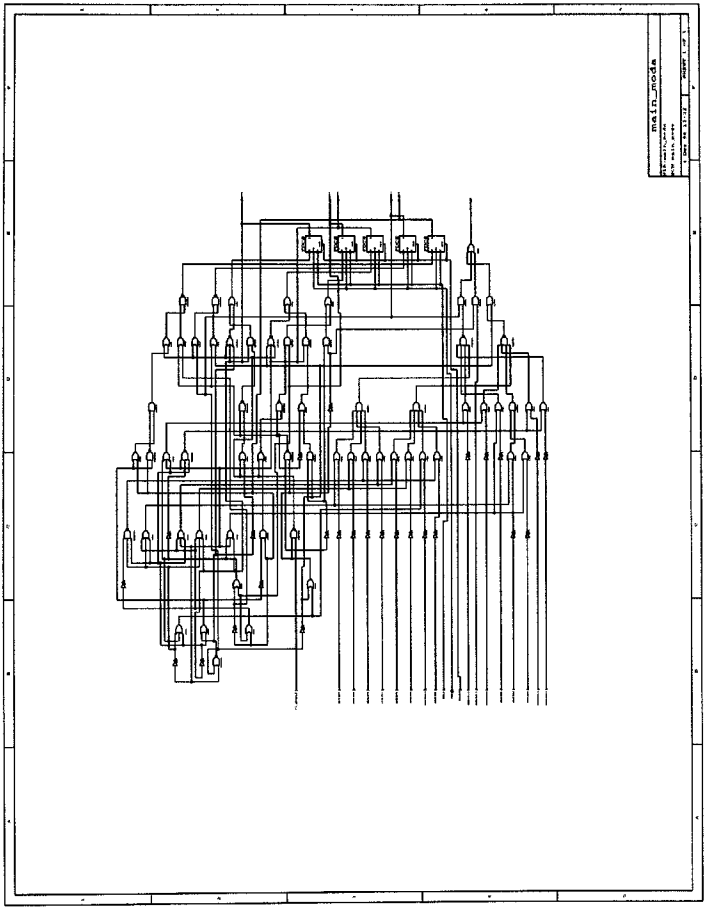

Figure C.14 Synthesized Schematic of the Main Module

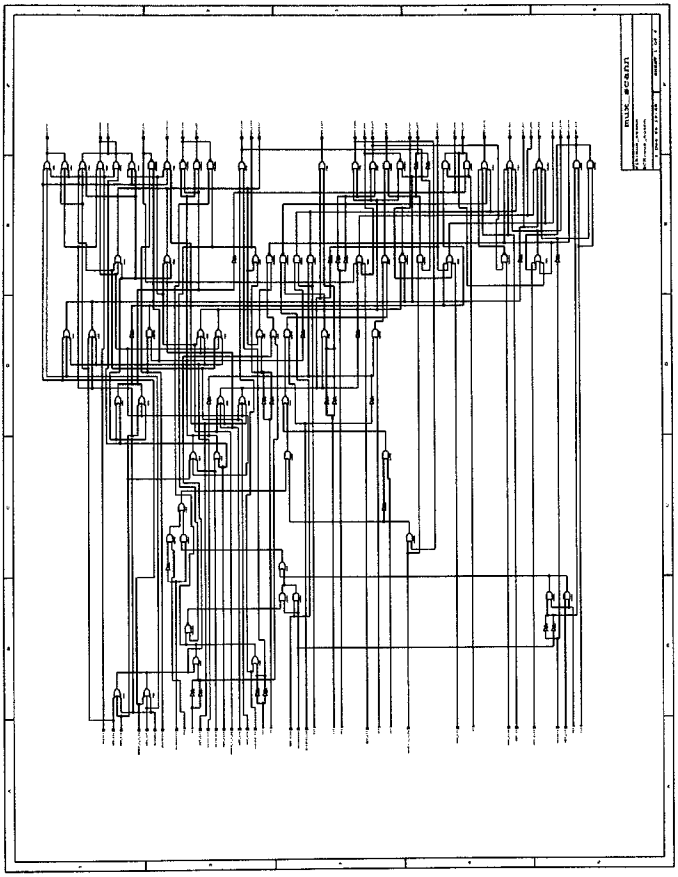

Figure C.15(a) Synthesized Schematic of a Mux_scann module.

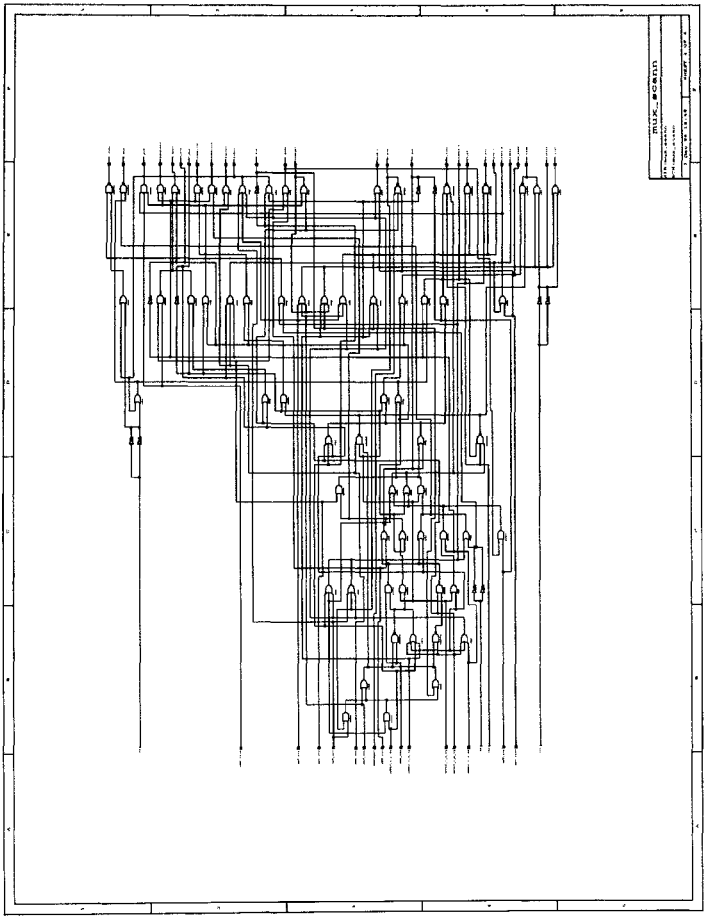

Figure C.15(b) Synthesized Schematic of a Mux_scann module.

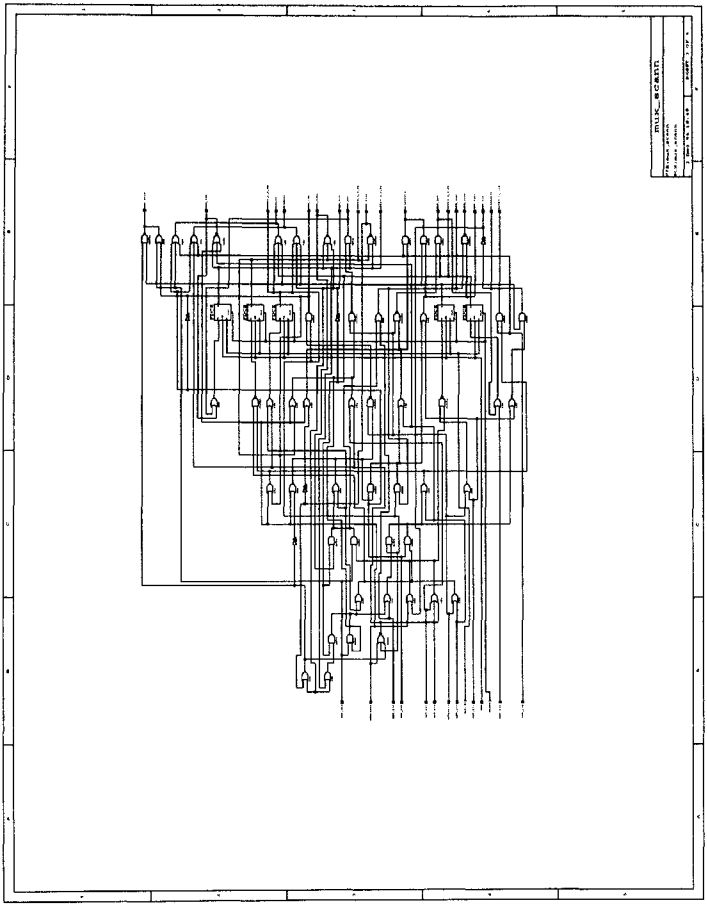

Figure C.15(c) Synthesized Schematic of a Mux_scann module

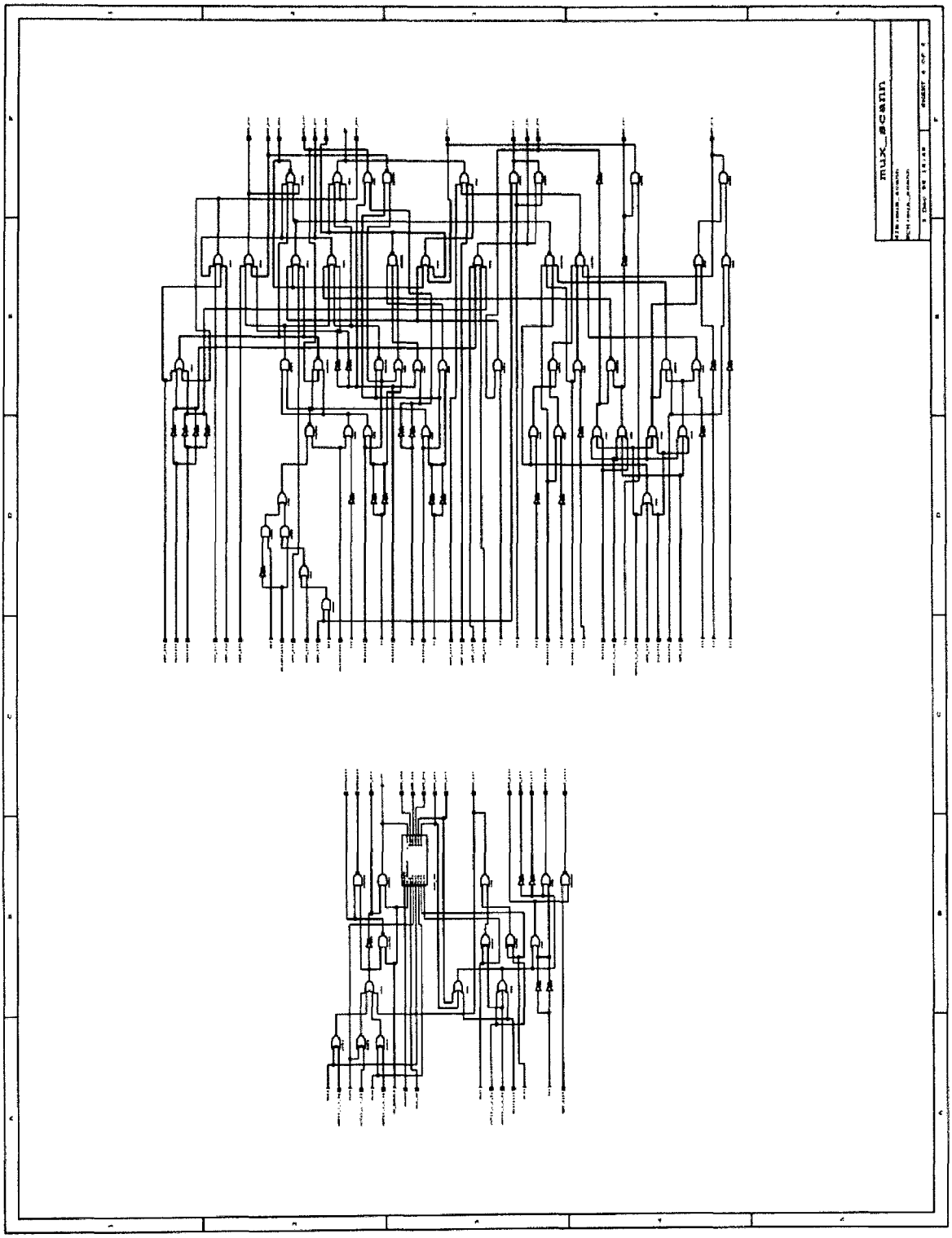

Figure C.15(d) Synthesized Schematic of a Mux_scann module

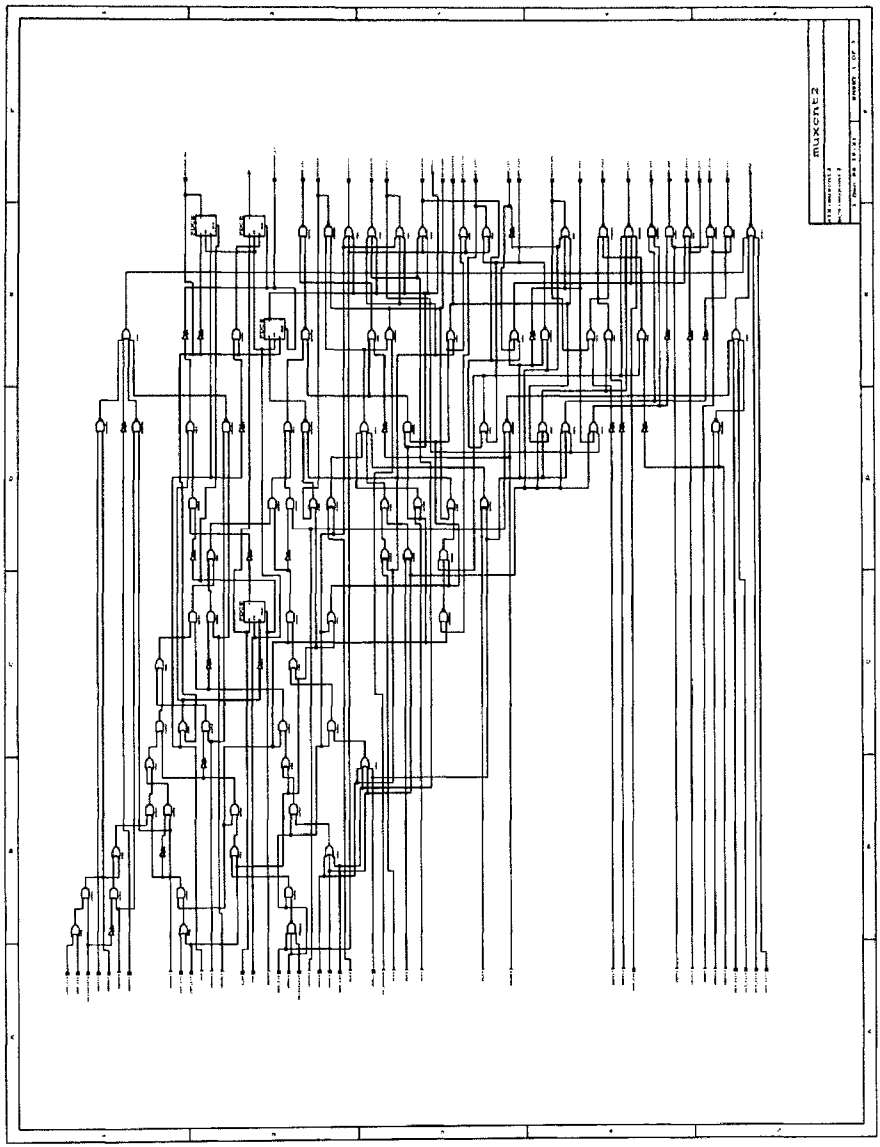

Figure C.16(a) Synthesized Schematic of Muxcnt2 Module.

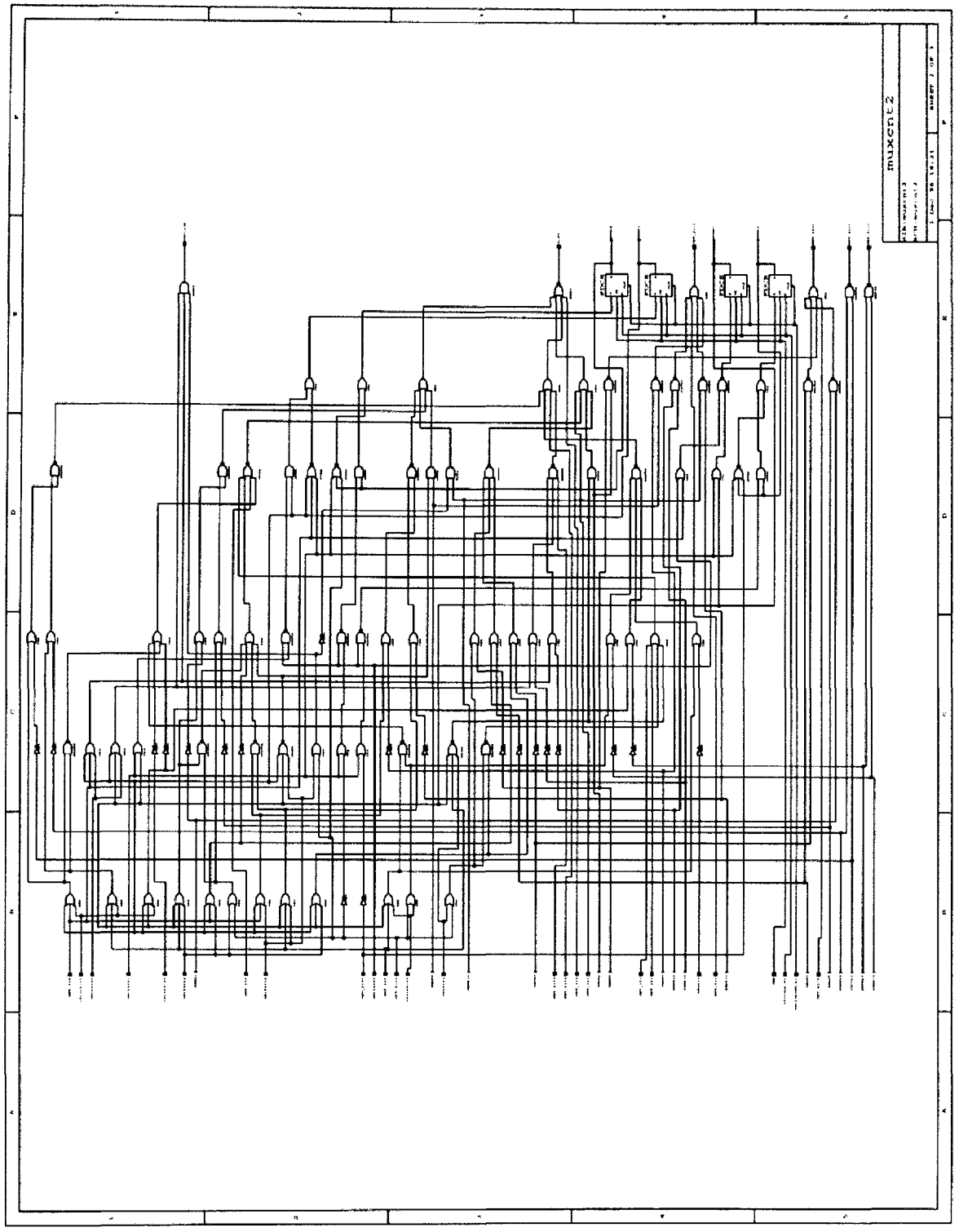

Figure C.16(b) Synthesized Schematic of Muxcnt2 Module.

Figure C.16(c) Synthesized Schematic of Muxcnt2 Module.

Figure C.17(a) Synthesized Schematic of Space_cntr Module

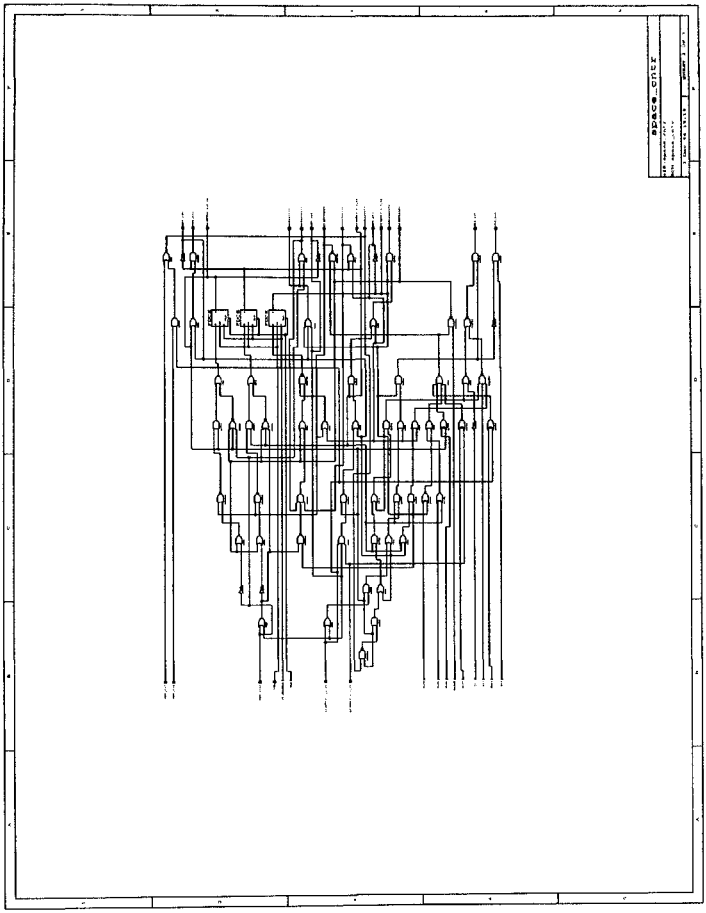

Figure C.17(b) Synthesized Schematic of Space_cntr module.

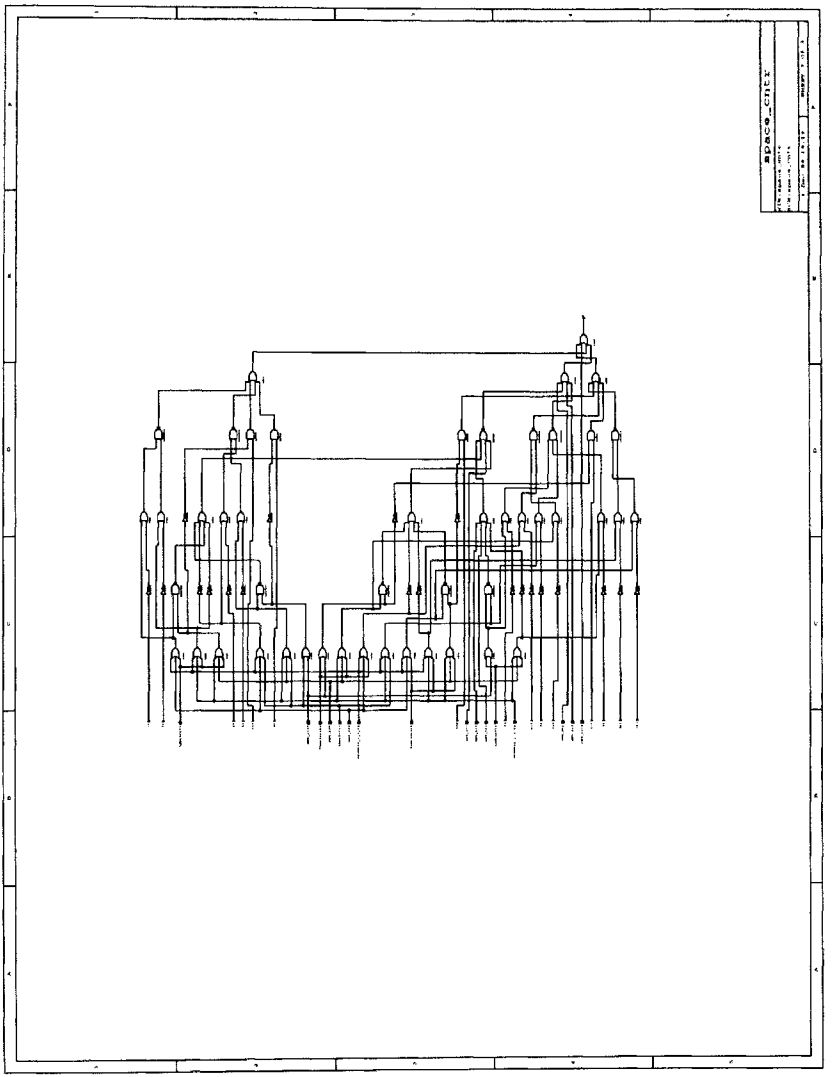

Figure C.17(c) Synthesized Schematic of Space_cntr module.

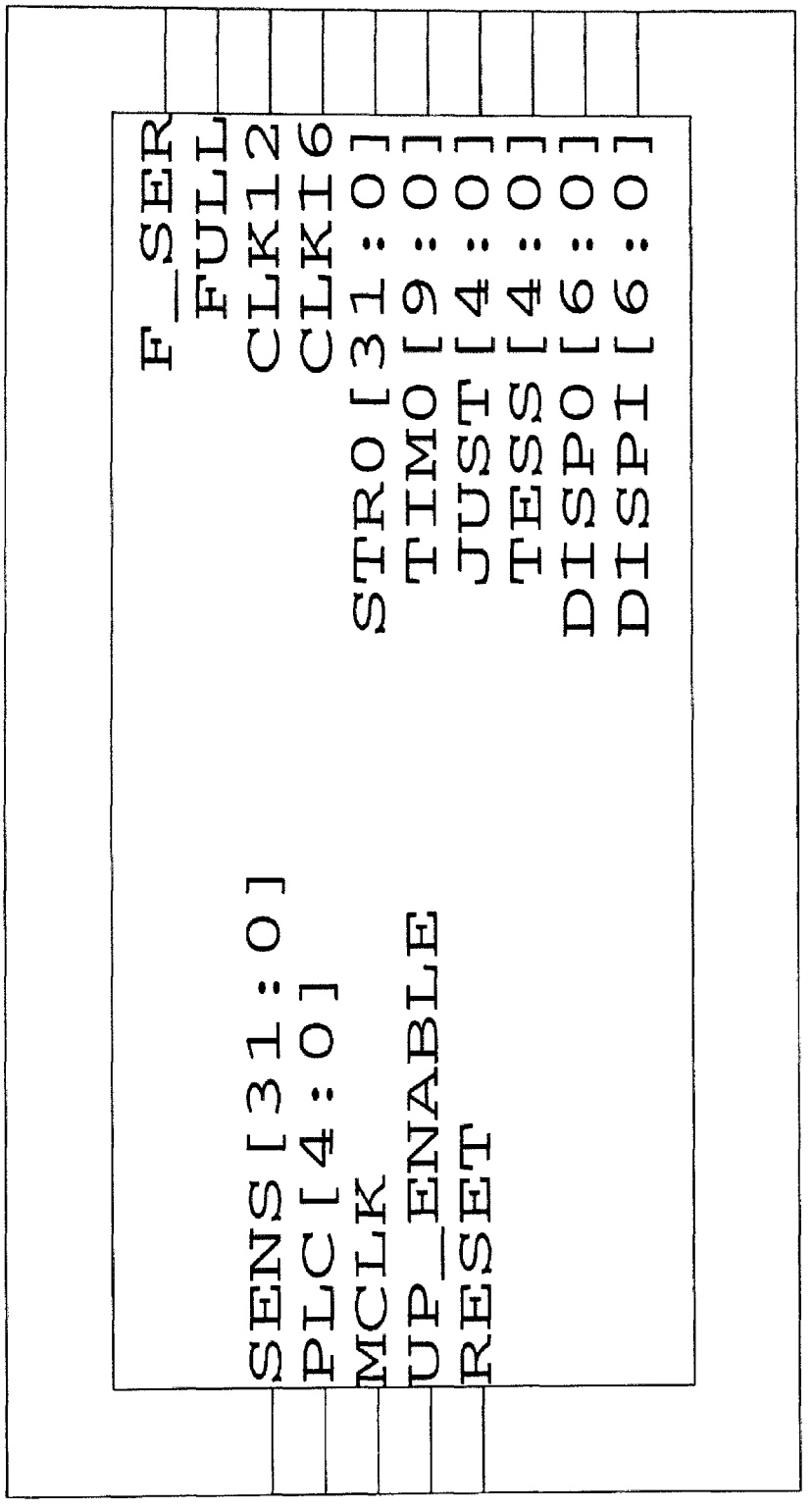

Figure C.18 Synthesized Symbol of Complete System.