Design of a SIMD Computer for Rough Set Theory

EE 510 - VHDL
Manjusha Dharmadhikari
Van Ngo
Torrey J. Lewis

Revision 2.0

June 11, 1999

Revision History

1.0 12/9 1998 - EE 572 - Advanced Logic Synthesis final paper

1.1 4/28 1999 - Added more detail to the calculation of Upper and Lower approximation for Table 9. Fixed UpperAPPROX algorithm from for $i = 1$ to m $E_i[m] := E_i[m] * B[m]; C[m] := 0/$ for $i = 1$ to m if $E_i[m] \neq 0$ then $C[m] := C[m] + E_i[m]$ to for $i = 1$ to m $S_i[m] := E_i[m] * B[m]; C[m] := 0/$ for $i = 1$ to m if $S_i[m] \neq 0$ then $C[m] := C[m] + E_i[m]$.
Fixed error in discussion of UpperAPPROX algorithm.
Fixed Table 17.
Fixed Table 18
Added more text to LowerAPPROX explanation.
Added UpperAPPROX explanation.

1.2 - 5/2 1999 - Added more explanation to generation of a solution for Table 8.

1.3 5/24 1999 - Added description of hardware design, added placeholders for behavioral code, memory interface, matrix.

1.4 6/9 1999 - Editing, added waveforms for memory interface.

2.0 6/11 1999 - Version turned in for EE 510VH.

To Do

- Change name of Positional Sums in algorithms, this is misleading. The operation is AND not OR, change S_i to something more meaningful.

- Change name of vertical vector $C[m]$ vector in algorithms, this can be confused with the C (Comperand) register.

- Explain generation of Table 8.

- put box around PRSComp Algorithms; widen margins

Abstract

This paper presents preliminary work for the design of a SIMD Computer to implement Rough Set Theory operations. Rough Set Theory is examined as it applies to logic minimization. The target platform for this project, the DEC PeRLe-1 board, is examined. PRSComp, Parallel Rough Set Computer, proposed by Mieczyslaw Muraszkiewicz and Henryk Rybinski is discussed as a possible hardware implementation of Rough Set Theory. Finally, an outline of future work is given.

Introduction

The FPGA SIMD machine for Rough Set Theory computations is particularly interesting for several reasons. The architecture of the DEC PeRLe-1 board a type of Programmable Active Memory, is a very unique approach. Second, the concepts of Rough Set Theory can be applied to logic minimization, as well as to many other applications and is not limited to binary values, but can also be used for multi-valued problems.

The DEC PeRLe-1 board [2] [4] is essentially an array of FPGAs that can be programmed by a host computer to perform any desired function. Typically there are two possibilities on how to implement a desired function. The function can be committed to application specific hardware, or it can be realized in software. There are drawbacks to both approaches, however. Hardware is fast, but is dedicated and can only perform the programmed function. Software has the flexibility to perform many different functions, through multiple programs, but is generally much slower than could be realized in hardware. The DEC PeRLe-1 board attempts to capture the advantages of both worlds and provide the speed of dedicated hardware, yet allow the flexibility of software. The DEC PeRLe-1 board has been used to implement several different functions and compares favorably with other implementations [4]. The DEC PeRLe-1 board will be covered more thoroughly in the section titled "DEC PeRLe-1 board".

Rough Set Theory is a mathematical notion presented by Zdzislaw Pawlak [1]. It can be used to minimize truth tables in much the same way as Karnaugh Maps. However, logic minimization is only one of the many applications of Rough Set Theory. For example, Rough Set Theory has been used in Data Mining [8]. Rough Set Theory and how it can be used for logic minimization is discussed in the section titled "Rough Set Examples".

Rough Set Examples

Following is a logic minimization example to explain how the Rough Set Theory can be applied to logic minimization. The example is taken from Rough Sets: Theoretical Aspects of Reasoning about Data by Zdzislaw Pawlak [1]. The same data will be used later in demonstrating Muraszkiewicz and Rybinski's PRSComp algorithms. This will give us the opportunity to compare the Rough Set Theory method to the PRSComp algorithms.

Table 1 shows the definition of a desired circuit in the form of a truth table, also known as a decision table. Variables a, b, c, d, and e are input variables and f is the output variable. The goal is to minimize the function described by Table 1 using the Rough Set Theory method. (The same problem can also be solved using Karnaugh maps, but our interest here lies in software/hardware realizable algorithms.) The procedure is to first eliminate unneeded input variables (so-called vacuous variables) from the table and then eliminate unnecessary values of variables from each decision rule. The result will be a minimum solution to

Table 1: Initial Truth Table

U	a	b	c	d	e	f
1	0	0	0	1	1	1
2	0	1	0	0	0	1
3	0	1	1	0	0	1
4	1	1	0	0	0	1
5	1	1	0	1	0	1
6	1	1	1	1	0	1
7	1	1	1	0	0	1
8	1	0	0	1	1	1
9	1	0	0	1	0	1
10	0	0	0	0	0	0
11	0	0	0	1	0	0
12	0	1	1	1	0	0
13	0	1	1	0	1	0
14	1	1	1	1	1	0
15	1	0	0	0	1	0

the problem. This procedure will be explained through example.

——————

Definition 1

Decision rule.

A decision rule is defined to be the relation of the set of input variables to the set of output variables. In our example it is a row in the truth table.
——————

The first step is to find the vacuous variables and remove them. This is performed by removing each input variable one at a time and then determining whether the resulting truth table is still consistent.

——————

Definition 2

Consistency of a table.

For every combination of input variables presented in the table there is a unique value for the output.
——————

For example, the initial truth table is known to be consistent because for every input combination of a, b, c, d, and e, there is a unique value for f. If there happened to be a row 16, that were identical to row 1 except that the output variable f were defined to be 0, the table would be inconsistent.

Another important concept of Rough Sets Theory is whether or not an input variable is dispensable.

————–

Definition 3

Dispensable

An input variable is defined as being dispensable if it is possible to remove the variable from the table and result in a consistent table. A dispensable variable is unneeded in the final table and is also referred to as a vacuous variable.

—————–

Hence, a dispensable input variable does not effect the value of the output variable and can thus be removed from the circuit without any changes. If a variable is not dispensable then it must be indispensable.

—————–

Definition 4

Indispensable

An input variable is defined to be indispensable if removing the variable results in an inconsistent table.

—————–

Hence, f-indispensable variables effect the output value and therefore must remain in the table.

By removing one variable (i.e. column) at a time from the table we can find which variables can be removed and still result in a consistent table. Tables 2 through 6 show the results of the removal of columns.

From Table 2, we can see that input variable a is f-indispensable because removing it results in an inconsistent table. The table is inconsistent because decision rules 6 and 12 yield different output variables for identical input variables. Both decision rule 6 and 12 have input values of 1, 1, 1 and 0 for a, b, c, d and e respectively. The determining factor is that the output values differ, 1 for decision rule 6 and 0 for decision rule 12. Similarly with decision rules 9 and 11.

From Table 3, we can see that input variable b is f-indispensable because rows 2 and 10 are inconsistent.

Table 4 shows that input variable c is f-dispensable because removing it from the table results in a consistent table. This means that input variable c does not need to be present in the final solution.

Table 2: Removal of a

U	b	c	d	e	f
1	0	0	1	1	1
2	1	0	0	0	1
3	1	1	0	0	1
4	1	0	0	0	1
5	1	0	1	0	1
6	1	1	1	0	1
7	1	1	0	0	1
8	0	0	1	1	1
9	0	0	1	0	1
10	0	0	0	0	0
11	0	0	1	0	0
12	1	1	1	0	0
13	1	1	0	1	0
14	1	1	1	1	0
15	0	0	0	1	0

Table 3: Removal of b

U	a	c	d	e	f
1	0	0	1	1	1
2	0	0	0	0	1
3	0	1	0	0	1
4	1	0	0	0	1
5	1	0	1	0	1
6	1	1	1	0	1
7	1	1	0	0	1
8	1	0	1	1	1
9	1	0	1	0	1
10	0	0	0	0	0
11	0	0	1	0	0
12	0	1	1	0	0
13	0	1	0	1	0
14	1	1	1	1	0
15	1	0	0	1	0

Table 4: Removal of c

U	a	b	d	e	f
1	0	0	1	1	1
2	0	1	0	0	1
3	0	1	0	0	1
4	1	1	0	0	1
5	1	1	1	0	1
6	1	1	1	0	1
7	1	1	0	0	1
8	1	0	1	1	1
9	1	0	1	0	1
10	0	0	0	0	0
11	0	0	1	0	0
12	0	1	1	0	0
13	0	1	0	1	0
14	1	1	1	1	0
15	1	0	0	1	0

Table 5: Removal of d

U	a	b	c	e	f
1	0	0	0	1	1
2	0	1	0	0	1
3	0	1	1	0	1
4	1	1	0	0	1
5	1	1	0	0	1
6	1	1	1	0	1
7	1	1	1	0	1
8	1	0	0	1	1
9	1	0	0	0	1
10	0	0	0	0	0
11	0	0	0	0	0
12	0	1	1	0	0
13	0	1	1	1	0
14	1	1	1	1	0
15	1	0	0	1	0

Table 6: Removal of e

U	a	b	c	d	f
1	0	0	0	1	1
2	0	1	0	0	1
3	0	1	1	0	1
4	1	1	0	0	1
5	1	1	0	1	1
6	1	1	1	1	1
7	1	1	1	0	1
8	1	0	0	1	1
9	1	0	0	1	1
10	0	0	0	0	0
11	0	0	0	1	0
12	0	1	1	1	0
13	0	1	1	0	0
14	1	1	1	1	0
15	1	0	0	0	0

Table 5 shows that input variable d is f-indispensable because rows 3 and 12, and rows 8 and 15 disagree with each other.

Table 6 shows that input variable e is f-indispensable because decision rules 1 and 11, 3 and 13, and 6 and 14 are inconsistent.

We do not remove variable f because it is the output variable and cannot be removed and result in the desired functionality.

Input variable c is the only f-dispensable variable in the original table. Table 7 is Table 4 with combining identical decision rules. For this reason further analysis will be performed on Table 7.

When used for referencing decision rules from Table 7, the first value listed in the U column will be used in the case of (2,3), (4,7), and (5,6).

Now that unneeded input variables have been removed from the original truth table we can now move on to the next step: removing unneeded values of input variables. This is known as finding the core values. To do this we must drop one input variable at a time and then determine that the intersection of the remaining input variables is included in the set of output variables having the same value. This must be done for each decision rule independently.

———

Definition 5

Core Values

Core values are those values that must be kept, rather than dropped, because dropping them would result in realizing a different function.

Table 7: Combining rules of Table 4

U	a	b	d	e	f
1	0	0	1	1	1
2,3	0	1	0	0	1
4,7	1	1	0	0	1
5,6	1	1	1	0	1
8	1	0	1	1	1
9	1	0	1	0	1
10	0	0	0	0	0
11	0	0	1	0	0
12	0	1	1	0	0
13	0	1	0	1	0
14	1	1	1	1	0
15	1	0	0	1	0

———

First, using the rough set notation, we must prove that the values of input variables for decision rule 1 are unique (i.e. no other decision rule have the same combination of input values).

$F = [1]_{\{a,b,d,e\}} = [1]_a \cap [1]_b \cap [1]_d \cap [1]_e = \{1,2,10,11,12,13\} \cap \{1,8,9,10,11,15\} \cap \{1,5,8,9,11,12,14\} \cap \{1,8,13,14,15\} = \{1\}$

Now for an explanation of the notation used above. The number appearing within the square brackets are the decision rule number. The set $[1]_a$ is the set of all decision rules that have the same value for input variable a. Decision rule 1 has the value of 0 for input variable a. From Table 7 we can see that decision rules 2, 10, 11, 12 and 13 also have a value of 0 for input variable a, thus they all belong to the same set, $[1]_a$. Similarly for the sets $[1]_b$, $[1]_d$ and $[1]_e$.

Next to determine the set $[1]_f$ that contains all decision rules that have the same value for the output variable f as decision rule 1.

$[1]_f = \{1,2,4,5,8,9\}$

To determine which input values are essential to decision rule 1 we must find the core values. To perform this each input variable must be dropped one at a time and then the resulting intersection is compared to $[1]_f$. If the resulting intersection is a subset of $[1]_f$ then the variable can be dropped, meaning it is not a core value. The following equations show the dropping of one variable at a time and obtaining the intersection.

$\bigcap(F - [1]_a) = [1]_b \cap [1]_d \cap [1]_e = \{1,8,9,10,11,15\} \cap \{1,5,8,9,11,12,14\} \cap \{1,8,13,14,15\} = \{1,8\}$

$\bigcap(F-[1]_b) = [1]_a \cap [1]_d \cap [1]_e = \{1,2,10,11,12,13\} \cap \{1,5,8,9,11,12,14\} \cap \{1,8,13,14,15\} = \{1\}$

$\bigcap(F-[1]_d) = [1]_a \cap [1]_b \cap [1]_e = \{1,2,10,11,12,13\} \cap \{1,8,9,10,11,15\} \cap \{1,8,13,14,15\} = \{1\}$

$\bigcap(F-[1]_e) = [1]_a \cap [1]_b \cap [1]_d = \{1,2,10,11,12,13\} \cap \{1,8,9,10,11,15\} \cap \{1,5,8,9,11,12,14\} = \{1,11\}$

The first equation gives the result $\{1,8\}$. $\{1,8\} \subseteq \{1,2,4,5,8,9\} = [1]_f$. This means that dropping input variable a from decision rule 1 has no adverse effect on the value of the output variable with regards to the other decision rules. Likewise for the second and third equation.

The fourth equation yields a result of $\{1,11\}$. $\{1,11\} \not\subseteq \{1,2,4,5,8,9\} = [1]_f$. This means that dropping input variable e from decision rule 1 does change the correct value for the output variable in another decision rule. Specifically if we were to drop input variable e, decision rule 11 would not retain the proper value for the output variable f. For this reason we must keep the value for input variable e for decision rule 1.

The same procedure is followed for decision rule 2 and is shown below. $F =$

$[2]_{\{a,b,d,e\}} = [2]_a \cap [2]_b \cap [2]_d \cap [2]_e = \{1,2,10,11,12,13\} \cap \{2,4,5,12,13,14\} \cap \{2,4,10,13,15\} \cap \{2,4,5,9,10,11,12\} = \{2\}$

$[2]_f = \{1,2,4,5,8,9\}$

$\bigcap(F-[2]_a) = [2]_b \cap [2]_d \cap [2]_e = \{2,4,5,12,13,14\} \cap \{2,4,10,13,15\} \cap \{2,4,5,9,10,11,12\} = \{2,4\} \subseteq [2]_f$

$\bigcap(F-[2]_b) = [2]_a \cap [2]_d \cap [2]_e = \{1,2,10,11,12,13\} \cap \{2,4,10,13,15\} \cap \{2,4,5,9,10,11,12\} = \{2,10\} \not\subseteq [2]_f$

$\bigcap(F-[2]_d) = [2]_a \cap [2]_b \cap [2]_e = \{1,2,10,11,12,13\} \cap \{2,4,5,12,13,14\} \cap \{2,4,5,9,10,11,12\} = \{2,12\} \not\subseteq [2]_f$

$\bigcap(F-[2]_e) = [2]_a \cap [2]_b \cap [2]_d = \{1,2,10,11,12,13\} \cap \{2,4,5,12,13,14\} \cap \{2,4,10,13,15\} = \{2,12\} \not\subseteq [2]_f$

Note that because removing input variables b, d and e result in extra decision rules appearing in the resulting intersection that are not a subset of $[1]_f$ these input variables cannot be removed from decision rule 2.

Appendix A shows the computations to generate the core values through decision rule 12.

Table 8 shows the results of computing the core values for all decision rules. It should be noted that Table 8 does not directly correlate to the Karnaugh map.

Now to compute the value reducts for each decision rule. The value reducts for each decision rule is a solution for that decision rule.

As when minimizing a decision table using a Karnaugh map, the procedure is to look for the largest possible circles, to make the final solution as simple as possible. Similarly with this method, first, a solution is sought with the least number of literals. If a solution is not found then another literal is added. Adding literals continues until a solution is found.

From above we see that the solution set for decision rule 1 is $[1]_f = \{1, 2, 4, 5, 8, 9\}$. First we check all combinations of 1 input variable to see if the result is a subset of the solution set.

$[1]_a = \{1, 2, 10, 11, 12, 13\} \not\subseteq [1]_f$ $[1]_b = \{1, 8, 9, 10, 11, 15\} \not\subseteq [1]_f$ $[1]_d = \{1, 5, 8, 9, 11, 12, 14\} \not\subseteq [1]_f$ $[1]_e = \{1, 8, 13, 14, 15\} \not\subseteq [1]_f$

Seeing that none is a subset of $[1]_f$, another literal must be added. Now, check all combinations of two input variables.

$[1]_a \cap [1]_b = \{1, 10, 11\} \not\subseteq [1]_f$ $[1]_a \cap [1]_d = \{1, 11, 12\} \not\subseteq [1]_f$ $[1]_a \cap [1]_e = \{1, 13\} \not\subseteq [1]_f$ $[1]_b \cap [1]_d = \{1, 8, 9, 11\} \not\subseteq [1]_f$ $[1]_b \cap [1]_e = \{1, 8, 15\} \not\subseteq [1]_f$ $[1]_d \cap [1]_e = \{1, 8, 14\} \not\subseteq [1]_f$

Again no result is a subset of $[1]_f$, so another literal is added. Now to check all combinations of three input variables.

$[1]_a \cap [1]_b \cap [1]_d = \{1, 11\} \not\subseteq [1]_f$ $[1]_a \cap [1]_b \cap [1]_e = \{1\} \subseteq [1]_f$ $[1]_a \cap [1]_d \cap [1]_e = \{1\} \subseteq [1]_f$ $[1]_b \cap [1]_d \cap [1]_e = \{1, 8\} \subseteq [1]_f$

Here three combinations of input variables were found to be a subset of the solution set $[1]_f$. Namely, the intersection of input variables: 1) a, b, e, 2) a, d, e, and 3) b, d, e all are a subset of $[1]_f$. The value reducts of decision rule 1 are:

$a = 0 b = 0 e = 1$ $a = 0 d = 1 e = 1$ $b = 0 d = 1 e = 1$

Note that input variable e is included in all three value reducts. A core value must be included in all value reducts found.

In order to "cover" decision rule 1, at least one of the value reducts must be included in the final solution.

The same method is used to determine all value reducts for decision rule 2. From above we see that the solution set for decision rule 2 is $[2]_f = \{1, 2, 4, 5, 8, 9\}$. First we check all combinations of one input variable to see if the result is a subset of the solution set.

$[2]_a = \{1, 2, 10, 11, 12, 13\} \not\subseteq [2]_f$ $[2]_b = \{2, 4, 5, 12, 13, 14\} \not\subseteq [2]_f$ $[2]_d = \{2, 4, 10, 13, 15\} \not\subseteq [2]_f$ $[2]_e = \{2, 4, 5, 9, 10, 11, 12\} \not\subseteq [2]_f$

Seeing that none is a subset of $[2]_f$, another literal must be added. Now, check all combinations of two input variables.

$[2]_a \cap [2]_b = \{2, 12, 13\} \not\subseteq [2]_f$ $[2]_a \cap [2]_d = \{2, 10, 13\} \not\subseteq [2]_f$ $[2]_a \cap [2]_e = \{2, 10, 11, 12\} \not\subseteq [2]_f$ $[2]_b \cap [2]_d = \{2, 4, 13\} \not\subseteq [2]_f$ $[2]_b \cap [2]_e = \{2, 4, 5, 12\} \not\subseteq [2]_f$ $[2]_d \cap [2]_e = \{2, 4, 10\} \not\subseteq [2]_f$

Again no result is a subset of $[2]_f$, so another literal is added. Now to check all combinations of three input variables.

$[2]_a \cap [2]_b \cap [2]_d = \{2, 13\} \not\subseteq [1]_f$ $[2]_a \cap [2]_b \cap [2]_e = \{2, 12\} \not\subseteq [1]_f$ $[2]_a \cap [2]_d \cap [2]_e = \{2, 10\} \not\subseteq [1]_f$ $[2]_b \cap [2]_d \cap [2]_e = \{2, 4\} \subseteq [1]_f$

Now that a solution has been found we stop. There is no need to add any more literals. Doing so, would needlessly complicate the final solution. Here only one combination of input variables were found to be a subset of the solution set $[2]_f$. Namely, the intersection of input variables: b, d, e. The value reduct of decision rule 2 is:

$b = 1 d = 0 e = 0$

Table 8: Results of computing cores

U	a	b	d	e	f
1	-	-	-	1	1
2,3	-	1	0	0	1
4,7	-	-	-	-	1
5,6	1	-	-	0	1
8	-	0	1	-	1
9	1	-	-	-	1
10	-	0	-	-	0
11	0	-	-	0	0
12	0	-	-	0	0
13	-	-	-	1	0
14	-	1	-	1	0
15	-	-	0	-	0

Table 9: Reducts of Table 7

U	a	b	d	e	f
1	0	0	x	1	1
1'	0	x	1	1	1
1"	x	0	1	1	1
2	x	1	0	0	1

Note that input variable b, d, e are included in all value reducts, thus they are also the core values for decision rule 2.

In order to "cover" decision rule 1, at least one of the value reducts must be included in the final solution. In order to "cover" decision rule 1, at least one of the value reducts must be included in the final solution. Once all value reducts for each decision rule is obtained all possible solutions to the problem have been found.

Table 9 shows the results of calculating all of the value reducts for Table 7.

By combining rules from Table 9 the minimal solution can be found to be the following.

$bd'e' \vee ae' \vee b'de \rightarrow f$

It should be noted that the same solution can be obtained through the use of Karnaugh maps. This method may be more familiar to circuit designers and therefore is presented below in Figure 1 to help the reader with an engineering background to analyze the minimal solution found above using Rough Set Theory. To simplify the Karnaugh map, input variable c has already been dropped.

When defining the truth table for a logic circuit there is never a case where the output value can take on different values for the same set of input values. However, in data mining, it is possible that the data collected could be erroneous or inaccurate, so called noisy data. In this case it is indeed possible to have different output values for the same input values in the data set. Table 10

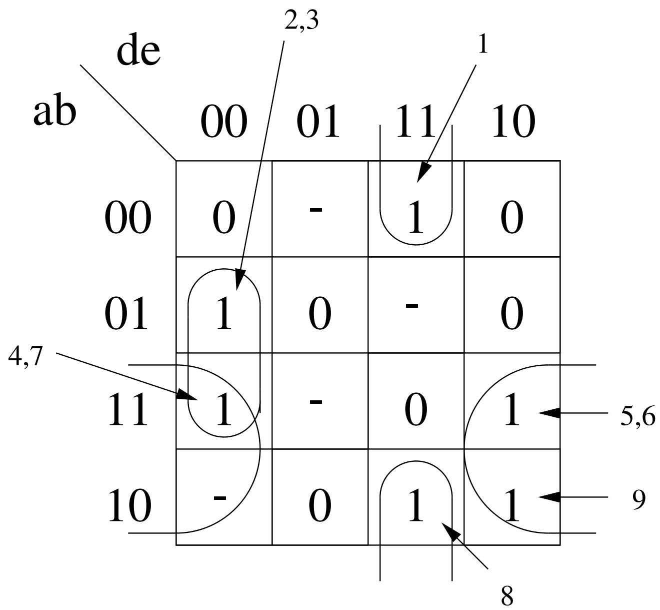

Figure 1: Karnaugh Map

shows a case with inaccurate data. Variables a, b, c and d are input variables while f is the output variable. Figure 2 presents the erroneous data in Karnaugh map form.

Three Rough Set Theory operations take erroneous data into account. They are the Lower and Upper Approximations, and the Boundary.

Pawlak [1] defines the lower and upper approximations and boundary in set notation as follows:

$\underline{R}X = \{x \in U : [x]_R \subseteq X\}$ lower approximation
$\overline{R}X = \{x \in U : [x]_R \cap X \neq \emptyset\}$ upper approximation
$BN_R(X) = \overline{R}X - \underline{R}X$ boundary

$\underline{R}X$ is defined as the lower approximation. x is a decision rule of the universe. U is the universe of knowledge as shown by Table 10. R is an equivalence relation over U and $[x]_R$ is a category in R containing an element $x \in U$. By saying that $[x]_R$ is a category in R it is meant that $x \subseteq R$. The lower approximation is the set of all elements in U that can be said with certainty that they are elements of X in the knowledge R.

$\overline{R}X$ is defined as the upper approximation. Elements in the upper approximation can possibly be classified as elements of X.

$BN_R(X)$ is defined as the boundary. Elements in the boundary cannot be classified as being elements of X or not being elements of X.

Table 10: Dataset with inaccurate data

U	a	b	c	d	f
1	0	0	0	0	1
2	0	0	1	0	0
3	0	0	1	0	1
4	0	0	1	1	0
5	0	0	1	1	1
6	0	1	0	0	1
7	0	1	0	1	1
8	0	1	1	0	0
9	0	1	1	1	0
10	1	0	0	1	1
11	1	0	1	0	1
12	1	0	1	1	1
13	1	1	0	1	0
14	1	1	1	0	0
15	1	1	1	1	0

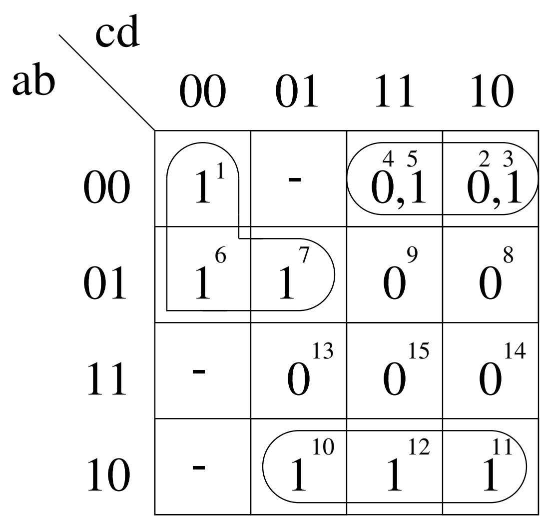

Figure 2: Karnaugh map showing Lower and Upper Approximation and Boundary

Pawlak goes on to define the following:

$POS_R(X) = \underline{R}X R - \mathit{positive}$ region of X
$NEG_R(X) = U - \overline{R}X R - \mathit{negative}$ region of X
$BN_R(X) R - \mathit{borderline}$ region of X

The positive region of X are elements that can be classified with certainty that they belong to X.

The negative region of X are those elements that can be classified with certainty that they do not belong to X.

The borderline region are those elements that can not be classified with certainty they belong to X or that they do not belong to X.

The following examples from Pawlak [1] are presented here in an attempt to clarify the definitions given above.

We are given a universe of knowledge $U = \{x_0, ..., x_{10}\}$ with equivalence classes:

$E_1 = \{x_0, x_1\}$
$E_2 = \{x_2, x_6, x_9\}$
$E_3 = \{x_4, x_5\}$
$E_4 = \{x_3, x_8\}$
$E_5 = \{x_7, x_{10}\}$

The example gives the following roughly R-definable sets.

$X_2 = \{x_0, x_3, x_4, x_5, x_8, x_{10}\}$
$Y_2 = \{x_1, x_7, x_8, x_{10}\}$
$Z_2 = \{x_2, x_3, x_4, x_8\}$

Using these sets, the example goes on to define the following approximations and boundaries.

$\underline{R}X_2 = E_3 \cup E_4 = \{x_3, x_4, x_5, x_8\}$
$\overline{R}X_2 = E_1 \cup E_3 \cup E_4 \cup E_5 = \{x_0, x_1, x_3, x_4, x_5, x_7, x_8, x_{10}\}$
$BN_R(X_2) = E_1 \cup E_5 = \{x_0, x_1, x_7, x_{10}\}$

To aid in understanding the meaning behind the Rough Set classifications the following should be noted.

The lower approximation, $\underline{R}X_2$, contains only E_3 and E_4 because each class have all of their elements in X_2. It can be said with certainty that E_3 and E_4 are elements of X_2 in the knowledge R.

The lower approximation, $\overline{R}X_2$, contains E_1, E_3, E_4 and E_5 because each have at least one element belonging to X_2. E_1, E_3, E_4 and E_5 can possibly be classified as elements of X_2.

The boundary, $BN_R(X_2)$, contains E_1 and E_5 because E_1 and E_5 contain at least one element that is a member of X_2 and at least one element that is not a member of X_2. E_1 and E_5 cannot be said with certainty that they belong to X_2 because each contains at least 1 element that is not in X_2.

$\underline{R}Y_2 = E_5 = \{x_7, x_{10}\}$
$\overline{R}Y_2 = E_1 \cup E_4 \cup E_5 = \{x_0, x_1, x_4, x_7, x_8, x_{10}\}$
$BN_R(Y_2) = E_1 \cup E_2$

$\underline{R}Z_2 = E_4 = \{x_4, x_8\}$
$\overline{R}Z_2 = E_2 \cup E_3 \cup E_4 = \{x_2, x_3, x_4, x_5, x_6, x_8, x_9\}$
$BN_R(Z_2) = E_2 \cup E_3 = \{x_2, x_3, x_5, x_6, x_9\}$

The reader is directed to Pawlak [1] pp. 18-19 for more information regarding this example.

Another way to present the same information is via the positive, negative and the borderline region. This will be demonstrated using Table 10 and Figure 2 as the Karnaugh map representation as an input dataset.

First, let us define equivalence classes based on the input values a, b, c and d.

$E_1 = \{x_1\}$ $E_2 = \{x_2, x_3\}$ $E_3 = \{x_4, x_5\}$ $E_4 = \{x_6\}$ $E_5 = \{x_7\}$ $E_6 = \{x_8\}$ $E_7 = \{x_9\}$ $E_8 = \{x_10\}$ $E_9 = \{x_11\}$ $E_{10} = \{x_12\}$ $E_{11} = \{x_13\}$ $E_{12} = \{x_14\}$ $E_{13} = \{x_15\}$

For this example let us define X to be the set of all decision rules that have an output value of 1.

$X = \{1, 3, 5, 6, 7, 10, 11, 12\}$

The following Rough Set classifications can be made:

$POS_R(X) = \underline{R}X = E_1 \cup E_4 \cup E_5 \cup E_8 \cup E_9 \cup E_10 = \{1, 6, 7, 10, 11, 12\}$
$\overline{R}X = E_1 \cup E_2 \cup E_3 \cup E_4 \cup E_5 \cup E_8 \cup E_9 \cup E_10 = \{1, 2, 3, 4, 5, 6, 7, 10, 11, 12\}$
$NEG_R(X) = U - \overline{R}X = \{8, 9, 13, 14, 15\}$
$BN_R(X) = \overline{R}X - \underline{R}X = \{2, 3, 4, 5\}$

From the above calculations the following are known:

From $POS_R(X)$ it is known that input values corresponding to 1,6,7,10,11 or 12 will definitely have an output of one.

From $\overline{R}X$ it is known that input values corresponding to 1,2,3,4,5,6,7,10,11 or 12 will possibly have an output of 1.

From $NEG_R(X)$ it is known that input values corresponding to 8,9,13,14 or 15 will definitely not have an output of 1.

From $BN_R(X)$ it is known that input values corresponding to 2,3,4 or 5 can not be classified whether the output will be 1 or not 1.

It should be noted the usage of "not 1" rather than 0. This will become important when expanding to multi-valued logic.

The Rough Set operations described above are not limited to only binary logic, but can also be extended to multi-valued logic as well.

DEC PeRLe-1 board

A possible target platform for this project is the DEC PeRLe-1 board that is available in the Department of Electrical and Computer Engineering at Portland State University. Qihong Chen's masters thesis "The Design of Cube Calculus Machine Using SRAM-based FPGA Reconfigurable Hardware DEC's PERLE-1 Board" [2] contains a chapter describing the DEC PeRLe-1 board. The reader is referred to this document for more information on the DEC PeRLe-1 board and its features.

The DEC PeRLe-1 board has many features that could be exploited and will be explored as the design process proceeds. Figure 3 shows the general architecture of the DEC PeRLe-1 board. The DEC PeRLe- 1 consists of 23 Xilinx 3090A FPGAs. These are shown in Figure 3 as FSW (FIFO Switch), SWN (North Switch), SWE (East Switch), SWS (South Switch), SWW (West Switch), CNE (North-East Controller), CSW (South-West Controller), M00 - M15 (Matrix Array).

The host processor communicates with the core (the 4x4 array of FPGAs) through the inbound and outbound FIFOs in the upper left corner of Figure 3. These are used during device programming as well as to allow communication between the Operating System and Application on the host computer and the FPGAs. It is also possible to use the LCBus to communicate between the software running on the host computer and the hardware implemented on the DEC PeRLe-1 board.

The central core is a 4x4 array of Xilinx 3090 FPGAs. The FPGAs can implement any kind of digital circuitry, taking into account gate counts and timing limitations. Each FPGA is connected to its four neighbors (North, East, South and West). Each FPGA has 16 data wires on each side. This can be seen in Figure 3 as the 4, 16-bit, buses on each side of the 4x4 array connecting the FPGAs to the four distributed memory banks.

The memory banks are connected to their sides of the FPGA processing array via a 64 bit data bus, under control of the associated Switch (SWN, SWE, SWS, and SWW).

The memory system consists of a total of 4MB high-speed static RAM. It is distributed equally on each side of the matrix array, with 1MB to the North, 1MB to the East, 1MB to the South and 1MB to the West. Each bank is independent of the others, with their own address, data and control buses.

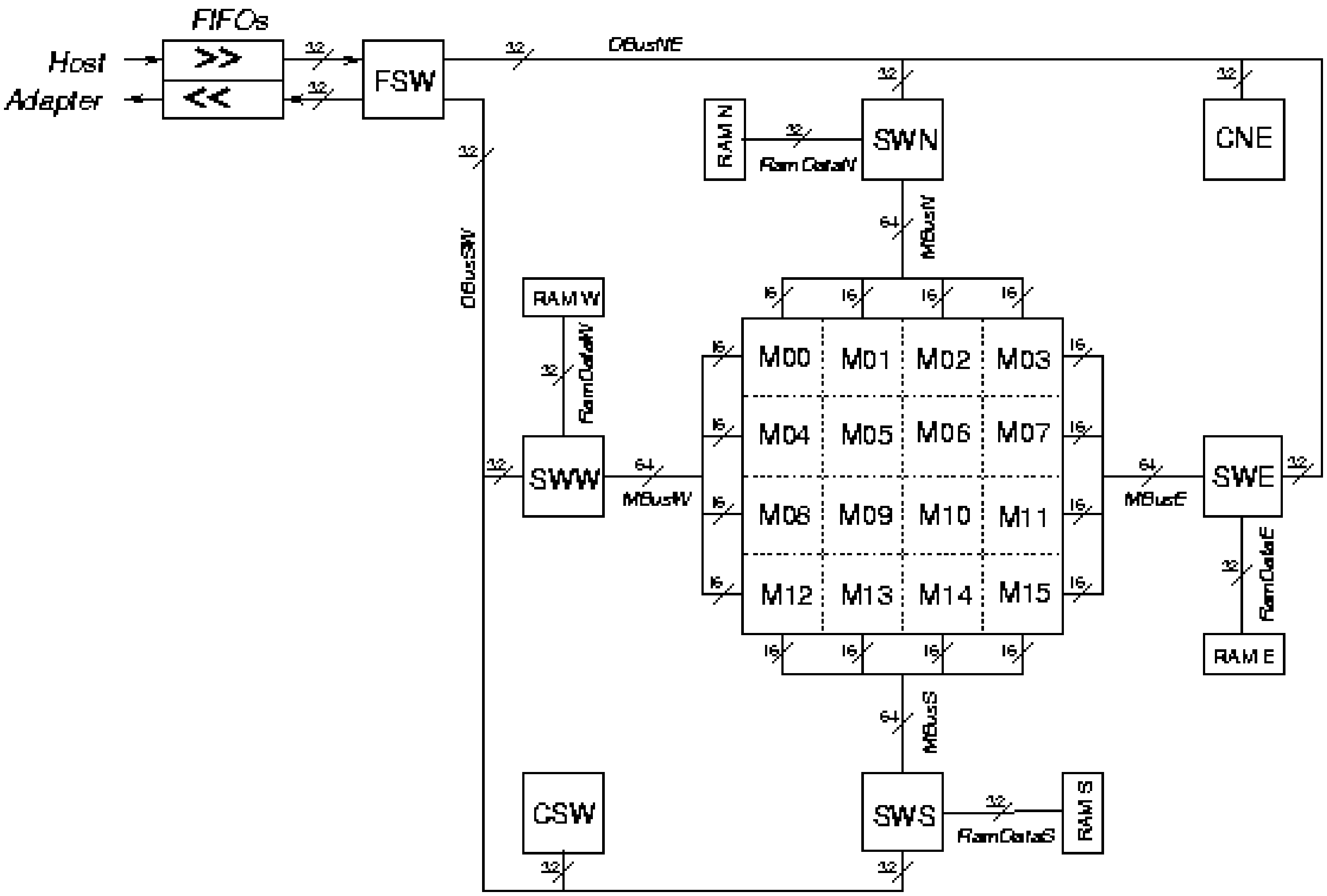

Figure 3: DEC PeRLe-1 architecture

Table 11: DEC PeRLe-1 board delays

Connection	Name	Delay
Matrix direct connection	DCnn-mmX	24 ns
Matrix bus I/O bus Switch ring Switch control FIFO Switch control	MBusX DbusXX RingSwXX SwCntrXX FswCntr	28 ns
Matrix ring	RingMat	43 ns
Matrix read, address to data	RamAddrX to RamDataX	46 ns
Matrix read, control to data	RamReadX to RamDataX	35 ns
Memory write, to any source		33 ns
FIFO data Tags	FIFOInData FIFOOutData TagAddr TagDataType TagDataSource	23 ns
FIFO Status	FIFOInEmpty FIFOOutFull	27 ns
FIFO Command	FIFOInRead FIFOOutWrite	30 ns

The two controllers (CNE and CNW) can be used to provide global control signals to the matrix FPGAs. There are 10 wires connecting the controllers and each FPGA. Also, because of loading issues of the connection 18 elements (16 FPGAs forming the array and the two control elements) the penalty for communication along this bus is higher than that of the local communication between two adjacent FPGAs. The 10 wires described above constitute the Matrix Rings as described in [2].

It will become important as the design process progresses to have an understanding of the timing characteristics of the different buses available for use in the DEC PeRLe-1 board. Table 11 shows the pertinent timing characteristics from Qihong.

PRSComp

Muraszkiewicz and Rybinski [3] present algorithms and a possible implementation of a machine to perform the fundamental Rough Set Theory calculations of upper approximation, lower approximation, reducts and cores. The name of their machine is the Parallel Rough Sets Computer or PRSComp.

Figure 4 shows the overall architecture of PRSComp.

Muraszkiewicz and Rybinski propose a machine consisting of $m * n$ primitive processors. An individual processor is connected to its four neighbors (north, east, south and west), as well as to global control signals. The processors execute in lock-step with one another under coordination of the global control signals. PRSComp operates as a SIMD (Single Instruction, Multiple Data) computer.

SIMD is one classification of a computer organization originally made by Flynn [6] [7]. Another classification is MIMD (Multiple Instruction Multiple Data). As can be observed by the names, both computer organizations operate on multiple data. This means that at any point in time, ideally, each processor is working with its own data item, thus the parallel nature. In a traditional von Neumann computer a single instruction operates on a single piece of data at any point in time. This can be classified as SISD of Single Instruction Single Data.

A question that arises when designing a parallel computer is that of the origin of instructions that the processors execute. Should the processors each run their own programs or should instructions come from a central source? In a SIMD parallel computer the instructions come from a single source and therefore each processor is executing the same instruction at the same time. In a MIMD parallel computer, each processor executes its own program and therefore each processor is completely independent of the instructions the other processors are executing.

In PRSComp the instructions are provided by a central resource: the global control signals to each processor (not shown in Figure 4). Each processor is connected to the global control signals and therefore each processor performs the same operation, based on the instruction, at the same moment in time. This explains the Single Instruction part of the SIMD classification of PRSComp.

The input data, which calculations are to be performed, is mapped into the $m * n$ processors as a binary matrix $A_{m,n}$, each processor takes on one element of the matrix. Note, that this means that a single processor operates on only a single bit at a time. Each processor operates on its own data that is independent of the other processors. This is the Multiple Data part of the SIMD term.

Three registers are utilized in the Rough Set Theory calculations: the column mask register (CM) (horizontal), the comperand register (C) (horizontal) and the word selection register (E) (vertical). These three registers, along with the global control signal direct the operation of each processor. The column mask register is used to inhibit the processing of matrix cells. The comperand register is used to transfer words to and from the processor array, as well as taking part in comparison operations. The word selection register contains the result of a comparison.

PRSComp Algorithms and Examples

Muraszkiewicz and Rybinski present six routines to compute fundamental notions of Rough Set Theory. The routines are: BasicCAT (Basic Category), UpperAPPROX (Upper Approximation), LowerAPPROX (Lower Approximation), Def (Definable), Indispensable and EXTCOMP (External Comparison). The algorithms are in Figure 4 for reference. The logical OR operation is rep-

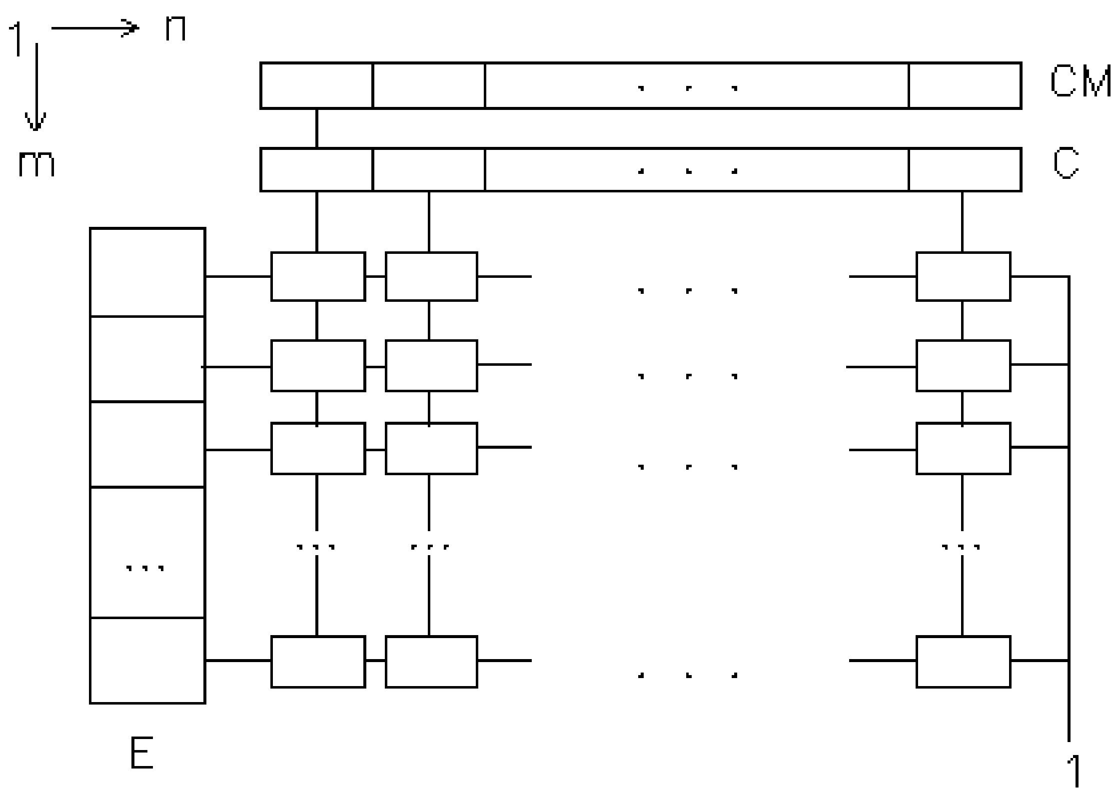

Figure 4: PRSComp Architecture

resented by "+". The logical AND shown operation is represented by "*".

The algorithms will be illustrated by an example. The example used is the same as the one used in describing the Rough Sets Theory.

Routine EXTCOMP ($A[m,n], C[n], E[m]$)
/* $C[n]$ is the comperand */
  $\overline{MASK}$
  $E[m] := 0$
  for $j = 0$ to n if $c[j] = a[i,j]$ then a couple is transparent
  labeled by "t", otherwise it is opaque labeled by "o"
  logic value of 1 is propagated through rows of the array, if
  all cells in the row are transparent then 1 is injected into
  that position in the E register

Routine BasicCAT (A[m,n], i, E[m]) /* Basic category */
/* The E register contains the characteristic vector that indicates
the words belonging to the basic category generated by $\underline{a}_i$ */
  $\overline{MASK}$
  $C[n] := \underline{a}_i$
  EXTCOMP($A[m,n], i, E[m]$)

Routine UpperAPPROX ($A[m,n], B[m], C[m]$) /* Upper approx. */
/* $B[m]$ stores the characteristic vector of the set X */
/* $C[m]$ the characteristic vector of the upper approximation of X */
  for $i = 1$ to m BasicCAT($A[m,n], i, E_i[m]$)
  for $i = 1$ to m $S_i[m] := E_i[m] * B[m]; C[m] := 0$
  for $i = 1$ to m if $S_i[m] \neq 0$ then $C[m] := C[m] + E_i[m]$

Routine LowerAPPROX ($A[m,n], B[m], C[m]$) /* Lower approx. */
/* $B[m]$ stores the characteristic vector of the set X */
/* $C[m]$ the characteristic vector of the lower approximation of X */
  for $i = 1$ to m BasicCAT($A[m,n], i, E_i[m]$)
  for $i = 1$ to m $S_i[m] := E_i[m] * B[m]; C[m] := 0$
  for $i = 1$ to m if $E_i[m] = S_i[m]$ then $C[m] := C[m] + E_i[m]$

Routine Def ($A[m,n], B[m]$) /* Definability */
/* $B[m]$ stores the characteristic vector of the set X */
  UpperAPPROX ($A[m,n], B[m], C[m]$)
  LowerAPPROX ($A[m,n], B[m], C'[m]$)
  if $C[m] = C'[m]$ then the set X is definable

Routine Indispensable ($A[m,n], j$) /* Indispensability */
/* j indicates the column number to be checked out */
  for $i = 1$ to m
  begin
    $\overline{MASK}$

BasicCAT ($A[m,n], i, B[m]$)
$(\overline{MASK} - \{j\})$
if $B[m] \neq C[m]$ then $\{j\}$ is indispensable
end

Figure 5 Algorithms presented by Muraszkiewicz and Rybinski

Another notion used in the PRSComp is the characteristic vector of a subset. If $X \subseteq U$ then we can say that the characteristic vector of X against $A(m,n)$ is:

$b_i(X) = 1$ for i such that $A(i,*) \subseteq X$
0 otherwise

$i = 1$ to m

m is equivalent to the number of decision rules (rows) in the input dataset. The characteristic vector $B[m]$ is thus a vertical vector of height m. If b_i is equal to 1 then the ith decision rule belongs to the set X.

The first step from Rough Set Theory was to determine which input variables were unnecessary (i.e. to find which input variables were indispensable). Previously, this was determined by removing the input variables one at a time and then determining whether or not the resulting table was consistent. This was shown in Table 2 through Table 6. The desire here is to investigate how this operation is performed in the PRSComp.

First note that each element of the original truth table shown in Table 1 is mapped to a single processor. The initial state of the PRSComp is shown in Table 12.

Please note that the U column is not stored anywhere within PRSComp but is merely shown in the table to aid in discussion. Initially the contents of the CM (column mask), C (comperand) and E (word selection) register values are undefined. Each box in Table 12 signifies one primitive processor within PRSComp.

Now to use the Indispensable routine, shown in Figure 5, to determine which input variables can be thrown out without adversely affecting the results. The for loop within the Indispensable routine executes its contents once for every row in Table 12.

The first step to the Indispensable routine is to set up the CM register, this is shown in the Indispensable routine by $\overline{MASK}$. In this instance the CM register is initialized to all 1s meaning that no processors are masked.

Next the BasicCAT routine is called. The BasicCAT routine sets up the CM register again with all 1s. The statement C[n] := a_i in the BasicCAT routine means that the current row is copied to the comperand register (C).

The EXTCOMP routine is then called and sets up the CM register again. The E register is then initialized to all 0s, E[m] := 0. The current state of the machine is shown in Table 13.

Table 12: Initial State of PRSComp

	CM						
	C						
E	U	a	b	c	d	e	f
	1	0	0	0	1	1	1
	2	0	1	0	0	0	1
	3	0	1	1	0	0	1
	4	1	1	0	0	0	1
	5	1	1	0	1	0	1
	6	1	1	1	1	0	1
	7	1	1	1	0	0	1
	8	1	0	0	1	1	1
	9	1	0	0	1	0	1
	10	0	0	0	0	0	0
	11	0	0	0	1	0	0
	12	0	1	1	1	0	0
	13	0	1	1	0	1	0
	14	1	1	1	1	1	0
	15	1	0	0	0	1	0

Table 13: Intermediate State of PRSComp

	CM	1	1	1	1	1	1
	C	0	0	0	1	1	1
E	U	a	b	c	d	e	f
0	1	0	0	0	1	1	1
0	2	0	1	0	0	0	1
0	3	0	1	1	0	0	1
0	4	1	1	0	0	0	1
0	5	1	1	0	1	0	1
0	6	1	1	1	1	0	1
0	7	1	1	1	0	0	1
0	8	1	0	0	1	1	1
0	9	1	0	0	1	0	1
0	10	0	0	0	0	0	0
0	11	0	0	0	1	0	0
0	12	0	1	1	1	0	0
0	13	0	1	1	0	1	0
0	14	1	1	1	1	1	0
0	15	1	0	0	0	1	0

Table 14: After first execution of EXTCOMP routine

	CM	1	1	1	1	1	1
	C	0	0	0	1	1	1
E	U	a	b	c	d	e	f
1	1	0 t	0 t	0 t	1 t	1 t	1 t
0	2	0 t	1 o	0 t	0 o	0 o	1 t
0	3	0 t	1 o	1 o	0 o	0 o	1 t
0	4	1 o	1 o	0 t	0 o	0 o	1 t
0	5	1 o	1 o	0 t	1 t	0 o	1 t
0	6	1 o	1 o	1 o	1 t	0 o	1 t
0	7	1 o	1 o	1 o	0 o	0 o	1 t
0	8	1 o	0 t	0 t	1 t	1 t	1 t
0	9	1 o	0 t	0 t	1 t	0 o	1 t
0	10	0 t	0 t	0 t	0 o	0 o	0 o
0	11	0 t	0 t	0 t	1 t	0 o	0 o
0	12	0 t	1 o	1 o	1 t	0 o	0 o
0	13	0 t	1 o	1 o	0 o	1 t	0 o
0	14	1 o	1 o	1 o	1 t	1 t	0 o
0	15	1 o	0 t	0 t	0 o	1 t	0 o

The remainder of the EXTCOMP routine is executed and the resulting state of the PRSComp is shown in Table 14.

It is interesting to note that for a particular row the set of all cells in a particular column that are labeled as transparent is identical to what was used above with the Rough Set example above as the set of decision rules having the same value for a particular variable. More specifically, in Table 14 , under the column for input variable a the row numbers that are labeled with a t are 1, 2, 3, 10, 11, 12 and 13. From the Rough Set Theory notation above, this is really $[1]_a = \{1, 2, 10, 11, 12, 13\}$. This is the case for every column.

The current E register is then saved somewhere and the position in the CM register corresponding to the column being checked, let's say the column for variable a is set to 0 . The EXTCOMP routine is called again and Table 15 shows the results after execution of the EXTCOMP routine.

The new E register is compared with the previously stored E register and they are found to not be equivalent. This means that input variable a is indispensable (i.e. it must be kept in the final solution).

It is also interesting to note that masking off input variable a is analogous to removing input variable a and checking the intersection.

$\bigcap(F - [1]_a) = [1]_b \cap [1]_d \cap [1]_e = \{1, 8, 9, 10, 11, 15\} \cap \{1, 5, 8, 9, 11, 12, 14\} \cap \{1, 8, 13, 14, 15\} = \{1, 8\}$

Also note that the result of the intersection is 1 propagated through the processors in the final E register, by way of the 1 wired to the right hand side of Figure fig:prscomparch. If a processor is marked as transparent then the 1

Table 15: Result after second call to EXTCOMP

	CM	0	1	1	1	1	1
	C	0	0	0	1	1	1
E	U	a	b	c	d	e	f
1	1	0 t	0 t	0 t	1 t	1 t	1 t
0	2	0 t	1 o	0 t	0 o	0 o	1 t
0	3	0 t	1 o	1 o	0 o	0 o	1 t
0	4	1 t	1 o	0 t	0 o	0 o	1 t
0	5	1 t	1 o	0 t	1 t	0 o	1 t
0	6	1 t	1 o	1 o	1 t	0 o	1 t
0	7	1 t	1 o	1 o	0 o	0 o	1 t
1	8	1 t	0 t	0 t	1 t	1 t	1 t
0	9	1 t	0 t	0 t	1 t	0 o	1 t
0	10	0 t	0 t	0 t	0 o	0 o	0 o
0	11	0 t	0 t	0 t	1 t	0 o	0 o
0	12	0 t	1 o	1 o	1 t	0 o	0 o
0	13	0 t	1 o	1 o	0 o	1 t	0 o
0	14	1 t	1 o	1 o	1 t	1 t	0 o
0	15	1 t	0 t	0 t	0 o	1 t	0 o

is propagated to the left. If a processor is marked as opaque then the 1 is not propagated.

In the hardware, all comparisons are done in parallel for every row. The E register is stored somehow. Then a column is masked and the comparisons occur once again with the exception that the masked column stays transparent. This new E register is compared to the old E register and if they are equivalent then the current input variable (column) is dispensable and can therefore be removed.

First the LowerAPPROX routine is executed.

For this example, we wish to find the lower approximation with respect to the set of all decision rules with an output value of 1.

For this let us continue to use Table 10 as the input dataset. From above, we should expect to find the lower approximation to be:

$POS_R(X) = \underline{R}X = \{1, 6, 7, 10, 11, 12\}$

The input data set is first loaded into the PRSComp machine, this is depicted in Table 16.

First we must calculate the characteristic vector of our set X. This, along with the dataset is input from the user. Remember that the characteristic vector is defined as:

$b_i(X) = 1$ for i such that $A(i,*) \subseteq X$
0 otherwise

Table 16: Initial State of PRSComp for LowerAPPROX routine

E	CM C U	a	b	c	d	f
	1	0	0	0	0	1
	2	0	0	1	0	0
	3	0	0	1	0	1
	4	0	0	1	1	0
	5	0	0	1	1	1
	6	0	1	0	0	1
	7	0	1	0	1	1
	8	0	1	1	0	0
	9	0	1	1	1	0
	10	1	0	0	1	1
	11	1	0	1	0	1
	12	1	0	1	1	1
	13	1	1	0	1	0
	14	1	1	1	0	0
	15	1	1	1	1	0

$i = 1$ to m

We want X to be the set of all decision rules that have an output value f of 1. Whenever f_i is a 1 we want b_i to be 1, everywhere else b_i should be 0. Here the vector is shown as horizontal, but it should be visualized as a vertical vector, because its length is equivalent to the height of the input dataset, in this case the number of decision rules in Table 16.

$B[m] = \{1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0\}$

Now to execute the LowerAPPROX routine. First to calculate the basic categories for each decision rule in Table 16. This is performed by the first line in the LowerAPPROX routine:

for $i = 1$ to m BasicCAT($A[m, n], i, E_i[m]$)

It should be noted that each time the BasicCAT routine is executed a new E_i is generated, so upon completion of this line of code there will be m E vectors denoted as $E_1, ..., E_m$. The BasicCAT routine is the same as that described in calculating the core values above. The BasicCAT routine generates an E_i vector that has 1s where the input variables are equivalent to the ith decision rule. Note that the dependent (output) variable is masked off via the $\overline{MASK}$ operation.

The values calculated for the E_i registers are shown in Table 17. For simplicity, when showing the E_i vertical vectors in the table 0 values are omitted.

Table 17: E_i registers

E_{15}	E_{14}	E_{13}	E_{12}	E_{11}	E_{10}	E_9	E_8	E_7	E_6	E_5	E_4	E_3	E_2	E_1
														1
												1	1	
												1	1	
										1	1			
										1	1			
									1					
								1						
							1							
						1								
					1									
				1										
			1											
		1												
	1													
1														

Next each $S_i[m]$ is calculated. This is the logical AND of each E_i vector against the characteristic vector of X, $B[m]$. Note again this creates m separate vertical vectors $S_1, \cdots, S_m$.

for $i = 1$ to $m S_i[m] := E_i[m] * B[m]; C[m] := 0$

The positional sums created are shown in Table 18.

Finally, the final loop is executed calculating the characteristic vector of the lower approximation.

for $i = 1$ to m if $E_i[m] = S_i[m]$ then $C[m] := C[m] + E_i[m]$

This says that if the E_i vertical vector is equal to the S_i vector in every position then take the logical OR of the characteristic vector of the lower approximation, $C[m]$ and the $E_i[m]$ vector. If the resulting vector (S_i) is equivalent to (E_i) then this means that for every decision rule that has the same input values they all agree that they are always included in the set X, thus all of these decision rules belong to the lower approximation of X, therefore OR them into $C[m]$ (lower approximation of X). If the resulting vector (S_i) is <u>not</u> equivalent to (E_i) then this means that <u>at least one</u> of the decision rules that have the same values for inputs <u>does not</u> belong to X, therefore do not OR them into the $C[m]$. This generates $C[m]$ of:

$C[m] = \{1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0\}$

From the positive region, also known as the lower approximation for Table 10 we saw that:

Table 18: Positional Sums

S_{15}	S_{14}	S_{13}	S_{12}	S_{11}	S_{10}	S_9	S_8	S_7	S_6	S_5	S_4	S_3	S_2	S_1
0	0	0	0	0	0	0	0	0	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	1	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	1	1	0	0	0
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
0	0	0	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	1	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	0	0	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	0	0	0	0	0	0	0	0

$POS_R(X) = \underline{R}X = \{1, 6, 7, 10, 11, 12\}$

From the value found for the positive region, we would expect a 1 in positions 1, 6, 7, 10, 11 and 12 of the characteristic vector for the lower approximation. Indeed, this is the case.

Now to move on to the UpperAPPROX routine.

First, note that the first two loops are identical for LowerAPPROX and UpperAPPROX. This fact can be used to make the calculation of the UpperAPPROX faster. E_i and S_i have already been calculated so there is no need to calculate them again.

Now picking up on the final loop of the UpperAPPROX routine.

for $i = 1$ to m if $S_i[m] \neq 0$ then $C[m] := C[m] + E_i[m]$

This means that if the resulting vector (S_i) is not empty (i.e. S_i does not have all zero's) then at least one of the decision rules that have the same input values belong to the set X, therefore all of the decision rules that have the same values for input variables belong to the Upper Approximation so OR the entire E_i vectors into $C[m]$.

If the resulting vector is empty (i.e. all of the S_i values are zero) then none of the decision rules that have the same input values belong to the set X, therefore do not OR the E_i vector into $C[m]$.

Following execution of this loop the characteristic vector of the Upper Approximation has been calculated.

$C[m] = \{1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0\}$

Previously, the upper approximation for Table 10 was found to be:

$$\overline{R}X = E_1 \cup E_2 \cup E_3 \cup E_4 \cup E_5 \cup E_8 \cup E_9 \cup E_10 = \{1, 2, 3, 4, 5, 6, 7, 10, 11, 12\}$$

From this value we would expect to have a 1 in positions 1, 2, 3, 4, 5, 6, 7, 10, 11, and 12. Our solution matches what was expected.

Design of the Rough Set Machine

Tools Used

The Behavioral code was simulated using Mentor Graphics available at Portland State University.

The Synthesizable code was synthesized to Xilinx XC3000 Family Device XC090ACB164 Speed Grade -6 FPGAs using Xilinx Foundation F1.5, Build 3.1.135, Student Edition. This tool is available from Xilinx. It should be verified that this is the same XC3090 FPGA that is on the DEC Perle-1 board.

Behavioral Rough Set Routines

Appendix B shows the VHDL code for the behavioral model of the Rough Set algorithms.

The routines presented by Muraszkiewicz and Rybinski to compute Rough Set Theory is coded behaviorally and simulated using Mentor Graphics available at Portland State University.

The input $A[m, n]$ matrix (set of decision rules) m decision rules and has n variables.

The Indispensable routine is first executed to mask the indispensable variables and reduce the matrix. This is done by Routine Ext. Comp and Basic Cat. And the result is stored in result matrix.

Again the Basic Cat. And Ext. Comp is performed but by masking the output variable and vector Ei[m] and Si[m] is obtained. The Lower Approximation and Upper Approximation routines are then executed. Table 16 is considered as an example. The decision rules in Table 16.

$B[m] = 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0$

Now first the Indispensable routine is executed and Result (2) matrix is generated. Then to execute Lower Approx. again Basic Cat is performed and $Ei[m]$ and $Si[m]$ is generated for each decision rule. $C[m] := 0$ The lower Approx. is then generated by

For $i = 1$ to m, if $Ei[m] = Si[m]$ then $C[m] := C[m] + Ei[m]$

To execute Upper Approx. the Ei[m] and Si[m] is reused and C[m] is generated using

For $i = 1$ to m, if $Si[m] = /0$ then $C[m] := C[m] + Ei[m]$

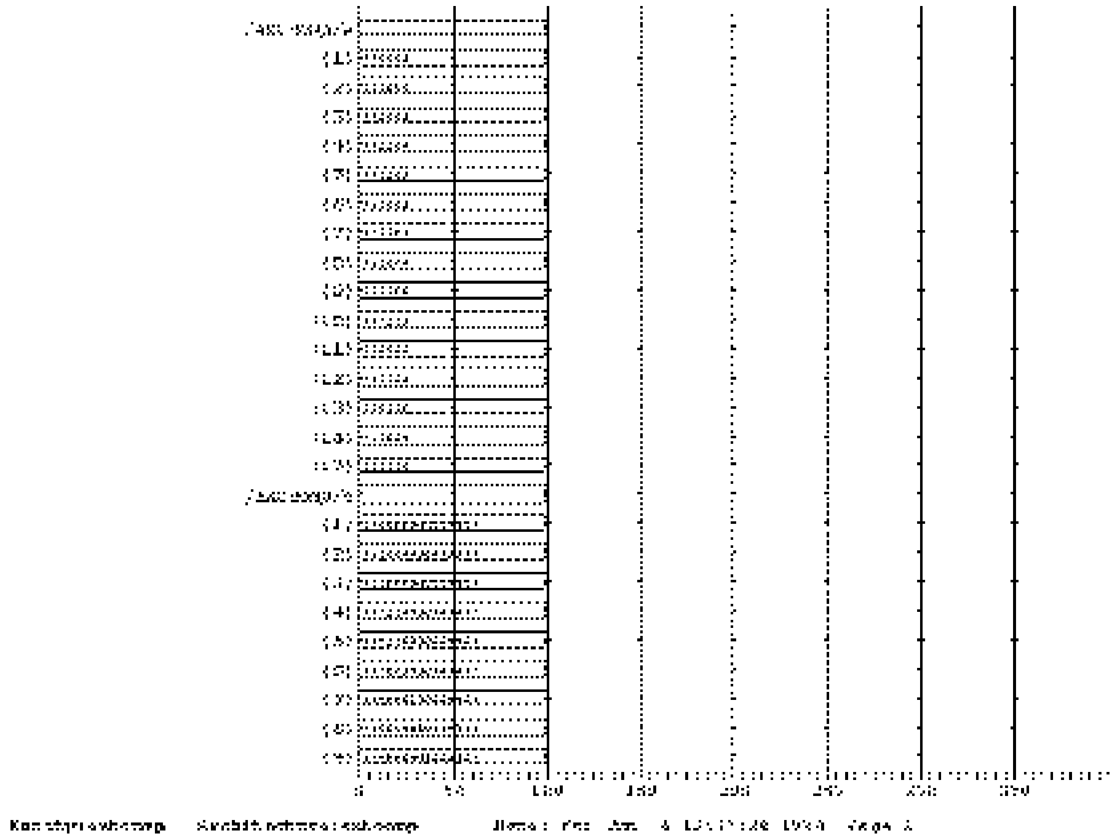

Figure 5: Simulation Waveforms for Behavioral VHDL Code

Overall Architecture

As we have seen, the Rough Set operations lend themselves nicely to a processor capable of performing simple operations in parallel. This section will give an overview of the current implementation of a machine to perform the operations of Rough Set Theory.

To simplify this stage of the design, only the LowerAPPROX, UpperAPPROX and EXTCOMP are implemented.

The major goals of our design are as follows:

- Take advantage of the similarities between the LowerAPPROX and UpperAPPROX routines.

- Make the processor general wherever possible. Do not only implement the Rough Set operations, but give software the ability to take advantage of the hardware wherever possible.

One way the goal of generalization was realized was in observing that the LowerAPPROX and UpperAPPROX routines are almost identical, with the exception of the low level operations performed on the registers. Specifically, the LowerAPPROX and UpperAPPROX routines differ in the final comparison operation on the $E_i[m]$ and $S_i[m]$ vertical vectors. The UpperAPPROX routine compares $S_i[m]$ to 0. The LowerAPPROX routine compares the equality of $E_i[m]$ and $S_i[m]$. Because of the similarities, this operation is a good candidate for generalization. It would be ideal if in some way, the software could give the hardware two pieces of information: instruct the hardware to perform the three loops of a generalized LowerAPPROX/UpperAPPROX routine, and instruct the hardware what comparison operation to perform in the final loop, then the hardware would be simpler and also give the programmer the flexibility to use different comparison operations than that used specifically in the LowerAPPROX routine or UpperAPPROX routine. This gives the programmer the ability to create a new routine, not previously thought of. This Lower/Upper APPROX routine could be defined as follows:

Routine Lower/UpperAPPROX ($A[m,n], B[m], C[m]$)
/* $B[m]$ stores the characteristic vector of the set X */
/* $C[m]$ the characteristic vector of the lower approximation of X */
  for $i = 1$ to m BasicCAT($A[m,n], i, E_i[m]$)
  for $i = 1$ to m $S_i[m] := E_i[m] * B[m]; C[m] := 0$
  for $i = 1$ to m if comp($E_i[m], S_i[m]$) then $C[m] := C[m] + E_i[m]$

To take the generalization idea one step further, our design gives the programmer the opportunity to change the operator in the second loop from an AND operation to something else. Likewise the programmer is given the opportunity to change the OR operator in the final loop. The implemented generalized routine follows:

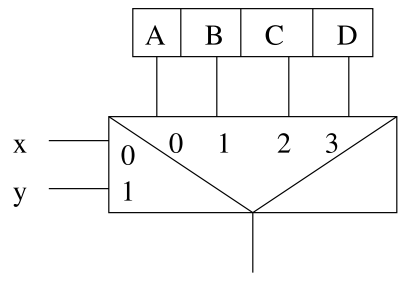

Figure 6: Hardware used to realize general operations

Table 19: Operation of Mux

y	x	F
0	0	A
0	1	B
1	0	C
1	1	D

Routine GeneralAPPROX ($A[m,n], B[m], C[m]$)
/* $B[m]$ stores the characteristic vector of the set X */
/* $C[m]$ the characteristic vector of the lower approximation of X */
for $i = 1$ to m BasicCAT($A[m,n], i, E_i[m]$)
for $i = 1$ to m $S_i[m] := E_i[m] op1 B[m]; C[m] := 0$
for $i = 1$ to m if comp($E_i[m], S_i[m]$) then $C[m] := C[m] op2 E_i[m]$

op1, op2 and the comp() function are implemented in hardware using a lookup table type approach that allows the programmer to perform up to 2 to the 4 different logic operations. As shown in Figure 6 a single bit of each the operands act as the mux selects (x, y). The user programmable register at the top of the figure are the inputs into the mux (A, B, C D). The output (F) is the result of the operation.

As with any mux, the operation of this is described by Table 19.

For example, in order to perform the AND operation on inputs x and y the user would load A=0, B=0, C=1 and D=1.

As we have seen when performing logic minimization, it is useful for the user to provide a MASK register, then using that MASK register call the GeneralAPPROX routine. The programmer is given the ability to load the MASK register directly through the use of a Load instruction.

The current implementation does not contain the Indispensible algorithm. By giving the programmer the ability to load the MASK register directly, the Indispensible routine could be implemented in hardware to generate the proper starting MASK for the GeneralAPPROX routine.

Mapping to DEC Perle Board

The architecture presented by Muraszkiewicz and Rybinski lends itself nicely to the DEC Perle board. The majority of their architecture remains the same in our design. Some modifications have been made in an effort to simplify the design, while at the same time improving the performance of the machine.

As with PRSComp, our machine maps a single bit from the data array to a single processor. Multiple processors may exist in each matrix FPGA, but the regularity on a processor by processor basis remains.

One modification involves the BasicCAT routine. The first operation that is always performed by the Muraszkiewicz and Rybinski BasicCAT routine is copying the row to be compared to a comperand register and then propagating this row to all processors. In our design, instead of spending a clock cycle to copy the row, the row to be compared against is selected and that row outputs its data contents onto the vertical buses connecting all processors in a given column.

The second modification made in our design involves the number of vertical registers. If the data array currently being worked with has 15 decision rules then 15 E registers and 15 S registers will be generated. Because these registers are used frequently in the Rough Set routines it makes sense to build these into the matrix, rather than continually swapping vertical registers to/from memory throughout operation.

Instructions and Horizontal data are stored in the North memory bank. Vertical data is stored in the East memory bank.

Instruction Set

The instruction encoding has been designed to be straightforward, yet flexible enough to be expanded upon in the future while maintaining backward compatibility. It was also important to design the instruction set with the architecture of the DEC Perle-1 board in mind.

The ability for an instruction to be transmitted to every FPGA simultaneously, was important in this design. Each FPGA is connected to the North memory bank where instructions are stored through 16 wires. This led to a maximum instruction width of 16 bits. In order to simplify the programmer's job we wanted to use a fixed instruction width that was as orthogonal as possible.

There are two subsets of instructions: memory operation instructions and matrix instructions. Bit 15 of an instruction signifies an instruction as being of the matrix type (bit 15=0) or of the memory operation type (bit 15=1). Figure 7 shows the general format of a matrix instruction. Note: This may not be what is being used in the matrix processors. Figure 8 shows the general format of a memory instruction.

Addressing the vertical and horizontal registers for memory operations is shown in Table 20. The addressing scheme allows up to 63 decision rules (rows). When loading or storing a vertical register (as specified by the instruction) address bit 6 acts as a select between the E_i and S_i registers.

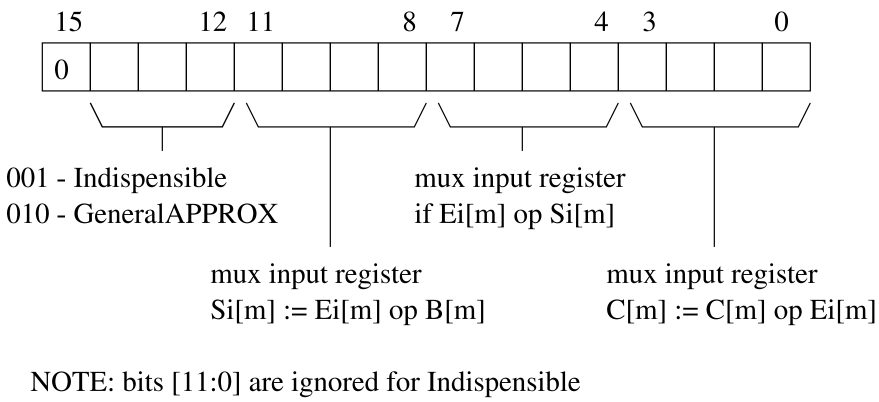

Figure 7: Matrix Instruction Format

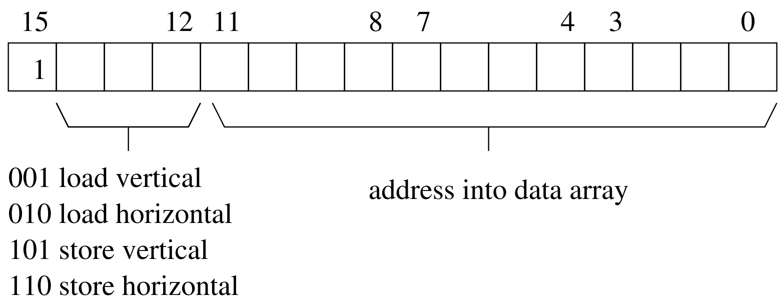

Figure 8: Memory Instruction Format

Table 20: Operation of Mux

Address [5:0]	Vertical Register bit 6 = 0	Vertical Register bit 6 = 1	Horizontal Register
0	$B[m]$	$C[m]$	$MASK$
1	E_1	S_1	row 1
2	E_2	S_2	row 2
...	...	...	
63	E_63	S_63	row 63

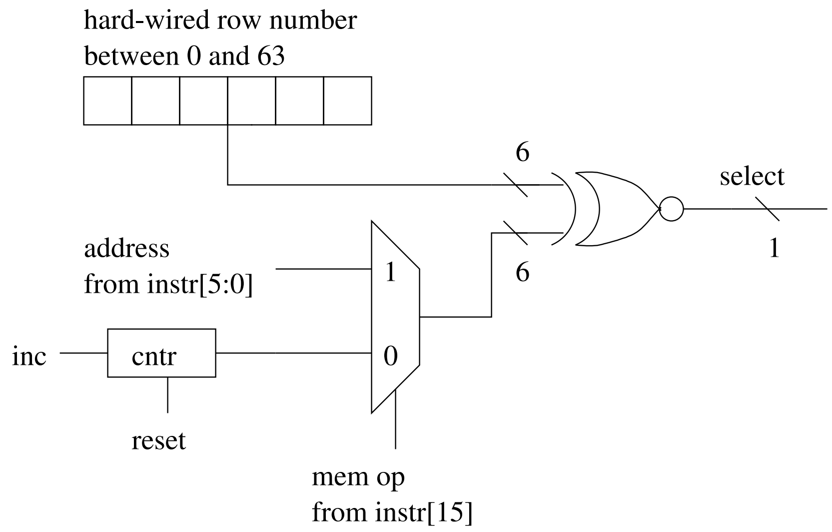

Figure 9: Distributed Address Select Logic

The address select logic is distributed, meaning that each row receives the address from the lower 7 bits of an matrix operation instruction via MBusN and each row decides whether or not it is the row selected. This logic is shown in Figure 9.

Memory Interface

The VHDL code for the memory interface is shown in Appendix C. The test script used to verify the memory interface and generate the waveforms presented is shown in Appendix D.

The memory interface logic is entirely contained in the controller CNE and switches SWN and SWE FPGAs. A number of signals connect the memory interface logic and the matrix. These connections are done via the Matrix ring. These signals are: INST_REQ, INST_RDY, WDATA_RDY and RDATA_RDY.

In order to simplify the design, a relatively slow clock period is used. The worst case access time to memory is 46 ns as shown in Table 11. The state machine used to access the memory assumes that data is valid within one clock period, so this implies that the fastest clock that can be used has a period of 46 ns + the delay through any combinational logic.

In order to keep the three FPGAs in sequence with each other without using extra signals between these FPGAs, each contain identical state machines. This state machine is described in the VHDL file cntl_sw_sm.vhd. The state diagram for this state machine is shown in Figure 10.

Figure 11 shows a Load Horizontal Instruction. First, the matrix requests an instruction through the INST_REQ signal. The instruction is fetched from memory by placing the instruction address on RAM_ADDRN and as-

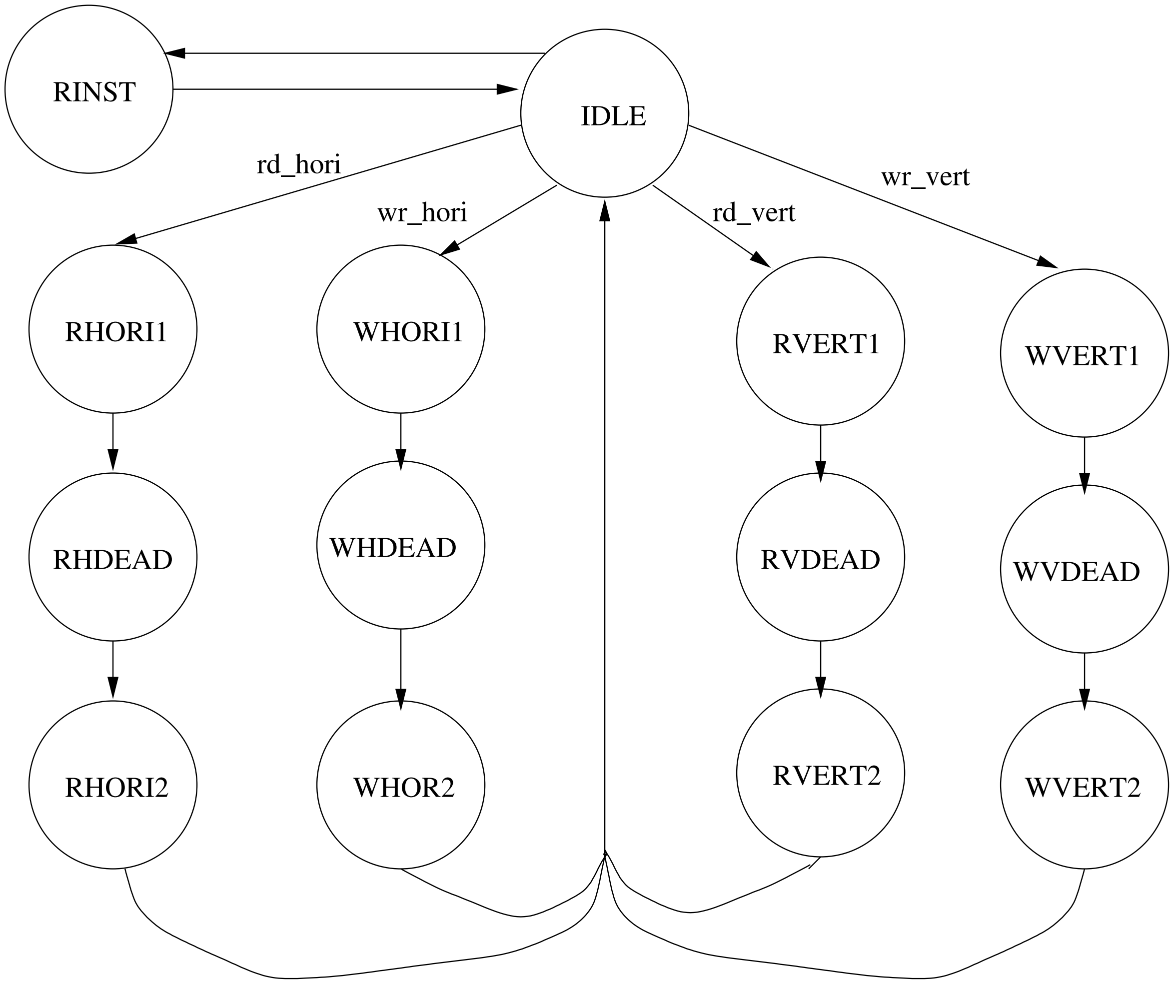

Figure 10: Memory Interface State Machine

serts RAM_READN_N. The _N at the end of a signal name designates it as an active low signal. Because instructions are only 16 bits, only 16 of the 32 bits returned from memory will be used at a time. To facilitate debugging RAM_DIS_HIGHN_N and RAM_DIS_LOWN_N are utilized. In this case 'A000' (Load Horizontal) is the instruction fetched. The address portion of the instruction is not used by the memory interface, it is used by the address selection logic within the matrix. The memory interface then pulses RAM_READN_N low twice to read a total of 64 bits from memory. The memory interface logic combines the two 32 bit words read from memory into one 64 bit quantity and presents it to the matrix along with RDATA_RDY.

Figure 12 shows a Load Vertical Instruction. The sequence of events is identical to a Load Horizontal Instruction, except that data is read from the East memory bank rather than the North.

Figure 13 shows a Store Horizontal Instruction. As before the matrix requests an instruction through the INST_REQ signal. The matrix then places the write data onto the North bank memory bus and pulses WDATA_RDY. The memory interface then cycles the data to memory, asserting RAM_WRITEN_N.

Figure 13 shows a Store Vertical Instruction. The sequence of events is identical to a Store Horizontal Instruction, except that data is read from the East memory bank rather than the North.

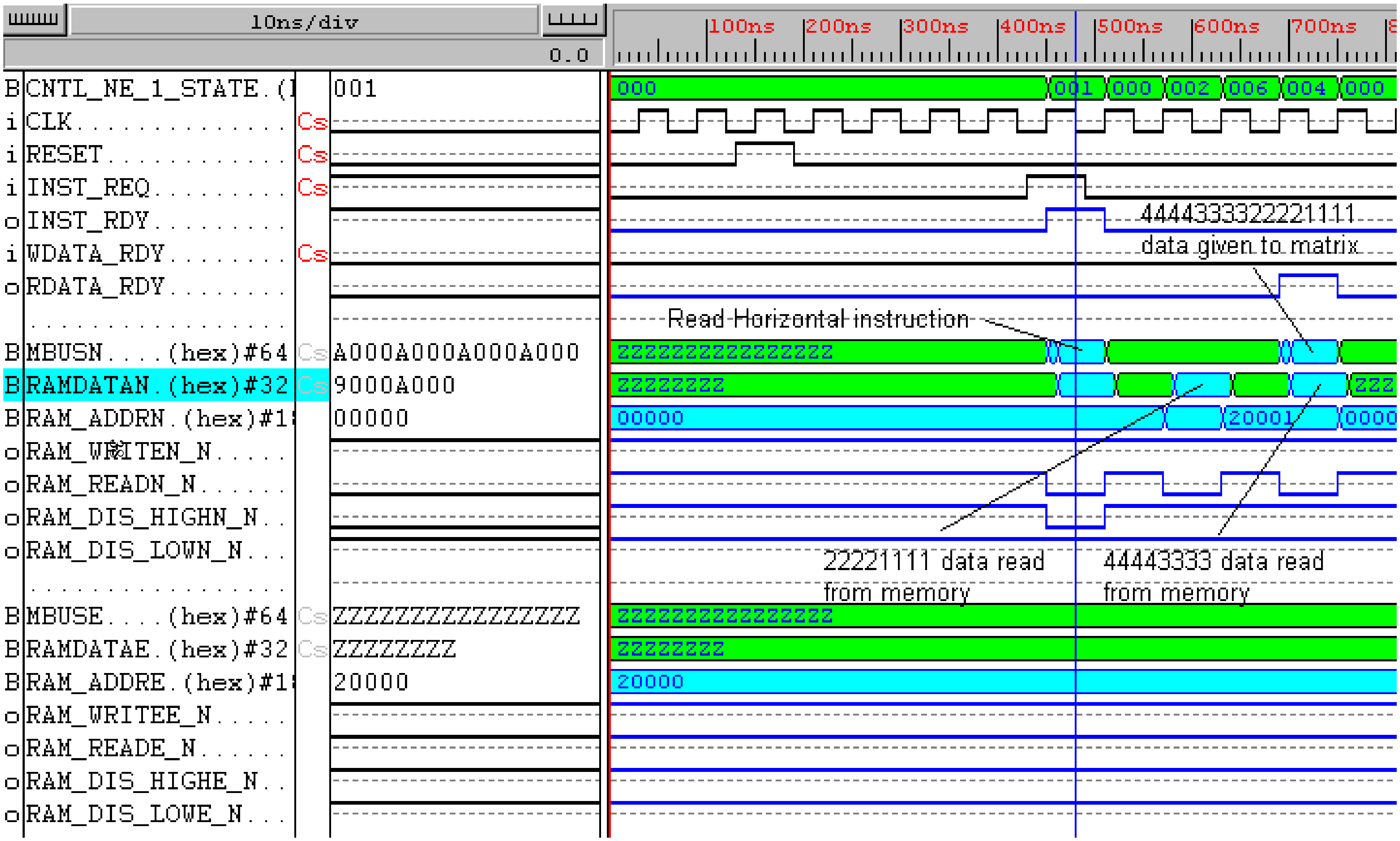

Figure 11: Load Horizontal Instruction

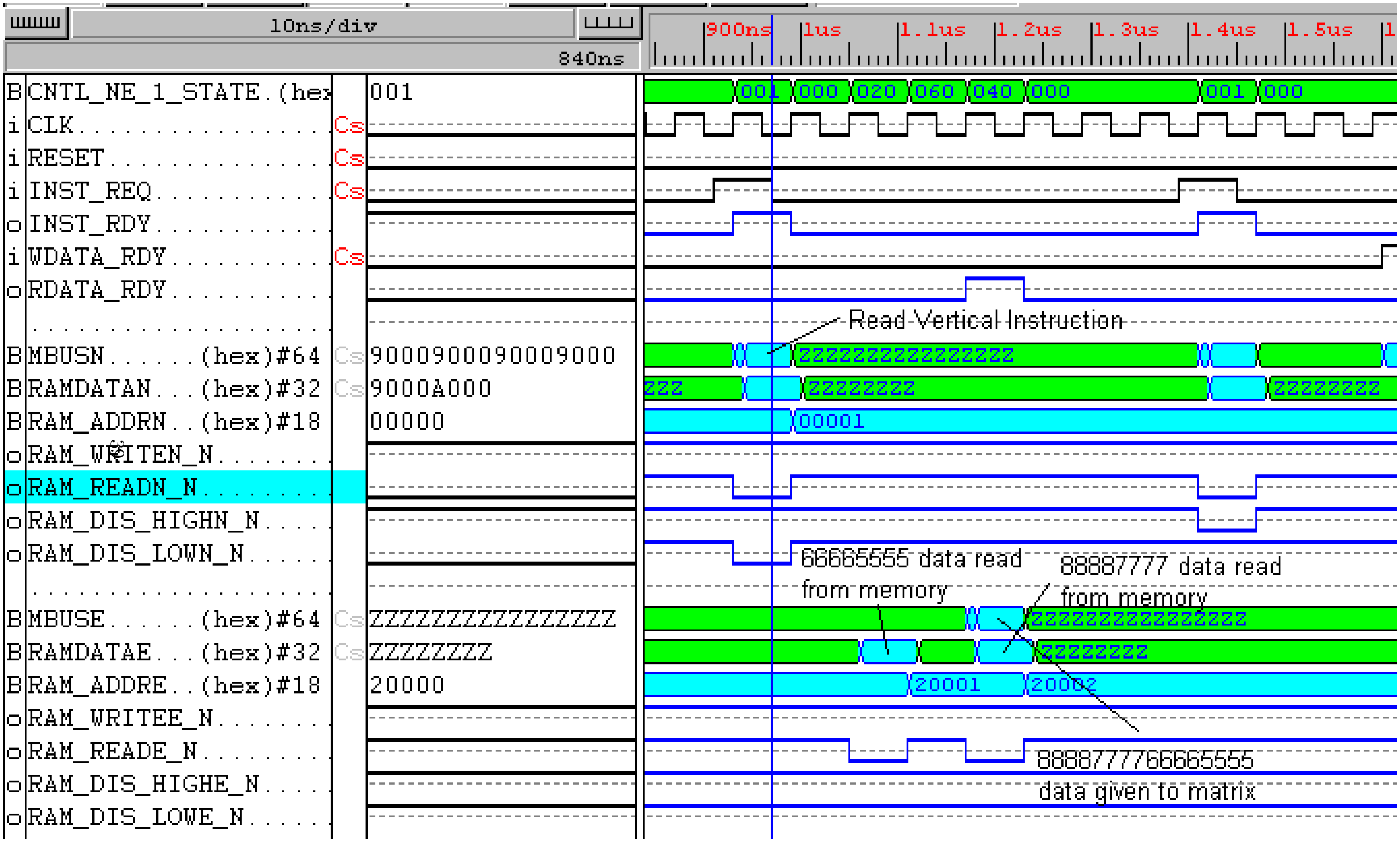

Figure 12: Load Vertical Instruction

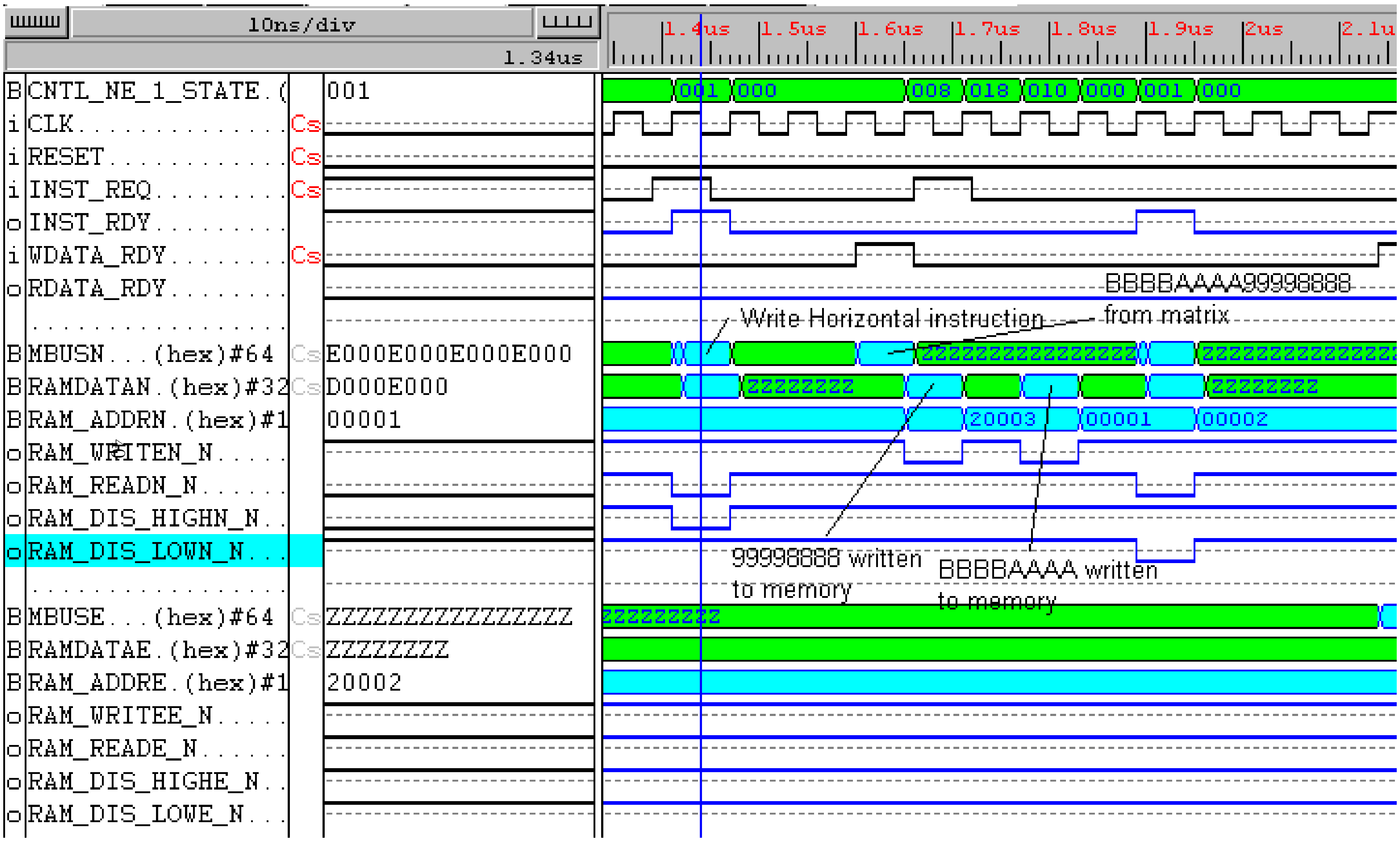

Figure 13: Store Horizontal Instruction

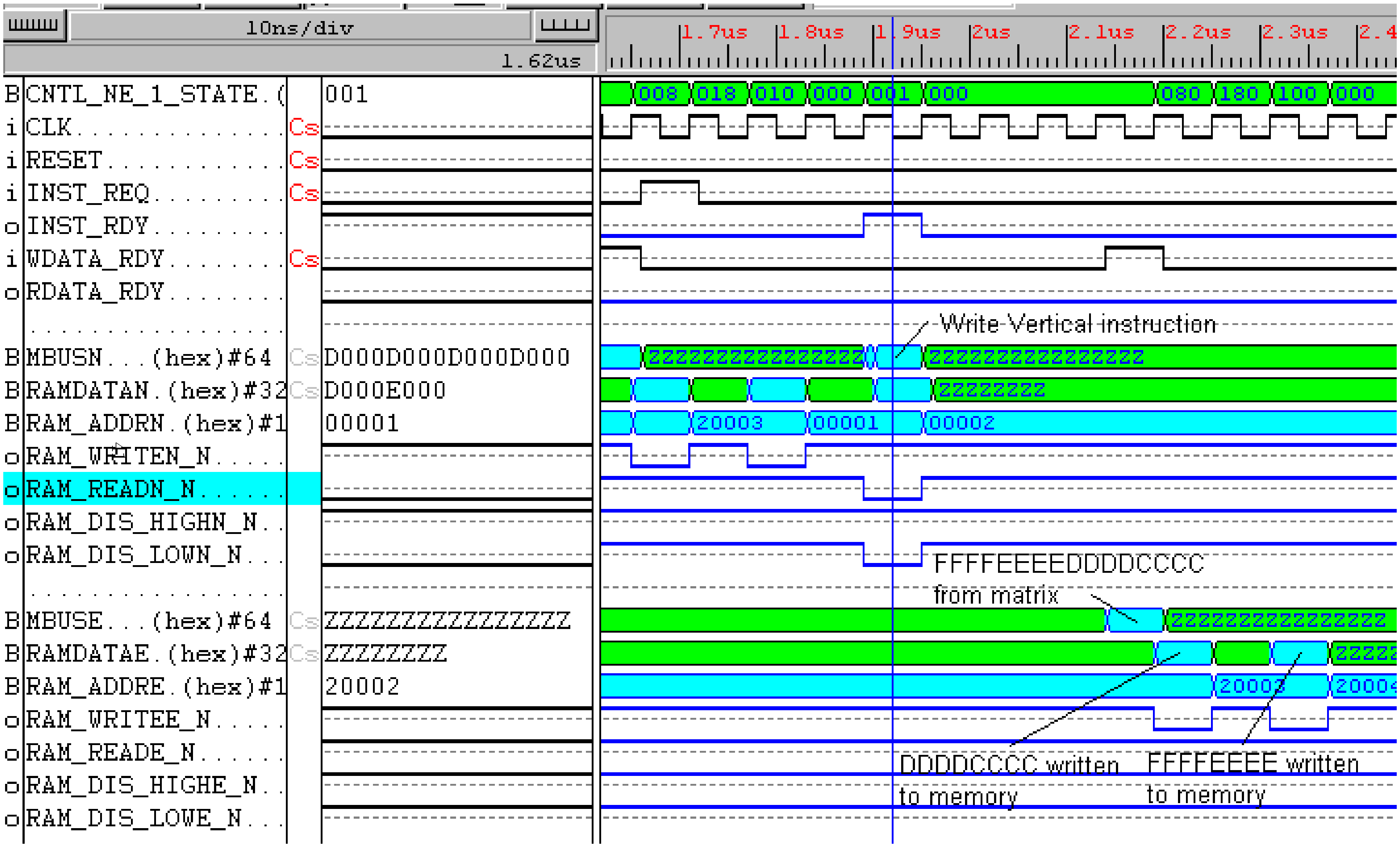

Figure 14: Store Vertical Instruction

Matrix

Simulation results for the address select logic is shown in Figure 15.

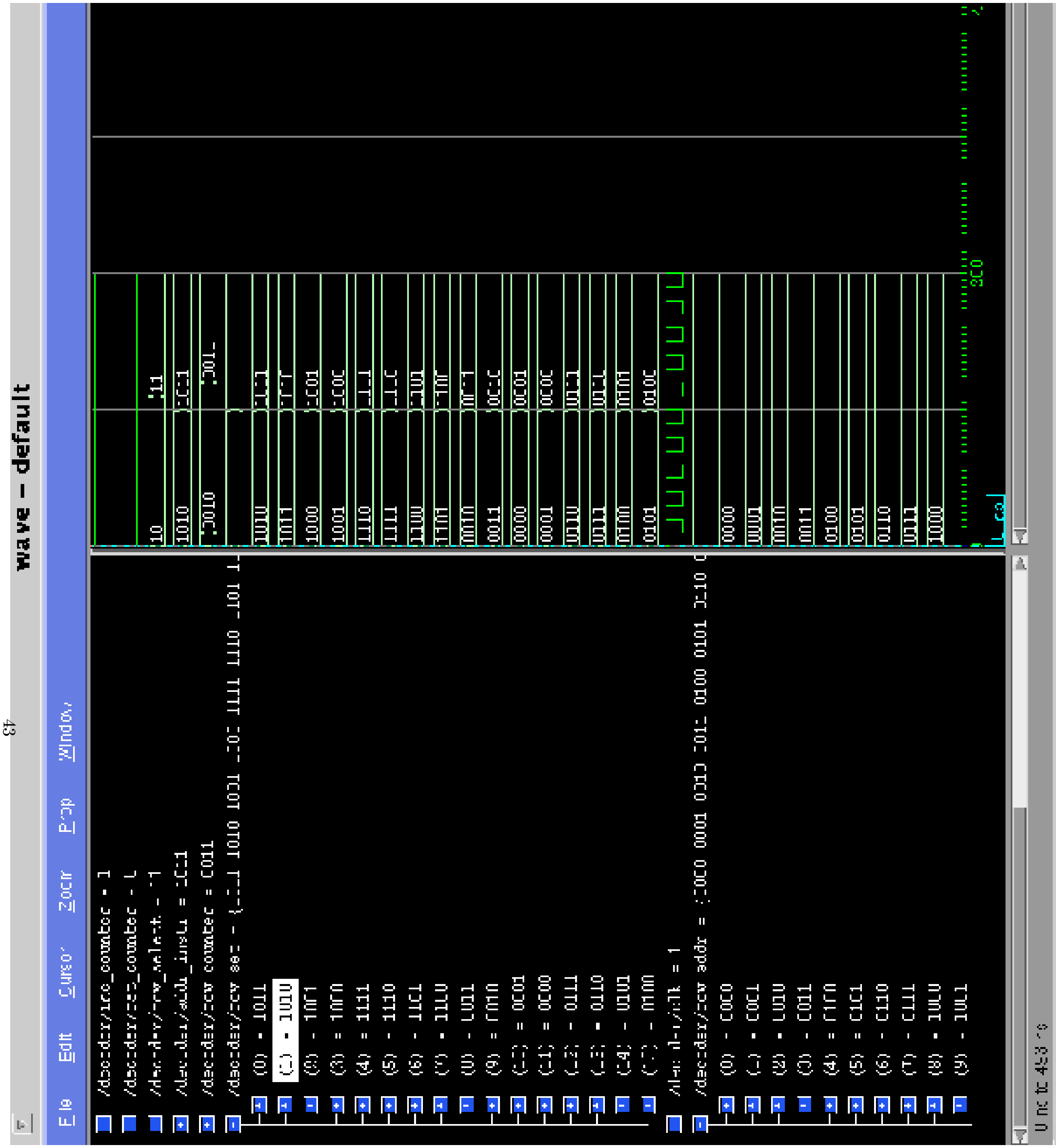

Figure 15: Simulation Results For Address Select Logic

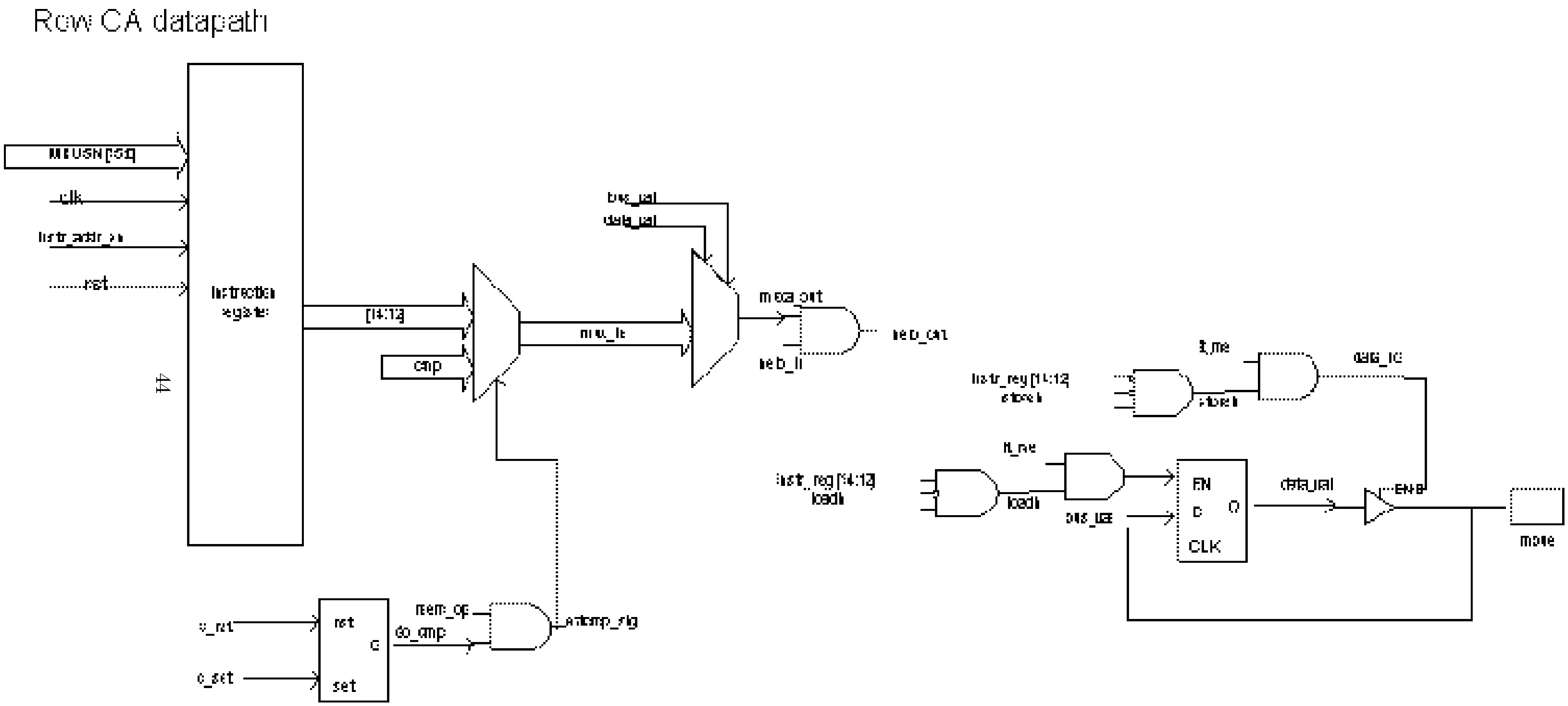

Figure 16: Row CA Datapath

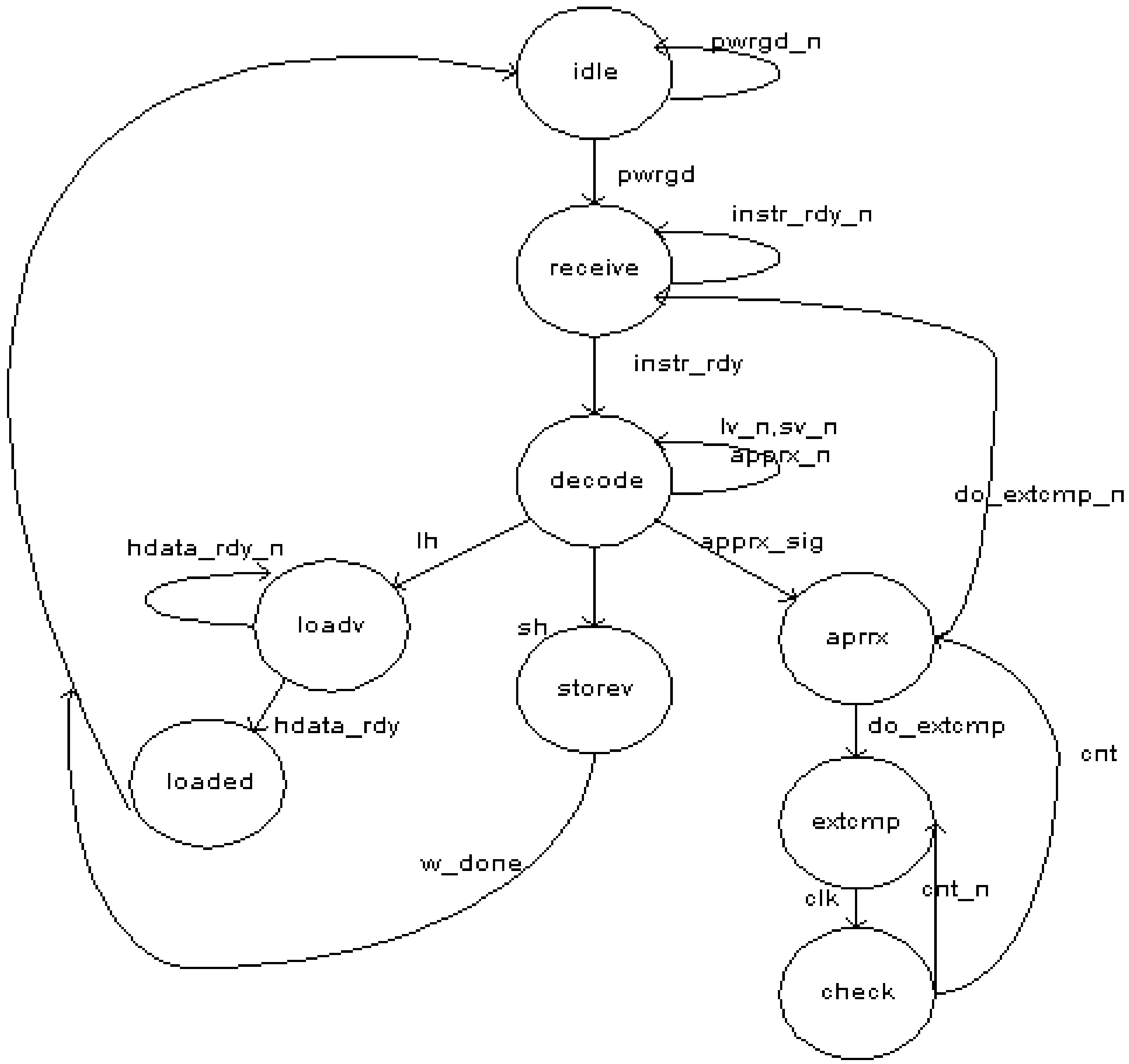

Figure 17: Row CA Controller

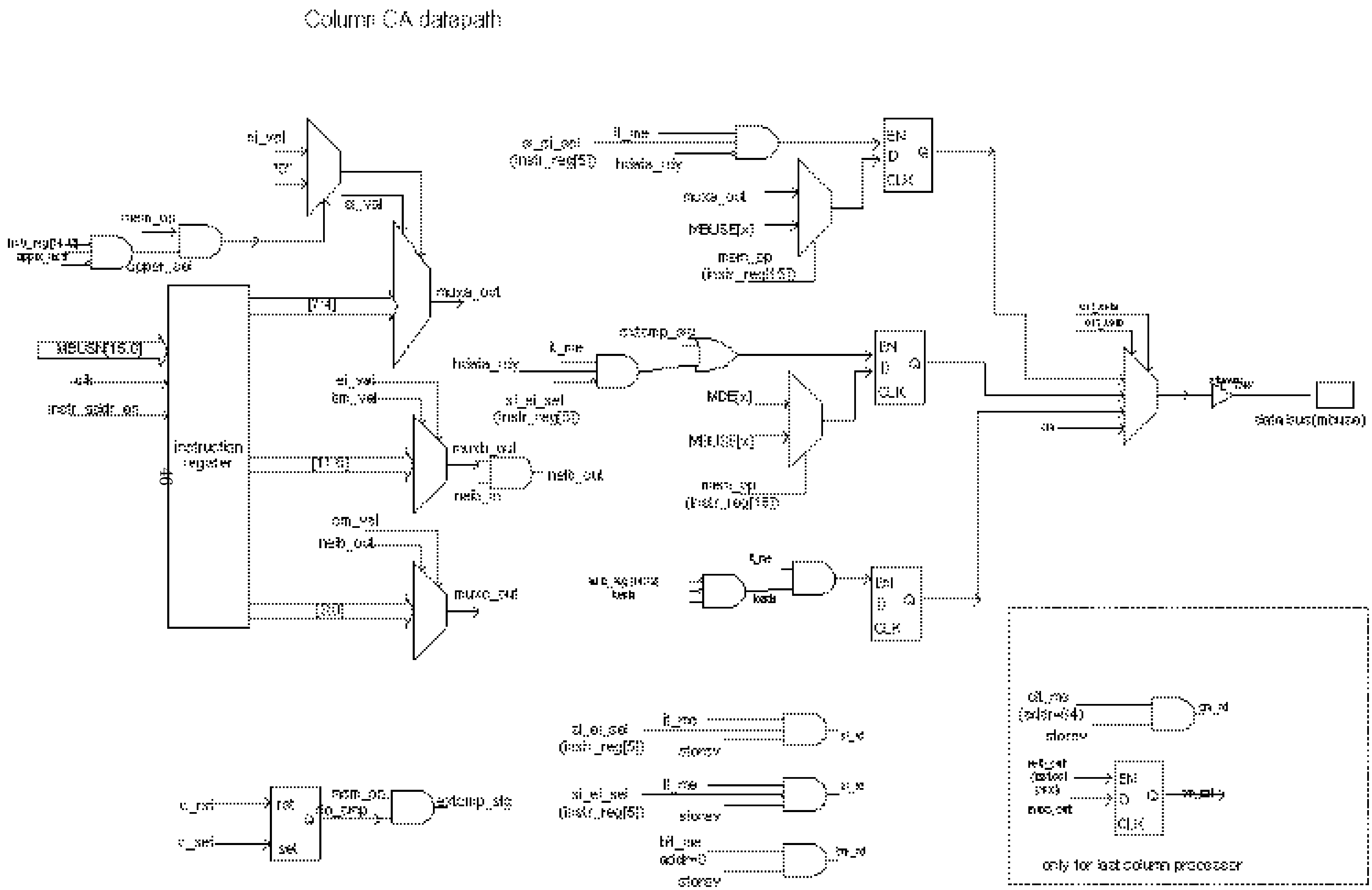

Figure 18: Column CA Datapath

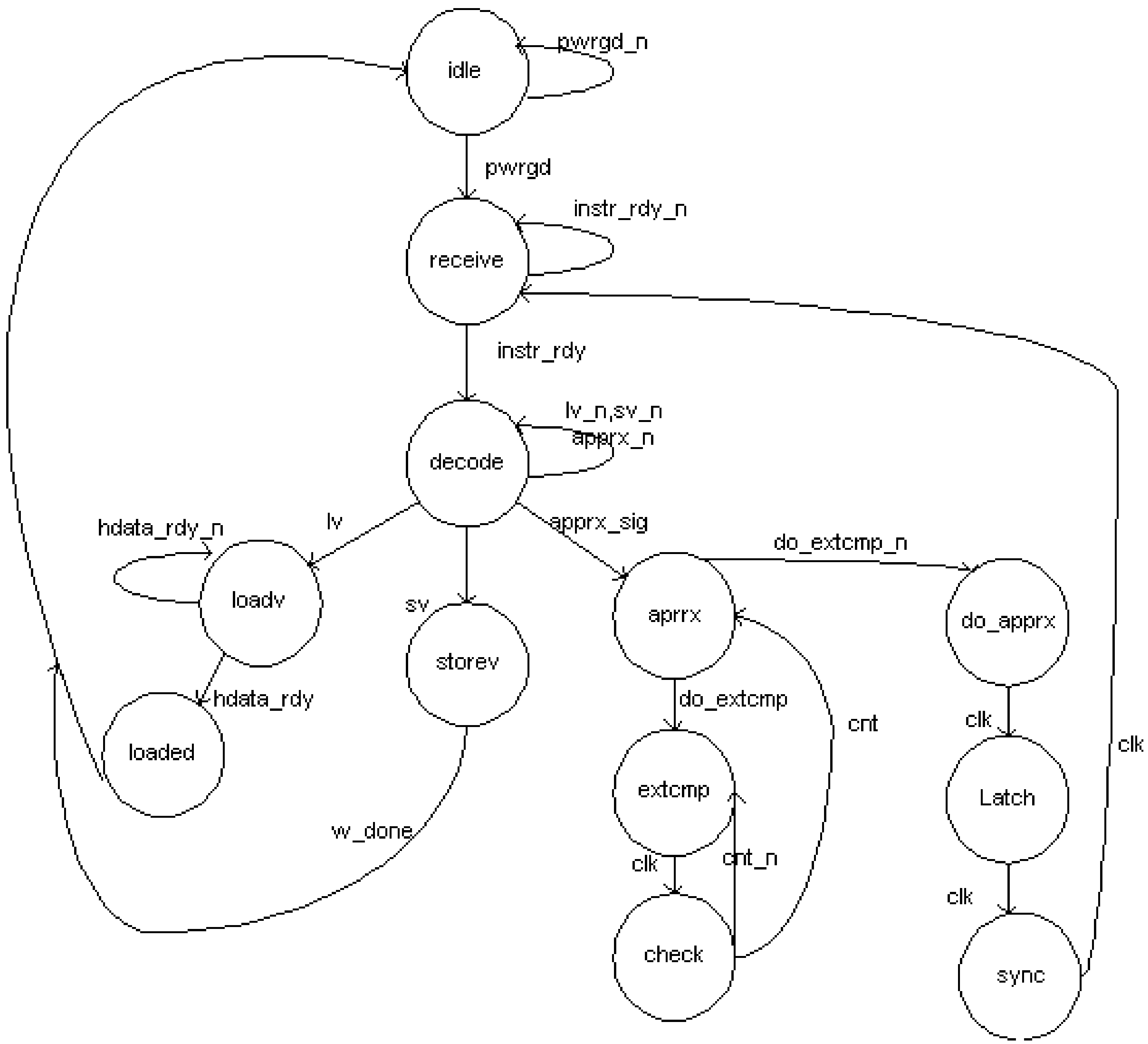

Figure 19: Column CA Controller

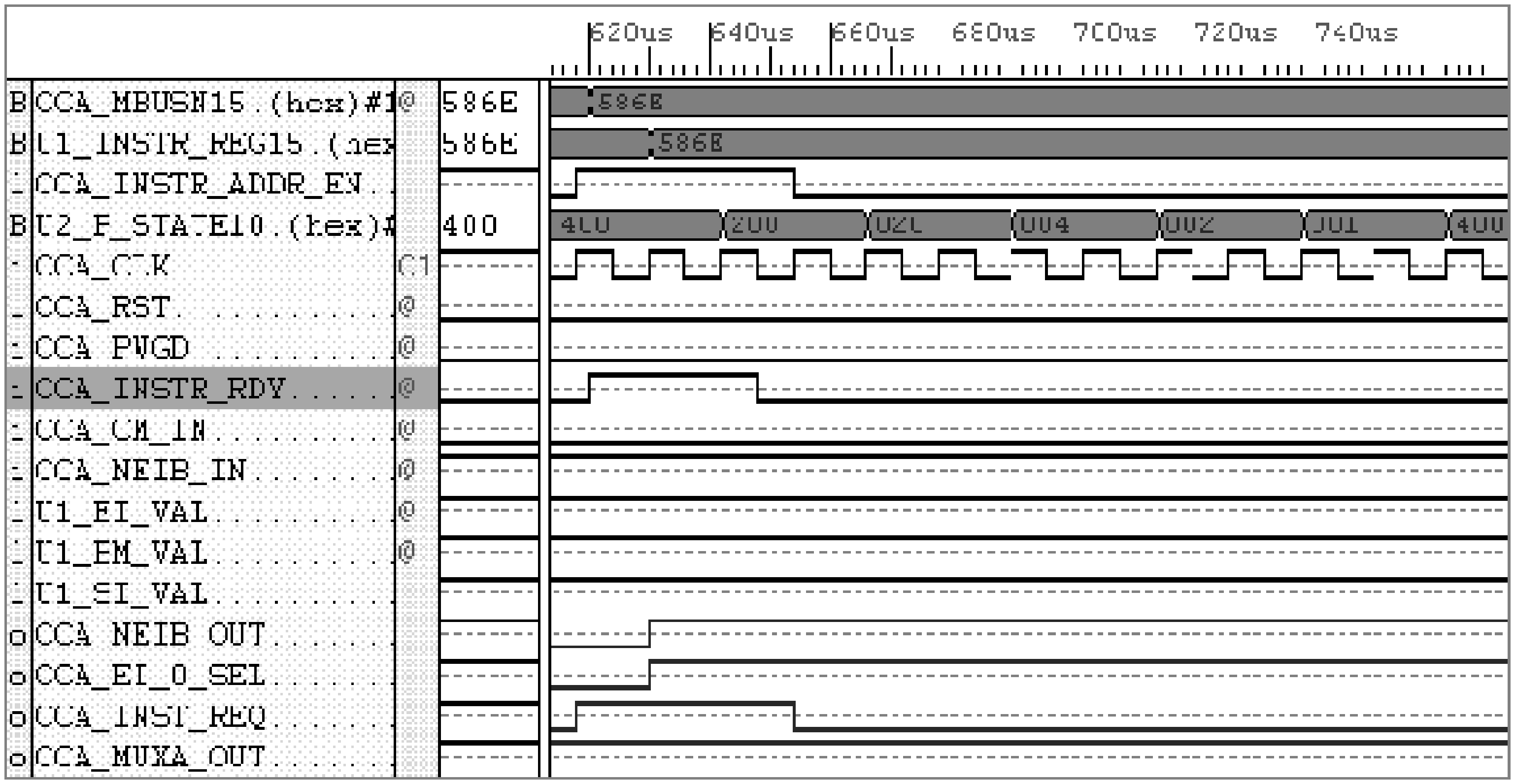

Figure 20: Simulation Waveforms for Matrix

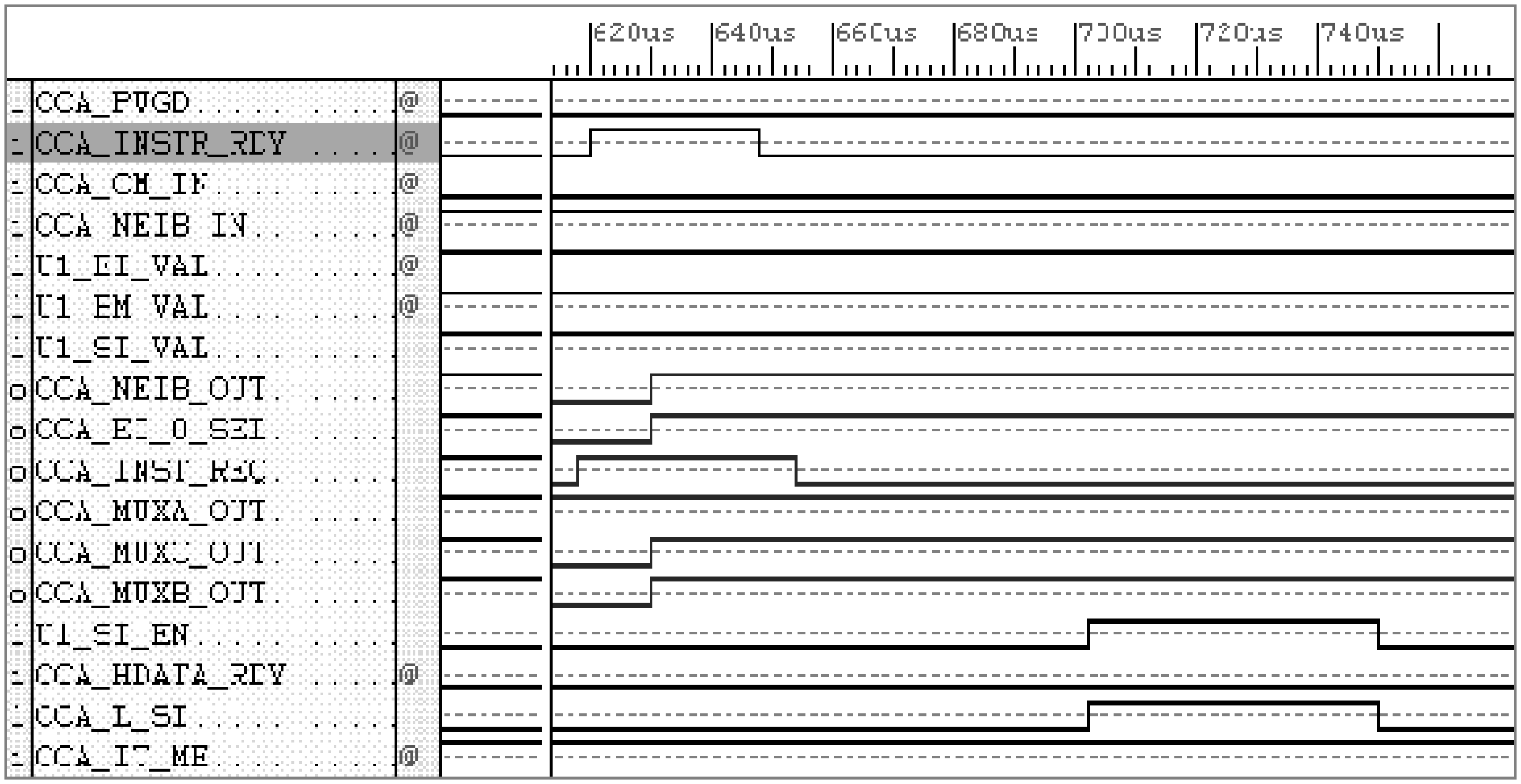

Figure 21: Simulation Waveforms for Matrix

Future Work

Due to time constraints there were several issues that we were not able to address during the course of the term. This section provides some ideas for future work.

The entire machine has not yet been tested together. The individual pieces have been tested as time allowed. Continuing testing should be one of the first things that happens when this project continues.

As stated earlier, this design only implements the UpperAPPROX, Lower-APPROX and EXTCOMP routines. The design could be expanded to handle the Indispensible and Def routines also.

A heavy restriction placed on the programmer of this Rough Set Machine is the inability to directly access the various memory pointers: instruction, vertical and horizontal. This could be alleviated by expanding the instruction set to include instructions to load/store these registers. This operation could be handled exclusively by the memory interface. The INST_RDY signal to the matrix would not be asserted on an access to these registers, but only when an instruction that must take place in the matrix is fetched.

On the simulation side of the design, it would be very useful to construct a behavioral VHDL model of the memory banks to allow more testing on the memory interface. It would be advantageous to design this model with the ability to load values from a file. This file would contain the instruction, horizontal and vertical data bits that would be loaded to the memory in the actual DEC Perle board. In this way, a programmer's code could be written, then tested in the simulator environment, then applied implemented in the actual memory banks on the DEC Perle board. The behavioral model of the DEC Perle-1 board's memory banks, not only has applications to the Rough Set machine, but to any future project utilizing the board.

The current implementation of the Rough Set Machine uses a relatively slow clock considering the fundamentally simple individual operations used in the matrix. This was done to simplify the memory interface. It would be possible to utilize the DEC Perle board's ability to use two separate clocks. One slow clock could run the memory interface logic, whereas a second higher speed clock could run the matrix logic.

The size of the data array handled by our implementation is fixed. It should be possible to allow for larger data and variable size data arrays. Possibly, this could be done by some shifting of the data array through the processors, only working on a portion of the data array at a time, the rest of it being stored in memory. Similar to what is done in image processing.

References

[1] Z. Pawlak, Rough Sets - Theoretical Aspects of Reasoning about Data, Kluwer Academic Publishers, Boston, London, Dordrecht, 1991.

[2] Q. Chen, "The Design of Cube Calculus Machine Using SRAM-Based FPGA Reconfigurable Hardware DEC's PeRLe-1 Board", Master's Thesis, Portland State University, 1998.

[3] M. Muraszkiewicz and H. Rybinski, "Towards a Parallel Rough Sets Computer", In Wojciech P. Ziarko (Ed.), Rough Sets, Fuzzy Sets and Knowledge Discovery (pp. 434-443), New York: Springer-Verlag.

[4] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard, "Programmable Active Memories: Reconfigurable Systems Come of Age", IEEE Transactions on VLSI System, Vol 4, No. 1, pp. 56-69, March 1996.

[5] L. Moll, J. Vuillemin and P. Boucard, "High-Energy Physics on DECPeRLe-1 Programmable Active Memory", Proceedings of the 1995 ACM International Symposium on FPGAs, Monterey, California, Feb 1995.

[6] M. Flynn, "Some Computer Organizations and Their Effectiveness," IEEE Transactions on Computers, Vol C-21, pp. 948-960, 1972.

[7] G. Almasi and A. Gottlieb, Highly Parallel Computing, The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1994.

[8] P.J. Lingras and Y.Y. Yao, "Data Mining Using Extensions of the Rough Set Model," Journal of the American Society for Information Science, 49(5):415-422, 1998.

Appendix A - Computations to Generate Core Values

$F = [4]_{\{a,b,c,d\}} = [4]_a \cap [4]_b \cap [4]_d \cap [4]_e = \{4,5,8,9,14,15\} \cap \{2,4,5,12,13,14\} \cap \{2,4,10,13,15\} \cap \{2,4,5,9,10,11,12\} = \{4\}$

$[4]_f = \{1,2,4,5,8,9\}$

$\bigcap(F - [4]_a) = [4]_b \cap [4]_d \cap [4]_e = \{2,4,5,12,13,14\} \cap \{2,4,10,13,15\} \cap \{2,4,5,9,10,11,12\} = \{2,4\} \subseteq [4]_f$
$\bigcap(F - [4]_b) = [4]_a \cap [4]_d \cap [4]_e = \{4,5,8,9,14,15\} \cap \{2,4,10,13,15\} \cap \{2,4,5,9,10,11,12\} = \{4\} \subseteq [4]_f$
$\bigcap(F - [4]_d) = [4]_a \cap [4]_b \cap [4]_e = \{4,5,8,9,14,15\} \cap \{2,4,5,12,13,14\} \cap \{2,4,5,9,10,11,12\} = \{4,5\} \subseteq [4]_f$
$\bigcap(F - [4]_e) = [4]_a \cap [4]_b \cap [4]_d = \{4,5,8,9,14,15\} \cap \{2,4,5,12,13,14\} \cap \{2,4,10,13,15\} = \{4\} \subseteq [4]_f$

Because all results are subsets of $[4]_f$ we can drop all input variables.

$F = [5]_{\{a,b,c,d\}} = [5]_a \cap [5]_b \cap [5]_d \cap [5]_e = \{4,5,8,9,14,15\} \cap \{2,4,5,12,13,14\} \cap \{1,5,8,9,11,12,14\} \cap \{2,4,5,9,10,11,12\} = \{5\}$

$[5]_f = \{1,2,4,5,8,9\}$

$\bigcap(F - [5]_a) = [5]_b \cap [5]_d \cap [5]_e = \{2,4,5,12,13,14\} \cap \{1,5,8,9,11,12,14\} \cap \{2,4,5,9,10,11,12\} = \{5,12\} \not\subseteq [5]_f$
$\bigcap(F - [5]_b) = [5]_a \cap [5]_d \cap [5]_e = \{4,5,8,9,14,15\} \cap \{1,5,8,9,11,12,14\} \cap \{2,4,5,9,10,11,12\} = \{5,9\} \subseteq [5]_f$
$\bigcap(F - [5]_d) = [5]_a \cap [5]_b \cap [5]_e = \{4,5,8,9,14,15\} \cap \{2,4,5,12,13,14\} \cap \{2,4,5,9,10,11,12\} = \{4,5\} \subseteq [5]_f$
$\bigcap(F - [5]_e) = [5]_a \cap [5]_b \cap [5]_d = \{4,5,8,9,14,15\} \cap \{2,4,5,12,13,14\} \cap \{1,5,8,9,11,12,14\} = \{5,14\} \not\subseteq [5]_f$

We must keep input variables a and e, because if they are dropped the resulting intersection contains decision rules that are not included in the set $[5]_f$.

$F = [8]_{\{a,b,c,d\}} = [8]_a \cap [8]_b \cap [8]_d \cap [8]_e = \{4,5,8,9,14,15\} \cap \{1,8,9,10,11,15\} \cap \{1,5,8,9,11,12,14\} \cap \{1,8,13,14,15\} = \{8\}$

$[8]_f = \{1,2,4,5,8,9\}$

$\bigcap(F - [8]_a) = [8]_b \cap [8]_d \cap [8]_e = \{1,8,9,10,11,15\} \cap \{1,5,8,9,11,12,14\} \cap \{1,8,13,14,15\} = \{1,8\} \subseteq [8]_f$

$\bigcap(F-[8]_b) = [8]_a \cap [8]_d \cap [8]_e = \{4,5,8,9,14,15\} \cap \{1,5,8,9,11,12,14\} \cap \{1,8,13,14,15\} = \{8,14\} \not\subseteq [8]_f$
$\bigcap(F-[8]_d) = [8]_a \cap [8]_b \cap [8]_e = \{4,5,8,9,14,15\} \cap \{1,8,9,10,11,15\} \cap \{1,8,13,14,15\} = \{8,15\} \not\subseteq [8]_f$
$\bigcap(F-[8]_e) = [8]_a \cap [8]_b \cap [8]_d = \{4,5,8,9,14,15\} \cap \{1,8,9,10,11,15\} \cap \{1,5,8,9,11,12,14\} = \{8,9\} \subseteq [8]_f$

Input variables b and d must be kept.

$F = [9]_{\{a,b,c,d\}} = [9]_a \cap [9]_b \cap [9]_d \cap [9]_e = \{4,5,8,9,14,15\} \cap \{1,8,9,10,11,15\} \cap \{1,5,8,9,11,12,14\} \cap \{2,4,5,9,10,11,12\} = \{9\}$

$[9]_f = \{1,2,4,5,8,9\}$

$\bigcap(F-[9]_a) = [9]_b \cap [9]_d \cap [9]_e = \{1,8,9,10,11,15\} \cap \{1,5,8,9,11,12,14\} \cap \{2,4,5,9,10,11,12\} = \{9,11\} \not\subseteq [9]_f$
$\bigcap(F-[9]_b) = [9]_a \cap [9]_d \cap [9]_e = \{4,5,8,9,14,15\} \cap \{1,5,8,9,11,12,14\} \cap \{2,4,5,9,10,11,12\} = \{9\} \subseteq [9]_f$
$\bigcap(F-[9]_d) = [9]_a \cap [9]_b \cap [9]_e = \{4,5,8,9,14,15\} \cap \{1,8,9,10,11,15\} \cap \{2,4,5,9,10,11,12\} = \{9\} \subseteq [9]_f$
$\bigcap(F-[9]_e) = [9]_a \cap [9]_b \cap [9]_d = \{4,5,8,9,14,15\} \cap \{1,8,9,10,11,15\} \cap \{1,5,8,9,11,12,14\} = \{8,9\} \subseteq [9]_f$

Input variable a must be kept.

$F = [10]_{\{a,b,c,d\}} = [10]_a \cap [10]_b \cap [10]_d \cap [10]_e = \{1,2,10,11,12,13\} \cap \{1,8,9,10,11,15\} \cap \{2,4,10,13,15\} \cap \{2,4,5,9,10,11,12\} = \{10\}$

$[10]_f = \{10,11,12,13,14,15\}$

$\bigcap(F-[10]_a) = [10]_b \cap [10]_d \cap [10]_e = \{1,8,9,10,11,15\} \cap \{2,4,10,13,15\} \cap \{2,4,5,9,10,11,12\} = \{10\} \subseteq [10]_f$
$\bigcap(F-[10]_b) = [10]_a \cap [10]_d \cap [10]_e = \{1,2,10,11,12,13\} \cap \{2,4,10,13,15\} \cap \{2,4,5,9,10,11,12\} = \{2,10\} \not\subseteq [10]_f$
$\bigcap(F-[10]_d) = [10]_a \cap [10]_b \cap [10]_e = \{1,2,10,11,12,13\} \cap \{1,8,9,10,11,15\} \cap \{2,4,5,9,10,11,12\} = \{10,11\} \subseteq [10]_f$
$\bigcap(F-[10]_e) = [10]_a \cap [10]_b \cap [10]_d = \{1,2,10,11,12,13\} \cap \{1,8,9,10,11,15\} \cap \{2,4,10,13,15\} = \{10\} \subseteq [10]_f$

Input variable b must be kept.

$F = [11]_{\{a,b,c,d\}} = [11]_a \cap [11]_b \cap [11]_d \cap [11]_e = \{1,2,10,11,12,13\} \cap \{1,8,9,10,11,15\} \cap \{1,5,8,9,11,12,14\} \cap \{2,4,5,9,10,11,12\} = \{10\}$

$[11]_f = \{10, 11, 12, 13, 14, 15\}$

$\bigcap(F - [11]_a) = [11]_b \cap [11]_d \cap [11]_e = \{1, 8, 9, 10, 11, 15\} \cap \{1, 5, 8, 9, 11, 12, 14\} \cap \{2, 4, 5, 9, 10, 11, 12\} = \{9, 11\} \not\subseteq [11]_f$

$\bigcap(F - [11]_b) = [11]_a \cap [11]_d \cap [11]_e = \{1, 2, 10, 11, 12, 13\} \cap \{1, 5, 8, 9, 11, 12, 14\} \cap \{2, 4, 5, 9, 10, 11, 12\} = \{11, 12\} \subseteq [11]_f$

$\bigcap(F - [11]_d) = [11]_a \cap [11]_b \cap [11]_e = \{1, 2, 10, 11, 12, 13\} \cap \{1, 8, 9, 10, 11, 15\} \cap \{2, 4, 5, 9, 10, 11, 12\} = \{10, 11\} \subseteq [11]_f$

$\bigcap(F - [11]_e) = [11]_a \cap [11]_b \cap [11]_d = \{1, 2, 10, 11, 12, 13\} \cap \{1, 8, 9, 10, 11, 15\} \cap \{1, 5, 8, 9, 11, 12, 14\} = \{1, 11\} \not\subseteq [11]_f$

Input variable a and e must be kept.

$F = [12]_{\{a,b,c,d\}} = [12]_a \cap [12]_b \cap [12]_d \cap [12]_e = \{1, 2, 10, 11, 12, 13\} \cap \{2, 4, 5, 12, 13, 14\} \cap \{1, 5, 8, 9, 11, 12, 14\} \cap \{2, 4, 5, 9, 10, 11, 12\} = \{12\}$

$[12]_f = \{10, 11, 12, 13, 14, 15\}$

$\bigcap(F - [12]_a) = [12]_b \cap [12]_d \cap [12]_e = \{2, 4, 5, 12, 13, 14\} \cap \{1, 5, 8, 9, 11, 12, 14\} \cap \{2, 4, 5, 9, 10, 11, 12\} = \{5, 12\} \not\subseteq [12]_f$

$\bigcap(F - [12]_b) = [12]_a \cap [12]_d \cap [12]_e = \{1, 2, 10, 11, 12, 13\} \cap \{1, 5, 8, 9, 11, 12, 14\} \cap \{2, 4, 5, 9, 10, 11, 12\} = \{11, 12\} \subseteq [12]_f$

$\bigcap(F - [12]_d) = [12]_a \cap [12]_b \cap [12]_e = \{1, 2, 10, 11, 12, 13\} \cap \{2, 4, 5, 12, 13, 14\} \cap \{2, 4, 5, 9, 10, 11, 12\} = \{2, 12\} \not\subseteq [12]_f$

$\bigcap(F - [12]_e) = [12]_a \cap [12]_b \cap [12]_d = \{1, 2, 10, 11, 12, 13\} \cap \{2, 4, 5, 12, 13, 14\} \cap \{1, 5, 8, 9, 11, 12, 14\} = \{12\} \subseteq [12]_f$

Input variable a and d must be kept.

Appendix B - Behavioral VHDL Code for Rough Set Operations

t1.vhd

```
library ieee;
use ieee.std_logic_1164.all;
— Routine for Extcomp (A[m,n],C[n],E[n])

— Routine for Indespensible variables.
entity extcomp is – entity declaration

generic (m : integer := 15 ; n : integer := 6);
port(
b: out std_logic_vector ( 1 to m));
end extcomp ;
–Architecture declaration :

architecture extcomp of extcomp is

–Signal Declaration
   type a1 is array (1 to m ,n downto 1) of std_logic;

  type a2 is array (n downto 1) of std_logic ;
  type a3 is array (1 to m , 1 to m) of std_logic;
  type a4 is array ( 1 to m ) of integer;
  type a5 is array (1 to 2,1 to m ,n downto 1) of std_logic;
  type a6 is array (n downto 1,1 to m ,n downto 1) of std_logic;
  type a7 is array (n downto 1,1 to m ,1 to m) of std_logic;
  signal bm :std_logic_vector (1 to m ) ;
  signal e,s : a3;
  signal result : a5;
  signal c1,temp  : a6;
  signal c  : a1;
  signal a :a1 :=
       (('0','0','0','0','0','1'),
       ('0','0','1','0','0','0'),
       ('0','0','1','0','0','1'),
       ('0','0','1','1','0','0'),
       ('0','0','1','1','0','1'),
       ('0','1','0','0','0','1'),
       ('0','1','0','1','0','1'),
       ('0','1','1','0','0','0'),
       ('0','1','1','1','0','0'),
       ('0','0','0','1','1','1'),
       ('0','0','1','0','1','1'),
       ('0','0','1','1','1','1'),
       ('0','1','0','1','1','0'),
```

```
        ('0','1','1','0','1','0'),
        ('0','1','1','1','1','0'));
begin

process(a,c,c1,e,temp,s,result)

-- Variable Declaration

        variable cnt :integer range 0 to 100 := 0;
        variable cnt2 :integer range 0 to 100 := 0;
        variable cnt1 :integer range 0 to 1000 := 0;
        variable lower_approx : a4 ;
        variable Upper_approx : a4 ;
        variable j1 : integer range 1 to n := 1;
        variable i : integer range 1 to m := 1;
        variable j : integer range 1 to n := 1;
        variable k : integer range 1 to m := 1;
begin

-- Def (A[m,m],B[m],C[m]) :

        r1 : for i in 1 to m loop
            r2 : for j in n downto 1 loop
                result(1,i,j) <= a(i,j);
            end loop r2;
        end loop r1;
--Routine for ExtComp.

--Each column of matrix is masked at a time and matrix rows are compared for
consistancy, if the matrix is consistant, the variable masked is despensible.
l1: for j in n downto 2 loop

cnt1 := 0;

        l3 : for j1 in n downto 1 loop

          l2 : for i in 1 to m loop
            temp(j,i,j1) <= a(i,j1);
          end loop l2;

        end loop l3;
-- all elements in the column is masked:
         l4: for i in 1 to m loop

           temp(j,i,j) <= '0';
        end loop l4;
-- each row is copied in C vector and compared with all rows and despensible

column is maked '0' and a new matrix is formed :
  l5: for i in 1 to m loop
        c1(j,i,1) <= a(i,1);
```

```
    l6: for j1 in n downto 2 loop
       c1(j,i,j1)<= temp(j,i,j1);
   end loop l6;
   l7: for k in 1 to m loop
      cnt := 0;
      l8: for j1 in n downto 2 loop
           if c1(j,i,j1) =  temp(j,k,j1) then
              cnt := cnt + 1;
           else
              cnt := cnt;
           end if;
      end loop l8;
       if cnt = n-1 then
         if c1(j,i,1) = temp(j,k,1) then
           cnt1 := cnt1 +1;
         else
           cnt1 := cnt1;
           exit l5;
         end if;
      end if;
    end loop l7;
    l9: for i1 in 1 to m loop
      if (cnt1 ¿ m )then
         result(2,i1,j)<= '0';
      else
         result(2,i1,j) <= a(i1,j);
      end if;
         result(2,i1,1) <= a(i1,1);
    end loop l9;
  end loop l5;
end loop l1;
```

–EXTCOMP (A[m,n],C[n],E[m]) and BasicCAT (A[m,n],i,e[m]) for LOWER APPROXIMATION and UPPER APPROXIMATION :

```
l10 : for i in 1 to m loop
   bm(i) <= result(2,i,1);
```

–Vector Ei and Si is formed :

```
 l11 : for j in n downto 1 loop
       c(i,j) <= result(2,i,j);
 end loop l11;
  l12 : for k in 1 to m loop
     cnt := 0;
```

```
    l13 : for j in 2 to n loop
       if c(i,j) =  result(2,k,j) then
          cnt := cnt + 1;
       else
         cnt := cnt;
       end if;
    end loop l13;
     if cnt = 5 then
      e(i,k) <= '1';
    else
      e(i,k) <= '0';
    end if;
  end loop l12;
end loop l10;
l14: for k in 1 to m loop

  l15: for i in 1 to m loop
      s(i,k)  <= bm(i) and e(i,k);
  end loop l15;
end loop l14;
l17: for k in 1 to m loop
-- LOWER APPROXIMATION (A[m,n],C[n],E[m]):
    cnt1 := 0;
  l16: for i in 1 to m loop
       if s(i,k) = e(i,k) then
         cnt1 := cnt1 +1;
      else
         cnt1 := cnt1;
      end if;
  end loop l16;
  l19 : for i in 1 to m loop
      if  cnt1 = m then
         lower_approx(k) := k;
      else
         lower_approx(k) := 0;
      end if;
 end loop l19;
-- UPPER APPROXIMATION (A[m,n],C[n],E[m]) :
  cnt:=0;
  l18 : for i in 1 to m-1 loop
      if s(i,k) = '1' then
```

```
            cnt := cnt + 1;
        else
            cnt := cnt;
        end if;
    end loop l18;
     if cnt ¿ 0 then

        Upper_approx(k) := k;
    else
        Upper_approx(k) := 0;
    end if;
end loop l17;

end process;

end extcomp;
```

Appendix C - Synthesizeable VHDL Code for Rough Set Operations

MEM_INTF

mem_top.vhd

```
library IEEE;
use IEEE.STD_Logic_1164.all;
use work.cntl_ne_package.all;
use work.sw_n_package.all;
use work.sw_e_package.all;
–

– mem_top.vhd
–
–   Top level of memory interface.
–
entity mem_top is

   port ( clk, reset      : in std_logic;
          – control signals from matrix via matrix ring

         inst_req          : in std_logic;
         inst_rdy          : out std_logic;
         rdata_rdy         : out std_logic;
         wdata_rdy         : in std_logic;
          – control signals to North memory bank

         ram_addrN         : out std_logic_vector(19 downto 2);
         ram_readN_n       : out std_logic;
         ram_writeN_n      : out std_logic;
         ram_dis_lowN_n    : out std_logic;
         ram_dis_highN_n   : out std_logic;
          – Data Path for North memory bank

         – 32-bit data path from the memory
         ramdataN          : inout std_logic_vector(31 downto 0);
         – 64-bit data path to the matrix, inout
         mbusN             : inout std_logic_vector(63 downto 0);
          – control signals to East memory bank

         ram_addrE         : out std_logic_vector(19 downto 2);
         ram_readE_n       : out std_logic;
         ram_writeE_n      : out std_logic;
         ram_dis_lowE_n    : out std_logic;
         ram_dis_highE_n   : out std_logic;
          – Data Path for North memory bank
```

```
        – 32-bit data path from the memory
        ramdataE        : inout std_logic_vector(31 downto 0);
        – 64-bit data path to the matrix, inout
        mbusE           : inout std_logic_vector(63 downto 0)
      );
end entity mem_top;
architecture rtl of mem_top is

  – signals between cntl_ne and north switch and east switch

  signal rd_hori, wr_hori, rd_vert, wr_vert  : std_logic;
  signal inst_req_out, wdata_rdy_out         : std_logic;
begin

  – cntl_ne is mapped to the CNE FPGA on the DEC Perle board

  cntl_ne_1: cntl_ne port map (
                          – external signals
                               clk        => clk,
                                 reset      => reset,
                           inst_req    => inst_req,

                            inst_rdy    => inst_rdy,
                            rdata_rdy   => rdata_rdy,
                            wdata_rdy   => wdata_rdy,

                            – control signals from north switch
                          rd_hori     => rd_hori,
                             wr_hori     => wr_hori,
                            rd_vert     => rd_vert,
                            wr_vert     => wr_vert,

                            – control signals to north switch
                          inst_req_out  => inst_req_out,
                            wdata_rdy_out => wdata_rdy_out,
                           – control signals to North memory bank

                            ram_addrN        => ram_addrN,
                            ram_readN_n      => ram_readN_n,
                            ram_writeN_n     => ram_writeN_n,
                            ram_dis_lowN_n   => ram_dis_lowN_n,
                            ram_dis_highN_n => ram_dis_highN_n,
                           – control signals to East memory bank

                                 ram_addrE        => ram_addrE,
                            ram_readE_n      => ram_readE_n,
                            ram_writeE_n     => ram_writeE_n,
                            ram_dis_lowE_n   => ram_dis_lowE_n,
                            ram_dis_highE_n => ram_dis_highE_n
                          );
```

```
– sw_n is mapped to the SWN FPGA on the DEC Perle board

sw_n1: sw_n port map (
                    – external signals
                        clk      => clk,
                          reset    => reset,
                      – control signals to cntl_ne

                    rd_hori   => rd_hori,
                      wr_hori   => wr_hori,
                     rd_vert   => rd_vert,
                     wr_vert   => wr_vert,

                     – control signals from cntl_ne
                    – This connection needs to come from the cntl_se
                    – because, the matrix ring is not directly connected
                    – to the matrix ring.
                    inst_req  => inst_req_out,
                     wdata_rdy  => wdata_rdy_out,
                      – 32-bit data path from the memory

                     ramdata   => ramdataN,
                      – 64-bit data path to the matrix, inout

                     mbus      => mbusN
                    );
– sw_e is mapped to the SWE FPGA on the DEC Perle board

sw_e1: sw_e port map (
                    – external signals
                        clk      => clk,
                          reset    => reset,
                      – control signals to cntl_ne

                    rd_hori   => rd_hori,
                      wr_hori   => wr_hori,
                     rd_vert   => rd_vert,
                     wr_vert   => wr_vert,

                     – control signals from cntl_ne
                    – This connection needs to come from the cntl_se
                    – because, the matrix ring is not directly connected
                    – to the matrix ring.
                    inst_req  => inst_req_out,
                     wdata_rdy  => wdata_rdy_out,
                      – 32-bit data path from the memory

                     ramdata   => ramdataE,
                      – 64-bit data path to the matrix, inout
```

```
                                mbus        => mbusE
                              );
end architecture rtl;
```

cntl_ne.vhd

```
library IEEE;
use IEEE.STD_Logic_1164.all;
use IEEE.std_logic_arith.all;
use work.cntlswsm_package.all;
–
– cntl_ne.vhd
–
–   Controller to memory, to be placed in CNE FPGA on DEC Perle board.
–
–   Timing for memory:
–           Connection                  Name              Delay
–   ————————————————————————————————————-
–   Memory read, address to data   RamAddrX to RamDataX  46 ns
–   Memory read, control to data   RamReadX to RamDataX  35 ns
–   Memory write, from any source                        33 ns
–
–   NOTE: this state machine assumes  60 ns clock, this should
–         allow plenty of time
–
package cntl_ne_package is

component cntl_ne is

   port ( clk, reset      : in std_logic;
          – control signals from matrix via matrix ring

         inst_req         : in std_logic;
         inst_rdy         : out std_logic;
         rdata_rdy        : out std_logic;
         wdata_rdy        : in std_logic;

         – control signals from north switch
         rd_hori          : in std_logic;
         wr_hori          : in std_logic;
         rd_vert          : in std_logic;
         wr_vert          : in std_logic;

         – control signals to north switch
         inst_req_out    : out std_logic;
```

```
        wdata_rdy_out  : out std_logic;
         -- control signals to North memory bank
        ram_addrN       : out std_logic_vector(19 downto 2);
        ram_readN_n     : out std_logic;
        ram_writeN_n    : out std_logic;
        ram_dis_lowN_n  : out std_logic;
        ram_dis_highN_n : out std_logic;
         -- control signals to East memory bank
        ram_addrE       : out std_logic_vector(19 downto 2);
        ram_readE_n     : out std_logic;
        ram_writeE_n    : out std_logic;
        ram_dis_lowE_n  : out std_logic;
        ram_dis_highE_n : out std_logic
      );
end component;
end cntl_ne_package;
library IEEE;

use IEEE.STD_Logic_1164.all;
use IEEE.std_logic_arith.all;
use work.cntlswsm_package.all;
entity cntl_ne is
   port ( clk, reset     : in std_logic;
         -- control signals from matrix via matrix ring
        inst_req        : in std_logic;
        inst_rdy        : out std_logic;
        rdata_rdy       : out std_logic;
        wdata_rdy       : in std_logic;

        -- control signals from north switch
        rd_hori         : in std_logic;
        wr_hori         : in std_logic;
        rd_vert         : in std_logic;
        wr_vert         : in std_logic;

        -- control signals to north switch
        inst_req_out    : out std_logic;
        wdata_rdy_out   : out std_logic;
         -- control signals to North memory bank
        ram_addrN       : out std_logic_vector(19 downto 2);
        ram_readN_n     : out std_logic;
        ram_writeN_n    : out std_logic;
        ram_dis_lowN_n  : out std_logic;
        ram_dis_highN_n : out std_logic;
```

```
        – control signals to East memory bank
        ram_addrE       : out std_logic_vector(19 downto 2);
        ram_readE_n     : out std_logic;
        ram_writeE_n    : out std_logic;
        ram_dis_lowE_n  : out std_logic;
        ram_dis_highE_n : out std_logic
      );
end entity cntl_ne;
architecture rtl of cntl_ne is
   signal state           : std_logic_vector(8 downto 0);
   – default values for pointers after reset
–  constant INST_PTR_DEF  : std_logic_vector(20 downto 1) := ”00000000000000000000”;
–  constant HDATA_PTR_DEF : std_logic_vector(20 downto 2) := ”1000000000000000000”;
–  constant VDATA_PTR_DEF : std_logic_vector(20 downto 2) := ”1000000000000000000”;
  constant INST_PTR_DEF  : integer range 0 to 1048575 := 0;
  constant HDATA_PTR_DEF : integer range 0 to 1048575 := 524288;
  constant VDATA_PTR_DEF : integer range 0 to 1048575 := 524288;
   signal inst_one      : std_logic_vector(20 downto 1);
   – pointers into memory
  signal inst_ptr    : std_logic_vector(19 downto 0);   – instruction pointer
  signal hdata_ptr   : std_logic_vector(19 downto 0);   – horizontal data pointer
  signal vdata_ptr   : std_logic_vector(19 downto 0);   – vertical data pointer
   signal inst_ptr_int    : integer range 0 to 1048575;   – instruction pointer
  signal hdata_ptr_int   : integer range 0 to 1048575;   – horizontal data pointer
  signal vdata_ptr_int   : integer range 0 to 1048575;   – vertical data pointer
begin
    inst_one <= ”00000000000000000001”;
    cntl_sw_sm3: cntl_sw_sm port map (
                          clk        => clk,
                             reset      => reset,
                             rd_inst    => inst_req,
                      rd_hori    => rd_hori,
                      wr_hori    => wr_hori,
                      rd_vert    => rd_vert,
                      wr_vert    => wr_vert,
                      state      => state
                    );
  – pointers
 process( clk, reset )
 begin
   if (reset=’1’) then
```

```
    inst_ptr_int <= INST_PTR_DEF;
  elsif rising_edge(clk) then
    if (state=RINST) then
        inst_ptr_int <= inst_ptr_int + 2;
    end if;
  end if;
end process;
 process( clk, reset )
begin
  if (reset='1') then
    hdata_ptr_int <= HDATA_PTR_DEF;
  elsif rising_edge(clk) then
  if (state=RHORI1 or state=RHORI2 or state=WHORI1 or state=WHORI2)
then
      hdata_ptr_int <= hdata_ptr_int + 4;
    end if;
  end if;
end process;
 process( clk, reset )
begin
  if (reset='1') then
    vdata_ptr_int <= VDATA_PTR_DEF;
  elsif rising_edge(clk) then
  if (state=RVERT1 or state=RVERT2 or state=WVERT1 or state=WVERT2)
then
      vdata_ptr_int <= vdata_ptr_int + 4;
    end if;
  end if;
end process;
 hdata_ptr <= CONV_STD_LOGIC_VECTOR(hdata_ptr_int, 20);
vdata_ptr <= CONV_STD_LOGIC_VECTOR(vdata_ptr_int, 20);
inst_ptr  <= CONV_STD_LOGIC_VECTOR(inst_ptr_int, 20);
 – assign control signals to memory
ram_addrN      <= hdata_ptr(19 downto 2)
                  when (state=RHORI1 or state=RHDEAD or state=RHORI2
                     or state=WHORI1 or state=WHDEAD or state=WHORI2)
else
                inst_ptr(19 downto 2);
  ram_readN_n       <= '1' when not ( state=RINST or state=RHORI1 or
state=RHORI2 ) else
                 '0';
 ram_writeN_n    <= '1' when not ( state=WHORI1 or state=WHORI2 ) else
                 '0';
```

```
 ram_dis_lowN_n  <= (not INST_PTR(1)) when (state=RINST) else
                    '1';
 ram_dis_highN_n <= (INST_PTR(1))     when (state=RINST) else
                    '1';
 ram_addrE       <= vdata_ptr(19 downto 2);
 ram_readE_n     <= '1' when not ( state=RVERT1 or state=RVERT2 ) else
                    '0';
 ram_writeE_n    <= '1' when not ( state=WVERT1 or state=WVERT2 ) else
                    '0';
 ram_dis_lowE_n  <= '1';
 ram_dis_highE_n <= '1';
 – assign control signals to matrix via matrix ring
 inst_rdy        <= '1' when (state=RINST) else
                    '0';
 rdata_rdy       <= '1' when (state=RVERT2) or (state=RHORI2) else
                    '0';
 – assign signals to north switch, must do this becuase inst_req and
 – wdata_rdy come to the controller via matrix ring and switches are
 – not connected to the matrix ring
 inst_req_out   <= inst_req;
 wdata_rdy_out  <= wdata_rdy;
end architecture rtl;
```

sw_n.vhd

```
library IEEE;
use IEEE.STD_Logic_1164.all;
use work.cntlswsm_package.all;
–
– sw_n.vhd
–
– North Switch between memory and matrix, takes data from memory and
– distributes it appropriately to the matrix.
–
– There are 4 16-bit data paths to the matrix.  16-bit instructions are
– replicated on all 4 data paths.  64 bits of horizontal data are
– presented over all 4 data paths.
–
package sw_n_package is
```

```
component sw_n is
   port ( clk, reset      : in std_logic;
          – control signals to cntl_ne
         rd_hori         : out std_logic;
         wr_hori         : out std_logic;
         rd_vert         : out std_logic;
         wr_vert         : out std_logic;

         – control signals from cntl_ne
         inst_req        : in std_logic;
         wdata_rdy       : in std_logic;
          – 32-bit data path from the memory
         ramdata         : inout std_logic_vector(31 downto 0);
          – 64-bit data path to the matrix, inout
         mbus            : inout std_logic_vector(63 downto 0)
       );
end component;
end sw_n_package;
library IEEE;

use IEEE.STD_Logic_1164.all;
use work.cntlswsm_package.all;
entity sw_n is
   port ( clk, reset      : in std_logic;
          – control signals to cntl_ne_sm
         rd_hori         : out std_logic;
         wr_hori         : out std_logic;
         rd_vert         : out std_logic;
         wr_vert         : out std_logic;

         – control signals from cntl_ne_sm
         inst_req        : in std_logic;
         wdata_rdy       : in std_logic;
          – 32-bit data path from the memory
         ramdata         : inout std_logic_vector(31 downto 0);
          – 64-bit data path to the matrix, inout
         mbus            : inout std_logic_vector(63 downto 0)
       );
end entity sw_n;
architecture rtl of sw_n is
   signal state            : std_logic_vector(8 downto 0);
```

```
signal ramdata_reg      : std_logic_vector(31 downto 0);
signal mbus_reg         : std_logic_vector(63 downto 0);
constant MAT_MEM        : integer := 15;
constant LO_ST          : integer := 14;
constant HORI_VERT      : integer := 12;
signal inst_reg         : std_logic_vector(15 downto 0);
signal inst_in          : std_logic_vector(15 downto 0);
signal inst_addr1       : std_logic;
signal rd_hori_buf, wr_hori_buf, rd_vert_buf, wr_vert_buf : std_logic;

begin
  cntl_sw_sm1: cntl_sw_sm port map (
                    clk       => clk,
                    reset     => reset,
                    rd_inst   => inst_req,
                    rd_hori   => rd_hori_buf,
                    wr_hori   => wr_hori_buf,
                    rd_vert   => rd_vert_buf,
                    wr_vert   => wr_vert_buf,
                    state     => state
                  );
  – inst_addr1 says where to get the instruction (31:16) or (15:0)
 – equivalent to inst_ptr(1)
 process( clk, reset )
 begin
   if (reset='1') then
     inst_addr1 <= '0';
   elsif rising_edge(clk) then
     if (state=RINST) then
       inst_addr1 <= not inst_addr1;
     end if;
   end if;
 end process;
  – need to register lower 32-bits of data when reading data
 process( clk, reset )
 begin
   if (reset='1') then
     ramdata_reg <= "00000000000000000000000000000000";
   elsif rising_edge(clk) then
     if (state=RHORI1) then
       ramdata_reg <= ramdata;
```

```
    end if;
  end if;
end process;
 – need to register data from matrix, later will cycle it to memory
process( clk, reset )
begin
  if (reset='1') then
   mbus_reg <= "0000000000000000000000000000000000000000000000000000000000000000";
  elsif rising_edge(clk) then
    if (wdata_rdy = '1') then
      mbus_reg <= mbus;
    end if;
  end if;
end process;
 inst_in <= ramdata(15 downto 0) when (inst_addr1='0') else
          ramdata(31 downto 16);
 – keep track of instruction for use later
process( clk, reset )
begin
  if (reset='1') then
    inst_reg <= "0000000000000000";
  elsif rising_edge(clk) then
    if (state=RINST) then
      – save the instruction, so we can drive a command to the cntl_ne
      – if it is a load/store
      inst_reg <= inst_in;
   elsif ( (state=RHORI1) or (state=RVERT1) or (state=WHORI1) or (state=WVERT1)
) then
      – clear the instruction register after we have used it
      inst_reg <= "0000000000000000";
    end if;
  end if;
end process;
 – only output MBus to matrix when we need to (i.e. on reads), otherwise float,
 – on writes matrix drives mbus and we cycle the data to the memory
– mbus(15 downto 0) <= ramdata(15 downto 0);
mbus(15 downto 0)  <= inst_in                 when (state=RINST) else
                 ramdata_reg(15 downto 0)  when (state=RHORI2) else
                 "ZZZZZZZZZZZZZZZZ"; – tri-state
mbus(31 downto 16) <= inst_in                 when (state=RINST) else
                 ramdata_reg(31 downto 16) when (state=RHORI2) else
                 "ZZZZZZZZZZZZZZZZ"; – tri-state
–mbus(47 downto 32) <=  ramdata(15 downto 0)  when ((state=RINST) and
```

(inst_addr1='0')) else
– ramdata(31 downto 16) when ((state=RINST) and (inst_addr1='1'))
else
– ramdata(15 downto 0) when (state=RHORI2) else
– "ZZZZZZZZZZZZZZZZ"; – tri-state
mbus(47 downto 32) <= inst_in when (state=RINST) else

ramdata(15 downto 0) when (state=RHORI2) else
"ZZZZZZZZZZZZZZZZ"; – tri-state
mbus(63 downto 48) <= inst_in when (state=RINST) else

ramdata(31 downto 16) when (state=RHORI2) else
"ZZZZZZZZZZZZZZZZ"; – tri-state
– only output ramdata when we need to (i.e. on writes), otherwise floats

– on reads memory drives ramdata and we cycle the data to the matrix
ramdata <= mbus_reg(31 downto 0) when (state=WHORI1) else
mbus_reg(63 downto 32) when (state=WHORI2) else
"ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ"; – tri-state
– pick off the instruction bits to send to cntl_ne

rd_hori_buf <= inst_reg(MAT_MEM) and not inst_reg(LO_ST) and not inst_reg(HORI_VERT);
rd_vert_buf <= inst_reg(MAT_MEM) and not inst_reg(LO_ST) and inst_reg(HORI_VERT);
wr_hori_buf <= not inst_reg(HORI_VERT) and wdata_rdy; – inst a write and
matrix has provided data
wr_vert_buf <= inst_reg(HORI_VERT) and wdata_rdy; – inst a write and matrix has provided data
rd_hori <= rd_hori_buf;

rd_vert <= rd_vert_buf;
wr_hori <= wr_hori_buf;
wr_vert <= wr_vert_buf;

end architecture rtl;

sw_e.vhd

library IEEE;
use IEEE.STD_Logic_1164.all;
use work.cntlswsm_package.all;
–

– sw_e.vhd
–
– East Switch between memory and matrix, takes data from memory and
– distributes it appropriately to the matrix.
–
– The East Switch is simpler than the North Switch. The East Switch

```
– does not need to handle instructions.
–
– There are 4 16-bit data paths to the matrix.
– 64 bits of vertical data are presented over all 4 data paths.
–
package sw_e_package is

component sw_e is

   port ( clk, reset      : in std_logic;
          – control signals from sw_n

         rd_hori          : in std_logic;
         wr_hori          : in std_logic;
         rd_vert          : in std_logic;
         wr_vert          : in std_logic;

         – control signals from cntl_ne
         inst_req         : in std_logic;
         wdata_rdy        : in std_logic;
          – 32-bit data path from the memory

         ramdata          : inout std_logic_vector(31 downto 0);
          – 64-bit data path to the matrix, inout

         mbus             : inout std_logic_vector(63 downto 0)
       );
end component;
end sw_e_package;
library IEEE;

use IEEE.STD_Logic_1164.all;
use work.cntlswsm_package.all;
entity sw_e is

   port ( clk, reset      : in std_logic;
          – control signals to cntl_ne_sm

         rd_hori          : in std_logic;
         wr_hori          : in std_logic;
         rd_vert          : in std_logic;
         wr_vert          : in std_logic;

         – control signals from cntl_ne_sm
         inst_req         : in std_logic;
         wdata_rdy        : in std_logic;
          – 32-bit data path from the memory

         ramdata          : inout std_logic_vector(31 downto 0);
          – 64-bit data path to the matrix, inout
```

```
        mbus            : inout std_logic_vector(63 downto 0)
       );
end entity sw_e;
architecture rtl of sw_e is

   signal state            : std_logic_vector(8 downto 0);

   signal ramdata_reg      : std_logic_vector(31 downto 0);

   signal mbus_reg         : std_logic_vector(63 downto 0);


begin
    cntl_sw_sm2: cntl_sw_sm port map (

                          clk         => clk,
                             reset       => reset,
                             rd_inst     => inst_req,
                      rd_hori     => rd_hori,
                      wr_hori     => wr_hori,
                      rd_vert     => rd_vert,
                      wr_vert     => wr_vert,
                      state       => state
                    );
 – need to register lower 32-bits of data when reading data

 process( clk, reset )
 begin
   if (reset='1') then
     ramdata_reg <= "00000000000000000000000000000000";
   elsif rising_edge(clk) then
     if (state=RVERT1) then
       ramdata_reg <= ramdata;
     end if;
   end if;
 end process;
 – need to register data from matrix, later will cycle it to memory

 process( clk, reset )
 begin
   if (reset='1') then
    mbus_reg <= "0000000000000000000000000000000000000000000000000000000000000000";
   elsif rising_edge(clk) then
     if (wdata_rdy='1') then
       mbus_reg <= mbus;
     end if;
   end if;
 end process;
 – only output MBus to matrix when we need to (i.e. on reads), otherwise float,
```

```
-- on writes matrix drives mbus and we cycle the data to the memory
 mbus(15 downto 0)  <=  ramdata_reg(15 downto 0)  when (state=RVERT2)
else
                  "ZZZZZZZZZZZZZZZZ";  -- tri-state
  mbus(31 downto 16) <=  ramdata_reg(31 downto 16) when (state=RVERT2)
else
                  "ZZZZZZZZZZZZZZZZ";  -- tri-state
  mbus(47 downto 32) <=  ramdata(15 downto 0)       when (state=RVERT2)
else
                  "ZZZZZZZZZZZZZZZZ";  -- tri-state
  mbus(63 downto 48) <=  ramdata(31 downto 16)      when (state=RVERT2)
else
                  "ZZZZZZZZZZZZZZZZ";  -- tri-state
  -- only output ramdata when we need to (i.e. on writes), otherwise floats
 -- on reads memory drives ramdata and we cycle the data to the matrix
 ramdata      <=  mbus_reg(31 downto 0)      when (state=WVERT1) else
                  mbus_reg(63 downto 32)     when (state=WVERT2) else
                  "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ"; -- tri-state
end architecture rtl;
```

cntl_sw_.vhd

```
library IEEE;
use IEEE.STD_Logic_1164.all;
--
-- cntl_sw_sm.vhd
--
--   State machine used in both the Controller and Switch FPGAs
--
package cntlswsm_package is
-- Make only one state bit change at a time
-- to minimize glitching on decoding of states.
-- This requires more flops, but we don't want signals
-- to memory to be glitchy.
  constant IDLE   : std_logic_vector(8 downto 0) := "000000000";
  constant RINST  : std_logic_vector(8 downto 0) := "000000001";
  constant RHORI1 : std_logic_vector(8 downto 0) := "000000010";
  constant RHDEAD : std_logic_vector(8 downto 0) := "000000110";
  constant RHORI2 : std_logic_vector(8 downto 0) := "000000100";
  constant WHORI1 : std_logic_vector(8 downto 0) := "000001000";
  constant WHDEAD : std_logic_vector(8 downto 0) := "000011000";
```

```
  constant WHORI2 : std_logic_vector(8 downto 0) := "000010000";
  constant RVERT1 : std_logic_vector(8 downto 0) := "000100000";
  constant RVDEAD : std_logic_vector(8 downto 0) := "001100000";
  constant RVERT2 : std_logic_vector(8 downto 0) := "001000000";
  constant WVERT1 : std_logic_vector(8 downto 0) := "010000000";
  constant WVDEAD : std_logic_vector(8 downto 0) := "110000000";
  constant WVERT2 : std_logic_vector(8 downto 0) := "100000000";
component cntl_sw_sm
   port ( clk, reset     : in std_logic;
          -- control signals from North Switch
         rd_inst        : in std_logic;
         rd_hori        : in std_logic;
         wr_hori        : in std_logic;
         rd_vert        : in std_logic;
         wr_vert        : in std_logic;

         -- state machine output
         state          : buffer std_logic_vector(8 downto 0)
       );
end component;
end cntlswsm_package;
library IEEE;

use IEEE.STD_Logic_1164.all;
use work.cntlswsm_package.all;
entity cntl_sw_sm is
   port ( clk, reset     : in std_logic;
          -- control signals from North Switch
         rd_inst        : in std_logic;
         rd_hori        : in std_logic;
         wr_hori        : in std_logic;
         rd_vert        : in std_logic;
         wr_vert        : in std_logic;

         -- state machine output
         state          : buffer std_logic_vector(8 downto 0)
       );
end entity cntl_sw_sm;
architecture rtl of cntl_sw_sm is
   signal next_state   : std_logic_vector(8 downto 0);

   signal hold_rd_inst : std_logic;
begin
```

```
– The matrix can give another INST_REQ (rd_inst) the clock after
– WDATA_RDY, but the memory interface cannot recognize an INST_REQ
until
– the write to memory has completed, so therefore must detect this case and
– hold the INST_REQ until it can be recognized (after the write to memory
has
– completed.
HOLD: process ( clk, reset )
begin
      if (reset = '1') then
        hold_rd_inst <= '0';
      elsif rising_edge(clk) then
        if (rd_inst='1' and state /= IDLE) then – if we see a rd_inst, but we are
not in IDLE state
          hold_rd_inst <= '1';
        elsif (state=RINST) then          – clear when we have recognized the
rd_hori
          hold_rd_inst <= '0';
        end if;
      end if;
end process HOLD;
– flip flops of state machine
SEQ: process ( clk, reset )
begin
     if (reset = '1') then
       state <= IDLE;
     elsif rising_edge(clk) then
       state <= next_state;
     end if;
end process SEQ;
– next state combinational logic
COMB: process (state, rd_inst, hold_rd_inst, rd_hori, rd_vert, wr_vert, wr_hori)
begin
   case (state) is
      when IDLE =>
         if (rd_inst = '1' or hold_rd_inst = '1') then
            next_state <= RINST;
         elsif (rd_hori = '1') then
            next_state <= RHORI1;
         elsif (rd_vert = '1') then
            next_state <= RVERT1;
         elsif (wr_vert = '1') then
            next_state <= WVERT1;
         elsif (wr_hori = '1') then
            next_state <= WHORI1;
```

```
        else
           next_state <= IDLE;
        end if;
      when RINST  =>  – read a 16 bit instruction

        next_state <= IDLE;
      when RHORI1 =>  – read the lower 32 bits of horizontal

        next_state <= RHDEAD;
     when RHDEAD  =>  – dead clock in between the two reads
        next_state <= RHORI2;
     when RHORI2 =>  – read the upper 32 bits of horizontal
        next_state <= IDLE;
      when WHORI1 =>  – write the lower 32 bits of horizontal

        next_state <= WHDEAD;
     when WHDEAD  =>  – dead clock in between the two writes
        next_state <= WHORI2;
     when WHORI2 =>  – write the upper 32 bits of horizontal
        next_state <= IDLE;
      when RVERT1   =>  – read the lower 32 bits of vertical

        next_state <= RVDEAD;
     when RVDEAD  =>  – dead clock in between the two reads
        next_state <= RVERT2;
     when RVERT2   =>  – read the upper 32 bits of vertical
        next_state <= IDLE;
      when WVERT1   =>  – write the lower 32 bits of vertical

        next_state <= WVDEAD;
     when WVDEAD  =>  – dead clock in between the two writes
        next_state <= WVERT2;
     when WVERT2   =>  – write the upper 32 bits of vertical
        next_state <= IDLE;
      when others   =>

        next_state <= IDLE;
    end case;

  end process COMB;
end architecture rtl;
```

Address Select Logic

```
library IEEE;
use IEEE.std_logic_1164.all;
–*********************************************
```

```
– toggle FF
_*********************************************
entity tff is
        port(
                t_en: in std_logic;–enable ff
                t_rst:in std_logic;–reset ff
                t_in: in std_logic;–input
                t_out:buffer std_logic);
end tff;
architecture tff_rtl of tff is
begin
process(t_in,t_en,t_rst)
begin
if t_rst='1' then
        t_out<='0';
elsif( t_in'event and t_in='1') then
        if t_en='1' then
                t_out<= not t_out;
        end if;
end if;
end process;
end tff_rtl;
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.std_logic_arith.all;
entity cnt6 is
        port(
                clk,crst: in std_logic;
                en: in std_logic;
                t_o: buffer std_logic_vector (0 to 5));
end cnt6;
architecture cnt6_rtl of cnt6 is
component tff
        port(
                t_en: in std_logic;
                t_rst:in std_logic;
                t_in: in std_logic;
                t_out:buffer std_logic);
end component;
signal en1,en2,en3,en4,en5: std_logic;
begin
u1: tff port map(en,crst,clk,t_o(0));
en1<=en and t_o(0);
u2: tff port map(en1,crst,clk,t_o(1));
en2<=en and t_o(0) and t_o(1);
```

```
u3: tff port map(en2,crst,clk,t_o(2));
en3<=en and t_o(0) and t_o(1) and t_o(2);
u4: tff port map(en3,crst,clk,t_o(3));
en4<=en and t_o(0) and t_o(1) and t_o(2) and t_o(3);
u5: tff port map(en4,crst,clk,t_o(4));
en5<=en and t_o(0) and t_o(1) and t_o(2) and t_o(3) and t_o(4);
u6: tff port map(en5,crst,clk,t_o(5));
end cnt6_rtl;
```

Appendix D - Script Used For Testing MEM_INTF

test1.cmd

```
— used to stimulate mem_intf design for
— ”SIMD Processor for Rough Set Machine”
restart    — start at poweron

— define multi-bit signals

vector MBUSN      MBUSN[63:0]
vector RAMDATAN  RAMDATAN[31:0]
vector RAM_ADDRN RAM_ADDRN[19:0]
vector MBUSE      MBUSE[63:0]

vector RAMDATAE  RAMDATAE[31:0]
vector RAM_ADDRE RAM_ADDRE[19:0]
vector CNTL_NE_1_STATE      CNTL_NE_1_STATE[8:0]

vector CNTL_NE_1_INST_PTR  CNTL_NE_1_INST_PTR[19:0]
vector CNTL_NE_1_HDATA_PTR CNTL_NE_1_HDATA_PTR[19:0]
— define which signals to show waveforms of

— North East Control signals
watch CLK
watch RESET
watch INST_REQ
watch INST_RDY
watch WDATA_RDY
watch RDATA_RDY
watch CNTL_NE_1_STATE
watch CNTL_NE_1_INST_PTR
watch CNTL_NE_1_HDATA_PTR
— North Switch signals

watch MBUSN
watch RAMDATAN
watch RAM_ADDRN
watch RAM_READN_N
watch RAM_WRITEN_N
watch RAM_DIS_HIGHN_N
watch RAM_DIS_LOWN_N
— South Switch signals

watch MBUSE
watch RAMDATAE
watch RAM_ADDRE
watch RAM_READE_N
watch RAM_WRITEE_N
```

watch RAM_DIS_HIGHE_N
watch RAM_DIS_LOWE_N
step 10ns

— define stimulus

clock clk 0 0 0 1 1 1
assign RESET 0

assign INST_REQ 0
assign WDATA_RDY 0
sim 10ns

— do a RESET

sim 120ns
assign RESET 1
sim 60ns

assign RESET 0
— INST_REQ from matrix (this gets a A000 = load horizontal)

sim 240ns
assign INST_REQ 1
sim 30ns

assign RAMDATAN 9000A000″
sim 30ns

assign INST_REQ 0
sim 30ns

release RAMDATAN
sim 60ns

assign RAMDATAN 22221111″
sim 60ns
release RAMDATAN
sim 60ns
assign RAMDATAN 44443333″
sim 60ns
release RAMDATAN
— INST_REQ from matrix (this gets a 9000 = load vertical)

sim 150ns
assign INST_REQ 1
sim 30ns
assign RAMDATAN 9000A000″
sim 30ns
assign INST_REQ 0
sim 30ns
release RAMDATAN

sim 60ns

assign RAMDATAE 66665555″
sim 60ns
release RAMDATAE
sim 60ns
assign RAMDATAE 88887777″
sim 60ns
release RAMDATAE
— INST_REQ from matrix (this gets a E000 = store horizontal)

sim 150ns
assign INST_REQ 1
sim 30ns
assign RAMDATAN D000E000″
sim 30ns
assign INST_REQ 0
sim 30ns
release RAMDATAN
sim 120ns

assign WDATA_RDY 1
assign MBUSN BBBBAAAA99998888″
sim 60ns
assign WDATA_RDY 0
release MBUSN
— INST_REQ from matrix (this gets a D000 = store vertical)

assign INST_REQ 1
sim 60ns
assign INST_REQ 0
sim 30ns
sim 150 ns

assign RAMDATAN D000E000″
sim 60ns
release RAMDATAN
sim 180ns

assign WDATA_RDY 1
assign MBUSE FFFFEEEEDDDDCCCC″
sim 60ns
assign WDATA_RDY 0
release MBUSE
sim 300ns