A Hardware Product Sorter-Absorber Circuit Designed for DEC-Perle Architecture

Pavel Kashubin, Pongthai Sakornsint, Narit Ruangjiratain
pavel@ee.pdx.edu, aroy7004@ee.pdx.edu, rnarit@ee.pdx.edu
EE574FP Fast Prototyping using FPGAs
Portland State University, Portland OR

Introduction

Often when solving problems with logic minimization, a cover may be generated with all possible Sum-Of-Products terms and then minimized with another program. In this paper we propose a hardware circuit designed to aid in function minimization by sorting products and removing redundant, "absorbed" values. Our previous work in this area produced a design that was not well suitable for implementation on DEC-Perle board but was a proof of a concept on such a circuit. This design was specifically targeted to DEC-Perle Architecture and Xilinx FPGAs. The paper consists of 5 chapters. Chapter 1 describes project's algorithm and absorption theory. Chapter 2 expands the project description. Chapter 3 develops operation of pipeline operation of sorters. Chapter 4 describes design of every component of the project. Chapter 5 has additional notes on VHDL platform, code design and testing.

Chapter 1 – Algorithm description and absorption

The circuit designed in this project has two different goals on processing data. First goal is to sort products by their size, placing products with fewer variables on higher then ones containing more variables. Products with fewer variables represent bigger cubes and because of this, sorting helps to implement circuits by using best solutions first. The sorting algorithm is relatively simple, often called bubble sort where 2 terms are compared and bigger cube is placed higher.
The second and more important goal of this circuit is to remove redundant cubes, the ones that can be covered with a bigger cube (fewer variables). Removal is done according to the law of absorption, so let us review it.
Absorption is the process of simplifying the Sum of Products or Product of Sums according to absorption law,

A+AB=A or A(A+B)=A

The form A+AB=A is known as the SOP form of this law. Also, note that the absorption law is valid for a negated terms as well, i.e. !A + !AB = !A.

Now, lets look at a little more complicated case. Suppose we are given a K-map with the following products:

CD

AB \ CD	00	01	11	10
00	0	1	1	0
01	1	1	1	1
11	0	0	0	0
10	0	0	1	0

Given is a solution that includes all possible cubes for this cover:
!A!B!CD + !A!BCD + !AB!C!D + !AB!CD + !ABCD + !ABC!D + A!BCD +
!A!BD + !AB!C + !A!CD + !ACD + !ABC + !BCD + !AB + !AD.
This SOP needs to have all redundant cubes removed and remaining cubes sorted. Also, we do not place constrain on our input that there would be no same terms on the input.
Looking at this particular example, we see that every small cube, such as !A!B!CD can be absorbed by a bigger cube, such as !AB!C or even !AB.
The output that we are looking to get for this particular example from our circuit is: !AB + !AD +!BCD. (*Note the bigger cubes are on top, because output is sorted*)

This is a very small example, but our circuit is designed to solve very large problems, up to 16 variables and 128k terms.

We modify our input from the example shown above in a way that would enable us to represent a name for a variable by its position in the input. All variables have a place associated with them in the input word of the circuit. For example A will be always in the first place, B in the second and so on. If a variable is missing, we need to have a value in the variable associated with this fact, such as 'X'. With this, AD becomes AXXD. (4 variables) An absorbed term may be also denoted by an "?" sign. We have agreed to use following representation of the variable in the term.
1 – indicates that the variable is present and asserted high (A)
0 – indicates that the variable is present and asserted low (!A)
X – indicates that the variable is not present – don't care
? -- indicates that the value has been absorbed

Since we have four possible states a variable may be in, we have chosen following 2-bit encoding for each variable.
"00" -- '?' -- Absorbed term
"11" -- 'x' -- Absent term
"10" – '0' – Asserted low
"01" – '1' – Asserted high

For example, if the input word is four variable wide, product term !ABC!D would become "10 01 11 10" and AC would be encoded as "01 11 01 11".

Let's further describe the law-of-absorption:

1) If both of the product terms contain missing variables in the same position or if they contain same values in those positions, then they are the same and therefore both can be absorbed into each other.

2) Given the rest of the variables are equal, if one of the variables is X, and same variable in the other term is either 1 or 0, then the product containing X absorbs other product. This absorption decision is "recorded" by either state of the state machine or carry of the iterative circuit. Also, there could not be previous decision made to perform absorption in the opposite way.

3) If one of the variables being compared are opposite and asserted (either 1 or 0 and another variable is opposite and 0 or 1), the products are different and absorption cannot take place between the given two terms.

Let's re-enforce these concepts with examples illustrating the four situations we will encounter.

<u>A>B or B>A</u>

This operation is a bit comparison of the two products. Both products contain same number of terms. If any product term is a ‘1’ for one of the products, and a ‘0’ for the other, then there can be no absorption, the product with the 1 term is the one that is greater. These two conditions are mutually exclusive.

Example:

A: 0 1 0 1
B: 0 1 1 0

Bit B(1) is larger than A(1), therefore B is Larger than A

<u>A=B</u>

This operation is a bit by bit comparison of the two product terms. If the two are equivalent, the A will be absorbed into B (this is an arbitrary decision).

Example:

A: 0 1 0 1
B: 0 1 0 1

<u>A absorbed into B</u>

This operation is a bit by bit comparison of A and B. If Ai is not equal to Bi, and Ai is not X, then the entire product A cannot be absorbed to B.

Example:

A: 0 1 0 1
B: 0 1 x 0

no absorption

Example:

A: 0 1 0 1
B: 0 1 x 1

All terms are equivalent or B term is x.

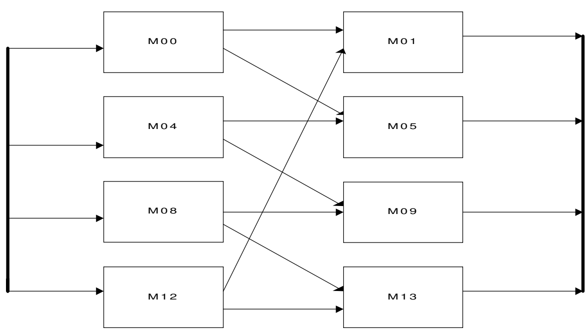
<u>B absorbed into A</u>

This operation is a bit by bit comparison of A and B. If Bi is not equal to Ai, and Bi is not X, then the entire product B cannot be absorbed to A.

Chapter 2 Design specifications

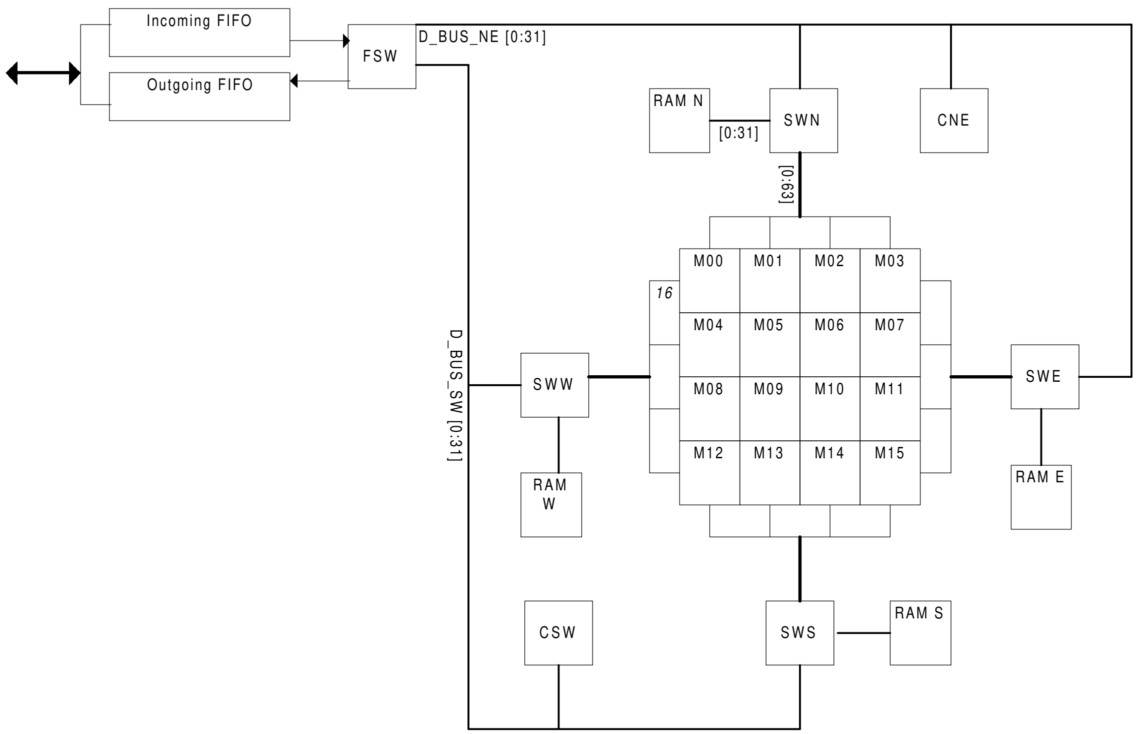
Figure 2.0 illustrates flow of data in sorter-absorber matrix. The sorting is done by loading 4 words to be processed in parallel into four FPGA chips containing sorter-absorber circuits. First column imlements what we call even sort where two adjacent numbers are sorted. Second column implements odd sort where one number from upper left and one from the left are compared. Sorted terms are written back to memory and next four are loaded in a pipeline manner. During sorting process, status is kept by controller. If sorting or absorption has occurred while data space is being sorted, sorting is repeated until no more sorting-absorption takes place. This sorting algorithm guarantees sorting, but not absorption, since data may be pre-sorted and two products that could possibly be absorbed did not cross each other in a sorter circuit. To guarantee that all values that can be absorbed have been absorbed, sorting is rerun completely, this time loading memory data from bottom up.
Once these cycles have been completed, data can be written back to host. Note that absorbed data is replaced by invalid code, and the controller writing back to host may discard this data, or it can be done by software running on host computer.

Fig 2.0 Flow of Data in sorter-absorber matrix.



Design is implemented on a DEC-Perle board with architecture shown in Fig 2.1. Board consists of a matrix 4 by 4 Xilinx 3090 FPGAs and four more 3090 FPGAs are used as switched for buses. Each chip in the array has connections to 4 buses and 4 neighbor chips. Buses are called MbusE, MnusN, MbusW, MbusS. And are 64 bits each, with 16 bits of each bus available to each chip in the array. Each chip has direct connections connecting it with its top, bottom, left and right neighbors. Direct connections are called DCHxx-xxV where xx contains a name of chip connected. There are also 10 lines for ring matrix connecting array processors and switches. There are four memories on board, 256k bytes each, along with two controller chips, each controlling two memories: CSW controlling S and W memories, and CNE controlling N and E memory. Also, one more FPGA called FSW connects two memory buses and host interface.

Fig 2.1 Perle-1 Architecture



Chapter 3 Design of Sorter-Absorber Pipeline

This chapter will explain and illustrate the developed pipelined sorting. Pipelining was developed to take maximum advantage of the DEC-Perle board due to limited memory and bus resources.
As can be seen from the figure 2.1 there are 16 bits available to each of the M00, M04, M08, and M12 FPGAs. Therefore the 32-bit word (16 variables) cannot be loaded from the same memory in one clock cycle. This bottleneck is the causing design to spend four clock cycles to load data into a sorter and is even more inefficient without pipelining.

Data Processing. Operations are denoted similarly as in many computer architecture texts. Loads are denoted by L, Stores by S and writes to other FPGAs as W. Memory writes are denoted by WB. A and B stand for two memory words, A is from one memory bank and B is from another memory bank. We denote *l* and *h* to mean low half-word and high half-word respectively. Also we will note what bus resources are used by the pipeline in the table 3.2 below.

For the pipeline design discussion that follows, please look at Table 3.1.
On left column of this table, there is a processor name and on top row there are clock cycle numbers and types. Also, operation of memory controllers are shown on the bottom of the table. Lets look at cycle 1 of the table 3.1. We can see below the name that this is init_1 cycle or initial pipeline fill cycle. There are 8 cycles 0 through 7 that are called initial pipeline cycles. What this means is that initially, when pipeline is being filled, second column of sorters do not yet have data available to sort and they are idle. In init_1 cycle, the data destined for processors M00 and M04 had been read by switch on previous cycle init_0 and now ready to be sent to processors. Since only 16 bit bus are available, low half word is sent first
M00: L l A_0 *Processor M00 Loads low half of word A*
M04 L l B_1 . *Processor M04 Loads low half of word B*

Note: the subscripts on A and B indicate address of data loaded into processor. As far as the processor is concerned, it only knows that it will get some half of first or second word in a given cycle. Subscripts are kept for design of memory controller address generator.

Processors M08 and M12 do not have data available in the switch on this clock cycle init_1 (data is being read in this cycle by CSW controller). Cycles init_2 through init_4 are very similar to init_1 and during these data is written to a processors in a way that it avoids stalls due to memory read cycles. This implies that the reads are from two banks and not sequential. If reads were sequential, as for example A0, B0; A1, B1; A2, B2; A3, B4; we would have a problem that only one processor can start loading data in a cycle. During init_1, processor M04 will wait until data becomes available in cycle init_2, M08 will start loading data in init_3, M12 in init_4 and so forth. The method used in our design is more complicated but yields better performance at the expense of a more complicated memory address generator, which is like a counter with an adder that adds or subtracts a small integer (less then 4). Now, lets look at what's happening in the pipeline on cycle init_5: processors M00 and M04 now have the data they need to perform sorting/absorption on this data. We assumed that the whole operation can be done in one clock cycle because we use iterative circuit to process data. However this will need to be verified after place and route of the design. Also processors M08 and M12 are still loading data because they had to wait for data to be loaded from memory one cycle later then M00 and M04.
During an init_6 cycle processors M00 and M04 have processed data ready to be sent to second column of sorter/absorber circuits. Data is sent over direct connects (16 bits) and using a bus for another 16 bits. Therefore each word is sent over one clock and two ticks are needed to transfer results. Look at figure 4.0 in the beginning of this paper – note that second column of sorters gets its data one across, one diagonally. A word that will be send diagonally is sent first, latched by neighbor on right and then sent to lower neighbor. This is how the skewed pipeline needed for bubble even/odd sort is implemented. In the second column, only 3 chips are doing sorting/absorption; the other one, we picked M01 is programmed to hold data for one cycle until it can start writing it back to memory.
Over the pipeline cycles sust_1 through sust_4, the second column of sorter/absorber circuits write back their data to memory. Write back to memory is also constrained by narrow buses – the sorter/absorber circuits in the second column take 4 clock ticks each to write 2 result words to memory.

Data written back to memory has not been completely sorted: 8 words require at worst case four passes through this 2 column even/odd sort. This project would work a lot better if 4 columns could be implemented into the DEC-Perle architecture. Since the buses are used to transfer the intermediate data, they become the constraining resource not allowing 4 columns.
When problem size becomes larger, the number of repetitions through the data becomes greater exponentially. To know when data is sorted it doesn't make sense to repeat the process over and over to reach theoretical worst case. Instead, each sorter can signal controller whether sorting or absorption occurred, and if so, controller will repeat sorting through whole data once again.

Proc essor	Cycle **0** init_0	1 init_1	2 init_2	3 init_3	4 init_4	5 init_5	6 init_6	7 init_7	**8** sust_0	9 sust_1	10 sust_2	11 sust_3	12 sust_4	13 sust_5	14 sust_6
M00	Read n-1	L *l* A_0	L *h* A_0	L *l* B_0	L *h* B_0	Sort(A_0, B_0)	W B_0	W A_0	L *l* A_4	L *h* A_4	L *l* B_4	L *h* B_4	Sort (A_4, B_4)	W B_4	W A_4
M04		L *l* B_1	L *h* B_1	L *l* A_1	L *h* A_1	Sort(A_1, B_1)	W B_1	W A_1	L *l* B_5	L *h* B_5	L *l* A_5	L *h* A_5	Sort(A_5, B_5)	W B_5	W A_5
M08		No-op	L *l* A_2	L *h* A_2	L *l* B_2	L *h* B_2	Sort (A_2, B_2)	W B_2	W A_2	L *l* A_6	L *h* A_6	L *l* B_6	L *h* B_6	Sort (A_6, B_6)	W B_6
M12		No-op	L *l* B_3	L *h* B_3	L *l* A_3	L *h* A_3	Sort($A_{,}$, B_3)	W B_3	W $A_{\square}$	L *l* B_7	L *h* B_7	L *l* A_7	L *h* A_7	Sort($A_{,}$, B_7)	W B_7
M01		No-op	No-op	No-op	No-op	No-op	L B_0	W B_0, L A_0	Hold A_0 L B_3	WB *l* A_0	WB *h* A_0	WB *l* B_3	WB *h* B_3	L B_4	L A_4 W B_4
M05		No-op	No-op	No-op	No-op	No-op	L B_1	W B_1 L $A_{1,}$ B_0	Sort (A_1, B_0)	WB *l* B_0	WB *h* B_0	WB *l* A_1	WB *h* A_1	L B_5	L $A_{5,}$ B_4 W B_5
M09		No-op	No-op	No-op	No-op	No-op	No-op	L B_2, L B_1	W B_2, L A_2	Sort (A_2, B_1)	WB *l* A_2	WB *h* A_2	WB *l* B_1	WB *h* B_1	L B_5, B_6
M13		No-op	No-op	No-op	No-op	No-op	No-op	L B_3	W B_3 , L $A_{3,}$ L B_2	Sort (A_3, B_2)	WB *l* B_2	WB *h* B_2	WB *l* A_3	WB *h* A_3	L B_7
CS W	R A_0 B_1	R A_2, B_3	RA$_1$, B_0	R A_3, B_2				R A_4, B_5	R A_6, B_7	R A_5, B_4	R A_7, B_6				R A_8 B_9
CNE												W A_0, B_0	W A_2, B_2	WB$_3$, A_1	W B_1, A_3

Process or/ Cycle	**15** sust_0	16 sust_1	17 sust_2	18 sust_3	19 sust_4	20 sust_5	21 sust_6	**22** sust_0	23 sust_1	24 sust_2	25 sust_3	26 sust_4	27 sust_5	28 sust_6	29
M00	L *l* A_8	L *h* A_8	L *l* B_8	L *h* B_8	Sort(A_8, B_8)	W B_8	W A_8	L *l* A_{12}	L *h* A_{12}	L *l* B_{12}	L *h* B_{12}	Sort (A_{12}, B_{12})	W B_{12}	W A_{12}	
M04	L *l* B_9	L *h* B_9	L *l* A_9	L *h* A_9	Sort(A_9, B_9)	W B_9	W A_9	L *l* B_{13}	L *h* B_{13}	L *l* A_{13}	L *h* A_{13}	Sort(A_{13}, B_{13})	W B_{13}	W A_{13}	
M08	W A_6	L *l* A_{10}	L *h* A_{10}	L *l* B_{10}	L *h* B_{10}	Sort (A_{10}, B_{10})	W B_{10}	W A_{10}	L *l* A_{14}	L *h* A_{14}	L *l* B_{14}	L *h* B_{14}	Sort (A_{14}, B_{14})	W B_{14}	
M12	W A_7	L *l* B_{11}	L *h* B_{11}	L *l* A_{11}	L *h* A_{11}	Sort(A_{11}, B_{11})	W B_{11}	W A_{11}	L *l* B_{15}	L *h* B_{15}	L *l* A_{15}	L *h* A_{15}	Sort(A_{15}, B_{15})	W B_{15}	
M01	hold A_4 L B_7	WB *l* A_4	WB *h* A_4	WB *l* B_7	WB *h* B_7	L B_8	W B_8, L A_8	Hold A_8 L B_{11}	WB *l* A_8	WB *h* A_8	WB *l* B_{11}	WB *h* B_{11}	L B_{12}	L A_{12} W B_{12}	
M05	Sort (A_5,B_4)	WB *l* B_4	WB *h* B_4	WB *l* A_5	WB *h* A_5	L B_9	W B_9 L $A_{9,}$ B_8	Sort (A_9, B_8)	WB *l* B_8	WB *h* B_8	WB *l* A_9	WB *h* A_9	L B_{13}	L A_{13} W B_{13}	
M09	L A_6 W B_6	Sort (A_6,B_5)	WB *l* A_6	WB *h* A_6	WB *l* B_5	WB *h* B_5	L B_{10}, L B_9	W B_{10}, L A_{10}	Sort (A_{10}, B_9)	WB *l* A_{10}	WB *h* A_{10}	WB *l* B_9	WB *h* B_9	L B_{12}	
M13	W B_7, L B_6,A_7	Sort (A_7,B_6)	WB *l* A_7	WB *h* A_7	WB *l* B_6	WB *h* B_6	L B_{11}	W B_{11} , L B_{10} L A_{11}	Sort (A_{11}, B_{10})	WB *l* B_{10}	WB *h* B_{10}	WB *l* A_{11}	WB *h* A_{11}	L B_{13}	
CSW	R A_{10}, B_{11}	RA$_9$, B_8	R A_{11}, B_{10}				R A_{12}, B_{13}	R A_{14}, B_{15}	R A_{13}, B_{12}	R A_{15}, B_{14}					
CNE				W A_4, B_4	W A_6, B_6	WB$_7$, A_5	W B_5, A_7				W A_8, B_8	W A_{10}, B_{10}	WB$_{11}$, A_9	W B_9, A_{11}	

Table 3.1 Pipelined Data Processing

Processor/ Cycle	28 sust_6	**29** final_0	30 final_1	31 final_2	32 final_3	33 final_4	34 final_5	**35**	36
M00	W A_{12}	No-op	No-op	No-op	No-op	No-op	No-op	No-op	
M04	W A_{13}	No-op	No-op	No-op	No-op	No-op	No-op	No-op	
M08	W B_{14}	W A_{14}	No-op	No-op	No-op	No-op	No-op	No-op	
M12	W B_{15}	W A_{15}	No-op	No-op	No-op	No-op	No-op	No-op	
M01	L A_{12} W B_{12}	Hold A_{12} L B_{15}	WB l A_{12}	WB h A_{12}	WB l B_{15}	WB h B_{15}	No-op	No-op	
M05	L A_{13}, B_{12} W B_{13}	Sort (A_{13}, B_{12})	WB l B_{12}	WB h B_{12}	WB l A_{13}	WB h A_{13}	No-op	No-op	
M09	L B_{14}, B_{13}	W B_{14}, L A_{14}	Sort (A_{14}, B_{13})	WB l A_{14}	WB h A_{14}	WB l B_{13}	WB h B_{13}	No-op	
M13	L B_{13}	W B_{15} , L A_{15}	Sort (A_{15}, B_{14})	WB l B_{10}	WB h B_{10}	WB l A_{11}	WB h A_{15}	No-op	
CSW									
CNE					W A_{12}, B_{12}	W A_{14}, B_{14}	WB$_{15}$, A_{13}	W B_{13}, A_{15}	

Table 3.2 Draining of the Pipeline Cycles.

Notice that the data transfer becomes periodic after 7 cycles. Cycles 0 through 7 are initial sequence and 8 through 14, 15 through 21, 22 through 29 very similar. Our sorter network will receive a cycle ID and will know what to do during a cycle.

Our controller will have at least 22 states representing 21 possible cycles at any given time.
0-7 Initial (init_0, init_1, init_2, init_3, init_4, init_5, init_6)
7 Sustained Pipeline Cycles
7 Ending Cycles
See Table 3.2 for Ending Cycles diagram. Ending cycles are different by the fact that no new data will be loaded into sorters M00 through M12.

For 21 different cycles we need 5 bits to signal state to the sorters. These bits come from the ring matrix, and we have others that will be used to signal back to controller, as well as reset.

Lets note here that first column and second column sorters will perform different kinds of data transfer and will need to be configured differently. There will be two sorter configurations and one FPGA in the second column that will be used as 2 32-bit registers.

Table 3.3 Resources Allocation

Cycle	Instruction or Data Transfer	Bus Resources
0 Initial Pipeline Cycles	No-ops, preload data from memory	
1	L l A_0 L l B_1 No-op	MbusW 00:15 MbusW 16:31
2	L h A_0 L h B_1 L l A_2 L l B_3	MbusW 00:15 MbusW 16:31 MbusW 32:47 MbusW 48:63
3	L l B_0	MbusW 00:15

	L *l* A_1 L *h* A_2 L *h* B_3	MbusW 16:31 MbusW 32:47 MbusW 48:63
4	L *h* B_0 L *h* A_1 L *l* B_2 L *l* A_3	MbusW 00:15 MbusW 16:31 MbusW 32:47 MbusW 48:63
5	L *h* B_2 L *h* A_3 Sort (A_0, B_0) Sort(A_1, B_1)	MbusW 32:47 MbusW 48:63
6	B_0 (M00:M01) B_1 (M04:M05) Sort (A_2, B_2) Sort (A_3, B_3)	DC00-01H, MbusE 00:15 DC01-05H, MbusE 16:31
7	B_0 (M01 : M05) A_1 (M04 : M05) B_1 (M05 : M09) B_2 (M08 : M09) B_3 (M12 : M13) A_0 (M00: M01)	DC01-05V, MbusN 16:31 DC01-05H, MbusE 16:31 DC05-09V, MbusS 16:31 DC08-09H, MbusE 32:47 DC12-13H, MbusE 48:63 DC00-02H, MbusE 00:15
8**– Sustained Pipeline Cycles**	L *l* A_4 L *l* B_5 B_3 (M13 : M01) A_2 (M08 : M09) A_3 (M12 : M13) B_2 (M09 : M13) Sort (A_1, B_0)	MbusW 00:15 MbusW 16:31 MbusN 16:31, MbusS 16:31 DC08-09H, MbusE 33:47 DC12-13H, MbusE 48:63 DC09-13V, MbusN 16:31 --
9	L *h* A_4 L *h* B_5 L *l* A_6 L *l* B_7 WB *l* A_0 WB *l* B_0 Sort (A_2, B_1) Sort (A_3, B_2)	MbusW 00:15 MbusW 16:31 MbusW 31:47 MbusW 48-63 MbusE 00:15 MbusE 16:31 - **-**
10	L *l* B_4 L *l* A_5 L h A_6 L *h* B_7 WB *h* A_0 WB *h* B_0 WB *l* A2 WB *l* B2	MbusW 00:15 MbusW 16:31 MbusW 31:47 MbusW 48-63 MbusE 00:15 MbusE 16:31 MbusE 31:47 MbusE 48-63
11	L *h* B_4 L *h* A_5 L *l* B_6 L *l* A_7 WB *l* B_3 WB *l* A_1 WB *h* A_2 WB *h* B_2	MbusW 00:15 MbusW 16:31 MbusW 31:47 MbusW 48-63 MbusE 00:15 MbusE 16:31 MbusE 31:47 MbusE 48-63
12	Sort (A_4, B_4) Sort (A_5, B_5) L h B_6	- - MbusW 31:47

	L *h* A_7 WB *h* B_3 WB *h* A_1 WB *l* B_1 WB *l* A_3	MbusW 48-63 MbusE 00:15 MbusE 16:31 MbusE 31:47 MbusE 48-63
13	Sort (A_6, B_6) Sort (A_7, B_7) B_4 (M00 : M01) B_5 (M04 : M05) WB *l* B_1 WB *l* A_3	-- -- DC00-01H, MbusE 00:15 DC04-05H, MbusE 16:31 MbusE 31:47 MbusE 48-63
14 (Same resources as Cycle 7)	B_4 (M01 : M05) A_5 (M04 : M05) B_5 (M05 : M09) B_6 (M08 : M09) B_7 (M12 : M13)	DC01-05V, MbusN 16:31 DC01-05H, MbusE 16:31 DC05-09V, MbusS 16:31 DC08-09H, MbusE 32:47 DC12-13H, MbusE 48:63
28	A_{12}(M00 : M01) B_{12} (M01 : M05) A_{13} (M04 : M05) B_{13} (M05 : M09) B_{14} (M08 : M09) B_{15} (M12 : M13)	DC00-01H, MbusE 00:15 DC01-05V, MbusN 16:31 DC01-05H, MbusE 16:31 DC05-09V, MbusS 16:31 DC08-09H, MbusE 32:47 DC12-13H, MbusE 48:63
Ending Cycles: 29	B_{15} (M13 : M01) A_{14} (M08 : M09) A_{15} (M12 : M13) B_{14} (M09 : M13) Sort (A_{13}, B_{12})	MbusN 16:31, MbusS 16:31 DC08-09H, MbusE 33:47 DC12-13H, MbusE 48:63 DC09-13V, MbusN 16:31 --
30	WB *l* A_{12} WB *l* B_{12} Sort (A_2, B_1) Sort (A_3, B_2)	MbusE 00:15 MbusE 16:31 - -
31	WB *h* A_{12} WB *h* B_{12} WB *l* A_{14} WB *l* B_{14}	MbusE 00:15 MbusE 16:31 MbusE 31:47 MbusE 48-63
32	WB *l* B_{15} WB *l* A_{13} WB *h* A_{14} WB *h* B_{14}	MbusE 00:15 MbusE 16:31 MbusE 31:47 MbusE 48-63
33	WB *h* B_{15} WB *h* A_{13} WB *l* B_{13} WB *l* A_{15}	MbusE 00:15 MbusE 16:31 MbusE 31:47 MbusE 48-63
34	WB *l* B_{13} WB *l* A_{15}	MbusE 31:47 MbusE 48-63

Note:

It is important to check that there are no resources used twice in a particular cycle. The outputs will be assigned in a sorter-absober circuit with the help of this table.

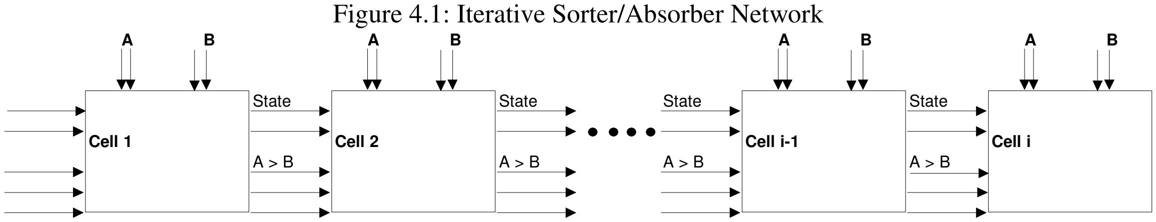
Chapter 4 Design of Components

Project consists of a number of hierarchical components, some of which have been created for testing purposes and design abstraction. From the top level we have a board schematic (automatically converted to VHDL by OrCAD Express) Board has all high-level components: main controller, memory controllers, four SRAM memories, two switches, and FPGA sorter/absorber array. FPGA array is a hierarchical block representing a schematic of the FPGA 4 by 2 chip array. All connections on the FPGA array schematic correspond to the way the chips are connected on the DEC-Perle board. Going one more hierarchy down, each FPGA consists of a hierarchical block with a sorting iterative network and a shell controlling when inputs are read or outputs are written back, as well as enable for the sort/absorb process inside the iterative network. The smallest hierarchical components is actual cell of the iterative circuit. In this chapter, we will start description from this smallest component of the iterative circuit, cell.

4.1 Iterative circuit theory and iterative circuit cell design

An iterative sort/absorb network is a structure composed of a cascade of identical circuits or cells. The iterative network will accomplish spatially what can be done by state machine. The structure of iterative network is shown in figure 4.1. Each input term is divided into 2-bit parts that represent one of the four possible states of the variables : absorbed, 0, 1 and missing and they are encoded as “00”, “10”, “01”, “11”, respectively. The input numbers or product terms are applied to each cell and inter-cell carry signals are propagated from first cell to the last cell. The data gets processed starting from highest significant bit and continues to the lowest. The condition of the data gets “recorded” in the signals propagated in this iterative network. In this implementation of the iterative circuit, unlike general definition of the iterative circuits, cells do not produce output, but the final carry from the last cell is taken to determine what will be sorted/absorbed output of the circuit. Lets look at the signals propagated in this network.

Figure 4.1: Iterative Sorter/Absorber Network



There are two groups of carry signals that are propagated from cell to cell. First group is called *state* and it keeps track of the state of absorption determination. The state signal is 2 bit wide and has following meaning:

“00” No absorption -- *Means that for all variables examined so far, it was determined that absorption cannot take place*
“01” B is absorbed into A -- *Means that for all variables examined so far, it was determined that B can be absorbed into A. The decision is “recorded” by this state*
“10” A is absorbed into B -- *Means that for all variables examined so far, it was determined that B can be absorbed into A. The decision is “recorded” by this state*
“11” Both products can be absorbed -- *Means that for all variables examined so far, it was determined that A and B are equal so far, that both B can be absorbed into A or A into B. Also means that they are equal products if this is the last cells output*

This absorption process is shown is a state diagram in Figure 4.2. This state diagram was used to generate next state signals for the iterative cells.

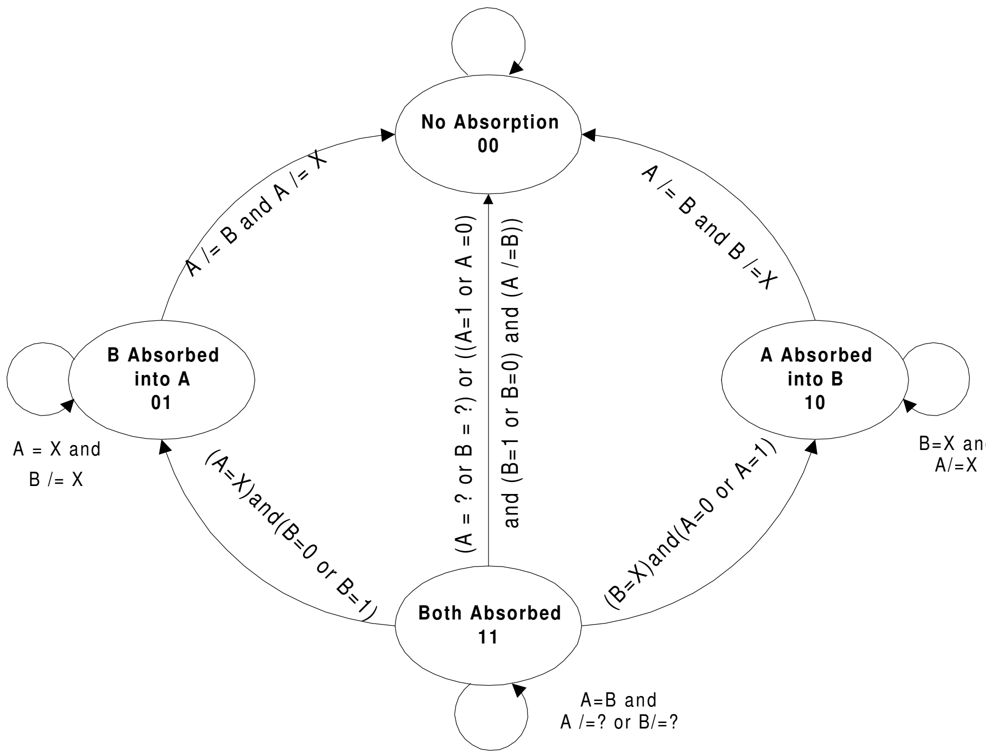


Figure 4.2 State diagram for absorption process

The second group of signals, denoted by “A>B” in Figure 4.1, indicates the state of comparison of the two products. Figure 4.3 shows the state diagram corresponding to this group of signals. When bit-by-bit comparison starts, the machine starts in default state of “A=B”. If the variables are both present, signal will remain in A=B or A>B or A<B transition takes place, States A>B or B<A may be used to implement integer number sorting in the circuit. When one variable is missing, 1BX or 1AX transition happens based upon whether A or B contained an “x”. Once a transition to 1BX or 1AX has happened, the machine will state in one of these states. When further “x”s are encountered, the machine transitions to 2BX or 2AX appropriately. What this means is that only delta between “x”s, in A and B are counted and not the absolute number of “x”s. This is desired as if one of the variables is x, we only want to sort the products based upon which products has *more* “x”s and as stated earlier absolute number of “x”s is irrelevant for our purpose.

As this is the most complex state transition diagram in this design, let’s illustrate this with an example. Suppose we have the following two terms:

A: 1 X X 1
B: 0 1 X 1

The first two encoded bits are compared (starting from the MSB on the left side of the table) and since A>B the “machine” transitions to A>B (001) state. On the next bit-by-bit comparison the machine transitions to 1AX state as X is stronger then 1. (The meaning is to sort bigger vs. smaller products, as opposed to sorting just numbers in A>B or A<B) Now the next bits are both “X”. Since circuit is in 1AX state, we know that A so far had one more missing variable, “X” and both “X” does not change this distance. So circuit continues in 1AX state. On the last bit comparison we do not change state.

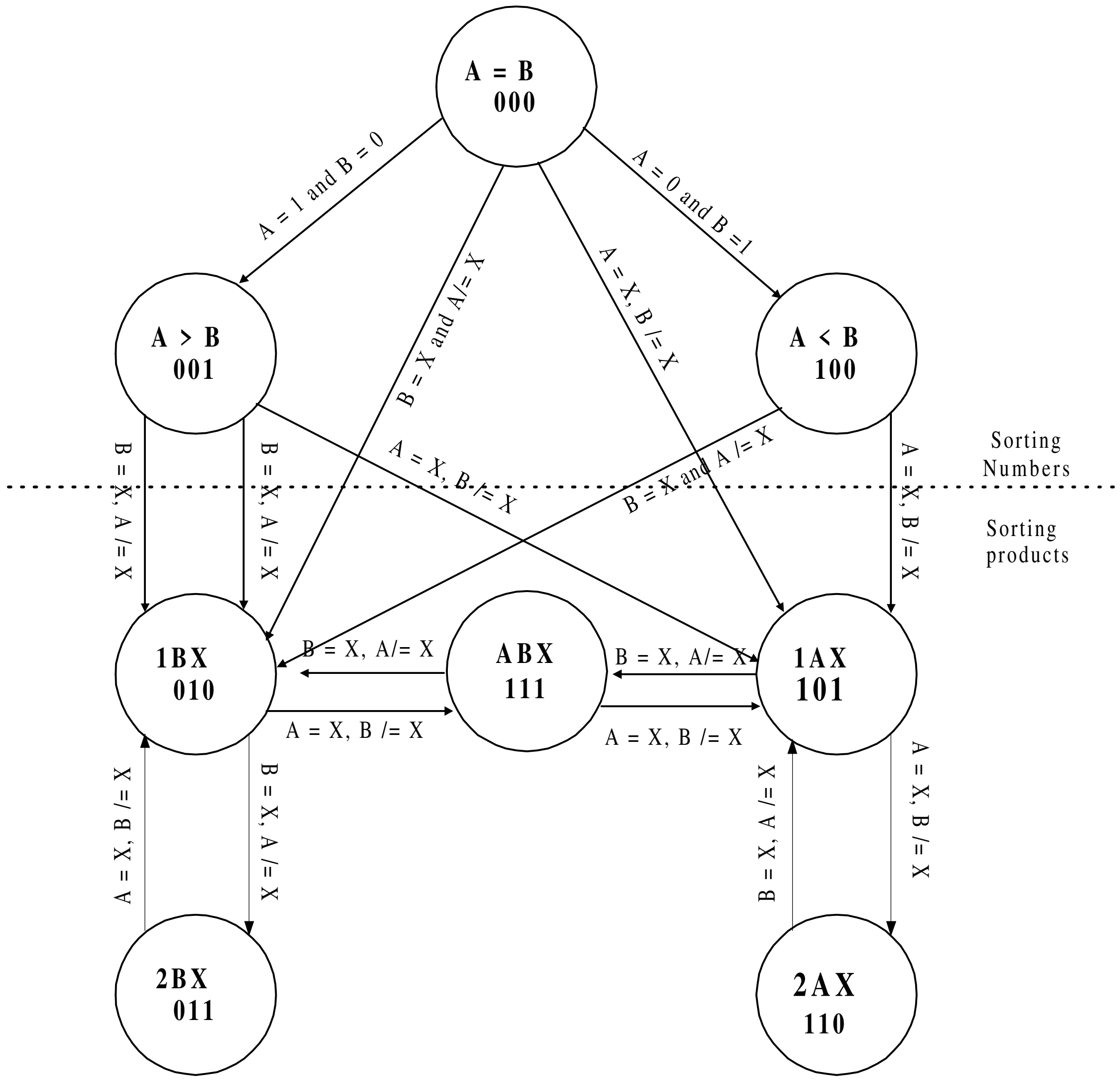


Figure 4.3: Comparison signals of the iterative circuit.

The operation of the cell is described in the table 4.1. This table specifies cell carry outputs based on the carry inputs from previous cell and cell inputs. Present state and next state represent the condition of the carry signals between cells as derived in the previous section. This iterative cell is the smallest part of the circuit and is called from the schematic consisting of 16 such cells combined into sorting network.

Table 4.1. Cell Table for Iterative Sorter/Absorber

Present State	Next State						$a_{i,j}$, $b_{i,j}$									
	0 0 *10 10*	**0 1** *10 01*	**1 0** *01 10*	**1 1** *01 01*	**X 0** *11 10*	**X 1** *11 01*	**0 X** *10 11*	**1 X** *01 11*	**X X** *11 11*	**? 0** *00 10*	**? 1** *00 01*	**0 ?** *10 00*	**1 ?** *01 00*	**? ?** *00 00*	**? X** *00 11*	**X ?** *11 00*
1 1	1 1	0 0	0 0	1 1	0 1	0 1	1 0	1 0	1 1	0 0	0 0	0 0	0 0	0 0	0 0	0 0
1 0	1 0	0 0	0 0	1 0	0 0	0 0	1 0	1 0	1 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0
0 1	0 1	0 0	0 0	0 1	0 1	0 1	0 0	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	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 0

4.2 Design of Sorter/Absorber Circuits

Sorter/Abosrber circuit consists of an iterative sorter/absorber network and a shell that describes its behavior in the pipeline according to the cycle information supplied. A block diagram is shown in figure 4.4. below. This circuit has different configurations and described in appendix VHDL files sorter_circuit[1-5]

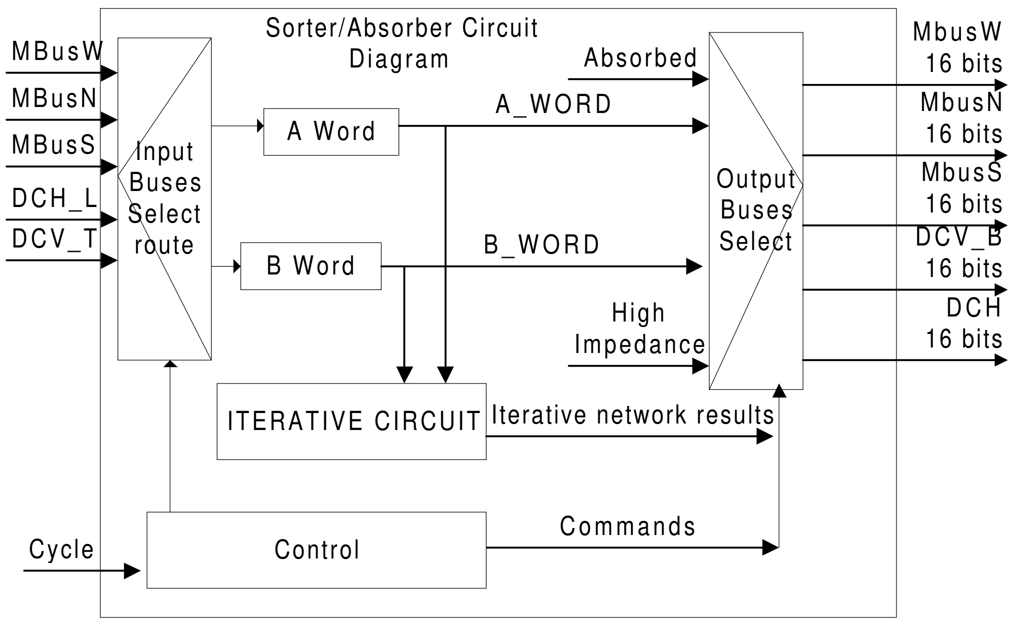


Figure 4.4 Sorter – Absorber circuit block diagram

4.3 Design of switches and SRAM memory.

Perhaps the simplest parts of the circuit are switches. All the switches have to do is to latch data on the bus according to the cycle data provided by controllers. All that is required for their operation are 22 states and operations as derived from pipeline description of chapter 3.

SRAM memory models are also writen to facilitate testing of the whole project. At the beginning of test, memory image is loaded from a file, with up to 4 files used for project. At any time, done signal being asserted from test fixture will result in memory writing its contents to a file to be examined. Memory has 18 adrress lines, memory read and write signals and 32 bit data bus.

4.4 Design of Main Controller and Memory Controllers.

As previously described, circuit can be in the following modes at any time: idle, sorting data from left to right, sorting from right to left, reading data from incoming FIFO, writing to incoming FIFO. These are called main tasks or states of the main controller state machine.
The main controller for the circuit is programmed in FSW. Its main function is to signal memory controllers top level operations such as writing data to outgoing FIFO, read data from incoming FIFO, mode of operation of memory controller – slave of master. As can be seen from figure ## there are 6 lines connecting FSW chip and CNE & CSW controllers on DEC-Perle board. This means that main controller will not be able to generate cycle signals that can be used by CNE and CSW as well as ring matrix cycle. What this means is that the cycle signals will have to be generated by one of the memory controllers depending on which one performing reading from memory on a current cycle.

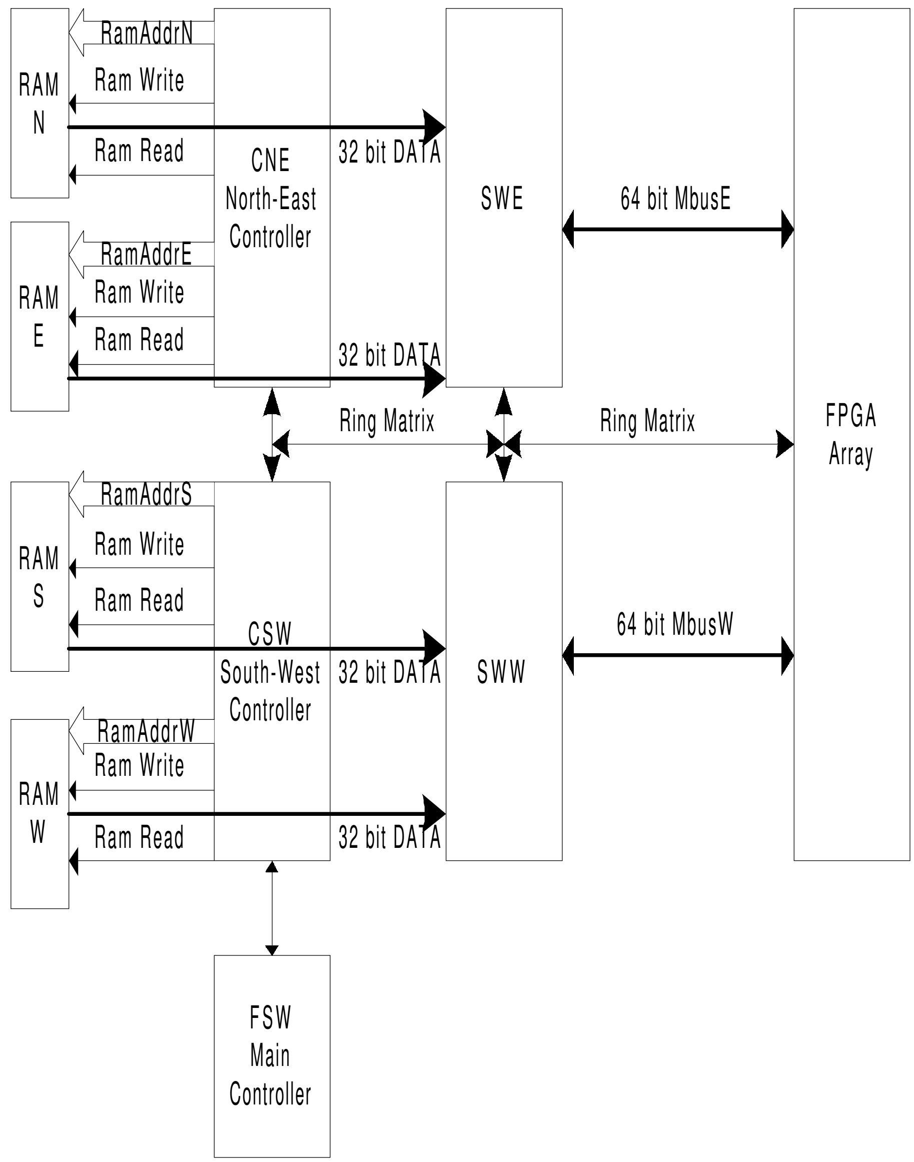
Thre are two memory controllers - CNE and CSW. Their and main controllers tasks may be described as follows:

1. Main controller determines when sorting starts and will start sorting from left-to-right data sorting (*CSW master*) and assert enable.
2. During a left to right data sorting, CSW controller will be responsible for generation of the cycle signals, keeping track of the progress of sorting relativley to data count, generating read addresses for two memories – S and W and keeping track of signal from sorters indicating whether sorting took place.
3. During same left to right data sorting, the duty of CNE is to read cycle signals from CSW and generate read addresses for N and E memories.
4. Once all data in memories S and W has been sorted, CNE will signal to main controller that left to right sorting has been completed and whether swap of data or absorption took place during left-to-right sorting.
5. If no absorption or swapping took place in left-to-right sort and this upside-down sort is done, main controller will initiate writeback state and controllers will generate addresses needed to write data back to host. If swap or absorption occurred main controller will then make CNE master controller and CSW slave controller and their functions will be reversed and similar as described above.
6. This left-to-right and right-to-left sorting will continue until there is no change of data (no sorting of absorption) takes place while data is processed. This step guarantees that sorting has taken place. It does not guarantee however that all possible absorption occurred because it is possible that data was pre-sorted and not every product met with another product in sorter/absorber chip. This means that sorting is repeated starting from buttom of the data array and coming up. Since data was already sorted once, all data members that were on top will have to propagate to the top again and therefore every product will «see» another product.
7. During the upside-down sort, addresses are generated in the opposite manner.

Chapter 5. VHDL code design and testing.

The platform for VHDL code design used in this project is OrCAD tools. Most of the project was done in straight VHDL, but schematic capture was used to help design of sorter iterative circuit and board. Each of the componets of the design has been written in VHDL including memory model. All of the componets are then combined using schematic capture into one large board that will be used to test and debug operation of the design.
To test our design we run simulation on the highest hierarchical level of the design *board* and examine contents of memory files for correct output. Memory files are currently created by hand, but there is a Perl script being written that will create memory files for the project.

Figure 4.2 Memory Controller Diagram



Conclusion and Future Work

The code developed in this project has been simulated and next step is to take blocks and map them into devices using place and route tools available from Xilinx. Then it can be processed by software that will create a program files to be downloaded to FPGA board to test the hardware.

Bibliography:

Masters of Science in Electrical Engineering dissertation of Qihong Chen, Portland State University, Portland, 1998.

A Guide to VHDL -- Stanley Mazor and Patricia Langstraat, Synopsis , 1996

VHDL for programmable logic – Kevin Skahill, Cypress Semiconductor, 1996

OrCAD Express 7.2 Program Manual and Libraries, OrCAD 1998