(64)

Micro-Architectural Analysis of *SPACERAM* Processing Element

by

Amrit R. Pant

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 20, 1998

[illegible]

Author ....
Department of Electrical Engineering and Computer Science
August 20, 1998

[illegible]

Certified by [illegible]
Thomas F. Knight, Jr.
Thesis Supervisor

Certified by [illegible]
Norman Margolus
Thesis Supervisor

Accepted by [illegible]
Arthur C. Smith
Chairman, Department Committee on Graduate Thesis

Micro-Architectural Analysis of *SPACERAM* Processing Element

by

Amrit R. Pant

Submitted to the
Department of Electrical Engineering and Computer Science

August 20, 1998

In partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

SPACERAM is a SIMD architecture optimized for symbolic spatial computations implemented with multiple banks of DRAM combined with an array of processing elements. Such an architecture facilitates very high processor-memory bandwidth and hence allows for applications requiring orders of magnitude higher processing and update rates per DRAM than any current hardware. The array of processing elements process data coming simultaneously from several memory blocks by applying appropriate shifting and lookup table updates to them. Every processing element contains a permuter which makes it possible to assign data bits from any DRAM block to any functional block within the processing element as specified by controller setup. The lookup table is implemented as a common bus shared by all the processing elements. Micro-architectural analysis of such a processing element presents various possible implementations and trade-off issues associated with them.

Thesis Supervisors:

Thomas F. Knight, Jr.
Senior Research Scientist, MIT Artificial Intelligence Laboratory

Norman Margolus
Research Scientist, MIT Artificial Intelligence Laboratory

Acknowledgement

I am grateful to Tom Knight for providing me the opportunity and funding to work on this project. Special thanks to Norm Margolus for all his help and answers to my questions after questions. It was a great learning opportunity. Thanks also to Jeremy Brown for helpful discussions and hints on programming techniques.

Thanks to my roommate, Adrian Vetta, for several fruitful discussions. Thank you to Niranjan and Usha Pandey and Marga Pandey for their support and all the good times together throughout my stay at MIT. And I can not miss mentioning that I am continually grateful to my parents and brother for their understanding and closeness, and unending support.

This work is carried out as part of technology development for combined memory and logic architectures, support for which comes from DARPA contract F30602-98-1-0172.

Table of Contents

List of Figures

Chapter 1

Introduction

A computational analog of the uniform and simultaneous structure of physical laws in nature involves direct mapping of physical space and locality in memory organization and application of local relations through dedicated processing elements. *SPACERAM* is one such computational system which organizes symbolic spatial representations as memory contents linked in a virtual lattice type setup with possibly multiple dimensions [1].

Cellular Automata (CA) is such a computational model that defines *SPACERAM*. A CA model is a synchronous digital analog of the application of uniform and local laws in natural world that is operating everywhere in parallel [3]. Space is represented in such a model as cells, each of few bits in memory organized in a grid like fashion [5]. Each such memory cell is accessed by a processor which applies local rules to its content. The rules used by the processor are a set of uniform laws implemented as a shared lookup table. The processor computes new state for the cell in relation to its neighboring cells using the lookup table and updates corresponding memory. Hence, an initial arrangement of cell states will evolve with time [4].

Based on CA principles, *SPACERAM* chip is a combined large memory and processing array architecture implemented with embedded DRAM (Dynamic Random Access Memory) on an ASIC (Application Specific Integrated Circuit) [1]. The large on-chip memory consists of several blocks of DRAM which communicate individually to the processing elements. Several processing elements constitute an array of processor which read data bits from DRAM blocks, process them and write them back to memory. The *SPACERAM* chip also contains a multiplexer and a shifter associated with each DRAM block which allows all the processing elements of a chip to access shifted data coming from DRAM. A mesh I/O interface allows each processing element to exchange data bits with processing elements of neighboring chips. That is, several chips can be connected through this interface in a mesh type setup to form an array of higher dimension and increased computing resources. The chip also has a Direct Rambus interface which provides high transfer rate of data from on-chip memory to external larger memory. A controller block on the chip provides control setup for all other functional blocks.

The processing method implemented by *SPACERAM* is SIMD (Single Instruction Multiple Data) processing. All processing elements access data from DRAM blocks simultaneously, apply lookup table processing to them if necessary and write them back to corresponding memory blocks. The processing applied by every processing element is same under SIMD processing as the term itself suggests. The address for memory access in each DRAM block corresponding to each processing element is also same. Such SIMD processing is attractive for *SPACERAM* because the architecture allows for large amounts of data to be accessed simultaneously and processed locally before being written back to memory. The processing element of *SPACERAM* hence has functionalities to carry out SIMD processing based on shared lookup table.

The Processing Element (PE) of *SPACERAM* is the functional block that applies local rules to cell states read from DRAM blocks. PE consists of a permuter which can produce any permutation mapping of DRAM output so that data from any DRAM block can be assigned to any of the functionalities inside PE. Since PE receives data from DRAM blocks shifted as necessary, it allows for writing back of same data it receives as well as apply lookup table processing on the data before providing them for write-back. The lookup table is formed by entries from all PEs and is also shared by all of them for processing data. Several multiplexers carry out selection processes, conditions for which are supplied internally.

This micro-architectural analysis of PE looks at its overall setup from the point of view of optimal implementation. Since each functional block of PE can be implemented in various ways, there are several trade-off issues involved, especially in case of permuter. Given the technologies of DRAM and ASIC needed for *SPACERAM*, there are also restrictions on silicon area and power available for PE. The analysis thus provides suitable possible options for implementation of each block inside a PE and compares them under parameters of interest.

Chapter 2

Overview of *SPACERAM*

The basic principle behind the *SPACERAM* architecture is to access a large amount of memory and process it within a short enough period of time to achieve very high processing performance [1]. The current setup has 20 blocks of memory each with a 2 Kbit row size having an access time of 96 ns to achieve performance of 0.42 Tbit per second. Major functional blocks necessary to achieve such high performance are as shown in Figure 2.1.

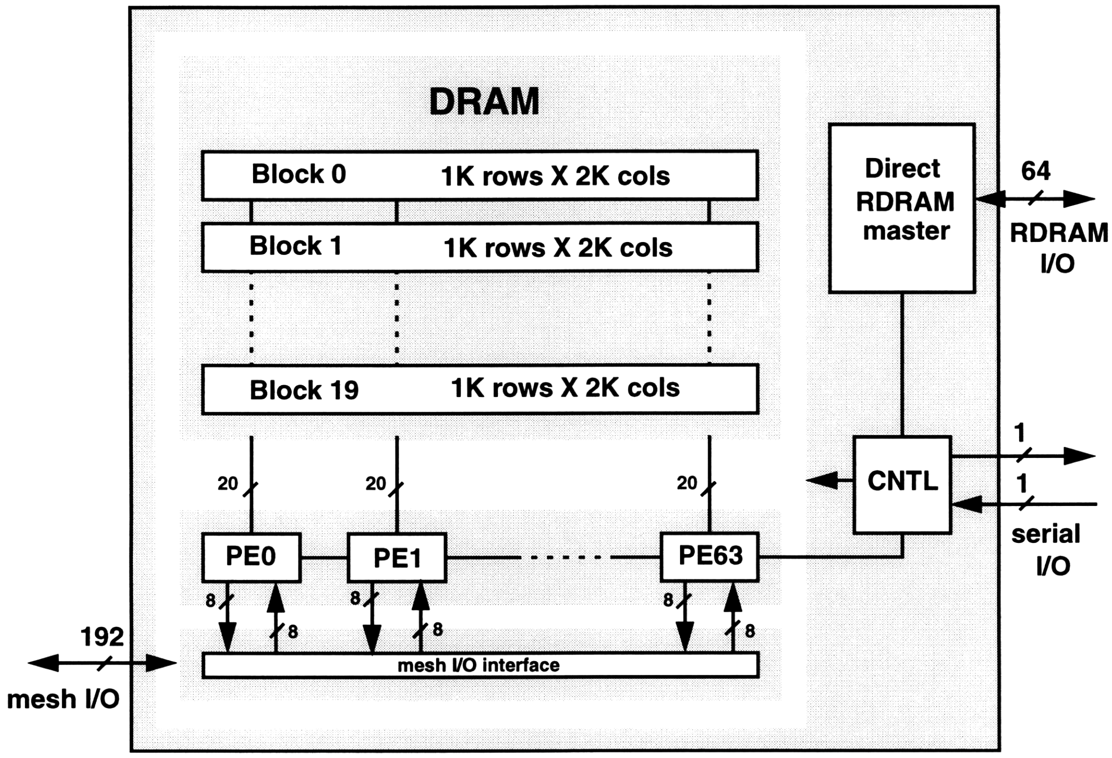

Figure 2.1: *SPACERAM* functional blocks

There are 20 memory blocks each containing a 2 Megabit DRAM segment plus ancillary circuitry needed to supply data as required by the processing elements. Each PE receives a bit-line from each memory block, hence making the word-line going to each PE 20 bits wide as shown. Each PE also provides two opposite directional 8 bit-lines to its corresponding mesh I/O interface to be able to shift bits to/from its neighboring chips. A Direct Rambus interface shown as Direct RDRAM master transfers large amount of data to/from DRAM in relatively short period of time, thus making it possible to swap parts of the memory in real time. The controller shown as CNTL generates all the control bits necessary for carrying out functionalities of the chip. The controller also has two uni-directional serial lines available for transferring initial setup information as well as for monitoring purposes.

2.1 Memory block

There are 20 memory blocks in a *SPACERAM* chip, and there are three major functional units in every memory block - DRAM segment, a multiplexer and a shifter as shown in Figure 2.2.

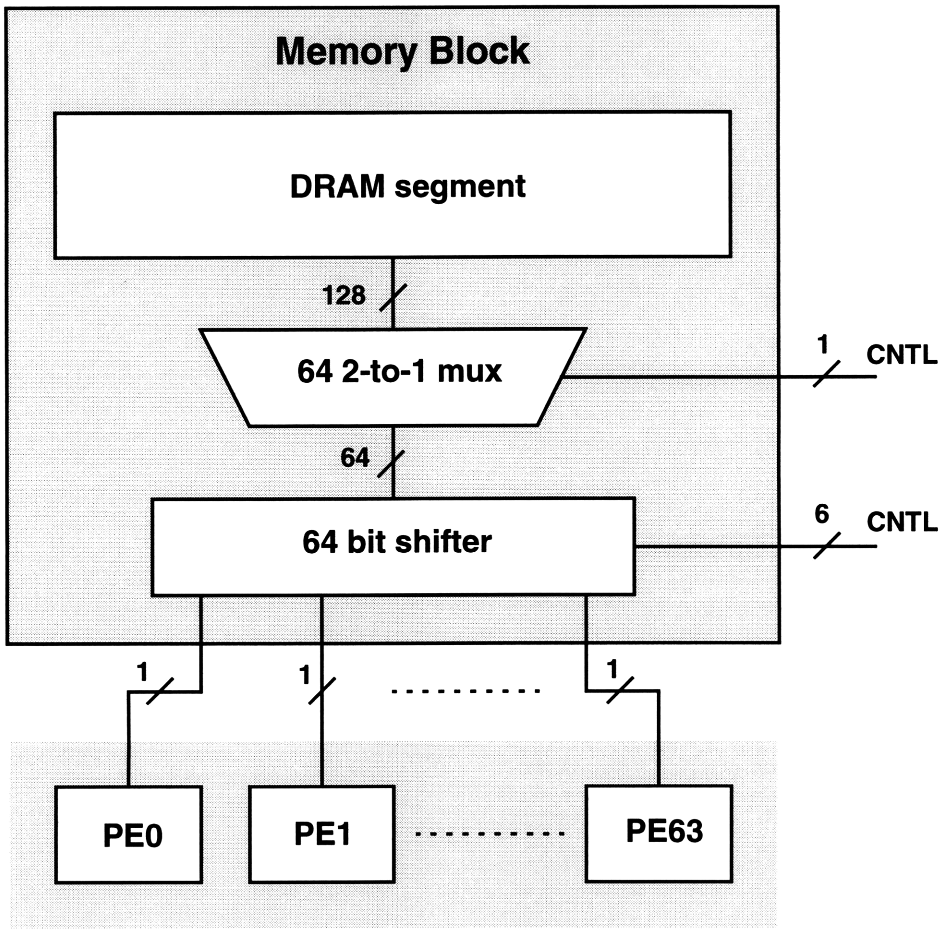

Figure 2.2: Structure of Memory block

The DRAM segment is the actual storage unit which holds cell state information between processing. The DRAM segment of one block has a storage capacity of 2 Megabits divided as 1K rows by 2K columns. To attain a row access time of 96 ns, 2K bits are divided into 128 bit chunks to be handed to PEs in one memory cycle and written back on completion of processing. Hence, 16 such cycles are necessary for accessing all of one DRAM row.

The 128 bits of memory content provided by a DRAM segment need to be provided to corresponding PE within one memory word cycle. However, there are only 64 PEs available at a time. Therefore, 64 2-to-1 multiplexers are used to route 64 bits at a time of the 128 bits to PEs through corresponding shifters. One bit of control specifies the selection on all 64 multiplexers. Having to multiplex the 128 bits of data coming from DRAM segment to produce two 64 bit sets also provides an indication to memory organization in DRAM. Rather than having two 64 bit halves within the accessed 128 bit word, it is more efficient to allow two adjacent locations to be multiplexed to same PE. This provides more regularity to the memory block structure and hence saves area in terms of needed wiring setup.

The shifter allows data coming from the DRAM segment through the multiplexers to be shifted in either direction as specified by the controller. Since the shifter takes 64 bits to be shifted to any possible position, 6 bits of control is necessary as shown in Figure 2.2. The basic functionality of the shifter, in other words, is to rotate incoming 64 bits to a desired sequence specified by the 6 control bits. One possible implementation for the shifter is based on barrel shifting [1].

2.2 Processing Element (PE)

SIMD processing of *SPACERAM* is based on lookup table processing. A lookup table consists of data bits coming from all 64 PEs. The lookup table accordingly extends among all PEs and is used as reference table to produce output data to be sent back to the memory block.

Every PE has a permuter element which can assign any incoming bit from memory block to any of the functional units of a PE. The permuter being bidirectional can also assign any of the bits generated by any functional unit within a PE to any of the outgoing lines. Hence, a PE can access any memory location within its corresponding memory block for any of its functional unit. Chapter 4 describes the permuter and its implementation in greater detail.

A PE also allows data to be passed on to its corresponding mesh I/O interface. Data bits coming back from the mesh I/O interface can be the same bits provided to it by PE or can contain new bits received from neighboring PEs on-chip or off-chip. Thus received data bits inside PE can be passed on as is to a memory block to be written back. In that case, no lookup table processing is applied. When lookup table processing is applied, a multiplexer is used to select data coming from lookup table to be passed on to the memory block.

A PE also has a serial I/O port to be used for various purposes including initial setup of the lookup table. A one-bit port is also provided by PE to transfer control sequence from memory block to the controller.

Chapter 3 describes PE structure in more detail.

2.3 Mesh I/O Interface

The Mesh I/O interface extends the virtual spatial coverage of the *SPACERAM* chip to include several such chips. The interface provides upto 32 bit links in 6 directions totalling 192 bits. Hence, the interface provides a way of setting up an array of *SPACERAM* chips to form a virtual lattice of multiple dimensions [1]. A two dimensional array of *SPACERAM* chips forming a virtual lattice of common spatial coverage is shown in Figure 2.3, where the smallest shaded squares represent unit of basic storage and processing [1].

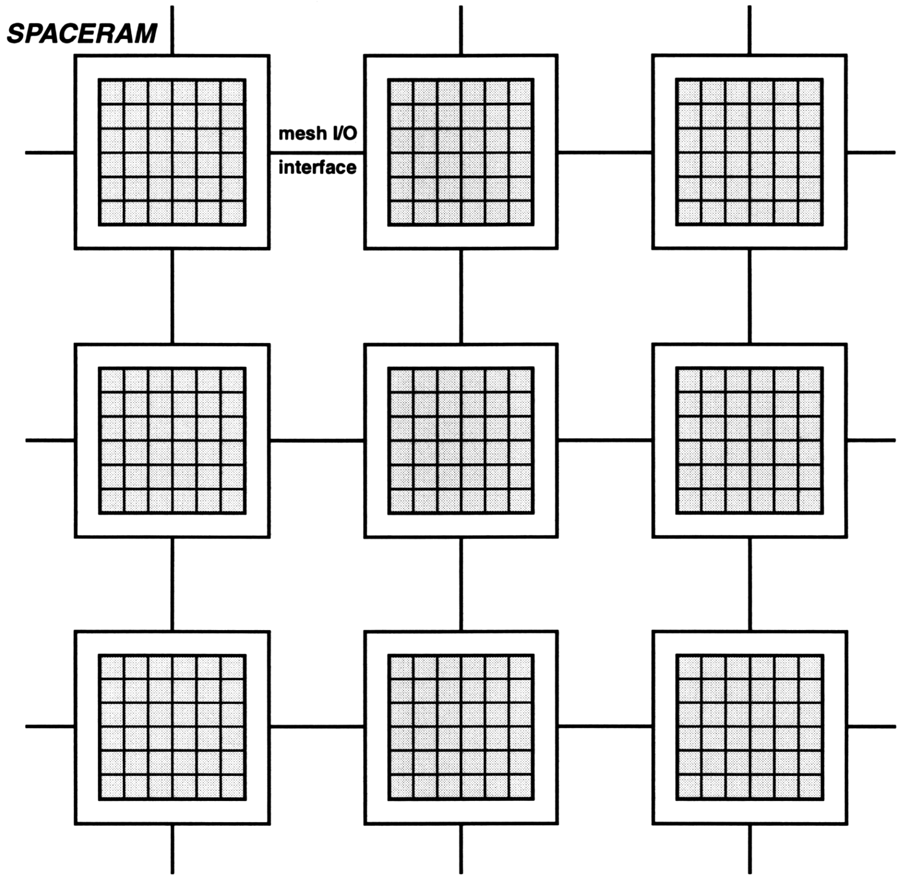

Figure 2.3: Two dimensional virtual lattice of *SPACERAM*

Every PE has its corresponding mesh I/O interface and only through that interface can it transfer bits to/from neighboring PE on-chip or off-chip. Due to continuity of spatial coverage of the virtual lattice, a PE passing bits to an adjacent PE through the mesh I/O interface should also receive equal number of bits from another adjacent PE through the interface.

2.4 Direct Rambus Interface

The Direct Rambus DRAM (RDRAM) interface is a commercially available product that can transfer data at 1.6 GByte per second in single channel setup [7]. The channel uses high speed signals to carry all address, data and control information. Direct RDRAM runs a two-byte wide data bus at 800 MHz to achieve the peak transfer rate of 1.6 GByte per second. The protocol used by Direct RDRAM is said to achieve 95% efficiency for 32 byte random transfers [7]. Figure 2.4 shows the single channel Direct RDRAM setup.

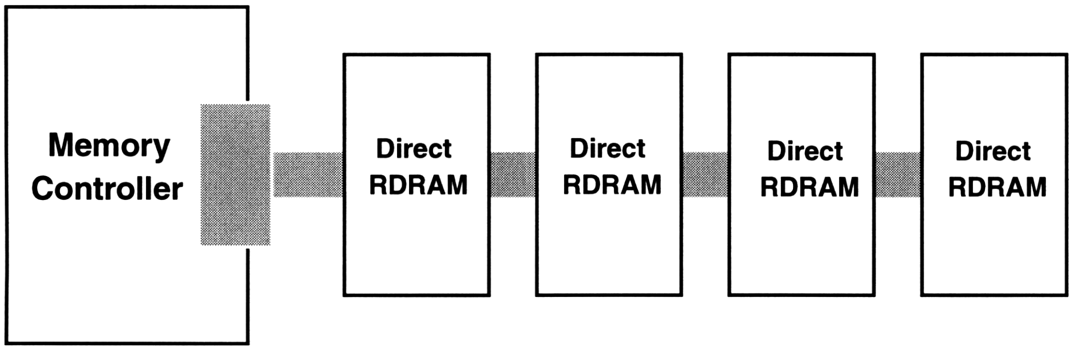

Figure 2.4: Single channel Direct RDRAM setup

The Direct RDRAM interface thus provides a large bit-pipe to the *SPACERAM* chip so that it can swap parts of memory while other parts of memory are being accessed. Such interface can also allow the on-chip memory of *SPACERAM* be used as a cache while a larger, cheap memory stays off-chip. Since such off-chip bandwidth is still two orders of magnitude less than on-chip bandwidth, applications requiring frequent off-chip access will be off-chip memory bandwidth limited even with Direct RDRAM interface [1].

2.5 Controller block

The Controller block provides the control words necessary to set states of all other functional blocks of *SPACERAM*. The controller receives control bits from memory through the PE's. Some examples of actions carried out by the controller block are address setup of each DRAM segment, permuter setting in every PE, enable/disable of lookup table update etc.

The controller block also has two uni-directional serial lines available for off-chip communication. The incoming line is used to setup initial control sequence of the chip. It can also be used to send requests to the controller to provide operational data when the chip is processing. The outgoing line is used to transfer the operational data so requested.

A whole DRAM row contains control information used by the controller. Hence, each column has one such bit accessed by each PE and supplied to the controller. The entire 2K control word so received by the controller can be applied repeatedly to subsequent row updates or it can be different as necessary [1].

Chapter 3

Processing Element

Each of 64 processing elements (PE) receives a bit-line from each of the 20 memory blocks. Hence, every PE has 20 bit input/output connection to the memory part of the chip. If we perceive all of the memory blocks as one DRAM block, then there is a 40K column spanning across all 20 DRAM segments (as each segment has column size of 2K). Therefore, a PE will be updating part of this 40K DRAM row. The row selection for this update by the PE's is controlled through the address generation for memory access. The method of update is by referencing the lookup table (LUT). Which bit-line from memory needs to be supplied to which functional unit inside the PE is determined by the permuter. An 8-bit 2-to-1 mux selects if any table lookup update is applied to 8-bit of data coming from the mesh I/O interface. Figure 3.1 presents the PE layout.

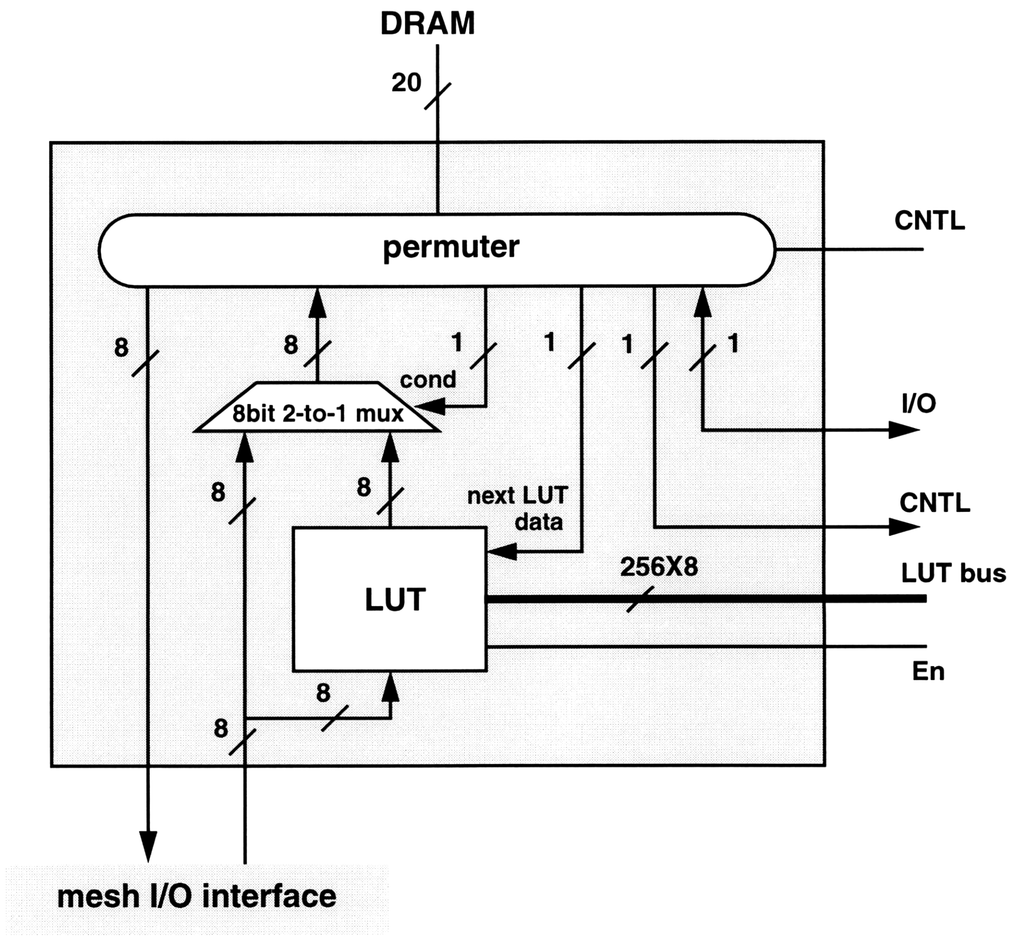

Figure 3.1: Functional blocks of PE

This chapter presents the functional implementation method used to represent PE units and describes the lookup table structure in detail. Chapter 4 looks at the permuter and its various implementation possibilities.

3.1 Functional implementation

Functional implementation emulates behavioral description of units of the PE. The Verilog hardware description language (HDL) has been used as the medium to carry out such implementations. As an example, the 8 bit 2-to-1 multiplexer that selects either of 8 bits coming directly from mesh I/O interface or through the lookup table can be implemented functionally in Verilog as follows.

```
module mux2in8bit(sel, out, in0, in1);

   input sel;
   input [7:0] in0, in1;
   output [7:0] out;

   assign out = sel ? in1 : in0;

endmodule /* mux2in8bit */
```

The priority of the representation given above is to carry out the assignment of output variable in the right order based on what the input variables are. The actual timing of assignment is irrelevant as long as the order is maintained. The functional characteristics of the above Verilog model is as shown in Figure 3.2.

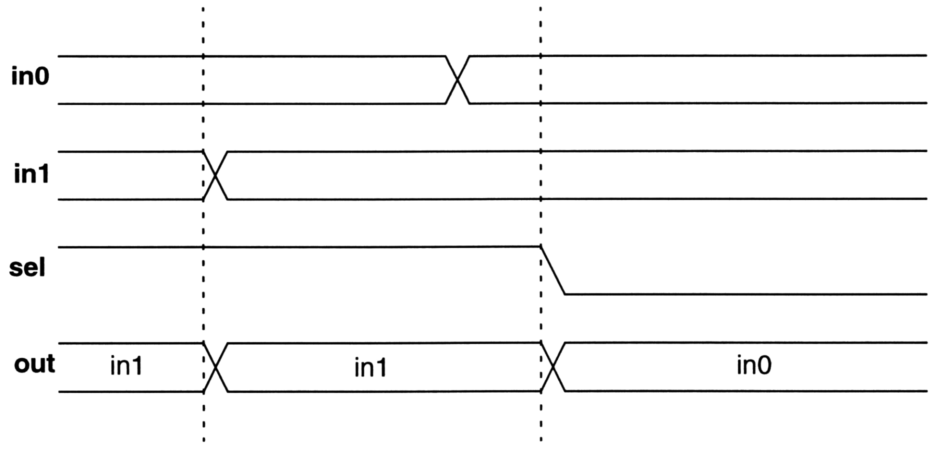

Figure 3.2: Functional behavior of 8 bit 2-to-1 multiplexer

3.2 Look-up table (LUT)

The implementation of the LUT is a shared 2048 bit bus spanning across all PEs. Since there are 64 PEs, each PE contributes 32 bits to the bus. For every access to the memory block by a PE, one bit is provided to the LUT towards setting up its 32-bit entry on the bus. The LUT block, as shown in Figure 3.3, collects these bits consecutively for 32 cycles before optionally driving it to the bus. The LUT block is double buffered so that it can load the next set of 32 bits while current 32 bits are being driven onto the bus. The lookup procedure is carried out by an 8 bit 256-to-1 multiplexer. The whole DRAM in the memory block has an entire row equal in size to the LUT bus. Hence, all data bits for setting up of the LUT bus come from one row of DRAM, although in 32 bit segments from each PE.

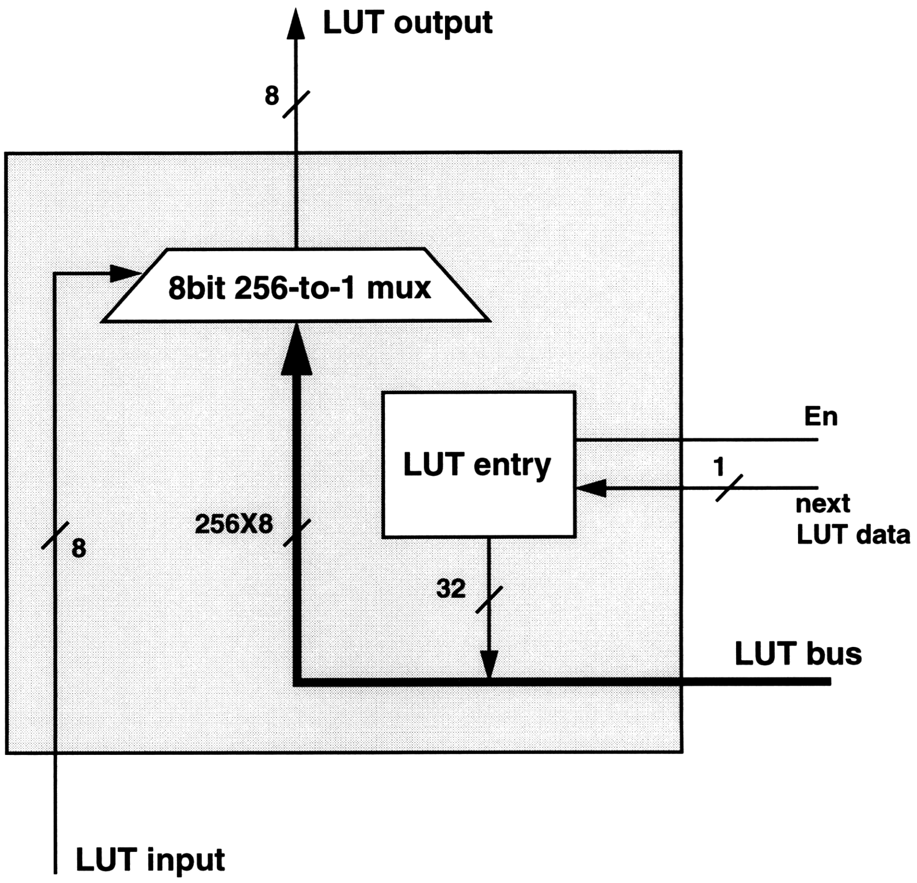

Figure 3.3: LUT block

As shown in Figure 3.3, there are two main functional units in the LUT block - an 8 bit 256-to-1 multiplexer and a LUT entry setup block. A Verilog HDL implementation of the multiplexer is presented in Appendix A.2. The principle behind the Verilog implementation is that a function called `selection` is called from the original module `mux256in8bit` with the LUT bus and selection byte as calling arguments. `selec-`

`tion` selects the appropriate byte from 256 bytes as specified by the selection byte and returns that byte to the calling module. The returned byte is then assigned to the LUT output lines.

The LUT entry setup block takes in a bit per PE cycle and accumulates them for 32 consecutive cycles. Then it drives that onto the LUT bus as its LUT entry if the LUT is changed by the controller. While the current 32 bits are being driven on the bus, the LUT entry setup block can take in bits to prepare the next 32 bit set. In other words, the LUT entry setup block is double buffered. There is also an enable (En) line that directs the LUT entry setup block when to read in the bit supplied to it by the permuter of its PE. When En is used to disable the block, the current 32 bit set is continuously being driven on the bus while no new bits are being read for next set. This occurs until En changes to enable status again, after which the LUT entry setup block continues exactly from the point when it was last disabled. That is, the next bit for the set is clocked-in at the position at which the last disabling happened. Figure 3.4 shows the functional implementation of the LUT entry setup block.

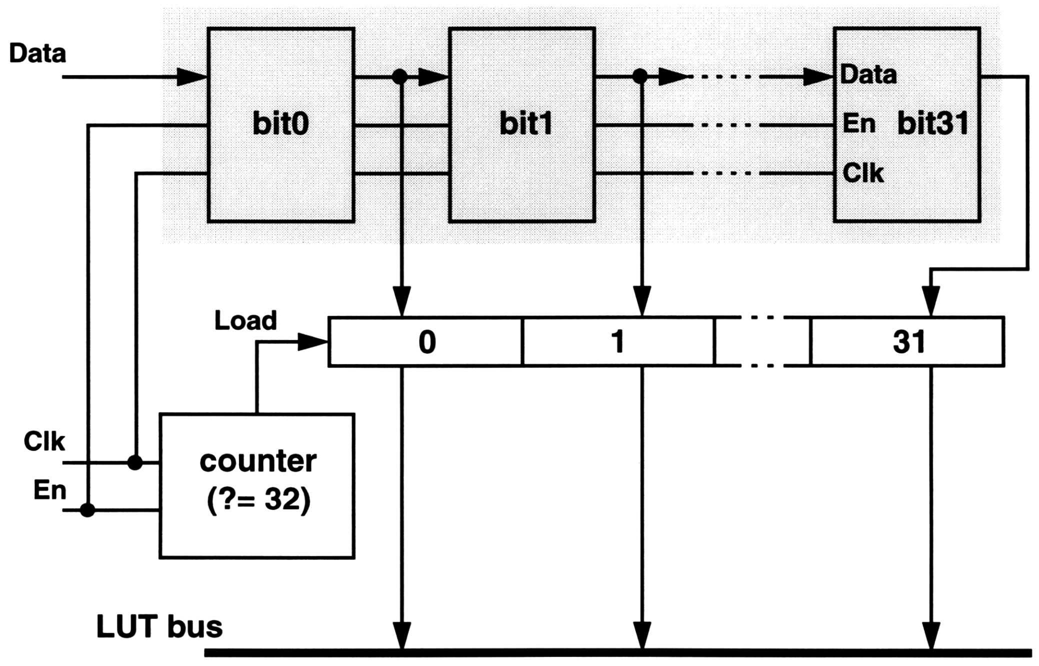

Figure 3.4: Functional representation of LUT entry setup block

A Verilog HDL implementation of the LUT entry setup block based on the model of Figure 3.4 is presented in Appendix A.3. The timing behavior of the implementation in comparison to the model is shown in Figure 3.5.

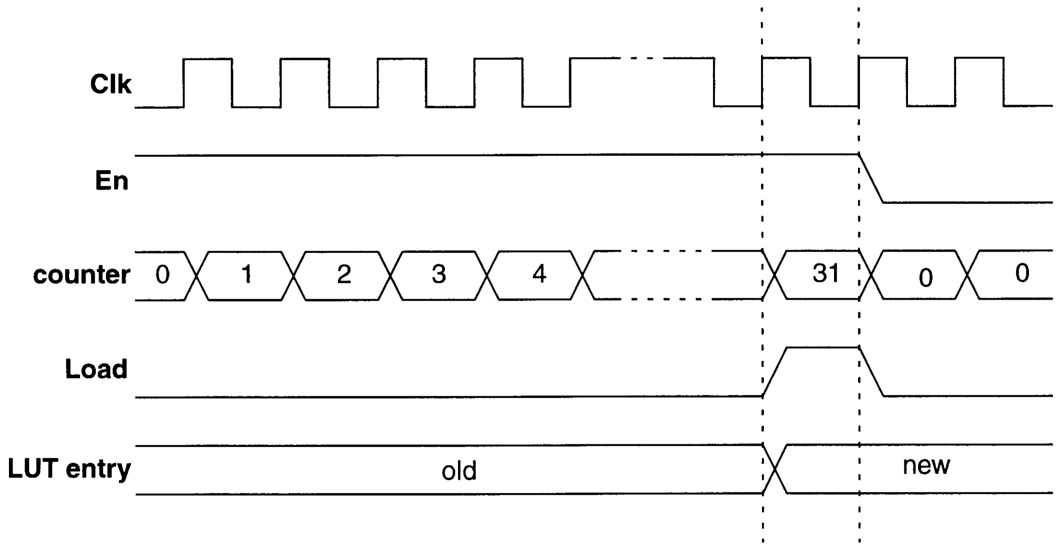

Figure 3.5: Functional behavior of LUT entry setup block implementation

Chapter 4

Permuter

4.1 Functionality

The permuter in every PE allows 20 bit-lines coming from the memory block to be assigned to any functional line inside the PE. Hence, the permuter is a rearrangeable switching network which produces a complete one-to-one mapping of bit-lines from two separate sets depending on the control setup supplied to it. Since some of the bit-lines are bi-directional, the permuter should be able to maintain the directionality of the line while assigning the permutation mapping among the lines. In other words, the permutation assignment among bit-lines of two separate sets (memory block side and internal PE side) should maintain the direction of data transfer on the lines. The permutation requirement is specified to the permuter by a control word supplied by the controller block external to the PE. The size of this control word varies depending on the implementation model.

Three different implementation models of a permuter based on these functionalities are presented here - cross bar network, multiplexer based network and Benes network.

4.2 Cross bar network

The cross bar network is a grid array network with switching element at every node. Hence, a switching element exists for every possible assignment of bit lines coming from two sets between which the permutation map has to be laid out.

In the case of the permuter in a PE, a cross bar network with 20 entries (each representing bit-lines coming from a memory block) in the domain set need to be mapped to any one of possible order of 20 entries (each representing internal bit-lines of PE) in the range set. Figure 4.1 shows such a 20 by 20 cross bar network as required for the permuter. Since every intersection of an entry in the domain to that in the range requires a switching element in the cross bar network, there are a total of 400 switching elements required for such an implementation. And since one bit of control is required to individually set one switching element, 400 control bits are also needed. With some additional hardware, 5 bits of control per column can be used to generate the right switch settings. Hence, a total of 100 control bits are necessary.

There is a large redundancy in switching resources available in the cross bar network at any given cycle of permutation mapping as in seen in Figure 4.1. On the bright side, implementation of cross bar network is regular compared to other networks discussed here which means area-wise efficiency in device level layout.

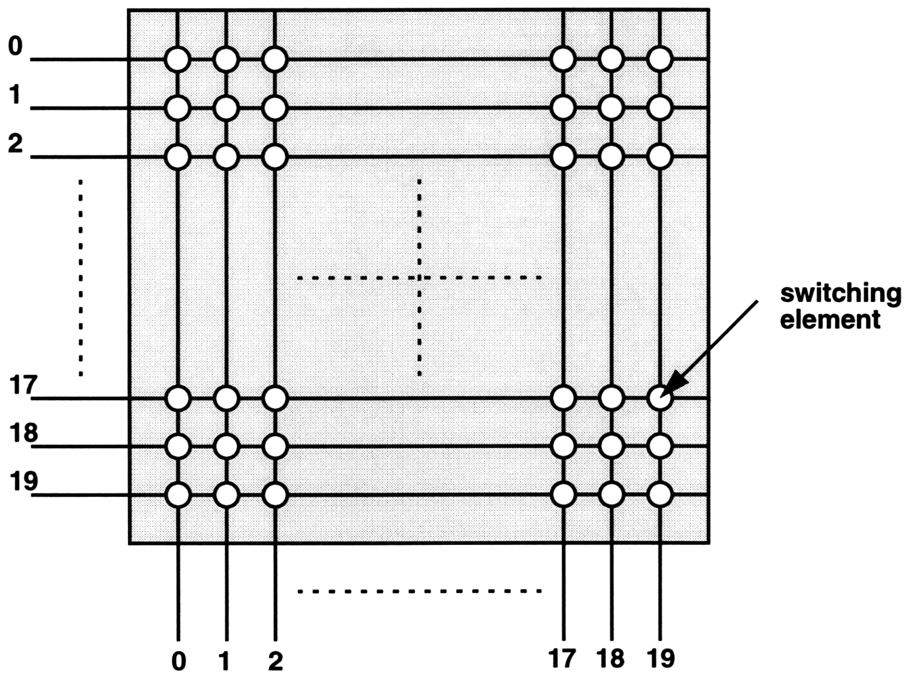

Figure 4.1: 20 by 20 Cross bar network (control lines not shown)

In terms of transistor count, one of the simplest implementations includes one NMOS pass-gate based switching element, which requires a total of 400 such transistors. A transmission gate in CMOS with one NMOS and one PMOS would give better signal quality at the expense of higher transistor count since an extra inverter is necessary per gate for such setup.

The permutation assignment timing delay associated with a cross bar network is also among the lowest because of its simplicity. The associated device delay is reduced since there is only one pass-gate switching element per permutation mapping assignment in the simplest implementation mentioned above. Bidirectionality is also easily implementable in cross bar networks as shown.

4.3 Multiplexer based permuter

The one-to-one permutation mapping being an inherently selection mechanism to associate an entry in the domain set to an entry in the range set, a multiplexer setup can be used for permuter implementation. A 20 entry domain-range permuter as needed for PE has one 20-to-1 multiplexer associated with every entry in the domain set. Then every entry in the range set will be connected to all multiplexers such that they can be assigned to any of the entries in the domain set. Each such multiplexer has a 5 bit selection input which specifies the mapping requirement. Figure 4.2 shows such permuter.

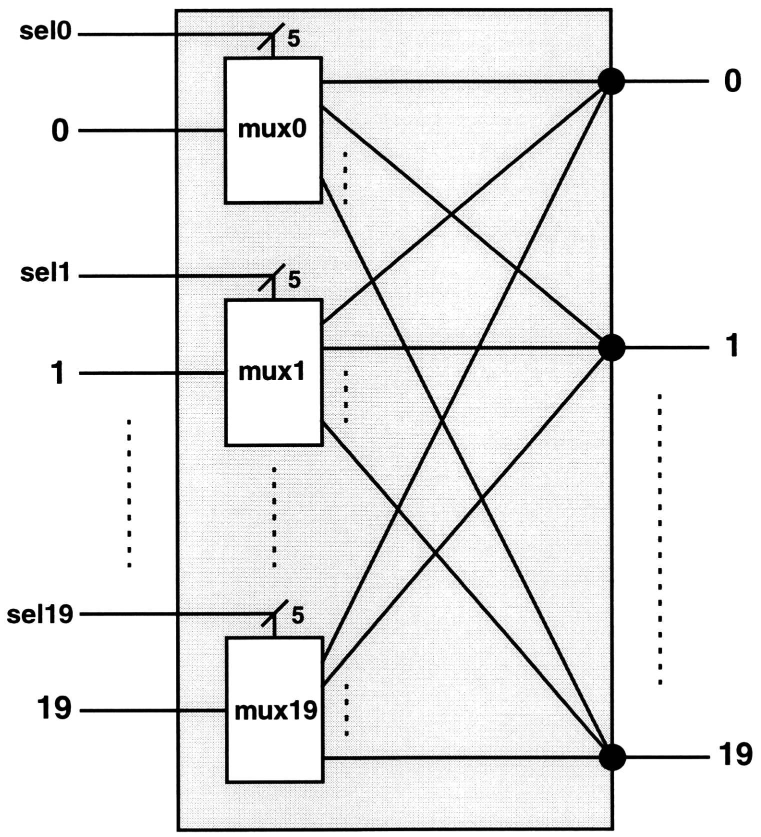

Figure 4.2: Multiplexer based 20 by 20 permuter

As is shown in Figure 4.2, each multiplexer requires 5 bit of selection input totalling the size of control word necessary to 100 bits. From the point of view of control word size, this implementation of permuter is one of the most efficient among those considered here.

In terms of device level implementation, a simple setup is considered which can be optimized to reduce transistor count. To maintain bidirectionality and to match the binary assignment of 5-bit selection word, a five transistor based pass gate has to be used per link in the multiplexer. That is, for every "0" in the selection word, a PMOS is used and for every "1" in the word, an NMOS is used in the link. For example, a link to be activated by selection word "00100" will be a pass-gate chain of two PMOS, one NMOS and two PMOS transistors in series. Therefore, one multiplexer requires 100 transistors (5 per link X 20 links) and the permuter in all requires 2000 transistors. An optimized setup can use a 2-to-1 selection gate per two entries and hence only require 38 transistors per multiplexer. Then total count is 760, which is still larger than in case of other networks discussed here. The bigger cost in device level implementation of such a permuter comes from the lack of regularity in the structure. Since a wire link has to be routed from every multiplexer to each entry bit-line in the range set (right side of Figure 4.2), there is greater requirement of silicon area for the whole extended structure. There is also a larger associate redundancy in the setup because only one link from twenty is used in each multiplexer for a given permutation mapping.

Another concern in the implementation of a multiplexer based permuter is that of assignment delay, and hence power requirement issues, associated with the structure. Since every signal has to pass through an extended pass-gate chain, it is going to have larger time delay. Drivers with more power can be used to overcome this delay which then translates to more area for the drivers. Intermediate buffering is not allowed in the chain to maintain bidirectionality. A bigger concern in terms of power requirement also comes from charging and discharging the extended wire routing needed to connect the multiplexers and bit-lines in the range set (as explained above). Given the diminishing size of transistors themselves, area and power issues are accentuated by this extended wiring requirement more than by the larger transistor count needed for the multiplexer based permuter. Power concern also comes from the fact that each entry in the range set has to drive 20 bit-lines going to all the multiplexers. Hence, bigger drivers are needed to keep up with timing requirement on these bit-lines.

4.4 Principle of Benes Network

A Benes network is a rearrangeable switching network based on 2-by-2 switching elements. It is a variation of the butterfly network consisting of two back to back butterfly networks [6]. Classification of such networks put Benes network as a member of Clos' type network [10]. The Benes network has been extensively studied for use in synchronous data permutation and in asynchronous interprocessor communication [12].

The fundamental unit of a Benes network is a 2-by-2 switching element having two states controlled through a one bit selection input. Figure 4.3 shows a functional model of such a switching element and its states with respect to the control bit.

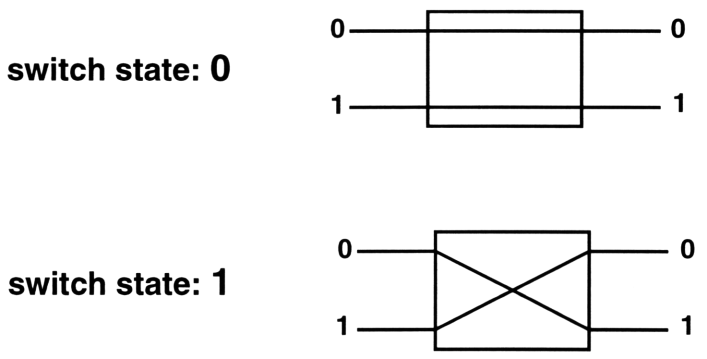

Figure 4.3: States (as specified by control bit) of switching element in a Benes network

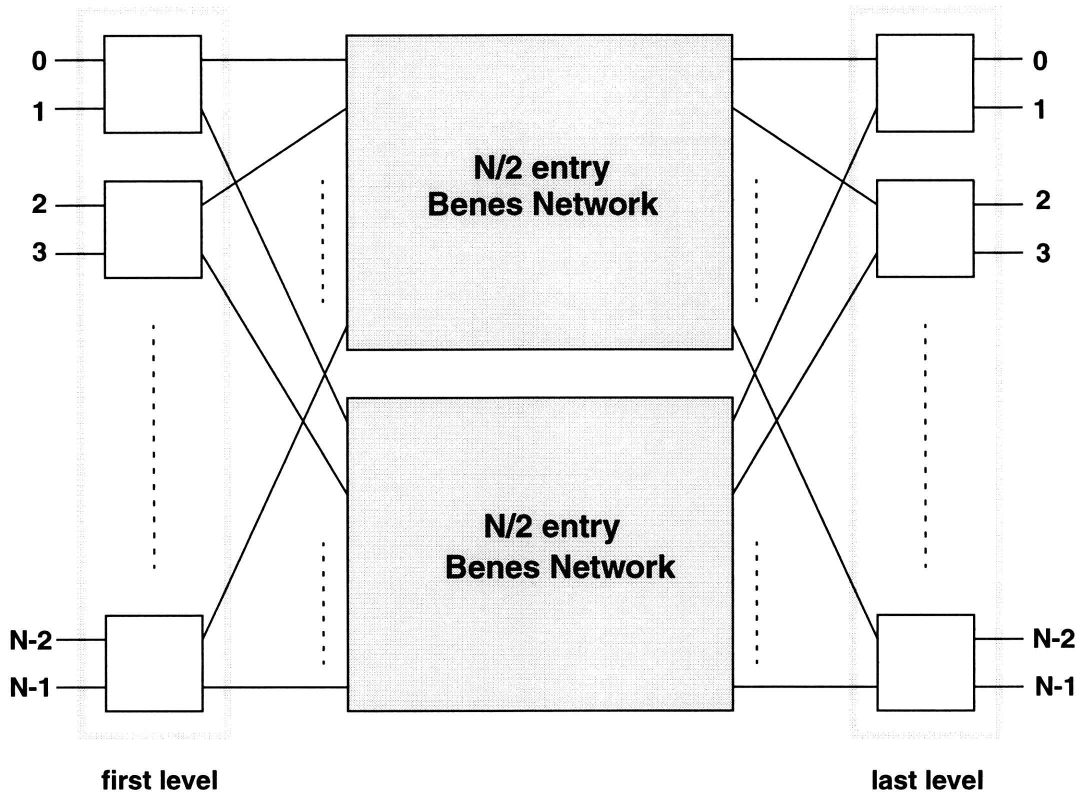

Figure 4.4: Recursive structure of N-entry Benes Network (N even)

Given the two state fundamental switching element, a Benes network is inherently a binary network with every level of network being divided into consequent base-2 structures. That is, the switching elements are connected in an order that produces a base-2 structure per level of the network necessary to fit all the entries in the permutation map set. Figure 4.4 shows the recursive base-2 structure of an N-entry (N here assumed to be exponential on 2, more on this below) Benes Network. [11] The *r*-dimensional network has *$2r+1$* level and 2^r nodes per level [6]. For instance, an 8 entry Benes network is 2 dimensional and has 5 level of switches each with 4 switching elements. Accordingly, an N-entry Benes network requires *N/2* switches per level and *$(2\log_2 N-1)$* levels, N being an exponential on 2 for such cases.

A Benes Network can also work as a permuter for a structure having number of entries that are non-exponential on 2 and even non-multiple of 2. So for permuter having even number of inputs, the basic structure of the network looks as in Figure 4.4 with repetitive recursion to follow to produce further levels of the network. However, the method of Figure 4.4 is not going to be sufficient to generate further network levels at some point when the number of entries to a sub-network becomes odd, and this is guaranteed to happen in our case. The separate setup necessary to produce such Benes network with odd number of entries N is shown in Figure 4.5. [8]

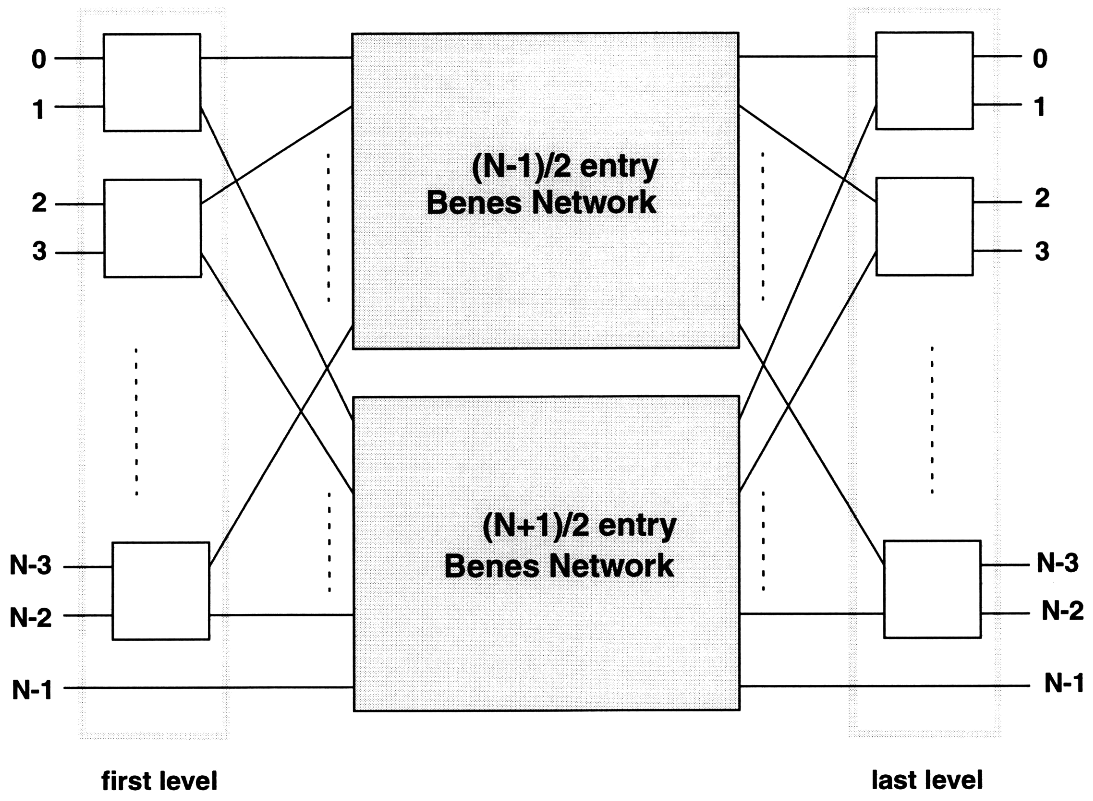

Figure 4.5: Benes network with odd number of entries N

The setup of Figure 4.5 with odd number of entries N is useful for optimization of Benes network based permuter for PE and will be looked at again in section 4.7.

The number of switching elements need for any number of entries N is given by - [8]

$$N\langle log_2 N\rangle - 2^{\langle log_2 N\rangle} + 1$$

where $\langle x\rangle$ is the smallest integer greater than x. The required number of switching elements is also shown to be bounded by $\langle log_2 N!\rangle$ [8]. The efficiency in terms of required resources (i.e. switches) per permuter greately increases as the number gets closer to this boundary.

As an example of Benes network as permuter, a 8-by-8 entry permuter is presented here and the method of setting all the switching elements is explained. As described above under recursive generation of the network, the complete layout of the 8-by-8 permuter looks as in Figure 4.6.

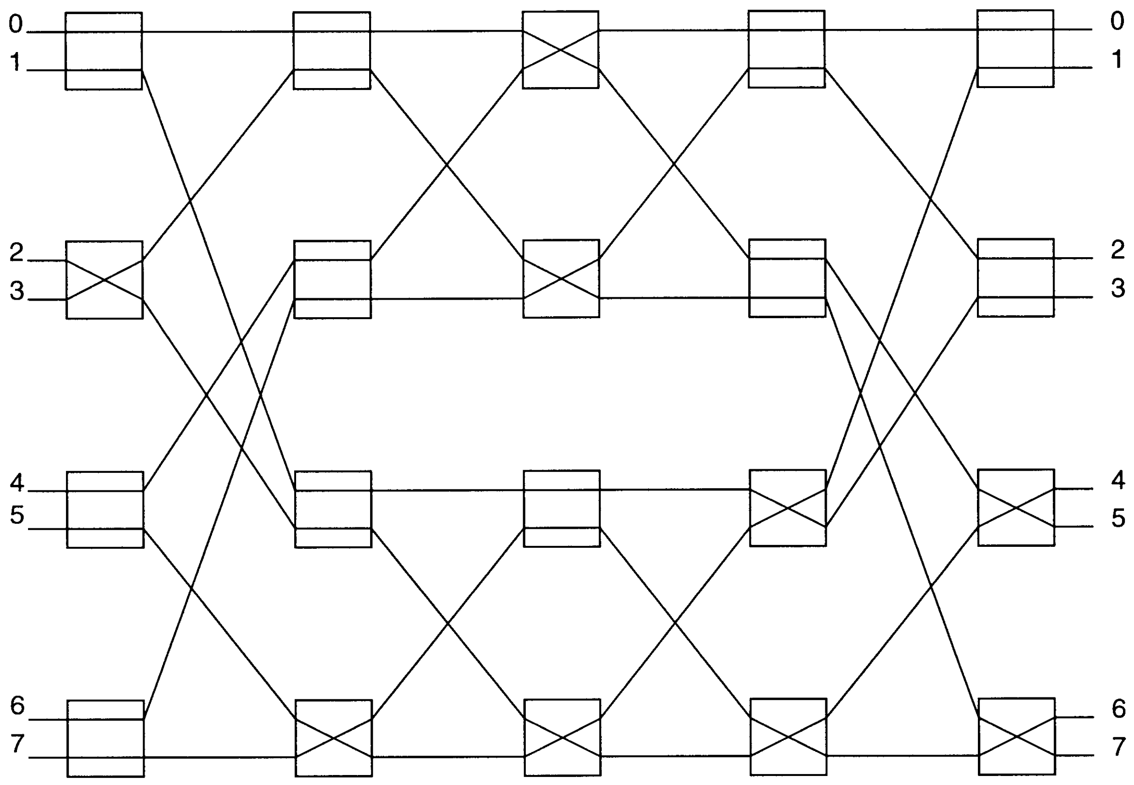

Figure 4.6: 8-by-8 Benes network permuter with switch states (control lines not shown)

The permutation map used by the 8-by-8 permuter of Figure 4.6 is -

$$P = \begin{pmatrix} 01234567 \\ 53470126 \end{pmatrix}$$

where the top corresponds to entries of the domain set (left side on Figure 4.6) and the bottom corresponds to entries of the range set (right side).

The basic principle behind determining the states of switching element in Benes network is that, for every pair of entries going to the same switch, if one of the entry is linked through upper sub-network then the other entry has to be routed through lower sub-network. The process of setting up of switches can be started arbitrarily from one of the entries on the domain set. In this case, entry 0 in the domain set is chosen and from the permutation map, it can be seen that it has to be linked to entry 5 in the range set. So the corresponding switch in first level is set to state 0, that in second level also to state 0, that in third level to state 1, then the next switch in the link in fourth level to state 0 and finally setting state 1 for final switch to reach entry 5. This way, a constraint has already been put on the corresponding switch at the last level that the other entry, i.e. 4, is routed to the lower sub-network. Therefore, the corresponding mapped entry in the domain set, i.e. 2, also has to be routed through the lower sub-network. So a link is followed by accordingly setting the switches in the lower sub-network to finally reach 2 in the domain set. Following on the same logic, now the constraint is on the adjacent entry to 2, i.e. 3, so that it has to be routed through the upper sub-network. The corresponding entry in the range set, 7, is reached thereafter by suitably setting switches in the upper sub-network. Hence, it can be seen that such cycles pursue until an already set switch in first level is reached during assignment. Such completion of cycle, however, does not insure that the permutation mapping is completed. Pre-mature cycle completion can occur depending on nature of the map itself. Hence, it is necessary to check to see if any switches in the first level are left unset. If there is/are such unset switch/es, then the same procedure is started by selecting arbitrarily one of the unassigned entries and repeating the same method as above. Depending on the given permutation mapping, several such cycles of assignment may be necessary to set all the switches. Figure 4.6 shows the final state of the network with all the switches properly set.

4.5 Benes network structure for 20-by-20 permuter

Based on the principle described in section 4.4, the 20 entry permuter necessary for the PE is constructed by first considering a 32-by-32 Benes network. N = 32 is being considered because of its being the next exponential on 2 after 20 such that both the implementation of the network as well as generation of control word is greatly simplified. Therefore, initially there are 16 switches per level and 9, i.e. ($2\log_2 32 - 1$) levels in the network. The routing between switches is done in base-2 structure recursively resulting in the network as shown in Figure 4.7.

For convenience of implementation in Verilog HDL, the switching elements of the network are identified in a grid like fashion as in *switch00* being the top switch of the first level, *switch80* being that of the last level, *switch8f* being the lowest switch of the last level etc. Verilog HDL of the network is presented in Appendix A.4. It also follows the same convention of naming the switches and their selection inputs (not shown in Figure 4.7). The entries of the domain set (left side of Figure 4.7) are represented as *a0, a1,* etc. and those of the range set (right side) are represented as *b0, b1,* etc. in the Verilog HDL implementation. Figure 4.7 also shows several switching elements with an asterix (*) marked on them. These switching elements are unnecessary for 20-by-20 permuter implementation and can be marked out from the 32-by-32 Benes Network leaving a total of 112 switches. Since the switches connected only to switches of previous level with asterix on them can also be neglected, the second and third level as well as seventh and eighth level also have switches with asterix on them. Verilog HDL shows instances of these asterix marked switches but they have been commented out. However, the asterix marked switches also need to be considered when generating control word, specially due to the fact that the control word generation method of section 4.6 generates control bits for each switch in continuous order. Therefore, compatibility has to be maintained in terms of switches that are marked out and the sequences of control bits provided to set switches of the network.

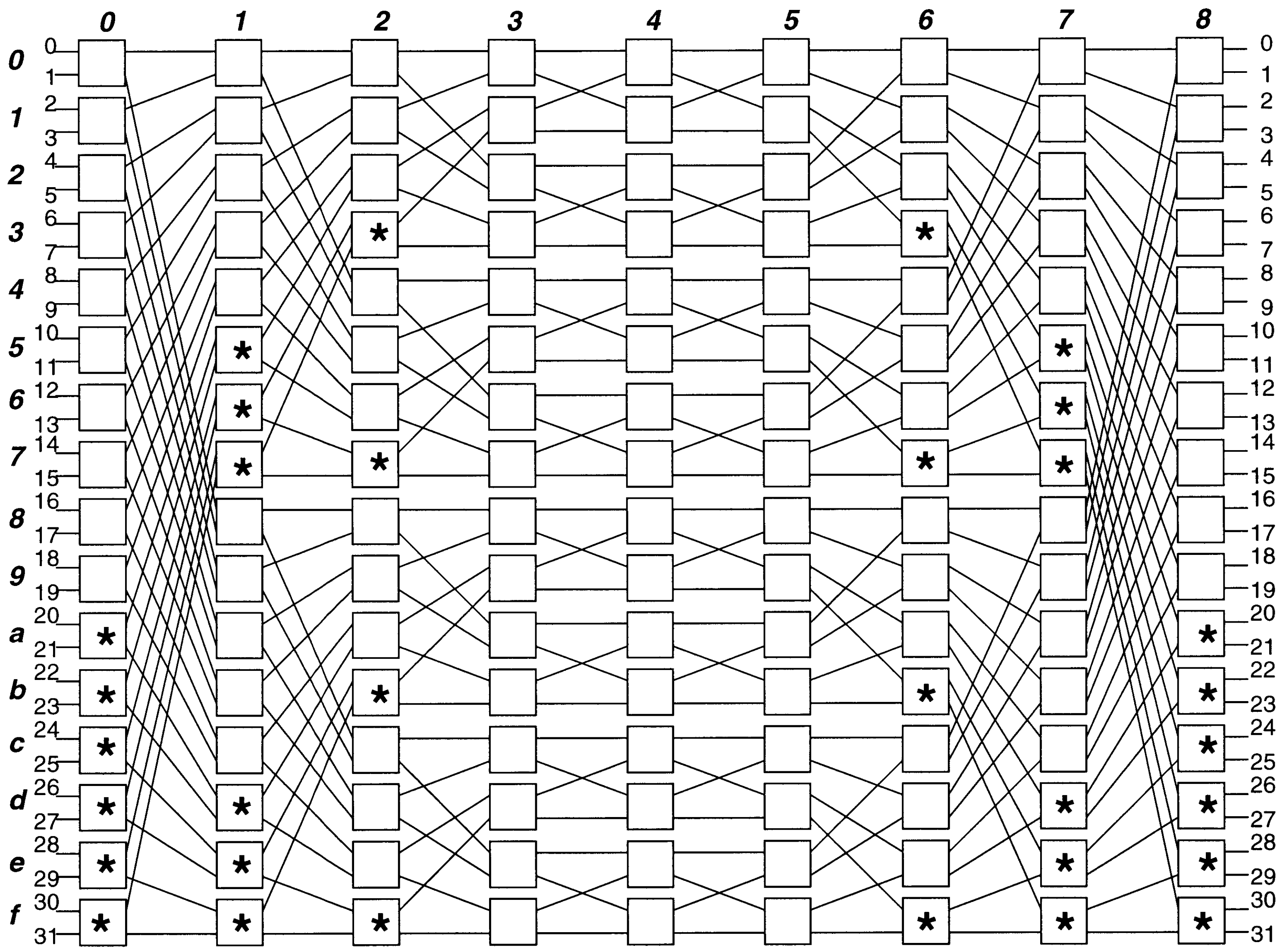

Figure 4.7: 32-by-32 Benes Network for 20-by-20 permuter

The Verilog HDL model first defines a switch module which carries out the functionality of a switching element as described in section 4.4 above. Then the model is instantiated for every necessary switch of the network with appropriate selection input and routing wires. The convention followed in the routing wire naming is that the first two digits after "w" represent the originating switch and the last two digits represent the target switch. For example, a wire going from *switch18* to *switch2c* is named *w182c*. The origin and target flow is set to be going from left to right. Bidirectionality is maintained throughout.

4.6 Control word generation for Benes Network

Because of binary nature of Benes network and base-2 structures being considered, control word generation method is also derived from recursive division by 2 within a given permutation map or its sub-map.

A given permutation map from a network can be described as

$$P = \begin{pmatrix} x_1 & x_2 & \cdots\cdots & x_N \\ \pi(x_1) & \pi(x_2) & \cdots\cdots & \pi(x_N) \end{pmatrix}$$

where x is an entry in the domain set and $\pi(x)$ is its corresponding entry in the range set according to given permutation map. The objective of control word generation algorithm is to derive local permutation maps for each of the sub-networks from P. The method used is that connection sets C_1 and C_2 are produced by decomposing P. [8] Then permutation set for each of the sub-networks (corresponding to P for the whole network) is constructed. The characteristics of the elements in the sub-network sets are then used to generate such permutation maps for the sub-networks. In case of base-2 structure, i.e. when total number of entries are exponential on 2, the decomposition of a network into two subnetworks involves assigning each entries of a pair to upper and lower network by setting switches of that level in proper states. Then regrouping of all the entries assigned to each sub-networks are done and the same procedure is started on each group again. Such group for a sub-network, say upper network J_u then satisfies following properties in the algorithm followed- [9]

1) $0 \in J_u$
2) $|J_u| = N/2$ where $|J_u|$ is the number of elements in J_u
3) if i and $j \in J_u$ and $i \neq j$, then $i \neq j^*$ and $E(i) \neq (E(j))^*$.

Here, i^* denotes the integer differing from i only in bit 0 of its binary representation. In other words, it represents the other entry which is also assigned to the same switch as i. $E(i)$ denotes the reverse mapping of i, i.e. $E(i) = \pi^{-1}(i)$. Iterative execution of such decomposition leads towards middle of the network and finally to an individual switch which can then be set to satisfy its corresponding connection set.

Appendix A.5 presents C programming language implementation of the algorithm to generate control word for 32-by-32 Benes network to be used for 20-by-20 permuter. That is, entries 20th through 31st are self-assigned and cannot be used in the permutation mapping to be provided. The implementation takes an entry by entry permutation map from 0 to 19 and produces a switch state control bit setup corresponding to layout of the network as shown in Figure 4.7. Here is an example of permutation map provided and the control sequence generated by the C implementation.

Permutation map: (where *D[i]* corresponds to i^{th} entry of domain set on the left side of Figure 4.7)

D[0] = 6	D[10] = 2
D[1] = 19	D[11] = 18
D[2] = 3	D[12] = 15
D[3] = 10	D[13] = 7
D[4] = 14	D[14] = 4
D[5] = 11	D[15] = 8
D[6] = 5	D[16] = 12
D[7] = 1	D[17] = 16
D[8] = 0	D[18] = 9
D[9] = 13	D[19] = 17

Control sequence generated: (Refer to Figure 4.3 and 4.7 for switch state and location)

```
0 0 0 0 0 0 1 0 0
0 0 1 0 0 0 1 1 1
1 0 0 0 0 1 0 1 1
0 0 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0 0
1 0 0 1 1 1 1 0 1
0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 1
0 0 0 0 1 1 0 1 1
1 0 0 0 0 0 0 1 0
0 1 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1 0
0 1 0 0 0 1 1 1 0
0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
```

The control word generated has been applied to the Verilog HDL implementation of section 4.5 to verify both of their functionalities. Appendix A.6 presents Verilog HDL model used for such purpose.

4.7 Issues and optimization of Benes Network as permuter

As seen in section 4.5 and 4.6, the size of control word for 20-by-20 permuter in 32-by-32 Benes network implementation is 112 bits. This size is relatively better compared to cross bar network but not as good as in the case of a multiplexer based permuter. However, there is the possibility of optimization in size of this control word in case of Benes network due to control word generation algorithm. Since the algorithm as presented in C implementation of Appendix A.5 sets the top-most switch of first level of every network or sub-network to "0", those bits in the control donot need to be included and the switches can be permanently set to such state. The number of such switches in the given setup being 15, the total number of control bits necessary becomes 97.

Another possibility of optimizing in terms of layout and control word size comes from using the non-base-2 structure of Benes network as discussed in section 4.4. Figure 4.8 presents such a structure which requires only 72 switching elements. The limitation boundary on the required number of switches, as discussed in section 4.4 is $<log_2 20!>$, which is about 61. Therefore, the structure of Figure 4.8 is a very efficient implementations of 20-by-20 permuter. The automatic generation of control word itself is, however, more complicated in this case than that of section 4.6.

The regularity of a Benes network is also a big attraction in terms of device level implementation. In one of the simpler implementations, the basic switching element is a four transistor (two NMOS, two PMOS) structure which makes the total transistor count for 112 switch network of section 4.6 add to 448. This number compares very well to those of both cross bar and multiplexer based permuter. Additional advantage in a device level implementation also comes from being a regular structure, although extended routing of control lines to every switch is necessary.

Because of the bidirectionality requirement, however, no buffering is permitted inside the network and this translates to larger delay in signal assignments in Benes network based permuter due to long chains of pass-gate switches. To keep up with this delay, more powerful drivers may be necessary which again raises the need for silicon area. A full CMOS transmission gate implementation can produce signals of higher quality in such case at an additional cost of larger transistor count.

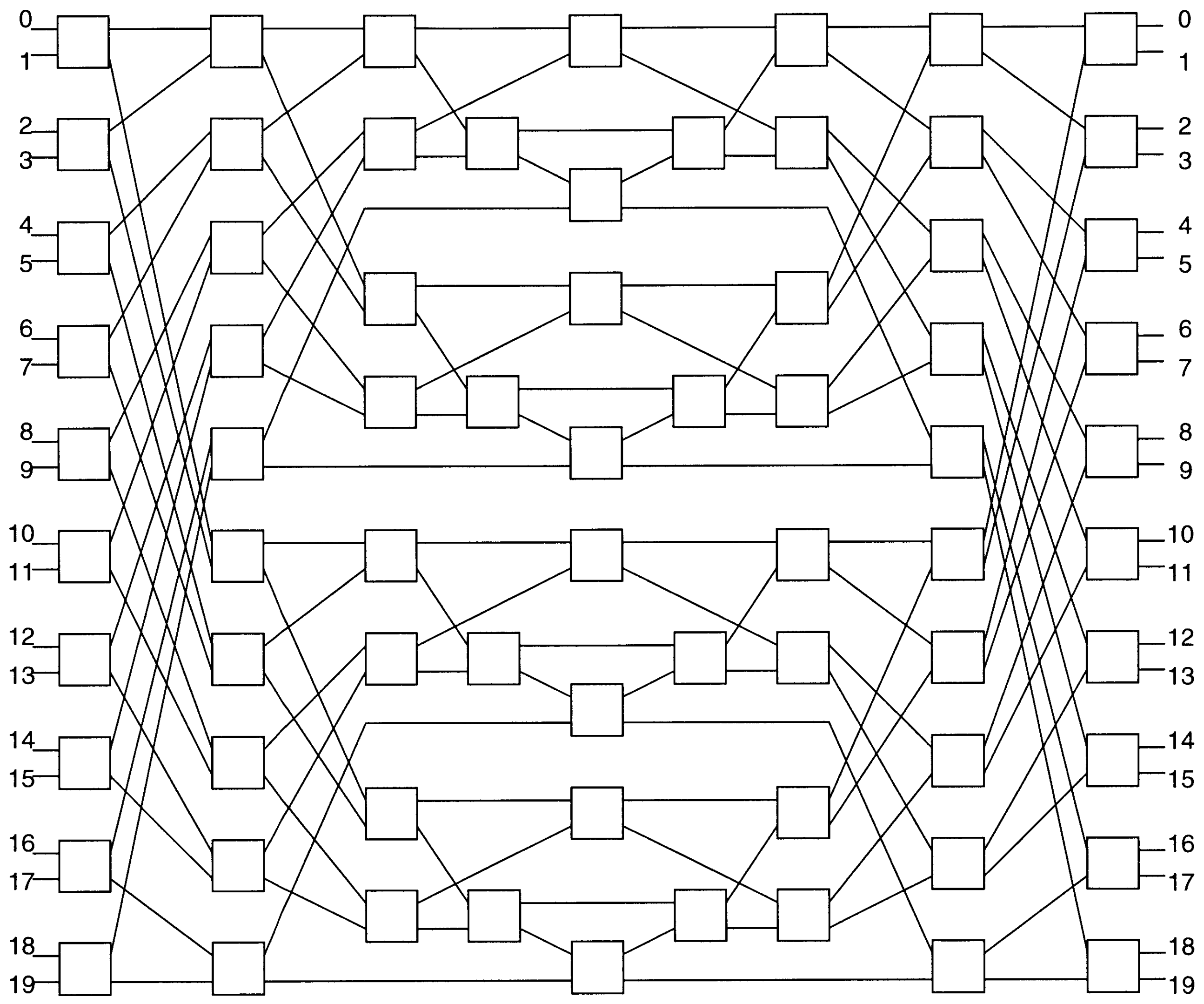

Figure 4.8: non-base-2 Benes network as 20-by-20 permuter

Chapter 5

Conclusion

The functional analysis of the Processing Element is determined mainly by the permuter and the lookuptable bus setup. The lower level implementation of the lookup table comes directly from its functional level description. In the case of the permuter, however, there are several possibilities. Depending on the parameter to be optimized and restrictions to be matched, a suitable implementation need to be selected.

The motivation for an optimized PE, and hence *SPACERAM* comes from looking at its potential computing power once constructed. Applications relying on the Cellular Automata model of computing are going to run on this platform with orders of magnitude higher performance. Large arrays of such chips can run large-scale lattice computations such as 3D lattice gas simulations of complex materials and fluids while a single chip can run complex image processing [1]. Applications like real time image manipulation as well as video manipulation are other possibilities. Large scale logic simulations as well as DSP-like applications will also be well suited for such a computer [2].

Future work on other blocks of *SPACERAM* will provide more light on any extra consideration to be given to block inside the PE or on any necessary modification to them. As technology issues like die size and pin count restrictions are decided upon, the structure of the PE will also become more solidified. Since there are multiple numbers of PE's on the chip, a regular structure will remain the priority from both area and power consideration points of view. As other issues like pipelining are considered, some of the restrictions on timing and hence area and power, loosen to some extent. The basic structure in terms of functionality will however remain the same and will only need to be optimized to meet the required criteria.

Appendix

A.1: 8-bit 2-to-1 multiplexer (Verilog HDL)

```
module mux2in8bit(sel, out, in0, in1);

   input sel;
   input [7:0] in0, in1;
   output [7:0] out;

   assign out = sel ? in1 : in0;

endmodule /* mux2in8bit */
```

A.2: 8-bit 256-to-1 multiplexer (Verilog HDL)

```
module mux256in8bit (sel, allin, out);

   input [7:0] sel;
   input [2047:0] allin;
   output [7:0] out;

   assign out = selection (sel, allin);

   function [7:0] selection;
      input [7:0] sel;
      input [2047:0] allin;

      case (sel)
        8'd0 : selection = allin[7:0];
        8'd1 : selection = allin[15:8];
         8'd2 : selection = allin[23:16];
         8'd3 : selection = allin[31:24];
         8'd4 : selection = allin[39:32];
         8'd5 : selection = allin[47:40];
         8'd6 : selection = allin[55:48];
         8'd7 : selection = allin[63:56];
         8'd8 : selection = allin[71:64];
         8'd9 : selection = allin[79:72];
         8'd10 : selection = allin[87:80];
         8'd11 : selection = allin[95:88];
         8'd12 : selection = allin[103:96];
         8'd13 : selection = allin[111:104];
         8'd14 : selection = allin[119:112];
         8'd15 : selection = allin[127:120];
         8'd16 : selection = allin[135:128];
         8'd17 : selection = allin[143:136];
         8'd18 : selection = allin[151:144];
         8'd19 : selection = allin[159:152];
         8'd20 : selection = allin[167:160];
         8'd21 : selection = allin[175:168];
         8'd22 : selection = allin[183:176];
         8'd23 : selection = allin[191:184];
```

```
8'd24 : selection = allin[199:192];
8'd25 : selection = allin[207:200];
8'd26 : selection = allin[215:208];
8'd27 : selection = allin[223:216];
8'd28 : selection = allin[231:224];
8'd29 : selection = allin[239:232];
8'd30 : selection = allin[247:240];
8'd31 : selection = allin[255:248];
8'd32 : selection = allin[263:256];
8'd33 : selection = allin[271:264];
8'd34 : selection = allin[279:272];
8'd35 : selection = allin[287:280];
8'd36 : selection = allin[295:288];
8'd37 : selection = allin[303:296];
8'd38 : selection = allin[311:304];
8'd39 : selection = allin[319:312];
8'd40 : selection = allin[327:320];
8'd41 : selection = allin[335:328];
8'd42 : selection = allin[343:336];
8'd43 : selection = allin[351:344];
8'd44 : selection = allin[359:352];
8'd45 : selection = allin[367:360];
8'd46 : selection = allin[375:368];
8'd47 : selection = allin[383:376];
8'd48 : selection = allin[391:384];
8'd49 : selection = allin[399:392];
8'd50 : selection = allin[407:400];
8'd51 : selection = allin[415:408];
8'd52 : selection = allin[423:416];
8'd53 : selection = allin[431:424];
8'd54 : selection = allin[439:432];
8'd55 : selection = allin[447:440];
8'd56 : selection = allin[455:448];
8'd57 : selection = allin[463:456];
8'd58 : selection = allin[471:464];
8'd59 : selection = allin[479:472];
8'd60 : selection = allin[487:480];
8'd61 : selection = allin[495:488];
8'd62 : selection = allin[503:496];
8'd63 : selection = allin[511:504];
```

```
8'd64 : selection = allin[519:512];
8'd65 : selection = allin[527:520];
8'd66 : selection = allin[535:528];
8'd67 : selection = allin[543:536];
8'd68 : selection = allin[551:544];
8'd69 : selection = allin[559:552];
8'd70 : selection = allin[567:560];
8'd71 : selection = allin[575:568];
8'd72 : selection = allin[583:576];
8'd73 : selection = allin[591:584];
8'd74 : selection = allin[599:592];
8'd75 : selection = allin[607:600];
8'd76 : selection = allin[615:608];
8'd77 : selection = allin[623:616];
8'd78 : selection = allin[631:624];
8'd79 : selection = allin[639:632];
8'd80 : selection = allin[647:640];
8'd81 : selection = allin[655:648];
8'd82 : selection = allin[663:656];
8'd83 : selection = allin[671:664];
8'd84 : selection = allin[679:672];
8'd85 : selection = allin[687:680];
8'd86 : selection = allin[695:688];
8'd87 : selection = allin[703:696];
8'd88 : selection = allin[711:704];
8'd89 : selection = allin[719:712];
8'd90 : selection = allin[727:720];
8'd91 : selection = allin[735:728];
8'd92 : selection = allin[743:736];
8'd93 : selection = allin[751:744];
8'd94 : selection = allin[759:752];
8'd95 : selection = allin[767:760];
8'd96 : selection = allin[775:768];
8'd97 : selection = allin[783:776];
8'd98 : selection = allin[791:784];
8'd99 : selection = allin[799:792];
8'd100 : selection = allin[807:800];
8'd101 : selection = allin[815:808];
8'd102 : selection = allin[823:816];
8'd103 : selection = allin[831:824];
```

```
8'd104 : selection = allin[839:832];
8'd105 : selection = allin[847:840];
8'd106 : selection = allin[855:848];
8'd107 : selection = allin[863:856];
8'd108 : selection = allin[871:864];
8'd109 : selection = allin[879:872];
8'd110 : selection = allin[887:880];
8'd111 : selection = allin[895:888];
8'd112 : selection = allin[903:896];
8'd113 : selection = allin[911:904];
8'd114 : selection = allin[919:912];
8'd115 : selection = allin[927:920];
8'd116 : selection = allin[935:928];
8'd117 : selection = allin[943:936];
8'd118 : selection = allin[951:944];
8'd119 : selection = allin[959:952];
8'd120 : selection = allin[967:960];
8'd121 : selection = allin[975:968];
8'd122 : selection = allin[983:976];
8'd123 : selection = allin[991:984];
8'd124 : selection = allin[999:992];
8'd125 : selection = allin[1007:1000];
8'd126 : selection = allin[1015:1008];
8'd127 : selection = allin[1023:1016];
8'd128 : selection = allin[1031:1024];
8'd129 : selection = allin[1039:1032];
8'd130 : selection = allin[1047:1040];
8'd131 : selection = allin[1055:1048];
8'd132 : selection = allin[1063:1056];
8'd133 : selection = allin[1071:1064];
8'd134 : selection = allin[1079:1072];
8'd135 : selection = allin[1087:1080];
8'd136 : selection = allin[1095:1088];
8'd137 : selection = allin[1103:1096];
8'd138 : selection = allin[1111:1104];
8'd139 : selection = allin[1119:1112];
8'd140 : selection = allin[1127:1120];
8'd141 : selection = allin[1135:1128];
8'd142 : selection = allin[1143:1136];
8'd143 : selection = allin[1151:1144];
```

```
8'd144 : selection = allin[1159:1152];
8'd145 : selection = allin[1167:1160];
8'd146 : selection = allin[1175:1168];
8'd147 : selection = allin[1183:1176];
8'd148 : selection = allin[1191:1184];
8'd149 : selection = allin[1199:1192];
8'd150 : selection = allin[1207:1200];
8'd151 : selection = allin[1215:1208];
8'd152 : selection = allin[1223:1216];
8'd153 : selection = allin[1231:1224];
8'd154 : selection = allin[1239:1232];
8'd155 : selection = allin[1247:1240];
8'd156 : selection = allin[1255:1248];
8'd157 : selection = allin[1263:1256];
8'd158 : selection = allin[1271:1264];
8'd159 : selection = allin[1279:1272];
8'd160 : selection = allin[1287:1280];
8'd161 : selection = allin[1295:1288];
8'd162 : selection = allin[1303:1296];
8'd163 : selection = allin[1311:1304];
8'd164 : selection = allin[1319:1312];
8'd165 : selection = allin[1327:1320];
8'd166 : selection = allin[1335:1328];
8'd167 : selection = allin[1343:1336];
8'd168 : selection = allin[1351:1344];
8'd169 : selection = allin[1359:1352];
8'd170 : selection = allin[1367:1360];
8'd171 : selection = allin[1375:1368];
8'd172 : selection = allin[1383:1376];
8'd173 : selection = allin[1391:1384];
8'd174 : selection = allin[1399:1392];
8'd175 : selection = allin[1407:1400];
8'd176 : selection = allin[1415:1408];
8'd177 : selection = allin[1423:1416];
8'd178 : selection = allin[1431:1424];
8'd179 : selection = allin[1439:1432];
8'd180 : selection = allin[1447:1440];
8'd181 : selection = allin[1455:1448];
8'd182 : selection = allin[1463:1456];
8'd183 : selection = allin[1471:1464];
```

```
8'd184 : selection = allin[1479:1472];
8'd185 : selection = allin[1487:1480];
8'd186 : selection = allin[1495:1488];
8'd187 : selection = allin[1503:1496];
8'd188 : selection = allin[1511:1504];
8'd189 : selection = allin[1519:1512];
8'd190 : selection = allin[1527:1520];
8'd191 : selection = allin[1535:1528];
8'd192 : selection = allin[1543:1536];
8'd193 : selection = allin[1551:1544];
8'd194 : selection = allin[1559:1552];
8'd195 : selection = allin[1567:1560];
8'd196 : selection = allin[1575:1568];
8'd197 : selection = allin[1583:1576];
8'd198 : selection = allin[1591:1584];
8'd199 : selection = allin[1599:1592];
8'd200 : selection = allin[1607:1600];
8'd201 : selection = allin[1615:1608];
8'd202 : selection = allin[1623:1616];
8'd203 : selection = allin[1631:1624];
8'd204 : selection = allin[1639:1632];
8'd205 : selection = allin[1647:1640];
8'd206 : selection = allin[1655:1648];
8'd207 : selection = allin[1663:1656];
8'd208 : selection = allin[1671:1664];
8'd209 : selection = allin[1679:1672];
8'd210 : selection = allin[1687:1680];
8'd211 : selection = allin[1695:1688];
8'd212 : selection = allin[1703:1696];
8'd213 : selection = allin[1711:1704];
8'd214 : selection = allin[1719:1712];
8'd215 : selection = allin[1727:1720];
8'd216 : selection = allin[1735:1728];
8'd217 : selection = allin[1743:1736];
8'd218 : selection = allin[1751:1744];
8'd219 : selection = allin[1759:1752];
8'd220 : selection = allin[1767:1760];
8'd221 : selection = allin[1775:1768];
8'd222 : selection = allin[1783:1776];
8'd223 : selection = allin[1791:1784];
```

```
            8'd224 : selection = allin[1799:1792];
            8'd225 : selection = allin[1807:1800];
            8'd226 : selection = allin[1815:1808];
            8'd227 : selection = allin[1823:1816];
            8'd228 : selection = allin[1831:1824];
            8'd229 : selection = allin[1839:1832];
            8'd230 : selection = allin[1847:1840];
            8'd231 : selection = allin[1855:1848];
            8'd232 : selection = allin[1863:1856];
            8'd233 : selection = allin[1871:1864];
            8'd234 : selection = allin[1879:1872];
            8'd235 : selection = allin[1887:1880];
            8'd236 : selection = allin[1895:1888];
            8'd237 : selection = allin[1903:1896];
            8'd238 : selection = allin[1911:1904];
            8'd239 : selection = allin[1919:1912];
            8'd240 : selection = allin[1927:1920];
            8'd241 : selection = allin[1935:1928];
            8'd242 : selection = allin[1943:1936];
            8'd243 : selection = allin[1951:1944];
            8'd244 : selection = allin[1959:1952];
            8'd245 : selection = allin[1967:1960];
            8'd246 : selection = allin[1975:1968];
            8'd247 : selection = allin[1983:1976];
            8'd248 : selection = allin[1991:1984];
            8'd249 : selection = allin[1999:1992];
            8'd250 : selection = allin[2007:2000];
            8'd251 : selection = allin[2015:2008];
            8'd252 : selection = allin[2023:2016];
            8'd253 : selection = allin[2031:2024];
            8'd254 : selection = allin[2039:2032];
            8'd255 : selection = allin[2047:2040];
        endcase /* sel */
    endfunction

endmodule /* mux256in8bit */
```

A.3: LUT entry setup block (Verilog HDL)

```
module lutbyte (data, en, clk, lutbyte);
   input data, en, clk;
   output [31:0] lutbyte;
   reg [31:0] lutbyte;
   reg bit0, bit1, bit2, bit3, bit4, bit5,
   bit6, bit7, bit8, bit9, bit10, bit11,
   bit12, bit13, bit14, bit15, bit16, bit17,
   bit18, bit19, bit20, bit21, bit22, bit23,
   bit24, bit25, bit26, bit27, bit28, bit29,
   bit30, bit31;
   reg [4:0] count;

   initial
     count = 5'b11111;
   always @ (posedge clk)
     begin
        if (en == 1)
         begin
   {bit0, bit1, bit2, bit3, bit4, bit5, bit6, bit7,
   bit8, bit9, bit10, bit11, bit12, bit13, bit14, bit15,
   bit16, bit17, bit18, bit19, bit20, bit21, bit22, bit23,
   bit24, bit25, bit26, bit27, bit28, bit29, bit30, bit31}
   = {data, bit0, bit1, bit2, bit3, bit4, bit5, bit6,
   bit7, bit8, bit9, bit10, bit11, bit12, bit13, bit14,
   bit15, bit16, bit17, bit18, bit19, bit20, bit21, bit22,
   bit23, bit24, bit25, bit26, bit27, bit28, bit29,
   bit30};
            count = count + 1;
            if (count == 5'b11111)
   lutbyte = {bit31, bit30, bit29, bit28, bit27, bit26,
   bit25, bit24, bit23, bit22, bit21, bit20, bit19, bit18,
   bit17, bit16, bit15, bit14, bit13, bit12, bit11, bit10,
   bit9, bit8, bit7, bit6, bit5, bit4, bit3, bit2, bit1,
   bit0};
         end
     end
endmodule
```

A.4: 32-by-32 Benes network for 20-by-20 permuter (Verilog HDL)

```
   module switch (select, inout0, inout1, outin0, outin1);

      input select;
      inout inout0, inout1, outin0, outin1;

      tranif0 sw0 (inout0, outin0, select);
      tranif0 sw1 (inout1, outin1, select);
      tranif1 sw2 (inout0, outin1, select);
      tranif1 sw3 (inout1, outin0, select);

   endmodule /* switch */

   module perm20by20 (a0, a1, a2, a3, a4, a5, a6, a7, a8, a9,
      a10, a11, a12, a13, a14, a15, a16, a17, a18, a19,
    b0, b1, b2, b3, b4, b5, b6, b7, b8, b9,
    b10, b11, b12, b13, b14, b15, b16, b17, b18, b19,
    sel00, sel01, sel02, sel03, sel04, sel05, sel06, sel07,
sel08, sel09,
     // break
    sel10, sel11, sel12, sel13, sel14, // break
    sel18, sel19, sel1a, sel1b, sel1c, // break
    sel20, sel21, sel22, // break
    sel24, sel25, sel26, // break
    sel28, sel29, sel2a, // break
    sel2c, sel2d, sel2e, // break
    sel30, sel31, sel32, sel33, sel34, sel35, sel36, sel37,
sel38, sel39,
    sel3a, sel3b, sel3c, sel3d, sel3e, sel3f,
    sel40, sel41, sel42, sel43, sel44, sel45, sel46, sel47,
sel48, sel49,
    sel4a, sel4b, sel4c, sel4d, sel4e, sel4f,
    sel50, sel51, sel52, sel53, sel54, sel55, sel56, sel57,
sel58, sel59,
    sel5a, sel5b, sel5c, sel5d, sel5e, sel5f,
    sel60, sel61, sel62, // break
    sel64, sel65, sel66, // break
   sel68, sel69, sel6a, // break
   sel6c, sel6d, sel6e, // break
   sel70, sel71, sel72, sel73, sel74, // break
   sel78, sel79, sel7a, sel7b, sel7c, // break
   sel80, sel81, sel82, sel83, sel84, sel85, sel86, sel87,
sel88, sel89
    // break
   );

     inout
   a0, a1, a2, a3, a4, a5, a6, a7, a8, a9,
   a10, a11, a12, a13, a14, a15, a16, a17, a18, a19,
   b0, b1, b2, b3, b4, b5, b6, b7, b8, b9,
   b10, b11, b12, b13, b14, b15, b16, b17, b18, b19;

     input
       sel00, sel01, sel02, sel03, sel04, sel05, sel06,
sel07, sel08, sel09,
      // break
      sel10, sel11, sel12, sel13, sel14, // break
      sel18, sel19, sel1a, sel1b, sel1c, // break
      sel20, sel21, sel22, // break
      sel24, sel25, sel26, // break
      sel28, sel29, sel2a, // break
      sel2c, sel2d, sel2e, // break
        sel30, sel31, sel32, sel33, sel34, sel35, sel36,
sel37, sel38, sel39,
      sel3a, sel3b, sel3c, sel3d, sel3e, sel3f,
        sel40, sel41, sel42, sel43, sel44, sel45, sel46,
sel47, sel48, sel49,
      sel4a, sel4b, sel4c, sel4d, sel4e, sel4f,
        sel50, sel51, sel52, sel53, sel54, sel55, sel56,
sel57, sel58, sel59,
      sel5a, sel5b, sel5c, sel5d, sel5e, sel5f,
      sel60, sel61, sel62, // break
      sel64, sel65, sel66, // break
      sel68, sel69, sel6a, // break
      sel6c, sel6d, sel6e, // break
      sel70, sel71, sel72, sel73, sel74, // break
      sel78, sel79, sel7a, sel7b, sel7c, // break
```

```
        sel80,  sel81,  sel82,  sel83,  sel84,  sel85,  sel86,
sel87, sel88, sel89;
      // break

      switch sw00 (sel00, a0, a1, w0010, w0018);
      switch sw01 (sel01, a2, a3, w0110, w0118);
      switch sw02 (sel02, a4, a5, w0211, w0219);
      switch sw03 (sel03, a6, a7, w0311, w0319);
      switch sw04 (sel04, a8, a9, w0412, w041a);
      switch sw05 (sel05, a10, a11, w0512, w051a);
      switch sw06 (sel06, a12, a13, w0613, w061b);
      switch sw07 (sel07, a14, a15, w0713, w071b);
      switch sw08 (sel08, a16, a17, w0814, w081c);
      switch sw09 (sel09, a18, a19, w0914, w091c);
   //   switch sw0a (sel0a, a20, a21, w0a15, w0a1d);
   //   switch sw0b (sel0b, a22, a23, w0b15, w0b1d);
   //   switch sw0c (sel0c, a24, a25, w0c16, w0c1e);
   //   switch sw0d (sel0d, a26, a27, w0d16, w0d1e);
   //   switch sw0e (sel0e, a28, a29, w0e17, w0e1f);
   //   switch sw0f (sel0f, a30, a31, w0f17, w0f1f);

      switch sw10 (sel10, w0010, w0110, w1020, w1024);
      switch sw11 (sel11, w0211, w0311, w1120, w1124);
      switch sw12 (sel12, w0412, w0512, w1221, w1225);
      switch sw13 (sel13, w0613, w0713, w1321, w1325);
      switch sw14 (sel14, w0814, w0914, w1422, w1426);
   //   switch sw15 (sel15, w0a15, w0b15, w1522, w1526);
   //   switch sw16 (sel16, w0c16, w0d16, w1623, w1627);
   //   switch sw17 (sel17, w0e17, w0f17, w1723, w1727);

      switch sw18 (sel18, w0018, w0118, w1828, w182c);
      switch sw19 (sel19, w0219, w0319, w1928, w192c);
      switch sw1a (sel1a, w041a, w051a, w1a29, w1a2d);
      switch sw1b (sel1b, w061b, w071b, w1b29, w1b2d);
      switch sw1c (sel1c, w081c, w091c, w1c2a, w1c2e);
   //   switch sw1d (sel1d, w0a1d, w0b1d, w1d2a, w1d2e);
   //   switch sw1e (sel1e, w0c1e, w0d1e, w1e2b, w1e2f);
   //   switch sw1f (sel1f, w0e1f, w0f1f, w1f2b, w1f2f);

      switch sw20 (sel20, w1020, w1120, w2030, w2032);
   switch sw21 (sel21, w1221, w1321, w2130, w2132);
   switch sw22 (sel22, w1422, w1522, w2231, w2233);
//   switch sw23 (sel23, w1623, w1723, w2331, w2333);
   switch sw24 (sel24, w1024, w1124, w2434, w2436);
   switch sw25 (sel25, w1225, w1325, w2534, w2536);
   switch sw26 (sel26, w1426, w1526, w2635, w2637);
//   switch sw27 (sel27, w1627, w1727, w2735, w2737);
   switch sw28 (sel28, w1828, w1928, w2838, w283a);
   switch sw29 (sel29, w1a29, w1b29, w2938, w293a);
   switch sw2a (sel2a, w1c2a, w1d2a, w2a39, w2a3b);
//   switch sw2b (sel2b, w1e2b, w1f2b, w2b39, w2b3b);
   switch sw2c (sel2c, w182c, w192c, w2c3c, w2c3e);
   switch sw2d (sel2d, w1a2d, w1b2d, w2d3c, w2d3e);
   switch sw2e (sel2e, w1c2e, w1d2e, w2e3d, w2e3f);
//   switch sw2f (sel2f, w1e2f, w1f2f, w2f3d, w2f3f);

   switch sw30 (sel30, w2030, w2130, w3040, w3041);
   switch sw31 (sel31, w2231, w2331, w3140, w3141);
   switch sw32 (sel32, w2032, w2132, w3242, w3243);
   switch sw33 (sel33, w2233, w2333, w3342, w3343);
   switch sw34 (sel34, w2434, w2534, w3444, w3445);
   switch sw35 (sel35, w2635, w2735, w3544, w3545);
   switch sw36 (sel36, w2436, w2536, w3646, w3647);
   switch sw37 (sel37, w2637, w2737, w3746, w3747);
   switch sw38 (sel38, w2838, w2938, w3848, w3849);
   switch sw39 (sel39, w2a39, w2b39, w3948, w3949);
   switch sw3a (sel3a, w283a, w293a, w3a4a, w3a4b);
   switch sw3b (sel3b, w2a3b, w2b3b, w3b4a, w3b4b);
   switch sw3c (sel3c, w2c3c, w2d3c, w3c4c, w3c4d);
   switch sw3d (sel3d, w2e3d, w2f3d, w3d4c, w3d4d);
   switch sw3e (sel3e, w2c3e, w2d3e, w3e4e, w3e4f);
   switch sw3f (sel3f, w2e3f, w2f3f, w3f4e, w3f4f);

   switch sw40 (sel40, w3040, w3140, w4050, w4051);
   switch sw41 (sel41, w3041, w3141, w4150, w4151);
   switch sw42 (sel42, w3242, w3342, w4252, w4253);
   switch sw43 (sel43, w3243, w3343, w4352, w4353);
   switch sw44 (sel44, w3444, w3544, w4454, w4455);
   switch sw45 (sel45, w3445, w3545, w4554, w4555);
   switch sw46 (sel46, w3646, w3746, w4656, w4657);
   switch sw47 (sel47, w3647, w3747, w4756, w4757);
   switch sw48 (sel48, w3848, w3948, w4858, w4859);
   switch sw49 (sel49, w3849, w3949, w4958, w4959);
   switch sw4a (sel4a, w3a4a, w3b4a, w4a5a, w4a5b);
   switch sw4b (sel4b, w3a4b, w3b4b, w4b5a, w4b5b);
   switch sw4c (sel4c, w3c4c, w3d4c, w4c5c, w4c5d);
   switch sw4d (sel4d, w3c4d, w3d4d, w4d5c, w4d5d);
   switch sw4e (sel4e, w3e4e, w3f4e, w4e5e, w4e5f);
   switch sw4f (sel4f, w3e4f, w3f4f, w4f5e, w4f5f);

   switch sw50 (sel50, w4050, w4150, w5060, w5061);
   switch sw51 (sel51, w4051, w4151, w5162, w5163);
   switch sw52 (sel52, w4252, w4352, w5260, w5261);
   switch sw53 (sel53, w4253, w4353, w5362, w5363);
   switch sw54 (sel54, w4454, w4554, w5464, w5465);
   switch sw55 (sel55, w4455, w4555, w5566, w5567);
   switch sw56 (sel56, w4656, w4756, w5664, w5665);
   switch sw57 (sel57, w4657, w4757, w5766, w5767);
   switch sw58 (sel58, w4858, w4958, w5868, w5869);
   switch sw59 (sel59, w4859, w4959, w596a, w596b);
   switch sw5a (sel5a, w4a5a, w4b5a, w5a68, w5a69);
   switch sw5b (sel5b, w4a5b, w4b5b, w5b6a, w5b6b);
   switch sw5c (sel5c, w4c5c, w4d5c, w5c6c, w5c6d);
   switch sw5d (sel5d, w4c5d, w4d5d, w5d6e, w5d6f);
   switch sw5e (sel5e, w4e5e, w4f5e, w5e6c, w5e6d);
   switch sw5f (sel5f, w4e5f, w4f5f, w5f6e, w5f6f);

   switch sw60 (sel60, w5060, w5260, w6070, w6071);
   switch sw61 (sel61, w5061, w5261, w6172, w6173);
   switch sw62 (sel62, w5162, w5362, w6274, w6275);
//   switch sw63 (sel63, w5163, w5363, w6376, w6377);
   switch sw64 (sel64, w5464, w5664, w6470, w6471);
   switch sw65 (sel65, w5465, w5665, w6572, w6573);
   switch sw66 (sel66, w5566, w5766, w6674, w6675);
//   switch sw67 (sel67, w5567, w5767, w6776, w6777);
   switch sw68 (sel68, w5868, w5a68, w6878, w6879);
   switch sw69 (sel69, w5869, w5a69, w697a, w697b);
   switch sw6a (sel6a, w596a, w5b6a, w6a7c, w6a7d);
//   switch sw6b (sel6b, w596b, w5b6b, w6b7e, w6b7f);
   switch sw6c (sel6c, w5c6c, w5e6c, w6c78, w6c79);
   switch sw6d (sel6d, w5c6d, w5e6d, w6d7a, w6d7b);
   switch sw6e (sel6e, w5d6e, w5f6e, w6e7c, w6e7d);
//   switch sw6f (sel6f, w5d6f, w5f6f, w6f7e, w6f7f);

   switch sw70 (sel70, w6070, w6470, w7080, w7081);
   switch sw71 (sel71, w6071, w6471, w7182, w7183);
   switch sw72 (sel72, w6172, w6572, w7284, w7285);
   switch sw73 (sel73, w6173, w6573, w7386, w7387);
   switch sw74 (sel74, w6274, w6674, w7488, w7489);
//   switch sw75 (sel75, w6275, w6675, w758a, w758b);
//   switch sw76 (sel76, w6376, w6776, w768c, w768d);
//   switch sw77 (sel77, w6377, w6777, w778e, w778f);
   switch sw78 (sel78, w6878, w6c78, w7880, w7881);
   switch sw79 (sel79, w6879, w6c79, w7982, w7983);
   switch sw7a (sel7a, w697a, w6d7a, w7a84, w7a85);
   switch sw7b (sel7b, w697b, w6d7b, w7b86, w7b87);
   switch sw7c (sel7c, w6a7c, w6e7c, w7c88, w7c89);
//   switch sw7d (sel7d, w6a7d, w6e7d, w7d8a, w7d8b);
//   switch sw7e (sel7e, w6b7e, w6f7e, w7e8c, w7e8d);
//   switch sw7f (sel7f, w6b7f, w6f7f, w7f8e, w7f8f);

   switch sw80 (sel80, w7080, w7880, b0, b1);
   switch sw81 (sel81, w7081, w7881, b2, b3);
   switch sw82 (sel82, w7182, w7982, b4, b5);
   switch sw83 (sel83, w7183, w7983, b6, b7);
   switch sw84 (sel84, w7284, w7a84, b8, b9);
   switch sw85 (sel85, w7285, w7a85, b10, b11);
   switch sw86 (sel86, w7386, w7b86, b12, b13);
   switch sw87 (sel87, w7387, w7b87, b14, b15);
   switch sw88 (sel88, w7488, w7c88, b16, b17);
   switch sw89 (sel89, w7489, w7c89, b18, b19);
//   switch sw8a (sel8a, w758a, w7d8a, b20, b21);
//   switch sw8b (sel8b, w758b, w7d8b, b22, b23);
//   switch sw8c (sel8c, w768c, w7e8c, b24, b25);
//   switch sw8d (sel8d, w768d, w7e8d, b26, b27);
//   switch sw8e (sel8e, w778e, w7f8e, b28, b29);
//   switch sw8f (sel8f, w778f, w7f8f, b30, b31);

endmodule /* perm20by20 */
```

A.5: Control word generator for 32-by-32 Benes network (C)

```
#include <stdio.h>
#define N 32            /* number of i/o */
#define x 9
#define y 16

typedef struct entry
{
  int used; /* keeps track of used entries
   on decomposition */
  int entry;          /* permutation map entries */
  int group;          /* P1 or P2 group assignment */
} map;                /* i/o permutation map */

map pmap[N];            /* permutation map array */
int cw[y][x];          /* control word array */

main()
{
  int n, m;
  int i, cnt;
  int llim, ulim;
  /*
  pmap[0].entry = 5;
  pmap[1].entry = 3;
  pmap[2].entry = 4;
  pmap[3].entry = 7;
  pmap[4].entry = 0;
  pmap[5].entry = 1;
  pmap[6].entry = 2;
  pmap[7].entry = 6;
  */

  printf("** Permutation entries to satisfy all require-
ments **\n");
  for (cnt=0; cnt<20; cnt++)   /* only top 20 of 32
   entries to be used */
    {
```

```
    printf("D[%d] = ", cnt);
    scanf("%d", &pmap[cnt].entry);
    pmap[cnt].used = 0;
    pmap[cnt].group = 0;
  }
for (cnt=20; cnt<N; cnt++)    /* rest self mapping */
  {
    pmap[cnt].entry = cnt;
    pmap[cnt].used = 0;
    pmap[cnt].group = 0;
  }

for (m=0; m<y; m++)
  {
    for (n=0; n<x; n++)
cw[m][n] = 5;   /* just for test since each entry to be
actively set */
  }

for (i=0; i<N; i++)
  {
    printf(" %d %d %d\n", pmap[i].used, pmap[i].entry,
      pmap[i].group);
  }

n = log(N);   /* finding log base 2 of N */

printf("n = %d\n", n);

for (m=0; m<n; m++)
  {
    llim = 0;
    i = N / pow(2, m);
    printf("i = %d\n", i);
    ulim = i;
    for (cnt=0; cnt < pow(2, m); cnt++)
 {
   decompose (llim, ulim, m);

   regroup (llim, ulim);
```

```
      llim = ulim;
      ulim = ulim + i;
    }
      }

  printf("--------\n");
  for (i=0; i<N; i++)
    {
      printf(" %d %d %d\n", pmap[i].used, pmap[i].entry,
        pmap[i].group);
    }
  printf("####### FINAL SWITCH SETUP ###########\n\n");
  for (m=0; m<y; m++)
    {
      for (n=0; n<x; n++)
   printf("%d ", cw[m][n]);
      printf("\n");
    }

  return 0;
}

decompose (int llim, int ulim, int n)
{                         /* decompose permutation array
   and set switches */
  int cnt;
  int u, v;
  int i;
  int p, q;

  for (cnt=llim; cnt<ulim; cnt++)
    {
      while (pmap[cnt].used == 0)
   {
     v = cnt / 2;
     u = x - n - 1;
     if (u != n)
       {
         printf(" v = %d, n = %d\n", v, n);
```

```
          cw[v][n] = (cnt+2) % 2;
        }
      printf("pmap[%d].entry = %d\n", cnt,
pmap[cnt].entry);
      v = (llim + pmap[cnt].entry) / 2;
      printf("v = %d, u = %d\n", v, u);
      cw[v][u] = (pmap[cnt].entry + 2) % 2;

      pmap[cnt].entry = pmap[cnt].entry / 2;
      pmap[cnt].used = 1;
      pmap[cnt].group = 0;

      i = llim;
      while ((pmap[i].used == 1) || ((llim +
pmap[i].entry)/2 != v))
        i++;

      pmap[i].used = 1;
      pmap[i].group = 1;

      pmap[i].entry = pmap[i].entry / 2;

      if ((i + 2) % 2 == 0)
        cnt = i + 1;
      else
        cnt = i - 1;

    }
      }
    /*
    for (p=0; p<y; p++)
      {
        for (q=0; q<x; q++)
    printf("%d ", cw[p][q]);
        printf("\n");
      }

    for (i=0; i<N; i++)
      {
        printf(" %d %d %d\n", pmap[i].used, pmap[i].entry,
          pmap[i].group);
        }
      printf("========\n");
      */
      return 0;
    }

    regroup (int llim, int ulim)  /* re-group entries in P1
and P2 */
    {
      map temp[(ulim-llim)/2];
      int c, d, e;
      int i;

      d = 0;
      e = llim;
      for (c=llim; c<ulim; c++)
        {
          if (pmap[c].group == 1)
            {
              temp[d].entry = pmap[c].entry;
              temp[d].group = pmap[c].group;
              temp[d].used = 1;              /* unnecessary */
              d++;
            }
          else
            {
              pmap[e].entry = pmap[c].entry;
              pmap[e].group = pmap[c].group;
              e++;
            }
          pmap[c].used = 0;
        }

      c = 0;
      while (e < ulim)
        {
          pmap[e].entry = temp[c].entry;
          pmap[e].group = temp[c].group;
          e++;
          c++;
    }
  /*
  for (i=0; i<N; i++)
    {
     printf("* %d %d %d\n", pmap[i].used, pmap[i].entry,
        pmap[i].group);
    }
    */
  return 0;
}

log (int num)    /* assuming num is power of 2 */
{
  int p = 0;

  while (num > 1)
    {
      num = num / 2;
      p++;
    }

  return p;
}

pow (int g, int h)   /* g raised to power h */
{
  int f = 1;

  while (h > 0)
    {
      f = f*g;
      h--;
    }
  return f;
}
```

A.6: Verification of functionality of permuter and control word generator (Verilog HDL)

```
  module test20by20perm;
      wire io0, io1, io2, io3, io4, io5, io6, io7, io8, io9,
io10, io11,
       io12, io13, io14, io15, io16, io17, io18, io19;
      wire oi0, oi1, oi2, oi3, oi4, oi5, oi6, oi7, oi8, oi9,
oi10, oi11,
       oi12, oi13, oi14, oi15, oi16, oi17, oi18, oi19;
      reg c0, c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11,
c12, c13,
       c14, c15, c16, c17, c18, c19, c20, c21, c22, c23, c24,
c25, c26,
       c27, c28, c29, c30, c31, c32, c33, c34, c35, c36, c37,
c38, c39,
       c40, c41, c42, c43, c44, c45, c46, c47, c48, c49, c50,
c51, c52,
       c53, c54, c55, c56, c57, c58, c59, c60, c61, c62, c63,
c64, c65,
       c66, c67, c68, c69, c70, c71, c72, c73, c74, c75, c76,
c77, c78,
       c79, c80, c81, c82, c83, c84, c85, c86, c87, c88, c89,
c90, c91,
       c92, c93, c94, c95, c96, c97, c98, c99, c100, c101,
c102, c103,
       c104, c105, c106, c107, c108, c109, c110, c111;

      initial
       begin
  /* control bit setup for a0..a19 mapped to b19..b0 as
derived from control word generator of Appendix A.5 */
          c0 = 0;
          c1 = 0;
          c2 = 0;
          c3 = 0;
          c4 = 0;
          c5 = 0;
          c6 = 0;
          c7 = 0;
```

```
c8 = 0;
c9 = 0;

c10 = 0;
c11 = 0;
c12 = 0;
c13 = 0;
c14 = 0;

c15 = 0;
c16 = 0;
c17 = 0;
c18 = 0;
c19 = 0;

c20 = 0;
c21 = 0;
c22 = 0;

c23 = 0;
c24 = 0;
c25 = 0;

c26 = 0;
c27 = 0;
c28 = 0;

c29 = 0;
c30 = 0;
c31 = 0;

c32 = 0;
c33 = 0;
c34 = 0;
c35 = 0;
c36 = 0;
c37 = 0;
c38 = 0;
c39 = 0;
c40 = 0;
```

```
c41 = 0;
c42 = 0;
c43 = 0;
c44 = 0;
c45 = 0;
c46 = 0;
c47 = 0;

c48 = 1;
c49 = 0;
c50 = 0;
c51 = 0;
c52 = 1;
c53 = 0;
c54 = 0;
c55 = 0;
c56 = 1;
c57 = 0;
c58 = 0;
c59 = 0;
c60 = 1;
c61 = 0;
c62 = 0;
c63 = 0;

c64 = 0;
c65 = 0;
c66 = 1;
c67 = 0;
c68 = 0;
c69 = 0;
c70 = 1;
c71 = 0;
c72 = 0;
c73 = 0;
c74 = 1;
c75 = 0;
c76 = 0;
c77 = 0;
c78 = 1;
```

```
c79 = 0;

c80 = 0;
c81 = 0;
c82 = 0;

c83 = 0;
c84 = 0;
c85 = 0;

c86 = 0;
c87 = 0;
c88 = 0;

c89 = 0;
c90 = 0;
c91 = 0;

c92 = 1;
c93 = 1;
c94 = 1;
c95 = 1;
c96 = 1;

c97 = 1;
c98 = 1;
c99 = 1;
c100 = 1;
c101 = 1;

c102 = 1;
c103 = 1;
c104 = 1;
c105 = 1;
c106 = 1;
c107 = 1;
c108 = 1;
c109 = 1;
c110 = 1;
c111 = 1;
```

```
        end

      initial
       begin
          $monitor("io =
%b%b%b%b%b%b%b%b%b%b%b%b%b%b%b%b%b%b%b%b\noi =
%b%b%b%b%b%b%b%b%b%b%b%b%b%b%b%b%b%b%b%b\n",  io0,
     io1, io2, io3, io4, io5, io6, io7, io8, io9, io10, io11,
io12,
     io13, io14, io15, io16, io17, io18, io19,
     oi0, oi1, oi2, oi3, oi4, oi5, oi6, oi7, oi8, oi9, oi10,
oi11, oi12,
     oi13, oi14, oi15, oi16, oi17, oi18, oi19);

          #5000 $stop;
       end

      assign io0 = 1;      // half setup for a's
      assign io1 = 1;
      assign io2 = 1;
      assign io3 = 1;
      assign io4 = 1;
      assign io5 = 1;
      assign io6 = 1;
      assign io7 = 1;
      assign io8 = 1;
      assign io9 = 1;

      assign oi0 = 0;      // other half setup for b's
      assign oi1 = 0;
      assign oi2 = 0;
      assign oi3 = 0;
      assign oi4 = 0;
      assign oi5 = 0;
      assign oi6 = 0;
      assign oi7 = 0;
      assign oi8 = 0;
      assign oi9 = 0;

      perm20by20 p0 (io0, io1, io2, io3, io4, io5, io6, io7,
io8, io9,
      io10, io11, io12, io13, io14, io15, io16, io17, io18,
io19,
      oi0, oi1, oi2, oi3, oi4, oi5, oi6, oi7, oi8, oi9,
oi10, oi11,
      oi12, oi13, oi14, oi15, oi16, oi17, oi18, oi19,
      c0, c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12,
c13, c14,
      c15, c16, c17, c18, c19, c20, c21, c22, c23, c24, c25,
c26, c27,
      c28, c29, c30, c31, c32, c33, c34, c35, c36, c37, c38,
c39, c40,
      c41, c42, c43, c44, c45, c46, c47, c48, c49, c50, c51,
c52, c53,
      c54, c55, c56, c57, c58, c59, c60, c61, c62, c63, c64,
c65, c66,
      c67, c68, c69, c70, c71, c72, c73, c74, c75, c76, c77,
c78, c79,
      c80, c81, c82, c83, c84, c85, c86, c87, c88, c89, c90,
c91, c92,
      c93, c94, c95, c96, c97, c98, c99, c100, c101, c102,
c103, c104,
      c105, c106, c107, c108, c109, c110, c111);

  endmodule /* test20by20perm */


   module switch (select, inout0, inout1, outin0, outin1);

     input select;
     inout inout0, inout1, outin0, outin1;

     tranif0 sw0 (inout0, outin0, select);
     tranif0 sw1 (inout1, outin1, select);
     tranif1 sw2 (inout0, outin1, select);
     tranif1 sw3 (inout1, outin0, select);

  endmodule /* switch */

  module perm20by20 (a0, a1, a2, a3, a4, a5, a6, a7, a8, a9,
     a10, a11, a12, a13, a14, a15, a16, a17, a18, a19,
   b0, b1, b2, b3, b4, b5, b6, b7, b8, b9,
   b10, b11, b12, b13, b14, b15, b16, b17, b18, b19,
   sel00, sel01, sel02, sel03, sel04, sel05, sel06, sel07,
sel08, sel09,
     // break (omission of switch for optimization)
   sel10, sel11, sel12, sel13, sel14, // break
   sel18, sel19, sel1a, sel1b, sel1c, // break
   sel20, sel21, sel22, // break
   sel24, sel25, sel26, // break
   sel28, sel29, sel2a, // break
   sel2c, sel2d, sel2e, // break
   sel30, sel31, sel32, sel33, sel34, sel35, sel36, sel37,
sel38, sel39,
   sel3a, sel3b, sel3c, sel3d, sel3e, sel3f,
   sel40, sel41, sel42, sel43, sel44, sel45, sel46, sel47,
sel48, sel49,
   sel4a, sel4b, sel4c, sel4d, sel4e, sel4f,
   sel50, sel51, sel52, sel53, sel54, sel55, sel56, sel57,
sel58, sel59,
   sel5a, sel5b, sel5c, sel5d, sel5e, sel5f,
   sel60, sel61, sel62, // break
   sel64, sel65, sel66, // break
   sel68, sel69, sel6a, // break
   sel6c, sel6d, sel6e, // break
   sel70, sel71, sel72, sel73, sel74, // break
   sel78, sel79, sel7a, sel7b, sel7c, // break
   sel80, sel81, sel82, sel83, sel84, sel85, sel86, sel87,
sel88, sel89
     // break
   );

     inout
   a0, a1, a2, a3, a4, a5, a6, a7, a8, a9,
   a10, a11, a12, a13, a14, a15, a16, a17, a18, a19,
   b0, b1, b2, b3, b4, b5, b6, b7, b8, b9,
   b10, b11, b12, b13, b14, b15, b16, b17, b18, b19;

     input
      sel00, sel01, sel02, sel03, sel04, sel05, sel06,
sel07, sel08, sel09,
       // break
      sel10, sel11, sel12, sel13, sel14, // break
      sel18, sel19, sel1a, sel1b, sel1c, // break
```

```
       sel20, sel21, sel22, // break
       sel24, sel25, sel26, // break
       sel28, sel29, sel2a, // break
       sel2c, sel2d, sel2e, // break
       sel30, sel31, sel32, sel33, sel34, sel35, sel36,
sel37, sel38, sel39,
       sel3a, sel3b, sel3c, sel3d, sel3e, sel3f,
       sel40, sel41, sel42, sel43, sel44, sel45, sel46,
sel47, sel48, sel49,
       sel4a, sel4b, sel4c, sel4d, sel4e, sel4f,
       sel50, sel51, sel52, sel53, sel54, sel55, sel56,
sel57, sel58, sel59,
       sel5a, sel5b, sel5c, sel5d, sel5e, sel5f,
       sel60, sel61, sel62, // break
       sel64, sel65, sel66, // break
       sel68, sel69, sel6a, // break
       sel6c, sel6d, sel6e, // break
       sel70, sel71, sel72, sel73, sel74, // break
       sel78, sel79, sel7a, sel7b, sel7c, // break
       sel80, sel81, sel82, sel83, sel84, sel85, sel86,
sel87, sel88, sel89;
       // break

      switch sw00 (sel00, a0, a1, w0010, w0018);
      switch sw01 (sel01, a2, a3, w0110, w0118);
      switch sw02 (sel02, a4, a5, w0211, w0219);
      switch sw03 (sel03, a6, a7, w0311, w0319);
      switch sw04 (sel04, a8, a9, w0412, w041a);
      switch sw05 (sel05, a10, a11, w0512, w051a);
      switch sw06 (sel06, a12, a13, w0613, w061b);
      switch sw07 (sel07, a14, a15, w0713, w071b);
      switch sw08 (sel08, a16, a17, w0814, w081c);
      switch sw09 (sel09, a18, a19, w0914, w091c);
   //   switch sw0a (sel0a, a20, a21, w0a15, w0a1d);   for
optimization
   //   switch sw0b (sel0b, a22, a23, w0b15, w0b1d);     on
control bits
   //   switch sw0c (sel0c, a24, a25, w0c16, w0c1e);
   //   switch sw0d (sel0d, a26, a27, w0d16, w0d1e);
   //   switch sw0e (sel0e, a28, a29, w0e17, w0e1f);
   //   switch sw0f (sel0f, a30, a31, w0f17, w0f1f);
   switch sw10 (sel10, w0010, w0110, w1020, w1024);
   switch sw11 (sel11, w0211, w0311, w1120, w1124);
   switch sw12 (sel12, w0412, w0512, w1221, w1225);
   switch sw13 (sel13, w0613, w0713, w1321, w1325);
   switch sw14 (sel14, w0814, w0914, w1422, w1426);
//   switch sw15 (sel15, w0a15, w0b15, w1522, w1526);
//   switch sw16 (sel16, w0c16, w0d16, w1623, w1627);
//   switch sw17 (sel17, w0e17, w0f17, w1723, w1727);

   switch sw18 (sel18, w0018, w0118, w1828, w182c);
   switch sw19 (sel19, w0219, w0319, w1928, w192c);
   switch sw1a (sel1a, w041a, w051a, w1a29, w1a2d);
   switch sw1b (sel1b, w061b, w071b, w1b29, w1b2d);
   switch sw1c (sel1c, w081c, w091c, w1c2a, w1c2e);
//   switch sw1d (sel1d, w0a1d, w0b1d, w1d2a, w1d2e);
//   switch sw1e (sel1e, w0c1e, w0d1e, w1e2b, w1e2f);
//   switch sw1f (sel1f, w0e1f, w0f1f, w1f2b, w1f2f);

   switch sw20 (sel20, w1020, w1120, w2030, w2032);
   switch sw21 (sel21, w1221, w1321, w2130, w2132);
   switch sw22 (sel22, w1422, w1522, w2231, w2233);
//   switch sw23 (sel23, w1623, w1723, w2331, w2333);
   switch sw24 (sel24, w1024, w1124, w2434, w2436);
   switch sw25 (sel25, w1225, w1325, w2534, w2536);
   switch sw26 (sel26, w1426, w1526, w2635, w2637);
//   switch sw27 (sel27, w1627, w1727, w2735, w2737);
   switch sw28 (sel28, w1828, w1928, w2838, w283a);
   switch sw29 (sel29, w1a29, w1b29, w2938, w293a);
   switch sw2a (sel2a, w1c2a, w1d2a, w2a39, w2a3b);
//   switch sw2b (sel2b, w1e2b, w1f2b, w2b39, w2b3b);
   switch sw2c (sel2c, w182c, w192c, w2c3c, w2c3e);
   switch sw2d (sel2d, w1a2d, w1b2d, w2d3c, w2d3e);
   switch sw2e (sel2e, w1c2e, w1d2e, w2e3d, w2e3f);
//   switch sw2f (sel2f, w1e2f, w1f2f, w2f3d, w2f3f);

   switch sw30 (sel30, w2030, w2130, w3040, w3041);
   switch sw31 (sel31, w2231, w2331, w3140, w3141);
   switch sw32 (sel32, w2032, w2132, w3242, w3243);
   switch sw33 (sel33, w2233, w2333, w3342, w3343);
switch sw34 (sel34, w2434, w2534, w3444, w3445);
switch sw35 (sel35, w2635, w2735, w3544, w3545);
switch sw36 (sel36, w2436, w2536, w3646, w3647);
switch sw37 (sel37, w2637, w2737, w3746, w3747);
switch sw38 (sel38, w2838, w2938, w3848, w3849);
switch sw39 (sel39, w2a39, w2b39, w3948, w3949);
switch sw3a (sel3a, w283a, w293a, w3a4a, w3a4b);
switch sw3b (sel3b, w2a3b, w2b3b, w3b4a, w3b4b);
switch sw3c (sel3c, w2c3c, w2d3c, w3c4c, w3c4d);
switch sw3d (sel3d, w2e3d, w2f3d, w3d4c, w3d4d);
switch sw3e (sel3e, w2c3e, w2d3e, w3e4e, w3e4f);
switch sw3f (sel3f, w2e3f, w2f3f, w3f4e, w3f4f);

switch sw40 (sel40, w3040, w3140, w4050, w4051);
switch sw41 (sel41, w3041, w3141, w4150, w4151);
switch sw42 (sel42, w3242, w3342, w4252, w4253);
switch sw43 (sel43, w3243, w3343, w4352, w4353);
switch sw44 (sel44, w3444, w3544, w4454, w4455);
switch sw45 (sel45, w3445, w3545, w4554, w4555);
switch sw46 (sel46, w3646, w3746, w4656, w4657);
switch sw47 (sel47, w3647, w3747, w4756, w4757);
switch sw48 (sel48, w3848, w3948, w4858, w4859);
switch sw49 (sel49, w3849, w3949, w4958, w4959);
switch sw4a (sel4a, w3a4a, w3b4a, w4a5a, w4a5b);
switch sw4b (sel4b, w3a4b, w3b4b, w4b5a, w4b5b);
switch sw4c (sel4c, w3c4c, w3d4c, w4c5c, w4c5d);
switch sw4d (sel4d, w3c4d, w3d4d, w4d5c, w4d5d);
switch sw4e (sel4e, w3e4e, w3f4e, w4e5e, w4e5f);
switch sw4f (sel4f, w3e4f, w3f4f, w4f5e, w4f5f);

switch sw50 (sel50, w4050, w4150, w5060, w5061);
switch sw51 (sel51, w4051, w4151, w5162, w5163);
switch sw52 (sel52, w4252, w4352, w5260, w5261);
switch sw53 (sel53, w4253, w4353, w5362, w5363);
switch sw54 (sel54, w4454, w4554, w5464, w5465);
switch sw55 (sel55, w4455, w4555, w5566, w5567);
switch sw56 (sel56, w4656, w4756, w5664, w5665);
switch sw57 (sel57, w4657, w4757, w5766, w5767);
switch sw58 (sel58, w4858, w4958, w5868, w5869);
switch sw59 (sel59, w4859, w4959, w596a, w596b);
```

```
   switch sw5a (sel5a, w4a5a, w4b5a, w5a68, w5a69);
   switch sw5b (sel5b, w4a5b, w4b5b, w5b6a, w5b6b);
   switch sw5c (sel5c, w4c5c, w4d5c, w5c6c, w5c6d);
   switch sw5d (sel5d, w4c5d, w4d5d, w5d6e, w5d6f);
   switch sw5e (sel5e, w4e5e, w4f5e, w5e6c, w5e6d);
   switch sw5f (sel5f, w4e5f, w4f5f, w5f6e, w5f6f);

   switch sw60 (sel60, w5060, w5260, w6070, w6071);
   switch sw61 (sel61, w5061, w5261, w6172, w6173);
   switch sw62 (sel62, w5162, w5362, w6274, w6275);
//   switch sw63 (sel63, w5163, w5363, w6376, w6377);
   switch sw64 (sel64, w5464, w5664, w6470, w6471);
   switch sw65 (sel65, w5465, w5665, w6572, w6573);
   switch sw66 (sel66, w5566, w5766, w6674, w6675);
//   switch sw67 (sel67, w5567, w5767, w6776, w6777);
   switch sw68 (sel68, w5868, w5a68, w6878, w6879);
   switch sw69 (sel69, w5869, w5a69, w697a, w697b);
   switch sw6a (sel6a, w596a, w5b6a, w6a7c, w6a7d);
//   switch sw6b (sel6b, w596b, w5b6b, w6b7e, w6b7f);
   switch sw6c (sel6c, w5c6c, w5e6c, w6c78, w6c79);
   switch sw6d (sel6d, w5c6d, w5e6d, w6d7a, w6d7b);
   switch sw6e (sel6e, w5d6e, w5f6e, w6e7c, w6e7d);
//   switch sw6f (sel6f, w5d6f, w5f6f, w6f7e, w6f7f);

   switch sw70 (sel70, w6070, w6470, w7080, w7081);
   switch sw71 (sel71, w6071, w6471, w7182, w7183);
   switch sw72 (sel72, w6172, w6572, w7284, w7285);
   switch sw73 (sel73, w6173, w6573, w7386, w7387);
   switch sw74 (sel74, w6274, w6674, w7488, w7489);
//   switch sw75 (sel75, w6275, w6675, w758a, w758b);
//   switch sw76 (sel76, w6376, w6776, w768c, w768d);
//   switch sw77 (sel77, w6377, w6777, w778e, w778f);
   switch sw78 (sel78, w6878, w6c78, w7880, w7881);
   switch sw79 (sel79, w6879, w6c79, w7982, w7983);
   switch sw7a (sel7a, w697a, w6d7a, w7a84, w7a85);
   switch sw7b (sel7b, w697b, w6d7b, w7b86, w7b87);
   switch sw7c (sel7c, w6a7c, w6e7c, w7c88, w7c89);
//   switch sw7d (sel7d, w6a7d, w6e7d, w7d8a, w7d8b);
//   switch sw7e (sel7e, w6b7e, w6f7e, w7e8c, w7e8d);
//   switch sw7f (sel7f, w6b7f, w6f7f, w7f8e, w7f8f);
```

```
   switch sw80 (sel80, w7080, w7880, b0, b1);
   switch sw81 (sel81, w7081, w7881, b2, b3);
   switch sw82 (sel82, w7182, w7982, b4, b5);
   switch sw83 (sel83, w7183, w7983, b6, b7);
   switch sw84 (sel84, w7284, w7a84, b8, b9);
   switch sw85 (sel85, w7285, w7a85, b10, b11);
   switch sw86 (sel86, w7386, w7b86, b12, b13);
   switch sw87 (sel87, w7387, w7b87, b14, b15);
   switch sw88 (sel88, w7488, w7c88, b16, b17);
   switch sw89 (sel89, w7489, w7c89, b18, b19);
//   switch sw8a (sel8a, w758a, w7d8a, b20, b21);
//   switch sw8b (sel8b, w758b, w7d8b, b22, b23);
//   switch sw8c (sel8c, w768c, w7e8c, b24, b25);
//   switch sw8d (sel8d, w768d, w7e8d, b26, b27);
//   switch sw8e (sel8e, w778e, w7f8e, b28, b29);
//   switch sw8f (sel8f, w778f, w7f8f, b30, b31);

endmodule /* perm20by20 */
```

References

[1] MARGOLUS, NORMAN and THOMAS F. KNIGHT, JR. ; A virtual-lattice SIMD+DRAM architecture optimized for symbolic spatial computations

[2] MARGOLUS, NORMAN; An FPGA architecture for DRAM-based systolic computations

[3] MARGOLUS, NORMAN; CAM-8: a computer architecture based on cellular automata; December 15, 1993

[4] MARGOLUS, NORMAN; Multidimensional cellular data array processing system which separately permutes stored data elements and applies transformation rules to permuted elements; United States Patent, Number 5159690, October 27, 1992

[5] TOFFOLI, TOMMASO, NORMAN MARGOLUS; Cellular Automata Machines: A new environment for modeling; MIT Press 1987

[6] LEIGHTON, F. THOMSON; Introduction to parallel algorithms and architectures: arrays, trees, hypercubes; Morgan Kaufmann Publishers 1992

[7] Direct Rambus Technology Disclosure; Rambus Inc., October 1997

[8] OPFERMAN, D.C., N.T.TSAO-WU; On a class of rearrangeable switching networks; The Bell System Technical Journal, Vol. 50, No. 5, May-June 1971

[9] NASSIMI, DAVID, SARTAJ SAHNI; Parallel algorithms to set up the Benes permutation network; IEEE Transactions on Computers, Vol. C-31, No. 2, February 1982

[10] LEE, KYUNGSOOK YOON; A new Benes network control algorithm; IEEE Transactions on Computers, Vol. C-36, No. 6, June 1987

[11] LEE, CHING-YI, A. YAVUZ ORUC; Fast parallel algorithms for routing one-to-one assignments in Benes networks; Computer science technical report series, University of Maryland, College Park, MD 20742, August 1992

[12] GEORGE, DAVID, MICHAEL TSAO, BANU OZDEN; Testing Benes interconnection networks; Research Report, IBM Research Division, May 2, 1990