# DESIGN OF THE INTERCONNECTION NETWORK FOR THE EARTH SYSTEM

OI-LING OLIVER TSUI

A THESIS

IN

THE DEPARTMENT

OF

ELECTRICAL AND COMPUTER ENGINEERING

PRESENTED IN PARTIALLY FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF APPLIED SCIENCE

CONCORDIA UNIVERSITY

MONTRÉAL, QUÉBEC, CANADA

April 1997

0-612-26006-2

# Abstract

## Design of the Interconnection Network for the EARTH System

Oi-Ling Oliver Tsui

This thesis describes the design of a low-cost and high-bandwidth interconnection network for the EARTH multithreaded multiprocessor system. The interconnection network consists of a hierarchy of 4 × 4 crossbar switches, and a network interface on each EARTH processor node. The interconnection network is capable of transferring multiple messages simultaneously between nodes. The interconnection network also supports random routing that allows the data connection to be routed automatically around the busy ports. In addition, the interconnection network provides broadcast facility which allows one node to send a message to all destination nodes. The logical design of the 4 × 4 crossbar switch and the network interface has been translated to a physical implementation based on LSI Logic's LCA300K compacted array technology. Although the 4 × 4 crossbar switch is designed for the needs of the EARTH system, it can be used in a wide variety of data communication networks.

## Acknowledgment

I would like to express my sincerest thanks to my thesis supervisor. Dr. Herbert Hum. for all his guidance and advice during the course of this research. I would also like to thank David Hargreaves, who is the engineer of the VLSI laboratory, for his technical support throughout the implementation and simulation of this project. Finally, I would also like to express my deepest thanks to my family for their unwavering support and encouragement throughout my study.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Since conventional serial RISC-based computers were invented in early 1970s, their performance has steadily improved to match the needs of engineering and business applications. However, it is impossible to achieve further improvements in the performance of such conventional serial computers indefinitely due to the fundamental physical limitation imposed by the speed of light. Recent trends show that the performance of these computers is beginning to saturate [16].

As very large scale integration (VLSI) technology advances, it is now possible to build very fast and low-cost microprocessors. The increase in demand and production of these microprocessors keeps driving their prices down. Therefore, an inexpensive parallel computer can be constructed by connecting off-the-shelf microprocessors together. Also, connecting microprocessors into parallel computer overcomes the saturation of performance in serial computers [16]. Typically, the cost of such a parallel computer is considerably much less than the massively parallel computers such as the Thinking Machine CM-5 or the super computer offerings of SGI/Cray.

## 1.1 Multiprocessors

According to Flynn [9], computers are classified into four categories based on the type of instruction and data streams. They are:

- Single instruction stream, single data stream (SISD);
- Single instruction stream, multiple data stream (SIMD);
- Multiple instruction stream, single data stream (MISD);
- Multiple instruction stream, multiple data stream (MIMD).

A conventional serial computer falls into the SISD category. A vector processor machine falls under the SIMD category in which the same instruction is manipulated by multiple processors using different data streams. A multiprocessor system in which each processor is capable of executing its own instruction and manipulating its own data is classified as MIMD.

MIMD architecture has become the choice for general-purpose multiprocessors system in recent years [7]. This view is supported by the following factor: MIMD architecture offers flexibility and scalability. MIMD machines can function either as single-user or multi-user machines. If MIMD machines are running on only one application, a high performance may be obtained. On the other hand, MIMD machines are capable of running many tasks simultaneously. Theoretically, a higher performance on MIMD machines can be achieved by adding more processors into the system. Ideally, MIMD machines with $n$ processors should simply be $n$ times faster than SISD machines built from the same technology but this is rarely the case. The primary reason is that the speedup of MIMD machines depends heavily on the ratio between the serial part and the parallel part of a given application [1, 13].

Although multiprocessors have been considered as the direction of the future development of the computer systems, Arvind and Iannucci state that two fundamental issues in multiprocessing remain to be addressed. They are [3]:

- Most von Neumann processors are likely to "idle" during long memory reference. and such references are unavoidable in parallel machines.
- Waits for synchronization events often require task switching, which is expensive on von Neumann machines. Therefore. only certain types of parallelism can be exploited efficiently.

Multiprocessor systems based on multithreaded architectures have been proposed to be the potential processing nodes for future parallel machines due to their toleration of long latencies for interprocessor communication and synchronization in parallel program executions [15]. Multithreaded processors hide the long memory latency by suspending the execution of the current thread upon encountering the long memory latency operation and performing rapid context switching to another thread.

## 1.2 The EARTH SYSTEM

EARTH (*Efficient Architecture for Running THreads*) [14] is a multiprocessor model based on multithreaded architectures. Although no prototype of EARTH has been implemented yet. an implementation of the EARTH's architecture would be similar to the MANNA (*Massively parallel Architecture for Non-numerical and Numerical Application*) [12] multiprocessor system developed at GMD-FIRST in Berlin. Germany.

The EARTH architecture is illustrated in Figure 1.1. Each processing element (PE) in the EARTH model consists of an Execution Unit (EU). Synchronization Unit (SU). The SU

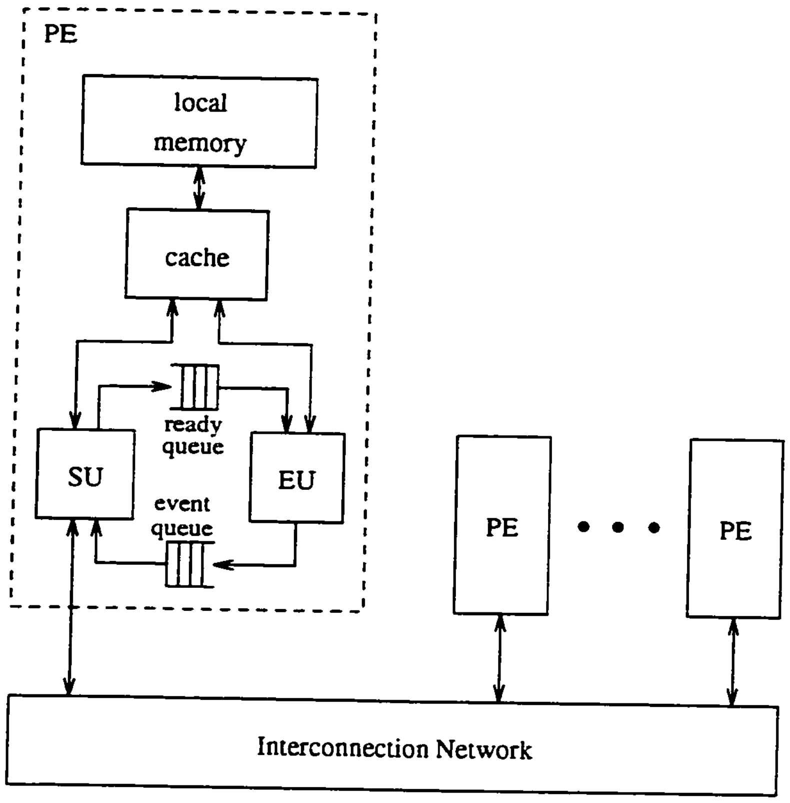


Figure 1.1: The EARTH Architecture

and EU share a local memory, which is cached for better performance. The local memory is part of a distributed shared memory architecture in which the aggregate of the local memories of all processing elements represents a global memory address space. The EU is a high-end off-the-shelf processor which is responsible for executing threads. The SU supports dataflow-like thread synchronizations and communication with other processing nodes. Unlike the MANNA machine which uses two Intel i860 XP RISC processors for execution unit and synchronization unit. EARTH recommends the use of one processor dedicated to be the EU while the SU is implemented as a customized VLSI chip. This is because synchronization events involve only simple ALU tasks which do not require floating point units. The EU and SU are linked together by two buffering queues as shown in Figure 1.1. The EARTH system operates as follows:

**EU:** The EU fetches a *thread id* from the ready queue, and executes the thread. If long latency memory operations are issued, EU will place them into the event queue and perform context switching to another ready thread.

**SU:** The SU fetches incoming messages from the event queue (from local EU), and the network (from remote processing node). In response to the messages, the SU reads/writes to local memory, sends messages, receives messages, updates synchronization variables, and arranges threads for execution by placing their *thread id's* to the ready queue.

**Interconnection Network:** The interconnection network provides physical connections among the EARTH nodes. All the incoming and outgoing messages handled by the SU are transferred through the interconnection network.

In the EARTH system, dividing the execution and synchronization into two separate units has two major advantages. First, incoming synchronization requests will not interrupt the processor since these requests will be handled by the SU. Second, long latency operations such as block moves will not tie up the processor especially when the interconnection network is congested and the block move is stalled. It is the responsibility of the compiler to place long latency operations at the end of a thread. When the processor encounters long latency operations, it will simply switch to another thread.

## 1.3 Proposed Interconnection Network for the EARTH System

The purpose of this thesis work is to design the interconnection network for the EARTH multiprocessor system. The organization of the interconnection network design is shown in Figure 1.2. The interconnection network consists of a data network, and a set of identical network interfaces which interface each EARTH processing node to the data network. The following goals drove our interconnection network design:

- The interconnection network must deliver high performance, and support point-to-point communications among EARTH processing nodes. We also want the interconnection network to be capable of handling multiple connections simultaneously so that the bandwidth can be maximized.

- The interconnection network must be cost-effective. Building a full crossbar network is costly [2]. Therefore, we focus our design on a multistage network since a multistage network can be constructed at significantly lower cost than a full crossbar network. A multistage network also produces a full connection network as a full crossbar network does. The full connection network means any source node can reach any destination

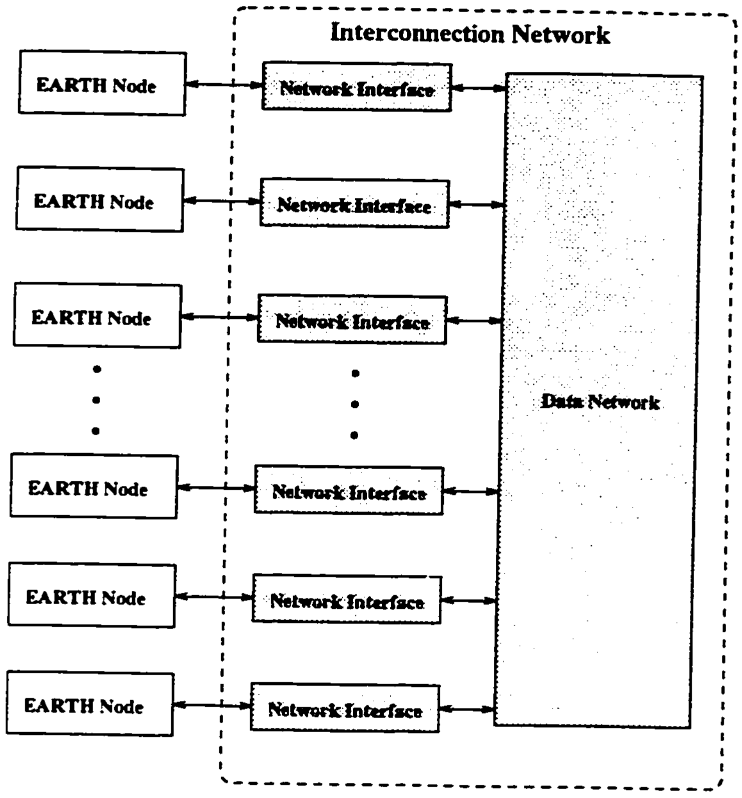


Figure 1.2: The Organization of Interconnection Network

node.

- The interconnection network must be scalable and flexible. The number of ports in the network are freely configurable. The ports can be connected to processing nodes or other units.

## 1.4 Problem Statement

A top-down approach is adopted for the design of the interconnection network. Many specific criteria are used in the design of the EARTH switching system but those considered important to the EARTH system are listed below:

- The interconnection network should be modularly expandable. It should be straightforward to add inputs and outputs, both physically and with respect to software.
- The interconnection network should be compact and should readily interface to processing nodes for the EARTH system.
- The design of network interface and data network could be implemented on two customized VLSI chips. There must be a minimum number of pins per input and output port. Since customized VLSI chips have only a relatively limited number of pins available, it is necessary to limit the number of pins required for control of the system in addition to the inputs and outputs which must be used.
- The system should use only standard subcircuits and available technology to assure practicality.

Other criteria which are more specific to the EARTH system are also considered important and are listed below:

- The interconnection network should be designed so that each input requests connection to needed outputs and no central controller is necessary. Control of the interconnection network should be simple and rapid, requiring a very short time with respect to actual data processing. There should be little control overhead.
- The interconnection network should be nonblocking in the sense that any input could be immediately connected to any output not already in use with no interference from existing connections.
- There must be a mechanism which allows the data connection to be routed automatically around busy ports. This mechanism is referred to as *random routing* [6].
- There must be a mechanism to inform the input that the output is busy.
- There should be a mechanism which allows the connection of one input to all outputs. This is often referred to as *message broadcast*. Message broadcast facility allows a single source node to send a message to all destination nodes in one pass. Hence, the source node is not required to duplicate the message and to send the message to each of the destination nodes individually.

The design of the interconnection network is written in Verilog HDL (Hardware Description Language). All circuit designs are synthesized using the Design Analyzer (DA) tool of Synopsys, and the synthesized circuits are mapped into LSI Logic's LCA300K technology.

Unlike PaRC [4, 5], which is the network chip used as building block for constructing the interconnection network for the Monsoon [11] multiprocessor system, the network chip designed for the EARTH system uses only a simple buffer management scheme (one buffer per input port) and random routing to achieve high throughput of network links.

## 1.5 Contributions

We have designed a high throughput and low latency $4 \times 4$ crossbar switch as a building block for constructing a data network to be used by the EARTH system. As shown in Section 4.1, high throughput can be achieved by taking advantage of the $4 \times 4$ crossbar switch to perform random routing. The implementation cost of random routing is only a relatively small fraction of the total cost of the $4 \times 4$ crossbar switch (see Section 2.1.2). In our design of the $4 \times 4$ crossbar switch, no complicated buffer management scheme is required by random routing. Each input port uses a 4-bit shifted register contents as a hint on which output port is chosen by the incoming message. If the selected output is begin blocked by another message, the contents of the register is shifted thus the incoming message is able to choose another output port which may be free. Since the hardware cost for a 4-bit shifted register is low, this means that better hardware support for delivering messages should focus primarily on reducing the connection latency which is defined to be the time it takes to transfer the first byte of data from the input port to the output port. Furthermore, our $4 \times 4$ crossbar switch also supports broadcast facility. In broadcasting, a data set is sent from one node to all other nodes without replication of data taking place. Therefore, the bandwidth requirement can be substantially decreased for this kind of communication. Our $4 \times 4$ crossbar switch is designed to form an interconnection network for the EARTH multithreaded multiprocessor system, but its architecture is general enough to be used by other multiprocessor systems as well.

The simulation in Chapter 3 shows that the $4 \times 4$ crossbar switch has data transfer rate of 66.7 Mbytes/second/port, and the connection latency is equal to 4 clock cycles.

## 1.6 Synopsis

The remainder of the thesis is organized as follows: To begin, Chapter 2 describes the logical design of the $4 \times 4$ crossbar switch. The simulation results for the $4 \times 4$ crossbar switch are given in Chapter 3. Chapter 4 describes how the data network can be constructed by using $4 \times 4$ crossbar switches as building blocks. Topics related to routing issues are discussed as well. Furthermore. the comparisons with other switching chips are also given. In Chapter 5, the logical design of the network interface is presented. The simulation results for the network interface are given in Chapter 6. Finally, Chapter 7 summarizes the work carried out in this thesis. The complete source code files for the designs of the $4 \times 4$ crossbar switch and the network interface are given in Appendix A.

# Chapter 2

# Design of $4 \times 4$ Crossbar Switch

Since building a full crossbar network is costly, the basic architecture of the data network is a *multistage* [8] interconnection network. The data network consists of a hierarchy of $4 \times 4$ crossbar switches. The data network is designed to perform only one simple job: delivering messages. Other jobs such as duplicating messages, combining messages, or acknowledging delivery messages will not be performed by the data network. It just focuses on exchanging data between EARTH nodes. In this chapter, we describe the design of $4 \times 4$ crossbar switch in detail.

## 2.1 Basic Architecture of $4 \times 4$ Crossbar Switch

The $4 \times 4$ crossbar switch is a four input, four output message router that routes a message entering on an input port to an output port selected by the message header. Each port is 1-byte wide.

The $4 \times 4$ crossbar switch employs virtual cut-through routing technique [17]. If an output port is not already occupied by another message, the head of the incoming message is advanced into this output port as soon as the head of the message is received and decoded.

The message is then moved forward through this established link until the link is closed by the tail of the message. If the selected output port is blocked by another message, the $4 \times 4$ crossbar switch will automatically reroute the message around the busy output port. This automatic routing around a busy port is known as random routing[1] [6]. If random routing fails to find an unoccupied output port, the incoming message is blocked and stored into the memory buffer at the input port. An example of random routing is given in Section 4.1. The $4 \times 4$ crossbar switch also supports broadcast routing. If the head of the incoming message is decoded to be a broadcast message, the message is then routed to the multiple output ports.

In the $4 \times 4$ crossbar switch a total interconnection of inputs and outputs is implemented. As a result, each input can be directly connected to all outputs. All these connections are independent of each other and therefore can function simultaneously without interfering each other. The $4 \times 4$ crossbar switch has four independent data ports and is therefore capable of performing four simultaneous communications.

The $4 \times 4$ crossbar switch has four first-in first-out (FIFO) memory buffers, four control units, a crossbar and an arbiter as is shown in Figure 2.1. The control units and the arbiter are driven by a single central clock signal. The main reason for the $4 \times 4$ crossbar switch having only one FIFO buffer per input port is that this buffering scheme allows for a high speed design. If multiple buffers are used for each of the input ports, each output port is required to connect to multiple loads. The impedance of the connection may become unstable. This leads to the degradation of the data transfer rate because the data transfer is encumbered by the need to cater for different numbers of connections and variable loads.

Each FIFO has its own clock signals to clock data into and out of the memory buffer.

[1]There are limitations on the random routing which are dependent on the interconnection topology.

The crossbar is pure combinational logic thus it is not driven by any clock signal.

### 2.1.1 FIFO

The main function of the four FIFO's is to provide data buffering for the 4 × 4 crossbar switch. If an incoming message is blocked due to the selected output port being already occupied, the incoming message is stored in the FIFO until the selected output port is free. Each FIFO is controlled by separate clocked read and write signals. The FIFO is designed to be a circular queue and its depth is 32 bytes. The minimum length of a message is 12 bytes long[2]. Therefore, each FIFO is able to store more than two short messages. As soon as the first message is sent off the switch, the second message is ready to make a connection to a output port immediately. The reason for doing this is to keep the output ports as busy as possible. If any output port is idle, it means part of the bandwidth of the switch is wasted. It is true that better bandwidth can be obtained by using a FIFO which is more than 32 bytes in depth. However, a larger FIFO also means higher hardware cost. The write operation is controlled by a write clock (WCLK). A byte of data is written into the FIFO on the rising edge of the WCLK signal. The read operation is controlled in a similar manner by a read clock (RCLK). A byte of data is read from the FIFO on the rising edge of the RCLK signal.

FIFO provides two status flags: AFULL and EMPTY. The status flags are responsible for controlling data flow. When the AFULL flag is HIGH, it indicates that the FIFO is full. Hence, no additional data can be written into the FIFO otherwise the FIFO will overflow. When the EMPTY flag is HIGH, it indicates that the FIFO is empty. Hence, no data can be read from the FIFO until the EMPTY flag becomes LOW. Both of the status flags

[2]Message format is described in Section 4.2.3

are synchronous. The transition of AFULL flag from LOW to HIGH is according to the WCLK clock signal while the transition from HIGH to LOW is according to the RCLK clock signal. Similarly, the transition of EMPTY flag from LOW to HIGH is according to the RCLK clock signal while the transition from HIGH to LOW is according to the WCLK clock signal. Both WCLK and RCLK are pulse signals with respect to the master clock, and their positive edges never overlap each other. The signal waveforms for WCLK and RCLK are shown in Figure 2.2. The synchronous flag architecture assures that the flags maintain their status for at least half clock cycle. Figure 2.3 shows the logic block diagram of the FIFO. The FIFO is composed of Reset Logic, Write Pointer Logic, Read Pointer Logic, Flags Logic, and Register File.

**Reset Logic:**

The FIFO has four 5-bit registers ($R_{tail}$, $R_{tp}$, $R_{head}$, and $R_{hp}$) to provide the input and output addresses for the Register File during read/write operations. On rising edge of the RESET signal. all four registers are reset to zero.

**Write Pointer Logic:**

As shown in Figure 2.4a. the Write Pointer Logic produces two 5-bit address signals: TAIL and TP. On the rising edge of WCLK, the result computed by the 5-bit ADDER is loaded into $R_{tail}$ to give TAIL. This is equivalent to incrementing TAIL by one on every positive edge of WCLK. On the other hand, the contents of $R_{tail}$ is loaded into $R_{tp}$ on the falling edge of WCLK to produce TP. TAIL goes to the Flags Logic whereas TP goes to Register File. Using two separate address signals to represent the tail position of FIFO makes sure that input data will be written into the right register within the Register File.

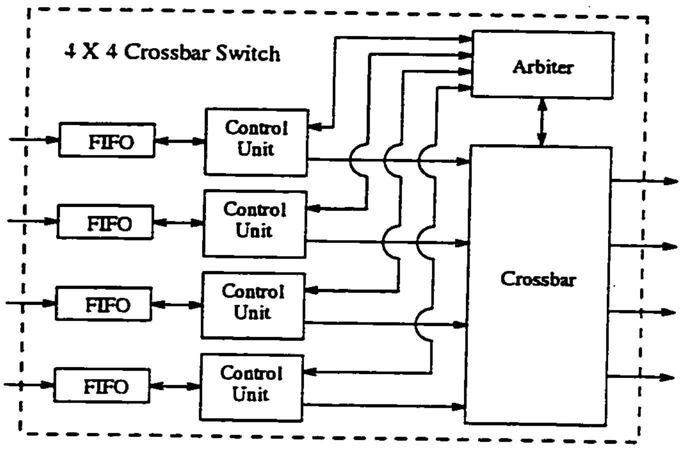


Figure 2.1: 4 × 4 Crossbar Switch Block Diagram

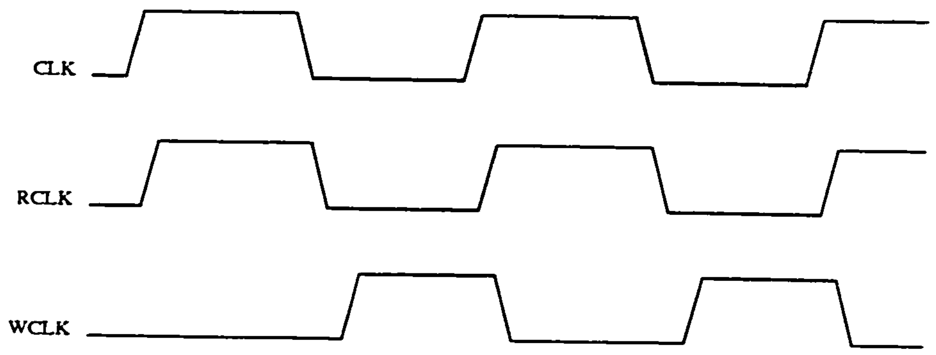


Figure 2.2: Signal Waveforms for RCLK and WCLK

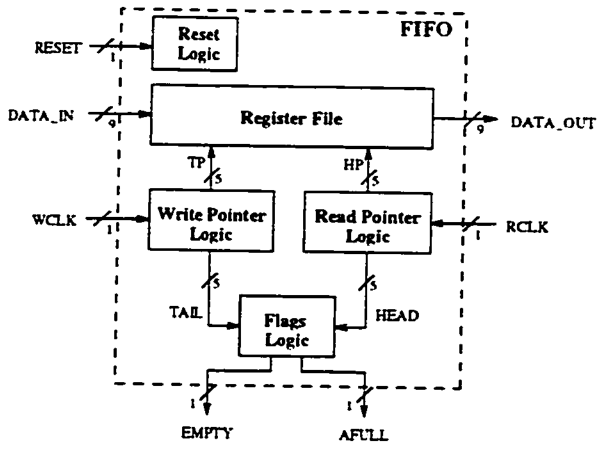


Figure 2.3: FIFO Block Diagram

Note that data is written into the Register File on the rising edge of WCLK, using TAIL as the address signal for the Register File may violate the address signal hold time because TAIL also changes its value on the rising edge of WCLK.

**Read Pointer Logic:**

Similar to Write Pointer Logic, the Read Pointer Logic produces two 5-bit address signals: HEAD and HP (Figure 2.4b). On the rising edge of RCLK, the result computed by the 5-bit ADDER is loaded into $R_{head}$ to produce HEAD. This is equivalent to incrementing HEAD by one on every positive edge of RCLK. On the other hand, the contents of $R_{head}$ is loaded into $R_{hp}$ on the falling edge of RCLK to produce HP. HEAD goes to the Flags Logic whereas HP goes to Register File. Using two separate address signals to represent the head position of FIFO makes sure that output data will read from the right register within the Register File. Note that data is read from the Register File on the rising edge of RCLK. using HEAD as the address signal for the Register File may violate the address signal hold time because HEAD also changes its value on the rising edge of RCLK.

**Flags Logic:**

The Flags Logic provides two status flags: AFULL and EMPTY. To generate EMPTY is straight forward. we just need to compare HEAD and TAIL to see whether they are equal or not. When HEAD and TAIL are equal, EMPTY is set to HIGH. AFULL is set to HIGH when the condition is met: $HEAD = TAIL + 1$. In order to determine AFULL. we first add 1 to TAIL and compare the result with HEAD. The detailed implementation of the Flags Logic is given in Figure 2.5.

**Register File:**

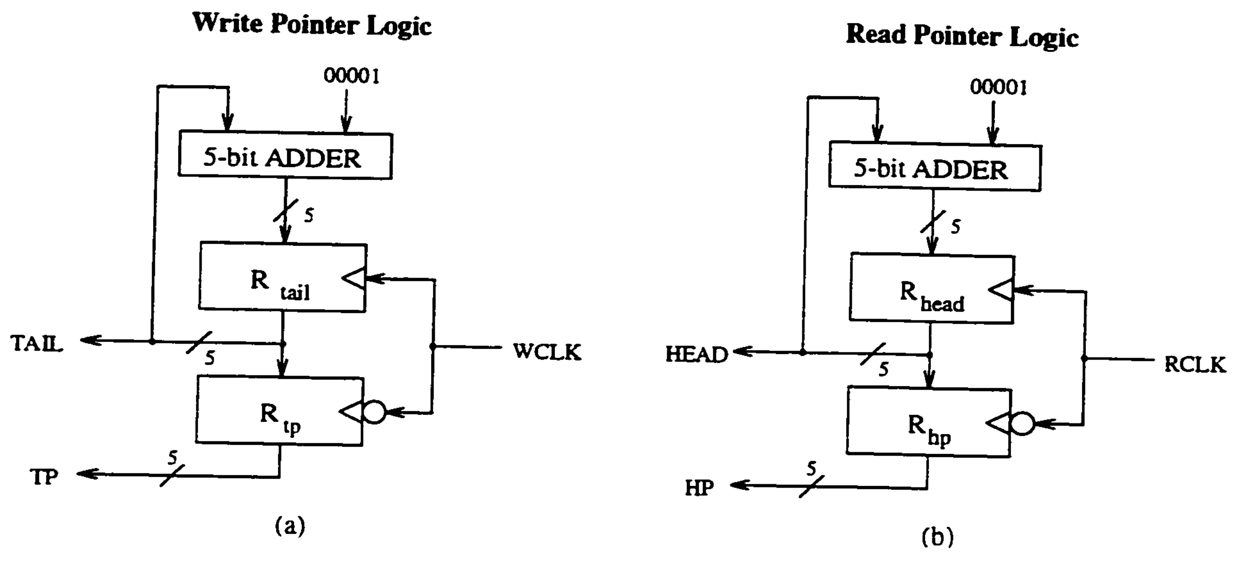


Figure 2.4: Write/Read Pointer Logic Design

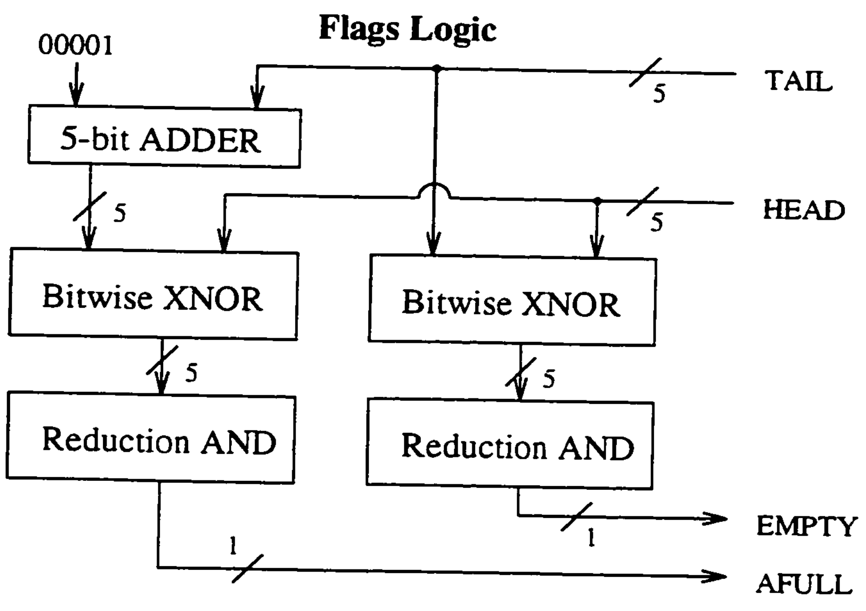


Figure 2.5: Flags Logic Design

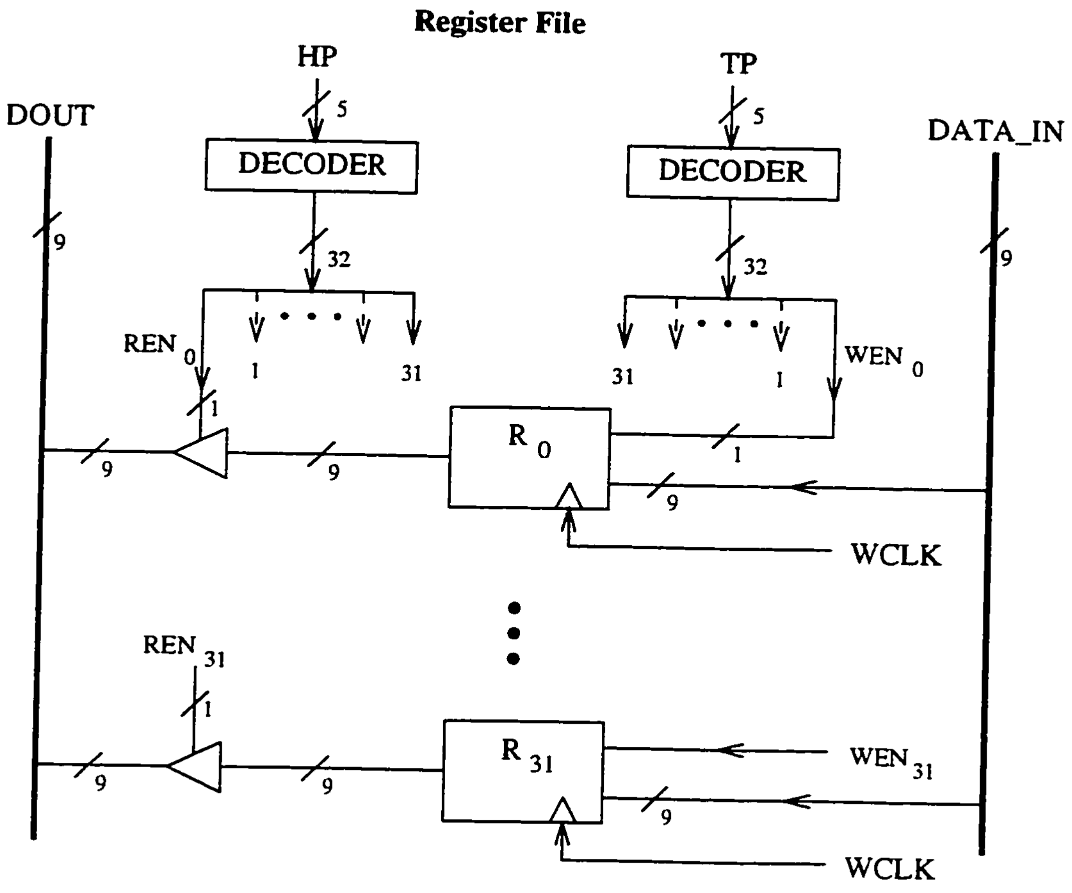


Figure 2.6: Register File Logic Design

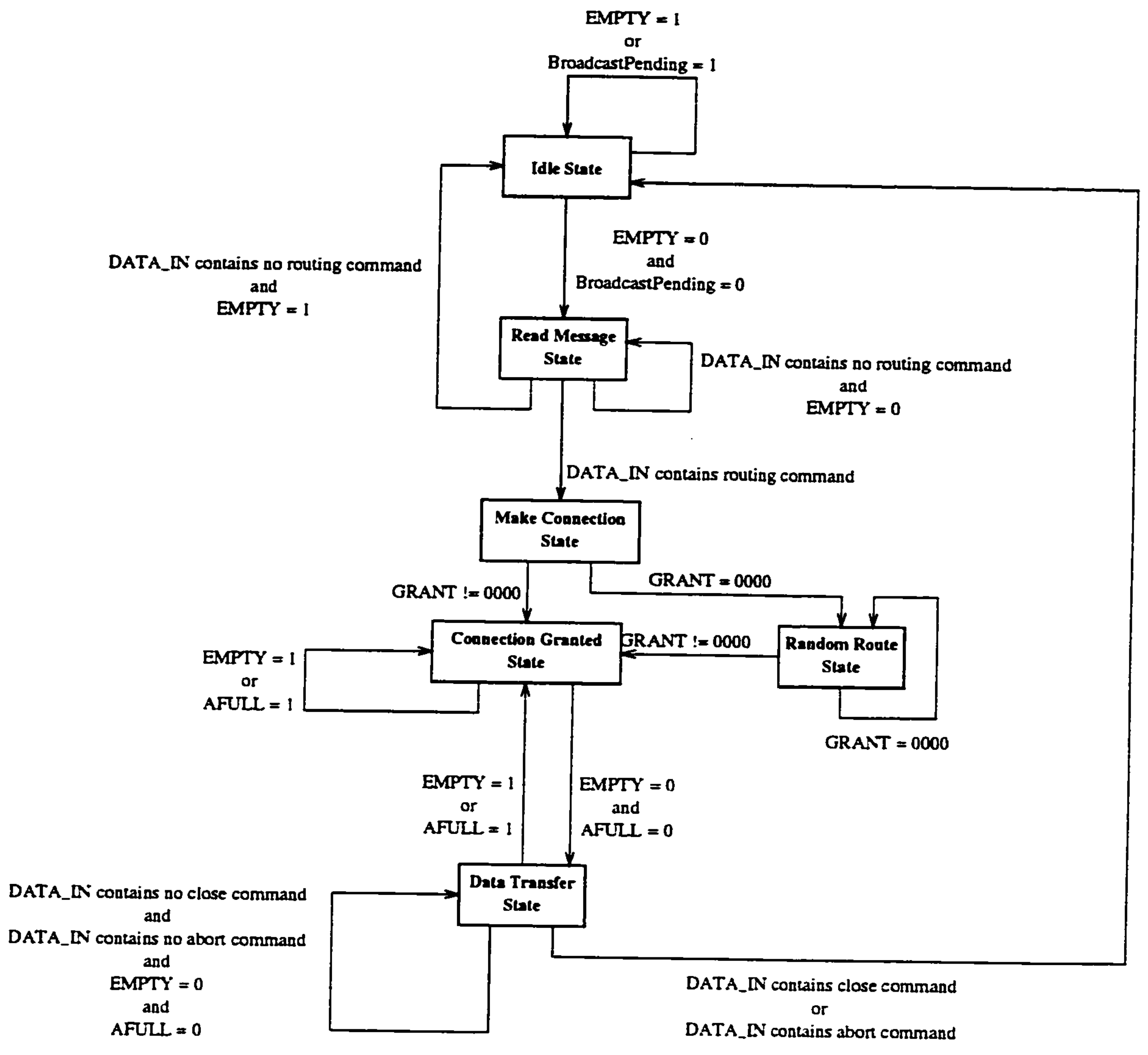
Register File consists of 32 9-bit registers ($R_0 \cdots R_{31}$). It has two input address signals, HP and TP. Both HP and TP are decoded into 32-bit control signal (REN and WEN respectively) which can give direct control to the Register File. REN serves as the read enable signal whereas WEN serves as the write enable signal for the Register File. REN is used to control the tristate buffers so that the data stored in the corresponding register is latched into the output data bus, DOUT. WEN is used to control the corresponding register to latch the content from the input data bus, DATA_IN. The design of the Register File is illustrated in Figure 2.6.

### 2.1.2 Control Unit

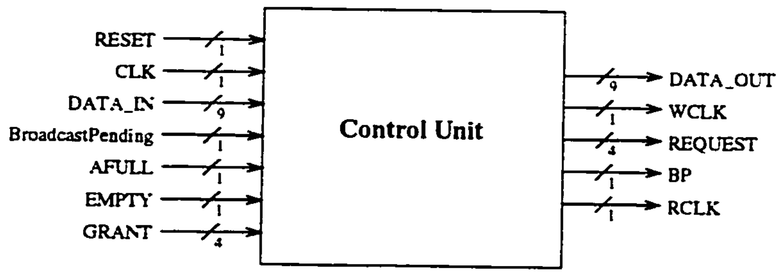
The main functions of the control unit are:

1. To monitor the incoming data appearing at the output bus of the FIFO.
2. To launch a request of connection if the head of a message is a routing command.
3. To perform random routing if the routing command is random routing type and the selected output port is busy.
4. To schedule data transfer after the request of connection is granted.
5. To close the connection after the tail of message is detected.

The control unit is implemented as a finite state machine, and it is clocked by a global clock signal (CLK). The control unit undergoes six distinct states when it is activated by the head of an incoming message until the transfer of the message is brought to completion. The change of states occurs only on the rising edge of CLK and some conditions must be met. Figure 2.7 depicts the six distinct states of the control logic, and its symbolic view.



(a)



(b)

Figure 2.7: Control Unit: (a) State Diagram (b) Symbolic View

On the rising edge of RESET signal, the control unit is reset to **Idle State**. In **Idle State**, the control unit idles. It proceeds to **Read Message State** if and only if both of the input signals EMPTY and BroadcastPending stay LOW. In **Read Message State**, the pulse signal RCLK is generated. On the rising edge of RCLK, a byte of data from the corresponding FIFO is latched into the 9-bit register $R_{data\_out}$. The content of the register is represented by the signal, DATA_OUT, which goes to the crossbar. The control unit proceeds to **Make Connection State** if and only if the contents of $R_{data\_out}$ is decoded to be a routing command. The output signal BP is set to HIGH if the routing command is broadcast type. In **Make Connection State**, the request of connection is made. Depending on the type of routing command, the request may ask for connection to one output port, two output ports or four output ports. The control unit proceeds to **Connection Granted State** if and only if the request of connection is granted, otherwise the control unit proceeds to **Random Route State** from **Make Connection State**. In **Random Route State**, control unit performs random routing if the the head of a message is a random routing type. The control unit selects new target output port(s) by changing the request so that an incoming message is routed around the busy port(s). If the new request of connection is granted, the control unit proceeds to **Connection Granted State**. In **Connection Granted State**, two input signal, EMPTY and AFULL, are examined. If either EMPTY or AFULL is HIGH, the control unit remains in **Connection Granted State**. If neither EMPTY nor AFULL is HIGH, the control unit proceeds to **Data Transfer State**. In **Data Transfer State**, the RCLK pulse is generated during the first half of clock cycle. On the rising edge of RCLK, a byte of data from the corresponding FIFO is latched into the 9-bit register $R_{data\_out}$. The content of the register is represented by the signal, DATA_OUT, which goes to the crossbar. During the second half of clock cycle, the

WCLK pulse is generated. If the byte of data read from the FIFO is either a close or an abort command, the control unit returns to **Idle State** or the idle condition.

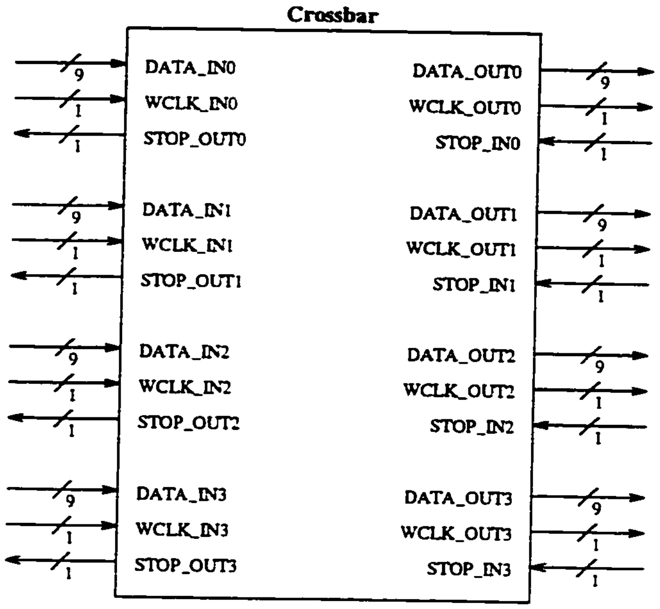
### 2.1.3 Crossbar

The symbol view of the crossbar is shown in Figure 2.8a. The crossbar provides a logical connection between an input port and an output port. It consists of four horizontal buses (rows) and four vertical buses (columns). A horizontal bus intersects a vertical bus at a *crosspoint*. Each of the horizontal bus and vertical bus is 9 bits wide. The four pairs of input data bus and output data bus are interconnected together as shown in Figure 2.8b.

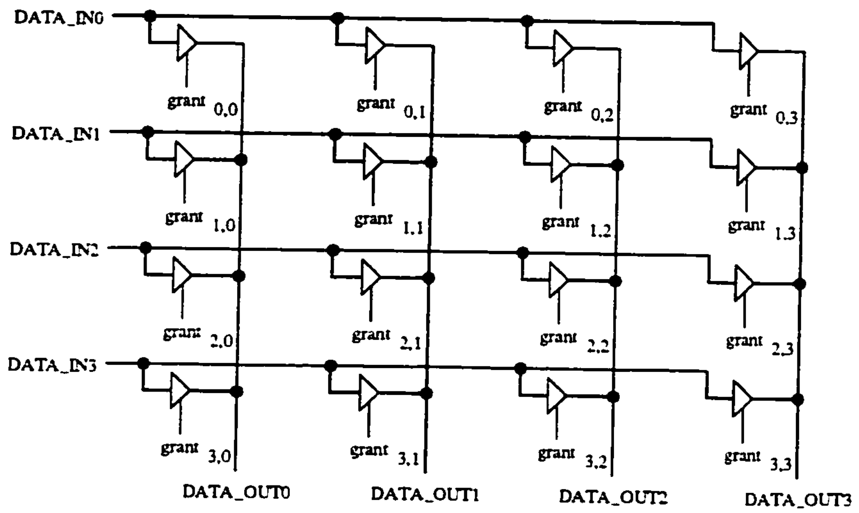
As shown in Figure 2.8a. each port has two data flow signals. For an input port, the data flow signals are (WCLK_IN$i$ and STOP_OUT$i$). For an output port, the data flow signals are (WCLK_OUT$j$ and STOP_IN$j$). The values of WCLK_OUT$j$ and STOP_OUT$i$ are determined by the following boolean equations:

$$\begin{aligned}\text{WCLK\_OUT}j &= (\text{WCLK\_IN0} \bullet \text{GRANT}_{0,j}) + (\text{WCLK\_IN1} \bullet \text{GRANT}_{1,j}) + \\ &\quad (\text{WCLK\_IN2} \bullet \text{GRANT}_{2,j}) + (\text{WCLK\_IN3} \bullet \text{GRANT}_{3,j}) \\ \text{STOP\_OUT}i &= (\,(\text{STOP\_IN0} \bullet \text{GRANT}_{i,0}) + (\text{STOP\_IN1} \bullet \text{GRANT}_{i,1}) + \\ &\quad (\text{STOP\_IN2} \bullet \text{GRANT}_{i,2}) + (\text{STOP\_IN3} \bullet \text{GRANT}_{i,3})\,) \bullet \\ &\quad (\text{GRANT}_{i,0} + \text{GRANT}_{i,1} + \text{GRANT}_{i,2} + \text{GRANT}_{i,3})\end{aligned}$$

The first boolean equation means that if the physical link between input port $i$ and output port $j$ is established, the data flow signal WCLK_OUT$_j$ will copy the value of WCLK_IN from input port $i$. The second boolean equation means that if input port $i$ is connected to one or more output ports, the data flow signal STOP_OUT$_i$ becomes HIGH if and only if any of the STOP_IN signal from the connected output port(s) is HIGH.



(a)



grant $_{i,j}$ — a control signal to the tristate buffer where i represents the row number and j represents the column number in the crossbar.

(b)

Figure 2.8: Crossbar: (a) Symbolic View (b) Data Bus Logic

A logical connection is made by a routing command and maintained by a control signal GRANT$_{i,j}$ at each crosspoint, where $i$ is the input port number and $j$ is the output port number ($0 \leq i \leq 3, 0 \leq j \leq 3$). Upon receipt of the close or abort command, the logical connection will be closed.

At each crosspoint, there is a tristate buffer which can be activated to form a connection between the corresponding input port and output port. When GRANT$_{i,j}$ is HIGH, the connection between input port $i$ and output port $j$ is established. Whatever the value appears at DATA_IN$i$ will also appears at DATA_OUT$j$.

### 2.1.4 Arbiter

Although the crossbar described in Section 2.1.3 is capable of supporting four logical connections simultaneously, a problem arises when more than one input port want to connect to a same output port. This conflicting demand for resources introduces delay to the interconnection network and reduces the network performance. Hence the $4 \times 4$ crossbar switch must have an arbiter to resolve the conflicts and provide efficient and fair scheduling of these resources.

The arbiter consists of 16 REQUEST$_{i,j}$ input signals, one per crosspoint. A REQUEST$_{i,j}$ signal is asserted (HIGH) when the use of the particular crosspoint is needed. The arbiter produces 16 GRANT$_{i,j}$ output signals, one per crosspoint. These output signals indicate which crosspoint requests have been granted.

In order for the arbiter to provide fair arbitration services, the *Wrapped Wave Front Arbiter* (WWFA) technique [21] is adopted in our arbiter design. The WWFA uses a wave front to select which of the four arbitration calls have top priority during the arbitration process. The wave front either moves diagonally from top left to the bottom right corner of the arbiter or moves vertically from the top row to the bottom row of the arbiter. The direction of movement depends on what mode the arbiter is operating in. If the arbiter is operating in non-broadcast mode, the wave front moves diagonally. If the arbiter is operating in broadcast mode, the wave front moves vertically. All the wave movements are controlled by a 4-bit circular shift register.

When the $4 \times 4$ crossbar switch is operating in the non-broadcast mode, the four top priority arbitration cells selected by the *diagonal* wave front are guaranteed to be located in different rows and columns. This maximizes the probability that multiple arbitration cells will win the arbitration. Figure 2.9 shows the arbitration process where the *diagonal* wave front consists of cells (0,1), (1,0), (3,2) and (2,3).

When the $4 \times 4$ crossbar switch is operating in the broadcast mode, the four top priority arbitration cells selected by the *horizontal* wave front are guaranteed to be located in the same row. This maximizes the probability that multiple arbitration cells in the same row will win the arbitration. Figure 2.10 shows the arbitration process where the *horizontal* wave front consists of cells (1,0), (1,1), (1,2) and (1,3).

The logic block diagram of the arbiter is illustrated in Figure 2.11. The arbiter consists of a 4-bit circular shift register, a wave front generator (WFG) and a conflict resolver (CR). On the rising edge of the RESET signal, the shift register is initialized to have value 0001 in binary. Then the shift register is shifted to the right by one bit for every clock cycle. The 4-bit output of the shift register is used by the WFG to generate the wave front. As mentioned above, two types of wave front can be generated depending on the value of input signal, BroadcastPending. If BroadcastPending is HIGH, it indicates that the $4 \times 4$ crossbar switch is operating in a broadcast mode. The horizontal wave front is generated. If the BroadcastPending is LOW, it indicates that the $4 \times 4$ crossbar switch is operating in a

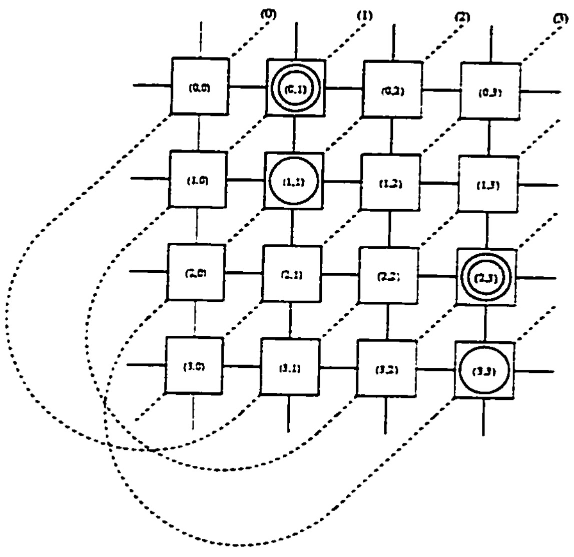


Figure 2.9: Arbiter with diagonal wrapped wave front. The diagonal wave for which (0.1). (1,0), (3,2) and (2,3) has the top priority. Double circles indicate the granted requests and the single circles indicate the denied requests

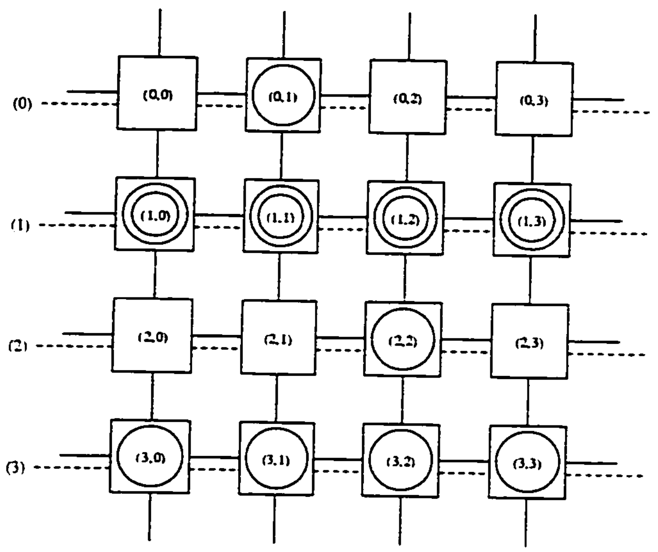


Figure 2.10: Arbiter with horizontal wrapped wave front. The horizontal wave for which (1.0). (1.1), (1.2) and (1.3) has the top priority. Double circles indicate the granted requests and the single circles indicate the denied requests

non-broadcast mode. The diagonal wave front is generated.

CR is used to resolve the conflicts when more than one input port make the requests for the same output port. The CR performs the arbitration process and those arbitration cells with top priority according to the wave front generated by the WFG will win the arbitration. In case there is only one request signal in a column, the request is granted immediately regardless of the wave front.

We have completed the design of the $4 \times 4$ crossbar switch. The detailed block diagram of the $4 \times 4$ crossbar switch with all signal connections is shown in Figure 2.12.

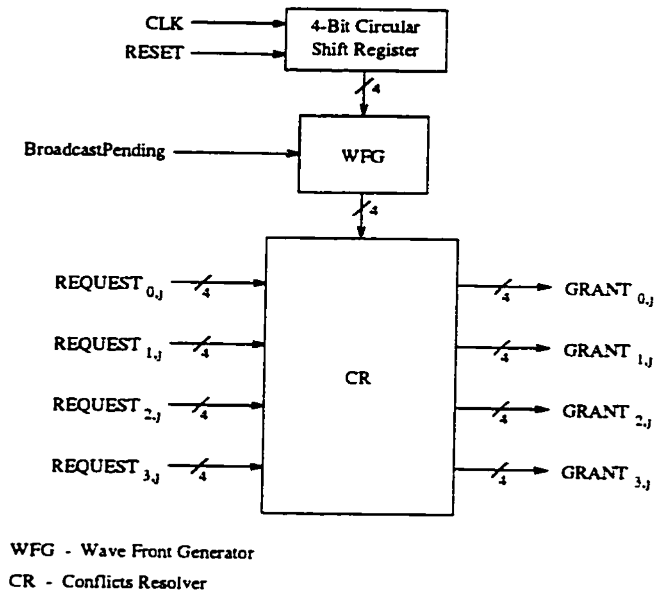


Figure 2.11: Arbiter Block Diagram

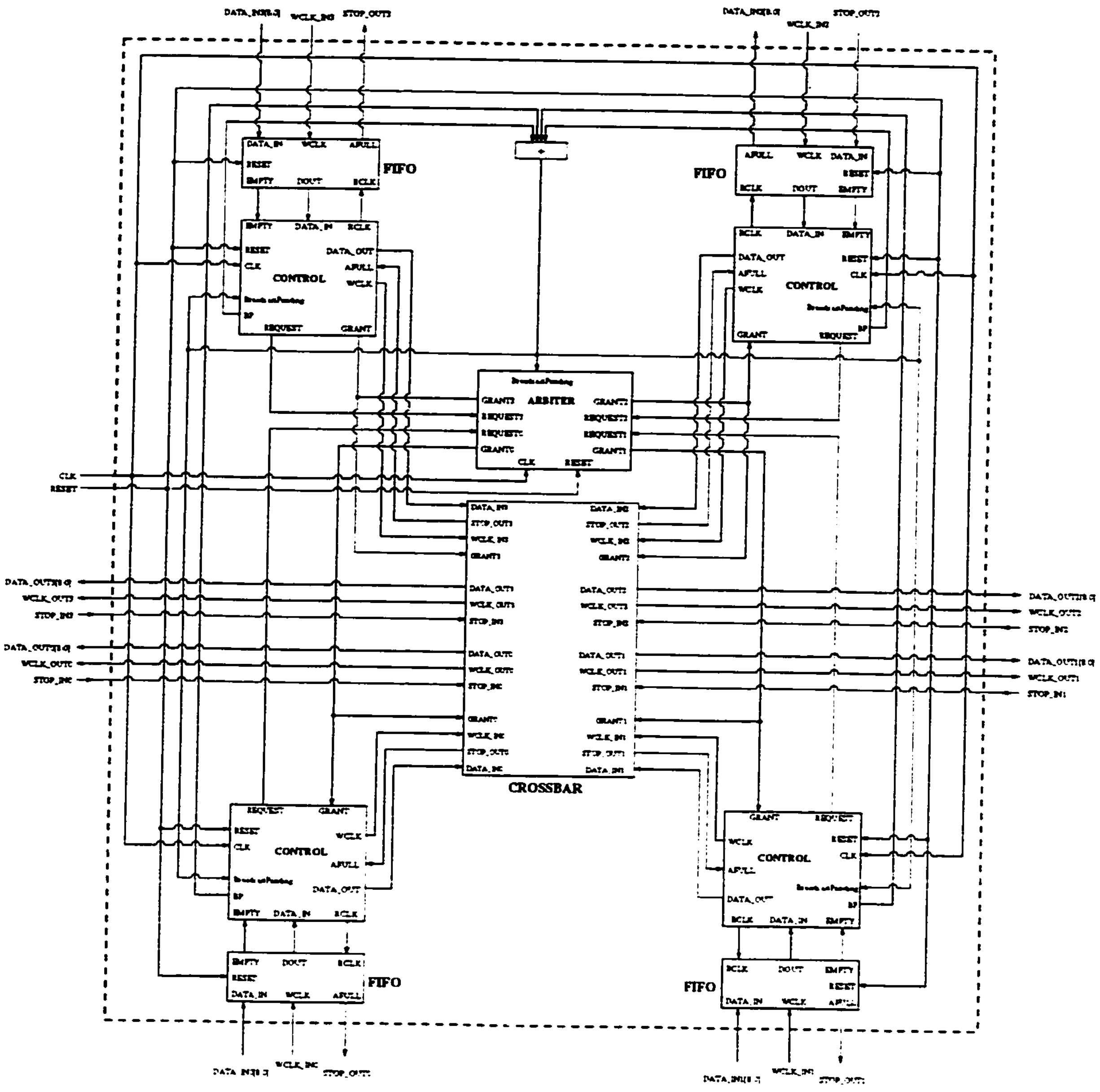


Figure 2.12: Detailed 4 × 4 Crossbar Switch Block Diagram

# Chapter 3

# Simulation Results of $4 \times 4$ Crossbar Switch

We have described the design of the $4 \times 4$ crossbar switch in Chapter 2. The design of the $4 \times 4$ crossbar switch has been synthesized using the Design Analyzer (DA) tool of Synopsys. and the synthesized circuit has been mapped into LSI Logic's LCA300K technology. In this chapter. the simulation results of the $4 \times 4$ crossbar switch will be presented.

## 3.1 Maximum Clock Frequency

In order to determine the maximum clock frequency for the $4 \times 4$ crossbar switch. we need to examine the critical path of the circuit. The critical path of the $4 \times 4$ crossbar switch is found to be located within a FIFO and it is highlighted in Figure 3.1. The critical path exists between the read clock (RCLK) and the data output (DOUT). The waveforms of the signals are illustrated in Figure 3.2, and the timing parameters of the waveforms are given in Table 3.1.

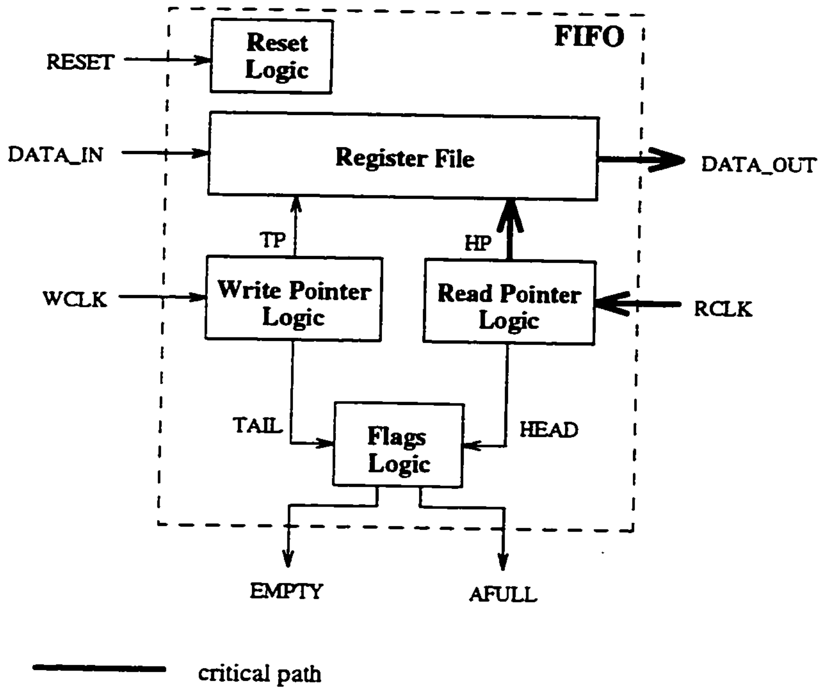


Figure 3.1: Critical Path of 4 × 4 Crossbar Switch

| Parameter | Description | Max. | Min. | Unit |
|---|---|---|---|---|
| $t_{CRN}$ | Delay from ↓CLK to ↓RCLK for negative edge | 1 | | ns |
| $t_{CRP}$ | Delay from ↑CLK to ↑RCLK | 3 | 1 | ns |
| $t_{DHP}$ | Delay between RCLK and HP | 1 | | ns |
| $t_{DTD}$ | Data access time | 5 | | ns |
| $t_{DTS}$ | Data setup time | | 3 | ns |
| $t_{RKH}$ | RCLK pulse width | | 2 | ns |

Table 3.1: Measurement of Timing Parameters Along Critical Path

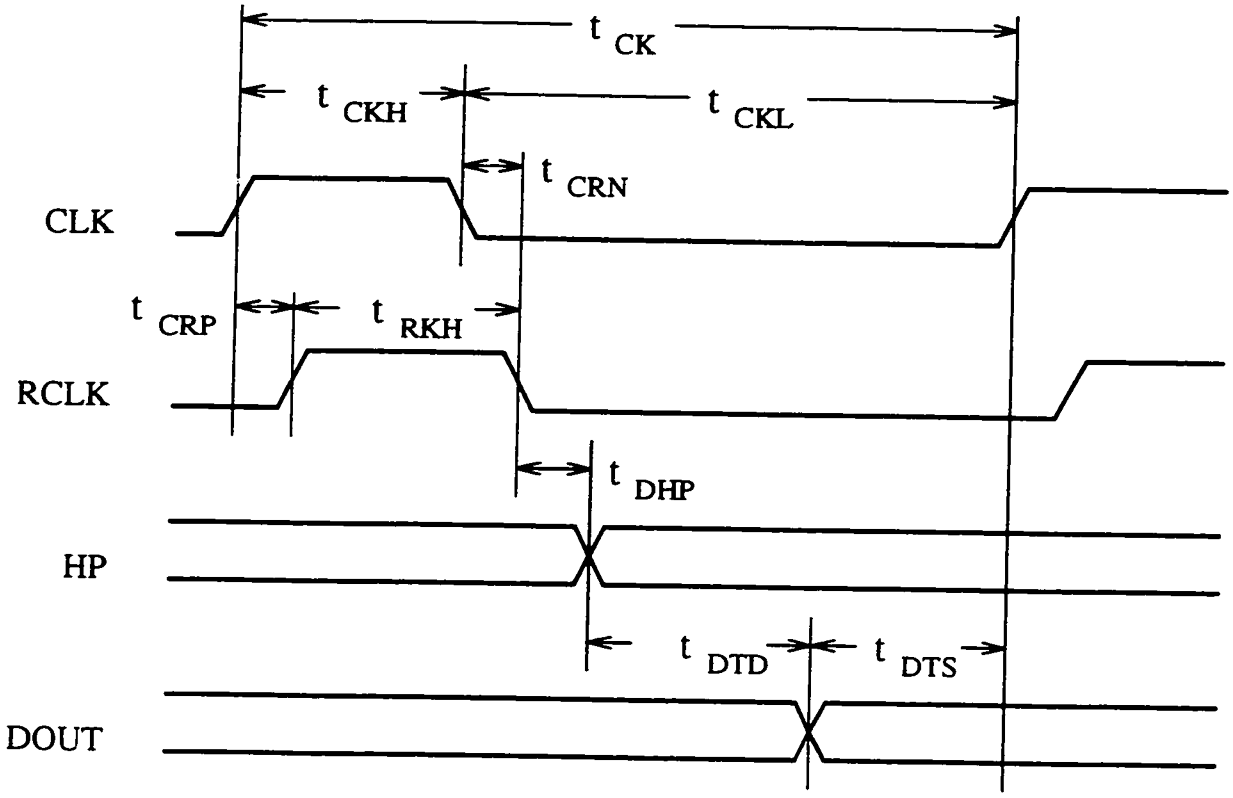


Figure 3.2: Signal Waveforms for the Critical Path

As described in Section 2.1.2, RCLK is a pulse generated by the corresponding control unit in state two and state six during the first half of clock cycle. When CLK changes from LOW to HIGH, RCLK will go HIGH after 3 ns due to the delay. In order for the registers ($R_{\mathbf{head}}$ and $R_{\mathbf{hp}}$) within Read Pointer Logic to be loaded properly, RCLK must stay HIGH for at least 2 ns. That means the minimum pulse width of RCLK is 2 ns. The minimum time for CLK to stay HIGH, $t_{CKH}$, is computed as:

$$\begin{aligned} t_{CKH} &= t_{CRP(max)} + t_{RKH} \\ &= 3\ ns + 2\ ns \\ &= 5\ ns \end{aligned}$$

When CLK changes from HIGH to LOW, RCLK will go LOW after 1 ns due to the delay. After the falling edge of RCLK, HP changes value after 1 ns delay. Then the register file will decode HP and put the data on DOUT. However, it takes 5 ns for DOUT to have correct value after HP is changed due to the internal delay of the register file. The minimum setup time for DOUT is 3 ns. If the setup time for DOUT is less than 3 ns, the corresponding control unit will not be able to latch the correct data from DOUT. The minimum time for CLK to stay LOW, $t_{CKL}$, is computed as:

$$\begin{aligned} t_{CKL} &= t_{CRN} + t_{DHP} + t_{DTD} + t_{DTS} \\ &= 1\ ns + 1\ ns + 5\ ns + 3\ ns \\ &= 10\ ns \end{aligned}$$

Hence, the minimum clock cycle time, $t_{CK}$, is calculated as:

$$\begin{aligned} t_{CK} &= t_{CKH} + t_{CKL} \\ &= 5\ ns + 10\ ns \\ &= 15\ ns \end{aligned}$$

Finally, the maximum clock frequency $f_{max}$ is obtained as:

$$\begin{aligned} f_{max} &= \frac{1}{t_{CK}} \\ &= \frac{1}{15\,ns} \\ &= 66.7\,MHz \end{aligned}$$

From the above result, the maximum bandwidth for the 4 × 4 crossbar switch is equal to 66.7 Mbytes/second/port.

Note that data access time for the FIFO consumes exactly one quarter of the minimum clock cycle time. Therefore, the overall performance of the 4 × 4 crossbar switch depends heavily on how fast the data can be retrieved from the FIFO. In order to minimize the data access time, we use tristate buffers instead of using a multiplexer to produce data on the output data bus in our FIFO design since a multiplexer not only requires more combinational logic to implement, but also introduces longer delays in our design.

## 3.2 Pin Numbers and Chip Area

The design of a 4 × 4 crossbar switch is mapped into the LSI Logic's 0.6-Micron LCA300K compacted array technology. The total number of pins used by the 4 × 4 crossbar switch is 90. The total area[1] for the 4 × 4 crossbar switch is summarized in Table 3.2.

## 3.3 Switching Waveforms

Figure 3.3 shows the switching waveforms for the 4 × 4 crossbar switch. Note that the port numbers in the signal names are dropped since the the timing parameters are measured for the worst case during simulations, and they are applicable to all ports. The measured values for the timing parameters are summarized in Table 3.3.

[1] A gate is defined as four transistors, the equivalent of one NAND gate.

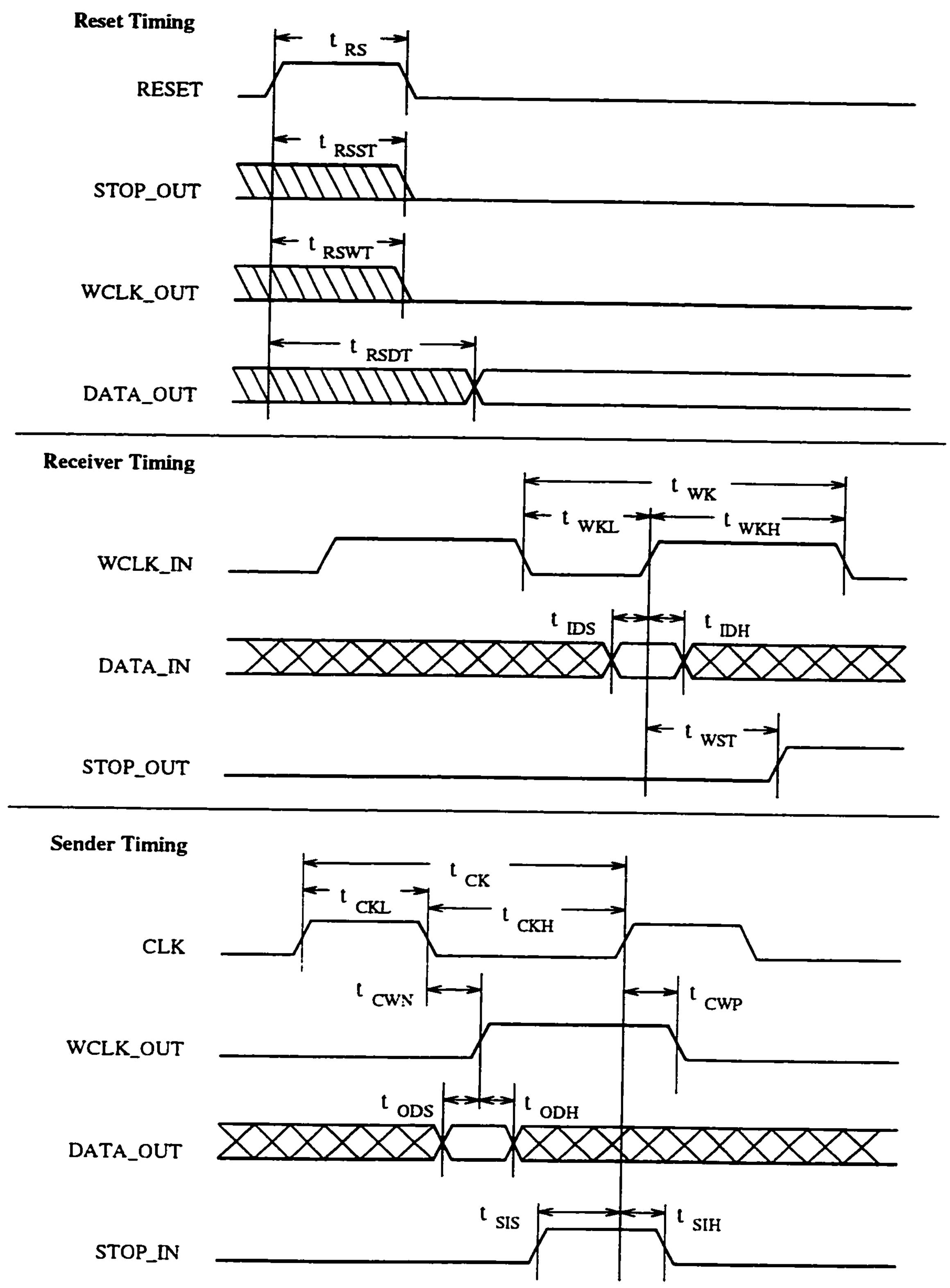


Figure 3.3: Switching Waveforms for 4 × 4 Crossbar Switch

## 3.4 Connection Latency

Connection latency is defined as the time from the point that the head of a message reaches an input port till the first byte of data reaches an output port. For connection latency with no route contention, the first byte of data takes up to 4 clock periods to reaches the output port.

| **Type** | **Area (gate equivalents)** |
|---|---|
| Combinational | 2056 |
| Non-combinational | 17056 |
| Wire Interconnection | 6837 |
| Total | 25949 |

Table 3.2: Area for 4 × 4 Crossbar Switch

| Parameter | Description | Max. | Min. | Unit |
|---|---|---|---|---|
| $t_{RS}$ | RESET pulse width | | 2 | ns |
| $t_{RSST}$ | RESET to STOP_OUT and output time | 2 | | ns |
| $t_{RSWT}$ | RESET to WCLK_OUT and output time | 3 | | ns |
| $t_{RSDT}$ | RESET to DATA_OUT and output time | 6 | | ns |
| $t_{WK}$ | WCLK_IN cycle time | | 15 | ns |
| $t_{WKH}$ | WCLK_IN HIGH time | | 10 | ns |
| $t_{WKL}$ | WCLK_IN LOW time | | 5 | ns |
| $t_{IDS}$ | DATA_IN set-up time | | 1 | ns |
| $t_{IDH}$ | DATA_IN hold time | | 1 | ns |
| $t_{WST}$ | WCLK_IN to STOP_OUT | 4 | | ns |
| $t_{CK}$ | CLK cycle time | | 15 | ns |
| $t_{CKH}$ | CLK HIGH time | | 5 | ns |
| $t_{CKL}$ | CLK LOW time | | 10 | ns |
| $t_{CWN}$ | Delay from ↓CLK to ↑WCLK_OUT | 1 | | ns |
| $t_{CWP}$ | Delay from ↑CLK to ↓WCLK_OUT | 1 | | ns |
| $t_{ODS}$ | DATA_OUT set-up time | | 1 | ns |
| $t_{ODH}$ | DATA_OUT hold time | | 1 | ns |
| $t_{SIS}$ | STOP_IN set-up time | | 3 | ns |
| $t_{SIH}$ | STOP_IN hold time | | 1 | ns |

Table 3.3: Switching Characteristics of $4 \times 4$ Crossbar Switch

# Chapter 4

# Data Network

In this chapter, we discuss how a data network with more than four EARTH nodes can be built by using $4 \times 4$ crossbar switches as basic elements. Then, a routing scheme for such data networks will be discussed.

## 4.1 Building a Data Network

A single $4 \times 4$ crossbar switch can connect up to four EARTH nodes together. Since a $4 \times 4$ crossbar switch described in Chapter 2 is designed to be scalable, a larger EARTH system with more than four nodes can be implemented by connecting various $4 \times 4$ crossbar switches. There are many different ways to connect various $4 \times 4$ crossbar switches to form a larger data network. One possible way is to construct the data network into multistage topology. For example, Figure 4.1 illustrates the data network for the EARTH systems with 8 nodes ($N = 8$). The data network shown in Figure 4.1 is a modified version of *indirect binary n-cube network* [19]. The indirect binary n-cube network originally consists of $\log_2 N$ stages of $2 \times 2$ switching elements but it has been modified to function with $\log_2 \frac{N}{2}$ stages of $4 \times 4$ switching elements.

The multistage topology of the data network supports the following features:

## Virtual Cut-Through Routing

The multistage data network employs virtual cut-through routing. Virtual cut-through routing allows the head of a message to be transferred out of a $4 \times 4$ crossbar switch before the end of the message has been received. With virtual cut-through routing, the message can be transferred before the entire path (from source node to destined node) is physically established.

## Scalable:

A larger data network can be built by using additional crossbar switches. However, one basic requirement for the data network still must be met: the data network must provide full access capability, which means that any input node of the network should be able to access any output node in one pass through the network. With $4 \times 4$ crossbar switches as basic elements, the multistage data network with 16, 32, 64 or even more processing nodes is possible.

## Random Routing:

Random routing allows a message to be routed to another output port of the switch so that a busy output port can be avoided. In order for the random routing to work properly, the data network topology is chosen carefully so that the following requirements for random routing are satisfied:

- Multiple paths exist between any two nodes.

- The distance between any two nodes in the data network includes the same number of $4 \times 4$ crossbar switches.

Figure 4.2 demonstrates how random routing is done from node 0 to node 6 in the 8-node data network. The third output port of the crossbar switch (shown shaded) is already occupied by another message while the fourth output port is free at the time. Therefore. the incoming message from node 0 is routed to the fourth output instead.

**Broadcast:**

The multistage data network provides a broadcast facility in which a node is able to send a block of data to all the destination nodes. This is supported by the use of a broadcast command in the head of the message.

**Full-Duplex Connection:**

Each input data port can directly connect to all output data ports in the multistage data network. All logical connections are independent of each other and thus can be established simultaneously without interfering each other. One input data port and one output data port can be linked together to form a full-duplex connection.

**Data Rate:**

The maximum clock rate[1] is defined to be 66.7 MHz. Each $4 \times 4$ crossbar switch in the multistage data network is capable of transferring a byte of data every clock cycle. Therefore. the data network has a bandwidth up to 66.7 Mbytes/second/port.

[1]Based on the simulation results in Chapter 3.

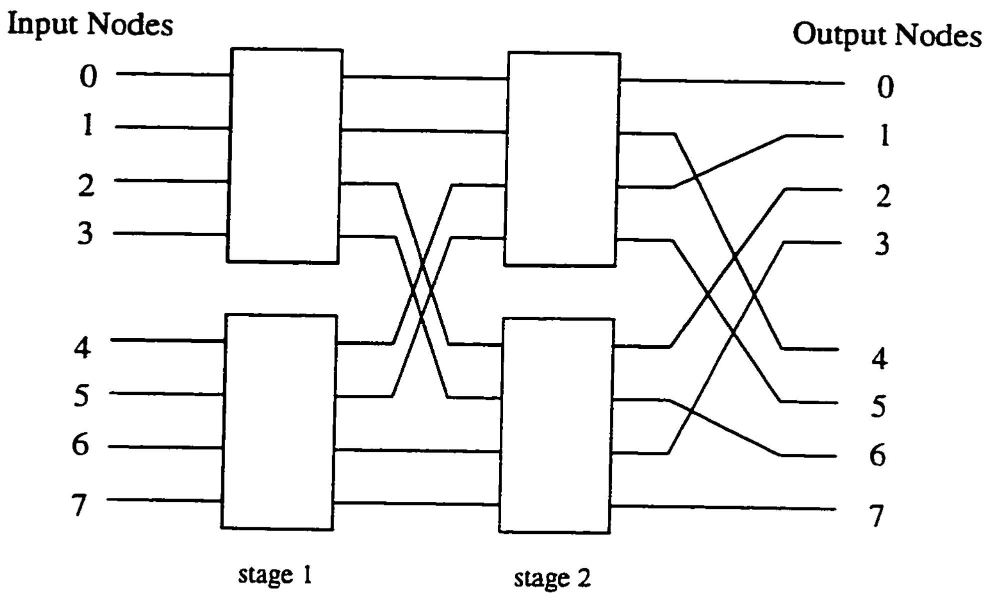


Figure 4.1: Data Network with 8 Nodes

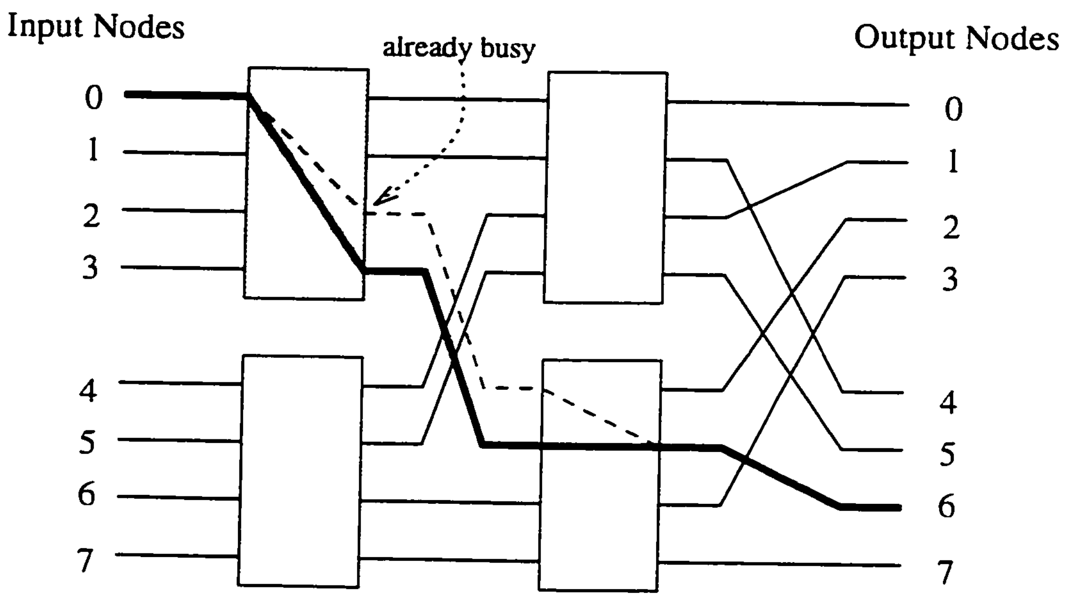


Figure 4.2: An example of random routing in 8-node data network A dashed line represents the original path between node 0 and node 6. A thick line represents the final path after random routing is used.

## 4.2 Routing in Data Network

The primary function of the data network is to transfer information among the processing nodes of the EARTH multiprocessor system in an efficient manner. *Routing* is the communication methods and algorithms used to implement the above function. This section provides an overview of some basic issues in routing applied to our proposed data network.

### 4.2.1 Routing Algorithms

The routing scheme of the data network can be straight forward if the modified version of indirect binary $n$-cube network is chosen to be our data network topology. The data network exhibits self-routing capability [2], that is, routing can be performed in a distributed manner using the destination-address as the reference when the routing header is created. Thus, any output node of the network can be reached from any input node by simply following the binary address of the output node. The routing header is assigned by the input node according to the following routing algorithms.

The routing algorithm uses the $(i+1)$th most significant bit of the destination address to set up the switch in the $i$th stage, selecting the upper output port (port 0 or port 1) of the switch if this bit is 0 and lower output port (port 2 or port 3) if this bit is 1. The selection of the output port at the last stage is given by the combination of the least significant bit and most significant bit. For example, in the 16-node data network of Figure 4.3, output node 10 (1010 in binary) can be reached from any of the input nodes by choosing the upper output port in stage 1, lower output port in stage 2 and upper output port in stage 3. In stage 3, the output port is determined by the combination of the least significant bit and the most significant bit (01 in binary) which is output port 1.

The above routing algorithm applies only to one-to-one message-passing. However, for one-to-all message-passing (broadcast), another routing algorithm is used. When an input node wants to broadcast message to all the output nodes, the input node routes the broadcast message to the upper port and the lower port of the switches between the first stage and the $(n-1)$th stage, where $n$ is the total number of stages in the data network. In the $n$th stage (the last stage), the broadcast message is routed to all output ports of each of the switches. Figure 4.4 illustrates the paths when the broadcast message traverses from input node 4 to all output nodes.

### 4.2.2 Control Commands

The physical port width of the data network is 9 bits, each transfer unit (data and control) consists of a byte data and a control bit. The most significant bit of the transfer unit is called the control bit and is used to distinguish between commands and data. The remaining 8 bits serves as data or commands. If the control bit is set to 1, the transfer unit carries either routing or control information. If control bit is set to 0, the transfer unit carries a data byte which has to be transferred to the corresponding output node. Table 4.1 shows the definitions of the control commands which are recognized by the data network.

In order to set up a communication path in the interconnection network properly, the SU must consist of a routing table in which the information of the network topology is stored. Before a message is ready to be sent, SU should retrieve the routing commands from the routing table according to the address of the destination node. Then, the routing commands are appended to the message. The routing table is programmable so that its contents can be written at boot time. Since the routing table is programmable, the network designer can configure the routing table in one of the three possible ways. First, the routing commands from stage 1 to stage $n-1$ in the network can be random routing type only.

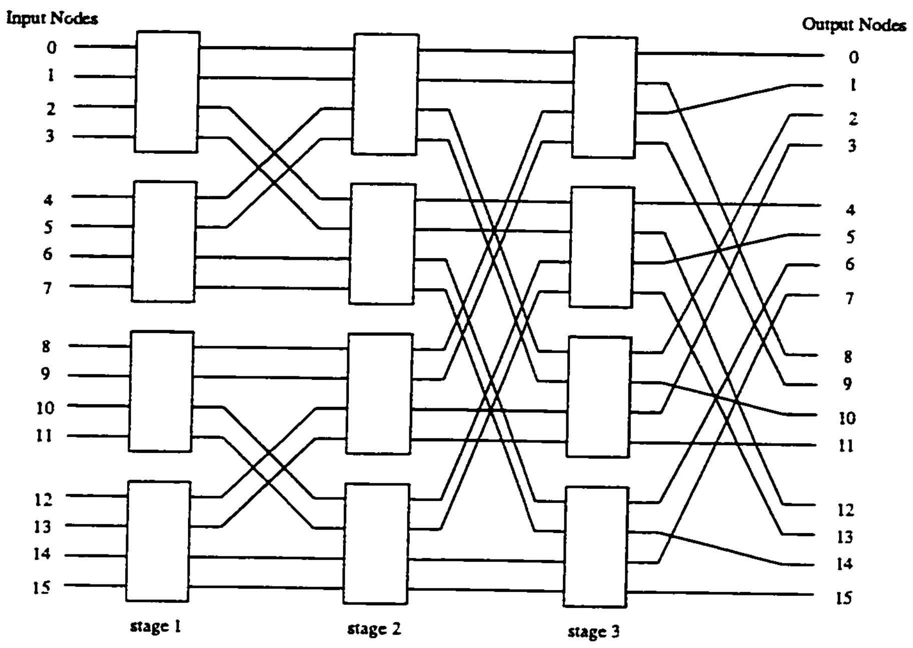


Figure 4.3: Data Network with 16 Nodes

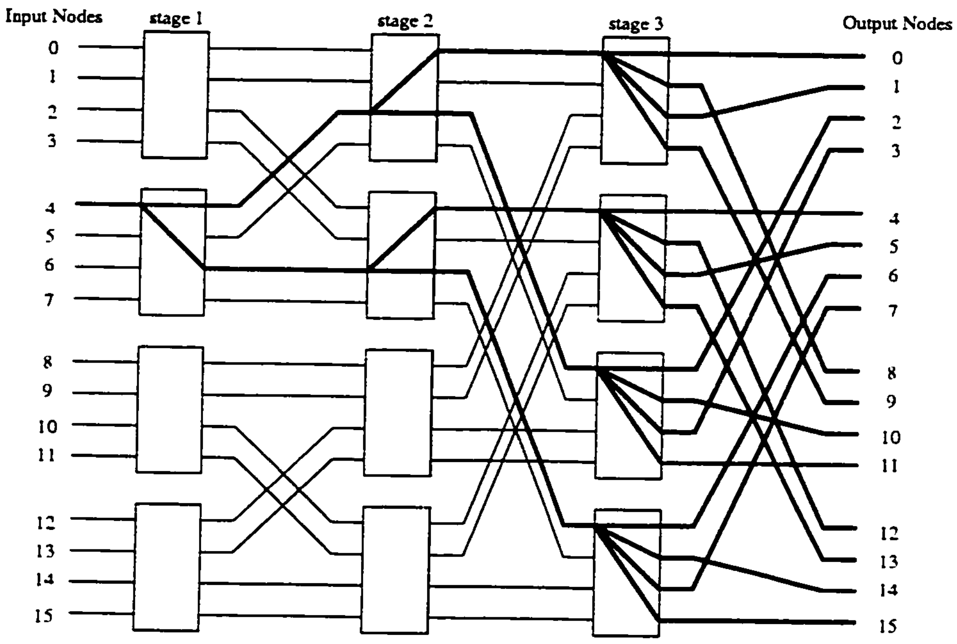


Figure 4.4: An example of routing algorithm for message broadcast. The thick lines indicate the paths for which the broadcasting message propagates through the data network.

| Name | Code | Data/Control | Function |
|---|---|---|---|
| DATA | 0xxxxxxxx | Data | Data byte |
| ROUTE_0 | 111110000 | Control | Route to port 0 |
| ROUTE_1 | 111110001 | Control | Route to port 1 |
| ROUTE_2 | 111110010 | Control | Route to port 2 |
| ROUTE_3 | 111110011 | Control | Route to port 3 |
| ROUTE_U | 111110100 | Control | Random route to upper port (port 0 or port 1) |
| ROUTE_L | 111110101 | Control | Random route to lower port (port 2 or port 3) |
| ROUTE_UL | 111110110 | Control | Random route to upper port and lower port |
| ROUTE_A | 111110111 | Control | Route to all ports |
| TAG_1 | 100xxxxxx | Control | First byte of the ordering tag which contains the binary address of input node. |
| TAG_2 | 101xxxxxx | Control | Second byte of the ordering tag which contains the message number assigned by input node. |
| CLOSE | 111000000 | Control | Close connection |
| ABORT | 111010000 | Control | Abort connection |
| FILLER | 111111111 | Control | Do nothing |

Table 4.1: Command Definitions

Second, the routing commands from stage 1 to stage $n-1$ can be a combination of random routing type and non-random routing type. Third, the routing commands from stage 1 to stage $n-1$ can be non-random routing type only. Table 4.2a through Table 4.2c illustrate three different sets of routing commands, each of them can be used to route a message from node 4 to node 13 in the 16 nodes 3-stage data network shown in Figure 4.3.

### 4.2.3 Message Format

A message is typically generated by the SU of an input node and injected into the interconnection network. Finally, the message reaches the SU of an output node. The message consists of a sequence of control commands and data. The control commands are interleaved with the data to form the message in order to accomplish flow control on the interconnection network. Normally, the sequences of commands and data are as follows:

**Message sent:** routing commands $\rightarrow$ ordering tag $\rightarrow$ data $\rightarrow$ close

**Message received:** ordering tag $\rightarrow$ data $\rightarrow$ close

At the beginning of each message, an input node should put one routing command per stage in the data network on the way to an output node, then the ordering tag followed by some data bytes, and finally a close command is at the end of the message.

The purpose of the routing commands is to provide desired path through the data network. When a message enters the switch, the first byte of the routing commands is examined. Once this byte is used to select an outgoing port of a switch, it is discarded from the head of the message. In any case, an output node should never receive any routing commands because they are already consumed by the data network. If the path between input node and output node goes through $n$ stages within the data network, there have to be $n$ routing commands in the header of every message.

| Stage | Routing Command | Type |
|---|---|---|
| 1 | 111110101 | Random Routing |
| 2 | 111110100 | Random Routing |
| 3 | 111110011 | Non-Random Routing |

(a)

| Stage | Routing Command | Type |
|---|---|---|
| 1 | 111110101 | Random Routing |
| 2 | 111110001 | Non-Random Routing |
| 3 | 111110011 | Non-Random Routing |

(b)

| Stage | Routing Command | Type |
|---|---|---|
| 1 | 111110010 | Non-Random Routing |
| 2 | 111110000 | Non-Random Routing |
| 3 | 111110011 | Non-Random Routing |

(c)

Table 4.2: The examples of routing commands for a 3-stage data network: (a) The routing commands from stage 1 to stage 2 in the network can be random routing type only. (b) The routing commands from stage 1 to stage 2 in the network can be a combination of random routing type and non-random routing type. (c) The routing commands from stage 1 to stage 2 can be non-random type only.

Our data network can have only one byte of routing command for each message. For the deterministic routing, only two bits of the routing command are needed for each stage since each switching elements has four output ports. However, using only one byte of routing command for each message imposes the limitation on the size of the data network. The size of the data network cannot be larger than four stages. Therefore, using one byte of routing command per stage gives us flexibility to expand the data network. No matter how large the data network is, it always requires one byte of routing command per stage.

When an input node sends several messages to the same output node, it is possible that the output node receives the messages out of order caused by random routing. In order to provide solution for this undesired situation, the input node must include the ordering tag in each of its outgoing messages. Each of the ordering tag is two bytes long. The first byte contains the binary address of the input node and the second byte contains the number which indicates how many messages have been sent to the corresponding output node. Therefore, the output node can use the information provided by the ordering tag to reorder the out-of-order messages after they are received.

Although the physical port width of the data network is 1 byte. the logical data structures that the data network handles are 8-byte wide. The communication protocol requires that the network interface to the data network provides eight data bytes as the unit of exchange. In case of data block transfer, the data block routed through the data network is composed of consecutive 8-byte quantities.

When the message is routed through the data network, routing commands are only executed and consumed by the switches. A switch only executes and consumes the first routing command. Then the switch will make logical connection to its corresponding output port. After the logical connection is established, all further commands and data are passed on until a close command is detected. When the close command is detected by the switch. the close command will be passed on and the logical connection will be closed. Figure 4.5 shows an example of message before and after transferring through the 3-stage data network. No data and commands (except routing commands) can be consumed or modified by the data network.

The filler command is created by the SU of an input node and is used to fill the unused data byte. The filler command does not generate any operation within the data network. The filler command will be destroyed by the network interface at the destination node.

## 4.3 Comparison with Other Switching Chips

Small switching chips are increasingly being viewed as a building block for interconnection network because of their low hardware cost. Furthermore, they can be connected in many different network topologies. This section briefly outlines some of the difference between our 4 × 4 crossbar switch and some of other designs.

The PaRC [4, 5] and Arctic [10] are designed to provide the interconnection networks for Monsoon [5] and *T [20] multiprocessor systems respectively. Both of them have 4 input ports and 4 output ports. They are different from our 4 × 4 crossbar switch in that there are multiple input buffers in each of the input ports. Each of their buffers can hold exactly one message. and each of them is directly connected to all output ports. The main reason for doing this is to maximize the chip's throughput. This buffering scheme allows large number of messages to be the candidates for output ports at the same time but it requires complex scheduling hardware support for the output ports.

Other switching chip such as Myrinet [18] switch has only one long FIFO in each of the input ports. A key difference between our 4 × 4 crossbar switch and Myrinet switch is

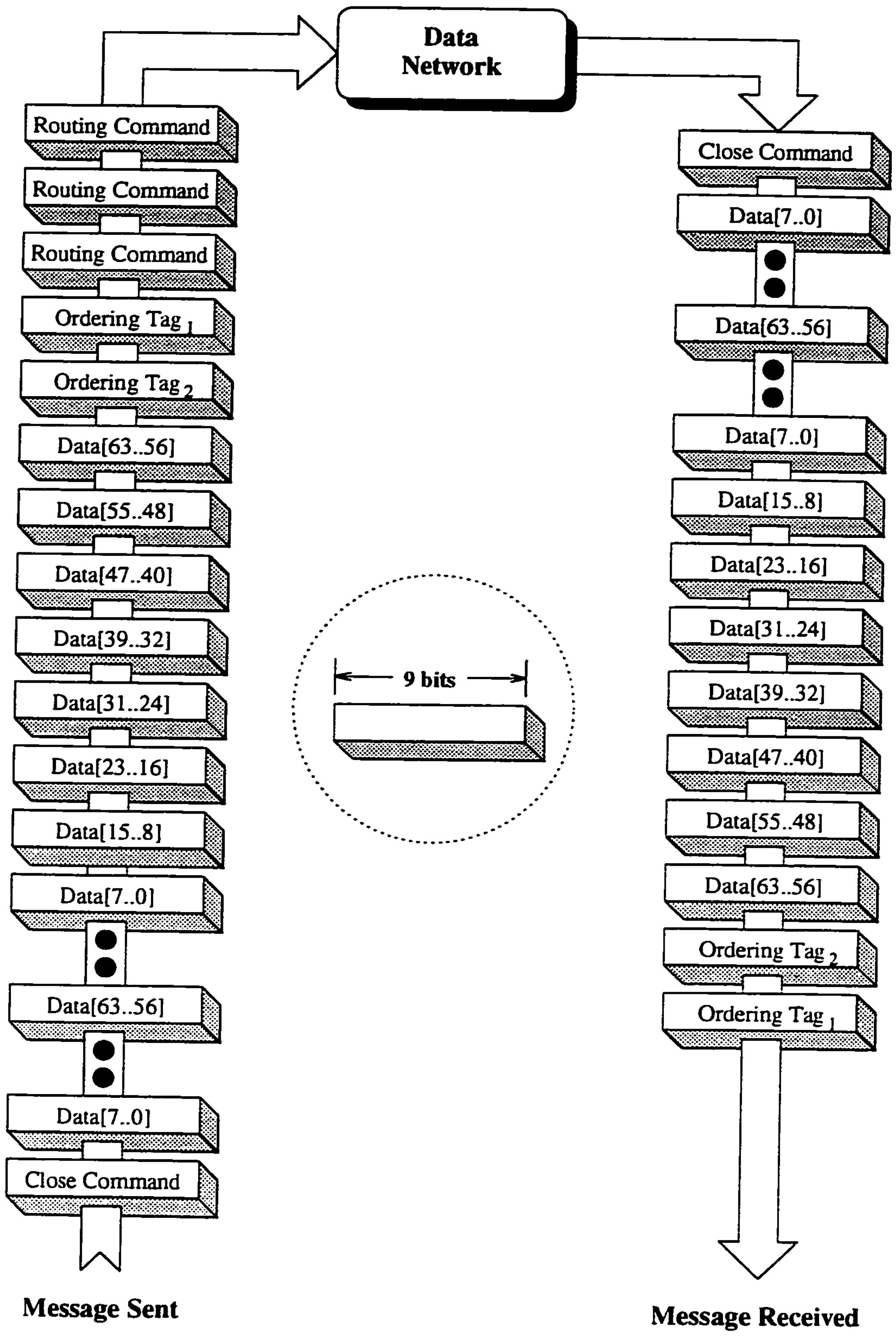


Figure 4.5: An Example of Message for 3-Stage Data Network.

that Myrinet switch does not support random routing. Without using random routing, the message at the head of the FIFO is blocked if the selected output port is being occupied by another message. Nothing can be read from this FIFO until the selected output port becomes free. This imposes a strict limitation on the throughput of the Myrinet switch. Our design offers random routing to reduce the number of blocking. Therefore, the throughput of our design is better than that of the Myrinet switch.

In order to compare the throughput of different designs, a simple analysis will be used to compute their output utilization rates. The output utilization rate is defined to be the probability that at least one out of $M$ messages is routed to a given output port, where $M$ is the total number of buffers of a chip.

PaRC has 4 separate buffers per input port, the probability that at least one out of 16 randomly addressed messages is routed to a given output port is $1 - (\frac{3}{4})^{16}$ which is approximately equal to 0.99. The term $(\frac{3}{4})^{16}$ computes the probability that none of the 16 messages is routed to a given output port. If we subtract this term from 1, it gives the probability that a given output port is utilized by at least one of the messages.

Similarly Arctic has 3 buffers per input port, the probability that at least one out of 12 randomly addressed messages is routed to a given output port is $1 - (\frac{3}{4})^{12}$ which is approximately equal to 0.97. Myrinet switch has one buffer per input port, the probability that at least one out of 4 randomly addressed messages is routed to a given output port is $1 - (\frac{3}{4})^{4}$ which is approximately equal to 0.68. Our design has one buffer per input but it employs random routing so that the probability that at least one out of 4 randomly addressed messages is routed to a given output port is $1 - (\frac{1}{2})^{3}(\frac{3}{4})$ which is approximately equal to 0.91.

This simple analysis shows that our $4 \times 4$ crossbar switch can achieve high output utilization rate by using random routing, even though it has only one FIFO buffer in each of the input ports. Note that the analysis is not precise because it assumes each of the buffers has a message which is randomly addressed. This analysis allows us to compute the best case utilization rate for the different switching chips. Nevertheless, this analysis provides a quick comparison between different switching chips based on their buffering schemes. Table 4.3 summarizes the output utilization rate for the different switching chips.

| **Switching Chip** | **Output Utilization Rate** |
|---|---|
| PaRC | 99% |
| Arctic | 97% |
| Myrinet | 68% |
| Our Design | 91% |

Table 4.3: Output Utilization Rate for Different Switching Chips

# Chapter 5

# Network Interface

This chapter describes the design of the network interface. Network interface performs data transfer and protocol conversion between an EARTH node and the data network. An EARTH node is connected to the data network via a network interface. The network interface interfaces an EARTH node to the data network. It isolates each from the details of others. Network interface gives an EARTH node a simple and uniform view of the data network. and the data network gets a simple and uniform view of an EARTH node. The architecture to implement the network interface will be explained.

## 5.1 Architecture of Network Interface

The network interface has one input channel to retrieve data from the data network. and one output channel to provide data to the data network. Both channels are independent of each other and they can work simultaneously without interfering with each other. Each channel has a buffer that is able to store up to 8 8-byte-words (64 bytes). The buffer is useful to connect both sides of the network interface (EARTH node and data network) which have different data transfer rates. The difference of data transfer rates result from the fact that the width of the data path on each side of the network interface is different. The data path is 8-byte wide on the EARTH node side whereas the data path is 1-byte wide on the data network side. On the EARTH node side, the network interface can execute one request per clock cycle in pipeline mode. The response to a request can be made within the same clock cycle. If there is enough space in the send buffer, the network interface can take an 8-byte word per clock cycle to send. If there is any data in the receive buffer, the network interface is able to provide one 8-byte-word per clock cycle to receive data.

The network interface does not start an operation on the EARTH node side, if chip select signal, CS, is LOW. In this case all other control signals (R/W and DATA_TYPE) are ignored. If CS is HIGH, the network interface latches the DATA_TYPE and R/W signals and executes the operation within the same clock cycle. The operation can be read, write or reset. For the read operation, the network interface provides data to the EARTH node. For the write operation, the network interface takes the data from the EARTH node. The EARTH node must first check the interface's status information before data is written to or read from the network interface. For the reset operation, all data and control information stored in the network interface will be cleared. The block diagram of the network interface is shown in Figure 5.1.

### 5.1.1 Send FIFO

The Send FIFO accepts data from the EARTH node. It then transfers the data through the data network to the RECEIVE FIFO of the network interface at the destination node. The logic block diagram of the SEND FIFO is shown in Figure 5.2.

The Send FIFO has 8-byte input port (DATA) and 1-byte output port (DATA_OUT). The input port is controlled by the master clock (CLK) and three other signals (CS, R/W

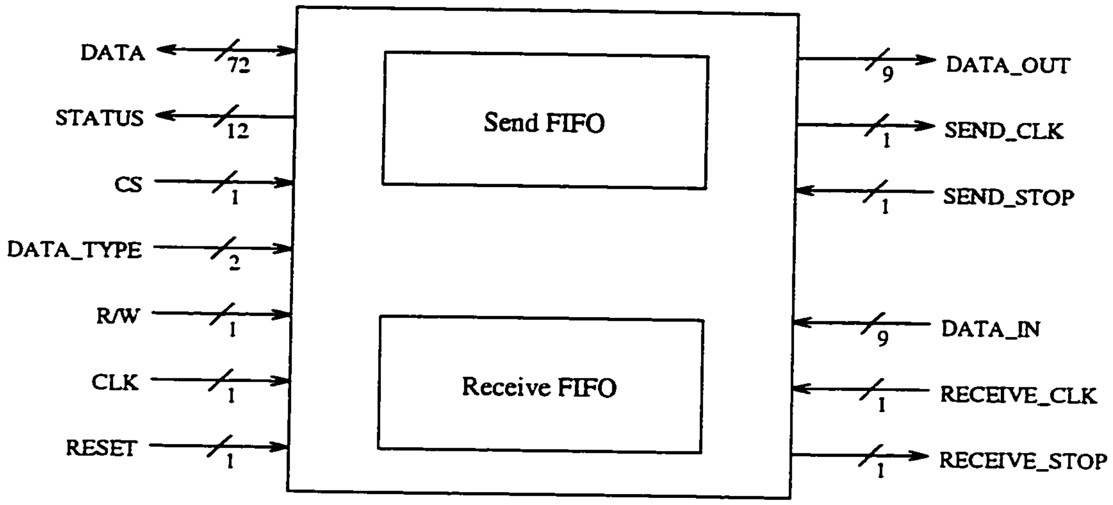


Figure 5.1: Network Interface Block Diagram

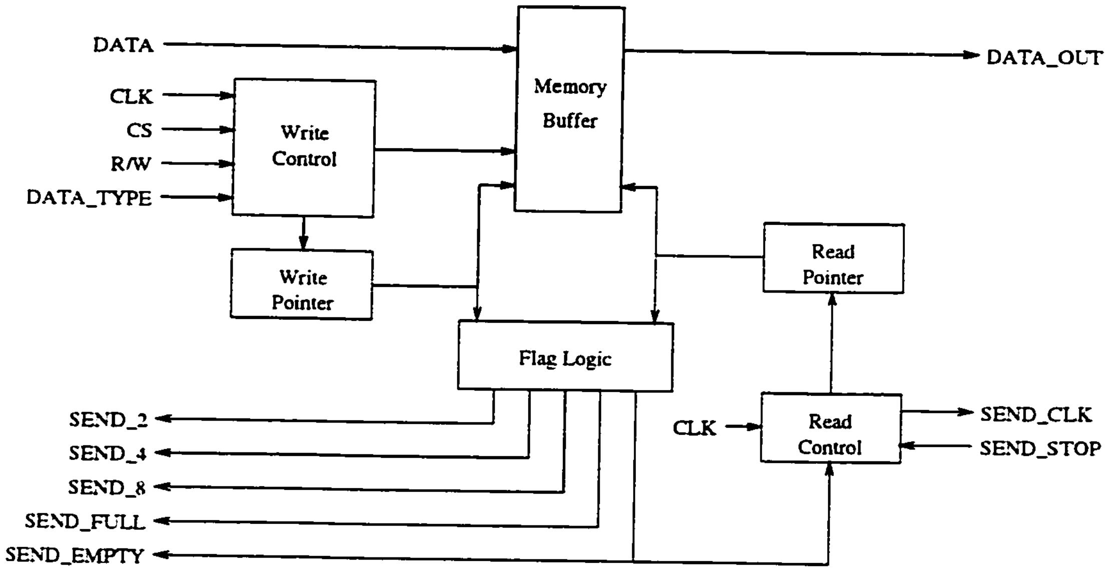


Figure 5.2: Send FIFO Logic Block Diagram

and DATA_TYPE). When both CS and R/W are HIGH and DATA_TYPE is not equal to 00 (in binary), data is written into the buffer on the rising edge of CLK signal. The 2-bit DATA_TYPE signal determines how many bytes of data are written in the buffer. If DATA_TYPE is equal to 01 (in binary), only the lower 2 bytes of DATA are written into the buffer. If DATA_TYPE is equal to 10 (in binary), only the lower 4 bytes of DATA are written into the buffer. If DATA_TYPE is equal to 11 (in binary), only the complete 8-byte words of DATA are written into the buffer.

In case DATA_TYPE is equal to 00 (in binary), and both CS and R/W are HIGH, the network interface performs reset after the rising edge of CLK. All data and control information stored in the network interface will be lost. After the reset, the network interface returns to the empty condition.

The output port of the SEND FIFO is controlled in a different manner. When the buffer is not empty and the input SEND_STOP is LOW, a byte of data is continually read from the buffer every clock cycle. A pulse signal, SEND_CLK, is generated for each byte of data read out from the buffer.

The SEND FIFO provides five status flags: SEND_EMPTY, SEND_FULL, SEND_2, SEND_4 and SEND_8. The flags represent the current state of the SEND FIFO and can change dynamically according to either the write or read operation. The definitions of the flags are given in Table 5.1.

Since the EARTH node can write faster than the network interface is able to transfer data, it is possible to overflow the SEND FIFO. In order to avoid an overflow, the EARTH node should check the current state of the SEND FIFO to determine how much space is free before writing data into the SEND FIFO.

### 5.1.2 Receive FIFO

The Receive FIFO receives data from the data network. It then provides the data to the EARTH node. The logic block diagram of the Receive FIFO is shown in Figure 5.3.

The Receive FIFO has 1-byte input port (DATA_IN) and 8-byte output port (DATA). The input port is controlled by a write clock (RECEIVE_CLK). On the rising edge of RECEIVE_CLK, the data is written into the buffer. If an incoming data from the data network is a filler command, it is consumed by the Receive FIFO and never reaches the EARTH node. When the Receive FIFO is full, RECEIVE_STOP is set to HIGH so that the output channel of the data network will stop transferring data until RECEIVE_STOP becomes LOW.

The output port of the Receive FIFO is controlled by the master clock (CLK) and two other signals (CS and R/W). When CS is HIGH and R/W is LOW, data is read from the buffer. If the head of the buffer contains a command, the EARTH node only takes the lower byte. If the head of the buffer contains data, the EARTH node take the complete 8-byte words.

The Receive FIFO provides seven status flags: RECEIVE_EMPTY, RECEIVE_FULL. RECEIVE_8. RECEIVE_16. RECEIVE_24. DAHOD and CAHOD. The flags represents the current state of the Receive FIFO and can change dynamically according to either the write or read operation. The definitions of the flags are given in Table 5.2.

Since the EARTH node can read faster than the network interface is able to receive data. it is possible for the EARTH node to read when the Receive FIFO is empty. In order to avoid an underflow. the EARTH node should check the current state of the Receive FIFO to determine how many bytes of data are already stored before reading data from the Receive FIFO.

| Flag Name | Description |
|---|---|
| SEND_EMPTY | When SEND_EMPTY is HIGH, the Send FIFO is empty. |
| SEND_FULL | When SEND_FULL is HIGH, the Send FIFO is full. |
| SEND_2 | When SEND_2 is HIGH. the free space of SEND FIFO is less than 2 bytes. |
| SEND_4 | When SEND_4 is HIGH, the free space of SEND FIFO is less than 4 bytes. |
| SEND_8 | When SEND_8 is HIGH, the free space of SEND FIFO is less than 8 bytes. |

Table 5.1: Status Flag Definitions for Send FIFO

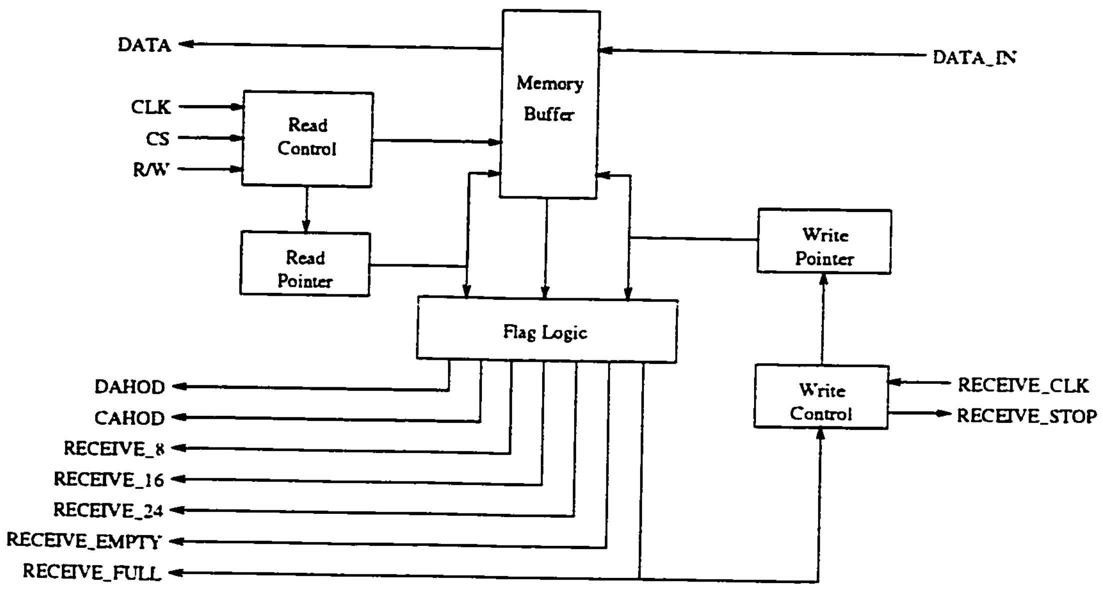


Figure 5.3: Receive FIFO Logic Block Diagram

| Flag Name | Description |
|---|---|
| RECEIVE_EMPTY | When RECEIVE_EMPTY is HIGH, the RECEIVE FIFO is empty. |
| RECEIVE_FULL | When RECEIVE_FULL is HIGH, the RECEIVE FIFO is full. |
| RECEIVE_8 | When RECEIVE_8 is HIGH, the RECEIVE FIFO contains at least 8 bytes of data. |
| RECEIVE_16 | When RECEIVE_16 is HIGH, the RECEIVE FIFO contains at least 16 bytes of data. |
| RECEIVE_24 | When RECEIVE_24 is HIGH, the RECEIVE FIFO contains at least 24 bytes of data. |
| DAHOD | When DAHOD is HIGH. the head of the RECEIVE FIFO contains a data. |
| CAHOD | When CAHOD is HIGH, the head of the RECEIVE FIFO contains a control command. |

Table 5.2: Status Flag Definitions for RECEIVE FIFO

# Chapter 6

# Simulation Results of Network Interface

The design of the network interface has been described in Chapter 5. The simulation results of the network interface will be presented in this chapter.

## 6.1 Maximum Clock Frequency

The speed of the network interface is limited by the data access time of the two FIFO's. Figure 6.1 shows the data waveforms of the Send FIFO and the Receive FIFO. and the related timing parameters are given in Table 6.1.

In Figure 6.1a. the maximum time for the data, DATA_OUT, to become ready after the CLK changing from LOW to HIGH is found to be 4 ns. Also the minimum setup time for the data before the CLK changing from High to LOW is found to be 3 ns. Therefore. the

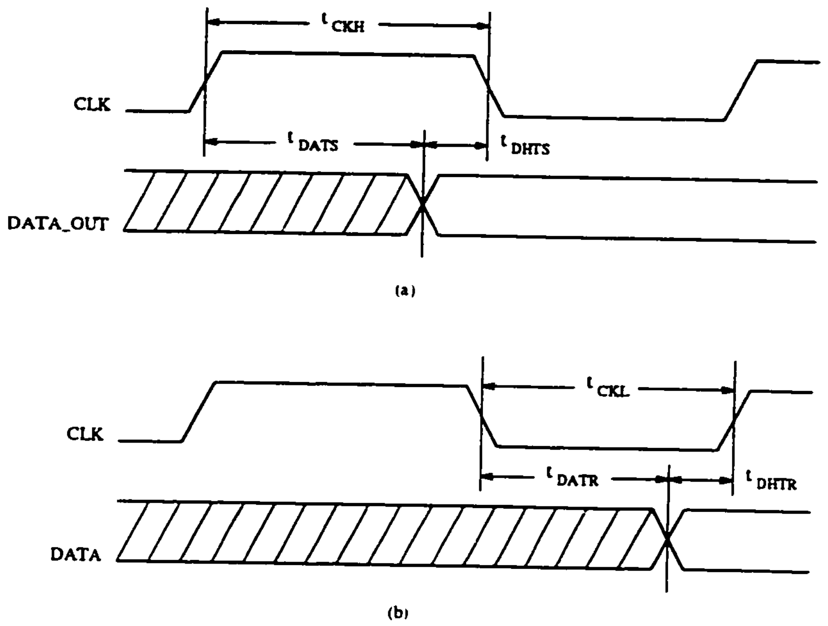


Figure 6.1: Data Waveforms: (a) Send FIFO (b) Receive FIFO

| **Parameter** | **Description** | **Max.** | **Min.** | **Unit** |
|---|---|---|---|---|
| $t_{DATS}$ | Data access time for Send FIFO | 4 | | ns |
| $t_{DHTS}$ | Data setup time for Send FIFO | | 3 | ns |
| $t_{DATR}$ | Data access time for Receive FIFO | 6 | | ns |
| $t_{DHTR}$ | Data setup time for Receive FIFO | | 2 | ns |

Table 6.1: Measurement of Timing Parameters for Data Waveforms

minimum time for the CLK to stay HIGH, $t_{CKH}$, can be computed as:

$$\begin{aligned} t_{CKH} &= t_{DATS} + t_{DHTS} \\ &= 4\,ns + 3\,ns \\ &= 7\,ns \end{aligned}$$

In Figure 6.1b, the maximum time for the data, DATA, to become ready after the CLK changing from HIGH to LOW is found to be 6 ns. Also the minimum setup time for the data before the CLK changing from LOW to HIGH is found to be 2 ns. Therefore, the minimum time for the CLK to stay LOW, $t_{CKL}$, can be computed as:

$$\begin{aligned} t_{CKL} &= t_{DATR} + t_{DHTR} \\ &= 6\,ns + 2\,ns \\ &= 8\,ns \end{aligned}$$

Hence, the maximum clock cycle time, $t_{CK}$, is calculated as:

$$\begin{aligned} t_{CK} &= t_{CKH} + t_{CKL} \\ &= 7\,ns + 8\,ns \\ &= 15\,ns \end{aligned}$$

Finally, the maximum clock frequency $f_{max}$ is obtained as:

$$\begin{aligned} f_{max} &= \frac{1}{t_{CK}} \\ &= \frac{1}{15\,ns} \\ &= 66.7\,MHz \end{aligned}$$

From the above calculation result, the peak transfer rates for the network interface are equal to 533.3 Mbytes/second on the EARTH node side and 66.7 Mbytes/second on the data network side.

## 6.2 Pin Numbers and Chip Area

The design of the network interface is mapped into the Logic's 0.6-Micron LCA300K compacted array technology. The total number of pins used by the network interface is 112. The total area for the network interface is summarized in Table 6.2.

## 6.3 Switching Waveforms

Figure 6.2 shows the switching waveforms for the Network Interface. The measured values for the timing parameters are summarized in Table 6.3.

| Type | Area (gate equivalents) |
|---|---|
| Combinational | 13725 |
| Non-combinational | 27923 |
| Wire Interconnection | 30198 |
| Total | 71846 |

Table 6.2: Area for Network Interface

| Parameter | Description | Max. | Min. | Unit |
|---|---|---|---|---|
| $t_{DAST}$ | DATA setup time | | 4 | ns |
| $t_{DAHT}$ | DATA hold time | | 1 | ns |
| $t_{CSST}$ | CS setup time | | 6 | ns |
| $t_{CSHT}$ | CS hold time | | 1 | ns |
| $t_{RWST}$ | R/W setup time | | 6 | ns |
| $t_{RWHT}$ | R/W hold time | | 1 | ns |
| $t_{DTST}$ | DATA_TYPE setup time | | 6 | ns |
| $t_{DTHT}$ | DATA_TYPE hold time | | 1 | ns |
| $t_{RCKS}$ | RECEIVE_CLK pulse width | | 1 | ns |
| $t_{DIST}$ | DATA_IN setup time | | 5 | ns |
| $t_{DIHT}$ | DATA_IN hold time | | 1 | ns |
| $t_{CTRS}$ | RECEIVE_CLK to RECEIVE_STOP | 4 | | ns |
| $t_{RS}$ | RESET pulse width | | 2 | ns |
| $t_{RSDO}$ | RESET to DATA_OUT and output time | 3 | | ns |
| $t_{RSSC}$ | RESET to SEND_CLK and output time | 3 | | ns |

Table 6.3: Switching Characteristics of Network Interface

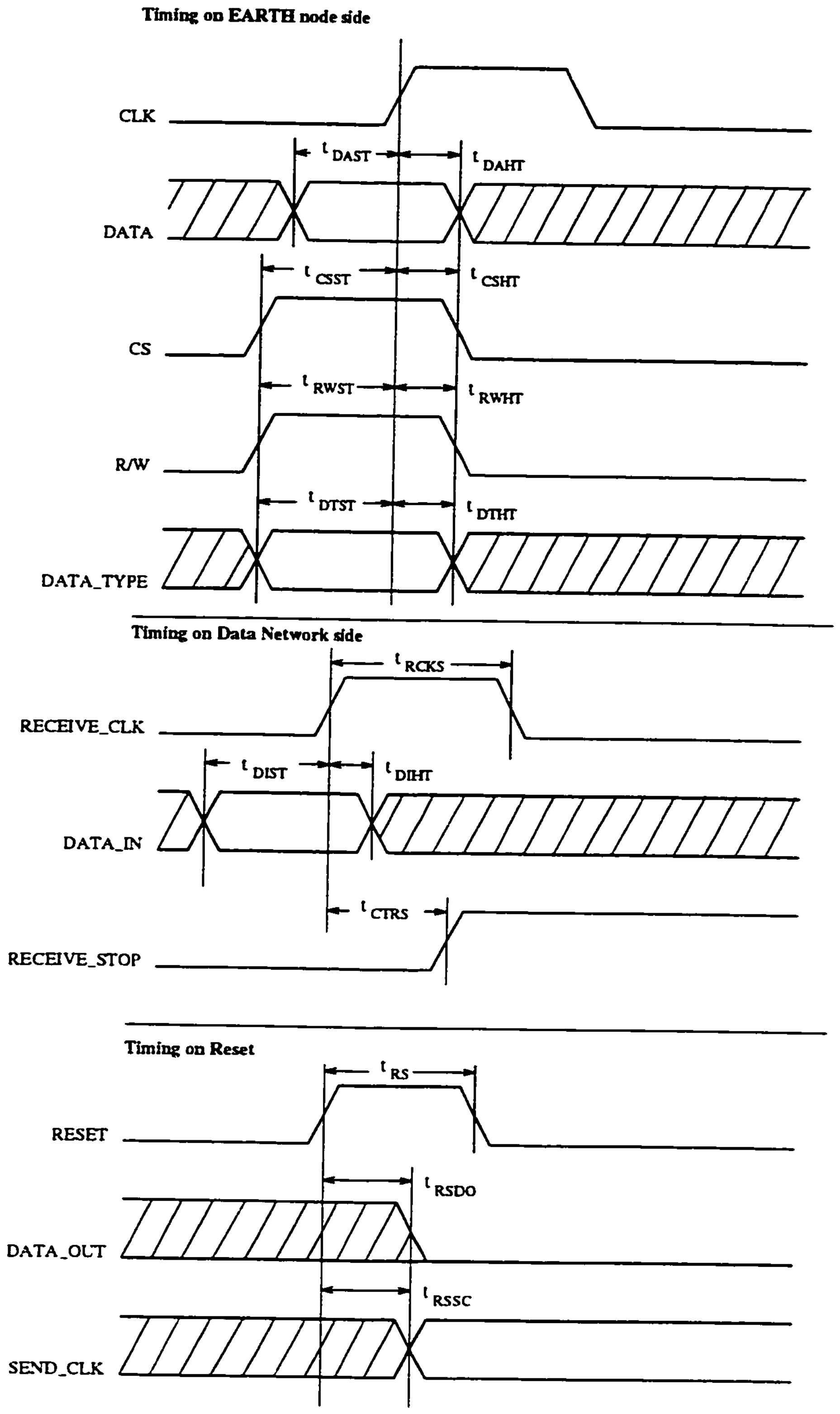


Figure 6.2: Switching Waveforms for Network Interface

# Chapter 7

# Conclusion

Interconnection networks play an important role in the performance of any multiprocessor system. An interconnection network with high network latency can significantly degrade the system performance. In this thesis, we designed a low-cost and high-bandwidth interconnection network for the EARTH multithreaded multiprocessor system. We believe that the contents of this thesis can be successfully used to build a prototype interconnection network for EARTH.

## 7.1 Summary

Our design of the $4 \times 4$ switching chip is a very powerful building block for constructing interconnection networks. The design of the $4 \times 4$ switching chip is not limited to the EARTH system, it can be used in a large variety of interconnection networks which require high throughput and low latency. Each port of our $4 \times 4$ switching chip provides 66.7 Mbytes/second of bandwidth. When a message comes into the $4 \times 4$ switching chip and its output port is available, the connection latency will be at most 4 clock cycle (60 ns).

A number of aspects of the design contribute to achieving this high performance. In particular, the employment of random routing helps enormously. Random routing can greatly reduce the amount of blocking that occurs since a message can be rerouted to around the busy output port. In addition, all logical connections between inputs and outputs are independent of each other and thus can be established simultaneously without hindering each other. Hence, the maximum data transfer rate of 266.7 Mbytes/second is achievable since up to 4 different connections can be established simultaneously. In order to provide fair and fast arbitration services for the large number of incoming messages, the arbiter uses WWFA technique to resolve conflicts among the messages which are competing for the same output port. The response time for the arbiter is very fast, the arbiter can process a request within the same clock cycle the request is made. Also the $4 \times 4$ switching chip allows a message to be sent off the chip before the entire message has been received. This virtual cut-through routing is essential in minimizing the connection latency of messages. Furthermore, the $4 \times 4$ switching chip provides broadcast facility. This facility greatly reduces the amount of traffic within the network if the network involves a lot of one-to-all communications.

The network interface also helps to make effective use of the bandwidth. The network interface does not require idle time between messages; as long as there are messages stored into the Send FIFO, they will be injected into the data network continuously. On the other hand, the SU can retrieve the messages from the network interface continuously as long as there are enough messages stored into the Receive FIFO.

## 7.2 Future Work

The design of the interconnection network for the EARTH multithreaded multiprocessor system has been completed, and the simulations of the $4 \times 4$ crossbar switch and the network interface have been done. In the future it may be desirable to build a prototype interconnection network for the EARTH system. There are several modifications to the $4 \times 4$ switching chip that any future version should consider.

One change would be to increase the reliability of the chip. It can be done by adding a cyclic-redundancy-check (CRC) error checking unit into the $4 \times 4$ switching chip. The CRC error checking unit is used to ensure that messages have not been corrupted during transmission.

The other change would be to add multicast facility into the $4 \times 4$ switching chip. The multicast facility allows a source node to send messages to a group of destination nodes. This feature is very useful when the interconnection network involves a lot of messages for which only destination nodes that are actively interested in a particular multicast service will have such messages routed to them. This reduces bandwidth consumption to the link between the source node and destination node of multicast messages.

# Appendix A

# Verilog Code

## A.1 $4 \times 4$ Crossbar Switch

```
module SWITCH (CLK, RESET,
               DATA_IN0, WCLK_IN0, STOP_OUT0,
               DATA_IN1, WCLK_IN1, STOP_OUT1,
               DATA_IN2, WCLK_IN2, STOP_OUT2,
               DATA_IN3, WCLK_IN3, STOP_OUT3,
               DATA_OUT0, WCLK_OUT0, STOP_IN0,
               DATA_OUT1, WCLK_OUT1, STOP_IN1,
               DATA_OUT2, WCLK_OUT2, STOP_IN2,
               DATA_OUT3, WCLK_OUT3, STOP_IN3);

  input CLK, RESET;

  input [8:0] DATA_IN0, DATA_IN1, DATA_IN2, DATA_IN3;
  input WCLK_IN0, WCLK_IN1, WCLK_IN2, WCLK_IN3;
  input STOP_IN0, STOP_IN1, STOP_IN2, STOP_IN3;

  output [8:0] DATA_OUT0, DATA_OUT1, DATA_OUT2, DATA_OUT3;
  output WCLK_OUT0, WCLK_OUT1, WCLK_OUT2, WCLK_OUT3;
  output STOP_OUT0, STOP_OUT1, STOP_OUT2, STOP_OUT3;

  wire BroadcastPending;

  /* FIFOs' signals */
  wire fifo0_EMPTY, fifo1_EMPTY, fifo2_EMPTY, fifo3_EMPTY;
  wire [8:0] fifo0_DATA_OUT, fifo1_DATA_OUT, fifo2_DATA_OUT, fifo3_DATA_OUT;

  /* CONTROLs' signals */
```

```
wire cont0_RCLK, cont1_RCLK, cont2_RCLK, cont3_RCLK;
wire cont0_WCLK, cont1_WCLK, cont2_WCLK, cont3_WCLK;
wire [3:0] cont0_request, cont1_request, cont2_request, cont3_request;
wire cont0_BP, cont1_BP, cont2_BP, cont3_BP;
wire [8:0] cont0_DATA_OUT, cont1_DATA_OUT, cont2_DATA_OUT, cont3_DATA_OUT;

/* CROSSBARs' signals */
wire cb0_STOP_OUT, cb1_STOP_OUT, cb2_STOP_OUT, cb3_STOP_OUT;

/* ARBITERs' signals */
wire [3:0] arb_grant0, arb_grant1, arb_grant2, arb_grant3;

assign BroadcastPending=cont0_BP|cont1_BP|cont2_BP|cont3_BP;

FIFO fifo0 (RESET, WCLK_IN0, cont0_RCLK, DATA_IN0, STOP_OUT0,
            fifo0_EMPTY, fifo0_DATA_OUT),
     fifo1 (RESET, WCLK_IN1, cont1_RCLK, DATA_IN1, STOP_OUT1,
            fifo1_EMPTY, fifo1_DATA_OUT),
     fifo2 (RESET, WCLK_IN2, cont2_RCLK, DATA_IN2, STOP_OUT2,
            fifo2_EMPTY, fifo2_DATA_OUT),
     fifo3 (RESET, WCLK_IN3, cont3_RCLK, DATA_IN3, STOP_OUT3,
            fifo3_EMPTY, fifo3_DATA_OUT);

CONTROL control0 (CLK, RESET, fifo0_DATA_OUT, cb0_STOP_OUT, fifo0_EMPTY,
                  arb_grant0, BroadcastPending, cont0_WCLK, cont0_RCLK,
                  cont0_request, cont0_BP,cont0_DATA_OUT),
        control1 (CLK, RESET, fifo1_DATA_OUT, cb1_STOP_OUT, fifo1_EMPTY,
                  arb_grant1, BroadcastPending, cont1_WCLK, cont1_RCLK,
                  cont1_request, cont1_BP,cont1_DATA_OUT),
        control2 (CLK, RESET, fifo2_DATA_OUT, cb2_STOP_OUT, fifo2_EMPTY,
                  arb_grant2, BroadcastPending, cont2_WCLK, cont2_RCLK,
                  cont2_request, cont2_BP,cont2_DATA_OUT),
        control3 (CLK, RESET, fifo3_DATA_OUT, cb3_STOP_OUT, fifo3_EMPTY,
                  arb_grant3, BroadcastPending, cont3_WCLK, cont3_RCLK,
                  cont3_request, cont3_BP,cont3_DATA_OUT);

CROSSBAR crossbar (cont0_DATA_OUT, cont0_WCLK,
                   cont1_DATA_OUT, cont1_WCLK,
                   cont2_DATA_OUT, cont2_WCLK,
                   cont3_DATA_OUT, cont3_WCLK,
                   STOP_IN0, STOP_IN1, STOP_IN2, STOP_IN3,
                   arb_grant0, arb_grant1, arb_grant2, arb_grant3,
                   cb0_STOP_OUT, cb1_STOP_OUT, cb2_STOP_OUT, cb3_STOP_OUT,
                   DATA_OUT0, WCLK_OUT0,
                   DATA_OUT1, WCLK_OUT1,
                   DATA_OUT2, WCLK_OUT2,
                   DATA_OUT3, WCLK_OUT3);
```

```
  ARBITER arbiter (CLK, BroadcastPending, RESET,
                   cont0_request, cont1_request, cont2_request,
                   cont3_request, arb_grant0, arb_grant1, arb_grant2,
                   arb_grant3);

endmodule


module FIFO (RESET,WCLK,RCLK,DATA_IN,AFULL,EMPTY,DOUT);
  input RESET,WCLK,RCLK;
  input [8:0] DATA_IN;
  output AFULL,EMPTY;
  output [8:0] DOUT;

  reg [4:0] HEAD,HP,TAIL,TP;
  wire [4:0] S,H;

  MEMORY_BANK_FF mbff (WCLK,TP,HP,DATA_IN,DOUT);


  ADDER_5B_FF add5_0 (TAIL,5'b00001,S,),
              add5_1 (HEAD,5'b00001,H,);

  assign EMPTY = &(HEAD~^TAIL),
         AFULL = &(S~^HEAD);

  always @(posedge RESET or posedge WCLK) begin
    if(RESET)
      TAIL = 5'b00000;
    else
      TAIL = S;
  end

  always @(posedge RESET or negedge WCLK) begin
    if(RESET)
      TP = 5'b00000;
    else
      TP = TAIL;
  end

  always @(posedge RESET or posedge RCLK) begin
    if(RESET)
      HEAD = 5'b00000;
    else
      HEAD = H;
  end

  always @(posedge RESET or negedge RCLK) begin
```

```
      if(RESET)
        HP = 5'b00000;
      else
        HP = HEAD;
    end
  endmodule


  module HALFADDER_FF (A,B,S,C);
    input A,B;
    output C,S;

    xor g0 (S,A,B);
    and g1 (C,A,B);
  endmodule


  module FULLADDER_FF (A,B,Ci,S,Co);
    input A,B,Ci;
    output S,Co;

    wire T1,T2,T3;

    HALFADDER_FF ha0 (A,B,T1,T2);
    HALFADDER_FF ha1 (Ci,T1,S,T3);
    or g0 (Co,T2,T3);
  endmodule


  module ADDER_5B_FF (A,B,S,C);
    input [4:0] A,B;
    output [4:0] S;
    output C;

    wire [3:0] carry;

    HALFADDER_FF ha0 (A[0],B[0],S[0],carry[0]);
    FULLADDER_FF fa1 (A[1],B[1],carry[0],S[1],carry[1]);
    FULLADDER_FF fa2 (A[2],B[2],carry[1],S[2],carry[2]);
    FULLADDER_FF fa3 (A[3],B[3],carry[2],S[3],carry[3]);
    FULLADDER_FF fa4 (A[4],B[4],carry[3],S[4],C);
  endmodule


  module MEMORYCELL_FF (WCLK,TPEQ,HPEQ,DIN,DOUT);
    input WCLK;
    input TPEQ,HPEQ;
    input [8:0] DIN;
  output [8:0] DOUT;

  reg [8:0] MEM;

  assign DOUT = (HPEQ==1'b1) ? MEM : 9'bzzzzzzzzz;

  always @(posedge WCLK) begin
    if(TPEQ)
      MEM = DIN;
  end
endmodule


module MEMORY_BANK_FF (WCLK,TP,HP,DIN,DOUT);
  input WCLK;
  input [4:0] TP,HP;
  input [8:0] DIN;
  output [8:0] DOUT;

  tri [8:0] DOUT;
  wire [31:0] HPEQ,TPEQ;

  FIFO_MUX5X32 fm0 (HP,HPEQ),
               fm1 (TP,TPEQ);

  MEMORYCELL_FF mc0  (WCLK,TPEQ[0] ,HPEQ[0] ,DIN,DOUT),
                mc1  (WCLK,TPEQ[1] ,HPEQ[1] ,DIN,DOUT),
                mc2  (WCLK,TPEQ[2] ,HPEQ[2] ,DIN,DOUT),
                mc3  (WCLK,TPEQ[3] ,HPEQ[3] ,DIN,DOUT),
                mc4  (WCLK,TPEQ[4] ,HPEQ[4] ,DIN,DOUT),
                mc5  (WCLK,TPEQ[5] ,HPEQ[5] ,DIN,DOUT),
                mc6  (WCLK,TPEQ[6] ,HPEQ[6] ,DIN,DOUT),
                mc7  (WCLK,TPEQ[7] ,HPEQ[7] ,DIN,DOUT),
                mc8  (WCLK,TPEQ[8] ,HPEQ[8] ,DIN,DOUT),
                mc9  (WCLK,TPEQ[9] ,HPEQ[9] ,DIN,DOUT),
                mc10 (WCLK,TPEQ[10],HPEQ[10],DIN,DOUT),
                mc11 (WCLK,TPEQ[11],HPEQ[11],DIN,DOUT),
                mc12 (WCLK,TPEQ[12],HPEQ[12],DIN,DOUT),
                mc13 (WCLK,TPEQ[13],HPEQ[13],DIN,DOUT),
                mc14 (WCLK,TPEQ[14],HPEQ[14],DIN,DOUT),
                mc15 (WCLK,TPEQ[15],HPEQ[15],DIN,DOUT),
                mc16 (WCLK,TPEQ[16],HPEQ[16],DIN,DOUT),
                mc17 (WCLK,TPEQ[17],HPEQ[17],DIN,DOUT),
                mc18 (WCLK,TPEQ[18],HPEQ[18],DIN,DOUT),
                mc19 (WCLK,TPEQ[19],HPEQ[19],DIN,DOUT),
                mc20 (WCLK,TPEQ[20],HPEQ[20],DIN,DOUT),
                mc21 (WCLK,TPEQ[21],HPEQ[21],DIN,DOUT),
                mc22 (WCLK,TPEQ[22],HPEQ[22],DIN,DOUT),
                    mc23 (WCLK,TPEQ[23],HPEQ[23],DIN,DOUT),
                    mc24 (WCLK,TPEQ[24],HPEQ[24],DIN,DOUT),
                    mc25 (WCLK,TPEQ[25],HPEQ[25],DIN,DOUT),
                    mc26 (WCLK,TPEQ[26],HPEQ[26],DIN,DOUT),
                    mc27 (WCLK,TPEQ[27],HPEQ[27],DIN,DOUT),
                    mc28 (WCLK,TPEQ[28],HPEQ[28],DIN,DOUT),
                    mc29 (WCLK,TPEQ[29],HPEQ[29],DIN,DOUT),
                    mc30 (WCLK,TPEQ[30],HPEQ[30],DIN,DOUT),
                    mc31 (WCLK,TPEQ[31],HPEQ[31],DIN,DOUT);
endmodule


module FIFO_MUX5X32 (A,B);

  input [4:0] A;
  output [31:0] B;

  wire [4:0] nA;

  assign nA = ~A;

  assign  B[0]   = nA[4]&nA[3]&nA[2]&nA[1]&nA[0],
          B[1]   = nA[4]&nA[3]&nA[2]&nA[1]& A[0],
          B[2]   = nA[4]&nA[3]&nA[2]& A[1]&nA[0],
          B[3]   = nA[4]&nA[3]&nA[2]& A[1]& A[0],
          B[4]   = nA[4]&nA[3]& A[2]&nA[1]&nA[0],
          B[5]   = nA[4]&nA[3]& A[2]&nA[1]& A[0],
          B[6]   = nA[4]&nA[3]& A[2]& A[1]&nA[0],
          B[7]   = nA[4]&nA[3]& A[2]& A[1]& A[0],
          B[8]   = nA[4]& A[3]&nA[2]&nA[1]&nA[0],
          B[9]   = nA[4]& A[3]&nA[2]&nA[1]& A[0],
          B[10]  = nA[4]& A[3]&nA[2]& A[1]&nA[0],
          B[11]  = nA[4]& A[3]&nA[2]& A[1]& A[0],
          B[12]  = nA[4]& A[3]& A[2]&nA[1]&nA[0],
          B[13]  = nA[4]& A[3]& A[2]&nA[1]& A[0],
          B[14]  = nA[4]& A[3]& A[2]& A[1]&nA[0],
          B[15]  = nA[4]& A[3]& A[2]& A[1]& A[0],
          B[16]  =  A[4]&nA[3]&nA[2]&nA[1]&nA[0],
          B[17]  =  A[4]&nA[3]&nA[2]&nA[1]& A[0],
          B[18]  =  A[4]&nA[3]&nA[2]& A[1]&nA[0],
          B[19]  =  A[4]&nA[3]&nA[2]& A[1]& A[0],
          B[20]  =  A[4]&nA[3]& A[2]&nA[1]&nA[0],
          B[21]  =  A[4]&nA[3]& A[2]&nA[1]& A[0],
          B[22]  =  A[4]&nA[3]& A[2]& A[1]&nA[0],
          B[23]  =  A[4]&nA[3]& A[2]& A[1]& A[0],
          B[24]  =  A[4]& A[3]&nA[2]&nA[1]&nA[0],
          B[25]  =  A[4]& A[3]&nA[2]&nA[1]& A[0],
          B[26]  =  A[4]& A[3]&nA[2]& A[1]&nA[0],
          B[27]  =  A[4]& A[3]&nA[2]& A[1]& A[0],
          B[28]  =  A[4]& A[3]& A[2]&nA[1]&nA[0],
          B[29]  =  A[4]& A[3]& A[2]&nA[1]& A[0],
          B[30]  =  A[4]& A[3]& A[2]& A[1]&nA[0],
          B[31]  =  A[4]& A[3]& A[2]& A[1]& A[0];

endmodule


module CONTROL (CLK, RESET, DATA_IN, AFULL, EMPTY,
                grant,
                BroadcastPending,
                WCLK, RCLK,
                request,
                BP,
                DATA_OUT);

   input CLK, RESET;
   input [8:0] DATA_IN;
   input AFULL, EMPTY;
   input [3:0] grant;
   input BroadcastPending;

   output [3:0] request;
   output BP;
   output WCLK, RCLK;
   output [8:0] DATA_OUT;

   reg [8:0] DATA_OUT;
   reg [2:0] PRES_STATE;
   reg [2:0] NEXT_STATE;
   reg [5:0] StateTable;
   reg [8:0] RegRoute;
   wire [3:0] InitialRequest;
   wire CLKbar;
   wire IsRequest, IsGrant, IsClose, IsAbort;
   wire IsRequest_1Port,IsRequest_2Port,IsRequest_4Port;
   wire IsGrant_1Port,IsGrant_2Port,IsGrant_2PortA,IsGrant_2PortB,
        IsGrant_4Port;

   wire ClearRegRoute,LoadRegRoute;
   wire LoadRequest,ClearRequest,ShiftRequest,
        S_All,S_3_2,S_1_0,Shift_3_2,Shift_1_0;

   parameter s0=3'b000;
   parameter s1=3'b001;
   parameter s2=3'b010;
   parameter s3=3'b011;
parameter s4=3'b100;
parameter s5=3'b101;

ClearLoadShiftReg CLSReg (Shift_3_2, Shift_1_0, LoadRequest,
                          ClearRequest, InitialRequest, request);

assign CLKbar=~CLK,
       ClearRegRoute=StateTable[0]&CLK,
       LoadRegRoute=StateTable[1]&CLKbar,
       RCLK=((StateTable[1]|StateTable[5])&
             (~NEXT_STATE[1]&NEXT_STATE[0]))&CLK,
       WCLK=StateTable[5]&CLKbar;

assign IsRequest=(&RegRoute[8:4])&(~RegRoute[3]),
       IsRequest_1Port=IsRequest&(~(RegRoute[2]&RegRoute[1])),
       IsRequest_2Port=IsRequest&RegRoute[2]&
                       RegRoute[1]&(~RegRoute[0]),
       IsRequest_4Port=IsRequest&(&RegRoute[2:0]),
       BP=IsRequest&RegRoute[2]&RegRoute[1];

assign IsGrant_1Port=IsRequest_1Port&(|grant),
       IsGrant_2PortA=IsRequest_2Port&(grant[0]|grant[1]),
       IsGrant_2PortB=IsRequest_2Port&(grant[3]|grant[2]),
       IsGrant_2Port=IsGrant_2PortA&IsGrant_2PortB,
       IsGrant_4Port=(&grant)&IsRequest_4Port,
       IsGrant=IsGrant_1Port|IsGrant_2Port|IsGrant_4Port;

assign IsClose=DATA_OUT[8]&DATA_OUT[7]&DATA_OUT[6]&
               (~DATA_OUT[5])&(~DATA_OUT[4]),
       IsAbort=DATA_OUT[8]&DATA_OUT[7]&DATA_OUT[6]&
               (~DATA_OUT[5])&DATA_OUT[4];

assign ClearRequest=StateTable[0]&CLK,
       LoadRequest=StateTable[2]&CLK,
       ShiftRequest=StateTable[3]&CLK;

assign S_All=(~IsGrant)&IsRequest&RegRoute[2]&(~RegRoute[1]),
       S_3_2=(~IsGrant)&IsRequest_2Port&(~IsGrant_2PortB),
       S_1_0=(~IsGrant)&IsRequest_2Port&(~IsGrant_2PortA),
       Shift_3_2=ShiftRequest&(S_All|S_3_2),
       Shift_1_0=ShiftRequest&(S_All|S_1_0);

assign InitialRequest[0]=IsRequest&((~(RegRoute[1]|RegRoute[0]))
                                    |(RegRoute[2]&RegRoute[1])),
       InitialRequest[1]=IsRequest&(RegRoute[0]&
                                   (RegRoute[1]^~RegRoute[2])),
       InitialRequest[2]=IsRequest&((RegRoute[2]&RegRoute[0])|
                                    (RegRoute[1]&(~RegRoute[0]))),
      InitialRequest[3]=IsRequest&RegRoute[1]&RegRoute[0];

 always @(posedge ClearRegRoute or posedge LoadRegRoute) begin
     if(ClearRegRoute) RegRoute=9'b000000000;
     else RegRoute=DATA_OUT;
     end

 always @(posedge RESET or posedge CLK) begin
    if(RESET==1'b1) PRES_STATE=s0;
    else PRES_STATE=NEXT_STATE;
    end

 always @(posedge RESET or posedge RCLK) begin
   if(RESET)
     DATA_OUT=9'b000000000;
   else
     DATA_OUT=DATA_IN;
   end

 always @(PRES_STATE or BroadcastPending or EMPTY or AFULL or IsRequest or
          IsGrant or IsClose or IsAbort) begin
   case (PRES_STATE)
      s0: begin
          if((EMPTY==1'b0)&&(BroadcastPending==1'b0))
            begin
            StateTable=6'b000001;
            NEXT_STATE=s1;
            end
          else
            begin
            StateTable=6'b000001;
            NEXT_STATE=s0;
            end
          end

      s1: begin
          if(IsRequest==1'b1)
            begin
            StateTable=6'b000010;
            NEXT_STATE=s2;
            end
          else if(EMPTY==1'b1)
            begin
            StateTable=6'b000010;
            NEXT_STATE=s0;
            end
          else
            begin
          StateTable=6'b000010;
          NEXT_STATE=s1;
          end
        end

    s2: begin
        if(IsGrant==1'b1)
          begin
          StateTable=6'b000100;
          NEXT_STATE=s4;
          end
        else
          begin
          StateTable=6'b000100;
          NEXT_STATE=s3;
          end
        end

    s3: begin
        if(IsGrant==1'b1)
          begin
          StateTable=6'b001000;
          NEXT_STATE=s4;
          end
        else
          begin
          StateTable=6'b001000;
          NEXT_STATE=s3;
          end
        end

    s4: begin
        if((EMPTY==1'b0)&&(AFULL==1'b0))
          begin
          StateTable=6'b010000;
          NEXT_STATE=s5;
          end
        else
          begin
          StateTable=6'b010000;
          NEXT_STATE=s4;
          end
        end

    s5: begin
        if((IsClose==1'b1)||(IsAbort==1'b1))
          begin
          StateTable=6'b100000;
```

```
                NEXT_STATE=s0;
                end
              else if((EMPTY==1'b1)||(AFULL==1'b1))
                begin
                StateTable=6'b100000;
                NEXT_STATE=s4;
                end
              else
                begin
                StateTable=6'b100000;
                NEXT_STATE=s5;
                end
              end
      endcase
    end

 endmodule


 module EDGE_C_L_S_DFF (q,d,Shift,Load,Clear,init);

    input d,Shift,Clear,Load,init;
    output q;
    reg q;

    always @(posedge Shift or posedge Load or posedge Clear) begin
       if(Clear) q=1'b0;
       else if (Load) q=init;
       else q=d;
       end
 endmodule


 module ClearLoadShiftReg (Shift_3_2, Shift_1_0, Load, Clear, init, r);
   input Shift_3_2, Shift_1_0, Load, Clear;
   input [3:0] init;
   output [3:0] r;

   EDGE_C_L_S_DFF c_l_s_dff0 (r[0],r[1],Shift_1_0,Load,Clear,init[0]),
                  c_l_s_dff1 (r[1],r[0],Shift_1_0,Load,Clear,init[1]),
                  c_l_s_dff2 (r[2],r[3],Shift_3_2,Load,Clear,init[2]),
                  c_l_s_dff3 (r[3],r[2],Shift_3_2,Load,Clear,init[3]);
endmodule


module ARBITER (CLK, BPENDING, RESET,
                request0, request1, request2, request3,
                grant0, grant1, grant2, grant3);
```

```
input CLK, BPENDING, RESET;
input [3:0] request0, request1, request2, request3;
output [3:0] grant0, grant1, grant2, grant3;

reg [3:0] pW;
wire [3:0] busy0, busy1, busy2, busy3;
wire [3:0] B;
wire [3:0] mux0, mux1, mux2, mux3;
wire [3:0] NBPPW,BPPW;
wire NBPENDING;
reg TEMP;

assign grant0 = request0 & busy0,
       grant1 = request1 & busy1,
       grant2 = request2 & busy2,
       grant3 = request3 & busy3;

assign B[0] = (~grant0[0]) & (~grant1[0]) & (~grant2[0]) & (~grant3[0]),
       B[1] = (~grant0[1]) & (~grant1[1]) & (~grant2[1]) & (~grant3[1]),
       B[2] = (~grant0[2]) & (~grant1[2]) & (~grant2[2]) & (~grant3[2]),
       B[3] = (~grant0[3]) & (~grant1[3]) & (~grant2[3]) & (~grant3[3]);

assign NBPENDING=~BPENDING;

assign NBPPW[0]=NBPENDING&pW[0],
       NBPPW[1]=NBPENDING&pW[1],
       NBPPW[2]=NBPENDING&pW[2],
       NBPPW[3]=NBPENDING&pW[3];

assign BPPW[0]=BPENDING&pW[0],
       BPPW[1]=BPENDING&pW[1],
       BPPW[2]=BPENDING&pW[2],
       BPPW[3]=BPENDING&pW[3];

assign mux0[0]=request0[0]&(((~request1[0])&(~request2[0])&
               (~request3[0]))|NBPPW[3]|BPPW[0]),
       mux1[0]=request1[0]&(((~request0[0])&(~request2[0])&
               (~request3[0]))|NBPPW[0]|BPPW[1]),
       mux2[0]=request2[0]&(((~request0[0])&(~request1[0])&
               (~request3[0]))|NBPPW[1]|BPPW[2]),
       mux3[0]=request3[0]&(((~request0[0])&(~request1[0])&
               (~request2[0]))|NBPPW[2]|BPPW[3]),

       mux0[1]=request0[1]&(((~request1[1])&(~request2[1])&
               (~request3[1]))|NBPPW[2]|BPPW[0]),
       mux1[1]=request1[1]&(((~request0[1])&(~request2[1])&
               (~request3[1]))|NBPPW[3]|BPPW[1]),
              mux2[1]=request2[1]&(((~request0[1])&(~request1[1])&
                      (~request3[1]))|NBPPW[0]|BPPW[2]),
              mux3[1]=request3[1]&(((~request0[1])&(~request1[1])&
                      (~request2[1]))|NBPPW[1]|BPPW[3]),

              mux0[2]=request0[2]&(((~request1[2])&(~request2[2])&
                      (~request3[2]))|NBPPW[1]|BPPW[0]),
              mux1[2]=request1[2]&(((~request0[2])&(~request2[2])&
                      (~request3[2]))|NBPPW[2]|BPPW[1]),
              mux2[2]=request2[2]&(((~request0[2])&(~request1[2])&
                      (~request3[2]))|NBPPW[3]|BPPW[2]),
              mux3[2]=request3[2]&(((~request0[2])&(~request1[2])&
                      (~request2[2]))|NBPPW[0]|BPPW[3]),

              mux0[3]=request0[3]&(((~request1[3])&(~request2[3])&
                      (~request3[3]))|NBPPW[0]|BPPW[0]),
              mux1[3]=request1[3]&(((~request0[3])&(~request2[3])&
                      (~request3[3]))|NBPPW[1]|BPPW[1]),
              mux2[3]=request2[3]&(((~request0[3])&(~request1[3])&
                      (~request3[3]))|NBPPW[2]|BPPW[2]),
              mux3[3]=request3[3]&(((~request0[3])&(~request1[3])&
                      (~request2[3]))|NBPPW[3]|BPPW[3]);

      assign busy0 = (B&mux0)|(~B&busy0),
             busy1 = (B&mux1)|(~B&busy1),
             busy2 = (B&mux2)|(~B&busy2),
             busy3 = (B&mux3)|(~B&busy3);

    always @(posedge RESET or negedge CLK) begin
      if(RESET)
        pW = 4'b0001;
      else begin
        TEMP=pW[3];
        pW[3]=pW[2];
        pW[2]=pW[1];
        pW[1]=pW[0];
        pW[0]=TEMP;
      end
    end
endmodule


module CROSSBAR (DATA_IN0,
                 WCLK_IN0,
                 DATA_IN1,
                 WCLK_IN1,
                 DATA_IN2,
                 WCLK_IN2,
                  DATA_IN3,
                  WCLK_IN3,
                  STOP_IN0, STOP_IN1, STOP_IN2, STOP_IN3,
                  grant0, grant1, grant2, grant3,
                  STOP_OUT0, STOP_OUT1, STOP_OUT2, STOP_OUT3,
                  DATA_OUT0,
                  WCLK_OUT0,
                  DATA_OUT1,
                  WCLK_OUT1,
                  DATA_OUT2,
                  WCLK_OUT2,
                  DATA_OUT3,
                  WCLK_OUT3);

input [8:0] DATA_IN0,DATA_IN1,DATA_IN2,DATA_IN3;
input WCLK_IN0,WCLK_IN1,WCLK_IN2,WCLK_IN3;
input STOP_IN0, STOP_IN1, STOP_IN2, STOP_IN3;
input [3:0] grant0, grant1, grant2, grant3;

output [8:0] DATA_OUT0,DATA_OUT1,DATA_OUT2,DATA_OUT3;
output WCLK_OUT0,WCLK_OUT1,WCLK_OUT2,WCLK_OUT3;
output STOP_OUT0, STOP_OUT1, STOP_OUT2, STOP_OUT3;

tri [8:0] DATA_OUT0,DATA_OUT1,DATA_OUT2,DATA_OUT3;
wire WCLK_OUT0,WCLK_OUT1,WCLK_OUT2,WCLK_OUT3;
wire STOP_OUT0, STOP_OUT1, STOP_OUT2, STOP_OUT3;

wire [3:0] SOUT;
wire [3:0] RGIR;

assign DATA_OUT0 = (grant0[0]==1'b1) ? DATA_IN0 : 9'bzzzzzzzzz,
       DATA_OUT0 = (grant1[0]==1'b1) ? DATA_IN1 : 9'bzzzzzzzzz,
       DATA_OUT0 = (grant2[0]==1'b1) ? DATA_IN2 : 9'bzzzzzzzzz,
       DATA_OUT0 = (grant3[0]==1'b1) ? DATA_IN3 : 9'bzzzzzzzzz,

       DATA_OUT1 = (grant0[1]==1'b1) ? DATA_IN0 : 9'bzzzzzzzzz,
       DATA_OUT1 = (grant1[1]==1'b1) ? DATA_IN1 : 9'bzzzzzzzzz,
       DATA_OUT1 = (grant2[1]==1'b1) ? DATA_IN2 : 9'bzzzzzzzzz,
       DATA_OUT1 = (grant3[1]==1'b1) ? DATA_IN3 : 9'bzzzzzzzzz,

       DATA_OUT2 = (grant0[2]==1'b1) ? DATA_IN0 : 9'bzzzzzzzzz,
       DATA_OUT2 = (grant1[2]==1'b1) ? DATA_IN1 : 9'bzzzzzzzzz,
       DATA_OUT2 = (grant2[2]==1'b1) ? DATA_IN2 : 9'bzzzzzzzzz,
       DATA_OUT2 = (grant3[2]==1'b1) ? DATA_IN3 : 9'bzzzzzzzzz,

       DATA_OUT3 = (grant0[3]==1'b1) ? DATA_IN0 : 9'bzzzzzzzzz,
       DATA_OUT3 = (grant1[3]==1'b1) ? DATA_IN1 : 9'bzzzzzzzzz,
       DATA_OUT3 = (grant2[3]==1'b1) ? DATA_IN2 : 9'bzzzzzzzzz,
          DATA_OUT3 = (grant3[3]==1'b1) ? DATA_IN3 : 9'bzzzzzzzzz;

  assign WCLK_OUT0 =(WCLK_IN0&grant0[0])|(WCLK_IN1&grant1[0])|
                    (WCLK_IN2&grant2[0])|(WCLK_IN3&grant3[0]),
         WCLK_OUT1 =(WCLK_IN0&grant0[1])|(WCLK_IN1&grant1[1])|
                    (WCLK_IN2&grant2[1])|(WCLK_IN3&grant3[1]),
         WCLK_OUT2 =(WCLK_IN0&grant0[2])|(WCLK_IN1&grant1[2])|
                    (WCLK_IN2&grant2[2])|(WCLK_IN3&grant3[2]),
         WCLK_OUT3 =(WCLK_IN0&grant0[3])|(WCLK_IN1&grant1[3])|
                    (WCLK_IN2&grant2[3])|(WCLK_IN3&grant3[3]);

  assign SOUT[0]=(STOP_IN0&grant0[0])|(STOP_IN1&grant0[1])|
                 (STOP_IN2&grant0[2])|(STOP_IN3&grant0[3]),
         SOUT[1]=(STOP_IN0&grant1[0])|(STOP_IN1&grant1[1])|
                 (STOP_IN2&grant1[2])|(STOP_IN3&grant1[3]),
         SOUT[2]=(STOP_IN0&grant2[0])|(STOP_IN1&grant2[1])|
                 (STOP_IN2&grant2[2])|(STOP_IN3&grant2[3]),
         SOUT[3]=(STOP_IN0&grant3[0])|(STOP_IN1&grant3[1])|
                 (STOP_IN2&grant3[2])|(STOP_IN3&grant3[3]);

  assign RGIR[0] = |grant0,
         RGIR[1] = |grant1,
         RGIR[2] = |grant2,
         RGIR[3] = |grant3;

  assign STOP_OUT0 = RGIR[0] & SOUT[0],
         STOP_OUT1 = RGIR[1] & SOUT[1],
         STOP_OUT2 = RGIR[2] & SOUT[2],
         STOP_OUT3 = RGIR[3] & SOUT[3];

endmodule
```

## A.2 Network Interface

```
module Network_Interface (CLK,RESET,DATA,CS,DATA_TYPE,WRITE_READ,STATUS,
                          DATA_OUT,SEND_CLK,SEND_STOP,
                          DATA_IN,RECEIVE_CLK,RECEIVE_STOP);

  inout [71:0] DATA;
  input CLK,RESET,CS,WRITE_READ,SEND_STOP,RECEIVE_CLK;
  input [1:0] DATA_TYPE;
  input [8:0] DATA_IN;

  output SEND_CLK;
  output RECEIVE_STOP;
  output [8:0] DATA_OUT;
  output [11:0] STATUS;

  wire [71:0] DTSF,DFRF;
  wire SEND_EMPTY,SEND_FULL,SEND_2,SEND_4,SEND_8;
  wire RECEIVE_EMPTY,RECEIVE_FULL,RECEIVE_8,RECEIVE_16,RECEIVE_24;
  wire DAHOD,CAHOD;

  SEND_FIFO  sf (CLK, RESET, DTSF, CS, DATA_TYPE, WRITE_READ,
                 SEND_EMPTY, SEND_FULL, SEND_2, SEND_4, SEND_8,
                 DATA_OUT, SEND_CLK, SEND_STOP);

  RECEIVE_FIFO rf (CLK,RESET,DFRF,CS,DATA_TYPE,WRITE_READ,
                   RECEIVE_EMPTY,RECEIVE_FULL,
                   RECEIVE_8,RECEIVE_16,RECEIVE_24,
                   DAHOD,CAHOD,
                   DATA_IN,RECEIVE_CLK);

  assign STATUS = {SEND_2,SEND_4,SEND_8,SEND_EMPTY,SEND_FULL,
                   RECEIVE_8,RECEIVE_16,RECEIVE_24,
                   RECEIVE_EMPTY,RECEIVE_FULL,
                   DAHOD,CAHOD};

  assign RECEIVE_STOP = {RECEIVE_FULL};

  assign DATA = (CS==1'b1 && WRITE_READ==1'b0) ? DFRF : 72'Hzzzzzzzzz;

  assign DTSF = (CS==1'b1 && WRITE_READ==1'b1) ? DATA : 72'Hzzzzzzzzz;

endmodule


module SEND_FIFO (CLK, RESET, DATA, CS, DATA_TYPE, WRITE_READ,
                  SEND_EMPTY, SEND_FULL, SEND_2, SEND_4, SEND_8,
                  DATA_OUT, SEND_CLK, SEND_STOP);
```

```
input CLK, RESET;
input [71:0] DATA;
input CS;
input [1:0] DATA_TYPE;
input WRITE_READ;

output SEND_EMPTY, SEND_FULL, SEND_2, SEND_4, SEND_8;
output [8:0] DATA_OUT;
output SEND_CLK;
input SEND_STOP;

wire SEND_EMPTY, SEND_FULL, SEND_2, SEND_4, SEND_8;
wire [8:0] DATA_OUT;

reg SEND_CLK;

/* On rising edge of TAIL_INCREMENT, TAIL is incremented by 2, 4 or 8 */
/* depends on DATA_TYPE.                                               */
wire TAIL_INCREMENT;

/* On the rising edge of TAIL_HEAD_RESET, both HEAD and TAIL are */
/* reset to 0.                                                   */
wire TAIL_HEAD_RESET;

/* On the rising edge of HEAD_INCREMENT, HEAD is incremented by 1. */
wire HEAD_INCREMENT;

wire [5:0] B,S,H;
wire [5:0] TP_1, TP_2, TP_3, TP_4, TP_5, TP_6, TP_7;
reg [5:0] TAIL, HEAD;
reg [5:0] TP,HP;
wire NSE;

SF_MEMORYBANK sfb (CLK,CS,WRITE_READ,DATA_TYPE,
                   TP,TP_1,TP_2,TP_3,TP_4,TP_5,TP_6,TP_7,
                   HP,DATA,DATA_OUT);

SF_STATUS sfs (HEAD, TAIL, SEND_EMPTY, SEND_FULL, SEND_2, SEND_4, SEND_8);

SF_ADDER_6B add8_0 (TAIL,B,S,),
            add8_1 (TP,6'b000001,TP_1,),
            add8_2 (TP,6'b000010,TP_2,),
            add8_3 (TP,6'b000011,TP_3,),
            add8_4 (TP,6'b000100,TP_4,),
            add8_5 (TP,6'b000101,TP_5,),
            add8_6 (TP,6'b000110,TP_6,),
            add8_7 (TP,6'b000111,TP_7,),
               add8_8 (HEAD,6'b000001,H,);

  always @(CLK) begin
    if(CLK)
      SEND_CLK=0;
    else begin
      if(NSE)
        SEND_CLK=1;
      else
        SEND_CLK=0;
    end
  end

  assign NSE = ~(SEND_EMPTY | SEND_STOP);
  assign TAIL_INCREMENT = CS & WRITE_READ & (|DATA_TYPE);
  assign TAIL_HEAD_RESET = CS & WRITE_READ & (~(|DATA_TYPE));
  assign HEAD_INCREMENT = NSE & (|DATA_TYPE);

  assign B[5] = 0,
         B[4] = 0,
         B[3] = &DATA_TYPE,
         B[2] = DATA_TYPE[1]&(~DATA_TYPE[0]),
         B[1] = (~DATA_TYPE[1])&DATA_TYPE[0],
         B[0] = 0;

  always @(posedge RESET or negedge CLK) begin
    if(RESET)
      TP = 0;
    else
      TP = TAIL;
  end

  always @(posedge RESET or posedge CLK) begin
    if(RESET)
      TAIL = 0;
    else begin
      if(TAIL_HEAD_RESET)
        TAIL = HEAD;
      else if(TAIL_INCREMENT)
        TAIL = S;
      else
        TAIL = TAIL;
    end
  end

  always @(posedge RESET or posedge CLK) begin
    if(RESET)
      HP = 0;
      else
        HP = HEAD;
    end

    always @(posedge RESET or negedge CLK) begin
      if(RESET)
        HEAD = 0;
      else begin
        if(NSE)
          HEAD = H;
        else
          HEAD = HEAD;
      end
    end
endmodule


module SF_STATUS(HEAD, TAIL, SEND_EMPTY, SEND_FULL, SEND_2, SEND_4, SEND_8);

   input [5:0] HEAD, TAIL;
   output SEND_EMPTY, SEND_FULL, SEND_2, SEND_4, SEND_8;

   wire sfa,sfb;
   wire s2a,s2b,s2c,s2d;
   wire s4a,s4b,s4c,s4d;
   wire s8a,s8b,s8c,s8d;
   wire TGTH,TLTH;

   assign SEND_FULL = (TGTH&sfa) | (TLTH&sfb),
          SEND_2 = (TGTH&(s2a|s2b))|(TLTH&(s2c|s2d)),
          SEND_4 = (TGTH&(s4a|s4b))|(TLTH&(s4c|s4d)),
          SEND_8 = (TGTH&(s8a|s8b))|(TLTH&(s8c|s8d));

   SF_COMPARE_UNIT comm0 (HEAD,6'b111111,TAIL,,,sfa);
   SF_COMPARE_UNIT comm1 (TAIL,6'b000001,HEAD,,,sfb);
   SF_COMPARE_UNIT comm2 (HEAD,6'b111110,TAIL,,s2a,s2b);
   SF_COMPARE_UNIT comm3 (TAIL,6'b000010,HEAD,s2c,,s2d);
   SF_COMPARE_UNIT comm4 (HEAD,6'b111100,TAIL,,s4a,s4b);
   SF_COMPARE_UNIT comm5 (TAIL,6'b000100,HEAD,s4c,,s4d);
   SF_COMPARE_UNIT comm6 (HEAD,6'b111000,TAIL,,s8a,s8b);
   SF_COMPARE_UNIT comm7 (TAIL,6'b001000,HEAD,s8c,,s8d);
   SF_COMPARATOR_6B comm8 (TAIL,HEAD,TGTH,TLTH,SEND_EMPTY);

endmodule


module SF_MEMORYCELL(CLK,TPEQ_0,TPEQ_1,TPEQ_2,TPEQ_3,TPEQ_4,
                     TPEQ_5,TPEQ_6,TPEQ_7,HPEQ,DATA,DOUT);
   input CLK,TPEQ_0,TPEQ_1,TPEQ_2,TPEQ_3,TPEQ_4,TPEQ_5,TPEQ_6,TPEQ_7,HPEQ;
   input [71:0] DATA;
   output [8:0] DOUT;

   reg [8:0] MEM;

   assign DOUT=(HPEQ==1'b1) ? MEM : 9'bzzzzzzzzz;

   always @(posedge CLK) begin
      if(TPEQ_0)
         MEM=DATA[8:0];
       else if(TPEQ_1)
         MEM=DATA[17:9];
       else if(TPEQ_2)
         MEM=DATA[26:18];
       else if(TPEQ_3)
         MEM=DATA[35:27];
       else if(TPEQ_4)
         MEM=DATA[44:36];
       else if(TPEQ_5)
         MEM=DATA[53:45];
       else if(TPEQ_6)
         MEM=DATA[62:54];
       else if(TPEQ_7)
         MEM=DATA[71:63];
  end

endmodule


module SF_MEMORYBANK(CLK,CS,WRITE_READ,DATA_TYPE,
                     TP,TP_1,TP_2,TP_3,TP_4,TP_5,TP_6,TP_7,
                     HP,DATA,DOUT);

  input CLK,CS,WRITE_READ;
  input [1:0] DATA_TYPE;
  input [5:0] TP,TP_1,TP_2,TP_3,TP_4,TP_5,TP_6,TP_7;
  input [71:0] DATA;
  input [5:0] HP;
  output [8:0] DOUT;

  tri [8:0] DOUT;
  wire [63:0] TPEQ_0,TPEQ_1,TPEQ_2,TPEQ_3,TPEQ_4,TPEQ_5,TPEQ_6,TPEQ_7,HPEQ;

  wire G2,G4,G8;

  assign G2=CS&WRITE_READ&(|DATA_TYPE),
        G4=CS&WRITE_READ&(DATA_TYPE[1]),
        G8=CS&WRITE_READ&(&DATA_TYPE);

 SF_MUX6X64 sfm0 (HP,HPEQ);

 SF_MUX6X64_EN sfm_en0 (TP,  TPEQ_0,G2),
               sfm_en1 (TP_1,TPEQ_1,G2),
               sfm_en2 (TP_2,TPEQ_2,G4),
               sfm_en3 (TP_3,TPEQ_3,G4),
               sfm_en4 (TP_4,TPEQ_4,G8),
               sfm_en5 (TP_5,TPEQ_5,G8),
               sfm_en6 (TP_6,TPEQ_6,G8),
               sfm_en7 (TP_7,TPEQ_7,G8);

 SF_MEMORYCELL sfmc0 (CLK,TPEQ_0[0],TPEQ_1[0],TPEQ_2[0],TPEQ_3[0],
                      TPEQ_4[0],TPEQ_5[0],TPEQ_6[0],TPEQ_7[0],
                      HPEQ[0],DATA,DOUT);
 SF_MEMORYCELL sfmc1 (CLK,TPEQ_0[1],TPEQ_1[1],TPEQ_2[1],TPEQ_3[1],
                      TPEQ_4[1],TPEQ_5[1],TPEQ_6[1],TPEQ_7[1],
                      HPEQ[1],DATA,DOUT);
 SF_MEMORYCELL sfmc2 (CLK,TPEQ_0[2],TPEQ_1[2],TPEQ_2[2],TPEQ_3[2],
                      TPEQ_4[2],TPEQ_5[2],TPEQ_6[2],TPEQ_7[2],
                      HPEQ[2],DATA,DOUT);
 SF_MEMORYCELL sfmc3 (CLK,TPEQ_0[3],TPEQ_1[3],TPEQ_2[3],TPEQ_3[3],
                      TPEQ_4[3],TPEQ_5[3],TPEQ_6[3],TPEQ_7[3],
                      HPEQ[3],DATA,DOUT);
 SF_MEMORYCELL sfmc4 (CLK,TPEQ_0[4],TPEQ_1[4],TPEQ_2[4],TPEQ_3[4],
                      TPEQ_4[4],TPEQ_5[4],TPEQ_6[4],TPEQ_7[4],
                      HPEQ[4],DATA,DOUT);
 SF_MEMORYCELL sfmc5 (CLK,TPEQ_0[5],TPEQ_1[5],TPEQ_2[5],TPEQ_3[5],
                      TPEQ_4[5],TPEQ_5[5],TPEQ_6[5],TPEQ_7[5],
                      HPEQ[5],DATA,DOUT);
 SF_MEMORYCELL sfmc6 (CLK,TPEQ_0[6],TPEQ_1[6],TPEQ_2[6],TPEQ_3[6],
                      TPEQ_4[6],TPEQ_5[6],TPEQ_6[6],TPEQ_7[6],
                      HPEQ[6],DATA,DOUT);
 SF_MEMORYCELL sfmc7 (CLK,TPEQ_0[7],TPEQ_1[7],TPEQ_2[7],TPEQ_3[7],
                      TPEQ_4[7],TPEQ_5[7],TPEQ_6[7],TPEQ_7[7],
                      HPEQ[7],DATA,DOUT);
 SF_MEMORYCELL sfmc8 (CLK,TPEQ_0[8],TPEQ_1[8],TPEQ_2[8],TPEQ_3[8],
                      TPEQ_4[8],TPEQ_5[8],TPEQ_6[8],TPEQ_7[8],
                      HPEQ[8],DATA,DOUT);
 SF_MEMORYCELL sfmc9 (CLK,TPEQ_0[9],TPEQ_1[9],TPEQ_2[9],TPEQ_3[9],
                      TPEQ_4[9],TPEQ_5[9],TPEQ_6[9],TPEQ_7[9],
                      HPEQ[9],DATA,DOUT);
 SF_MEMORYCELL sfmc10 (CLK,TPEQ_0[10],TPEQ_1[10],TPEQ_2[10],TPEQ_3[10],
                      TPEQ_4[10],TPEQ_5[10],TPEQ_6[10],TPEQ_7[10],
                      HPEQ[10],DATA,DOUT);
 SF_MEMORYCELL sfmc11 (CLK,TPEQ_0[11],TPEQ_1[11],TPEQ_2[11],TPEQ_3[11],
                      TPEQ_4[11],TPEQ_5[11],TPEQ_6[11],TPEQ_7[11],
                      HPEQ[11],DATA,DOUT);
  SF_MEMORYCELL sfmc12 (CLK,TPEQ_0[12],TPEQ_1[12],TPEQ_2[12],TPEQ_3[12],
                      TPEQ_4[12],TPEQ_5[12],TPEQ_6[12],TPEQ_7[12],
                      HPEQ[12],DATA,DOUT);
  SF_MEMORYCELL sfmc13 (CLK,TPEQ_0[13],TPEQ_1[13],TPEQ_2[13],TPEQ_3[13],
                      TPEQ_4[13],TPEQ_5[13],TPEQ_6[13],TPEQ_7[13],
                      HPEQ[13],DATA,DOUT);
  SF_MEMORYCELL sfmc14 (CLK,TPEQ_0[14],TPEQ_1[14],TPEQ_2[14],TPEQ_3[14],
                      TPEQ_4[14],TPEQ_5[14],TPEQ_6[14],TPEQ_7[14],
                      HPEQ[14],DATA,DOUT);
  SF_MEMORYCELL sfmc15 (CLK,TPEQ_0[15],TPEQ_1[15],TPEQ_2[15],TPEQ_3[15],
                      TPEQ_4[15],TPEQ_5[15],TPEQ_6[15],TPEQ_7[15],
                      HPEQ[15],DATA,DOUT);
  SF_MEMORYCELL sfmc16 (CLK,TPEQ_0[16],TPEQ_1[16],TPEQ_2[16],TPEQ_3[16],
                      TPEQ_4[16],TPEQ_5[16],TPEQ_6[16],TPEQ_7[16],
                      HPEQ[16],DATA,DOUT);
  SF_MEMORYCELL sfmc17 (CLK,TPEQ_0[17],TPEQ_1[17],TPEQ_2[17],TPEQ_3[17],
                      TPEQ_4[17],TPEQ_5[17],TPEQ_6[17],TPEQ_7[17],
                      HPEQ[17],DATA,DOUT);
  SF_MEMORYCELL sfmc18 (CLK,TPEQ_0[18],TPEQ_1[18],TPEQ_2[18],TPEQ_3[18],
                      TPEQ_4[18],TPEQ_5[18],TPEQ_6[18],TPEQ_7[18],
                      HPEQ[18],DATA,DOUT);
  SF_MEMORYCELL sfmc19 (CLK,TPEQ_0[19],TPEQ_1[19],TPEQ_2[19],TPEQ_3[19],
                      TPEQ_4[19],TPEQ_5[19],TPEQ_6[19],TPEQ_7[19],
                      HPEQ[19],DATA,DOUT);
  SF_MEMORYCELL sfmc20 (CLK,TPEQ_0[20],TPEQ_1[20],TPEQ_2[20],TPEQ_3[20],
                      TPEQ_4[20],TPEQ_5[20],TPEQ_6[20],TPEQ_7[20],
                      HPEQ[20],DATA,DOUT);
  SF_MEMORYCELL sfmc21 (CLK,TPEQ_0[21],TPEQ_1[21],TPEQ_2[21],TPEQ_3[21],
                      TPEQ_4[21],TPEQ_5[21],TPEQ_6[21],TPEQ_7[21],
                      HPEQ[21],DATA,DOUT);
  SF_MEMORYCELL sfmc22 (CLK,TPEQ_0[22],TPEQ_1[22],TPEQ_2[22],TPEQ_3[22],
                      TPEQ_4[22],TPEQ_5[22],TPEQ_6[22],TPEQ_7[22],
                      HPEQ[22],DATA,DOUT);
  SF_MEMORYCELL sfmc23 (CLK,TPEQ_0[23],TPEQ_1[23],TPEQ_2[23],TPEQ_3[23],
                      TPEQ_4[23],TPEQ_5[23],TPEQ_6[23],TPEQ_7[23],
                      HPEQ[23],DATA,DOUT);
  SF_MEMORYCELL sfmc24 (CLK,TPEQ_0[24],TPEQ_1[24],TPEQ_2[24],TPEQ_3[24],
                      TPEQ_4[24],TPEQ_5[24],TPEQ_6[24],TPEQ_7[24],
                      HPEQ[24],DATA,DOUT);
  SF_MEMORYCELL sfmc25 (CLK,TPEQ_0[25],TPEQ_1[25],TPEQ_2[25],TPEQ_3[25],
                      TPEQ_4[25],TPEQ_5[25],TPEQ_6[25],TPEQ_7[25],
                      HPEQ[25],DATA,DOUT);
  SF_MEMORYCELL sfmc26 (CLK,TPEQ_0[26],TPEQ_1[26],TPEQ_2[26],TPEQ_3[26],
                      TPEQ_4[26],TPEQ_5[26],TPEQ_6[26],TPEQ_7[26],
                      HPEQ[26],DATA,DOUT);
  SF_MEMORYCELL sfmc27 (CLK,TPEQ_0[27],TPEQ_1[27],TPEQ_2[27],TPEQ_3[27],
                    TPEQ_4[27],TPEQ_5[27],TPEQ_6[27],TPEQ_7[27],
                    HPEQ[27],DATA,DOUT);
  SF_MEMORYCELL sfmc28 (CLK,TPEQ_0[28],TPEQ_1[28],TPEQ_2[28],TPEQ_3[28],
                    TPEQ_4[28],TPEQ_5[28],TPEQ_6[28],TPEQ_7[28],
                    HPEQ[28],DATA,DOUT);
  SF_MEMORYCELL sfmc29 (CLK,TPEQ_0[29],TPEQ_1[29],TPEQ_2[29],TPEQ_3[29],
                    TPEQ_4[29],TPEQ_5[29],TPEQ_6[29],TPEQ_7[29],
                    HPEQ[29],DATA,DOUT);
  SF_MEMORYCELL sfmc30 (CLK,TPEQ_0[30],TPEQ_1[30],TPEQ_2[30],TPEQ_3[30],
                    TPEQ_4[30],TPEQ_5[30],TPEQ_6[30],TPEQ_7[30],
                    HPEQ[30],DATA,DOUT);
  SF_MEMORYCELL sfmc31 (CLK,TPEQ_0[31],TPEQ_1[31],TPEQ_2[31],TPEQ_3[31],
                    TPEQ_4[31],TPEQ_5[31],TPEQ_6[31],TPEQ_7[31],
                    HPEQ[31],DATA,DOUT);
  SF_MEMORYCELL sfmc32 (CLK,TPEQ_0[32],TPEQ_1[32],TPEQ_2[32],TPEQ_3[32],
                    TPEQ_4[32],TPEQ_5[32],TPEQ_6[32],TPEQ_7[32],
                    HPEQ[32],DATA,DOUT);
  SF_MEMORYCELL sfmc33 (CLK,TPEQ_0[33],TPEQ_1[33],TPEQ_2[33],TPEQ_3[33],
                    TPEQ_4[33],TPEQ_5[33],TPEQ_6[33],TPEQ_7[33],
                    HPEQ[33],DATA,DOUT);
  SF_MEMORYCELL sfmc34 (CLK,TPEQ_0[34],TPEQ_1[34],TPEQ_2[34],TPEQ_3[34],
                    TPEQ_4[34],TPEQ_5[34],TPEQ_6[34],TPEQ_7[34],
                    HPEQ[34],DATA,DOUT);
  SF_MEMORYCELL sfmc35 (CLK,TPEQ_0[35],TPEQ_1[35],TPEQ_2[35],TPEQ_3[35],
                    TPEQ_4[35],TPEQ_5[35],TPEQ_6[35],TPEQ_7[35],
                    HPEQ[35],DATA,DOUT);
  SF_MEMORYCELL sfmc36 (CLK,TPEQ_0[36],TPEQ_1[36],TPEQ_2[36],TPEQ_3[36],
                    TPEQ_4[36],TPEQ_5[36],TPEQ_6[36],TPEQ_7[36],
                    HPEQ[36],DATA,DOUT);
  SF_MEMORYCELL sfmc37 (CLK,TPEQ_0[37],TPEQ_1[37],TPEQ_2[37],TPEQ_3[37],
                    TPEQ_4[37],TPEQ_5[37],TPEQ_6[37],TPEQ_7[37],
                    HPEQ[37],DATA,DOUT);
  SF_MEMORYCELL sfmc38 (CLK,TPEQ_0[38],TPEQ_1[38],TPEQ_2[38],TPEQ_3[38],
                    TPEQ_4[38],TPEQ_5[38],TPEQ_6[38],TPEQ_7[38],
                    HPEQ[38],DATA,DOUT);
  SF_MEMORYCELL sfmc39 (CLK,TPEQ_0[39],TPEQ_1[39],TPEQ_2[39],TPEQ_3[39],
                    TPEQ_4[39],TPEQ_5[39],TPEQ_6[39],TPEQ_7[39],
                    HPEQ[39],DATA,DOUT);
  SF_MEMORYCELL sfmc40 (CLK,TPEQ_0[40],TPEQ_1[40],TPEQ_2[40],TPEQ_3[40],
                    TPEQ_4[40],TPEQ_5[40],TPEQ_6[40],TPEQ_7[40],
                    HPEQ[40],DATA,DOUT);
  SF_MEMORYCELL sfmc41 (CLK,TPEQ_0[41],TPEQ_1[41],TPEQ_2[41],TPEQ_3[41],
                    TPEQ_4[41],TPEQ_5[41],TPEQ_6[41],TPEQ_7[41],
                    HPEQ[41],DATA,DOUT);
  SF_MEMORYCELL sfmc42 (CLK,TPEQ_0[42],TPEQ_1[42],TPEQ_2[42],TPEQ_3[42],
                    TPEQ_4[42],TPEQ_5[42],TPEQ_6[42],TPEQ_7[42],
                    HPEQ[42],DATA,DOUT);
  SF_MEMORYCELL sfmc43 (CLK,TPEQ_0[43],TPEQ_1[43],TPEQ_2[43],TPEQ_3[43],
                      TPEQ_4[43],TPEQ_5[43],TPEQ_6[43],TPEQ_7[43],
                      HPEQ[43],DATA,DOUT);
  SF_MEMORYCELL sfmc44 (CLK,TPEQ_0[44],TPEQ_1[44],TPEQ_2[44],TPEQ_3[44],
                      TPEQ_4[44],TPEQ_5[44],TPEQ_6[44],TPEQ_7[44],
                      HPEQ[44],DATA,DOUT);
  SF_MEMORYCELL sfmc45 (CLK,TPEQ_0[45],TPEQ_1[45],TPEQ_2[45],TPEQ_3[45],
                      TPEQ_4[45],TPEQ_5[45],TPEQ_6[45],TPEQ_7[45],
                      HPEQ[45],DATA,DOUT);
  SF_MEMORYCELL sfmc46 (CLK,TPEQ_0[46],TPEQ_1[46],TPEQ_2[46],TPEQ_3[46],
                      TPEQ_4[46],TPEQ_5[46],TPEQ_6[46],TPEQ_7[46],
                      HPEQ[46],DATA,DOUT);
  SF_MEMORYCELL sfmc47 (CLK,TPEQ_0[47],TPEQ_1[47],TPEQ_2[47],TPEQ_3[47],
                      TPEQ_4[47],TPEQ_5[47],TPEQ_6[47],TPEQ_7[47],
                      HPEQ[47],DATA,DOUT);
  SF_MEMORYCELL sfmc48 (CLK,TPEQ_0[48],TPEQ_1[48],TPEQ_2[48],TPEQ_3[48],
                      TPEQ_4[48],TPEQ_5[48],TPEQ_6[48],TPEQ_7[48],
                      HPEQ[48],DATA,DOUT);
  SF_MEMORYCELL sfmc49 (CLK,TPEQ_0[49],TPEQ_1[49],TPEQ_2[49],TPEQ_3[49],
                      TPEQ_4[49],TPEQ_5[49],TPEQ_6[49],TPEQ_7[49],
                      HPEQ[49],DATA,DOUT);
  SF_MEMORYCELL sfmc50 (CLK,TPEQ_0[50],TPEQ_1[50],TPEQ_2[50],TPEQ_3[50],
                      TPEQ_4[50],TPEQ_5[50],TPEQ_6[50],TPEQ_7[50],
                      HPEQ[50],DATA,DOUT);
  SF_MEMORYCELL sfmc51 (CLK,TPEQ_0[51],TPEQ_1[51],TPEQ_2[51],TPEQ_3[51],
                      TPEQ_4[51],TPEQ_5[51],TPEQ_6[51],TPEQ_7[51],
                      HPEQ[51],DATA,DOUT);
  SF_MEMORYCELL sfmc52 (CLK,TPEQ_0[52],TPEQ_1[52],TPEQ_2[52],TPEQ_3[52],
                      TPEQ_4[52],TPEQ_5[52],TPEQ_6[52],TPEQ_7[52],
                      HPEQ[52],DATA,DOUT);
  SF_MEMORYCELL sfmc53 (CLK,TPEQ_0[53],TPEQ_1[53],TPEQ_2[53],TPEQ_3[53],
                      TPEQ_4[53],TPEQ_5[53],TPEQ_6[53],TPEQ_7[53],
                      HPEQ[53],DATA,DOUT);
  SF_MEMORYCELL sfmc54 (CLK,TPEQ_0[54],TPEQ_1[54],TPEQ_2[54],TPEQ_3[54],
                      TPEQ_4[54],TPEQ_5[54],TPEQ_6[54],TPEQ_7[54],
                      HPEQ[54],DATA,DOUT);
  SF_MEMORYCELL sfmc55 (CLK,TPEQ_0[55],TPEQ_1[55],TPEQ_2[55],TPEQ_3[55],
                      TPEQ_4[55],TPEQ_5[55],TPEQ_6[55],TPEQ_7[55],
                      HPEQ[55],DATA,DOUT);
  SF_MEMORYCELL sfmc56 (CLK,TPEQ_0[56],TPEQ_1[56],TPEQ_2[56],TPEQ_3[56],
                      TPEQ_4[56],TPEQ_5[56],TPEQ_6[56],TPEQ_7[56],
                      HPEQ[56],DATA,DOUT);
  SF_MEMORYCELL sfmc57 (CLK,TPEQ_0[57],TPEQ_1[57],TPEQ_2[57],TPEQ_3[57],
                      TPEQ_4[57],TPEQ_5[57],TPEQ_6[57],TPEQ_7[57],
                      HPEQ[57],DATA,DOUT);
  SF_MEMORYCELL sfmc58 (CLK,TPEQ_0[58],TPEQ_1[58],TPEQ_2[58],TPEQ_3[58],
                      TPEQ_4[58],TPEQ_5[58],TPEQ_6[58],TPEQ_7[58],
                      HPEQ[58],DATA,DOUT);
  SF_MEMORYCELL sfmc59 (CLK,TPEQ_0[59],TPEQ_1[59],TPEQ_2[59],TPEQ_3[59],
                         TPEQ_4[59],TPEQ_5[59],TPEQ_6[59],TPEQ_7[59],
                         HPEQ[59],DATA,DOUT);
  SF_MEMORYCELL sfmc60 (CLK,TPEQ_0[60],TPEQ_1[60],TPEQ_2[60],TPEQ_3[60],
                         TPEQ_4[60],TPEQ_5[60],TPEQ_6[60],TPEQ_7[60],
                         HPEQ[60],DATA,DOUT);
  SF_MEMORYCELL sfmc61 (CLK,TPEQ_0[61],TPEQ_1[61],TPEQ_2[61],TPEQ_3[61],
                         TPEQ_4[61],TPEQ_5[61],TPEQ_6[61],TPEQ_7[61],
                         HPEQ[61],DATA,DOUT);
  SF_MEMORYCELL sfmc62 (CLK,TPEQ_0[62],TPEQ_1[62],TPEQ_2[62],TPEQ_3[62],
                         TPEQ_4[62],TPEQ_5[62],TPEQ_6[62],TPEQ_7[62],
                         HPEQ[62],DATA,DOUT);
  SF_MEMORYCELL sfmc63 (CLK,TPEQ_0[63],TPEQ_1[63],TPEQ_2[63],TPEQ_3[63],
                         TPEQ_4[63],TPEQ_5[63],TPEQ_6[63],TPEQ_7[63],
                         HPEQ[63],DATA,DOUT);
endmodule


module SF_MUX6X64 (A,B);

  input [5:0] A;
  output [63:0] B;

  wire [5:0] nA;

  assign nA = ~A;

  assign B[0]  = nA[5]&nA[4]&nA[3]&nA[2]&nA[1]&nA[0],
         B[1]  = nA[5]&nA[4]&nA[3]&nA[2]&nA[1]& A[0],
         B[2]  = nA[5]&nA[4]&nA[3]&nA[2]& A[1]&nA[0],
         B[3]  = nA[5]&nA[4]&nA[3]&nA[2]& A[1]& A[0],
         B[4]  = nA[5]&nA[4]&nA[3]& A[2]&nA[1]&nA[0],
         B[5]  = nA[5]&nA[4]&nA[3]& A[2]&nA[1]& A[0],
         B[6]  = nA[5]&nA[4]&nA[3]& A[2]& A[1]&nA[0],
         B[7]  = nA[5]&nA[4]&nA[3]& A[2]& A[1]& A[0],
         B[8]  = nA[5]&nA[4]& A[3]&nA[2]&nA[1]&nA[0],
         B[9]  = nA[5]&nA[4]& A[3]&nA[2]&nA[1]& A[0],
         B[10] = nA[5]&nA[4]& A[3]&nA[2]& A[1]&nA[0],
         B[11] = nA[5]&nA[4]& A[3]&nA[2]& A[1]& A[0],
         B[12] = nA[5]&nA[4]& A[3]& A[2]&nA[1]&nA[0],
         B[13] = nA[5]&nA[4]& A[3]& A[2]&nA[1]& A[0],
         B[14] = nA[5]&nA[4]& A[3]& A[2]& A[1]&nA[0],
         B[15] = nA[5]&nA[4]& A[3]& A[2]& A[1]& A[0],
         B[16] = nA[5]& A[4]&nA[3]&nA[2]&nA[1]&nA[0],
         B[17] = nA[5]& A[4]&nA[3]&nA[2]&nA[1]& A[0],
         B[18] = nA[5]& A[4]&nA[3]&nA[2]& A[1]&nA[0],
         B[19] = nA[5]& A[4]&nA[3]&nA[2]& A[1]& A[0],
         B[20] = nA[5]& A[4]&nA[3]& A[2]&nA[1]&nA[0],
         B[21] = nA[5]& A[4]&nA[3]& A[2]&nA[1]& A[0],
          B[22]  = nA[5]& A[4]&nA[3]& A[2]& A[1]&nA[0],
          B[23]  = nA[5]& A[4]&nA[3]& A[2]& A[1]& A[0],
          B[24]  = nA[5]& A[4]& A[3]&nA[2]&nA[1]&nA[0],
          B[25]  = nA[5]& A[4]& A[3]&nA[2]&nA[1]& A[0],
          B[26]  = nA[5]& A[4]& A[3]&nA[2]& A[1]&nA[0],
          B[27]  = nA[5]& A[4]& A[3]&nA[2]& A[1]& A[0],
          B[28]  = nA[5]& A[4]& A[3]& A[2]&nA[1]&nA[0],
          B[29]  = nA[5]& A[4]& A[3]& A[2]&nA[1]& A[0],
          B[30]  = nA[5]& A[4]& A[3]& A[2]& A[1]&nA[0],
          B[31]  = nA[5]& A[4]& A[3]& A[2]& A[1]& A[0],
          B[32]  =  A[5]&nA[4]&nA[3]&nA[2]&nA[1]&nA[0],
          B[33]  =  A[5]&nA[4]&nA[3]&nA[2]&nA[1]& A[0],
          B[34]  =  A[5]&nA[4]&nA[3]&nA[2]& A[1]&nA[0],
          B[35]  =  A[5]&nA[4]&nA[3]&nA[2]& A[1]& A[0],
          B[36]  =  A[5]&nA[4]&nA[3]& A[2]&nA[1]&nA[0],
          B[37]  =  A[5]&nA[4]&nA[3]& A[2]&nA[1]& A[0],
          B[38]  =  A[5]&nA[4]&nA[3]& A[2]& A[1]&nA[0],
          B[39]  =  A[5]&nA[4]&nA[3]& A[2]& A[1]& A[0],
          B[40]  =  A[5]&nA[4]& A[3]&nA[2]&nA[1]&nA[0],
          B[41]  =  A[5]&nA[4]& A[3]&nA[2]&nA[1]& A[0],
          B[42]  =  A[5]&nA[4]& A[3]&nA[2]& A[1]&nA[0],
          B[43]  =  A[5]&nA[4]& A[3]&nA[2]& A[1]& A[0],
          B[44]  =  A[5]&nA[4]& A[3]& A[2]&nA[1]&nA[0],
          B[45]  =  A[5]&nA[4]& A[3]& A[2]&nA[1]& A[0],
          B[46]  =  A[5]&nA[4]& A[3]& A[2]& A[1]&nA[0],
          B[47]  =  A[5]&nA[4]& A[3]& A[2]& A[1]& A[0],
          B[48]  =  A[5]& A[4]&nA[3]&nA[2]&nA[1]&nA[0],
          B[49]  =  A[5]& A[4]&nA[3]&nA[2]&nA[1]& A[0],
          B[50]  =  A[5]& A[4]&nA[3]&nA[2]& A[1]&nA[0],
          B[51]  =  A[5]& A[4]&nA[3]&nA[2]& A[1]& A[0],
          B[52]  =  A[5]& A[4]&nA[3]& A[2]&nA[1]&nA[0],
          B[53]  =  A[5]& A[4]&nA[3]& A[2]&nA[1]& A[0],
          B[54]  =  A[5]& A[4]&nA[3]& A[2]& A[1]&nA[0],
          B[55]  =  A[5]& A[4]&nA[3]& A[2]& A[1]& A[0],
          B[56]  =  A[5]& A[4]& A[3]&nA[2]&nA[1]&nA[0],
          B[57]  =  A[5]& A[4]& A[3]&nA[2]&nA[1]& A[0],
          B[58]  =  A[5]& A[4]& A[3]&nA[2]& A[1]&nA[0],
          B[59]  =  A[5]& A[4]& A[3]&nA[2]& A[1]& A[0],
          B[60]  =  A[5]& A[4]& A[3]& A[2]&nA[1]&nA[0],
          B[61]  =  A[5]& A[4]& A[3]& A[2]&nA[1]& A[0],
          B[62]  =  A[5]& A[4]& A[3]& A[2]& A[1]&nA[0],
          B[63]  =  A[5]& A[4]& A[3]& A[2]& A[1]& A[0];

endmodule


module SF_MUX6X64_EN (A,B,ENABLE);
```

```
input [5:0] A;
input ENABLE;
output [63:0] B;

wire [5:0] nA;

assign nA = ~A;

assign  B[0]    = nA[5]&nA[4]&nA[3]&nA[2]&nA[1]&nA[0]&ENABLE,
        B[1]    = nA[5]&nA[4]&nA[3]&nA[2]&nA[1]& A[0]&ENABLE,
        B[2]    = nA[5]&nA[4]&nA[3]&nA[2]& A[1]&nA[0]&ENABLE,
        B[3]    = nA[5]&nA[4]&nA[3]&nA[2]& A[1]& A[0]&ENABLE,
        B[4]    = nA[5]&nA[4]&nA[3]& A[2]&nA[1]&nA[0]&ENABLE,
        B[5]    = nA[5]&nA[4]&nA[3]& A[2]&nA[1]& A[0]&ENABLE,
        B[6]    = nA[5]&nA[4]&nA[3]& A[2]& A[1]&nA[0]&ENABLE,
        B[7]    = nA[5]&nA[4]&nA[3]& A[2]& A[1]& A[0]&ENABLE,
        B[8]    = nA[5]&nA[4]& A[3]&nA[2]&nA[1]&nA[0]&ENABLE,
        B[9]    = nA[5]&nA[4]& A[3]&nA[2]&nA[1]& A[0]&ENABLE,
        B[10]   = nA[5]&nA[4]& A[3]&nA[2]& A[1]&nA[0]&ENABLE,
        B[11]   = nA[5]&nA[4]& A[3]&nA[2]& A[1]& A[0]&ENABLE,
        B[12]   = nA[5]&nA[4]& A[3]& A[2]&nA[1]&nA[0]&ENABLE,
        B[13]   = nA[5]&nA[4]& A[3]& A[2]&nA[1]& A[0]&ENABLE,
        B[14]   = nA[5]&nA[4]& A[3]& A[2]& A[1]&nA[0]&ENABLE,
        B[15]   = nA[5]&nA[4]& A[3]& A[2]& A[1]& A[0]&ENABLE,
        B[16]   = nA[5]& A[4]&nA[3]&nA[2]&nA[1]&nA[0]&ENABLE,
        B[17]   = nA[5]& A[4]&nA[3]&nA[2]&nA[1]& A[0]&ENABLE,
        B[18]   = nA[5]& A[4]&nA[3]&nA[2]& A[1]&nA[0]&ENABLE,
        B[19]   = nA[5]& A[4]&nA[3]&nA[2]& A[1]& A[0]&ENABLE,
        B[20]   = nA[5]& A[4]&nA[3]& A[2]&nA[1]&nA[0]&ENABLE,
        B[21]   = nA[5]& A[4]&nA[3]& A[2]&nA[1]& A[0]&ENABLE,
        B[22]   = nA[5]& A[4]&nA[3]& A[2]& A[1]&nA[0]&ENABLE,
        B[23]   = nA[5]& A[4]&nA[3]& A[2]& A[1]& A[0]&ENABLE,
        B[24]   = nA[5]& A[4]& A[3]&nA[2]&nA[1]&nA[0]&ENABLE,
        B[25]   = nA[5]& A[4]& A[3]&nA[2]&nA[1]& A[0]&ENABLE,
        B[26]   = nA[5]& A[4]& A[3]&nA[2]& A[1]&nA[0]&ENABLE,
        B[27]   = nA[5]& A[4]& A[3]&nA[2]& A[1]& A[0]&ENABLE,
        B[28]   = nA[5]& A[4]& A[3]& A[2]&nA[1]&nA[0]&ENABLE,
        B[29]   = nA[5]& A[4]& A[3]& A[2]&nA[1]& A[0]&ENABLE,
        B[30]   = nA[5]& A[4]& A[3]& A[2]& A[1]&nA[0]&ENABLE,
        B[31]   = nA[5]& A[4]& A[3]& A[2]& A[1]& A[0]&ENABLE,
        B[32]   =  A[5]&nA[4]&nA[3]&nA[2]&nA[1]&nA[0]&ENABLE,
        B[33]   =  A[5]&nA[4]&nA[3]&nA[2]&nA[1]& A[0]&ENABLE,
        B[34]   =  A[5]&nA[4]&nA[3]&nA[2]& A[1]&nA[0]&ENABLE,
        B[35]   =  A[5]&nA[4]&nA[3]&nA[2]& A[1]& A[0]&ENABLE,
        B[36]   =  A[5]&nA[4]&nA[3]& A[2]&nA[1]&nA[0]&ENABLE,
        B[37]   =  A[5]&nA[4]&nA[3]& A[2]&nA[1]& A[0]&ENABLE,
        B[38]   =  A[5]&nA[4]&nA[3]& A[2]& A[1]&nA[0]&ENABLE,
        B[39]   =  A[5]&nA[4]&nA[3]& A[2]& A[1]& A[0]&ENABLE,
            B[40]  =  A[5]&nA[4]& A[3]&nA[2]&nA[1]&nA[0]&ENABLE,
            B[41]  =  A[5]&nA[4]& A[3]&nA[2]&nA[1]& A[0]&ENABLE,
            B[42]  =  A[5]&nA[4]& A[3]&nA[2]& A[1]&nA[0]&ENABLE,
            B[43]  =  A[5]&nA[4]& A[3]&nA[2]& A[1]& A[0]&ENABLE,
            B[44]  =  A[5]&nA[4]& A[3]& A[2]&nA[1]&nA[0]&ENABLE,
            B[45]  =  A[5]&nA[4]& A[3]& A[2]&nA[1]& A[0]&ENABLE,
            B[46]  =  A[5]&nA[4]& A[3]& A[2]& A[1]&nA[0]&ENABLE,
            B[47]  =  A[5]&nA[4]& A[3]& A[2]& A[1]& A[0]&ENABLE,
            B[48]  =  A[5]& A[4]&nA[3]&nA[2]&nA[1]&nA[0]&ENABLE,
            B[49]  =  A[5]& A[4]&nA[3]&nA[2]&nA[1]& A[0]&ENABLE,
            B[50]  =  A[5]& A[4]&nA[3]&nA[2]& A[1]&nA[0]&ENABLE,
            B[51]  =  A[5]& A[4]&nA[3]&nA[2]& A[1]& A[0]&ENABLE,
            B[52]  =  A[5]& A[4]&nA[3]& A[2]&nA[1]&nA[0]&ENABLE,
            B[53]  =  A[5]& A[4]&nA[3]& A[2]&nA[1]& A[0]&ENABLE,
            B[54]  =  A[5]& A[4]&nA[3]& A[2]& A[1]&nA[0]&ENABLE,
            B[55]  =  A[5]& A[4]&nA[3]& A[2]& A[1]& A[0]&ENABLE,
            B[56]  =  A[5]& A[4]& A[3]&nA[2]&nA[1]&nA[0]&ENABLE,
            B[57]  =  A[5]& A[4]& A[3]&nA[2]&nA[1]& A[0]&ENABLE,
            B[58]  =  A[5]& A[4]& A[3]&nA[2]& A[1]&nA[0]&ENABLE,
            B[59]  =  A[5]& A[4]& A[3]&nA[2]& A[1]& A[0]&ENABLE,
            B[60]  =  A[5]& A[4]& A[3]& A[2]&nA[1]&nA[0]&ENABLE,
            B[61]  =  A[5]& A[4]& A[3]& A[2]&nA[1]& A[0]&ENABLE,
            B[62]  =  A[5]& A[4]& A[3]& A[2]& A[1]&nA[0]&ENABLE,
            B[63]  =  A[5]& A[4]& A[3]& A[2]& A[1]& A[0]&ENABLE;

endmodule


module SF_HALFADDER (A,B,S,C);

  input A,B;
  output C,S;

  xor g0 (S,A,B);
  and g1 (C,A,B);

endmodule


module SF_FULLADDER (A,B,Ci,S,Co);

  input A,B,Ci;
  output S,Co;

  wire T1,T2,T3;

  SF_HALFADDER ha0 (A,B,T1,T2);
  SF_HALFADDER ha1 (Ci,T1,S,T3);
  or g0 (Co,T2,T3);

endmodule


module SF_ADDER_6B (A,B,S,C);

  input [5:0] A,B;
  output [5:0] S;
  output C;

  wire [4:0] carry;

  SF_HALFADDER ha0 (A[0],B[0],S[0],carry[0]);
  SF_FULLADDER fa1 (A[1],B[1],carry[0],S[1],carry[1]);
  SF_FULLADDER fa2 (A[2],B[2],carry[1],S[2],carry[2]);
  SF_FULLADDER fa3 (A[3],B[3],carry[2],S[3],carry[3]);
  SF_FULLADDER fa4 (A[4],B[4],carry[3],S[4],carry[4]);
  SF_FULLADDER fa5 (A[5],B[5],carry[4],S[5],C);

endmodule


module SF_COMPARATOR_6B (A,B,AGTB,ALTB,AETB);

  input [5:0] A,B;
  output AGTB,ALTB,AETB;

  wire [5:0] x,A_nB,nA_B;
  wire [3:0] y;

  assign A_nB = A&(~B),
         nA_B = (~A)&B;

  assign y[3] = x[5]&x[4],
         y[2] = x[5]&x[4]&x[3],
         y[1] = x[5]&x[4]&x[3]&x[2],
         y[0] = x[5]&x[4]&x[3]&x[2]&x[1];

  assign x=A~^B,
         AETB = &x,
         AGTB=A_nB[5]|(x[5]&A_nB[4])|(y[3]&A_nB[3])|(y[2]&A_nB[2])|
                      (y[1]&A_nB[1])|(y[0]&A_nB[0]),
         ALTB=nA_B[5]|(x[5]&nA_B[4])|(y[3]&nA_B[3])|(y[2]&nA_B[2])|
                      (y[1]&nA_B[1])|(y[0]&nA_B[0]);
endmodule
```

```
module SF_COMPARATOR_7B (A,B,AGTB,ALTB,AETB);

  input [6:0] A,B;
  output AGTB,ALTB,AETB;

  wire [6:0] x,A_nB,nA_B;
  wire [4:0] y;

  assign A_nB = A&(~B),
         nA_B = (~A)&B;

  assign y[4] = x[6]&x[5],
         y[3] = x[6]&x[5]&x[4],
         y[2] = x[6]&x[5]&x[4]&x[3],
         y[1] = x[6]&x[5]&x[4]&x[3]&x[2],
         y[0] = x[6]&x[5]&x[4]&x[3]&x[2]&x[1];

  assign x=A~^B,
         AETB = &x,
         AGTB=A_nB[6]|(x[6]&A_nB[5])|(y[4]&A_nB[4])|(y[3]&A_nB[3])|
                    (y[2]&A_nB[2])|(y[1]&A_nB[1])|(y[0]&A_nB[0]),
         ALTB=nA_B[6]|(x[6]&nA_B[5])|(y[4]&nA_B[4])|(y[3]&nA_B[3])|
                    (y[2]&nA_B[2])|(y[1]&nA_B[1])|(y[0]&nA_B[0]);

endmodule


module SF_COMPARE_UNIT (INPUT1,NUMBER,INPUT2,AGTB,ALTB,AETB);

  input [5:0] INPUT1,NUMBER,INPUT2;
  output AGTB,ALTB,AETB;

  wire [5:0] A;
  wire C;
  reg [6:0] X,Y;

  SF_ADDER_6B ad0 (INPUT1,NUMBER,A,C);
  SF_COMPARATOR_7B comm0(X,Y,AGTB,ALTB,AETB);

  always @(A or C) begin
    X[6] = C;
    X[5:0] = A;
  end
  always @(INPUT2) begin
    Y[6]=0;
    Y[5:0] = INPUT2;
  end
endmodule
```

```
module RECEIVE_FIFO (CLK,RESET,DATA,CS,DATA_TYPE,WRITE_READ,
                     RECEIVE_EMPTY,RECEIVE_FULL,RECEIVE_8,
                     RECEIVE_16,RECEIVE_24,
                     DAHOD,CAHOD,
                     DATA_IN,RECEIVE_CLK);

  input CLK,RESET;
  input [8:0] DATA_IN;
  input CS;
  input [1:0] DATA_TYPE;
  input WRITE_READ;

  output RECEIVE_EMPTY,RECEIVE_FULL,RECEIVE_8,RECEIVE_16,RECEIVE_24;
  output DAHOD,CAHOD;
  output [71:0] DATA;
  input RECEIVE_CLK;

  reg [5:0] TAIL,HEAD;
  reg [5:0] HP,TP;
  wire [5:0] A,B,S,H;
  wire [5:0] HP_1,HP_2,HP_3,HP_4,HP_5,HP_6,HP_7;
  wire TAIL_HEAD_RESET;
  wire HEAD_INCREMENT;


  RF_MEMORYBANK rmb (RECEIVE_CLK,HP,HP_1,HP_2,HP_3,
                     HP_4,HP_5,HP_6,HP_7,TP,DATA,DATA_IN);

  RF_STATUS rfs (HEAD,TAIL,RECEIVE_EMPTY,RECEIVE_FULL,
                 RECEIVE_8,RECEIVE_16,RECEIVE_24);

  assign TAIL_HEAD_RESET = CS & WRITE_READ & (~(|DATA_TYPE));

  assign HEAD_INCREMENT = CS & (~WRITE_READ);

  assign CAHOD = DATA[8] & (~RECEIVE_EMPTY),
         DAHOD = ~(DATA[8]|RECEIVE_EMPTY);


  RF_ADDER_6B add8_0 (TAIL,A,S,),
              add8_1 (HEAD,B,H,),
              add8_2 (HP,6'b000001,HP_1,),
              add8_3 (HP,6'b000010,HP_2,),
              add8_4 (HP,6'b000011,HP_3,),
              add8_5 (HP,6'b000100,HP_4,),
              add8_6 (HP,6'b000101,HP_5,),
               add8_7 (HP,6'b000110,HP_6,),
               add8_8 (HP,6'b000111,HP_7,);

  assign A[5] = 0,
         A[4] = 0,
         A[3] = 0,
         A[2] = 0,
         A[1] = 0,
         A[0] = ~(&DATA_IN);

  assign B[5] = 0,
         B[4] = 0,
         B[3] = DAHOD & (~CAHOD) & RECEIVE_8,
         B[2] = 0,
         B[1] = 0,
         B[0] = CAHOD & (~DAHOD) & ~(RECEIVE_EMPTY);

  always @(posedge RESET or posedge CLK) begin
    if(RESET)
      TP = 0;
    else
      TP = TAIL;
  end

  always @(posedge RESET or posedge RECEIVE_CLK) begin
    if(RESET)
      TAIL = 0;
    else
      TAIL = S;
  end

  always @(posedge RESET or posedge CLK) begin
    if(RESET)
      HEAD = 0;
    else begin
      if(TAIL_HEAD_RESET)
        HEAD = TAIL;
      else if(HEAD_INCREMENT)
        HEAD = H;
      else
        HEAD = HEAD;
    end
  end

  always @(posedge RESET or negedge CLK) begin
    if(RESET)
      HP = 0;
    else
```

```
        HP = HEAD;
     end

  endmodule


  module RF_STATUS(HEAD,TAIL,RECEIVE_EMPTY,RECEIVE_FULL,
                              RECEIVE_8,RECEIVE_16,RECEIVE_24);

    input [5:0] HEAD, TAIL;
    output RECEIVE_EMPTY,RECEIVE_FULL,RECEIVE_8,RECEIVE_16,RECEIVE_24;

    wire rfa,rfb;
    wire r8a,r8b,r8c,r8d;
    wire r16a,r16b,r16c,r16d;
    wire r24a,r24b,r24c,r24d;
    wire TGTH,TLTH;

    assign RECEIVE_FULL = (TGTH&rfa) | (TLTH&rfb),
           RECEIVE_8    = (TGTH&(r8a|r8b))|(TLTH&(r8c|r8d)),
           RECEIVE_16   = (TGTH&(r16a|r16b))|(TLTH&(r16c|r16d)),
           RECEIVE_24   = (TGTH&(r24a|r24b))|(TLTH&(r24c|r24d));

    RF_COMPARE_UNIT comm0 (HEAD,6'b111111,TAIL,,,rfa),
                    comm1 (TAIL,6'b000001,HEAD,,,rfb),
                    comm2 (HEAD,6'b001000,TAIL,,r8a,r8b),
                    comm3 (TAIL,6'b111000,HEAD,r8c,,r8d),
                    comm4 (HEAD,6'b010000,TAIL,,r16a,r16b),
                    comm5 (TAIL,6'b110000,HEAD,r16c,,r16d),
                    comm6 (HEAD,6'b011000,TAIL,,r24a,r24b),
                    comm7 (TAIL,6'b101000,HEAD,r24c,,r24d);

   RF_COMPARATOR_6B comm8 (TAIL,HEAD,TGTH,TLTH,RECEIVE_EMPTY);
 endmodule


 module RF_MEMORYCELL(RECEIVE_CLK,
                      HPEQ_0,HPEQ_1,HPEQ_2,HPEQ_3,
                      HPEQ_4,HPEQ_5,HPEQ_6,HPEQ_7,
                      TPEQ,DATA,DIN);

   input RECEIVE_CLK;
   input HPEQ_0,HPEQ_1,HPEQ_2,HPEQ_3,HPEQ_4,HPEQ_5,HPEQ_6,HPEQ_7;
   input TPEQ;
   output [71:0] DATA;
   input [8:0] DIN;

   reg [8:0] MEM;
   assign DATA[8:0]   = (HPEQ_0==1'b1) ? MEM : 9'bzzzzzzzzz,
          DATA[17:9]  = (HPEQ_1==1'b1) ? MEM : 9'bzzzzzzzzz,
          DATA[26:18] = (HPEQ_2==1'b1) ? MEM : 9'bzzzzzzzzz,
          DATA[35:27] = (HPEQ_3==1'b1) ? MEM : 9'bzzzzzzzzz,
          DATA[44:36] = (HPEQ_4==1'b1) ? MEM : 9'bzzzzzzzzz,
          DATA[53:45] = (HPEQ_5==1'b1) ? MEM : 9'bzzzzzzzzz,
          DATA[62:54] = (HPEQ_6==1'b1) ? MEM : 9'bzzzzzzzzz,
          DATA[71:63] = (HPEQ_7==1'b1) ? MEM : 9'bzzzzzzzzz;

  always @(posedge RECEIVE_CLK) begin
    if(TPEQ)
      MEM = DIN;
  end
endmodule


module RF_MEMORYBANK(RECEIVE_CLK,HP,HP_1,HP_2,HP_3,
                     HP_4,HP_5,HP_6,HP_7,TP,DATA,DIN);

  input RECEIVE_CLK;
  input [5:0] HP,HP_1,HP_2,HP_3,HP_4,HP_5,HP_6,HP_7;
  input [5:0] TP;
  output [71:0] DATA;
  input [8:0] DIN;

  tri [71:0] DATA;
  wire [63:0] HPEQ_0,HPEQ_1,HPEQ_2,HPEQ_3,HPEQ_4,HPEQ_5,HPEQ_6,HPEQ_7;
  wire [63:0] TPEQ;

  RF_MUX6X64  m0 (HP,HPEQ_0),
              m1 (HP_1,HPEQ_1),
              m2 (HP_2,HPEQ_2),
              m3 (HP_3,HPEQ_3),
              m4 (HP_4,HPEQ_4),
              m5 (HP_5,HPEQ_5),
              m6 (HP_6,HPEQ_6),
              m7 (HP_7,HPEQ_7),
              m8 (TP,TPEQ);

  RF_MEMORYCELL rfmc0 (RECEIVE_CLK,HPEQ_0[0],HPEQ_1[0],HPEQ_2[0],
                       HPEQ_3[0],HPEQ_4[0],HPEQ_5[0],HPEQ_6[0],
                       HPEQ_7[0],TPEQ[0],DATA,DIN);
  RF_MEMORYCELL rfmc1 (RECEIVE_CLK,HPEQ_0[1],HPEQ_1[1],HPEQ_2[1],
                       HPEQ_3[1],HPEQ_4[1],HPEQ_5[1],HPEQ_6[1],
                       HPEQ_7[1],TPEQ[1],DATA,DIN);
  RF_MEMORYCELL rfmc2 (RECEIVE_CLK,HPEQ_0[2],HPEQ_1[2],HPEQ_2[2],
                       HPEQ_3[2],HPEQ_4[2],HPEQ_5[2],HPEQ_6[2],
                    HPEQ_7[2],TPEQ[2],DATA,DIN);
  RF_MEMORYCELL rfmc3 (RECEIVE_CLK,HPEQ_0[3],HPEQ_1[3],HPEQ_2[3],
                    HPEQ_3[3],HPEQ_4[3],HPEQ_5[3],HPEQ_6[3],
                    HPEQ_7[3],TPEQ[3],DATA,DIN);
  RF_MEMORYCELL rfmc4 (RECEIVE_CLK,HPEQ_0[4],HPEQ_1[4],HPEQ_2[4],
                    HPEQ_3[4],HPEQ_4[4],HPEQ_5[4],HPEQ_6[4],
                    HPEQ_7[4],TPEQ[4],DATA,DIN);
  RF_MEMORYCELL rfmc5 (RECEIVE_CLK,HPEQ_0[5],HPEQ_1[5],HPEQ_2[5],
                    HPEQ_3[5],HPEQ_4[5],HPEQ_5[5],HPEQ_6[5],
                    HPEQ_7[5],TPEQ[5],DATA,DIN);
  RF_MEMORYCELL rfmc6 (RECEIVE_CLK,HPEQ_0[6],HPEQ_1[6],HPEQ_2[6],
                    HPEQ_3[6],HPEQ_4[6],HPEQ_5[6],HPEQ_6[6],
                    HPEQ_7[6],TPEQ[6],DATA,DIN);
  RF_MEMORYCELL rfmc7 (RECEIVE_CLK,HPEQ_0[7],HPEQ_1[7],HPEQ_2[7],
                    HPEQ_3[7],HPEQ_4[7],HPEQ_5[7],HPEQ_6[7],
                    HPEQ_7[7],TPEQ[7],DATA,DIN);
  RF_MEMORYCELL rfmc8 (RECEIVE_CLK,HPEQ_0[8],HPEQ_1[8],HPEQ_2[8],
                    HPEQ_3[8],HPEQ_4[8],HPEQ_5[8],HPEQ_6[8],
                    HPEQ_7[8],TPEQ[8],DATA,DIN);
  RF_MEMORYCELL rfmc9 (RECEIVE_CLK,HPEQ_0[9],HPEQ_1[9],HPEQ_2[9],
                    HPEQ_3[9],HPEQ_4[9],HPEQ_5[9],HPEQ_6[9],
                    HPEQ_7[9],TPEQ[9],DATA,DIN);
  RF_MEMORYCELL rfmc10 (RECEIVE_CLK,HPEQ_0[10],HPEQ_1[10],HPEQ_2[10],
                    HPEQ_3[10],HPEQ_4[10],HPEQ_5[10],HPEQ_6[10],
                    HPEQ_7[10],TPEQ[10],DATA,DIN);
  RF_MEMORYCELL rfmc11 (RECEIVE_CLK,HPEQ_0[11],HPEQ_1[11],HPEQ_2[11],
                    HPEQ_3[11],HPEQ_4[11],HPEQ_5[11],HPEQ_6[11],
                    HPEQ_7[11],TPEQ[11],DATA,DIN);
  RF_MEMORYCELL rfmc12 (RECEIVE_CLK,HPEQ_0[12],HPEQ_1[12],HPEQ_2[12],
                    HPEQ_3[12],HPEQ_4[12],HPEQ_5[12],HPEQ_6[12],
                    HPEQ_7[12],TPEQ[12],DATA,DIN);
  RF_MEMORYCELL rfmc13 (RECEIVE_CLK,HPEQ_0[13],HPEQ_1[13],HPEQ_2[13],
                    HPEQ_3[13],HPEQ_4[13],HPEQ_5[13],HPEQ_6[13],
                    HPEQ_7[13],TPEQ[13],DATA,DIN);
  RF_MEMORYCELL rfmc14 (RECEIVE_CLK,HPEQ_0[14],HPEQ_1[14],HPEQ_2[14],
                    HPEQ_3[14],HPEQ_4[14],HPEQ_5[14],HPEQ_6[14],
                    HPEQ_7[14],TPEQ[14],DATA,DIN);
  RF_MEMORYCELL rfmc15 (RECEIVE_CLK,HPEQ_0[15],HPEQ_1[15],HPEQ_2[15],
                    HPEQ_3[15],HPEQ_4[15],HPEQ_5[15],HPEQ_6[15],
                    HPEQ_7[15],TPEQ[15],DATA,DIN);
  RF_MEMORYCELL rfmc16 (RECEIVE_CLK,HPEQ_0[16],HPEQ_1[16],HPEQ_2[16],
                    HPEQ_3[16],HPEQ_4[16],HPEQ_5[16],HPEQ_6[16],
                    HPEQ_7[16],TPEQ[16],DATA,DIN);
  RF_MEMORYCELL rfmc17 (RECEIVE_CLK,HPEQ_0[17],HPEQ_1[17],HPEQ_2[17],
                    HPEQ_3[17],HPEQ_4[17],HPEQ_5[17],HPEQ_6[17],
                    HPEQ_7[17],TPEQ[17],DATA,DIN);
  RF_MEMORYCELL rfmc18 (RECEIVE_CLK,HPEQ_0[18],HPEQ_1[18],HPEQ_2[18],
                    HPEQ_3[18],HPEQ_4[18],HPEQ_5[18],HPEQ_6[18],
                      HPEQ_7[18],TPEQ[18],DATA,DIN);
RF_MEMORYCELL rfmc19 (RECEIVE_CLK,HPEQ_0[19],HPEQ_1[19],HPEQ_2[19],
                      HPEQ_3[19],HPEQ_4[19],HPEQ_5[19],HPEQ_6[19],
                      HPEQ_7[19],TPEQ[19],DATA,DIN);
RF_MEMORYCELL rfmc20 (RECEIVE_CLK,HPEQ_0[20],HPEQ_1[20],HPEQ_2[20],
                      HPEQ_3[20],HPEQ_4[20],HPEQ_5[20],HPEQ_6[20],
                      HPEQ_7[20],TPEQ[20],DATA,DIN);
RF_MEMORYCELL rfmc21 (RECEIVE_CLK,HPEQ_0[21],HPEQ_1[21],HPEQ_2[21],
                      HPEQ_3[21],HPEQ_4[21],HPEQ_5[21],HPEQ_6[21],
                      HPEQ_7[21],TPEQ[21],DATA,DIN);
RF_MEMORYCELL rfmc22 (RECEIVE_CLK,HPEQ_0[22],HPEQ_1[22],HPEQ_2[22],
                      HPEQ_3[22],HPEQ_4[22],HPEQ_5[22],HPEQ_6[22],
                      HPEQ_7[22],TPEQ[22],DATA,DIN);
RF_MEMORYCELL rfmc23 (RECEIVE_CLK,HPEQ_0[23],HPEQ_1[23],HPEQ_2[23],
                      HPEQ_3[23],HPEQ_4[23],HPEQ_5[23],HPEQ_6[23],
                      HPEQ_7[23],TPEQ[23],DATA,DIN);
RF_MEMORYCELL rfmc24 (RECEIVE_CLK,HPEQ_0[24],HPEQ_1[24],HPEQ_2[24],
                      HPEQ_3[24],HPEQ_4[24],HPEQ_5[24],HPEQ_6[24],
                      HPEQ_7[24],TPEQ[24],DATA,DIN);
RF_MEMORYCELL rfmc25 (RECEIVE_CLK,HPEQ_0[25],HPEQ_1[25],HPEQ_2[25],
                      HPEQ_3[25],HPEQ_4[25],HPEQ_5[25],HPEQ_6[25],
                      HPEQ_7[25],TPEQ[25],DATA,DIN);
RF_MEMORYCELL rfmc26 (RECEIVE_CLK,HPEQ_0[26],HPEQ_1[26],HPEQ_2[26],
                      HPEQ_3[26],HPEQ_4[26],HPEQ_5[26],HPEQ_6[26],
                      HPEQ_7[26],TPEQ[26],DATA,DIN);
RF_MEMORYCELL rfmc27 (RECEIVE_CLK,HPEQ_0[27],HPEQ_1[27],HPEQ_2[27],
                      HPEQ_3[27],HPEQ_4[27],HPEQ_5[27],HPEQ_6[27],
                      HPEQ_7[27],TPEQ[27],DATA,DIN);
RF_MEMORYCELL rfmc28 (RECEIVE_CLK,HPEQ_0[28],HPEQ_1[28],HPEQ_2[28],
                      HPEQ_3[28],HPEQ_4[28],HPEQ_5[28],HPEQ_6[28],
                      HPEQ_7[28],TPEQ[28],DATA,DIN);
RF_MEMORYCELL rfmc29 (RECEIVE_CLK,HPEQ_0[29],HPEQ_1[29],HPEQ_2[29],
                      HPEQ_3[29],HPEQ_4[29],HPEQ_5[29],HPEQ_6[29],
                      HPEQ_7[29],TPEQ[29],DATA,DIN);
RF_MEMORYCELL rfmc30 (RECEIVE_CLK,HPEQ_0[30],HPEQ_1[30],HPEQ_2[30],
                      HPEQ_3[30],HPEQ_4[30],HPEQ_5[30],HPEQ_6[30],
                      HPEQ_7[30],TPEQ[30],DATA,DIN);
RF_MEMORYCELL rfmc31 (RECEIVE_CLK,HPEQ_0[31],HPEQ_1[31],HPEQ_2[31],
                      HPEQ_3[31],HPEQ_4[31],HPEQ_5[31],HPEQ_6[31],
                      HPEQ_7[31],TPEQ[31],DATA,DIN);
RF_MEMORYCELL rfmc32 (RECEIVE_CLK,HPEQ_0[32],HPEQ_1[32],HPEQ_2[32],
                      HPEQ_3[32],HPEQ_4[32],HPEQ_5[32],HPEQ_6[32],
                      HPEQ_7[32],TPEQ[32],DATA,DIN);
RF_MEMORYCELL rfmc33 (RECEIVE_CLK,HPEQ_0[33],HPEQ_1[33],HPEQ_2[33],
                      HPEQ_3[33],HPEQ_4[33],HPEQ_5[33],HPEQ_6[33],
                      HPEQ_7[33],TPEQ[33],DATA,DIN);
RF_MEMORYCELL rfmc34 (RECEIVE_CLK,HPEQ_0[34],HPEQ_1[34],HPEQ_2[34],
                      HPEQ_3[34],HPEQ_4[34],HPEQ_5[34],HPEQ_6[34],
                                HPEQ_7[34],TPEQ[34],DATA,DIN);
   RF_MEMORYCELL rfmc35 (RECEIVE_CLK,HPEQ_0[35],HPEQ_1[35],HPEQ_2[35],
                                HPEQ_3[35],HPEQ_4[35],HPEQ_5[35],HPEQ_6[35],
                                HPEQ_7[35],TPEQ[35],DATA,DIN);
   RF_MEMORYCELL rfmc36 (RECEIVE_CLK,HPEQ_0[36],HPEQ_1[36],HPEQ_2[36],
                                HPEQ_3[36],HPEQ_4[36],HPEQ_5[36],HPEQ_6[36],
                                HPEQ_7[36],TPEQ[36],DATA,DIN);
   RF_MEMORYCELL rfmc37 (RECEIVE_CLK,HPEQ_0[37],HPEQ_1[37],HPEQ_2[37],
                                HPEQ_3[37],HPEQ_4[37],HPEQ_5[37],HPEQ_6[37],
                                HPEQ_7[37],TPEQ[37],DATA,DIN);
   RF_MEMORYCELL rfmc38 (RECEIVE_CLK,HPEQ_0[38],HPEQ_1[38],HPEQ_2[38],
                                HPEQ_3[38],HPEQ_4[38],HPEQ_5[38],HPEQ_6[38],
                                HPEQ_7[38],TPEQ[38],DATA,DIN);
   RF_MEMORYCELL rfmc39 (RECEIVE_CLK,HPEQ_0[39],HPEQ_1[39],HPEQ_2[39],
                                HPEQ_3[39],HPEQ_4[39],HPEQ_5[39],HPEQ_6[39],
                                HPEQ_7[39],TPEQ[39],DATA,DIN);
   RF_MEMORYCELL rfmc40 (RECEIVE_CLK,HPEQ_0[40],HPEQ_1[40],HPEQ_2[40],
                                HPEQ_3[40],HPEQ_4[40],HPEQ_5[40],HPEQ_6[40],
                                HPEQ_7[40],TPEQ[40],DATA,DIN);
   RF_MEMORYCELL rfmc41 (RECEIVE_CLK,HPEQ_0[41],HPEQ_1[41],HPEQ_2[41],
                                HPEQ_3[41],HPEQ_4[41],HPEQ_5[41],HPEQ_6[41],
                                HPEQ_7[41],TPEQ[41],DATA,DIN);
   RF_MEMORYCELL rfmc42 (RECEIVE_CLK,HPEQ_0[42],HPEQ_1[42],HPEQ_2[42],
                                HPEQ_3[42],HPEQ_4[42],HPEQ_5[42],HPEQ_6[42],
                                HPEQ_7[42],TPEQ[42],DATA,DIN);
   RF_MEMORYCELL rfmc43 (RECEIVE_CLK,HPEQ_0[43],HPEQ_1[43],HPEQ_2[43],
                                HPEQ_3[43],HPEQ_4[43],HPEQ_5[43],HPEQ_6[43],
                                HPEQ_7[43],TPEQ[43],DATA,DIN);
   RF_MEMORYCELL rfmc44 (RECEIVE_CLK,HPEQ_0[44],HPEQ_1[44],HPEQ_2[44],
                                HPEQ_3[44],HPEQ_4[44],HPEQ_5[44],HPEQ_6[44],
                                HPEQ_7[44],TPEQ[44],DATA,DIN);
   RF_MEMORYCELL rfmc45 (RECEIVE_CLK,HPEQ_0[45],HPEQ_1[45],HPEQ_2[45],
                                HPEQ_3[45],HPEQ_4[45],HPEQ_5[45],HPEQ_6[45],
                                HPEQ_7[45],TPEQ[45],DATA,DIN);
   RF_MEMORYCELL rfmc46 (RECEIVE_CLK,HPEQ_0[46],HPEQ_1[46],HPEQ_2[46],
                                HPEQ_3[46],HPEQ_4[46],HPEQ_5[46],HPEQ_6[46],
                                HPEQ_7[46],TPEQ[46],DATA,DIN);
   RF_MEMORYCELL rfmc47 (RECEIVE_CLK,HPEQ_0[47],HPEQ_1[47],HPEQ_2[47],
                                HPEQ_3[47],HPEQ_4[47],HPEQ_5[47],HPEQ_6[47],
                                HPEQ_7[47],TPEQ[47],DATA,DIN);
   RF_MEMORYCELL rfmc48 (RECEIVE_CLK,HPEQ_0[48],HPEQ_1[48],HPEQ_2[48],
                                HPEQ_3[48],HPEQ_4[48],HPEQ_5[48],HPEQ_6[48],
                                HPEQ_7[48],TPEQ[48],DATA,DIN);
   RF_MEMORYCELL rfmc49 (RECEIVE_CLK,HPEQ_0[49],HPEQ_1[49],HPEQ_2[49],
                                HPEQ_3[49],HPEQ_4[49],HPEQ_5[49],HPEQ_6[49],
                                HPEQ_7[49],TPEQ[49],DATA,DIN);
   RF_MEMORYCELL rfmc50 (RECEIVE_CLK,HPEQ_0[50],HPEQ_1[50],HPEQ_2[50],
                                HPEQ_3[50],HPEQ_4[50],HPEQ_5[50],HPEQ_6[50],
                             HPEQ_7[50],TPEQ[50],DATA,DIN);
    RF_MEMORYCELL rfmc51 (RECEIVE_CLK,HPEQ_0[51],HPEQ_1[51],HPEQ_2[51],
                             HPEQ_3[51],HPEQ_4[51],HPEQ_5[51],HPEQ_6[51],
                             HPEQ_7[51],TPEQ[51],DATA,DIN);
    RF_MEMORYCELL rfmc52 (RECEIVE_CLK,HPEQ_0[52],HPEQ_1[52],HPEQ_2[52],
                             HPEQ_3[52],HPEQ_4[52],HPEQ_5[52],HPEQ_6[52],
                             HPEQ_7[52],TPEQ[52],DATA,DIN);
    RF_MEMORYCELL rfmc53 (RECEIVE_CLK,HPEQ_0[53],HPEQ_1[53],HPEQ_2[53],
                             HPEQ_3[53],HPEQ_4[53],HPEQ_5[53],HPEQ_6[53],
                             HPEQ_7[53],TPEQ[53],DATA,DIN);
    RF_MEMORYCELL rfmc54 (RECEIVE_CLK,HPEQ_0[54],HPEQ_1[54],HPEQ_2[54],
                             HPEQ_3[54],HPEQ_4[54],HPEQ_5[54],HPEQ_6[54],
                             HPEQ_7[54],TPEQ[54],DATA,DIN);
    RF_MEMORYCELL rfmc55 (RECEIVE_CLK,HPEQ_0[55],HPEQ_1[55],HPEQ_2[55],
                             HPEQ_3[55],HPEQ_4[55],HPEQ_5[55],HPEQ_6[55],
                             HPEQ_7[55],TPEQ[55],DATA,DIN);
    RF_MEMORYCELL rfmc56 (RECEIVE_CLK,HPEQ_0[56],HPEQ_1[56],HPEQ_2[56],
                             HPEQ_3[56],HPEQ_4[56],HPEQ_5[56],HPEQ_6[56],
                             HPEQ_7[56],TPEQ[56],DATA,DIN);
    RF_MEMORYCELL rfmc57 (RECEIVE_CLK,HPEQ_0[57],HPEQ_1[57],HPEQ_2[57],
                             HPEQ_3[57],HPEQ_4[57],HPEQ_5[57],HPEQ_6[57],
                             HPEQ_7[57],TPEQ[57],DATA,DIN);
    RF_MEMORYCELL rfmc58 (RECEIVE_CLK,HPEQ_0[58],HPEQ_1[58],HPEQ_2[58],
                             HPEQ_3[58],HPEQ_4[58],HPEQ_5[58],HPEQ_6[58],
                             HPEQ_7[58],TPEQ[58],DATA,DIN);
    RF_MEMORYCELL rfmc59 (RECEIVE_CLK,HPEQ_0[59],HPEQ_1[59],HPEQ_2[59],
                             HPEQ_3[59],HPEQ_4[59],HPEQ_5[59],HPEQ_6[59],
                             HPEQ_7[59],TPEQ[59],DATA,DIN);
    RF_MEMORYCELL rfmc60 (RECEIVE_CLK,HPEQ_0[60],HPEQ_1[60],HPEQ_2[60],
                             HPEQ_3[60],HPEQ_4[60],HPEQ_5[60],HPEQ_6[60],
                             HPEQ_7[60],TPEQ[60],DATA,DIN);
    RF_MEMORYCELL rfmc61 (RECEIVE_CLK,HPEQ_0[61],HPEQ_1[61],HPEQ_2[61],
                             HPEQ_3[61],HPEQ_4[61],HPEQ_5[61],HPEQ_6[61],
                             HPEQ_7[61],TPEQ[61],DATA,DIN);
    RF_MEMORYCELL rfmc62 (RECEIVE_CLK,HPEQ_0[62],HPEQ_1[62],HPEQ_2[62],
                             HPEQ_3[62],HPEQ_4[62],HPEQ_5[62],HPEQ_6[62],
                             HPEQ_7[62],TPEQ[62],DATA,DIN);
    RF_MEMORYCELL rfmc63 (RECEIVE_CLK,HPEQ_0[63],HPEQ_1[63],HPEQ_2[63],
                             HPEQ_3[63],HPEQ_4[63],HPEQ_5[63],HPEQ_6[63],
                             HPEQ_7[63],TPEQ[63],DATA,DIN);
endmodule


module RF_MUX6X64 (A,B);

  input [5:0] A;
  output [63:0] B;
```

```
wire [5:0] nA;

assign nA = ~A;

assign  B[0]   = nA[5]&nA[4]&nA[3]&nA[2]&nA[1]&nA[0],
        B[1]   = nA[5]&nA[4]&nA[3]&nA[2]&nA[1]& A[0],
        B[2]   = nA[5]&nA[4]&nA[3]&nA[2]& A[1]&nA[0],
        B[3]   = nA[5]&nA[4]&nA[3]&nA[2]& A[1]& A[0],
        B[4]   = nA[5]&nA[4]&nA[3]& A[2]&nA[1]&nA[0],
        B[5]   = nA[5]&nA[4]&nA[3]& A[2]&nA[1]& A[0],
        B[6]   = nA[5]&nA[4]&nA[3]& A[2]& A[1]&nA[0],
        B[7]   = nA[5]&nA[4]&nA[3]& A[2]& A[1]& A[0],
        B[8]   = nA[5]&nA[4]& A[3]&nA[2]&nA[1]&nA[0],
        B[9]   = nA[5]&nA[4]& A[3]&nA[2]&nA[1]& A[0],
        B[10]  = nA[5]&nA[4]& A[3]&nA[2]& A[1]&nA[0],
        B[11]  = nA[5]&nA[4]& A[3]&nA[2]& A[1]& A[0],
        B[12]  = nA[5]&nA[4]& A[3]& A[2]&nA[1]&nA[0],
        B[13]  = nA[5]&nA[4]& A[3]& A[2]&nA[1]& A[0],
        B[14]  = nA[5]&nA[4]& A[3]& A[2]& A[1]&nA[0],
        B[15]  = nA[5]&nA[4]& A[3]& A[2]& A[1]& A[0],
        B[16]  = nA[5]& A[4]&nA[3]&nA[2]&nA[1]&nA[0],
        B[17]  = nA[5]& A[4]&nA[3]&nA[2]&nA[1]& A[0],
        B[18]  = nA[5]& A[4]&nA[3]&nA[2]& A[1]&nA[0],
        B[19]  = nA[5]& A[4]&nA[3]&nA[2]& A[1]& A[0],
        B[20]  = nA[5]& A[4]&nA[3]& A[2]&nA[1]&nA[0],
        B[21]  = nA[5]& A[4]&nA[3]& A[2]&nA[1]& A[0],
        B[22]  = nA[5]& A[4]&nA[3]& A[2]& A[1]&nA[0],
        B[23]  = nA[5]& A[4]&nA[3]& A[2]& A[1]& A[0],
        B[24]  = nA[5]& A[4]& A[3]&nA[2]&nA[1]&nA[0],
        B[25]  = nA[5]& A[4]& A[3]&nA[2]&nA[1]& A[0],
        B[26]  = nA[5]& A[4]& A[3]&nA[2]& A[1]&nA[0],
        B[27]  = nA[5]& A[4]& A[3]&nA[2]& A[1]& A[0],
        B[28]  = nA[5]& A[4]& A[3]& A[2]&nA[1]&nA[0],
        B[29]  = nA[5]& A[4]& A[3]& A[2]&nA[1]& A[0],
        B[30]  = nA[5]& A[4]& A[3]& A[2]& A[1]&nA[0],
        B[31]  = nA[5]& A[4]& A[3]& A[2]& A[1]& A[0],
        B[32]  =  A[5]&nA[4]&nA[3]&nA[2]&nA[1]&nA[0],
        B[33]  =  A[5]&nA[4]&nA[3]&nA[2]&nA[1]& A[0],
        B[34]  =  A[5]&nA[4]&nA[3]&nA[2]& A[1]&nA[0],
        B[35]  =  A[5]&nA[4]&nA[3]&nA[2]& A[1]& A[0],
        B[36]  =  A[5]&nA[4]&nA[3]& A[2]&nA[1]&nA[0],
        B[37]  =  A[5]&nA[4]&nA[3]& A[2]&nA[1]& A[0],
        B[38]  =  A[5]&nA[4]&nA[3]& A[2]& A[1]&nA[0],
        B[39]  =  A[5]&nA[4]&nA[3]& A[2]& A[1]& A[0],
        B[40]  =  A[5]&nA[4]& A[3]&nA[2]&nA[1]&nA[0],
        B[41]  =  A[5]&nA[4]& A[3]&nA[2]&nA[1]& A[0],
        B[42]  =  A[5]&nA[4]& A[3]&nA[2]& A[1]&nA[0],
        B[43]  =  A[5]&nA[4]& A[3]&nA[2]& A[1]& A[0],
            B[44]  =  A[5]&nA[4]& A[3]& A[2]&nA[1]&nA[0],
            B[45]  =  A[5]&nA[4]& A[3]& A[2]&nA[1]& A[0],
            B[46]  =  A[5]&nA[4]& A[3]& A[2]& A[1]&nA[0],
            B[47]  =  A[5]&nA[4]& A[3]& A[2]& A[1]& A[0],
            B[48]  =  A[5]& A[4]&nA[3]&nA[2]&nA[1]&nA[0],
            B[49]  =  A[5]& A[4]&nA[3]&nA[2]&nA[1]& A[0],
            B[50]  =  A[5]& A[4]&nA[3]&nA[2]& A[1]&nA[0],
            B[51]  =  A[5]& A[4]&nA[3]&nA[2]& A[1]& A[0],
            B[52]  =  A[5]& A[4]&nA[3]& A[2]&nA[1]&nA[0],
            B[53]  =  A[5]& A[4]&nA[3]& A[2]&nA[1]& A[0],
            B[54]  =  A[5]& A[4]&nA[3]& A[2]& A[1]&nA[0],
            B[55]  =  A[5]& A[4]&nA[3]& A[2]& A[1]& A[0],
            B[56]  =  A[5]& A[4]& A[3]&nA[2]&nA[1]&nA[0],
            B[57]  =  A[5]& A[4]& A[3]&nA[2]&nA[1]& A[0],
            B[58]  =  A[5]& A[4]& A[3]&nA[2]& A[1]&nA[0],
            B[59]  =  A[5]& A[4]& A[3]&nA[2]& A[1]& A[0],
            B[60]  =  A[5]& A[4]& A[3]& A[2]&nA[1]&nA[0],
            B[61]  =  A[5]& A[4]& A[3]& A[2]&nA[1]& A[0],
            B[62]  =  A[5]& A[4]& A[3]& A[2]& A[1]&nA[0],
            B[63]  =  A[5]& A[4]& A[3]& A[2]& A[1]& A[0];
endmodule


module RF_HALFADDER (A,B,S,C);

  input A,B;
  output C,S;

  xor g0 (S,A,B);
  and g1 (C,A,B);

endmodule


module RF_FULLADDER (A,B,Ci,S,Co);

  input A,B,Ci;
  output S,Co;

  wire T1,T2,T3;

  RF_HALFADDER ha0 (A,B,T1,T2);
  RF_HALFADDER ha1 (Ci,T1,S,T3);
  or g0 (Co,T2,T3);
endmodule


module RF_ADDER_6B (A,B,S,C);
```

```
  input [5:0] A,B;
  output [5:0] S;
  output C;

  wire [4:0] carry;

  RF_HALFADDER ha0 (A[0],B[0],S[0],carry[0]);
  RF_FULLADDER fa1 (A[1],B[1],carry[0],S[1],carry[1]);
  RF_FULLADDER fa2 (A[2],B[2],carry[1],S[2],carry[2]);
  RF_FULLADDER fa3 (A[3],B[3],carry[2],S[3],carry[3]);
  RF_FULLADDER fa4 (A[4],B[4],carry[3],S[4],carry[4]);
  RF_FULLADDER fa5 (A[5],B[5],carry[4],S[5],C);
endmodule


module RF_COMPARATOR_6B (A,B,AGTB,ALTB,AETB);

  input [5:0] A,B;
  output AGTB,ALTB,AETB;

  wire [5:0] x,A_nB,nA_B;
  wire [3:0] y;

  assign A_nB = A&(~B),
         nA_B = (~A)&B;

  assign y[3] = x[5]&x[4],
         y[2] = x[5]&x[4]&x[3],
         y[1] = x[5]&x[4]&x[3]&x[2],
         y[0] = x[5]&x[4]&x[3]&x[2]&x[1];

  assign x=A~^B,
         AETB = &x,
         AGTB=A_nB[5]|(x[5]&A_nB[4])|(y[3]&A_nB[3])|(y[2]&A_nB[2])|
                                     (y[1]&A_nB[1])|(y[0]&A_nB[0]),
         ALTB=nA_B[5]|(x[5]&nA_B[4])|(y[3]&nA_B[3])|(y[2]&nA_B[2])|
                                     (y[1]&nA_B[1])|(y[0]&nA_B[0]);
endmodule


module RF_COMPARATOR_7B (A,B,AGTB,ALTB,AETB);

  input [6:0] A,B;
  output AGTB,ALTB,AETB;

  wire [6:0] x,A_nB,nA_B;
  wire [4:0] y;
  assign A_nB = A&(~B),
         nA_B = (~A)&B;

  assign y[4] = x[6]&x[5],
         y[3] = x[6]&x[5]&x[4],
         y[2] = x[6]&x[5]&x[4]&x[3],
         y[1] = x[6]&x[5]&x[4]&x[3]&x[2],
         y[0] = x[6]&x[5]&x[4]&x[3]&x[2]&x[1];

  assign x=A~^B,
         AETB = &x,
         AGTB=A_nB[6]|(x[6]&A_nB[5])|(y[4]&A_nB[4])|(y[3]&A_nB[3])|
                                    (y[2]&A_nB[2])|(y[1]&A_nB[1])|
                                    (y[0]&A_nB[0]),
         ALTB=nA_B[6]|(x[6]&nA_B[5])|(y[4]&nA_B[4])|(y[3]&nA_B[3])|
                                    (y[2]&nA_B[2])|(y[1]&nA_B[1])|
                                    (y[0]&nA_B[0]);
endmodule


module RF_COMPARE_UNIT (INPUT1,NUMBER,INPUT2,AGTB,ALTB,AETB);

  input [5:0] INPUT1,NUMBER,INPUT2;
  output AGTB,ALTB,AETB;

  wire [5:0] A;
  wire C;
  reg [6:0] X,Y;

  RF_ADDER_6B ad0 (INPUT1,NUMBER,A,C);
  RF_COMPARATOR_7B comm0(X,Y,AGTB,ALTB,AETB);

  always @(A or C) begin
    X[6] = C;
    X[5:0] = A;
  end
  always @(INPUT2) begin
    Y[6]=0;
    Y[5:0] = INPUT2;
  end
endmodule
```

# Bibliography

[1] G.M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings, Apr. 18-20*, volume 30, pages 483–486. Atlantic City, N.J., 1967.

[2] Anujan Varma and C.S. Raghavendra, editor. *Interconnection Networks for Multiprocessors and Multicomputers Theory and Practice.* IEEE Computer Society Press, Los Alamitos, CA. 1994.

[3] Arvind and Robert A. Iannucci. Two Fundamental Issues in Multiprocessing. Technical report, MIT Laboratory for Computer Computer Science, May 1987. Computer Structures Group Memo 226-6.

[4] C. F. Joerg. Design and Implementation of a Packet Switching Routing Chip. Technical report. MIT Laboratory for Computer Science, 1990. TR 482.

[5] C. F. Joerg. The Monsoon Interconnection Network. In *Proceedings of the 1991 IEEE International Conference on Computer Design*, October 1991.

[6] Cypress Semiconductor Corporation. *Raceway Crossbar*. July 1995.

[7] David A. Patterson and John L. Hennessy, editor. *Computer Architecture: A Quantitative Approach. Second Edition*, pages 634–641. Morgan Kaufman Publishers, 1994.

[8] D.P. Agrawal. Graph Theoretical Analysis and Design of Multistage Interconnection Networks. *IEEE Transactions on Computers*, C-32(7):637–684, July 1983.

[9] M.J. Flynn. Very high-speed computers. In *Proceedings of IEEE*, volume 54, pages 1901–1909, December 1966.

[10] G. Andrew Boughton. Arctic Routing Chip. Technical report, MIT Laboratory for Computer Computer Science, March 1994. Computer Structures Group Memo 373.

[11] G. M. Papadopoulos and D. E. Culler. Monsoon. In *Proceedings of the 17th International Symposium on Computer Architecture*, Seattle, Washington, May 1990.

[12] GMD-FIRST, Berlin, Germany. *MANNA Hardware Reference Manual*, 1993.

[13] J.L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31(5):532–533, May 1988.

[14] Herbert H.J. Hum, Kevin B. Theobald and Guang R. Gao. Building multithreaded architectures with off-the-shelf microprocessors. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 288–294, Cancun, Mexico, April 1994. IEEE Computer Society.

[15] Herbert H.J. Hum, Oliver C. Maquelin and Guang R. Gao. Costs and Benefits of Multithreading with Off-the-Shelf RISC Processors. In *Proceedings of EURO-PAR'95*, pages 117–128. Springer-Verlag, August 1995.

[16] Kai Hwang and Fayé A. Briggs, editor. *Computer Architecture and Parallel Processing*, pages 1–49. McGraw-Hill Book Company, 1984.

[17] P. Kermani and L. Kleinrock. Virtual Cut-Through: A New Computer Communication Switching Technique. *Computer Networks*, pages 267–286, March 1979.

[18] Myricom Inc. *Myrinet Links and Routing*, 1994.

[19] M. C. Pease. The Indirect Binary n-Cube Microprocessor Array. *IEEE Transactions on Computers*, C-26(5):458–473, May 1977.

[20] Rishiyur S. Nikhil, Gregory M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 156–167, Gold Coast, Australia, May 1992.

[21] Yuval Tamir and Hsin-Chou Chi. Symmetric Crossbar Arbiters for VLSI Communication Switches. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):13–27, 1993.

IMAGE EVALUATION
TEST TARGET (QA-3)