THE PSU-EUT RECONFIGURABLE COMPUTING PROJECT: THE UNIVERSAL LOGIC MACHINE

Marek Perkowski, and Lech Jozwiak

Portland State University,
Dept. of Electr. Engn.
Portland, Oregon 97207
Tel: 503-725-5411, Fax: 503-725-4882
e-mail: mperkows@ee.pdx.edu

Faculty of Electronics Engineering,
Eindhoven University of Technology,
5600 MB Eindhoven, The Netherlands
e-mail: lech@eb.ele.tue.nl

March 23, 1997

Contents

List of Figures

Example 3.1. Intersection of cubes A = $X^{\{0,1,2\}}\ Y^{\{1,2,3\}}$ = [1110000-0111000] and B = $X^{\{0,1,3,6\}}\ Y^{\{0,2,5\}}$ = [1101001-1010010] is cube C = $X^{\{0,1\}}\ Y^2$ = [1100000-0010000].

The second group of operations, the *complex combinational operations* produce one cube whose literals are calculated by conditional operations on the literals of the operands. For instance, the *prime operation* (used in Exclusive Sum of Products synthesis) is defined as follows:

$A \text{ prime } B = {X_1}^{A_1} \cdots {X_j}^{A_j \cup B_j}\ {X_{i-1}}^{A_{i-1}}\ {X_i}^{A_i \cup B_i}$
${X_{i+1}}^{A_{i+1}} \cdots {X_k}^{A_k \cup B_k}\ {X_N}^{A_N}$

where $A_k \cup B_k$ is calculated for all those variables X_k for which the relation $A_k \cap B_k \neq \emptyset$ is satisfied.

Example 3.2. Prime of binary cubes A = $X^0\ Y^1\ Z^1\ V^1$ = [10-01-01-01] and B = $X^1\ Y^0\ Z^0\ V^{0,1}$ = [01-10-10-11] is cube C = $X^0\ Y^1\ Z^1\ V^{0,1}$ = [10-01-01-11].

Example 3.3. XXX1 # 111X = { 0XX1, X0X1, XX01}. Let us observe that simple symbol 0, shifted from left to right in the above cubes, corresponds to all values of i for which relation $(\neg(B_i \supseteq A_i)\ ) = 1$ is satisfied.

Examples.

For binary logic: $x^0 = \bar{x}$, $x^1 = x$.

For four-valued input logic: $X^{\{0,1,2\}} = 1$ if X $\times$ {0,1,2}, which means, $X^{\{0,1,2\}} = 1$ if (X=0) or (X=1) or (X=2). Otherwise $X^{\{0,1,2\}} = 0$.

For (X=3), $X^{\{0,1,2\}} = 0$.

A product of literals, ${X_1}^{S_1}\ {X_2}^{S_2}\ \ldots\ {X_n}^{S_N}$, is referred to as a *product term* (also called *term* or *product* for short). Such term is represented as a cube. A product term that includes literals for all function variables X_1, X_2, ..., X_N is called a *full term.* Any literal of the form ${X_i}^{P_i}$ is identically equal 1. Hence, we often write ${X_i}^{P_i}\ {X_j}^{S_j}$ as ${X_j}^{S_j}$. A sum of products is denoted as a *sum-of-products expression (SOPE)* while a product of sums is called a *product-of-sums expression (POSE).* An EXOR of products will be called a *Exclusive Sum of Products Form* (ESOP). A product of EXORs will be called a *Product of Exclusive Sums expression* (POES). SOPE, POSE, ESOP and POES are all represented as *arrays of cubes.* One uses also products of SOPEs (PSOPEs) as the *Generalized Propositional Formulas* from [**?**]. They are represented as *arrays of arrays of cubes.*

0.1 EXAMPLE OF STATE TRANSITIONS

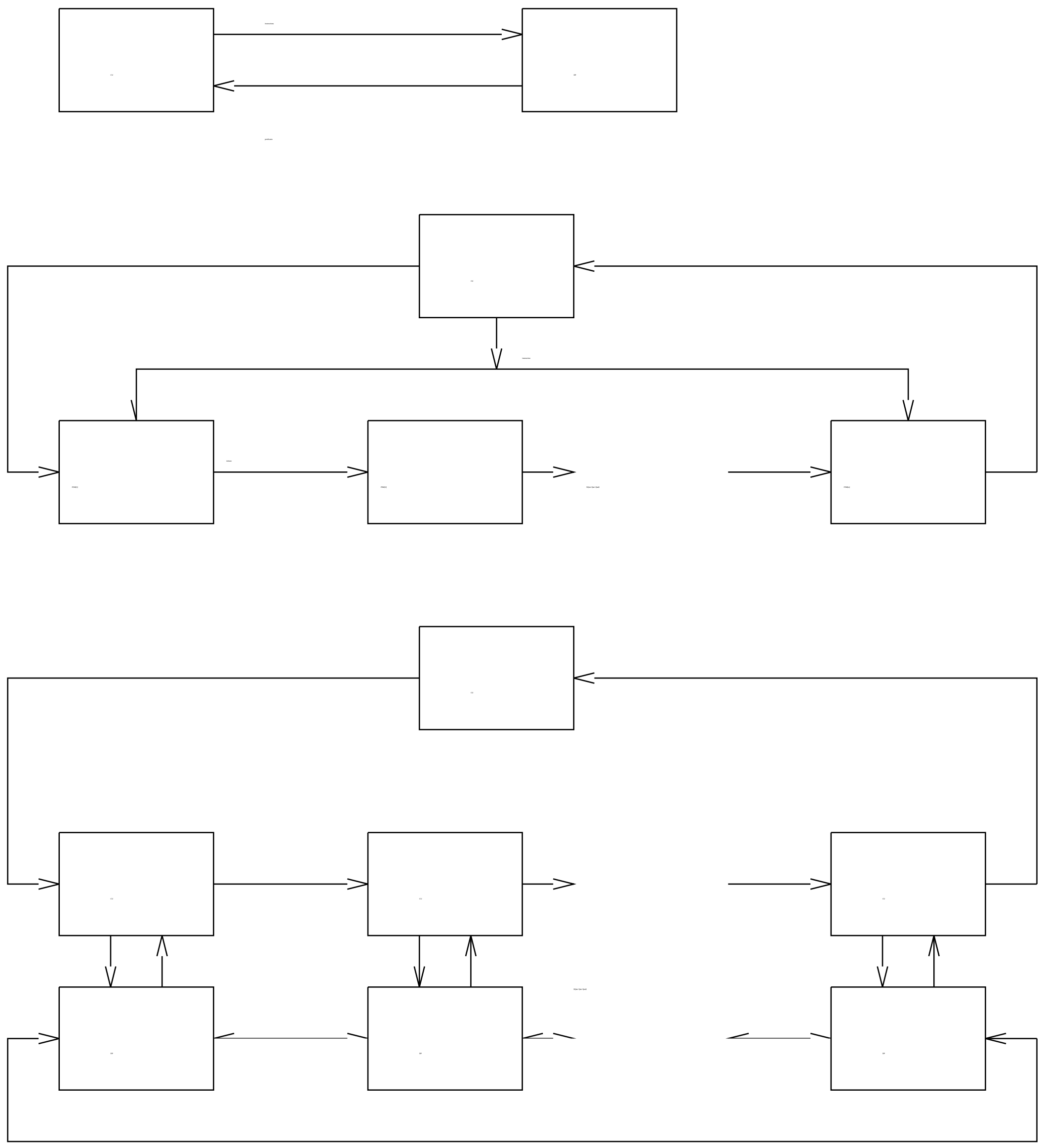


Figure 0.1: Example of Pseudo Generalized Kronecker Tree .so Figg1.pic

before(bef)	active(act)	after(aft)	before(bef)	active(act)	after(aft)
sharp	$A\#B$	$\neg B \subseteq A$	A	$\neg B \cap A$	A
disjoint sharp	$A\#dB$	$\neg = (B \subseteq A)$	A	$\neg B \cap A$	$A \cap B$
asymmetric consensus	$A * aB$	$\neg(B \subseteq A)$	$A \cap B$	$A \cup B$	$A \cap B$
symmetric consensus	$A * sB$	1	$A \cap B$	$A \cup B$	$A \cap B$
crosslink	$A \cap B$	$A \cap B = \emptyset$	A	$A \cup B$	B

Table 0.1: *Table 5.1. Sequential Cube Calculus Operations.*

operation	before	active	after	relation
sharp	$\neg Bj[i] \cap Aj[i]$	$Aj[i]$	$\neg Bj[i] \cap Aj[i]$	$Aj[i]$
hline disjoint sharp	$\neg Bj[i] \cap Aj[i]$	$Aj[i]$	$\neg Bj[i] \cap Aj[i]$	$Aj[i] \cap Bj[i]$
asymmetric consensus	$\neg Bj[i] \cap Aj[i]$	$Aj[i] \cap Bj[i]$	$Aj[i] + Bj[i]$	$Aj[i] \cap Bj[i]$
symmetric consensus	1	$Aj[i] \cap Bj[i]$	$Aj[i] + Bj[i]$	$Aj[i] \cap Bj[i]$
crosslink	$\neg Aj[i] \cup \neg Bj[i]$	$Aj[i]$	$Aj[i] + Bj[i]$	$Bj[i]$

Table 0.2: *Table 5.2. Sequential Cube Calculus Operations on bits.*

0.2 THE IMPLEMENTATION OF THE CONTROL UNIT OF THE CCM2

The task of the Control Unit CU is to provide the ILU with the necessary control signals. The operations being directly provided by the instruction from the host computer do not have to be determined by the CU. The only exception is the setting of the relation in the *pipeline mode.*

Therefore the CU has only to perform the algorithms for sequential cube calculus operations and provide signals to the ILU according to the different modes.

0.2.1 THE IR SIGNALS AND THEIR MEANING

All signals described below are without the connections with the signals coming from another CCM while using the *chain mode.* Therefore all signals from the CU have to be OR-ed with the matching input pin that helds the signal from another CCM. This means that all input and output signals from the CU have to be available on pins of the CCM.

The instructions coming from the host computer are stored in the instruction register IR. Instructions concerning the Control Unit are:

$IR[0]$: sequential or combinational cube calculus operation

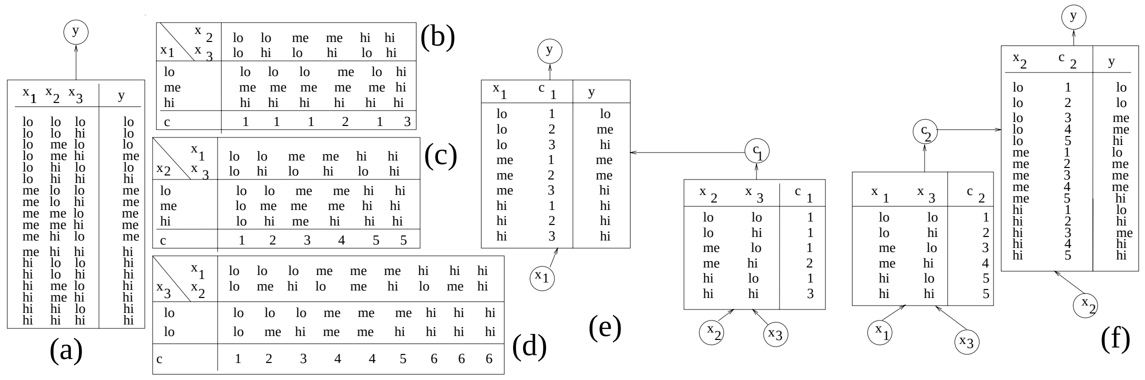


Figure 0.2: *The three architectural models of a general-purpose computer.*

$IR[0] = 0$ for sequential operation.
$IR[0] = 1$ for combinational operation.

$IR[1]$: chain or stand-alone mode.
$IR[1] = 0$ for stand-alone mode.
$IR[1] = 1$ for chain mode.

$IR[2]$: counter mode
$IR[2] = 0$ for counter mode $cnt0$ which means counting in all ITs (Equation (10.7)).
$IR[2] = 1$ for counter mode $cnt1$ which means counting all the variables that are satisfied (Equation (10.8)).

$IR[3]$: AND/OR type relation for the sequential part of a Cube Calculus Operation (RELATION).
$IR[3] = 0$ for OR_type relation for RELATION.
$IR[3] = 1$ for AND_type relation for RELATION.

$IR[4]$: AND/OR type relation for the condition (the combinational part - PRERELATION) of a sequential cube calculus operation.
$IR[4] = 0$ for OR_type relation for PRERELATION.
$IR[4] = 1$ for AND_type relation for PRERELATION.

$IR[5, 6]$: determines the kind of sequential algorithm

$IR[7]$: prime operation, not for control unit

See also Table ?? how sequential CC operations are specified.

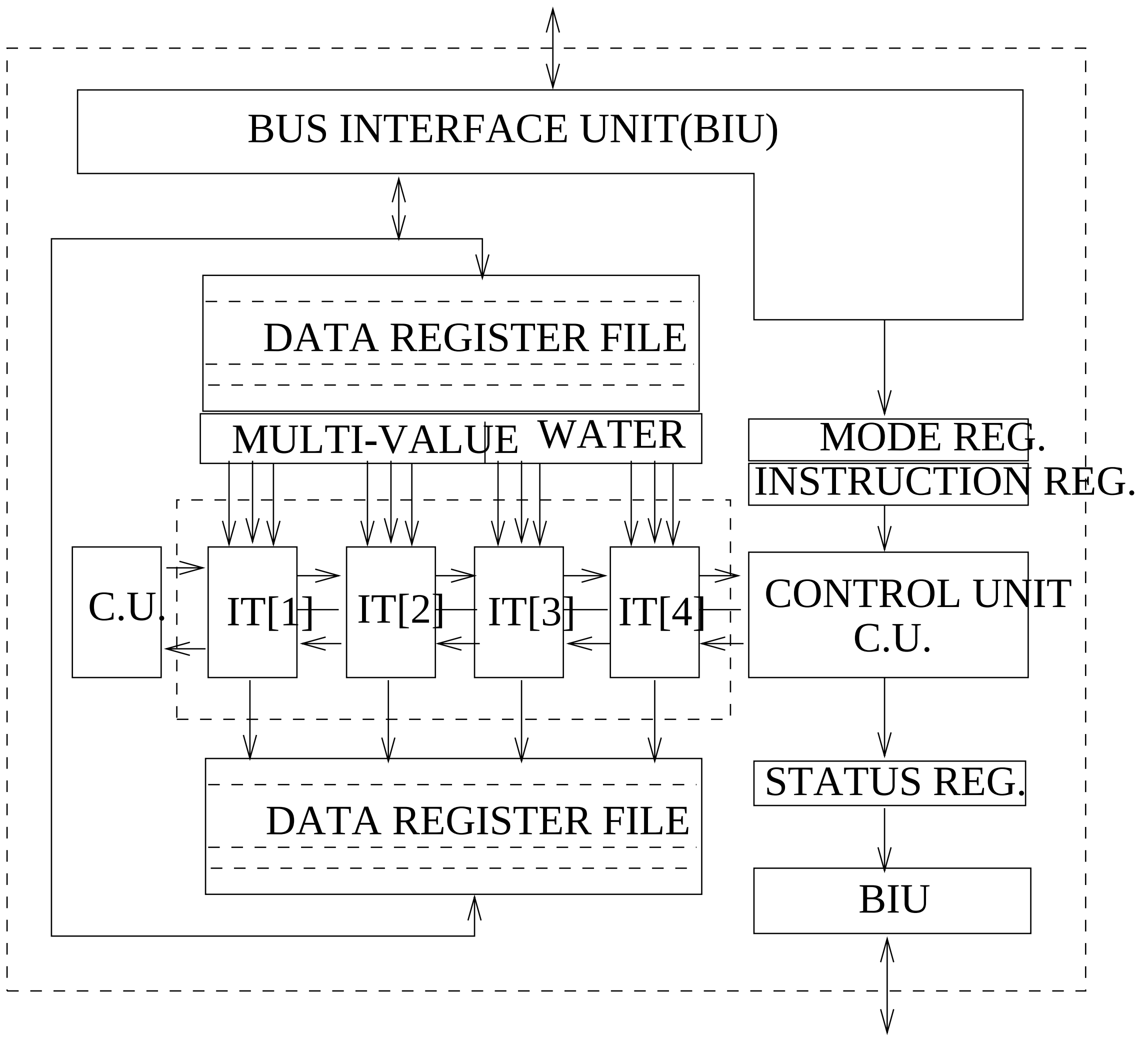
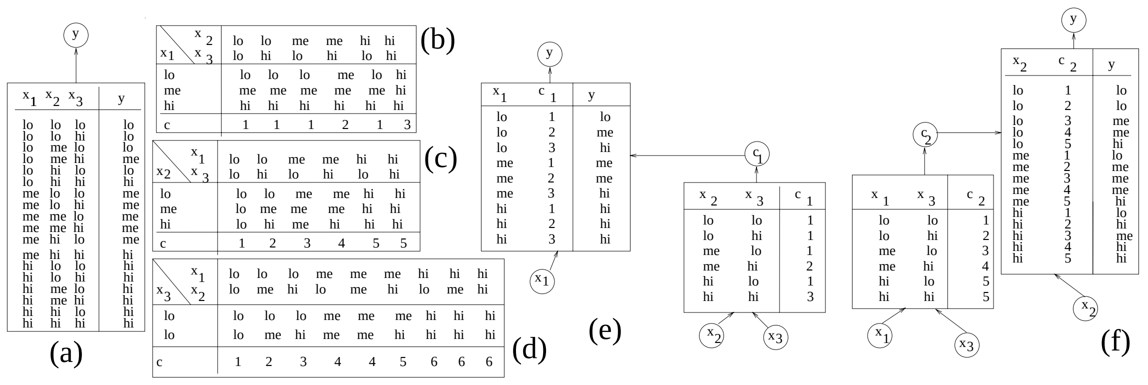


Figure 0.3: Example of Pseudo Generalized Kronecker Tree .so ccm_block.pic



x_1	x_2	x_3	y
lo	lo	lo	lo
lo	lo	hi	lo
lo	me	lo	lo
lo	me	hi	me
lo	hi	lo	lo
lo	hi	hi	hi
me	lo	lo	me
me	lo	hi	me
me	me	lo	me
me	me	hi	me
me	hi	lo	me
me	hi	hi	hi
hi	lo	lo	hi
hi	lo	hi	hi
hi	me	lo	hi
hi	me	hi	hi
hi	hi	lo	hi
hi	hi	hi	hi

(a)

x_1 \ x_2 x_3	lo lo	lo hi	me lo	me hi	hi lo	hi hi
lo	lo	lo	lo	me	lo	hi
me	me	me	me	me	me	hi
hi	hi	hi	hi	hi	hi	hi
c	1	1	1	2	1	3

(b)

x_2 \ x_1 x_3	lo lo	lo hi	me lo	me hi	hi lo	hi hi
lo	lo	lo	me	me	hi	hi
me	lo	me	me	me	hi	hi
hi	lo	hi	me	hi	hi	hi
c	1	2	3	4	5	5

(c)

x_3 \ x_1 x_2	lo lo	lo me	lo hi	me lo	me me	me hi	hi lo	hi me	hi hi
lo	lo	lo	lo	me	me	me	hi	hi	hi
lo	lo	me	hi	me	me	hi	hi	hi	hi
c	1	2	3	4	4	5	6	6	6

(d)

x_1	c_1	y
lo	1	lo
lo	2	me
lo	3	hi
me	1	me
me	2	me
me	3	hi
hi	1	hi
hi	2	hi
hi	3	hi

x_2	x_3	c_1
lo	lo	1
lo	hi	1
me	lo	1
me	hi	2
hi	lo	1
hi	hi	3

(e)

x_1	x_3	c_2
lo	lo	1
lo	hi	2
me	lo	3
me	hi	4
hi	lo	5
hi	hi	5

x_2	c_2	y
lo	1	lo
lo	2	lo
lo	3	me
lo	4	me
lo	5	hi
me	1	lo
me	2	me
me	3	me
me	4	me
me	5	hi
hi	1	lo
hi	2	hi
hi	3	me
hi	4	hi
hi	5	hi

(f)

Figure 0.4: *Fig. 7.1. The block diagram of CCM architecture.*

There are two different kinds of instructions for the CU. One type determines if the CU has to perform an algorithm for a sequential cube calculus operations (Table **??**), where the cube calculus operations given only represent the type of performed algorithm). The second determines the different operations modes of the CU:

- stand_alone mode :

the CCM is used either as a stand alone CCM or is the rightmost CCM in a chain of CCM's.

- chain mode :

several CCM's are connected in a chain, therefore only the rightmost CU has to provide the signals for all CCM's. All CU's except this rightmost have to be overridden.

- pipeline :

if several CCM's are ordered in a certain structure for pipelining, the CU has to check if there are contradictory cubes as input before performing the chosen cube calculus operation. This flag is set in the BIU register.

- combinational :

the CCM has to perform a combinational cube calculus operation. Therefore it has only to generate the *VALID* and *BUSY* signals.

- sequential :

the CU performs the algorithm for the chosen sequential cube calculus

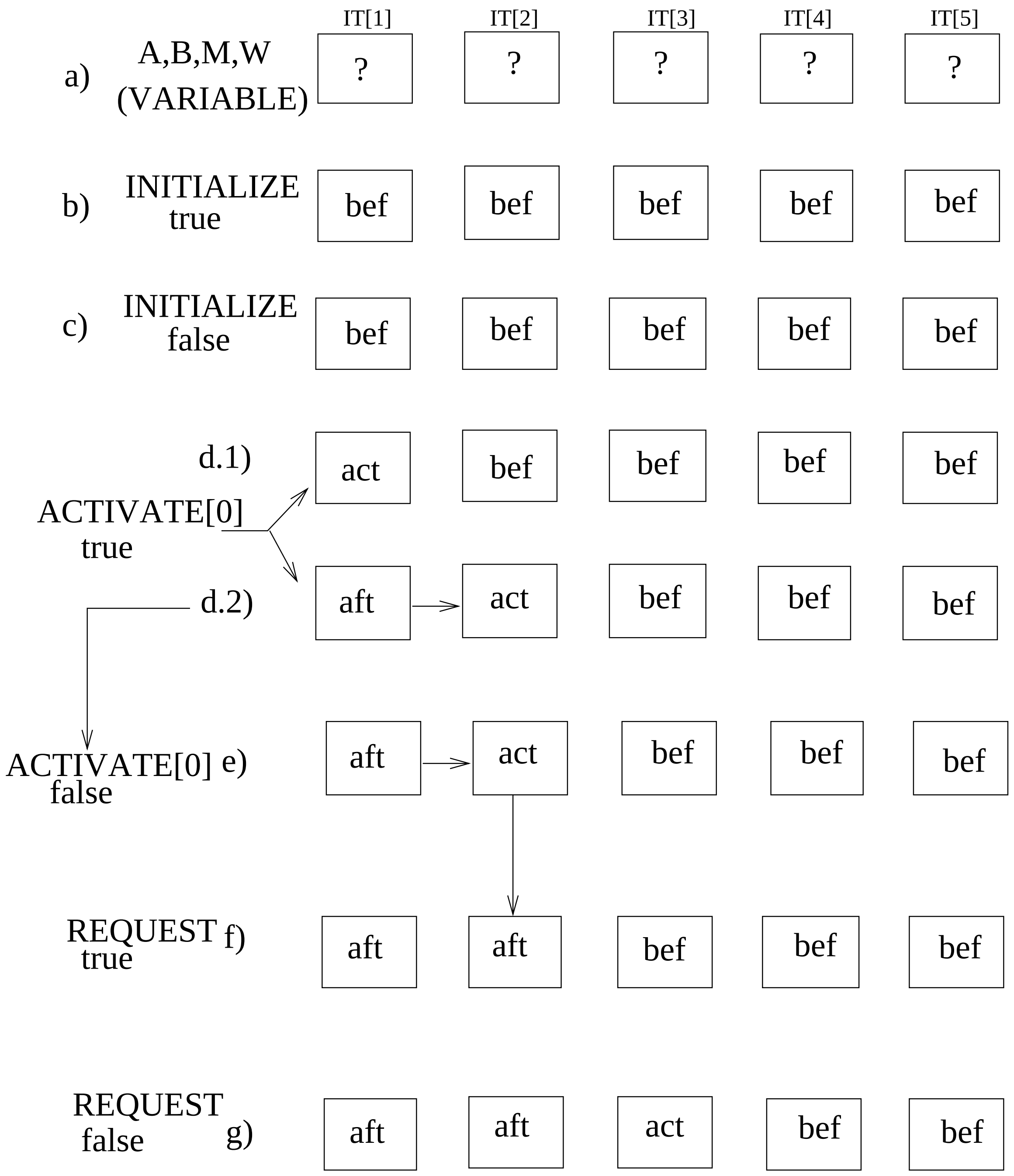


Figure 0.5: *Example of Pseudo Generalized Kronecker Tree .so AFSM_eg2.pic*

.

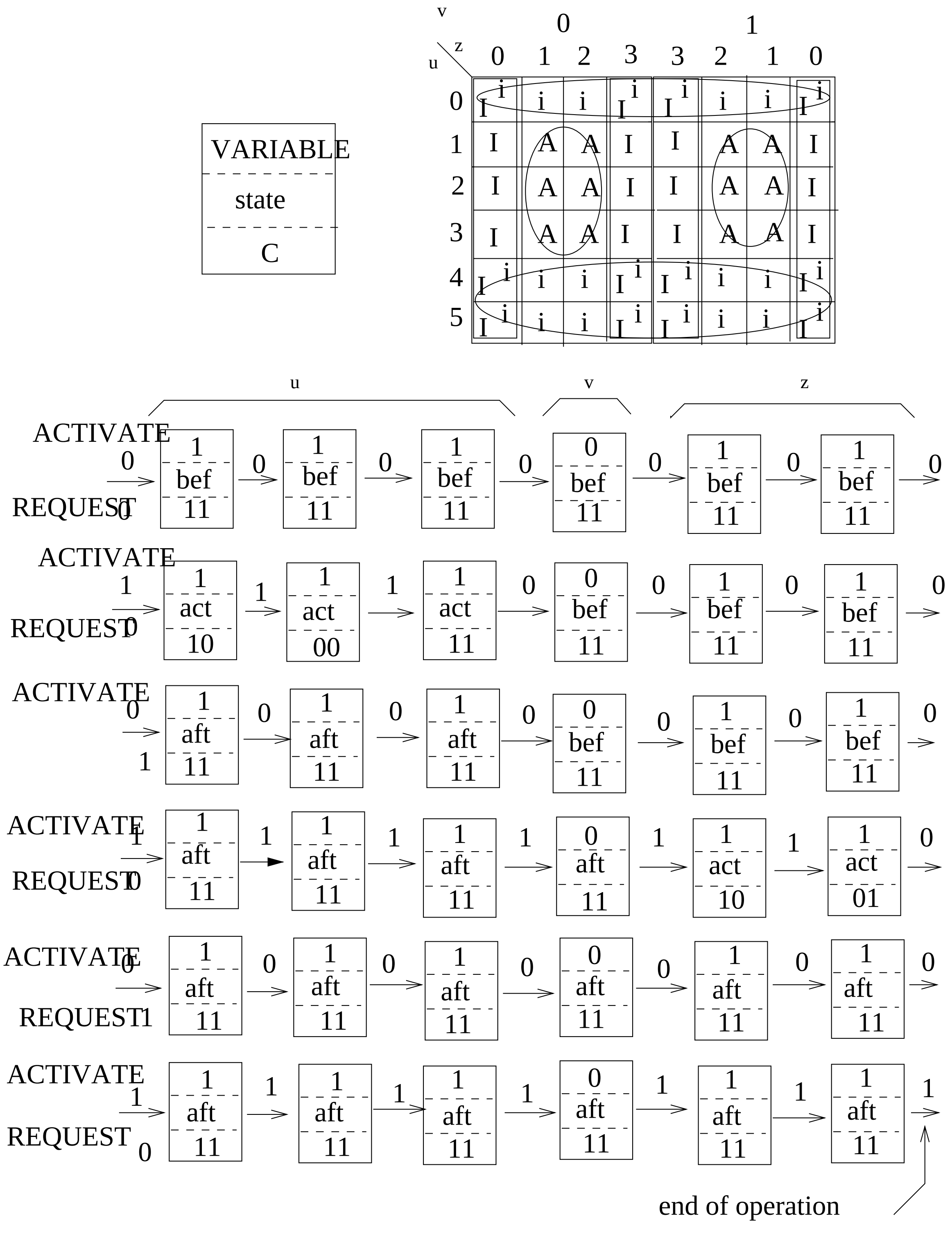


Figure 0.6: *.so inversion2.pic*

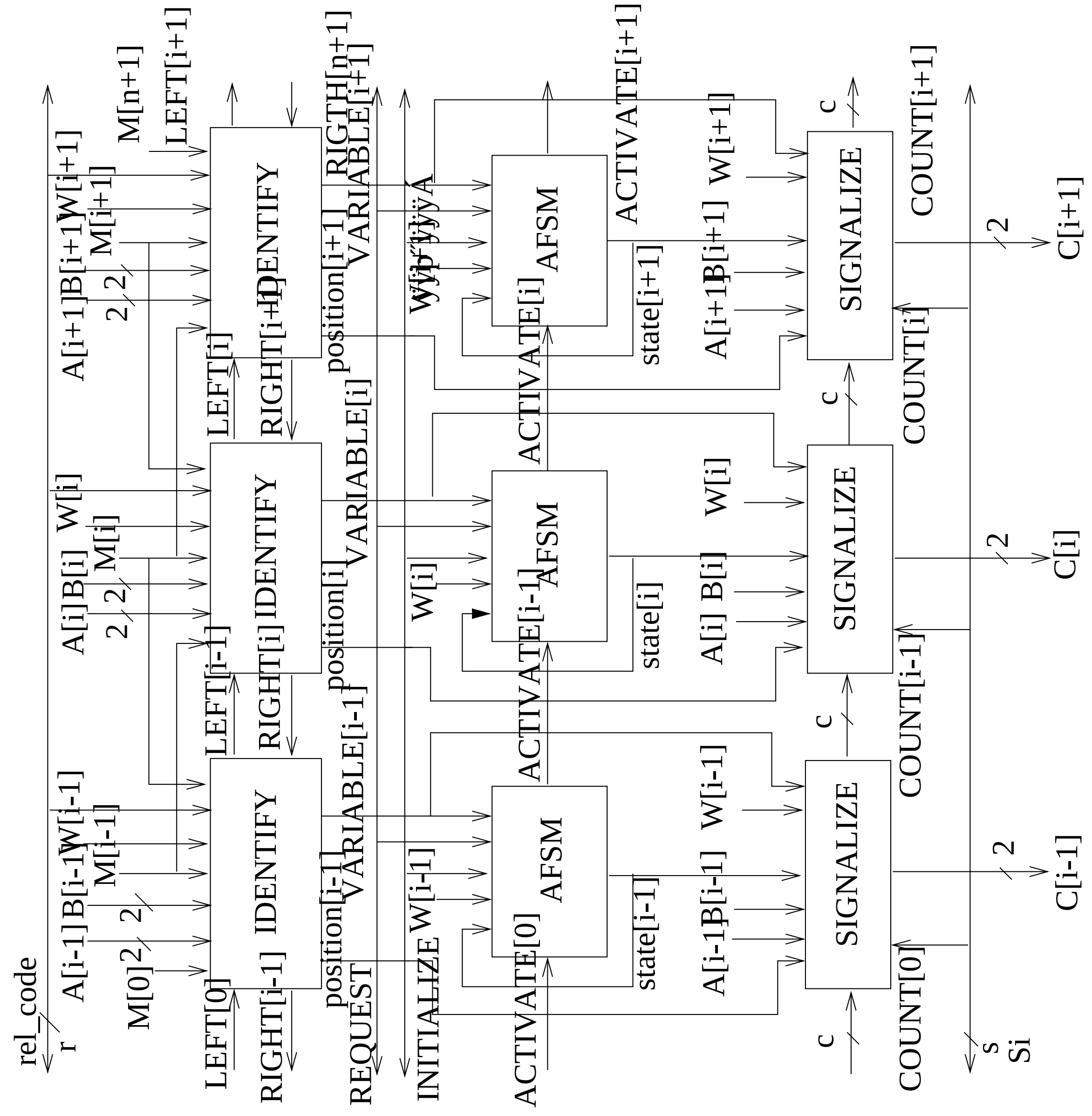


Figure 0.7: .so module.pic

Figure 0.8: *.so figgg6.pic*

Figure 0.9: *.so figg7.pic*

Figure 0.10: *.so fig8.pic*

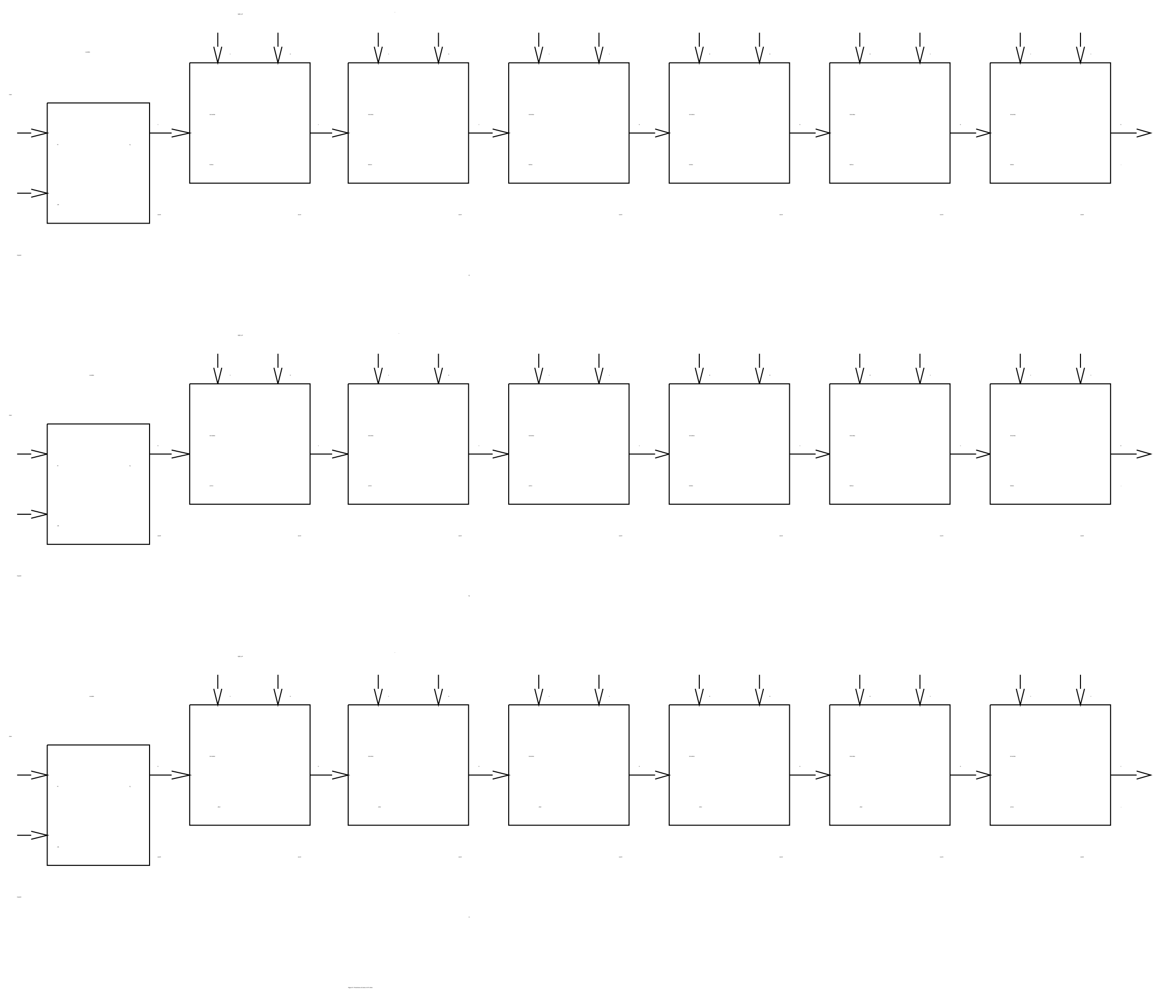


Figure 0.11: *.so figg9.pic*

		U[01]	U[23]	U[45]	V[01]	Z[01]	Z[23]	CU=IT[n+1]
1	i	0	1	2	3	4	5	6
7								
2	IT[i]	IT[1]	IT[2]	IT[3]	IT[4]	IT[5]	IT[6]	
3	A[i]	11	01	00	11	01	10	
4	RELATION[i]	0	1	1	0	1	1	$(A\neg = X)$
hline 5	CARRY[i]	1=¿	0	1	1	0	1	1
⇒								
6	CONF[i] ¡=	1	1	1	0	1	1	¡=0
7	VARIABLE[i]	1	1	1	0	1	1	$(A\neg = X)$
8	COUNT[i]	0=¿	0	0	1	1	1	2
=¿	(2 resultant cubes)							
9	s[i]	00	10	11	11	11	11	
10	S[i]	11	11	11	11	10a	01	

Table 0.3: *Table 9.1.*

$IR[5]$	$IR[6]$	sequential operation
0	0	only main loop
0	1	sharp
1	0	consensus
1	1	crosslink

Table 0.4: *Table 11.1.*

operation.

0.3 DESCRIPTION OF THE SEQUENTIAL CUBE CALCULUS OPERATIONS

This section will describe only the basic instructions. Because of the freedom of the use of the instruction register one can generate a lot of additional operations.

Each of this sequential operations has the same *main algorithm* (described in the sequel) which generates the signals to generate all output cubes for the operation.

To perform the operations shown in Table **??** there are different *preconditions* for each function to be fulfilled.

The main algorithm has to perform the generation of the resultant cubes. The number of the resultant cubes is determined by the number

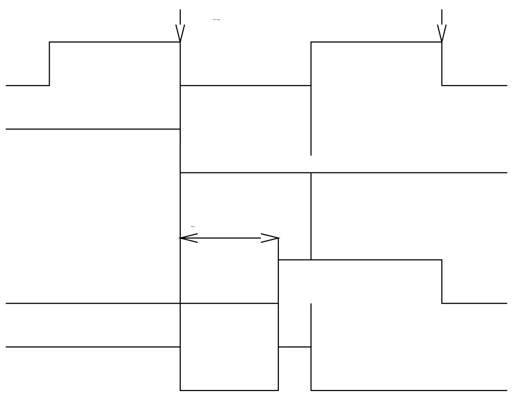


Figure 0.12: *.so f5.pic*

RELATION0	RELATION1	chosen relation
0	0	relation
0	1	prerelation
1	0	A = 0
1	1	B = 0

Table 0.5: *Table 11.2.*

of literals which match a certain relation - the specific literals. These relations - *rel* are shown in Table 8.3 (page 13).

The relations that have to be performed on the literals are determined by their position to the **specific literals.** According to their position left *(after)* or right *(before)* of the **specific literals** itself *(active)* the performed operation has to be determined. This output operations are also shown in Table 8.3.

Doing this for each **specific literal** one obtains as many resultant cubes as there are **specific literals.** The coding which uses signals *RELATION0* and *RELATION1* is shown in Table **??**.

0.4 THE GENERAL ALGORITHM

The general algorithm contains the *main algorithm* which generates the signals to generate all solution cubes and the part to perform certain operations according to the chosen type of algorithm. The possible four algorithms are described in Table **??**.

For example, if $IR[5,6]$ = [1,0] then the *PRERELATION* with the

Cube	Prerelation	Preoperation	Relation	Condition	Operation
Operations	bef	act	aft		
Intersection	$A_i \cap B_i = 0$	count0	$A_i \cap B_i$		
section				$\neg count0$	C=0
Supercube					$A_i \cup B_i$
Prime					
hline Crosslink			$A_i \subseteq B_i = 0$	count0	A_i
$A_i \cup B_i$	B_i				
				count0	C=0
	heightSharp	$A_i \cap B_i = 0$	$\neg count0$	$A_i \cap \neg B_i$	$\neg count0$
A_i	$A_i \cap \neg B_i$	A_i			
		C=A		count0	C=0
Disjoint	$A_i \cap B_i = 0$	$\neg count0$	$A_i \cap \neg B_i$	$\neg count0$	A_i
$A_i \cap \neg B_i$	$A_i \cap B_i$ Sharp		C=A		count0
C=0					
Symmetric	$A_i \cap B_i = 0$	count0,$A_i \cap B_i$	1		$A_i \cap B_i$
$A_i \cup B_i$	$A_i \cap B_i$				
Consensus		count1,C=0			$A_i \cap B_i$
$A_i \cup B_i$	$A_i \cap B_i$				
Asymmetric	$A_i \cap B_i = 0$	count0,$A_i \cap B_i$	$A_i \cap \neg B_i$	$\neg count0$	$A_i \cap B_i$
$A_i \cup \neg B_i$	$A_i \cap B_i$				
Consensus		count1,C=0		count0	C=0

Table 0.6: *Table 11.3.*

and/or_type selected by *AND_OR* from the instruction register will be performed at first. The result of the relation will be checked according to the signal *COUNT1* which means that the relation was *true* for more than 1 literal. If the condition is *true* than the operation determined by *BEFORE, ACTIVE, AFTER,* and *RELATION* with *AND_OR* selected type given also in the instruction register will be performed. If the condition is not fulfilled then the operation determined in the *PREOPERATION* field will be performed.

0.5 INPUT AND OUTPUT SIGNALS OF THE CU

FLAGS All flags are determined by two states: 1 or 0. To set and reset these flags two different signals are necessary.

VALID : result in output register is valid

BUSY : CU is busy, performing an algorithm

GO : signal from BIU to CU to start chosen instruction

DEL_START : tell delay counter to start

DEL_END : signal from delay counter that finished

0.6 OUTPUT SIGNALS OF THE CU

0.6.1 STABLE OUTPUT SIGNALS OF THE CU

The CU has to provide the ILU and the Bus Interface Unit (BIU) with stable control signals. Stable signals that have to be set and reset from the CU are described below. A flip-flop to keep a stable signal is shown in Figure 9.10.

Figure 11.1.

A flip-flop to keep a stable signal for the CU.

One can observe that for setting the latch the signal *SET...* and the signal *VALUE...* have to be "1". To reset the latch the signal *SET...* has to be "0", and the signal *VALUE* has to be "1".

Stable Signals (SET ..., VALUE ...):

BUSY

The signal *BUSY* is set while the CU is "busy", performing an algorithm.

SET_BUSY is a signal from CU to flip-flop @@@

BUSY_VALUE is a signal from flip-flop to ?@@@

OPERATION

If the conditions for a sequential cube calculus operation are nor fulfilled, a combinational cube calculus operation can be performed. To distinguish between these two types of cube calculus operations the signal *OPERATION* is used.

OPERATION = 1 when operation is to be performed.

OPERATION = 0 when preoperation has to be performed.

It determines if *PREOPERATION* or *BEFORE* field has to be used.

REL1, REL2 The two signals *REL1* and *REL2* determine the different types of relations on *A* and *B* that have to be checked. In Table **??** the code for these signals is shown:

The relation A=0 of the third row is 0011, the relation B=0 is described by 0101.

REL0	REL1	RELATION
0	0	RELATION
0	1	PRERELATION
1	0	A = 0
1	1	B = 0

Table 0.7: *Table 11.4.*

The values of the signals *PRERELATION[1:4]* and *RELATION[1:4]* are taken directly from the IR register. The relations A=0, and B=0 are used to determine the contradictory cubes in the pipeline mode.

Because *REL0,* and *REL1* are always set together, only one *SET* signal *SET_RELATION* is used.

AND_OR

The signal *AND_OR* is directly taken from the Instruction Register (IR). The bit IR[4] determines if the relation that has to be fulfilled for the sequential cube calculus operation is of the AND-type or OR-type. Bit IR[3] determines the type of the relation that has to be used to calculate the specific positions.

AND_OR = 1 when AND-type relation is checked.

0.7 THE SINGLE PULSE OUTPUT SIGNALS OF THE CU

Several signals have to be stable for only one clock pulse. Thus, they don't have to be kept in flip-flops.

The Pulse Signals are:

CLEAR

Resets the PSMs in the ILU.

NEXT0ILU

This signal is the initial *NEXT* from CU to ILU. It activates the Interlock Mechanism of the PSMs in the ILU.

REQUEST

The clock signal only for the PSMs in the ILU. Do not confuse with the global clock of the CCM2.

RESET_GO

Resets the *GO* signal (flip-flop) from the BIU.

SET_VALID

When a resultant cube is calculated by the ILU the *VALID* signal is set to 1.

VALID = 1 means "do not disturb the ILU and the CU".

DELAY_START

If a result of the iterative COUNT counter is necessary, a delay time has to be inserted to be sure that the result is stable. This signal starts the Delay Counter.

C_TO_OUT

The result of the ILU (denoted as combinational signal C) is copied to the Output Register.

RESET_OUTPUT

Resets the Output Register. This is done for contradictory cubes in the pipeline mode.

The following signals are not provided by the CU but necessary for the ILU. The signals *CARRY* and *COMPARE* do not have to be set by the CU, because they are only used inside of multivalued literals. The first and last bit of the *Water register W* being bits W_1 and W_{n+1}

@@or W_1 and W_n???@@

do not necessarily have to be generated from the CU. The four signals to determine the relation for the pipeline mode do not have to be generated from the CU. They can directly be generated by setting the matching input lines of the multiplexer to ground or Vdd. With this approach one more multiplexer is necessary. The signals *RELATION0* and *RELATION1* are still sufficient for the multiplexing of four different relation operations (*PRERELATION, RELATION, A=0,* and *B=0*).

CARRY : left signal that comes either form the next CCM or 'goes to ground'

COMPARE : similar to the

CARRY signal $W[-1]$ @@@

$W[n+1]$

0.8 INPUT SIGNALS TO THE CU

GO

Signal from the BIU (through a flip-flop) to start the CU with a selected instruction.

VALID

Signal from the BIU (through a flip-flop) if the resultant cube has been read from the host computer.

VALID = 1 when it has been succesfully read.

READYCU

Signal from the ILU to tell that a resultant cube has been calculated. This corresponds to the sum of all *READY[i]* signals.

NEXTI = NEXT16

Signal form the ILU to tell that the last resultant cube has been calculated. This is the last signal from the Interlock Mechanism.

COUNT0

This signal is equal 1 when the result of counter COUNT is = 0.

COUNT1

This signal is equal 1 when the result of counter COUNT is ¿ 1.

COUNT2

This signal is equal 1 when the result of counter COUNT is ¿ 2.

PIPELINE

The signal from BIU to indicate that the CU has to operate in a pipeline mode. *PIPELINE* = 1 for the pipeline mode.

DELAY_END

Signal from the Delay Counter to tell that the delay time is over. *DELAY_END* = 1 means that all signals are now stable and new action can be initialized by CU.

GLOBAL_RESET Signal from input pin to CCM chip. Resets everything, like a "reset" key in a personal computer.

0.9 THE ALGORITHMS OF THE CU

As mentioned in the Introduction the CU has two tasks:

1. providing the signals to generate sequential cube calculus operations.
2. checking for contradictory cubes in the pipeline mode.

These two parts of the CU will be explained with algorithms in the next two sections.

0.9.1 THE PIPELINE MODE

To use the CCM in certain structures for pipelining it is necessary to detect contradictory cubes in the input. A special preprocessing algorithm was added to the CU which is only used if the *pipline mode* flag *PIPELINE* is set. A contradictory cube means an empty cube, therefore in this preprocessing algorithm the both input cubes have to be checked for contradiction (inclusion of ϵ). If there is a contradictory cube in the input the output has to be an empty cube (no cube generated at all).

The *pipeline* mode is the only mode in which the CU directly has to set the relation that should be performed from the ILU. It has to set the relations A $\neq$ 0 and B $\neq$ 0. In Figure 9.11. the Karnaugh maps and the resulting code for these operations is shown.

In the pipeline mode, the two input cubes, A and B, have to be checked for contradictory input cubes. Thus, the CU has to provide the ILU with the signals to compare if register A or B is empty (see states 3 - 6 in the flowchart).

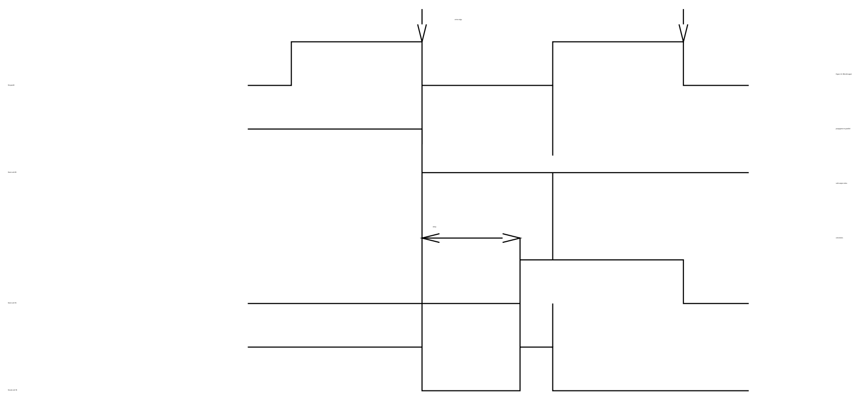


Figure 0.13: *Fig. 11.2.*

The pipeline algorithm is as follows:

```
if ( PIPELINE )

{

if ( A == 0 || B == 0 )  /* contradictory cube ?        */

RESET_OUTPUT;      /* set output register to zero */

goto END;          /* state 15 in the flowchart   */

else if ( IRO )

goto START;        /* state 1 in the flowchart    */

}
```

0.10 THE MAIN ALGORITHM

The main algorithm has to provide the ILU with the signals for the generation of the resultant cubes for the sequential cube calculus operations. The number of the resultant cubes is determined by the number of the specific positions. Those positions are specified by relations shown in Section **??** (in Table 8.3).

The algorithm shown below is reflected in the states 8-14 in the flowchart.
The main algorithm is as follows:

```
CLEAR; /* reset the FSMs of the ILU */

NEXTO =
true;
/* activate the ILU for the sequential operation */

       LOOP : if ( !READYCU || VALID )

                 goto LOOP;   /* state 12 in the flowchart   */

else {



              C_TO _OUT; /* copy the resultant cube to the output register*/

             SET_VALID;} /* indicate that the resultant cube is valid
*/


     if ( NEXTI ) /* all resultant cubes have been calculated */


     goto END;

else {


               REQUEST; /* clock the ILU to generate the next resultant cube */

goto LOOP;}
```

0.10.1 COMPLETE ALGORITHM AND FLOW-CHART OF THE CONTROL UNIT

In this section the flowchart of the Control Unit will be described briefly.
$\longrightarrow$ *State*_1

The Initialization of the Control Unit is done by the *GLOBAL_RESET* signal, which resets the CU from any random state to State_1. The CU stays always in State_1 if the CCM is operated in the chain mode ($IR[1] = 1$). If a sequential operation is chosen ($GO * \overline{IR[1]} * \overline{IR[0]},$) or a pipeline mode and combinational operation is chosen ($GO * \overline{IR[1]} * IR[0] * PIPELINE$), *it will transit to State_2.*

$State\_1 \longrightarrow State\_2 \longrightarrow State\_3$

State_2 is only an intermediate state, because for the transition $State\_1 \longrightarrow State\_2$ the signal em GO which is changed during $State\_2 \longrightarrow State\_3$, is used. The GO signal is resetted and the $BUSY$ flip-flop is set to tell the BIU that it is not allowed to interrupt the CU. Signals $CLEAR$ and $REQUESTILU$ clear all PSMs in ILU to states *before.*

$State\_3 \longrightarrow State\_4 \longrightarrow State\_5 \longrightarrow State\_6$

The CU will transit into State_4 if the *pipeline mode* is set. In the following states the two input cubes A and B are checked if they are contradictory. Because the result of the successor counter is needed the Delay Counter is invoked. If an contradictory input cube occurs, the output will be resetted and the CU will transit to State_11.

In the transit *State_5 ⟶ State_6* the same operation as in transit *State_3 ⟶ State_6* has to be performed: the relation is selected to 01 (REL1 = 1, REL0 = 0) which means prerelation.

"$State\_3 \longrightarrow State\_6$"

If the CCM is not operated in the pipeline mode it will directly transit from State_3 to State_6. During the transition the relation determined by the field prerelation in the instruction register is set.

$State\_6 \longrightarrow State\_1$

This path is for the combinational operation.

$State\_6 \longrightarrow State\_7, State\_6 \longrightarrow State\_8$

The chosen conditions (*AND_OR, COUNT0, COUNT1, COUNT2*) for the main algorithm have to be checked. Because we use in this step the result of the iterative COUNT Counter, the Delay Counter is invoked. If there is no condition the CU will transit immediately from State_6 to State_8.

$State\_7 \longrightarrow State\_8$

$State\_7 \longrightarrow State\_15$

If the conditions are not fulfilled, the alternative combinational cube calculus operation has to be performed in transit *State_7 ⟶ State_1*. This operation is determined by the field *PREOPERATION* in the IR. Execution of the combinational operation is done by setting the *OPERATION* flip-flop to 1 (which means *SET_OPERATION* = 1, *VALUE_OPERATION* = 1).

"$State\_8 \longrightarrow State\_9$"

The and/or type flip-flop is set, according to IR[3]. If IR[3] = 1, it is set to 1, else to 0. Also the operations/relation for the basic sequential cube calculus operation are set.
"*State*_9 $\longrightarrow$ *State*_10"

Necessary timing of the signal *REQUESTILU* and *NEXT0ILU* to calculate the first resultant cube. This corresponds to the D-type flip-flop circuit that was explained in Figure 9.10
"*State*_9 $\longrightarrow$ *State*_1"

For crosslink, if there is no specific position then send empty cube to output.
"*State*_10 $\longrightarrow$ *State*_11 $\longrightarrow$ *State*_12"

If the ILU is ready with the calculation of the resultant cube (*READYCU* = 1) and the flip-flop *VALID* is zero (either the first resultant cube, or the last resultant cube has been read from the host computer) the CU transits to *State*_11. Because the *VALID* flip-flop, beeing the input to transition *State*_10 $\longrightarrow$ *State*_11 has to be set to one, an additional transition (*State*_11 $\longrightarrow$ *State*_12) is necessary. In the transition *State*_11 $\longrightarrow$ *State*_12 also the signal to copy the resultant cube from C of ILU to the output register is given.
"*State*_12 $\longrightarrow$ *State*_10"

If the calculated resultant cube was not the last one *(NEXTI = 0)*, the next resultant cube has to be calculated. Thus, the ILU has to be clocked with the *REQUESTILU* signal.
"*State*_12 $\longrightarrow$ *State*_1" **The last resultant cube has been generated *(NEXTI = 1)*, thus the CU has only to wait until this cube is read *(VALID = 0)*. Then the *BUSY* signal is resetted.**

0.11 UPDATE OF CU FLOWCHART

We observed in the previous flow-chart that some optimization can be done be shifting output signals and combining states.

The new meg.file obtained as the result of those transformations is as follows:
INPUTS:

```
IR0 IR1 IR3 IR4 IR5 IR6

NEXT1 GO READYCU COUNT0 COUNT1 COUNT2 PIPELINE VALID DELAY_END;

\noindent
{\bf  OUTPUTS:}
```

```
SET_BUSY RESET_BUSY RESET_GO REQUESTILU DELAY_START CLEAR NEXT0ILU

SET_AND_OR VALUE_AND_OR C_TO_OUTPUT SET_VALID SET_RELATION

RESET_OUTPUT VALUE_REL1 VALUE_REL0 VALUE_OPER SET_OPERATION;

st1: case ( GO IR0 IR1 PIPELINE)

1 0 0 ?=> st2(REQUESTILU CLEAR);

1 1 0 1=> st2(REQUESTILU CLEAR);

endcase => st1;

st2: goto st3 ( SET_BUSY RESET_GO CLEAR);

st3: if PIPELINE then st4( SET_RELATION VALUE_REL0 DELAY_START )

else st6 ( SET_RELATION  VALUE_REL1 );

st4: case ( DELAY_END COUNT0 )

1 0 => st8 ( SET_RELATION VALUE_REL0 VALUE_REL1 DELAY_START );

1 1 => st1 ( RESET_OUTPUT RESET_BUSY);

endcase => LOOP;

st5: case ( DELAY_END COUNT0 )

1 1 => st1 ( RESET_OUTPUT RESET_BUSY);

1 0 => st6 ( SET_RELATION VALUE_REL1 );

endcase => LOOP;

st6: case ( IR0 IR4 IR5 IR6 )

1 ? ? ? => st1;

0 0 0 0 => st8 ( SET_AND_OR VALUE_AND_OR);
```

```
0 1 0 0 => st8 ( SET_AND_OR );

0 0 1 ? => st7 ( SET_AND_OR DELAY_START );

0 0 0 1 => st7 ( SET_AND_OR DELAY_START );

0 1 1 ? => st7 ( SET_AND_OR VALUE_AND_OR DELAY_START );

0 1 0 1 => st7 ( SET_AND_OR VALUE_AND_OR DELAY_START );

endcase => ANY;

st7: case ( DELAY_END IR5 IR6 COUNT0 COUNT1 COUNT2 )

0 ? ? ? ? ? => st7 ;

1 0 ? 1 ? ? => st8 ;

1 ? 0 ? 1 ? => st8 ;

1 1 1 ? ? 1 => st8 ;

endcase => st1 ( VALUE_OPER SET_OPERATION SET_VALID RESET_BUSY);

st8: if IR3 then st9 ( SET_AND_OR VALUE_AND_OR NEXT0ILU REQUESTILU SET_RELATION SET_OPERATION )

else st9 ( SET_AND_OR NEXT0ILU REQUESTILU SET_RELATION SET_OPERATION );

st9: goto st10 ( NEXT0ILU );

st10: case ( READYCU VALID )

1 0 => st11;

endcase => LOOP;

st11: goto st12 ( C_TO_OUTPUT SET_VALID );

st12: if NEXT1 then st1(RESET_BUSY)

else  st10 ( REQUESTILU );
```

```
\section{ EXAMPLES OF OPERATION }

Disjoint sharp operation.

This is an or_type operation. The preset REL signals are as follows:
```

A	B	REL
0	0	0
0	1	0
1	0	1
1	1	0

```
Disjoint sharp operation is described with the following Table.

# CU=IT[0] U[01] U[23] U[45] V[01] Z[01] Z[23] CU=IT[n+1]
1 i 0 1 2 3 4 5
2 IT[i] IT[1] IT[2] IT[3] IT[4] IT[5]
3 A[i] 01 00  11 00 11
4 B[i] 11 11  01 00 10
5 REL0[i]+REL1[i] 0  0  1 0 1
6 RIGHT_ED[i] 1 0  1  1 0 1
7 CARRY[i+1] 1 0   0   1 0 1
8 CONF[i-1] 0  0  1 1 1
9 VARIABLE[i] 0  0  1 1 1
```

Table

0.12 COMMUNICATION-BASED OPERATIONS.

Just as the DSP or AI architectures can be build from regular structures of few kinds of blocks through which data vectors are transmitted, the principle of CCM is to allow for pipelined/systolic execution of the most common operations on clists and cclists. Those operations include for instance the Cartesian Products, the dot products, the permutations, and the generations of all subsets or combinations of a clist. Micro-Programming of CCM consists then in assigning values to two kinds of control fields:

1. Those that describe the processing inside the CCM, for example, the operations on cubes and short clists described in section ??.

2. Those that describe the communication of the CCM with other processors and memories.

There are currently two styles of communication:

1. Address Generator based.

2. Data Flow based.

Address Generators are used to send out the address of the desired operand in Dual-Port Memories. The data flow style is based on FIFOs. In this case the CCM chip (CCM processor?) sends a signal to a FIFO to get the next cube. A Ping-Pong communication mechanism is used to transfer the data from Memory A to Memory B while the processing. Possibly, the the data-flow in the processor can be changed for every change of direction. and this is controlled as well.

There are three main types of communication-based operations:

1. simple operations on a clist,

2. simple operations on two clists,

3. (3) complex operations on a clist.

Ad. (1).
Simple operations on a clist include *Global Reduction, Mapping, Pairing, Selection, Inversion, Mapping with Insertion, Inversion with Insertion, Replacement, All Pairs with Reduction, All Ordered Pairs,* **and other operations. Examples of some of those operations on clist A = {** a_1, a_2, a_3 **} are as follows.**
NOTE:
In the following, { } denotes a clist, {{ }} a cclist,

a cube, $agroupofvariables, ||||aset,' asymbol.OPERandOP\_CLISTaretwo-operandcubeoperations.$

Global Reduction , in which a cube is created from a clist:

REDUCTION (A, OPER) $= [a_1 \ OPER \ a_2 \ OPER \ a_3]$

Examples of such operations in cube calculus are the supercube of all cubes from the clist, and the intersection of all cubes from the clist.

Mapping , for which the clist produces a new clist of the same size:

MAPPING(A, OPER) $= \{ \ OPER(a_1), \ OPER(a_2), \ OPER(a_3) \ \}$

Pairing , where the operation is repeated on subsequent pairs of arguments and the results are directly inserted in a new clist:

PAIRING(A, OPER, OP$_C$$LIST) = \{$OPER (a$_1$, $a_2) \ OP_C LIST \ OPER(a_3, \ \emptyset) \ \}$

Sorting , can be realized as repeated pairing.

Sorting_with_Absorption can be realized as repeated pairing.

Reducing Operations such as:

$ab \vee a \implies a,$

$a \vee a \implies a,$

$ab \oplus a \implies a\bar{b},$

or

$a \oplus 0 \implies a,$

can be used as OP_CLIST. Repetitive Ping-Pong transfer of a clist through a CCM with an exchange of neighbor cubes (based on a specified ordering relation between them) leads to sorting the clist. When the compare/exchange operation also executes cube absorption, algorithms such as POS $\implies$ SOP transformation by multiplication can be executed [?].

=item[Inversion], for which the clist creates a new clist of the same size:

INVERSION(A, OPER) = { OPER(a_3), OPER(a_2), OPER(a_1) }

All Pairs of a Clist:

ALL_PAIRS(A, OPER) $= \{ \ OPER(a_1, \ a_2), \ \ OPER(a_1, \ a_3), \ OPER(\ a_2, \ a_3) \ \}$.

Selection , is an operation known from CAM-based and associative processing. It selects cube(s) for which a certain Boolean function of literal-related predicates is satisfied. For instance the operation:

$"age > 34"$ AND $character = `good"$ OR $properties \subseteq \{||\}wealth,\ good - contactsroman\{||\}"$

operates on cubes with numeric variable *"age"*, symbol variable *"character"*, and set variable *"properties"* and selects all people who either are older than 34 and have good character, or have wealth and good contacts among their properties.

Replacement , is a similar operation. It does some cube operation on selected cubes such as adding \$ 1000 to the numeric variable *"salary"* or adding set value "good-contacts" to the variable "properties".

The following operations of Cube Pairing and Cube Pairing with Reduction are executed on a clist and a cube:

CUBE$_C LIST\_PAIR(A, cube, OPER) = \{$**OPER** ($\mathbf{a}_1$, $cube), OPER(a_2,\ cube), OPER(a_3,\ cube)$
$\}$

CUBE$_C LIST_R EDUCE(A, cube, OPER, OP_C LIST) =$

$\{ OPER(a_1,\ cube)\ OP\_CLIST\ OPER(a_2,\ cube)\ OP\_CLIST\ OPER(a_3,\ cube)$
$\}$

Ad (2)
Simple operations on two lists include *Dot Product, Vector Product, Simple Cartesian Product, Sequential Cartesian Product, Convolution, Vector Product with Insertion, Vector Product with Reduction,* and other. Some of those operations on clists A = $\{ a_1,\ a_2,\ a_3 \}$ and B = $\{ b_1,\ b_2,\ b_3$
$\}$ are the following:

Dot Product , for which two clists create a cube:

DOT_PRODUCT(A, B, OPER, OP_CLIST) =

$[OPER(a_1,\ b_1)\ OP\_CLIST\ OPER(a_2,\ b_2)\ OP\_CLIST\ OPER(a_3,\ b_3)]$

As an example, OPER can be a cube intersection operation, and OP_CLIST can be a cube supercube operation.

Vector Product , for which two clists create a clist:

VECTOR_PRODUCT(A, B, OPER) = $\{ OPER(a_1,\ b_1)\ ,\ OPER(a_2,\ b_2)\ ,\ OPER(a_3,\ b_3)$
$\}$

Simple Cartesian Product , where two clists create a clist (in particular representing a matrix, or a set of pairs):

SIMPLE_CARTESIAN_PRODUCT(A, B, OPER) = { $OPER(a_1,\ b_1)$, $OPER(a_1,\ b_2$
$OPER(a_2,\ b_1)$, $OPER(a_2,\ b_2)$, $OPER(a_2,\ b_3)$, $OPER(a_3,\ b_1)$, $OPER(a_3,\ b_2)$, $OPER($
}

This matrix is returned row by row, from left to right.

Sequential Cartesian Product , where two clists create a clist. In this operation the OPER is applied sequentially to the result of the operation of A with the first cube of B, the second cube of B, etc, as follows:

SEQUENTIAL_CARTESIAN_PRODUCT(A, B, OPER, OP_CLIST)
= { CUBE_CLIST_REDUCE((CUBE_CLIST_REDUCE((CUBE_CLIST_REDUC
b_1, OPER, OP_CLIST) b_2, OPER, OP_CLIST)) b_3, OPER, OP_CLIST)) }

This is applied in sharp, intersection and union of clists.

Convolution . A new clist of length $2n - 1$ is created according to a convolution algorithm such as that used in polynomial multiplication. This operation is a base of many image-processing operations.

Ad. (3) *Complex operations on a single clist* include the following: *All permutations, All subsets, All combinations,* and other. Such operations, which are in fact the mappings of tree-search algorithms to pipes, are based on two coordinated pipes. For instance, *All Subsets Operation* uses the method from [?] to generate all subsets of a set. This operation can be used to generate all cubes that include a specified cube.

Many other pipelined/systolic/data-flow MVCC operations can be created that are especially useful for problem-solving and spectral applications.

8. CONCLUSION.

The paper presented our new Generalized Multiple-Valued Cube Calculus and the basic features of our Cube Calculus Machine CCM2, the first universal multiple-valued logic machine. The CCM2 bit-slice chip and CCM2 processor board are currently under development at PSU. We are also working on microprogramming, development of a high-level Lisp-based programming environment, and a next generation CCM to implement an even more general multiple-valued logic.

0.13 INDEX OF TERMS AND SIGNAL NAMES

AAAAAAAAAAAAA

AND_OR determines if the AND- or the OR-type relation shall be performed

ACTIVE[i] AFTER[i]

ACT function

Active literal

Additional Control Logic of ILU

AFT function

AND-type relation

Asymmetric consenus

BBBBBBBBBBB

BUSY_VALUE

BUSY CU is busy, performing an algorithm

BEFORE[i]

Basic sharp

BEF function.

Before active

Binary consensus

BIU

CCCCCCCCCCCC

CARRY[i]

C[i]

CC operation

Chain mode

CLEAR

COMPARE

CON[i]

CONF[i]

COUNT0 result of counter COUNT is 0

COUNT1 result of counter COUNT is larger than 1

COUNT2 result of counter COUNT is larger than 2

COUNT[1:c] counter

C_TO_OUPTUT content of the result register is copied to the output register

CNT[i]

Combinational CC operations C_TO_OUTPUT

Complex combinational operations

Contradiction in cube

Control Unit CU

Crosslink

Cube Calculus
Cube Calculus Operations
DDDDDDDDDDDDD
DELAY_END, Del_end, signal from Delay Counter that finished
DELAY_START Del_start, tells delay counter to start
Data Path (DP)
Delay Counter
Disjoint sharp
Double Prime Operation
EEEEEEEEEEEE
EX0[i]
EX1[i]
Exclusive Sum of Products Form ESOP
FFFFFFFFFFF
Finite State Machine (FSM)
Frontier between variables
Full term
GGGGGGGGGGGG
GLOBAL_RESET
GO signal from BIU to CU to start chosen instruction
HHHHHHHHHHHHH
IIIIIIIIIII
IDENTIFY[i]
Instruction Register (IR)
Interlock mechanism
Intersection
$IR[0]$: sequential (0) or combinational (1) cube calculus operation
$IR[1]$: chain or single mode
$IR[2]$: counter mode
@@ Zamiast IR[2] wstaw count (co zliczamy - 0: bit by bit 1: specific literals) @@
$IR[3]$: and/or type relation for relation
$IR[4]$: and/or type relation for prerelation
$IR[5,6]$: determines the kind of sequential algorithm
$IR[7]$: prime operation, not for control unit = 1 for operation prime.
Iterative Cell IT
Iterative Logic Block (ILU)
JJJJJJJJJJJJJJ
KKKKKKKKKKKKkk

0.14 DESIGN EVALUATION AND COMPARISON OF CCM2

Now that the ILU has been verified in its operation, a proper timing analysis must be done to complete the evaluation of the CCM2 hardware. By analyzing the CCM2's timing characteristics, both the internal delays of the logic devices and the amount of time it takes to process data received at the inputs and produce the results at the outputs must be taken into account. Since only the ILU has been investigated in this thesis, all external delays to the hardware will not be considered.

0.15 TIMING ANALYSIS

When determining the internal timing of the ILU, there are three types of delays that must be investigated: vertical signals received from the host computer at the inputs and leaving the ILUs outputs; horizontal signals received from the CU at the inputs of the first IT and returned from the outputs of the last IT back to the CU; and lastly the delays incurred from executing sequential and complex combinational operations upon the input cubes.

The vertical signals include input, output, prerelation, relation and instruction register signals. The delay of concern here is the time it takes to compute a combinational operation once the input cube values have been received, then passed to the outputs of the device. For simple combinational operations such as intersection and supercube, the CU is not needed, so the time involved in calculating the output cubes is determined by the delay of the signals through the RELATION subsection of each IT. With the most recent placement and routing of the CCM2 design, it was found that the greatest delay was in the first IT of each device. The delay of 145.8 nanoseconds was the greatest delay found, while a delay of 89.2 nanoseconds was the smallest delay found in the last IT of each Xilinx device. All ITs within the ILU have the identical design, so the reason for the 56.6 nanoseconds difference in delays is caused by the placement and routing of the design. Thus, these delays are likely to vary each time the automatic placement and routing routine is run.

The horizontal signals include NEXT, READY, CLEAR, PRIME and REQUEST, and are the signals used in the generation of cubes from sequential and complex combinational operations. All of these signals are received by or returned to the CU, however, only the NEXT and READY signals will be considered here, since they are critical for the generation of output cubes. CLEAR, PRIME and REQUEST are received at each IT at approximately the same time and do not get passed to adjacent cells during operation. The NEXT and READY signals are used to coordinate the states at which specific ITs of the ILU should be for the generation of multiple output cubes. These signals ripple through the chain of ITs under the control of the CU. Obviously the delay is dependent upon the size of the input cubes and the number of output cubes produced, so the delay times are based upon all ITs being used, and the ILU producing a single resultant cube. The signal NEXT has a delay of 353.4 nanoseconds while READY has a delay of 219.4 nanoseconds. It will be shown in the following section that these two signals are the limiting factor in the speed of my design. The attached timing sheets found in Appendix B, show the delays of the two signals found for a single device. It can be seen that a major source of delay is from the signal entering and exiting the device to the I/O pins. Once the design is fit into a single programmable device, these delays will decrease significantly since the signals must currently pass through two devices.

To justify the need for special hardware to accelerate the processing of logic functions, we must first evaluate existing logic minimization programs and determine which software routines might benefit from this hardware. Many logic minimization software tools such as ESPRESSO [?, ?, ?] MINI [?], and MIS II [?, ?] may benefit from the addition of the CCM2. In ESPRESSO, there is no direct relation between any one measure of problem size (inputs, outputs, literals, terms, and so forth) and the computing time. If a routine is detected that would greatly benefit from the addition of the CCM2, it may then justify the additional cost depending upon the frequency that the routine is called during typical operation.

It was found that one essential algorithm used in ESPRESSO-II could have its processing time significantly reduced with the addition of the CCM2. The COMPLEMENT procedure is executed exactly once during typical operation, and consumes 14% of the total CPU time on average[?]. It is second in time consumption to the EXPAND procedure which on average requires 29% of the CPU time. The EXPAND procedure uses the results of the complement to quickly determine primes covering a given cube and to help guide which of the primes will be chosen to produce the best cover. One might chose to eliminate the COMPLEMENT algorithm, however, the initial cost of the procedure is offset by the increase in computing time needed by EXPAND. In addition, the quality of the cover obtained after each expand step would be reduced.

On the CCM2, a combinational operation such as complement [?, ?, ?, ?] may be executed in six clock cycles of the CU for a single resulting output cube. Assuming an Intel 80486 or equivalent clone processor with a system clock of 33 MHz, giving a period of 30.30 nanoseconds per clock cycle, the result of a single cube output from the complement of an input function may be calculated in 181.80 nanoseconds. This may be seen in Figure 21 on page 48, showing the single resultant cube being calculated after 6 clock cycles. For multiple resultant cubes, the first cube is produced after 6 clock cycles, then the subsequent output values are produced after every additional three clock pulses, until the NEXT signal has been received. Thus, the complement of a function producing two resultant cubes will take 272.70 nanoseconds, three resultant cubes will take 363.60 nanoseconds, and so on. These figures are not entirely true since the calculation cannot be written to the data bus until the NEXT signal is received. This takes 353.4 nanoseconds to be received, then an additional 30.30 nanoseconds to be output to the data bus, for a total of 383.7 nanoseconds for one or two resultant cubes. If three resultant cubes were produced, it would take 393.30 nanoseconds. From this data, it can be seen that for cube calculus operations producing multiple cubes, there is a great speedup benefit since only 3 clock cycles are needed for each additional resultant cube after the first one has been generated.

Assuming that the COMPLEMENT function was sped up by one thousand times, this still does not justify the addition of special hardware since we are essentially speeding up 14 percent of the entire processing operation. The remaining 86 percent will be calculated in the same amount of time as before. However, many software tools and special hardware accelerators are designed to handle specific minimization problems and operations, and this is where we feel the CCM2 will find its niche. Two such possible problems presented below are satisfiability and tautology [?, ?]. Both of these problems may take advantage of the complement and sharp (#) operations available on the CCM2.

Satisfiability The satisfiability problem asks the question "for what values of arguments is a particular formula satisfied". Basically, *"are there any cells or groups of cells of a Karnaugh map of function f that equal 1."* The satisfiability problem may be reduced to the tautology problem and vice-versa through the complementation operation.

If the function f(a,b,c) = 1, this implies that 1# f(a,b,c) = $\emptyset$. This may also be written as 1 # f = $1 \cdot \bar{f} = \bar{f}$, which is the complement of the function f. Figure 9.32 shows the function f = X^{56} $Y^{012345678}$ for which the complement of the function equals $1 \cdot \bar{f} = \bar{f}$.

The complement of the input cube A is as follows:

$\bar{A} = \{{X_1}^1 \cdot \cdot \cdot {X_{i-1}}^1 {X_i}^{\neg A_i} {X_{i+1}}^1 \cdot \cdot \cdot {X_N}^1 \mid for\ such\ i = 1, ..., N,\ that\ A_i \neq 1\}$

The resultant cube of the complement function on input cube A is equal to X^{012347} $Y^{01234567}$ and may be seen in Figure 9.33.

Tautology

The tautology problem is the verification of a logical formula to see if it is always true such as verifying whether ab + a = a. If the function f(a,b,c) = 1, this implies that 1 # f(a,b,c) = $\emptyset$. If f = $\bar{a}\ \bar{c} + \bar{a}\ \bar{b}\ c + bc + ab\bar{c} + a\bar{b}$, then $1\#f$ must equal $\emptyset$. The sharp of two cubes A and B denoted A#B, is a set of all of the largest cubes included in A and not included in B. Given the function $f(a,b,c) = \bar{a}\ \bar{c} + \bar{a}\ \bar{b}\ c + bc + ab\bar{c} + a\bar{b}$, we can see in the Karnaugh map in Figure 9.36 that the function equals one although that is not apparent to us by just looking at the expression of the function.

In order to verify if the function is equal to one, our methodology would be to sharp a cube of the input function from the function f = 1, then take the resultant function f' and sharp the next cube of the original input function from it, iterate for all cubes of the input function and check if the final result is empty. This method may be seen in Figure 9.35.

Observing the methodology that is used in Figure 9.35 to prove that our input function F is equal to one, the first input cube $\bar{a}\ \bar{c}$, is sharped from a function equal to "1". The two resultant cubes from this operation are stored, then they each sharp the second cube of the input function, storing the resultant cubes from each nondisjoint sharp operation. This process is repeated until there are no further input cubes left. The final result is checked, and if the final resultant cube is equal to zero, then we have indeed verified that our input function of $f(a,b,c) = \bar{a}\ \bar{c} + \bar{a}\ \bar{b}^c + bc + ab\bar{c} + a\bar{b}$ equals 1. From this example of tautology, I have written a small program that executes the

Figure 0.14: *Input cube* $A = X^{56}\ Y^{01234567}$

Figure 0.15: Resultant cube $C = \bar{f} = X^{012347}\, Y^{01234567}$

Figure 0.16: $f(a,b,c) \;=\; \bar{a}\;\bar{c} \;+\; \bar{a}\;\bar{b}\;c \;+\; bc \;+\; ab\bar{c} \;+\; a\bar{b}$

Figure 0.17: *Tautology Verification*

Figure 0.18: *(Continued). Tautology Verification*

tautology algorithm based on these principles.

This program was created in order to determine the speedup of such a problem using the CCM2 hardware instead of using a software manipulation. Located in Appendix D, is the C code program that simulates the above tautology example. The code was developed on an Intel 486DX-33 platform, using a Microsoft C version 6.0 compiler. The granularity of the time reported back from a PC is accurate to approximately 1/18th of a second, so the program was placed in a loop that iterated the tautology problem one million times in order to obtain a reasonable time measurement. The program was able to compute the tautology example in approximately 33.56 microseconds. Using the times that were calculated earlier for sequential operations to be processed in the CCM2's ILU, I was then able to determine the processing time for the same tautology example.

It was found that the nondisjoint sharp operation was performed ten times for this particular example, with each operation producing anywhere from zero to two resultant cubes per sharp operation. This time was determined to take a total of 2.30 microseconds. Comparing the software and hardware times shows that the ILU of the CCM2 is able to accelerate the operation by approximately 13.59 times. This speedup is based only upon the ILU functionality, and does not reflect external signals to the Xilinx devices such as the host and bus control operations. These times will change significantly when the bus control features are added by future classes. Once the final design, placement and routing changes are made, the ILU delays will likely decline. Additionally, the LCA design may be ported to an ASIC device, and the programmable connections will be hardwired, dropping delay times considerably.

After observing the differences in the processing time between the hardware and software implementations of the three variable tautology example, I decided to try some examples of other tautology problems with varying numbers of input variables. Table 9.18 shows the variation of problems attempted. There is a definite increase in the speedup ratio between the hardware and software, however, the smaller problems had two input cube features changing, thus I was unable to determine what the true cause of the speedup was. I then tried some larger input functions, holding the number of input cubes within each function to be constant, thus making the additional variable a don't-care value in the input function. By doing this, I was able to see that

Number of Variables	2	3	4	5	6
Number of Input Cubes	3	5	7	7	7
Software Sharp Simulation (ms)	12.53	33.56	135.67	162.58	182.63
CCM2 Sharp Operation (ms)	1.15@2.30	7.32	7.32	7.32	
Hardware Speedup	10.90	14.59	18.53	22.21	24.95

Table 0.8: *Differing variable size sharp comparison.*

the CCM2 was able to process input functions of 4, 5, and 6 variables at the same amount of time. The software simulation, however, steadily increased in time with the addition of extra variables. This is due to the software's need to iterate through an another loop for each additional variable that is presented in the input function. Figure 54 shows the speedup comparison of multi-variable tautology functions that were tested.

The number of variables was then held constant at three, and we constructed 10 different covers, ranging from one to ten input cubes. Table 9.19 shows the results obtained from this trial. As you can see, the software ramps up at a linear increase in processing time as the variable count rises. The graphical representation of this may be seen if Figure ??. A large hit in processing time is taken on the seven input cube example. This may be attributed to the manner in which the algorithm processes the tautology example. If the function were ordered in a slightly different manner, less resultant cubes would likely result, since the generation of multiple resultant cubes adds an additional loop of computation. The CCM2 also increases linearly, however, it tends to level off. This is attributed to the fact that the tautology has likely been proven at this point. With a three variable function, the cover may be made with eight of the smallest possible input cubes.

It can be noted from the internal delays of the Xilinx devices, that large functions with many variables will be what fully utilizes the functionality of the CCM2. Small functions such as the three variable tautology example presented in this thesis, don't necessarily utilize the parallel architecture of the ILU. These small functions are crippled by the delays inflicted by the horizontal signals that are used to control the output of the resultant cubes. The CCM2 must also target logic minimization problems that are quite repetitive in their structure such as the tautology example, or applications that are extremely cube calculus intensive.

hline					
hline Number of Input Cubes	**1**	**2**	**3**	**4**	**5**
Software Sharp Simulation (ms)	**5.33**	**10.55**	**14.86**	**19.50**	**33.56**
CCM2 Sharp Operation (ms)	**0.38**	**0.77**	**1.15**	**1.53**	**2.30**
Hardware Speedup	**14.03**	**13.70**	**12.92**	**12.75**	**14.59**
Number of Input Cubes	**6**	**7**	**8**	**9**	**10**
hline Software Sharp Simulation (ms)	**30.37**	**128.18**	**85.90**	**45.97**	**39.88**
CCM2 Sharp Operation (ms)	**2.30**	**5.76**	**6.94**	**5.76**	**3.07**
Hardware Speedup	**13.20**	**18.47**	**14.91**	**14.97**	**17.34**

Table 0.9: *3-Variable sharp comparison.*

To simply replace a small percentage of cube calculus operations within an existing logic minimization package would not be worth the added expense.

0.16 BDD COMPARISON

An alternative method to Cube Calculus in the minimization of logical functions is through the use of Binary Decision Diagrams (BDDs) [?, ?, ?, ?, ?]. BDDs are a data structure used to represent boolean functions and an associated set of manipulation algorithms. BDDs support many common logical operations including and, or, xor, smoothing, consensus, containment, tautology and satisfiability. Functions are represented by directed, acyclic graphs commonly referred to as DAGs. The resulting graph is a concise description of a logical function telling the user how to determine the output value of the function by examining the values at the inputs. Considering the function $F = A + \bar{B}C$, the resulting BDD is created in Figure 9.37.

Each node in the diagram refers to a single variable from the function F, with each branch leading in the direction of the value of the variable. A node may have the value of 0 or 1. With n variables, there will be an initial $2^n - 1$ nodes in such a diagram. Reducing this number has been the focus of research in BDD implementations over the past decade.

Nodes may be easily removed by observing the structure of the diagram and removing ones that lead to identical output values regardless of the variable value. In addition, two separate nodes may both lead to the same output value and are capable of

Figure 0.19: *Function* $F = A + \bar{A}C$ *in BDD representation.*

being merged. These two methodologies provide simplification to the diagram, however, with the exponential increase in the number of nodes, large logic functions require significant amounts of processing time.

The size of the BDD representation for a given function is extremely sensitive to the selection of variable order. The problem of finding the optimal variable ordering is NP-complete, with all known exact methods being based upon exhaustive searches. As a result of this ordering dependence, poor orderings can limit the use of BDDs in complex problems due to finite memory resources. In addition, optimal CPU times are achieved by good variable orderings since the time to perform a BDD operation is proportional to the size of the BDD. Thus, considerable research is focusing upon heuristics for finding good orderings.

Ordered BDDs (OBDDs) are currently being explored for use in synthesizing and analyzing combinational and sequential circuits, with the emphasis of their power being directed towards design verification [?, ?, ?]. By developing clever representations and efficient manipulation algorithms, it is possible to avoid the time consuming exponential computations of simple BDD structures. Reduced Ordered BDDs (ROBDDs) have been used in different stages of logic design due to their representative form of Boolean functions and canonical structure [?, ?, ?]. ROBDDs have been proven in signal and fault detection applications since they may be constructed to represent the detection function. Modified BDDs (MBDDs) are simply ROBDDs with a third terminal used as a don't-care variable [?]. The method is based on the replacement of the don't-care terminal by other subgraphs of the MBDD, thus leading to the reduction of the graphs size. This technique may be used in FPGA synthesis to simplify circuit subfunctions before mapping the design to the device.

These methodologies are all based upon heuristics which reduce the number of possible input variables, which directly effect the size of the graph structure. These heuristic techniques use partial simulations which allow the possibility that comparisons between two functions may not be detected. Additionally, these algorithms may require large amounts of memory for their storage, increasing cost and affecting the speed of computation. For large circuits, this is infeasible since the trade-off between speed and cost is so high. However, it has been determined that BDDs give much better results than with cube calculus.

The BDD structures have just recently gained researchers attention over the last decade, and due to the before mentioned problems, not all logic designers have moved away from using cube calculus. At this time, it is not known by the author, the extent of ROBDD or MBDD structures that have been realized in hardware. Standard BDDs' have a simplistic structure which may be easily realized using 2-to-1 multiplexors. With this ability, recent research has focused on multiplexor-based FPGA synthesis. FPGA architectures such as ACTEL have configurable memory cells that are multiplexor-based, allowing easy mapping of Boolean functions of the BDD form. Due to advances in multiplexor-based programmable logic devices, mapping BDD designs to programmable resources will be quite convenient.

To do a proper comparison between the CCM2 design and a BDD logic minimization package, some comparison between existing packages using the two methodologies had to be found. Jacobi and Trullemans [?], present a logic minimization method using MBD's that they determined to perform at a 35 percent efficiency over ESPRESSO [?, ?, ?]. This efficiency is measured in the size of the function after the minimization process. From our data in Table 9.19, we achieve a speedup of over 14 times that of the software representation. If I make the assumption that my particular code is half as optimized as that of the sharp operation in ESPRESSO, I would be making the statement that the CCM2 is seven to eight times faster than ESPRESSO. The efficiency in terms of minimization would be equal since they would both use the sharp operation to perform the optimization. I can then make the statement that the CCM2 should be able to out-perform the MBD optimization by over 4 times.

If these two methods were available in an off-the-shelf fashion to perform the same function for the same price, the CCM2 would provide the best performance for the price. The BDD implementation might provide better performance in some instances such as the case seen in Figure ??, where the input cube ordering has a direct effect on the number of resultant cubes produced and the processing time needed. This appears to happen approximately 20 percent of time for the examples that have been run.

Due to the programmable resources available, and the progress already achieved in the CCM2 design, cube calculus remained as the methodology of choice for the design. Future research should consider building a BDD-based accelerator board, or the possibility of a combined cube calculus and BDD board. With a universal hardware implementation such as this, logic minimization

programs will be able to take advantage of both methodologies, adding flexibility and processing power to many applications.

Chapter 1

THE CCM 2.5 PROCESSOR

1.1 CCM2.0 VERSUS CCM2.5

As we have seen in the previous chapter, the CCM 2.0 processor can not execute all existing Cube Calculus operations, even though its design is based on a general format for these operations. One of the restrictions of the CCM 2.0 is the fact that each of the output bits $C_j[i]$ of an IT has only two input bits. Another restriction is caused by the single CARRY and CONF lines, which limit the number of relations that can be checked. In the CCM 2.5 processor these points have been improved. However,the main difference between the CCM 2.0 and the CCM 2.5 is the way in which the processors implement any possible Boolean function of a certain number of input bits.

The CCM 2.0 processor is largely based on the implementation of any possible Boolean function by using multiplexers. These multiplexers are mapped to the logic blocks of the FPGAs. A new function can be programmed by changing the signals to the data inputs of the multiplexers. The CCM 2.5 processor takes a different approach. In stead of programming the FPGA's with a series of multiplexers, they are programmed with the actual Boolean functions that need to be implemented. If the ILU needs to execute a different CC operation, all the FPGAs that hold the CCM 2.5 processor will be reprogrammed. However, the functions *prerelation*, *relation*, *before*, *active* and *after* are all available, so no reprogramming is necessary to switch between them.

This approach seems to remove the advantage of the large set of operations that the CCM processor can perform, since the operation that the ILU performs can no longer be changed by simply changing the input signals for the ILU. However, this is only a small disadvantage. The FPGAs can be easily reprogrammed to execute a new CC operation, even while the program that is using the CCM as a coprocessor is running. The new operation can be previously mapped to FPGAs and saved in a file. When the new operation is needed, the already available FPGA program needs only to be loaded into the static RAM of the FPGAs. Due to the serial interface of the FPGA, however, this still takes at least several seconds.

However, a program that uses the CCM as a coprocessor, e.g. a logic synthesis application, will typically perform the same CCM operation over and over again, on a large amount of data, before changing to another CCM operation. The few seconds that will then be needed to reprogram the CCM processor will be insignificant to the total runtime of the program and the time saved by calculating CC operations in hardware rather than software.

There are also advantages in programming the FPGAs with the actual Boolean functions in stead of multiplexers. Since the functions of the different parts of an IT, like the relation and the operation that generates the bits of the resultant cube C, are already programmed, the number of signals that the CU needs to send to the ILU is greatly reduced. This simplifies the design of both the CU and the ILU. Also, some multiplexers in the CCM 2.0 switch between groups of four bits. Many logic blocks are needed to implement this in an FPGA. The CCM 2.5 does not need these multiplexers.

The main advantage of the CCM 2.5 over the CCM 2.0 is its increased versatility. The general structure of the parts of an IT, like IDENT_MAIN[i] and STATE[i], and the signals between those parts, like $CARRY[i+1]$ and $CONF[i-1]$, will not be changed. In the CCM 2.5, the logic inside these blocks can be programmed to any possible Boolean function with a certain number of inputs and a certain number of outputs. For example, the block IDENT_MAIN is not programmable in the CCM 2.0, and the equations for the $CARRY[i+1]$ and $CONF[i-1]$ are the same for all operations. In the CCM 2.5, $CARRY[i+1]$ can be any function of the inputs of the block IDENT_MAIN, and so can $CONF[i-1]$. This allows new ways of determining specific literals in the input cubes. More relations can be checked, and more CC operations are possible.

The CCM2.0 supports a large number of cube calculus operations, but is still has some of limitations. For instance, it can only check relations and execute operations that can be decomposed to a function of two bits A_i and B_i. In the CCM 2.5, the block SIGNALIZE of the IT consists of a universal logic module with 4 inputs $A_0[i]$, $A_1[i]$, $B_0[i]$ and $B_1[i]$, and two outputs $C_0[i]$ and $C_1[i]$. This universal logic module must not be confused with a Configurable Logic Block (CLB) of the Xilinx FPGAs. A universal logic module of the CCM can consist of several CLBs. This universal logic module can execute any function that can be realized with these four inputs and these two outputs. This means that $C_0[i]$ no longer has to be a function of $A_0[i]$ and $B_0[i]$, but can now be a function of $A_1[i]$ and $B_1[i]$ as well. This enables the CCM2.5 to execute a bigger set of cube calculus operations than the CCM2.0.

CCM 2.0 processor

Programmable Boolean functions are implemented by multiplexers

CCM 2.5 processor

All Boolean functions are implemented in universal logic blocks, which are able to calculate any function of a certain number of inputs and with a certain number of outputs.

CCM 2.0 processor

The current operation and cube sizes are indicated by the CU and can be instantly changed.

CCM 2.5 processor

The operation and cube sizes can only be changed by reprogramming the FPGAs that hold the CCM. The functions *prerelation*, *relation*, *before*, *active* and *after* are all available.

CCM 2.0 processor:

The relation and operation can be decomposed to single bits. The output bits $C_j[i]$ are calculated by two separate functions with one input bit from each argument cube, and one output bit.

CCM 2.5 processor:

The relation and operation can be decomposed to groups of two bits. The output bits $C_j[i]$ are calculated by a universal logic module with two input bits from each argument cube, and two output bits.

CCM 2.0 processor:

The Boolean functions *prerelation*, *relation*, *before*, *active* and *after* are equal for all ITs.

CCM 2.5 processor:

The Boolean functions *prerelation*, *relation*, *before*, *active* and *after* can be different for each IT.

CCM 2.0 processor

Only the functions *prerelation*, *relation*, *before*, *active* and *after* can be programmed.

CCM 2.5 processor:

Every function can be programmed, limited only by the available inputs for the Universal logic module that calculates the function.

Another limitation of the CCM2.0 processor is the fact that the *prerelation*, the *relation* and the functions *before*, *active* and *after* are equal for all ITs in the ILU. In the CCM2.5 processor, each IT can have its own definitions of these functions. In theory, this could also be achieved with the multiplexer approach of the CCM 2.0, but this would result in a very high number of input signals for the ILU. Every IT would need its own set of signals to define the functions *before*, *active*, *after* and *relation*. Including the bits $A_j[i]$ and $B_j[i]$, a total of 20 input bits would be needed to calculate just two output bits, which is not very practical.

Due to the new approach to implementing Boolean functions in the CCM 2.5, several control signals from the CU to the ILU are no longer necessary. Signals that are now obsolete include the 4 data input signals for each of the 5 multiplexers that generated the functions *relation*, *before*, *active* and *after*. The large amount of data on which the same CCM operation will be repeated over and over again, will typically consist of a series of input cubes of the same size. In other words, there will be no changes in the way the different literals are distributed over the different ITs. This makes the signals from the CCM 2.0 processor, that are used to indicate the borders of the different literals, obsolete in the CCM 2.5 processor. These signals are RIGHT_EDGE[i] and W[i]. It is also no longer necessary for IR[7] to indicate that the operation PRIME will be executed, since it can be simply programmed as any other function. The signal *AND_OR* from the CU, which indicates the type of relation that is to be used, is still necessary. This is because there are two different relations that need to be checked in a CC operation, the *prerelation* and the *relation*. Since these can be of a different type, the CU needs to indicate the type of the relation by setting the signal *AND_OR*.

The differences between the CCM 2.0 and the CCM 2.5 processors are summarized in table ??. One major difference is not mentioned in this table. The CCM 2.5 processor is capable of more operations than the current set of Cube Calculus operations. The logic modules, like IDENT_MAIN[i] and STATE[i], are not minimized for just CC operations, but are generally programmable. Also, some signals have been added that are not necessary for CC operations. The goal was to make the CCM 2.5 as generally programmable as possible, within practical limitations of the hardware. This will be discussed in the next chapter.

1.2 TRADE-OFFS IN THE DESIGN OF THE CCM 2.5 PROCESSOR

The goal of the CCM 2.5 processor is a design that is as generally programmable as possible. It not only has to be able to calculate the current set of Cube Calculus operations, but must be fit for possible new Cube Calculus operations without the need to redesign the circuit. It can also be used to easily implement other cellular automata. The CCM 2.5 has the same basic structure as the CCM 2.0. Each signal that is generated in an IT has the same inputs as in the CCM 2.0, but the signal can now be programmed to be any possible Boolean function of these inputs. Also, a number of extra signals was added, to increase the number of operations that the CCM 2.5 can execute. This chapter describes the options that were considered to determine what extra features could be realized within a reasonable hardware limits.

The CCM 2.5 can be divided into a number of universal logic modules. Each module is able to implement any possible Boolean function of a certain number of inputs and outputs. The CCM 2.5 is to be mapped to FPGAs of the Xilinx XC3000 series. The Configurable Logic Blocks (CLBs) of these FPGAs can be configured to 3 different modes:

1. Any function of 5 inputs

2. Any two functions of 4 inputs, whith a maximum of five inputs for both functions together.

3. Any two functions of four inputs, multiplexed by a fifth input.

These functions can either be directly connected to the output pins of the CLB, or used as the data input of the two flip-flops of the CLB.

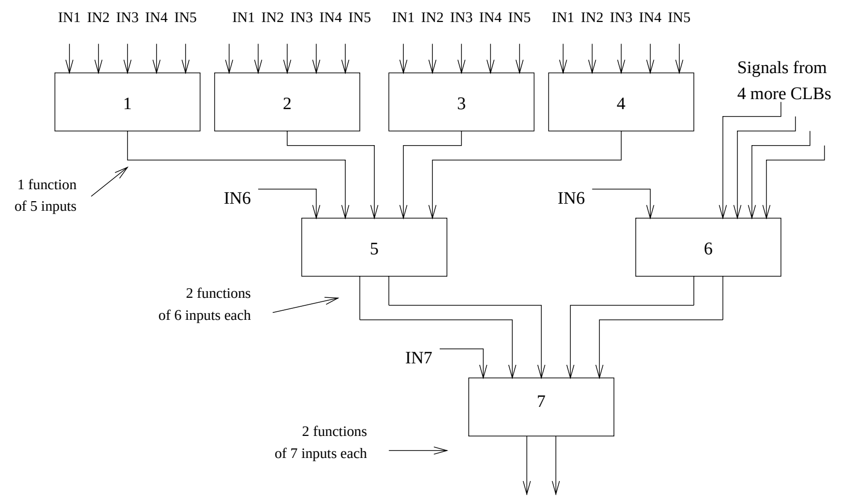


Figure 1.1: *Implementing functions of 6 or 7 inputs*

Functions of up to five inputs can be calculated by a single IT, but we also need functions of six or seven inputs. Fig. 10.1 shows the most efficient way to implement functions of more than five inputs. CLBs 1 to 4 each calculates a function of the same 5 inputs. To implement a function of 6 inputs and one output, CLB 5 selects between the outputs of CLB 1 and 2. Many functions can be realized with just CLB 1 and CLB 6, but the CCM 2.5 must be able to calculate any possible function of these inputs, so all three CLBs are needed. To implement two functions of the same 6 inputs, CLBs 1 and 2 can be repeated as CLBs 3 and 4. It seems that CLB 5 now calculates 2 functions of five inputs each, but it actually calculates two functions of 3 inputs each. To implement a 7-input function, CLB 7 is added. To implement 2 functions of the same 7 inputs, all CLBs in the figure are needed. For more inputs, the number of CLBs that are needed to keep the circuit fully generally programmable rises very quick. This is shown in table 10.1. In the rest of this chapter, we will not show the schematics of each logic module that is discussed, but will simply state the number of CLBs that is needed.

1.2.1 The block RELATION[i]

This block calculates the two signals $REL_j[i]$ from the 4 input bits $A_j[i]$ and $B_j[i]$. However, it is necessary to be able to switch

# of inputs	#of outputs	# of CLBs
se 5	1	1
se 5	2	2
6	1	3
6	2	5
7	1	6
7	2	11
8	1	12
8	2	23
9	1	24
9	2	47

Table 1.1: *Number of CLBs for a universal logic module*

between the *relation* and the *prerelation* without reprogramming the FPGAs. Also, two other relations are needed to detect empty resultant cubes when the processor is operating in the *Pipeline* mode. In the CCM 2.0, two signals REL_CU_j from the CU select the correct four bits that denote the relation that is to be used. This is realized with a multiplexer outside the ILU. The four output bits of this multiplexer are sent to the data inputs of the multiplexer that implements the block RELATION in the CCM 2.0. In the CCM 2.5, the block RELATION is a universal logic module with 6 inputs: $A_j[i]$, $B_j[i]$ and REL_CU_j. Please observe that there are only two REL_CU signals for the complete ILU. Since there are two outputs $REL_j[i]$, 5 CLBs are needed.

We expect that four different relations of four inputs will be sufficient for future operations. Execpt when the CCM operates in pipeline mode, only two relations are used. To add an extra input would almost double the number of CLBs to 11 CLBs.

1.2.2 The block IDENT_MAIN[i]

For the CCM 2.0, the block IDENT_MAIN has 5 inputs, $REL_j[i]$, AND_OR, $CARRY[i]$ and $CONF[i]$. It has 4 outputs, $CARRY[i+1]$, $CONF[i-1]$, $VARIABLE[i]$ and $CNT[i]$. It will not be necessary to use two different methods of counting specific positions for the same operation, so there is only one signal $CNT[i]$. We already observed that the single $CARRY$ and $CONF$ signals were insufficient for some operations. To check the realation $A < B$, for example, two $CARRY$ signals are necessary. There are three possible situations that must be indicated to IT[i]:

$A < B$ for the ITs up to IT[i-1], the relation is true for the ITs up to IT[i-1].

$A = B$ for the ITs up to IT[i-1], the relation is undetermined for the ITs up to IT[i-1].

$A > B$, the relation is not true.

We want to keep the iterative signals between the ITs symmetrical, so we not only add a second $CARRY[i]$ signal, but a second $CONF[i]$ signal as well. This brings the totals to 7 inputs and 6 outputs. This can be realized with 33 CLBs. However, the signal CNT does not have any iterative signals as inputs in the CCM 2.0. In the CCM 2.5, it is generated by a single CLB, with $REL_j[i]$, $VARIABLE[i]$, one $CARRY[i]$ and one $CONF[i]$ as inputs. This saves 4 CLBs. Another consideration was the fact that in the CCM 2.0, the signal $VARIABLE[i]$ equals $CONF[i-1]$. This is sufficient only for the specific CCM 2.0 operations, and severely limits the number of other applications that can be realized. Therefore, $VARIABLE$ in the CCM 2.5 is a function of all 7 inputs of IDENT_MAIN, which requires 6 CLBs.

The possibility of 3 $CARRY$ and 3 $CONF$ signals has been rejected, since it would require to many CLBs. Each iterative signal would need to be a function of all inputs of the block IDENT_MAIN. This would result in 6 functions of 9 inputs each for the iterative signals alone, which would require 141 CLBs! To realize this we would only be able to program two ITs in the largest device of the XC3000 series, the XC3090 with 320 CLBs. If we would implement an asymmetrical circuit with 3 $CARRY$ and 2 $CONF$ signals, still 58 CLBs would be needed, with questionable gain in the number of possible applications. Other configurations with some signals as functions of less than all inputs have been considered, but all resulted in 50 CLBs or more. It is quite possible to realize an iterative circuit with 3 iterative signals in each direction with less CLBs, but only if it is one specific circuit. If the circuit should be univerally programmable, 2 signals in each direction is the practical limit.

1.2.3 The block STATE[i]

The block STATE[i] has 4 inputs in the CCM 2.0. Two of these inputs are the clock signal $REQUEST$ and the reset signal $CLEAR$ from the CU. These control the two flip-flops that hold the signals $STATE_j[i]$. The other two inputs are $VARIABLE[i]$ and $NEXT[i]$. Besides the $STATE_j[i]$ signals, there are 4 outputs, which are the two signals $SLCT_j[i]$, $NEXT[i+1]$ and $READY[i]$. To make the iterative circuit symmetrical in the CCM 2.5, a signal $PREVIOUS[i-1]$ was added as the counterpart of $NEXT[i]$. We will see that this did not require an extreme number of CLBs. Since the signal $CLEAR$ is a global reset signal, it is not necessary to add it as an input to all functions. Only the flip-flops and $READY[i]$ have it as an input, as is the case in the CCM 2.0.

Two CLBs are used to generate the excitation signals $EX_j[i]$, each with the 5 inputs $STATE_j[i]$, $VARIABLE[i]$ $NEXT[i]$ and $PREVIOUS[i]$. A third CLB holds the two flip-flops for $STATE_j[i]$, and has $EX_j[i]$ and $CLEAR$ as logical inputs and $REQUEST$ as the clock input. Another CLB, with inputs $STATE_j[i]$ and $VARIABLE[i]$, generates the signals $SLCT_j[i]$ which control the block OPERATION[i]. The bit $IR[7]$ of the instruction register, which indicates the *prime* operation, is not used in the CCM 2.5, which leaves a spare input for that CLB. It would not be very logical to connect $NEXT[i]$ or $PREVIOUS[i]$ to this input, since these iterative signals control the changes in $STATE_j[i]$. The $STATE_j[i]$ control the block OPERATION[i]. Since it is not likely that iterative signals will be needed for $SLCT_j[i]$, the extra input is left unconnected.

A single CLB generates $READY[i]$, which has $NEXT[i]$, $VARIABLE[i]$ and $REQUEST$ as inputs in the CCM 2.0. In the CCM 2.5, $PREVIOUS[i]$ has been added, while the fifth input is used for $CLEAR$.

Two CLBs are used to generate the iterative signals $NEXT[i+1]$ and $PREVIOUS[i-1]$ respectively. They each have $STATE_j[i]$, $NEXT[i]$, $PREVIOUS[i]$ and $VARIABLE[i]$ as inputs. It has been considered to add extra iterative signals, but this would result in 11 CLBs for $EX_j[i]$, 11 for $NEXT[i+1]$ and $PREVIOUS[i-1]$ combined, and 5 for $READY[i]$. We rejected this option, since it would mean a total of 22 additional CLBs.

1.2.4 The block OPERATION[i]

The block OPERATION[i] is very similar to RELATION[i]. The only difference are the two control signals, which now come from the block STATE[i] in stead of from the CU. OPERATION[i] is therefore a carbon copy of RELATION[i]. Four different operations can be realized, of which one, for the state *test*, is not used at the moment. We don't need this extra state to test the CCM 2.5, since it can be tested by normal FPGA programming hardware and software.

1.2.5 The block COUNT[i]

The pseudo-random counter is realized by 3 CLBs. The same number of CLBs can be used to implement a normal binary counter. In that case some of the inputs need to be rerouted. For both the pseudo-random as the binary counter, a sixth count signal can be added without the need of extra logic. However, the advantage of the binary counter is that '00000' is also a possible value. This makes a binary counter with only 4 lines sufficient for an ILU with 16 ITs, which would require only 2 CLBs.

1.2.6 The CCM 2.5 IT

Fig. 10.2 shows the block level schematic of the CCM 2.5 IT. The total number of CLBs that is needed, is 55.

1.3 MAPPING A CCM 2.5 IT TO A XILINX FPGA

When a single IT is mapped to an FPGA in a regular symmetrical pattern, it can easily be repeated without the need of automatic place and route software running for days. Such a mapping was done on the XC3090, the largest device of the Xilinx XC3000 series. The main purpose was to test if the interconnect resources of the FPGA would be sufficient. Due to the limitations of the CAD-software that was used, simple copying of this IT to create an ILU is not yet possible.

The IT was entered in the FutureNet DASH-LCA schematic capture program, and subsequently mapped by the Xilinx XACT FPGA software. The design was entered entirely in the form of the 'CLB' symbol from the XC3000 symbol library. This symbol can be configured like the actual CLB, by typing equations for the desired output functions. Most output functions contain just a dummy 5-input NAND, to save time. The locations of the separate CLBs was also entered, to force the CLBs to be mapped to certain places. However, the XACT Design Editor (version 2.60) does not allow manually changing the routing of the circuit. The only option was designating nets to be unrouted or (re)routed. This resulted in some rather interesting detours. Several signals were manually designated as 'long lines'. This was done either because they were clock signals that would have a high fan-out in a complete ILU, or to minimize the use of interconnect resources for signals that had to cross a long distance. One of thee clock signals, *REQUEST*, should be routed on a horizontal long line,

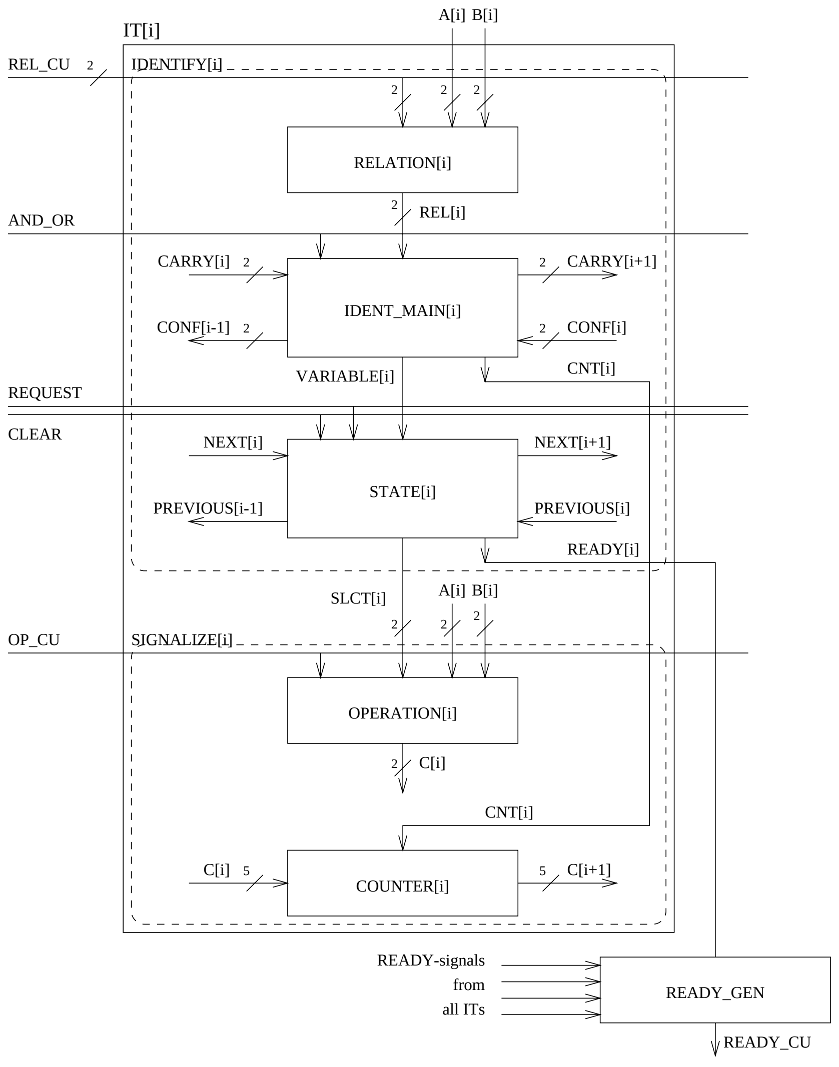


Figure 1.2: *Fig. 14.2 Block diagram of the CCM2.5 processor*

passing all ITs. This was achieved, but the signal also travels on a long line all the way to the bottom edge of the FPGA, and then all the way up again, apparently without making a new connection. One of the $B[i]$ inputs, which was designated to a long line because it needs to cross the length of the IT between the blocks RELATION and OPERATION, traveled all the way to the right edge of the CLB matrix, hopped on a long line to the bottom edge, and then an yet another long line back to the block OPERATION. After some rerouting, this was easily resolved, and I am confident that the used mapping leaves enough interconnect to keep signals within the IT. The automatic routing program will probably do better if the empty space is occupied by other ITs.

At the moment, it is not possible to copy ITs with this mapping to the rest of the FPGA, since every single CLB and output block has been mapped by hand. Perhaps better results can be achieved with a hardware description language such as PALASM or VHDL. The XC3090 FPGA has a matrix of 20 by 32 CLBs, while the IT occupies4 by 14 CLBs. Thus, there is room for 4 IT's in each FPGA chip, with some remaining CLBs free to implement some of the logic between the ILU and the CU.

Unfortunately, the ancient schematic capture program that was used does not support any laserprinters, and just a few dot matrix printers. The schematic could not yet be printed. It can probably only be printed with the use of a screen capture program, such as "grab.sys" of WordPerfect. Such a program was not yet available. The XACT software also caused some printing problems, which is why the CLB floorplan was printed with the 'print display' option. In this floorplan, which is shown in Chapter ??, each CLB has the name of the output signal on its X-output. CR stands for *CARRY*, CF for *CONF*, and so on.

Chapter 2

CURRENT AND FUTURE WORK

The purpose of the Cube Calculus Machine Version II (CCM2) is to execute logic operations in high-level formal systems in a real-time fashion. The CCM2 targets applications in logic optimization, logic synthesis, pattern recognition and image processing. It may be implemented as a coprocessor to a host computer, or developed as an application-specific add-in card, much like existing hardware accelerator cards.

Presented in this report in most detail has been the Iterative Logic Unit (ILU) of the CCM2. A 16-bit ILU, consisting of 8 iterative cells (ITs) has been realized using commercial, off-the-shelf Field Programmable Gate Arrays (FPGAs). The ILU is quite regular in its design, allowing equal portions (4 ITs) of the design to be divided between the two Xilinx Logic Cell Arrays (LCAs). FPGAs allow an ideal environment for university projects due to their off-the-shelf availability, PC-based design tools, and time-to-market capabilities. Design modifications may be made in hours versus weeks with the same design being realized in VLSI. Also, a single device may be used across multiple design applications consuming only minutes changing from one application to another.

The demonstration board developed by David Foote and Coen Engelbarts was constructed merely for the testing of the ILU. The current design is a 16-bit ILU shared between two LCA devices and only consumes 40 percent of each device. One of the first things that should be done is to extend the ILU to a 32-bit machine. This would double the size of the design, consuming 80 percent of each device. Currenly, 51 percent of the I/O Blocks (I/OBs) are being used for this design. By doubling the size of the ILU design, an additional 32 I/OBs are needed pushing the total IOB consumption to 75 percent. This design change may be made in a short period of time without many problems. Additional wiring would be needed as well as 6 additional dip-switch banks and 16 LEDs to handle the inputs and outputs respectively. This will take some to organize since the majority of user I/O pins becomes more scarce. This addition makes the CCM scalable in design by 16 bits. It is the first goal of our group now te evaluate how large functions can be loadable to DEC-PERLE-1 realization of CCM2. We assume that at least we will be able to handle 64 variables. This has been currently the most pessimistic estimation, and the most optimistic was 256 binary variables. We plan to re-use the bus interface circuitry and CU to interface with a host computer from previous DEC PERLE designs. Much of their software will be also used and only a tiny software will have to be designed to allow the user to input arrays of cubes and receive resultant values back from the board, storing the values in specified registers.

Chapter 3

BOOLEAN FUNCTION PROCESSING AND ARCHITECTURES

A new concept of Boolean Function Processing (BFP) is introduced, being an analogue of Digital Signal Processing (DSP). It is shown that DSP provides a powerful methodology and building blocks to realize an entirely new class of computers, which, by analogy, will be called Boolean Function Processsors. It is presented how the DSP architectures and BFP architectures can be combined to build comprehensive, real-time systems to solve a variety of important combinatorial problems. This chapter attempts to outline an entirely new concept of computing and to set basic ideas for methodology to design such computers. Several examples, including: graph-coloring approach to Boolean minimization, spectral-based multi-level Boolean decomposition with EXOR gates, minimization of Boolean relations, realization of multi-valued mixed polarity Reed Muller Forms, and realization of Brayton's Finite State Machines, are used to illustrate our approach. The new concept of Function-Programmable Computers, being enabled by this approach is also briefly introduded.

3.1 INTRODUCTION

The Digital Signal Procesing (DSP) methods were for long not widely applied in practical applications until the (relatively) inexpensive digital processing VLSI building blocks and processors have arrived in the 1980's. Recently, very sophisticated DSP algorithms are efficiently and inexpansively build and applied in the areas of control, communication, image processing, and instrumentation. DSP becomes a new and powerful paradigm for designing a wide variety of digital systems. A growing family of basic building blocks is now available that includes dual-port memories, FIFO's, ALU's, multipliers, complex multipliers, multipliers-adders, CAM's, FFT processors, general-purpose DSP processors, convolvers (convolution processors), digital FIR and IIR filters, correlators and other. The DSP theory gives a deep and comprehensive methodology to design systems from high level specifications and constraints. Moreover, each year brings many new programming tools and comprehensive DSP software design systems.

In these authors opinion there are three main reasons of DSP success:

- existence of a wide market for products,
- existence of good supporting theory,
- possibility of inexpensive hardware realization of a wide category of algorithms with a relatively limited number of VLSI building blocks.

One of important principles of DSP are those of data-flow, pipelining, fine-grain parallelism, SIMD, and systolic processing. For instance, usage of simple data-flow architectures (corresponding to programs with no "if" and "case" statements) permits to use Harvard architectures and recursive architectures with address generators that are both very fast and relatively inexpensive, so that a power of a Cray-class supercomputer is available for a small fraction of cost for selected applications; when they can be described using signal flow-graphs.

The basic concept of DSP architectures is that of Data-Flow (Signal) Graph. Several methodologies exist to convert such graphs to hardware realizations.

Another reason of the success of DSP building blocks is that they permit to realize *in hardware the most important and often used DSP operations* like multiply, complex multiply, multiply-add, trigonometric functions, and so on.

Let us observe that the success of DSP methods and hardware was first modest and restricted to few areas of aerospace industry.

Next the methods have spread widely to control, image processing, instrumentation, and various branches of military electronics. The proceedings of Annual IEEE Conferences on Acoustics, Speech and Signal Processing, as well as ISCAS Conference give a real tribute to the width and depth of applications. This technology is now where the microprocessors were ten years ago, and is growing extremely fast, so it is expected that the cost of the DSP building blocks will be essentially reduced in the near future.

Recently, there is a growing interest in neural nets, learning, genetic algorithms, real-time AI, logic synthesis and hardware accelerators. However, there are two main disadvantages that are slowing down development of respective hardware accelerators:

1. no comprehensive and unified theory and methodology to develop them,
2. no set of readily available basic building blocks, other than the general purpose digital components.

When we compare the situation of CAD hardware accelerators with the glowing success of DSP, the following fundamental questions, therefore, naturally arise:

1. "why don't we emulate the success of DSP?"
2. "can we use the DSP methodologies, hardware and experience to process Boolean functions?".

Let us first observe, that as documented in previous chapters, there were already single attempts at using DSP methods to design CAD hardware. For instance, Sahni proposed a systolic design rules checker with recursive architecture, Perkowski proposed a systolic satisfiability checker with SIMD ring architecture, Sasao proposed a PAL-based tautology checker, Rutenbar proposed to use the image processing archtecture for various physical design tasks. If we will be able to develop a methodology and corresponding set of building blocks similar to DSP, we will be able to build accelerators much more inexpensively, and with smaller research/development effort.

In this chapter we will show that:

1. All DSP VLSI building blocks can be used for designing CAD accelerators, especially for logic synthesis.
2. Much of the DSP theory can be used to design such accelerators.

3. We propose few VLSI blocks to be added to the building blocks repertuar.

4. Our methods are applicable to one- and multi-output, binary or multi-valued input, binary or multi-valued output Boolean functions.

5. The possibility of real-time processing of Boolean functions will permit to design a new category of computers: computers with programmable both control (like in RAM-based microprogrammed units) and data path. We will call them Function-Programmable Processors.

3.2 BOOLEAN FUNCTION PROCESSORS

In the area of digital signal processing, operations are executed on signals being vectors or arrays of integers or floating point numbers. In Boolean minimization, the operations are done on data being vectors and arrays of binary vectors. A well-defined and commonly used theory of Boolean functions processing is used, that uses the so-called multi-valued input arrays of cubes. On the other hand the spectral approach to logic synthesis [?, ?, ?, ?] operates on vectors and arrays of integers or tuples of integers, similarly to DSP. Many combinatorial algorithms operate on matrices, graphs and other data structures that can be also represented using binary vectors and arrays. This is our first important similarity - an observation that is a basis of this chapter.

By Boolean Function Processing we will understand a new, proposed here area of research that will consider algorithms, structures, and methodologies to design Boolean Function Processors.

The building blocks that are used in DSP are:

1. Memory structures for data flow, pipelined, parallel and systolic architectures: FIFO's, stacks, dual-port memories, shift registers, video RAM's, Content Addressable Memories (CAM's).

2. Data Order Changing Operators: sorters, absorbers, multiplexers, demultiplexers, data broadcasters, data collectors.

3. Controllers: SIMD architecture controllers, Address Generators.

4. Special DSP function processors: Convolvers, FFT processors, FWT (Fast Walsh Transform) Processors, correlators, Filters, Adaptive Filters.

5. General Purpose Arithmetic Processors. image/graphics processors, CODEC's, trigonometric and logarithmic processors.

Currently, we see three different methodologies of designing accelerator architectures.

1. DSP methodologies can be used with minimum changes to design architectures based on spectral methods.

2. Cube calculus algorithms, are first rewritten to data flow graph formalism, that is next mapped using DSP methods to hardware.

3. Combinatorial algorithms, like satisfiability, tautology, graph coloring, shortest path, maximum clique, covering closure, are solved in adaptive, neural like, self-optimizing structures.

We want to develop a comprehensive theory of Boolean Function Procesors, based on ideas of Digital Signal Processing, Cube Calcus, spectral approach to logic synthesis, and general methods of solving combinatorial problems, including matrix algebra, integer, linear and dynamic programming. As the first attempt, the three above methodologies will be presented and illustrated with examples.

3.3 DSP ARCHITECTURES

By DSP Architectures we will understand all the CAD accelerator architectures that use only the blocks of typical DSP applications. To design such architectures one does not have to design special building blocks. It is assumed that all functions that cannot be mapped to DSP blocks, like conditional operations, are implemented in microprocessors or general purpose DSP chips, such as Texas Instruments TMS 320. All Boolean and vector operations, like AND, OR can be implemented either in microprocessors and general-purpose DSP chips, or in special DSP chips, like Honeywell's DASP.

We will assume that the DSP algorithm is represented as a hierarchical data-flow graph with nodes that are of one of the following types:

1. basic DSP operation: multiply, multiply-add, subtract, etc. or a address-generator-based data-flow component, like Honeywell's DASP chip.

2. node with conditional operations, which require general procesor type architecture.

3. hierarchy of nodes of types 1 and 2.

There are basically two groups of algorithms that can be implemented with DSP architectures:

1. Algorithms based on spectral methods [?]. Such algorithms use all blocks, like FFT processors, corelators, etc. They need, however, also to realize few operations in general purpose blocks.

2. Other algorithms that make use of logical and other special vector operations that are available in DSP processors.

Before we will present the methods to design the first kind of architectures, let us analyse the analogies that exist between DSP and Logic Synthesis.

A. DSP.

B. Logic Synthesis

1A. digital signal

1B. Boolean function

2A. digital signal in time domain

2B. truth table of Boolean function. Array of cubes

3A. spectrum of a digital signal (Fourier, Cosine, Slant, Haar, Walsh, Karhunen-Loevy)

3B. spectrum of Boolean function (Walsh, Haar, Christenson, Reed-Muller, Perkowski/Falkowski new transform).

4A. signal correlation function

4B. complexity measure of a Boolean function

5A. cross-corelation of two signals

5B. measure of similarity of two Boolean functions

6A. digital filter

6B. decomposed component of some frequencies of a Boolean function. Many types of filters for many kinds of transforms exist that creates various possible decompositions.

7A. adaptive filter

7B. method of decomposition that optimizes the first plane of multi-level realization accoridng to some power (least-mean squares) criteria.

8A. cascaded adaptive filter

8B. method of cascaded decomposition into linear (EXOR), NAND, NOR, NOT, OR planes.

9A. convolution of signals $X * Y = FFT^{-1}$ [FFT(X) . FFT(Y)] many applications in vision , speech procesing, etc.

9B. convolution of Boolean functions: X * Y = FFT^{-1} [FFT(X) . FFT(Y)] I don't know yet.

10A. image compression

10B. compression of Boolean function (perhaps for some probabilistic computers)

11A. image coding

11B. Boolean minimization into disjoint cubes (Falkowski algorithm).

12A. inverse transform from a spectrum to a coded image

12B. Falkowski's inverse transforms (Walsh, Reed Muller, Perkowski/Falkowski to disjoint cubes.

13A. Besslich transform

13B. Generation of prime implicants.

It would be good to look for more analogies like the above.

3.4 PRINCIPLES OF DATA-FLOW ARCHITECTURES

Any textbook on DSP or company data sheet of DSP building blocks presents excellent examples how to realize pure data-flow. As an example, let us consider the data-flow graph from Fig. ??. The inputs are vectors [A], [B], [C], and [D], each of them of length n. The output is the vector [F] of length n+1 and equation

[[[A]+[B]]*[[C]+[D]]] & [[A]+[B]]

The scalar h is the multiplication reduction operator of the vector sum of vectors [A] and [B]. The vector [E] is the vector product of the vector sums: [A] + [B] and [C] + [D]. Vector [E] is next concatenated (operation &) with scalar h, where h is the and-reduction operator (denoted by) of vector sum of [A] and [B].

This data flow can be realized in an address-generator based recursive architecture as follows (see Fig. 3.2). It is assumed that the building block, Processor, has two adders and two multipliers in the structure from Fig. ??. In the first pass the values of [[A]+[B]] and [[C]+[D]] are calculated and stored in memories. In the second pass the values of [E] and h are created. The concatenation is created by Address Generators. As we see, all operations require only instructions from Address Generators to Memories and to Processor. No conditions are checked and used to modify the data flow.

Now we are ready to present a principle of using DSP hardware for Boolean minimization. Boolean function of n variables can be represented as vector of 2^n of integers (Better method, of using cubes, is presented by Falkowski and Perkowski in [?], but it is not discussed here for brevity).

Let us observe that most of them can be used for general-purpose processing of vectors and arrays. Also, the building blocks that are specific to DSP, like FFT processor, have applications in spectral-based logic synthesis methods. Therefore, all the above blocks will be used by us to design accelerator architectures, especially BFP's.

Additionally, the following blocks will be added.

Special BFP function processors: Cube Calculus Processor, Satisfiability Processor.

1. basic DSP operation: multiply, multiply-add, subtract, etc. or a address-generator-based data-flow component, like Honeywell's DASP chip.

2. node with conditional operations, which require general procesor type architecture.

3. hierarchy of nodes of types 1 and 2.

Any textbook on DSP or company data sheet of DSP building blocks presents excellent examples how to realize pure data-flow. As an example, let us consider the data-flow graph from Fig. 7.2. The inputs are vectors [A], [B], [C], and [D], each of them of length n.

The output is the vector [F] of length $n+1$ and equation:

F = [[[A]+[B]]*[[C]+[D]]] & [[A]+[B]]

The scalar h is the multiplication reduction operator (denoted by) of the vector sum of vectors [A] and [B]. The vector [E] is the vector product of the vector sums: [A] + [B] and [C] + [D]. Vector [E] is next concatenated (operation &) with scalar h.

This data flow can be realized in an address-generator based recursive architecture as follows (see Fig. 3.2). It is assumed that the building block, Processor, has two adders, two multipliers, register file, and buses, in the structure from Fig. ??. In the first pass the values of [[A]+[B]] and [[C]+[D]] are calculated and stored in outside memories run from Address Generators. In the second pass the values of [E] and h are created. The concatenation is created by Address Generators again. As we see, all operations require only instructions from Address Generators to Memories and to Processor. No conditions are checked and used to modify the data flow.

Now we are ready to present a principle of using DSP hardware for spectral-based Boolean minimization. Boolean function of n variables can be represented as vector of 2^n of integers (Better method, of using cubes, is presented in [Falkowski/Perkowski], but it is not discussed here for brevity). This Boolean function is an input to a Fast Walsh Transform block FWT (Fig. ??). As it is well-known FWT looks for correlations with Walsh functions, which are exors of all groups of variables. The output of the FWT is the memory wih 2^n spectral coefficients. The spectrum is now adaptively filtered to find the frequencies that contain maximum of signal energy. The frequency S_{abc} was found. This frequency is now tunning an adaptive digital filter. The output of the filter is a residue function, which will be now subject to new minimization. This time a Perkowski/Falkowski (find better name) spectral transform is executed on the residuum. This transform is defined exactly the same as the Walsh transform, but correlation with AND functions of all combinations of arguments is now looked for. The new residue is 0, so the network is found. The entire process is illustrated in Fig. ??. All details to understand this method will be given in next section. Here we want to explain the basic principle only.

3.5 CUBE CALCULUS ARCHITECTURES

Cube calculus architectures are basically hardware realizations of cube calculus algorithms that assume the data-flow operations as often as possible. We assume also special hardware realizations of several basic cube calculus operations for multiple-valued logic:

1. consensus,
2. inhibition (negation), sharp and disjoint sharp,
3. xlinking [?, ?],
4. and, or, not,
5. absorption, inclusion, supercube,
6. - several other cube calculus iperations.

Other operations on cubes and binary vectors that we need, and that are not yet realized in hardware are the following:

1. calculate the number of ones in a vector,
2. calculate the number of selected symbols (0, 1, X - for binary logic; 0, 1, 2, 3, ... , 01, 02, ... 0123 = X for quaternary logic, etc.),
3. replace the symbol V to symbol U in a cube,
4. calculate the number of the first bit 0, or 1 in a binary vector,
5. calculate the number of the first symbol 0 =10, 1=01, X=11, E=00 etc., in a binary cube,
6. calculate the number of the first symbol 0 =100, 1=010, 2=001, 01 = 110, 12=011, ... in a ternary cube, and so on.
7. generate the binary vector with a one on the position of the first from left one in an input binary word.

8. generate the cube with ones on the position of the symbol from left an input cube.

9. generate a binary word with one in a random position of ones from a binary word.

A PSU team is now designing a complete CCM2.0 to be used in BFP's. This processor together with a sorter can be used to design a reverse-tree architecture for solving the satisfiability problem and related combinatorial and logic synthesis problems [See paper Ho/Perkowski in ISCAS'89 []]. There is also a need to design a general purpose VLSI sorter for binary vectors and cubes, that would sort using various criteria, and would absorb or remove idenitical cubes while sorting with respect to their costs. (See examples in [?] and [?, ?]).

As an example of this type of an architecture for a cube calculus problem, let us discuss the creation of the coloring graph for the PLA minimization problem [?, ?]. For simplification of presentation, we assume below a binary function represented by positive and negative minterms, however there exist an algorithm that uses multi-valued cubes. It can be easily shown that the entire algorithm from Ciesielski [?] can be relized as ideal BFP, without conditional statements.

3.5.1 GENERALIZED MULTIPLE-VALUED CUBE CALCULUS.

What all "logic machines" have in common is that they use special hardware to do some kind of processing of Boolean or other switching functions. These tasks include: evaluation, Boolean operations such as intersection or complementation, checking for tautology or satisfiability, verification, solving logic equations, optimizing decision functions (like Petrick or its generalizations), performing resolution and unification, operating on binary images and quadtrees represented as switching functions, stack filters, data-base operators, inferring facts in logic, and many other.

Many well-known problem reductions exist that support our approach both theoretically (reductions of NP-complete problems [105]) and practically (several CAD and Operations Research algorithms).

It was observed that all NP-complete problems can be reduced to few generic problems such as graph coloring, maximum clique, shortest path, SOP to POS transformation, etc. Those in turn are reducible to basic binary logic problems such as satisfiability

[Garey]. Solving the Satisfiability problem can be next reduced to using some subset of the cube calculus operations such as sharp, absorption, intersection and other.

It can be shown that the absolute most of problems in logic synthesis, test, graph theory, problem solving, image processing, robotics, Operations Research, logic programming and other can be reduced to combinations of basic binary cube calculus operations. Since such calculus is a special case of the multiple-valued cube calculus [350, 358], and since several practically important problems can be directly reduced to non-binary cube calculus operations our interest is on multiple-valued cube calculus. Multi-valued cube calculus describes a logic with multi-valued inputs and binary outputs. We create an extended formal model of such calculus including new operations such as several kinds of crosslinks [137, 295]) and consensuses (consensus [353], asymmetric consensus [180]). Several recent papers reduce many logic synthesis, testing, verification, high-level synthesis, scheduling, parallelization, and physical design problems to logic decision or optimization problems listed above. Especially, many problems have been reduced to (Boolean and multiple-valued logic) satisfiability, tautology and inference (usually unification). In particular, set covering, binate covering and several physical design problems were reduced to satisfiability. In theory, every combinatorial problem can be reduced to it [?, ?, ?]. Many recent research papers from U.C. Berkeley and U. Colorado are based on tautology. Approach based on unification is presented in [?]. Very many other reductions to operations efficiently realized in our machine can be found in recent DAC, ICCD, ICCAD, ISMVL, ISCAS, ICASSP conferences and Logic Synthesis workshops, and there is even no space here to list them all.

There are two representation methods for switching functions used in logic synthesis programs: "(Binary or Multiple-Valued) cube calculus" and "(Binary or Multiple-Valued) Decision Diagrams". In this paper only the first one will be discussed, however the new variant of our machine (TP) allows to deal also with data other than the "multiple-valued input cubes" presented here, and includes the MDDs as well.

The binary "cube calculus" [?] has been extended for a logic with multiple-valued inputs [?, ?, ?, ?, ?] by Sasao, and is called a *positional cube notation.* This calculus has been used for many two-, three- and many level Boolean minimizers, tautology and satisfiability checkers, verifiers, programs for complementation of Boolean functions, synthesis of mixed and fixed generalized Reed-Muller forms, generation of prime, minimal and disjoint implicants, spectral transforms (Walsh, Reed-Muller, Arithmetic), and many other [?, 73, 1, 145, 169, ?, 355, 385, ?, ?, ?, ?, ?, ?, ?]. It is also useful in programs using the other efficient representation of Boolean functions: Binary Decision Diagrams and Multiple-valued Decision Diagrams. Multiple-valued Cube Calculus seems then to be one of the most general currently known internal representation of data in propositional and predicate logic, logic synthesis, logic programming, logic simulation and sequential evaluation of combinational logic, data-bases, image processing and several areas of AI and problem-solving. Methods based on it have been successfully applied in practically realized products. Analogously to other formalisms which were not fully appreciated before building special hardware for them (such as fuzzy logic, morphological image algebras or some digital signal processing methods) we may reasonably expect that special computers which will operate in Cube Calculus will find their "applications niche" among computer architectures. Implementing a hardware processor to operate in this calculus will have wide applications, including real-time AI and optimization, where recent devices such as "fuzzy chips" [?, ?, ?] find use.

3.6 THE TUPLE PROCESSOR.

In addition to operations on sets and numbers, known from CCM-2, our interest in solving discrete combinatorial prolems is in *finite formal systems* that have a finite (and from practical point of view - also limited) number of symbols as well as a limited number of operations. We assume that all operations on symbols can be decomposed to simple operations, which in turn can be described as tables (two-dimensional matrices). We are already familiar with such matrices describing sharp, consensus or other binary operators. This assumption allows to encode efficiently symbols as binary strings, and implement efficiently the matrices using ROMs, RAMs, CAMs, PLDs or other standard binary logic. Such systems will be called Limited Finite Systems (LFS). Let us observe that integer arithmetics is not an LFS since truth-table description of the addition operation is impractical for hardware realization, while (assuming small value of integer n) the modulo n Galois Field is an LFS, since such matrix description can be used to design the logic of modulo n operations.

The LFS formal systems include multi-valued cube calculus, fuzzy logic, matrix logic, several formalizations of truly multi-valued logics (in those logics both the inputs and outputs are multi-valued), several logics and algebras used in logic programming, multi-dimensional solid algebras, data-base and parallel algebras, image processing (for instance the image and morphological algebras), spectral transforms and signal processing. We propose to solve the LFS problems using the GMVCC model and corresponding GMVCC processors attached to the host. We are especially investigating a subset of GMVCC systems where the operations on each mv-literal in a cube are partitionable to small groups of bits.

Solving a problem with GMVCC hardware requires selection of an appropriate GMVCC formal model and next reduction of a high-level formal model to this model.

3.7 TUPPLE PROCESSOR: AREAS OF INTEREST AND FUTURE APPLICATIONS

Each word of TP will represent

1. a number (as in CCM-2),
2. a pair of numbers (an interval),
3. a set (as in CCM-2),
4. a symbol.

Each symbol can have various meanings, it can be simple or complex. Simple symbol can be associated with a symbol of multiple-valued algebra, modal algebra, or any discrete system (such as consistent labeling), in which the number of such systems is limited. In TP, which will be constructed using FPGA technology, particularly XILINX, we take into account the limits of technology. Since the cell of Xilinx has five inputs and can realize any cell of five variables and some functions of 6 variables, we restrict ourselves to symbols on three bits. This allows to use one or two cells for all two argument operations in algebras having up to 8 values ($2^3 = 8$). This is sufficient for most known algebras, except for 16-valued algebras used in simulation. However, we can still use complex symbols composed of simple symbols in the same way as the set operations in literals are composed in CCM2 from binary operations in ITs.

Our machine introduces the dynamically modifiable size; the operations on all words of an N-tuple will be done in parallel.

Our goal of introducing this computer will be achieved step-by-step by introducing a number of simpler architectures and algorithms for classes of problems. We will show and discuss all steps leading to our ultimate goal. However, at this point the reader must understand understand the Cube Calculus Machine 2 really well, to be able to generalize it.

The TP is a comprehensive generalization of ideas and architectures used in several areas:

- *Multiple-valued input "cube calculus",* an algebraical model popularly used to process and minimize Boolean functions. The TP uses a "positional cube representation" which finds many applications in logic synthesis and combinatorial problem solving. It supports all operations including sharp, consensus, supercube and crosslink, as well as many new operations.

- Multiple-valued notations for *truly multiple-valued* (both input and output) logics. Operations of many multiple-valued logics such as the *modal logic, matrix logic, quantum logic* and other will be realized.

- *Fuzzy logic.* Although it is just a one more multiple-valued logic, we pay special attention to it since it finds recently several practical applications.

- *Associative N-tuples* used in Artificial Intelligence and Image Processing to represent and process knowledge.

- *Grey-scale* and *binary Morphological Image Algebras.* They are used to process images, encode them and recognize image properties.

- *Data-Base Algebras* used for hardware realization of relational data bases.

- *Digital Simulation algebras* used to simulate computers and digital systems in special hardware accelerators [].

- Search, evaluation and inference mechanisms of *relational, functional and logical languages* such as Lisp, Prolog, and Equational Logic.

- Formalism used to solve *graph theory problems.*

- General formalisms to represent and solve combinatorial problems [?, ?, ?, ?, ?].

- Formalisms and architectures for *computer image processing and image recognition.*

- Spectral transforms (Fourier, Walsh, Haar, Arithmetic, Adding, Hough) and signal processing ideas (convolution, digital filters).

- *Neural-like learning networks and recognizers.* Let us observe that the taught binary neural network is the same as multi-input, multi-output Boolan function. Learning of the network is the process of constructing this functions (possibly by modifying some initial function). This can be generalized to arbitrary discrete neural nets, which correspond to multiple-valued input, multiple-valued output switching functions. It is possible that using the cube calculus operations the learning process will be faster, think for instance about the n-input EXOR function, which is very hard to learn by neural nets, and is trivial to learn using for instance crosslink operator.

- Programmable architectures like those base od on Xilink chips - [?].

- *Thinking Machines Connection Machine.* Data-Programming.

- Transputer, GAPP, systolic processors, iWARP and DSP architectures (AIM Sharp).

- Brayton's approach to using mv CC to solve combinatorial problems.

  Another aspect of this architecture is the implementation of the processing unit TP as a microprogram-controlled iterative ring of FSMs. There are two levels of parallelism: micro-parallelism in the N-tuple, macro-parallelism in data-flow structures operating on N-tuples. This seems to be a totally new concept in computer architecture.

3.8 AN EXPANDABLE, FOUR-DIMENSIONAL FPGA COMPUTER FOR NON-NUMERICAL PROGRAMMING

We present the concept of Universal Logic Machine, a generalization to standard computers, logic machines, associative processors, VLIW, and other architectures, that is easily realizable in FPGAs because of its regular hierarchical structure of Virtual Processors, based on iterative logic, universal logic modules and cellular automata. Application of the machine to generalized satisfiability is also discussed and simulation results are analysed.

3.8.1 INTRODUCTION

FPGA technology proved itself already practical in two areas: to realize parts of digital systems (especially computers), and in the area of fast prototyping/hardware emulation of digital systems.

In the *first area* there are three types of products:

1. In case when a product of not high volume is expected, FPGA is preferred to ASIC to decrease the cost. A verified product that is in high demand is next migrated to an ASIC.
2. One piece of hardware plays several roles. This is done to decrease the cost or power consumption, to allow further modifications by software upgrades, decrease size, or any combination of the above.
3. The circuit cooperates with a smart sensor or other analog/digital/mechanical subsystem which is changing its operation with time and needs therefore tunning and modification as the time progresses. Or, this subsystem "user-upgradeable".

The *second application area* is to replace system's software simulation with its emulation in hardware. This allows for quick design cycles and concurrent development of software and hardware. Also, it allows to observe effects that are not possible in standard simulation, for instance observing images, or hearing to sounds while developing multi-media systems. Companies like Quickturn [323], PiE Design Systems [313], and INCA [420] are selling massive general-purpose hardware emulation engines, and a patent for another concept of a general purpose hardware emulator has been awarded to employees of Mentor (now in Quickturn) [226].

In addition, several other companies are developing all kinds of in-house specialized systems for fast prototyping and experimental FPGA-based emulators.

Both these application areas will have influence on the arising arena of *the FPGA computers* outlined below. Hardware emulation will perhaps migrate to VHDL and even higher-level specification languages. It will then merge with some aspects of the FPGA computers.

While these two areas can be treated as pretty well established, with quickly increasing and potentially large markets, there has been much controversy about another concept made possible by FPGAs - the *configurable computer architectures.* Below we will call them *"FPGA computers".*

Such computers have been proposed in the last few years by several research groups [3, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, 130, ?, ?, ?, ?, ?, ?, 425].

The main obstacle to accept the FPGA computer technology will be software. As a comparison, there are no widely accepted methods for writing pipelined, or parallel programs, or converting sequential to parallel programs. According to the existing approaches, FPGA-based computers programming will require at least these techniques [?, ?, ?, ?, ?, ?, 130, ?, ?, ?]. Similarly, there are no widely accepted and commercially available tools for converting high-level descriptions (such as VHDL) to systolic processors. Even much simpler related approaches such as microprogramming, which are known for years in industry and for which a number of tools have been developed, are difficult, time consuming and less used in practice than their already existing technologies allow. New methods will be also required, such as transformation of recursion and complex data structures to hardware. All these techniques and many others would be necessary to realize a truly universal FPGA computer, unless some powerful niche will be found with narrower software base, or unless some breakthrough in programming technology will be found. Again, finding an "application niche" seems then to be the key to the step-by-step development of this required software technology to support the FPGA computers.

A notably higher mathematical and computer science sophistication of engineers and program developers will be needed, and the concepts such as program verification, theorem proving, software/hardware codesign, transformational design, and very high level languages will have to be acquired. This can be done (see how the concepts of logic synthesis and VHDL have been acquired in the eighties) but will take much time and will require new educational efforts.

For easier acceptance, the FPGA computers should adhere as much possible to all existing software/hardware, CAD-tool, interface, language, and mechanical standards.

Below we will propose a concept of an FPGA computer that attempts to create a possible niche in the applications market and tries to face, at least some, of the challenges mentioned above.

3.9 EXPANDABLE FIELD-PROGRAMMABLE CELLULAR TISSUE AND MAPPING OF VIRTUAL ARCHITECTURES TO IT

As observed also by Hartenstein, there is a third, better way between the two extremes of a totally electrically programmable logic (FPGAs), and a totally hard-wired one (microprocessors). Here we propose a regular computer architecture with a fixed hierarchical floor-plan in which some blocks and some connections are electrically programmable. We call it *Field-Programmable Cellular Tissue (FPCT).* It is a crossfertilization of hard-wired data processors and Field Programmable Gate Arrays. It is a "generic data processor", personalized to its specific functions by electrical programmability of some of its component blocks and/or connection networks. Although we apply this concept in our CCM2 realization, the concept of FPCT is a general one and not necessarily related to CCM2 class of applications (see also [?, ?, ?, ?, ?, 130, ?, ?, ?, ?, ?]).

To make our presentation simpler, we will write names of physical components of CCM2 in small lettters (configurable logic block (clb), section (quadrant) of a chip, chip, board, data-flow cage). The names of the general-purpose virtual components will be in capital letters (Point Processor, Cube Processor, Pipelined

Processor, Hypercube, Data Flow Architecture). The names of virtual components of particular application architectures will have star character in the beginning (i.e. *Inclusion_Processor, *GPF_Solver).

The physical CCM2 computer is shown in Figure 3.1. It includes the host, the processor boards and the interconnection network. The board is shown in Figure 3.2. It has an $n * n$ (4 * 4 array of Xilinx 3090 chips in our case) with constant physical connection pattern of two-dimensional array. Pins of each chip are separated to six groups, four of them go to neighbors, one to the local memory associated to this FPGA chip, and one to one of the board's controller FPGA chips.

We can think about such structure as one large two-dimensional array that is folded to strips with four processors in a strip and next folded again to squares, each square four by four processors. This realization allows to avoid special large and expensive mechanical constructions used by emulation companies. When more computing power is needed one purchases more boards, like now we can buy more memory on SIMMs. Future boards will also possibly include microprocesors, RAMs, Contents Addressable Memories, and DSP components.

Now, when we understand the physical structure of our machine, let us derive its general virtual structure. It is absolutely crucial to distinguish the *physical structure of the tissue* from the *virtual structure of the Data Flow* of the Universal Logic Machine, which is mapped to it, and an application architecture that is a particular "high-level personalization" of generic virtual Universal Logic Machine.

The virtual structure of CCM2 is based on the *hierarchy of processors,* which we will list from the simplest:

1. A *Iterative Cell (IT)* (called also *Variable Processor* in cube calculus terminology and *Point Processor* in image processing terminology). This processor can be as simple as an AND gate and as complex as parallel multiplier/adder.

2. A *Group Processor (GP)* (for group of variables in GMVCC) is an array of Variable Processors.

3. A *Cube Processor* (accepts GMVCC cubes), called also *Window Processor* in image processing. It is an array of Group Processors controlled by a Controller (CU).

4. A *Pipelined Processor.* It is a pipeline of Cube Processors.

5. A *Data Flow of Processors* is a data flow of Pipelined Processors.

Both IT and GP have no separate control and can only modify, in their state machines (being part of the cellular automaton of the Cube Processor) the instructions obtained from the neighbors and the controller of the Cube Processor. In current implementation the layers 4 and 5 have no controllers as well. Additionally Host is used for FPGA programming and other standard tasks.

In layers 1-4 each virtual processor of higher level is a structure of processors of one level below in hierarchy. This structure can be of one of the following types: one-dimensional array, one-dimensional array with bus, two-dimensional array, two-dimensional array with busses, three-dimensional array, hypercube, binary tree, trellis, and snake-folded one-dimensional array. On the lowest layer the rectangles of the architecture are combinational functions, state machines and macros. The special logic synthesis methods to minimize logic functions and machines to cellular, rectangle shaped tissues, such as in CLI 6000, have been implemented [435, 368, 288]. Each processor, or part of it, is a rectangle and has four sides: North, East, West and South, and has ordered ports on each side. Ports have user-assigned names and types (input, output, input/output, bidirectional, global_bus (going above and not touching internal logic)). In every level, processor can communicate with other processors only through abutting ports, or by regularly placed busses resulting from one- or two-dimensional placements of virtual structures. For instance, hypercubes are placed in the same way as Karnaugh maps are placed on the plane, which creates busses for non-abutting processors. The busses can be also user-declared. Some examples of hierarchical regular structures of virtual processors are in Figure ??.

The approach outlined above restricts the floorplans in that all *non-local* connections are always "single segment", vertical or horizontal, local or global, busses. All *local* connections are done by programming cells to wires, but this is done *only for abutting cells.* Some classes of problems have their respective structures; for instance, for cube calculus one-dimensional arrays are used, while for low-level image processing we use two-dimensional arrays. An arbitrary structure can be declared for layer 5, but this may require custom wiring of the interconnection network. From what was told above, down the Cube Processor layer, all connections are regular. One can think about the resources to which we are mapping as a continuous two-dimensional regular "tissue" of configurable logic blocks (clbs), to which both logic and connections are placed. With certain kind of physical interconnection network, corresponding to "twice folded two-dimensional array", the tissue extends to all clbs from all boards from the rack. Whenever an application architecture can be described as a syntactically correct virtual Universal Logic Machine, it can be also realized in physical CCM2 hardware. The approach does not lead to problems related to placement/routing, that are so painful in current emulators (non-routable designs, low usage of chips and boards).

The "FPGA Compilation" is somewhat similar to old-fashioned silicon compilation (Bristle Blocks, DPL, etc). It consists in finding the mapping from the virtual specification to the physical resources. The mapping is simplified thanks to the use of standard parametrizable operators in all levels, and the regular hierarchical floorplans. One can think about Virtual Processor specification as a dynamic floorplan assembled from a hierarchy of large and small rectangles, that can be not only scaled and have their width/height ratios modifed, but also shifted, rotated and mirrored as macro-blocks of clbs can be in the CLI 6000 layout [?, ?].

The Cube Processor has three kinds of data/control movements associated with it. The Vertical Movement (both data and control) is between Group and Variable Processors. In case of linear arrays it can be imagined as vertical signals and shifts between processors (see [287] and section ??). This movement corresponds to Cellular Automata type of communication. The Ping-Pong Movement is the "address-generator" controlled movement to and from memory in each processor (an idea from FFT processor from Sharp). The Pipelined Movement is from a Cube Processor to another Cube Processor in a Pipelined Processor. The movements are illustrated in Figure 6.

Each type of movement and associated geometry is like one dimension of our architecture. The architecture is *expandable in four dimensions:*

1. In case of one-dimensional arrays the Vertical Movement Dimension is expandable in a "bit-slice" way, by adding more Group or Variable Processors. For two dimensions this type of expansion is known from GAPP chips.

2. The Ping-Pong Movement Dimension is expandable by adding more memory to each processor and (option) modifying the address generator.

3. The Pipelined Movement Dimension is expandable by adding more Cube Processors to the pipeline (making it longer).

4. The Data Flow Movement Dimension is expandable by adding more Pipelined Processors to the Data Flow.

Physically, the expansions are done by electrical reprogramming, adding boards, upgrading RAMs, and changing ribbon cables. (We do not intent to do this early, but DEC-PERLE allows for this.).

How can one arrive at this kind of "generic regular architecture generator"? We have done it by comparing various architectures that have some kind of geometrical regularity. Let us for instance focus on standard processors. A standard processor performs various groups of operations: arithmetic, logic, shifting, etc. While the operations may vary from processor to processor, the general structure will not. It will always include register file with registers of the length of the word, respective buses, multiplexers, and interfaces, as well as some kind of ALU which operates on words. Although ALUs will vary, they will always have some blocks corresponding to basic units of information (like bits), which will be iterated for the length of the word. While for binary logic the units will be one bit wide, they will be n-bit wide for a logic with 2^n values. There will be also *carry signals* between these blocks for operations like shifting, addition, or comparison. All those processors can be then implemented as a regular hard-wired structure of: *hard-wired blocks* (such as registers and multiplexers) and *electrically programmable blocks.* The latter are some combinational or sequential circuits, which are not generic and of few types as clbs in FPGAs, but of various types and sizes respective to the assigned to them operational or control tasks of the entire architecture. Following this observation and comparing fundamental structures of microprocessors, VLIW Processors, string matching computers, associative processors, Geometric Arithmetic Processors (GAPP), and other special and general purpose pipelined, systolic and SIMD processors (DAP, CLIP, MPP, CM), as well as logic machines: MCCC [280], HART [?], GPFS [?] and CCM1 [?], we designed the generic processor system - CCM2 [287].

3.10 MECHANIZATION OF GENERALIZED MULTI-VALUED CUBE CALCULUS

Now that we understand the basic principle of virtual versus physical architectures, let us explore the relation of the virtual processors to the GMVCC, since the relation to image processing hardware and systolic processors is standard and better understood.

The Cube Processor operates not only on numbers but predominantly on generalized multiple-valued cubes. This architecture is based on a data path composed of cellular automata which take over the lowest level loop of sequential control, and are electrically programmable to perform various functions. In our CCM2 the data path has been designed to execute operations of "cube calculus", an algebraic model popularly used to process Boolean functions. CCM2 realizes efficiently all cube calculus operations such as sharp and consensus. The "positional cube representation" used by CCM2 can also represent Generalized Multiple-Valued Cube Calculus (GMVCC) which has applications in many computational methods based on logic. *The Multiple-Valued input, binary output Cube Calculus (MVCC)* seems to be one of the most general internal representations of data in propositional logic, logic synthesis, logic programming, logic simulation, data-bases, and several areas of AI and problem-solving. Generalized Multiple-valued Cube Calculus (GMVCC), our extension of MVCC [287], is even more powerful than the MVCC because it can represent multiple-valued input multiple-valued-output logic (called *truly mv logic).* This means that it can be used for real-time AI applications, image processing, genetic algorithms, fuzzy logic and logic programming. As it will be described, the heart of the Cube Processor is a one dimensional "bit-slice" data path constructed as an iterative network of electrically programmable *Cellular Automata (CA).* In fact, our concept of Cellular Automata is more general than one from literature [?, ?], since the excitation function of each register can be a combinational function of *all* registers from the array, and not only the *direct neighbors* (see Figure ??).

CCM2 word (cube, window) includes a flexible number of multiple-valued (mv) literals. While CCM1 executed only set-theoretical operations on literals [?], CCM2 brings simple and complex symbols as an intermediate level between bits and literals. These are realized by Group Processors. This allows to realize arbitrary truly mv logic and to deal with the literals being various kinds of numbers. CCM2 can also deal with data such as: number intervals, symbolic predicates, associative tuples, and multi-valued multi-output relations. Both the homogeneous and heterogenous groups of variables are possible (various kinds of Group Processors within a Cube Processor). The same is true on other layers of virtual processors. These capabilities greatly expand CCM2 semantics and the range of potential applications. It becomes close to vector machines, VLIW computers, associative processors, or logic programming machines.

All the known CC and MVCC software subroutines [?] process the literals sequentially, but for most of the literals the resultant cubes generated will have contradictions so that they will have to be removed later. CCM2 implements a completely new architecture to take advantage of the peculiarities of sequential cube calculus operations. The architecture is an iterative logic/cellular_automata array with "carry-like" signals running from left to right *and from right to left* through the iterative circuit of CAs. The fundamental advantage of this approach is that only cubes without contradictions are generated. The Cube Processor consists of a set of Group Processors, a Controller and a memory. The Group Processor is a (horizontal) linear cellular automaton build from *buiding blocks, ITerative Cells (ITs)*. Each simple symbol is processed by an iterative cell IT. The number of ITs is denoted by n, so that the number of bits is $2n$ and we can process n binary variables. Besides combinational logic each IT[i] includes a *Position State Machine (PSM[i])* that influences the local interpretation of the Controller's micro-instructions. In this sense each IT is a small processing unit that processes a part of a cube in parallel (or a Kit serially), and communicates with neighbor IT processors. Each IT[i] operates on arguments from its register file or memory, and some registers of neighbors. Each IT[i] is divided into four programmable blocks according to the function performed: *RELATION[i], PSM[i], MATCH_COUNTER[i],* and *OPERATION[i].* (See Figure ??).

MATCH_COUNTER is used in global cube data Counting Operations. It counts the number of satisfied predicates in all ITs. *MATCH_COUNTER[i]* adds 0 or 1 to *MATCH_COUNTER[i-1].* It takes predicate values from *PSM[i]* or *RELATION[i].*

OPERATION[i] creates bits of resultant cubes by performing the operation on bits of the operand cubes. This is where the matrix of the GMVCC operator is programmed in. It takes arguments from *A[i]*, *B[i]* and *PSM[i]*, and has two outputs *C[i]* and *D[i]* that go to register file and neighbors.

The idea of combining basic programmability functions and structures to create complex operations was observed and used also for other kinds of operations for which our machine was designed, for instance the morphological image processing. While the above observation allowed us to map the "variable loop" to one dimension of the CCM2 (horizontal communication inside a CCM2 Processor), other operation patterns are mapped to pipelined data movements (shifts) between several CCM2 Processors connected in a linear array - as in WARP computer of Kung. The principle is: *"Implement the lowest loop in CCM2 Processor, and the next from bottom loop (usually the "cube loop") in a linear array of CCM2 Processors"*. Similarly mapped are two-dimensional movements, horizontal (inside CCM2 Processor) and vertical (between CCM2 Processors) - as in SIMD meshes. Three-dimensional shifts (using as the third dimension RAM memories connected to CCM Processors) are mapped as in GAPP and image processing architectures. Additional advantages are created by using *Content Addresable Memories (CAMs)* instead of RAMs for storing the results from CCM2 processors.

Concluding, each GMVCC operation is a point in a multidimensional space created by a *Cartesian Product of Basic Programmable Features* (such as rel, after, before, relation_type, composition, pipelining, etc.). The number of all possible operations is extremely large. We were able to realize all known GMVCC operations, and many useful new ones. We have not yet found practical applications for the astronomical numbers of new operations created as a byproduct of this *"Cartesian Product"* concept. Automatic learning of the point and subsequent reconfiguration can be implemented per analogy with (cellular) neural nets.

3.11 CONCLUSION

The concepts of Universal Logic Machine and Field Programmable Cellular Tissue are very general and not restricted to the realization of CCM2.5 and CCM3 as illustrated above. While in CCM2.5 most of the electrical programmability is to reconfigure logic, one can also create system architectures where the configurability would be used primarily for communication. The reconfigurable architectures build from general-purpose FPGA chips [?, ?] were on several problems superior to supercomputers, and

proved the usefullness of the concept of electrical reconfigurability applied to special purpose massively parallel processors. We believe that the ideas outlined here and illustrated on CCM2 will allow for designing even more powerful FPGA computer architectures with new cellular FPGAs [3, ?, ?, 321, 322] and their fortcoming respective MCMs.

Chapter 4

COMMUNICATION-BASED OPERATIONS.

4.1 COMMUNICATIONS IN DSP APPLICATIONS.

Just as the DSP or AI architectures can be build from regular structures of few kinds of blocks through which data vectors are transmitted, the principle of CCM is to allow for pipelined/systolic execution of the most common operations on clists and cclists. Those operations include for instance the Cartesian Products, the dot products, the permutations, and the generations of all subsets or combinations of a clist. Micro-Programming of CCM consists then in assigning values to two kinds of control fields:

1. Those that describe the processing inside the CCM, for example, the operations on cubes and short clists described in section ??.

2. Those that describe the communication of the CCM with other processors and memories.

There are currently two styles of communication:

1. Address Generator based.

2. Data Flow based.

Address Generators are used to send out the address of the desired operand in Dual-Port Memories. The data flow style is based on FIFOs. In this case the CCM chip (CCM processor?) sends a signal to a FIFO to get the next cube. A Ping-Pong communication mechanism is used to transfer the data from Memory A to Memory B while the processing. Possibly, the data-flow in the processor can be changed for every change of direction. and this is controlled as well.

4.2 COMMUNICATION BASED OPERATIONS IN ULM

There are three main types of communication-based operations:

1. simple operations on a clist,
2. simple operations on two clists,
3. complex operations on a clist.

Ad. (1).
Simple operations on a clist include *Global Reduction, Mapping, Pairing, Selection, Inversion, Mapping with Insertion, Inversion with Insertion, Replacement, All Pairs with Reduction, All Ordered Pairs,* and other operations. Examples of some of those operations on clist A = { a_1, a_2, a_3 } are as follows.
NOTE:
In the following, { } denotes a clist, {{ }} a cclist, [ ] a cube, <> a group of variables, | [ ] | a set, ' a symbol. OPER and OP_CLIST are two-operand cube operations.

Global Reduction , in which a cube is created from a clist:

REDUCTION (A, OPER) = $[a_1 \; OPER \; a_2 \; OPER \; a_3]$

Examples of such operations in cube calculus are the supercube of all cubes from the clist, and the intersection of all cubes from the clist.

Mapping , for which the clist produces a new clist of the same size:

MAPPING(A, OPER) = { $OPER(a_1)$, $OPER(a_2)$, $OPER(a_3)$ }

Pairing , where the operation is repeated on subsequent pairs of arguments and the results are directly inserted in a new clist:

PAIRING(A, OPER, OP_CLIST) = { $OPER(a_1,\ a_2)\ OP\_CLIST\ OPER(a_3,\ \emptyset)$ }

Sorting , can be realized as repeated pairing.

Sorting_with_Absorption can be realized as repeated pairing.

Reducing Operations such as:

$ab \vee a \implies a,$

$a \vee a \implies a,$

$ab \oplus a \implies a\bar{b},$

or

$a \oplus 0 \implies a,$

can be used as OP_CLIST. Repetitive Ping-Pong transfer of a clist through a CCM with an exchange of neighbor cubes (based on a specified ordering relation between them) leads to sorting the clist. When the compare/exchange operation also executes cube absorption, algorithms such as POS $\implies$ SOP transformation by multiplication can be executed [?].

=item[Inversion], for which the clist creates a new clist of the same size:

INVERSION(A, OPER) = { OPER(a_3), OPER(a_2), OPER(a_1) }

All Pairs of a Clist:

ALL_PAIRS(A, OPER) = { $OPER(a_1,\ a_2),\ \ OPER(a_1,\ a_3),\ OPER(\ a_2,\ a_3)$ }.

Selection , is an operation known from CAM-based and associative processing. It selects cube(s) for which a certain Boolean function of literal-related predicates is satisfied. For instance the operation:

"$age > 34$" AND $character =$ '$good$" OR $properties \subseteq \{|$ $[\}wealth,\ good - contacts\{]\ |\}$"

operates on cubes with numeric variable *"age"*, symbol variable *"character"*, and set variable *"properties"* and selects all people who either are older than 34 and have good character, or have wealth and good contacts among their properties.

Replacement , is a similar operation. It does some cube operation on selected cubes such as adding $ 1000 to the numeric variable *"salary"* or adding set value "good-contacts" to the variable "properties".

The following operations of Cube Pairing and Cube Pairing with Reduction are executed on a clist and a cube:

CUBE_CLIST_PAIR(A, cube, OPER) = $\{ OPER(a_1,\ cube),\ OPER(a_2,\ cube),\ OPER($
$\}$

CUBE_CLIST_REDUCE(A, cube, OPER, OP_CLIST) =

$\{ OPER(a_1,\ cube)\ OP\_CLIST\ OPER(a_2,\ cube)\ OP\_CLIST\ OPER(a_3,\ cube)$
$\}$

Ad (2)
Simple operations on two lists include *Dot Product, Vector Product, Simple Cartesian Product, Sequential Cartesian Product, Convolution, Vector Product with Insertion, Vector Product with Reduction,* and other. Some of those operations on clists A = $\{ a_1,\ a_2,\ a_3 \}$ and B = $\{ b_1,\ b_2,\ b_3$ $\}$ are the following:

Dot Product , for which two clists create a cube:

DOT_PRODUCT(A, B, OPER, OP_CLIST) =

$[OPER(a_1,\ b_1)\ OP\_CLIST\ OPER(a_2,\ b_2)\ OP\_CLIST\ OPER(a_3,\ b_3)]$

As an example, OPER can be a cube intersection operation, and OP_CLIST can be a cube supercube operation.

Vector Product , for which two clists create a clist:

VECTOR_PRODUCT(A, B, OPER) = $\{ OPER(a_1,\ b_1)\ ,\ OPER(a_2,\ b_2)\ ,\ OPER(a_3,\ b$
$\}$

Simple Cartesian Product , where two clists create a clist (in particular representing a matrix, or a set of pairs):

SIMPLE_CARTESIAN_PRODUCT(A, B, OPER) = $\{ OPER(a_1,\ b_1)\ ,\ OPER(a_1,\ b_2)$
$OPER(a_2,\ b_1)\ ,\ OPER(a_2,\ b_2)\ ,\ OPER(a_2,\ b_3)\ ,\ OPER(a_3,\ b_1)\ ,\ OPER(a_3,\ b_2)\ ,\ OPER($
$\}$

This matrix is returned row by row, from left to right.

Sequential Cartesian Product , where two clists create a clist. In this operation the OPER is applied sequentially to the result of the operation of A with the first cube of B, the second cube of B, etc, as follows:

**SEQUENTIAL_CARTESIAN_PRODUCT(A, B, OPER, OP_CLIST)
= { CUBE_CLIST_REDUCE((CUBE_CLIST_REDUCE((CUBE_CLIST_REDUCE(A, b_1, OPER, OP_CLIST) b_2, OPER, OP_CLIST)) b_3, OPER, OP_CLIST)) }**

This is applied in sharp, intersection and union of clists.

Convolution . A new clist of length $2n - 1$ is created according to a convolution algorithm such as that used in polynomial multiplication. This operation is a base of many image-processing operations.

Ad. (3) *Complex operations on a single clist* **include the following:** *All permutations, All subsets, All combinations,* **and other. Such operations, which are in fact the mappings of tree-search algorithms to pipes, are based on two coordinated pipes. For instance,** *All Subsets Operation* **uses the method from [?] to generate all subsets of a set. This operation can be used to generate all cubes that include a specified cube.**

Many other pipelined/systolic/data-flow MVCC operations can be created that are especially useful for problem-solving and spectral applications.

Chapter 5

PROBLEMS FOR SELF-EVALUATION RELATED TO UNIVERSAL LOGIC MACHINE.

1. BASE OF THE CCM MACHINES.

   Draw in detail the basic IT cells for:

   a) K = 1,

   b) K = 2,

   c) K =3.

   The blocks should use universal binary logic gates (lookup table types, such as in Xilinx). Write how many inputs and outputs from each module.

   Discuss advantages and disadvantages of each.

2. BASE OF THE CCM MACHINES. Present table of sharp operation realized on CCM of base 3. (similar to the table of complementation operation of base 2 machine from the class).

3. **BASE OF THE CCM MACHINES. The truly 4-valued logic requires 4 * 4 matrices and 2 bits per symbol. The set logic of binary logic requires 2 bits to represent values 0 = 0, 1 = 1, 0,1 = X, = ϵ = empty set = contradiction. Make a comparizon of a general case of truly k-valued logic and set logic for a set with k elements. How are they related to base K.**

4. **BASE OF THE CCM MACHINES. Explain why the operations such as ¿ and ¡ cannot be implemented on binary CCM. Present how these operations can be realized on CCM with larger K. What value of K is enough. Draw the circuit.**

5. **GENERALIZED MULTIPLE-VALUED CUBE CALCULUS.**

   As we said earlier, in Generalized Multiple-Valued Cube Calculus (GMVCC), each cube is a vector of variables. Each variable X has its *type* $T(X)$ **and** *length,* **denoted by** $L(X)$. *$L(X) = R$ * $K(IT)$, where $K(IT)$ is the base of the IT cells used in the implementation of the variable X. The types are:* a symbol, a number, and *a set.*

   Symbols *are used in several discrete combinatorial problems, such as consistent labeling, so the ability to work with symbols is an important innovation of CCM3. For practical reasons the number of different symbols is usually limited. In CCM3 there are two kinds of symbols:* simple symbols *and* complex symbols. *Simple symbols are implemented as binary strings and operations on them are efficiently realized as Boolean functions described by matrices. Such systems will be called Limited Finite Systems (LFS). Complex symbols are ordered sets of simple symbols, For instance a binary number is a complex symbol of simple symbols 0 and 1. A ternary number is a complex symbol of simple symbols 0, 1 and 2.*

   Present how the machine handles ternary numbers for arithmetic operations.

   How addition and multiplication operations can be executed in Galois Field(3).

6. *GENERALIZED MULTIPLE-VALUED CUBE CALCULUS.*

   Restricting the number of logic values allows efficient operations on symbols. Operations on complex symbols are done by iterative operations on ordered sets of simple symbols using multi-bit internal variables. *This is realized as* linear iterative circuits *(Unger) of ITs* *with two kinds of internal variables:* carry signals *going from left to right,* and *confirm signals* going from right to left. *Operations on two simple symbols can be described by two dimensional matrices called* operator tables. *Matrices describing sharp, consensus, and other binary operators can be found in [?]. Examples of matrices for various multiple-valued logics can be found for instance in [?] and [?].*

Use the same method as applied in CCM2.5 to implement all operations from [?].

Use the same method as applied in CCM2.5 to implement all operations from [?].

Use the same method as applied in CCM2.5 to implement all operations from [?] and [?].

7. *GENERALIZED MULTIPLE-VALUED CUBE CALCULUS.* Numbers *in ULM are represented as complex or simple symbols. Operations on numbers are realized as LFS. For instance, when $K = 2$, one is able to realize binary, ternary and quaternary counting. Binary arithmetic operations for arbitrary $v > K$ (such as* addition *and* subtraction modulo v, minus, max and min, *as well as* equality *and* order relations) *are not decomposable to any groups of bits. They are, however, decomposable to groups of K bits assuming one or more* carry signals, *going from left to right. The same is true about* ternary *or* quaternary arithmetic. Multiplication, division, *and other operations can be only described by matrices, and only for numbers* $\leq 2^K$.

   Illustrate how ternary arithmetic operations are realized in CCM with $K = 2$.

   Illustrate how quaternary arithmetic operations are realized in CCM with $K = 2$.

   Illustrate how ternary arithmetic operations are realized in CCM with $K = 3$.

   Illustrate how quaternary arithmetic operations are realized in CCM with $K = 3$.

   Illustrate how ternary arithmetic operations are realized in CCM with $K = 4$.

   Illustrate how quaternary arithmetic operations are realized in CCM with $K = 4$.

   Try to generalize to any value of K being a (a) small, or (b) large natural number.

8. *SET TYPES.*

   Set types *represent ordered sets, in particular, sets of logic values. These types are used in MVCC. As in the positional cube calculus notation or in Set Logic: 1 is for an existing element, 0 for a non-existing one. For values of tuples (relations) the set values are as follows: 10 = false, 01 = true, 11- value irrelevant, 00 = contradiction. Set Operations and Binary Number Operations are the most efficiently realized since they are decomposable to single bits, which means that the same Boolean function can be executed on all pairs of bits of literals.*

   In GMVCC each cube can have two parts: input variables *and* output variables. *We call these the* input-cube *and the* output-cube. *The value of each output variable can be a set (in a set logic) or a number (in a truly mv logic). For instance, in the case of a truly mv logic, each value of the output variable in the output-cube represents the value of that variable for the given input-cube. Thus, [¡01-0110-1011¿ – ¡3,4¿] means the input cube [01-0110-1011] has value 3 in the first output variable and value 4 in the second output variable.*

   Extend all the MVCC operations for this kind of cube calculus.

9. *Additionally to the input and output cubes introduced in GMVCC, any number of adjacent variables can be combined as a* Variable Group.

   Examples: *Two numbers are combined as a* number interval group. *A* tuple *can be created from four symbols; first (simple) symbol stands for tuple value, second (complex) symbol for name of relation, next two (complex) symbols for elements of this relation [415]. For instance the group ¡10,'PARENT, 'Mary, 'Robert¿, denoted as 0 PARENT(Mary, Robert) states that it is not true that Mary is a parent of Robert. Variables in linked groups can be of various types. For instance, assuming set variable $PEOPLE = \mid$ [Mary,Carol,Robert,John,Adam,Julie] $\mid$ $(L(PEOPLE) = 6)$ the set of Mary and John is represented by literal $PEOPLE^{\{Mary,\ John\}}$, or in positional notation $\mid$ [100100] $\mid$.*

   Assuming the relations:

   1 PARENT(Mary, Robert), 1 PARENT(Mary, Carol), 1 PARENT(John, Robert),

   1 PARENT(John, Carol),

   one can create the relation group:

   PARENT(100100, 011000)

which means that Mary and John are Parents of Robert and Carol. This notation compacts four PARENT relations to a tuple with mv variables.

See [415] and [?] for more explanation.

Extend the Cube Calculus for these operations.

Realize all operations of Ulug on the new CCM3 processor.

10. *PREDICATES.*

    The Predicates *check if some relation of operands is satisfied. This relation is set-theoretical in the above examples, but can be arbitrary in general. The most common relations are ¡, ¿,* $=$, $\neq$, $\leq$, $\geq$, $\subseteq$, , $<$, *and* $>$. *They can be either local (in a variable), or global (in a group or a cube).*

    How to implement each of these operations in:

    - CCM2

    - CCM2.5

    - CCM3.0

    Can we implement each of them in CCM2? If no, explain why?

11. *CUBE OPERATIONS.*

    The MVCC Cube Operations *use set-theoretical operations on pairs of variables of operand cubes to calculate the resultant cubes. These operations are set-theoretical in MVCC for CCM1 and CCM2, but in GMVCC they can be arbitrary. The set theoretical operations can be realized by Boolean functions of two variables (simple symbol operations for K=1). The results of operations are complex symbols, i.e. compositions of simple symbols.*

    Discuss with examples the advantage of using CCM2.5 over CCM2, and CCM3.0 over CCM2.5 on all those operations.

12. *NUMERICAL OPERATIONS:*

    Numerical operations on complex symbols include: +, -, max, min, truncated_max, truncated_min. *Numerical operations on simple symbols include:* (high_radix_plus, high_radix_minus, modulo_plus, modulo_minus, high_radix_multiply, high_radix_divide. *They can produce generalized carry signals to create complex symbols as for instance in* radix 3 addition. *Many other symbol operations exist for* $K > 1$.

    Show schematically how relations ¡, ¿, $=$, $\neq$, $\leq$, $\geq$, $\subseteq$, , $\subset$, *and are realized in CCM of base* $K = 2$.

13. *COUNTING OPERATIONS.*

    Counting Operations *count the number of the* occurrences of satisfied relations on variables. *For instance, a numerical value of the* Hamming Distance *of binary cubes A and B is calculated by counting the number of symbols epsilon in cube $C = A \subseteq B$. The distance of mv cubes is calculated by counting the number of symbols es in variables of cube $C = A \subseteq B$. In general, counting serves to evaluate the quality of perfect/imperfect matching of the operand cubes. This allows for the realization of* approximate string matching *and* fuzzy logic *algorithms.*

    Show how to realize in CCM3.0 operation of exact match (number of simple symbols that agree).

    Show how to realize on CCM3.0 operation of approximate match (number of simple symbols that agree more or less).

    There is a table that for every two simple symbols gives numerical match result: 0 - no match, 1 - weak match, 2 - strong match, 3 - perfect match.

    How is this table realized in CCM3.0?

    How the approximate match results are used?

14. *Similarly to MVCC, operations in GMVCC are on cubes, clists and cclists. Operations on cubes are* simple combinational, complex combinational *and* sequential. *However, the relations and operators can be different for each variable, and can be treated differently in different variable groups and in different parts of a cube. It is this flexibility which makes it possible to use GMVCC for the wide range of formal systems listed in the Introduction with our first mention of GMVCC.*

    EXAMPLES OF GMVCC OPERATIONS.

    Give any examples of GMVCC operations that are not in MVCC. Show how they are realized in CCM.

15. *THE MAIN OPERATIONS OF ULM.*

    Below we will present the main principles that are applicable to logic machines of any base, K. *Here they are described for $K = 2$, but the reader can easily modify them for any K.*

    Each sequential operation can be described by a pattern:

    $A\ (\ OP_{sm}\ )\ B\ =\ \{X_1^{aft(A_1,\ B_1)} \ldots X_{i-1}^{aft(A_{i-1},\ B_{i-1})} X_i^{act(A_i,\ B_i)} X_{i+1}^{bef(A_{i+1},\ B_{i+1})} \ldots X_N^{bef(A_N,\ B_N}$

An important property of functions before (for short - *bef*), active (*act*), after *(aft), and* relation *(rel) is that they are K-wise functions, e.g., for K=2, bits* C^s, C^{s+1} *of the resultant cube of each of the functions* $bef^{s,s+1}$, $act^{s,s+1}$, $aft^{s,s+1}$, *and* $rel^{s,s+1}$ *of a simple symbol are dependent only on bits* A^s, A^s+1 *and* B^s, B^s+1 *of the arguments. A complex symbol which represents the value of variable* C_i *of length = R * K(IT) is a composition of R simple symbols which are the results calculated in each of ITs representing this variable.*

Illustrate the above operation for K = 3. The formula and the circuit. What is the difference with respect to K = 2?

16. *The value returned by the* rel_i *of variable* C_i *is determined by the function relation_type. Relation_type, here is* OR, AND, *or one of many other* Boolean functions *of signals corresponding to* **partial relations** $rel^{s,\ s+1}$ *for simple symbols. This function is selected for the desired type of ULM operation. An iterative circuit similar to* multi-rail Maitra cascades *is used to realize the relation_type function. A similar mechanism is used to create more global relations for groups of variables or cubes. In general, rel signals are arbitrary base K signals, so Boolean predicates are not treated in a distinct way.*

    In class we presented such cicuits for relations AND, OR, EXOR, and few other. Show some other relation that can be realized in this circuit and explain its practical usage.

    The first resultant cube for a sequential CC operation is produced for the first **specific** *literal selected as the* **active** *one starting from the left. Later, the next* **specific** *literal to the right is selected as the* **active** *one, and the next resultant cube is produced. This procedure is repeated until the last* **specific** *literal has been selected as the* **active** *one. When producing a particular resultant cube, all the literals with numbers less than the number of the* **active** *literal are of the* **after** *type, all the literals with numbers greater than the number of the* **active** *literal are of the* **before** *type. All these operations are* totally executed in hardware *by the iterative network of state machines.*

    Invent examples of such operations and their applications.

17. *GENERAL PATTERNS FOR GMVCC MAIN ALGORITHM.*

    The more general pattern of the main algorithm for GMVCC operations is:

    $A\ (\ OP_s g\ )\ B\ = \{X_1^{fun_1(state_1,\ A_1,\ B_1)} \dots X_{i-1}^{fun_{i-1}(state_{i-1},\ A_{i-1},\ B_{i-1})} X_i^{fun_i(state_i,\ A_i,\ B_i)} X_{i+1}^{fun_{i+1}(state_{i+1},\ A_{i+1},}$
    $|$ *where all* $state_i$ *are determined by functions* $rel_i(A_i,\ B_i)$ $\}$

In general, functions $fun_i(state_i,\ A_i,\ B_i)$ *are determined as follows:*

$fun_i(state_i,\ A_i,\ B_i) = \quad if\ rel_{i,1} = 1\ then\ fun_{i,1,1}(state_i,\ A_i,\ B_i)$

$else\ if\ rel_{i,1} = 2\ then\ fun_{i,1,2}(state_i,\ A_i,\ B_i)...$

$else\ if\ rel_{i,1} = 3\ then\ fun_{i,1,3}(state_i,\ A_i,\ B_i)...$

$else\ if\ rel_{i,1} = 4\ then\ fun_{i,1,4}(state_i,\ A_i,\ B_i)...$

$else\ if\ rel_{i,2} = 1\ then\ fun_{i,2,1}(state_i,\ A_i,\ B_i)...$

......

$else\ fun_{i,h}(state_i,\ A_i,\ B_i)$

where $state_i$ *is an internal state of a state machine associated with variable* i. *The set of states for the machine includes states* bef, act, *and* aft. *The set* $state_i$ *should have at least* 2^K *states.*

Give some application of this formula that is not a standard MVCC operation as sharp or consensus.

18. *SIMPLE COMBINATIONAL OPERATIONS.*

Combinational *CC operations are defined as K-wise functions on the bits of the arguments. There is no need for defining* **act**, **bef** *and* **aft** *or other states for combinational CC operations. The same computational mechanism can be used to calculate both sequential and combinational CC functions. Simple combinational operations are determined by the formula:*

$A\ (\ OP_sc\ )\ B\ =$

$\{X_1^{fun(A_1,B_1)} ... X_i - 1^{fun(A_{i-1},B_{i-1})} X_i^{fun(A_i,B_i)} X_{i+1}^{fun(A_{i+1},B_{i+1})} ... X_N^{fun(A_N,B_N)}\ \}$

Give example of operation that is not in MVCC.

19. *COMPLEX COMBINATIONAL OPERATIONS.*

In Complex Combinational operations *there is for each pair of* operand cubes *at most a single resultant cube. This cube is produced by a K-wise logic operation* conditioned for each literal by some relation/pattern of the input cubes. *In this category are the operations* prime, double prime, *and* "binary consensus", *as well as conditional value settings such as those described by* C^{XYZ} *operations of Dietmeyer [?] and those used in set logic [?]. Complex combinational operations can be described as:*

$A\ (\ OP_cc\ )\ B\ =$

$\{X_1^{fun_1(A_1,\ B_1)} ... X_{i-1}^{fun_{i-1}(A_{i-1},\ B_{i-1})} X_i^{fun_i(A_i,\ B_i)} X_{i+1}^{fun_i(A_{i+1},\ B_{i+1})} ... X_N^{fun_N(A_N,\ B_N)}\ \mid\ for$

In general, functions $fun_i(A_i,\ B_i)$ *are determined as follows:*

```
$fun_i( A_i, B_i) =$      if $rel_{i,1} = 1$ then $fun_{i,1,1}(A_i, B_i)$
    else if $rel_{i,1} = 2$ then $fun_{i,1,2}(A_i, B_i)$...
    else if $rel_{i,1} = 3$ then $fun_{i,1,3}(A_i, B_i)$...
    else if $rel_{i,1} = 4$ then $fun_{i,1,4}(A_i, B_i)$...
    else if $rel_{i,2} = 1$ then $fun_{i,2,1}(A_i, B_i)$...
      ......
     else $fun_{i,h}(A_i, B_i)$
```

Give example of an operation of this kind that is not in MVCC.

20. *SHIFT AND COUNTING OPERATIONS.*

It follows from the above considerations, that each of the functions fun_i $fun_{i,j,t}$ and $rel_{i,j}$ is actually composed of functions for simple symbols, where each function for simple symbols is a K-bit-output Boolean function of two K-tuples of bits. Function $fun_i(state_i, A_i, B_i)$ is composed of functions that additionally have bits of the encoded signals of states for machines in ITs corresponding to simple symbols. The number of bits used to program the function relation_type depends on K and the particular realization of the machine. A K-bit signal called **carry** *is used to help combine partial relations or functions for particular pairs of bits. For relations like $A = B$, $A \subseteq B$ a single bit carry signal is enough, while for relations like $A \leq B$, $A < B$ or $A \in B$ a two-bit carry signal is required. All these programmable functions define the* programmability pattern of the machine.

The simple combinational operations of ULM are vector-like extensions of standard computer instructions (logical, numerical, jumps, etc) expanded in a SIMD pattern to separate fields (variables), each of arbitrary length. The complex combinational operations are similar extensions, but they allow us to execute different operations in the fields. They are useful for tagged operations, flag conditional setting, etc. Sequential operations generalize shifts and string matching operations. For instance, various types of shifts can be executed in parallel for all variables, as well as shifts between variables. In addition to standard arithmetical/logical shifts, shifts with unary K-wise operation in the shift loop are possible, that are the generalizations of the K=1 case of the Johnson counter with the controlled EXOR in the shift loop.

In the previous chapters we demonstrated several of those shift operations. Present in a very systematic way all possible shifting/counting operations for $K = 2$ and for $K = 3$.

21. *THE HARDWARE ARCHITECTURE OF ULM PROCESSOR. BASIC COMPONENTS OF ULM PROCESSOR.*

    All the known software subroutines process the literals sequentially, but for most of the literals the resultant cubes generated will have contradictions and that will have to be removed later. ULM implements a completely new architecture to take advantage of the peculiarities of sequential cube calculus operations. The architecture is an iterative logic array (ILU) with "carry" signals running from left to right **and** *from right to left through the iterative circuit of* **Position State Machines (PSMs). The fundamental advantage of this approach is that only cubes without contradictions are generated.**

    The ULM Processor consists of a set of bit-slice ILU processing units, an interface controller, and a control unit. The processing unit is implemented as an iterative logic array (ILU) of basic **buiding blocks.** *A single cell (block) from an ILU is called a* ITerative Cell (IT). *Basically, ULM has a base of two (K(IT) = 2), which allows us to realize matrices of all logic operators in logics with not more than* $2^2 = 4$ *values. The four simple symbols for base 2 are:*

    0 (negated variable),

    1 (positive variable),

    X (don't care),

    and epsilon (contradiction).

    They are encoded as 10, 01, 11, and 00, respectively. Since it is important to realize resolution/unification operations of theorem-proving [415] and other truly mv logics, we can also realize base 3 operations in ULM. The CUBEX concept of theorem-proving [415] has three variants: propositional logic, predicate logic *and* hybrid. *This concept uses new "cross" cube calculus operations (cross-intersection, cross-sharp, cross-consensus, cross-subsume), which require four additional symbols. The total set of symbols for base 3 ULM is then* $\{0, 1, X, \epsilon, R, L, Y, Z\}$*, which leads to K=3. They are coded: 010, 001, 011, 000, 111, 100, 101, 110, respectively (which is different from that in [415]).*

    By a W-input K-base universal cell *we mean a logic function with W inputs and one output, each input or output being a base K signal. It means that when multiple-valued logic is realized using binary signals, one has K wires to represent each simple symbol of a set of* 2^K *symbols. A universal cell of base K realizes all matrices of* base 2^K universal logic, *or base K set logic. In ULM each simple symbol is* *processed by an iterative cell IT. A K-base symbol requires a K-wise IT cell.*

Essentially, ULM is a logic machine of K=2 but it can be hardware-reconfigured to machine with K=3 and smaller number of ITs. As a "by-product" of this scheme one is also able to realize some functions for K=4.

Illustrate how this scheme of hardware reconfiguration from $K = 2$ to $K = 3$ can be realized.

22. *For simplification, only case of $K(IT) = 2$ will be discussed below but the principles are exactly the same regardless of the base. Moreover, they would remain the same even for a realization of universal cells in a truly mv logic, and not using universal cells with binary gates as in ULM.*

IT blocks will be enumerated from left to right: IT[1], IT[2], ..IT[i], ..., IT[n]. (The number of ITs is denoted by n, so that the number of bits is 2n and we can process n binary variables). Besides combinational logic each IT[i] includes a **Position State Machine (PSM)** *that influences the local interpretation of the micro-instructions. In this sense each IT is a small processing unit that processes a part of a cube in parallel and communicates with other processors that are connected in a linear organization. A processor with two ITs of K=2 is shown in Fig. 9.20. For explanation purposes we will divide each IT[i] into four blocks according to the function that it performs, and we will discuss simplified signals for MVCC operations from section 5 and K=2. The combinational blocks of IT[i] are:* **RELATION[i]**, **MATCH_COUNTER[i]**, *and* **OPERATION[i]**. *The sequential PSM block is called* **STATE[i]**. *A, B are the input (operand) cubes and C, D are the output (resultant) cubes.*

(1) *Block* **RELATION[i] has the task of identifying the position of the IT within the literal and generating a Boolean signal VARIABLE[i] that is true when the IT[i] is a part of a literal that satisfies the selected relation** *"rel"*. **To calculate the value of VARIABLE[i], the RELATION[i] block uses two iterative signals. CARRY[i] is an iterative signal that runs from left to right and is true when all ITs of the same literal to the left of the IT satisfy the AND-type relation encoded in rel. CONF[i] (confirm) says that at this position all ITs have satisfied the relation. As we see, the signal CARRY goes from left to right, up to the end of variable, and next returns as signal CONF, back to all ITs of this variable. This**

explanation is only for AND-type relations. Similar explanation for OR-type and other relations can be given.

(2) Block STATE[i] is an FSM essential to executing sequential operations. The state of the STATE[i] block represents the position of the IT[i] in relation to the active literal. The STATE[i] is in state active if the IT[i] is a part of an active literal; it is in the before state if the IT[i] is to the right of active literal; and in after state if the IT[i] is to the left of the active literal. All STATE[i] are initialized to the state *before* **with the global signal CLEAR from CU.**

(3) MATCH_COUNTER[i] counts the number of satisfied predicates in IT[i].

(4) OPERATION[i] *creates bits of resultant cubes by performing the operation on bits of the operand cubes. This is where the operator matrix is programmed in.*

The **Control Unit (CU)** *of the CCM receives the code of a high level cube calculus operation* **(CCM instruction)** *in the* **Instruction Register (IR)** *and translates it into simpler basic operations implemented in the processing unit.*

The **Bus Interface Unit (BIU)** *handles the communication between the host computer (or a distributed system of CCMs and controllers) and the CCM. The communication between the BIU and the ILU is done through shared registers/memory, as is the communication between the BIU and the CU. The interfaces between the BIU and the ILU and the CU were made independent, asynchronous and through a protocol to let the design of the ILU and CU be independent of the BIU.*

A shared data register file is used for storing the input cubes, the output cubes, and the intermediate results. This shared file prevents also a loss of performance due to differences in the processing rates of the system and the ILU.

Some other registers used for communication and control are $\mathbf{RIGHT}_E(rightedgeofvariable),$
$valuedvariablewhichwillusethefirst2ITs, thesecond2bitsareusedbyanother4-$
$valuedvariablewhichwilltakeanothertwoITs, thethirdandthefourthvariablesare2-$
$valued(binary)andwilluseonlyoneiterativecelleach.$

Register $RIGHT_E isusedtoseparatevariables.Nowwewantalsotoseparateinputandoutputparts$

Draw the schematic of complete IT for CCM3.0.

23. THE BASIC CYCLE.

The basic cycle is a generalization of one from chapter ??. The ILU recognizes the next specific (active) literal and generates a resultant cube in each cycle. It realizes, using internal distributed control, the lowest level iterative loop, as described by formulas ??. Therefore, ILU does not need the control unit to execute the basic cube operations. While generating the resultant cube the role of CU is limited to generating REQUEST signals and checking NEXT signals. The ILU is controlled by two types of signals, *iterative signals* and *global signals.* Two of those signals, global REQUEST and iterative NEXT work in an "interlock mechanism" that replaces the "clock" of synchronous machines with a "two-phase non-overlapping waveform" in a linear sequence of cellular automata.

A general concept in computer architecture is to remove the lowest level control loop from the control unit and put it directly in the data path in a distributed form. The state machine in the CU in CCM3 becomes relatively simple and a great part of the control task is distributed in the ILU itself.

Sequential operation begins by loading the operand cubes to let RELATION[i] blocks recognize all the specific literals. The CU keeps the global signals REQUEST, CLEAR, as well as the initial signal NEXT[0] false and lets all VARIABLE[i] signals in ITs reach their final values. The interval of time before CU is ready to do something more has to be long enough for IT[n] to have the VARIABLE[n] signal stabilized (delay of 1 IT for a binary variable, 2 IT for a 4-valued variable,...). At the end of this phase all specific literals are marked by 1 values of VARIABLE[i] in the corresponding ITs. The CU resets the ILU to its initial condition by asserting CLEAR. The control unit does not have the control over signals VARIABLE[i] but since A, B, AND_OR and M remain stable, VARIABLE[i] will remain stable as well. The execution of the instruction really begins with the assertion of NEXT for the leftmost IT, NEXT[0]. The first literal that has VARIABLE[i] $=$ true will become active. (If many ITs are used to represent a literal, all of them will have VARIABLE[i] $= 1$, and all will become active. Remember, however, that there is a signal *RIGHT_E[i]* that is used to distinguish between neighboring variables). Either the first literal becomes active or signal NEXT may propagate through one or more literals.

After short time, with stable states of STATE, CU transmits resultant cube(s), C (and D), to the output(s). CU then samples NEXT[n]. If it is true there are no resultant cubes to generate and the operation is finished, if it is false, it means that the signal NEXT has been stopped by a next specific literal and the CU has to output the resultant cube. If NEXT[n] = 0 the CU generates the active edge of the REQUEST signal to prepare for the next cycle. This will make the ITs in the active positions transit to the after state and let the NEXT signal pass to the cells after it. A new cycle can begin for those literals in state before as soon as the cells change from the active state to the after state. Signal *READY* informs CU that the resultant cubes C and D are ready. Such process is iterated until the NEXT = true signal passes through the entire ILU and, as NEXT[n], reaches the CU. At this point the operation is finished.

Step-by-step, analyse how the described above mechanism will work in CCM3.0. What is the essence of generalization?

24. COMMUNICATION-BASED OPERATIONS.

Just as the DSP and AI architectures can be build from regular structures of few kinds of blocks through which data vectors are transmitted, the principle of CCM is to allow for pipelined/systolic execution of the most common operations on clists and cclists. Those operations include for instance Cartesian products, dot products, permutations, and generations of all subsets or combinations of a clist. Micro Programming of CCM consists then in assigning values to two kinds of control fields:

P 1. those that describe the processing inside the CCM, for example, operations on cubes and short clists as described in section 5;

P 2. those that describe the *communication* of CCM with other processors and memories.

There are currently two styles of communication:

(a) Address Generator based, and

(b) Data Flow based.

Address Generators are used to send out the address of the desired operand in Dual-Port Memories. The data flow style is based on FIFOs. In this case the CCM chip sends a signal to a FIFO to get the next cube. In most clist operations CCM uses the *Ping-Pong communication mechanism,* known from DSP, particularly FFT realizations. The principle of this mechanism is to transfer the data iteratively through the CCM processor: from Memory A to Memory B, next from Memory B to Memory A, next from Memory A to Memory B, etc., until certain conditions are met. With each change of the transfer direction, the operations executed in the processor and the data-flow in it can be modified. There are also three main types of communication-based operations: (1) *simple operations on a clist,* (2) *simple operations on two clists,* and (3) *complex operations on a clist.* NOTE: In this document, denotes a clist, a cclist, [ ] a cube, ¡ ¿ a group of variables, —[ ]— a set, ' a symbol, ¬ is a negation, *OP_type*, OPER and OP_CLIST are two-operand cube operations.

Explain on an example how the Ping-Pong architecture works. Show address generator sequences.

25. SIMPLE OPERATIONS ON A CLIST.

(1) *Simple operations on a clist* include *Global Reduction, Mapping, Pairing, Selection, Inversion, Mapping with Insertion, Inversion with Insertion, Replacement, All Pairs with Reduction, All Ordered Pairs,* and other operations. Examples of some of those operations on clist A = a_1, a_2, a_3 are as follows.

P 1. Global Reduction, in which a cube is created from a clist:

REDUCTION (A, OPER) = $[a_1 \; OPER \; a_2 \; OPER \; a_3]$

Examples of such operations in cube calculus are the supercube of all cubes from the clist, and the intersection of all cubes from the clist.

Show how to use these operations for executing:

(a) A $\#_d$ B , where A and B are arbitrary arrays.

(b) A $\#_{nd}$ B , where A and B are arbitrary arrays.

(c) A $\cdot$ B , where A and B are arbitrary arrays.

(d) A supercube B , where A and B are arbitrary arrays.

26. **MAPPING. Mapping is an operation for which the clist produces a new clist of the same size:**

    MAPPING(A, OPER) = $OPER(a_1),\ OPER(a_2),\ OPER(a_3)$

    Give examples of Mapping applications.

27. **PAIRING. Pairing is an operation where the operation is repeated on subsequent pairs of arguments and the results are directly inserted in a new clist:**

    PAIRING(A, OPER, OP_CLIST) = $OPER(a_1,\ a_2)\ OP\_CLIST\ OPER(a_3,\ \phi)$

    Show application of Pairing.

    Sorting **and** *sorting with absorption* **can be realized as repeated pairing. Show how.**

28. **Reducing Operations such as:**

    "ab + a $\longrightarrow$ a", "a $\oplus$ $\longrightarrow$ 0" "ab $\oplus$ a $\longrightarrow a\,\bar{b}$ or " a $\oplus$ 0 $\longrightarrow a$" can be used as OP_CLIST. Repetitive Ping-Pong transfer of a clist through a CCM with an exchange of neighbor cubes (based on a specified ordering relation between them) leads to sorting the clist. When the compare/exchange operation also executes cube absorption, algorithms such as POSE -¿ SOPE transformation by multiplication can be executed [?].

    Show architectures for such applications based on the SM, CM, CCM2, CCM2.5 and CCM3.0.

29. **INVERSION. Inversion is an operation for which the clist creates a new clist of the same size:**

    INVERSION(A, OPER) = OPER(a_3), OPER(a_2), OPER(a_1)

    Give examples of Inversion.

30. **ALL PAIRS OF A CLIST.**

    All Pairs of a Clist:

    ALL_PAIRS(A, OPER) = $OPER(a_1,\ a_2),\ OPER(a_1,\ a_3),\ OPER(\ a_2,\ a_3)$
    .

    Give two applications from different areas of a practical application of this operation. Write address generator sequences for them.

31. SELECTION.

Selection is an operation known from CAM-based and associative processing. It selects cube(s) for which a certain Boolean function of literal-related predicates is satisfied. For instance the operation:

$age > 34$

AND

$character = `good$

OR

$properties \subseteq ||wealth, \; good - contacts||$

operates on cubes with numeric variable *"age"*, symbol variable *"character"*, and set variable *"properties"* and selects all people who either are older than 34 and have good character, or have wealth and good contacts among their properties.

Show practical architecture for data-base search based on these principles.

32. REPLACEMENT. The similar operation of Replacement does some cube operation on selected cubes such as adding $1000 to the numeric variable " salary" or adding set value " good-contacts" to the variable " properties"$.

Show application and realization.

33. The following operations of Cube Pairing and Cube Pairing with Reduction are executed on a clist and a cube:

CUBE_CLIST_PAIR(A, cube, OPER) =

$OPER(a_1, \; cube), \; OPER(a_2, \; cube), \; OPER(a_3, \; cube)$

CUBE_CLIST_REDUCE(A, cube, OPER, OP_CLIST) =

$OPER(a_1, \; cube) \; OP\_CLIST \; OPER(a_2, \; cube) \; OP\_CLIST \; OPER(a_3, \; cube)$
E

Show applications in CCM3.0.

34. SIMPLE OPERATIONS ON TWO LISTS.

Simple operations on two lists include *Dot Product, Vector Product, Simple Cartesian Product, Sequential Cartesian Product, Convolution, Vector Product with Insertion, Vector Product with Reduction,* and other. Some of those operations on clists A = $a_1, \; a_2, \; a_3$ and B = $b_1, \; b_2, \; b_3$ are the following:

P 1. Dot Product, for which two clists create a cube:

DOT_PRODUCT(A, B, OPER, OP_CLIST) =

$[OPER(a_1,\ b_1)\ OP\_CLIST\ OPER(a_2,\ b_2)\ OP\_CLIST\ OPER(a_3,\ b_3)]$

As an example, OPER can be a cube intersection operation, and OP_CLIST can be a cube supercube operation.

Show applications.

35. VECTOR PRODUCT.

P 2. Vector Product, for which two clists create a clist:

VECTOR_PRODUCT(A, B, OPER) =

$OPER(a_1,\ b_1)\ ,\ OPER(a_2,\ b_2)\ ,\ OPER(a_3,\ b_3)$

Discuss applications.

SIMPLE CARTESIAN PRODUCT.

P 3. Simple Cartesian Product, where two clists create a clist (in particular representing a matrix, or a set of pairs):

SIMPLE_CARTESIAN_PRODUCT(A, B, OPER) =

$OPER(a_1,\ b_1)\ ,\ OPER(a_1,\ b_2)\ ,\ OPER(a_1,\ b_3)\ ,$

$OPER(a_2,\ b_1)\ ,\ OPER(a_2,\ b_2)\ ,\ OPER(a_2,\ b_3)\ ,$

$OPER(a_3,\ b_1)\ ,\ OPER(a_3,\ b_2)\ ,\ OPER(a_3,\ b_3)$

This matrix is returned row by row, from left to right.

List applications. Discuss Galois Field applications in particular.

36. SEQUENTIAL CARTESIAN PRODUCT

P 4. Sequential Cartesian Product, is an operation where two clists create a clist. In this operation the OPER is applied sequentially to the result of the operation of A with the first cube of B, the second cube of B, etc, as follows:

SEQUENTIAL_CARTESIAN_PRODUCT(A, B, OPER, OP_CLIST) =

CUBE_CLIST_REDUCE((CUBE_CLIST_REDUCE((CUBE_CLIST_REDUCE(A, b_1, OPER, OP_CLIST) b_2, OPER, OP_CLIST)) b_3, OPER, OP_CLIST))

This is applied in sharp, intersection and union of clists.

Give examples of applications to CCM2.0. Give a practical algorithm to generate all primes that would use this method.

37. CONVOLUTION.

    P 5. Convolution. A new clist of length $2n-1$ is created according to a convolution algorithm such as that used in polynomial multiplication. This operation is a base of many image-processing operations.

    Show any image processing operation using this concept.

38. COMPLEX OPERATIONS ON SINGLE LISTS.

    Complex operations on a single clist include the following: *All permutations, All subsets, All combinations,* and other. Such operations, which are in fact the mappings of tree-search algorithms to pipes, are based on two coordinated pipes. For instance, *All Subsets Operation* uses the method from [?, ?] to generate all subsets of a set. This operation can be used to generate all cubes that include a specified cube.

    Many other pipelined/systolic/data-flow GMVCC operations can be created that are especially useful for problem-solving and spectral applications.

    Illustrate each of the above operations with one application of a practical sense. Think first about logic operations and image processing operations presented in the class.

39. Can you invent other communication operations than all those listed above?

40. Can you explain in a systematic way how all such operations can be created for vectors of GMVCC cubes?

41. Think about programming languages as APL, Prolog or LISP. Show solutions.

42. Think about mathematical and logical (set theory) notations in learned at school and at university. What communication operations are associated with them? Show solutions.

ACKNOWLEDGMENTS

Marek Perkowski would like to thank many people who over many years helped in one way or another to develop the ideas, machines and software described in part in this document: by designing chips, helping to find sources of information, contributing criticizm and ideas, helping to write code and documentation. Mr. Haitham Akkary from Intel wrote VHDL code for CCM. Dr. Joseph Brandenburg of Intel Supercomputers helped me to learn about parallel processing while I was a summer professor at Intel in 1986. Prof. Robert Brayton of U.C.Berkeley helped to understand and appreciate many important issues in logic synthesis especially related to SOP minimization. Tuan Caominh of Intel wrote the first assembler of CCM. David Chalupsky of Intel designed the first version of BIU. Prof. Maciej Ciesielski of Univ. of Mass, Amherst, collaborated on ideas of graph coloring and implicant generation. Coen Englebarts from Holland developed the first Xilinx prototype and put many of my reports and students reports together. Cecilia Espinosa of University of Santo Tomas, Philippines, developed the Hough Transform on CCM. Prof. Bogdan Falkowski, Nanyang Technical University, helped to understand better relation to spectral methods. David Foote from Intel developed second Xilinx prototype and have done software comparisons. Jeff Fox, of GTE Labs made me interested in EXOR logic, which become my source of inspiration for many ideas. Prof. Andrzej Goralski from Polish Academy of Sciences made me interested in history of computers and Ramon Lullus. Prof. Doug Hall of PSU improved presentation and helped to clarify and simulate the design of CCM-2. Martin Helliwell, US Navy, invented the binary primary and secondary crosslink operations. Dr. Wlodzimierz Holsztynski enthused me to massively parallel VLSI and explained some ideas of his GAPP chip. James Howard invented the EXOR-based counter circuit. Phuong Minh Ho of Microsoft developed and simulated the first version of Satisfiability Machine. Dr. Y.I. Hsieh from Sharp Microelectronics helped to understand difference between university research and industrial product development. He helped me also to learn about massively parallel DSP design and Ping-Pong architectures. Luis Kida from BIT invented the ring of asynchronous machines of CCM-1. He helped also to write [?, 170] and several additional reports, and with Cecilia Espinoza designed the VLSI chip of CCM-1. Prof. Malgorzata Marek-Sadowska of U.C. Santa Bar-

bara provided a very useful criticism of my ideas over the years. Dieudonne Mayi from Microsoft helped with CCM-1 chip and many simulations. Prof. Ryszard Michalski of (then) MIT told me about importance of set covering problem in multivalued logic. Prof. Piotr Misiurewicz of WTU developed logic lab and equipment that helped me to develop the prototypes of my first logic machines. Chintamani M.Patwardhan simulated several versions. Jolanta Pawelska-Stepien worked with me on a never published book about cube calculus. Edmund Pierzchala of PSU proved with his design that parallel CCM is not practical. Keith Pirkl wrote parallel software for several combinational operations of CCM. Prof. David C. Rine from Washington University helped with literature on multi-valued logic which improved my understanding of many applications. Dr. Richard Rudell of Synopsys was a creatively critical reviewer and helped to develop better formulation of some ideas. Dr. Andisheh Sarabi from Viewlogic Corp. invented few cube operations and developed software. Prof. Tsutomu Sasao from Kyushu University, Japan, asked me be an invited speaker at ISMVL in Sendai, which caused me to prepare better presentations of machine and in result improved it much. Dr. Ingo Schaefer of Mentor Graphics improved ring mechanism and adapted to synchronous logic. Ning Song from Lattice Logic Corporation invented multivalued exorlink operation and removed bugs from equations. Prof. Wieslaw Traczyk of WTU first introduced me to logic design and was my Ph.D. supervisor. Dr. Pan Wu of AT&T Bell Labs developed few new CC operations. Lixin Zhou from Mentor Graphics helped with VHDL simulation and developed testability analysis.

Chapter 6

LITERATURE

AAAAAAAAAAA

Bibliography

[1] *Proc. ACM FPGA'92 Workshop,* Berkeley, 16-18 Febr. 1992.

[2]

[3] Algotronix, The CHS 2*4, the World's First Custom Computer, 1992.

bibitem1 bibitemb2 T. Aoki, et al, *Proc. IEEE ISMVL,* pp. 364-367, May 1989.

b3 bibitema3 Arnold, et al, "SPLASH 2", *Report,*

b4 bibitema4 A. Ast, R. Hartenstein, et al, "Novel High Performance Machine Paradigms and Fast-Turnaround ASIC Design Methods: A Consequence of, and, a Challenge to, Field-programmable Logic", in *FPL '92.*

[4] Aristotle, "Politics", translated by Ernest Baker, *Oxford University Press,* Oxford, England.

[5] A.A. Arkhangelskyaya, V.G. Lazarev, and V.N. Roginskii, "The Mechanization of the Process of Synthesizing Relay Networks", *Vses. Mos. Sov. Teorii Ust. Rel. Deist.,* p. 73, 1957.

[6] A.A. Arkhangelskyaya, V.G. Lazarev, and V.N. Roginskii, "A Machine for the Synthesis of Contact Networks", *Prob. Pered. Infor.,* Vol. 1., pp. 41-52, 1959.

[7] A.A. Arkhangelskyaya, V.G. Lazarev, et al., "A Machine for the Synthesis of Relay-Contact Networks", *Prob. Pered. Infor.,* No. 6., pp. 5-23, 1960.

[8] M. Amamiya, M. Takesue, R. Hasegawa, and H. Mikami, "Implementation and Evaluation of a List-Processing-Oriented Data Flow Machine", *Proc. 13th Intern. Symp. on Computer Architecture,* pp. 10-19, 1986.

BBBBBBBBBBB

[9] Ch. Babbage, *"Passages from the Life of a Philosopher",* London: Longman, Green, 1864. Reprinted in 1968 by Dawsons of Pall Mall (London). Gdzie ja to czytalem?

b5

[10] P. Barrie, P. Cockshott, at al, "SPACE: A Scalable Cellular Array Architecture", *in FPL '92.*

[11] K.E. Batcher, "Design of a Massively Parallel Processor", *IEEE Trans. on Computers*, C-29,, 9,

[12] E.T. Bell, *"Men of Mathematics"*, New York: Simon & Schuster, 1965.

[13] V.E. Benes, "Mathematical Theory of Connecting Networks and Telephone Traffic", *Academic Press*, 1965.

b6

[14] P. Bertin, D. Roncin, and J. Vuillemin, "Introduction to Programmable Active Memories", *Research Report No. 3,* DEC Paris Research Laboratory, 1989.

[15] Bic, L., "Execution of Logic Programs on a Dataflow Architecture", *SIGARCH Newsletter,* Vol. 12., No. 3., pp. 290-296, June 1984.

[16] D. Bobrow, "A Symbolic Logic Machine to Minimize Boolean Functions of Four Variables, and Application to Switching Circuits", 12 pp., privately printed manuscript, Bronx 1952.

[17] M. Boden, *"Artificial Intelligence and Natural Man",* New York, Basic Books, 1977.

[18] B. Boghosian and D. Levermore, “A cellular automaton for Burgers' equation,” *Complex Systems*, 1, 1987, 17.

[19] Bolt, Beranek an Newman, Inc., "Development of a Butterfly Test Bed", Rep. 5872, *Quaterly Technical Report,* Number 1, 1985.

[20] Boole, (George?) *"The Mathematical Analysis of Logic"*, Cambridge, 1847. reprinted Oxford, 1948; in Rush Rhees (ed.), Studies in Logic and Probability, Watts and Co., London, 1952.

[21] G. Boole, *"The Laws of Thought"*, London 1854, in Rhees, Studies in Logic.

[22] A. Borning, "Thinglab: A Constraint-Oriented Simulation Laboratory", *Stanford University Computer Science Department*, Rep. 79-746, 1979.

[23] D. Brand, and T. Sasao, "On the Minimization of And-Exor Expressions", *23 FTC*, pp. 1 - 9, 1990.

b7 Bray 84

[24] R.K. Brayton, G.D. Hachtel, C.T. McMullen, and A.L. Sangiovanni-Vincentelli, "Logic Minimization Algorithms for VLSI Synthesis", *Kluwer Academic Publishers,* 1984.

[25] Y. Breitbart, and R. Vairavan, "The Computational Complexity of a Class of Minimization Algorithms for Switching Functions". *IEEE TC*, Vol.C-20, No. 12, December 1979, pp.941-943.

[26] M.A. Breuer, (ed.), "Design Automation of Digital Systems". Vol.1, *Prentice Hall*, Englewood Cliffs, New Jersey, 1972.

[27] M.A. Breuer, and K. Shamsa, "A hardware router", *J.Digital Systems*, vol IV, issue 4, pp. 393-408, 1981.

[28] G. Broomell, and J.R. Heath, "Classification Categories and Historical Development of Circuit Switching Topologies:, *Computing Surveys*, 15, (2), pp. 95-133.

[29] M.F. Brown, "Reduced Solutions of Boolean Equations", *IEEE Trans. on Computers,* Vol. C-19, pp. 976-981, 1970.

[30] M.F. Brown, "Single-parameter Solutions of Flip-Flop Equations", *IEEE Trans. on Computers,* Vol. C-20, pp. 452-454, April, 1971.

[31] M.F. Brown, "On a Convenient Division of Labor in the Generation of Prime Implicants", *Computers and Electrical Engineering,* Vol. 6., pp. 267-271, 1979.

[32] M.F. Brown, and S. Rudeanu, "Consequences, Consistency and Independence in Boolean Algebra", *Notre Dame J. Formal Logic,* Vol. 22, No. 1., pp. 45-62, 1981.

[33] M.F. Brown, "Segmental Solutions of Boolean Equations", *Discrete Applied Mathematics,* Vol. 4, pp. 87-96, 1982.

[34] M.F. Brown, and S. Rudeanu, "Recurrent Covers and Boolean Equations", *Proc. Colloq. on Lattice Theory,* Szeged, Hungary, Aug. 1980. Published in *Colloquia Mathematica Societatis Janos Bolyai,* North-Holland Pub. Co., Vol. 33, pp. 55-86, 1983.

[35] M.F. Brown, and S. Rudeanu, "Prime Implicants of Dependency Functions", *Analele Universitatii Bucuresti,* Vol. 37, No. 2., pp. 6-11, 1988.

b8

[36] F.M. Brown, "Boolean Reasoning. The Logic of Boolean Equations", *Kluwer Academic Publishers,* Boston/Dordrecht/London, 1990.

[37] S.D. Brown, R.J. Francis, J. Rose, Z.G. Vranesic, "Field Programmable Gate Arrays", *Kluwer Academic Publishers*

[38] S.A. Browning, "A Tree Machine", *Lambda Magazine* 1, 2, pp. 31-36.

[39] J.A. Brzozowski, and M. Yoeli, "Digital Networks", *Englewood Cliffs,* N.J., Prentice-Hall, 1976.

[40] B. Burack, "An Electrical Logic Machine", *Science,* Vol. 109, June 17, 1949, p. 610.

[41] A.W. Burks, D.W. Warren, and J.B. Wright, "An Analysis of a Logical Machine Using Parenthesis-free Notation", *Mathematical Tables and other Aids to Computation,* Vol. 8., April, 1954, p. 53.

CCCCCCCCCCC

[42] C.R. Carroll, "Hardware Path Finders", *Caltech VLSI Conference Proceedings*, California Institute of Technology, 1980.

[43] Cellular Automata and the Modeling of Complex Physical Systems, *Proc. of 1989 Les Houches Worshop*, Springer, Berlin, 1989.

[44] S. Chakradhar, V. Agrawal, and M. Bushnell, "Neural Net and Boolean Satisfiability Models of Logic Circuits", *IEEE Design and Test of Computers,* October 1990.

[45] U.S. Chakravarthy, S. Kasif, M. Kohli, J. Minker, and D. Cao, "Logic Programming on ZMOB: A Highly Parallel Machine" *Departament of Computer Science*, University of Maryland, College Park, MD.,

[46] A.H. Chan, "Using decision trees to derive the complement of a binary function with multiple-valued inputs," *IEEE Trans. on Comp.,* Vol. C-36, pp. 212 - 214, Febr. 1987.

b9

[47] P.K. Chan, M. Schlag, and M. Martin, "BORG: A Reconfigurable Prototyping Board using Field-Programmable Gate Arrays", in *FPGA '92.*

[48] H. Chang, "Bubbles for Relational Database", *Fourth Annual Workshop on Computer Architecture for Non-Numeric Processing*, pp. 110-115.

[49] D. Chyan, and M.A. Breuer, "A placement algorithm for array processors," *Proc.20th Design Automation Conference,* pp.182- 188, June 1983.

[50] M. Ciesielski, FIND- Minimization Based on Graph Coloring", *Proc. of the International Conference on Computer Design:*, VLSI in Computers, ICCD'89, October 1989, pp. 262-265.

[51] E.F. Codd, "Relational Completeness of Data Base Sublanguages", in "Database Systems", R. Rustin (ed), *Prentice Hall.*

[52] M. Cohn, "Switching Function Canonical Forms over Integer Fields", *Ph.D. Dissertation,* Harvard University, Cambridge, MA, Dec. 1960.

[53] M. Cohn, "Inconsistent canonical forms of switching functions", *IRE Trans. Electron. Comput.,*

b10

[54] Concurrent Logic, Application Notes and other company materials, 1992.

[55] G.P. Copeland, G.J. Lipovski, and S.Y.W. Su, "The Architecture of CASSM: A Cellular System for Non-numeric Processing", *Proceedings of the 1st Annual Symposium on Computer Architectures*, pp. 121-128.

b11

[56] C.E. Cox, and W.E. Blanz, "GANGLION - A fast field programmable gate array gate array implementation of a connectionist classifier," *Report No. RJ 8290 (75651), IBM Research Division,* Almaden Research Centre, 1990.

[57] L. Csanky, M. Perkowski, and I. Schaefer, "Exact Minimization of Mixed-Radix Exclusive Sums of Products and Generalized Reed-Muller Forms", *IEE Transactions,* FIND. 1992?.

[58] L. Csanky, M. Perkowski, and I. Schaefer, "Canonical restricted mixed-polarity exclusive sum of products and the efficient algorithm for their minimization", *ISCAS'92.*

b12

[59] R. Cucchhiara, et al, and T.S. Cinotti, "FPGAs boost flexibility and performance of a fine grain mesh connected SIMD array for computer vision applications", *in FPL '92.*

DDDDDDDDDDDD

[60] C.J. Date, "An Introduction to Database Systems", *Addison-Wesley*, 1975.

[61] Daumas, (NAME?) *"History of Technology",* Vols. I + II, (EDITOR??)

[62] M. Davio, and J.P. Deschamps, "Classes of Solutions of Boolean Equations", *Philips Research Report,* Vol. 24., pp. 373-378, October 1969.

[63] M. Davio, J.P. Deschamps, and A. Thayse, *"Discrete and Switching Functions",* McGraw-Hill Book Co., Inc., New York, 1978.

[64] M. Davis, and H. Putnam, "A Computing Procedure for Qualification Theory", *J. Assoc. for Computing Machinery,* Vol. 7., pp. 201-215, 1960.

[65] R. Davis, and D. Thomas, "Geometric Arithmetic Parallel Processor", *NCR*, 1984, preprint.

[66] C. Delobel, and R.G. Casey, "Decomposition of a Data Base and the Theory of Boolean Switching Functions", *IBM J. Res. & Develop.,* Vol. 17., pp. 374-386, 1973.

[67] G. De Michelli, A.L. Sangiovanni-Vincentelli, and T. Villa, "Computer-Aided Synthesis of PLA-Based Finite State Machines", *Proc. IEEE 1983 ICCAD*, pp. 154-156. Sept. 1983.

[68] G. DeMicheli, R. Brayton, and A. Sangiovanni-Vincentelli: "Optimal State Assignment for Finite State Machines", *IEEE Trans. on CAD,* Vol. CAD-4, No. 3, July 1985, pp. 268-284.

[69] M.M. Denneau, "The Yorktown simulation engine," *In Proc. 19th Design Automation Conf.*, pp.55-59, June 1982.

[70] J.B. Dennis, and K.S. Weng, "Applications of Data Flow Computation to the Weather Problem", in "High Speed Computer and Algorithm Organization", D.J. Kuck, D.H. Lawrie, and A. Sameh (eds.), *Academic Press*, pp. 143-157.

[71] J.B. Dennis, "Data Flow Supercomputers", *Computer*, 18, pp. 42-56.

[72] J.P. Deschamps, "Maximal Classes of Solutions of Boolean Equations", *Philips Research Reports,* Vol. 26., pp. 249-260, August 1971.

b13

[73] D.L. Dietmeyer, "Logic Design of Digital Systems," *Allyn and Bacon,* 1971.

[74] E. Domany and W. Kinzel, Equivalence of cellular automata to Ising models and directed percolation, *Phys. Rev. Lett.* 53, 1984, 311.

[75] B. Dunahm, and H. Wang, "Towards Feasible Solutions to the Tautology Problem", *Ann. Math. Logic*, Vol. 10., pp. 117-154, 1976.

EEEEEEEEEEEE

b14

[76] ELTEC GmbH, "An Important Step in Image Processing," 1991.

[77] C. Espinosa, and M.A. Perkowski: "Implementation of a Hierarchical Hough Transform on a Personal Computer", Preprint. 1991.

[78] C. Espinosa, M.A. Perkowski, "Hierarchical Hough Transform for the Vision System of a Wheelchair Robot", *IPCCC'92, IEEE International Phoenix Conference on Computers and Communications,*

[79] A.C. Ewing, et al., "Algorithms for Logical Design", *Comm. & Electronics,* No. 56., pp. 450-458, 1961.

FFFFFFFFFFFF

[80] B.J. Falkowski, and M.A. Perkowski: "Algorithms for the Calculation of Hadamard-Walsh Spectrum for Completely and Incompletely Specified Boolean Functions", *Proceedings of IEEE International Phoenix Conference on Computers and Communication*, Scottsdale, Arizona, March 1990, pp. 868-869.

[81] B.J. Falkowski, and M.A. Perkowski: "Walsh Type Transforms for Completely and Incompletely Specified Multiple-Valued Input Binary Functions" *Proc. of the 20th ISMVL, International Symposium on Multiple-Valued Logic,* Charlotte, NC, pp. 75-82, May 1990.

[82] B.J. Falkowski, and M.A. Perkowski: "Algorithm and Architecture for Gray Code Ordered Fast Walsh Transform", *Proc. of the ISCAS'90, International Symposium on Circuits and Systems,* New Orleans, 1-3 May 1990, pp. 1596-1599.

[83] B.J. Falkowski, and M.A. Perkowski: "A Family of All Essential Radix-2 Addition/Subtraction Multi-Polarity Transforms: Algorithms and Interpretations in Boolean Domain", *Proc. of the ISCAS'90, International Symposium on Circuits and Systems,* New Orleans, 1-3 May 1990, pp. 2913-2916.

[84] B.J. Falkowski, and M.A. Perkowski: "Essential Relations between Classical and Spectral Approaches to Analysis, Synthesis and Testing of Completely and Incompletely Specified Boolean Functions", *Proc. of the ISCAS'90, International Symposium on Circuits and Systems,* New Orleans, 1-3 May 1990, pp. 1656-1659.

[85] B.J. Falkowski, and M.A. Perkowski, "One More Method for the Calculation of the Hadamard-Walsh Spectrum for Completely and Incompletely Specified Boolean Functions", *International Journal of Electronics,* Vol. 69, No. 5, pp. 595-602, Nov. 1990.

[86] B. Falkowski, I. Schaefer, and M. Perkowski, "A Fast Computer Algorithm for the Generation of Disjoint Cubes for Completely and Incompletely Specified Boolean Functions", *Proc. of the Midwest Symp. on Circuits and Systems,* August 1990, Alberta, Canada, pp. 1119-1122.

[87] B.J. Falkowski, and M.A. Perkowski, "On the Calculation of Generalized Reed-Muller Canonical Expansions from Disjoint Representation of Boolean Functions", *Proc. of the 33rd Midwest Symp. on Circuits and Systems*, August 1990, Alberta, Canada, pp. 1131-1134.

[88] B.J. Falkowski, and M.A. Perkowski, "An Algorithm for the Generation of Disjoint Cubes for Completely and Incompletely Specified Boolean Functions", *International Journal of Electronics,* Vol. 70, No. 3, pp. 533-538, March 1991.

[89] B.J. Falkowski, and M.A. Perkowski, "One More Way to Calculate Generalized Reed-Muller Expansions of Boolean Functions", *International Journal of Electronics,* Vol. 71, No. 3, pp. 383-396, September 1991.

[90] B.J. Falkowski, and M.A. Perkowski, "An Algorithm for the Generation of GRM forms", *International Journal of Electronics.* September, 1991.

[91] B. Falkowski, I. Schaefer, and M. Perkowski, "Effective Computer Methods for the Calculation of Rademacher-Walsh Spectrum for Completely and Incompletely Specified Boolean Functions", *IEEE Trans. on Computer-Aided Design,* ??? 1993.

[92] D. Farmer, T. Toffoli, and S. Wolfram, (Eds) Cellular automata: Proceedings of the interdisciplinary workshop, *Physica D*, 10 (1,2), 1984.

[93] J. Florine, "Optimization of Binary Functions with a Special-Purpose Electronic Computer", *Automation and Remote Control,* Vol. 28., pp. 956-962, 1967.

[94] J. Florine, "The Design of Logical Machines", *Crane, Russak & Co.,* New York, 1973.

[95] D. Foote, “The Design, Realization and Testing of the ILU of the CCM2 Using FPGA Technology,” *Master Thesis*, Portland State University, Department of Electrical Engineering, 1994.

b15

[96] J.A.B. Fortes, K.S. Fu, and B.J. Wah, ”Systematic Approaches for Algorithmically Specified Systolic Arrays in Computer Architecture: Concepts and Systems, *Milutinovic ed.,* North Holland, 1988.

[97] R. Freivald, ”Probabilistic Machines Can Use Less Running Time”, *IFIP 77*, North-Holland, Amsterdam, 1977, pp. 839-842.

[98] U. Frisch, B. Hasslacher, and Y. Pomeau, Lattice-gas automata for the Navier-Stokes equation, *Phys. Rev. Lett.* 56, 1986, 1505.

[99] M. Fujita, Y. Matsunaga, T. Kakuda, and K-Ch. Chen, ”On Application of Boolean Unification to Combinational Logic Synthesis”. *Proc. Intern. Workshop on Logic Synthesis*, May 7-10, 1991. Vol 1.

[100] M. Fujita, Y. Tamija, Y. Kukimoto, and K-Ch. Chen, ”Application of Boolean Unification to Combinational Logic Synthesis”. *Proc. ICCAD’91*, Nov. 1991, pp.510-513.

[101] H. Fujiwara, *Logic Testing and Design for Testability,* Computer System Series, The MIT Press, 1986.

b16

[102] F. Furtek, G. Stone, and I.W. Jones, ”Labyrinth: A Homogeneous computational medium”, *Proc. IEEE Custom Integrated Circuits Conference,* May 1990, paper 31.1.

b17

[103] F. Furtek, ”An FPGA Architecture for Massively Parallel Computing”, in *FPL’92.*

GGGGGGGGGG

[104] M. Gardner, ”Logic Machines and Diagrams”, McGraw-Hill, 1958. Harvester, Brighton, 1983. (FIND IT, IT IS GREAT, I SAW IT IN MINNESOTA BUT NOT HERE).

b18

Gare 79

[105] M.R. Garey, and D.S. Johnson, "Computers and Intractability. A Guide to the Theory of NP-Completeness", *W.H. Freeman and Company,* San Francisco 1979.

[106] H.L. Garner, and J.S., Squire, "Iterative Circuit Computers" in "Computer Organization", Barnum (ed.), *Spartan Books,* 1963.

[107] M.A. Gavrilov, and A.D. Zakrevskii (eds.), "LYaPAS: A Programming Language for Logic and Coding Algorithms", *Academic Press,* New York, 1969.

[108] M.R. Genesereth, and N.J. Nilsson, "Logical Foundations of Artificial Intelligence", *Morgan Kaufmann,* Los Altos, CA, 1987.

b19

[109] G.B. Gerace, et al, "TOPI-A Special-Purpose Computer for Boolean Analysis and Synthesis", *IEEE TC,* Vol. C-20, pp. 837-842, Aug. 1971.

[110] P. Gilliam, "A Practical Parallel Algorithm for the Minimization of Kronecker Reed-Muller Expansions", *M.S. Thesis*, Portland State Univ., August 1991.

b20

[111] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweely, and D. Lopresti, "Building and Using a Highly Parallel Programmable Logic Array", in *FPL '92*.

[112] H.H. Goldstein, *"The Computer from Pascal to von Neumann",* Princeton University Press, Princeton, 1972.

[113] L. Gomez-Gonzalez, "Estudio Teorico, Concepcion y Realizacion de un Sistema Electronico Para Simplificar Funciones Logicas", *Dissertation,* Dpto. Electricidad y Electronica, Facultad de Ciencias, Universidad de Granada, Spain, 1977.

[114] R.L. Goodstein, *"Development of Mathematical Logic",* New York, Springer Verlag, 1971.

[115] A. Gottlieb, and J.T., Schwartz, "Networks and Algorithms for Very-Large-Scale Computation", *Computer*, January 1982, pp. 27-36.

[116] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer", *IEEE Trans. on Computers*, C-32, 2, pp. 175-189.

b21

[117] J.P. Gray, and T.A. Kean, "Configurable Hardware: A New Paradigm for Computation", *Proc. Decennial Caltech Conference on VLSI,* Pasadena, CA, March 1989.

[118] D. Green, *Modern Logic Design,* Electronic Systems Engineering Series, 1986.

[119] H. Gutowitz, (Ed), Cellular Automata, MIT/North-Holland, 1991. (reprints from Physica D), published by *Center for Nonlinear Studies at Los Alamos National Laboratory.*

HHHHHHHHHHHHHHHh

[120] P.R. Halmos, "Lectures on Boolean Algebras", *Springer-Verlag,* New York, 1974.

b22

[121] R. Halverson, "Hawaii Parallel Computer Project: the Boolean Processor", in *FPL'92.*

[122] P.L. Hammer, and S. Rudeanu, "Boolean Methods in Operations Research", *Springer-Verlag,* New York, 1968.

[123] M.A. Harrison, "Introduction to Switching and Automata Theory", *McGraw-Hill,* New York, 1965.

[124] R.W. Hartenstein, K. Schmidt, et al, "A Novel Compilation Technique for a Machine Paradigm Based on Field-Programmable Logic," in *FPGAs,* pp. 255-270.

b24

[125] R.W. Hartenstein, and K. Lemmert, "A CHDL-based CAD-system for the synthesis of Systolic Architectures," *Proc. Intl. Symposium on Hardware Description Languages and their*

Applications, Washington D.C., North Holland Publ. Company, Amsterdam 1989.

b25

[126] R.W. Hartenstein, A.G. Hirschbiel, and M. Weber, "A Novel Paradigm of Parallel Computation and its Use to Implement Simple High Performance Hardware", *Proc. of Joint Conference on Vector and Parallel Processing, CONPAR 90*, VAPP IV, Zurich, Sept, 1990.

b26

[127] R.W. Hartenstein, A.G. Hirschbiel, and M. Weber, "XPUTERS: An Open Family of Non-von Neumann Architectures", *Nr. 195/89,* Universitaet Kaiserlautern, Fachbereich Informatik, Postfach 3049, D-6750 Kaiserlautern, 1989.

b27

[128] R.W. Hartenstein, et al, "A Novel Paradigm of Parallel Computation and its use to implement simple High Performance Hardware", *Intl. Conf. on Information Technology,* Tokyo, Japan. Oct. 1990.

[129] R.W. Hartenstein, K. Schmidt, H. Reinig, and M. Weber, "A Novel Compilation Technique for a Machine Paradigm Based on Field-Programmable Logic," in *FPGAs,* pp. 255-270.

[130]

[131] R.W. Hartenstein, et al, "The machine Paradigm of Xputers and its Application to Digital Signal Processing", *Proc. of 1990 Intern. Conf. on Parallel Processing,* St. Charles, Oct. 1990.

b29

[132] R.W. Hartenstein, et al, "A Novel ASIC Design Approach Based on New Machine Paradigm", *IEEE Journal of Solid-State Circuits,* July 1991, Vol. 26, Nr. 7, pp..

[133] R. Hartley, "The Stanhope Demonstrator", *Mind,* Vol. 4, April 1979.

[134] P.B. Hawthorn, and D. DeWitt, "Performance Analysis of Alternative Database Machine Architectures", *IEEE Trans. on Software Engineering* SE-8, 1, pp. 61-74, 1982.

[135] L.S. Haynes, R.L. Lau, D.P. Siewiorek, and D.W. Mizell, "A Survey of Highly Parallel Computing", *Computer*, January 1982.

b30

[136] B. Heeb, and C. Pfister, "Chameleon: A Workstation of a Different Colour", in *FPL '92.*

[137] M. Helliwell, and M.A. Perkowski: "A Fast Algorithm to Minimize Multi-Output Mixed-Polarity Generalized Reed-Muller Forms", *Proc. 25-th ACM/IEEE Design Automation Conference,* paper 28.2, pp. 427-432, June 12- June 15, 1988.

[138] M. Helliwell, and M.A. Perkowski, "GAL-based Hardware Accelerator to Find Exact Minimum Solutions for Mixed-Polarity Generalized Reed Muller Forms", *report PSU.*

[139] C.E. Hewitt, "The Apiary Network Architecture for Knowledgeable Systems", *Proceedings of Lisp Conference*, Stanford, pp. 107-118, 1980.

[140] W.D. Hillis, *"The Connection Machine",* The MIT Press, Cambridge, Massachusetts, 1985.

[141] D.R. Hofstadter, *"Goedel, Escher, Bach: An Eternal Golden Braid",* Vintage Books, 1980.

[142] A. Hodges, *"Alan Turing: The Enigma of Intelligence",* Unwin, London, 1983.

Hong74

[143] S.J. Hong, R.G. Cain, and D.L. Ostapko: "MINI: A heuristic approach for logic minimization", *IBM J. Res. Develop.,* Vol. 18, pp. 443 - 458, Sept. 1974.

[144] B. Ho, "NAND Synthesis of Multiple-Output Combinational Logic Using Implicants Containing Output Variables", *Ph.D. Dissertation,* U. of Wisconsin, 1976.

b33

[145]

[146] P.M. Ho, and M.A. Perkowski, "Systolic Architecture for Solving NP-Hard Combinational Problems of Logic Design and Related Areas", *Proc. ISCAS'89,* pp. 1170-1173, 1989.

[147] P.M. Ho, and M.A. Perkowski: "Evaluation of Systolic Architecture for Solving NP-Hard Combinatorial Problems of Logic Design and Related Areas", *Report*, PSU.

b32 P.M. Ho, and M.A. Perkowski, "Massively Parallel Processors for Solving Combinational Problems", *Portland State University*, Portland, OR, 1992.

[148] F. Hohn, "Applied Boolean Algebra", *Macmillan,* Second Edition, New York & London, 1966.

[149] D.A. Huffman, "Solvability Criterion for Simultaneous Logical Equations", *MIT Research Lab. of Electronics,* Quarterly Progress Report No. 48, AD 156-161, 15 Jan. 1958.

[150] S.L. Hurst, *The Logical Processing of Digital Signals, Crane-Russak,* New York and Edward Arnold, London, 1978.

[151] Hwang, K., Briggs, F.A, "Computer Architecture and Parallel Processing" *McGraw-Hill*, New York, 1984.

IIIIIIIIIII

[152] T. Ibaraki, "Theoretial comparisons of search strategies in branch-and-bound algorithms," *Intern. Journ. Comp. Sci.,* 5 (1976), pp. 315-344.

[153] M.E. Isenman, and D.E. Shasha, "Performance and Architectural Issues for String Matching", *IEEE JSSC,* Vol. 39, No. 2, pp. 238-250, Febr. 1990.

JJJJJJJJJJJ

[154] P.C. Jackson, *"Introduction to Artificial Intelligence",* New York, Petrocelli Charter, 1975.

[155] W.S. Jevons, *"The Principles of Science",* Dover, New York, 1958.

[156] W.S. Jevons, "Pure Logic or the Logic of Quality Apart from Quantity", *Stanford,* London, 1864.

[157] W.S. Jevons, "On the Mechanical Performance of Logical Inference", *Philosophical Transactions of the Royal Society,* CLX, 1870. pp. 497-518.

b35 John xx

[158] D. Johnson, "The NP-Completeness Column: An Ongoing Guide". *Journal of Algorithms,* Academic Press, each issue.

KKKKKKKKKKKKK

[159] W.C. Kabat, and A.S. Wojcik, "Automated Synthesis of Combinational Logic Using Theorem-Proving Techniques", *Proc. 12 Int. Symp on Multiple-Valued Logic,* pp. 178-199, May 1982. IEEE Trans. on Comp. Vol. C-34, No. 7., pp. 610-632, July 1985.

[160] D. Kalish, and R. Montague, "Logic: Techniques of Formal Reasoning", *Harcourt Brace Jovanovich,* New York, 1964.

[161] R. Kane, and S. Sahni, "A systolic design rule checker". TR-83-13, Computer Science Dept., University of Minnesota, July 1983.

b36

[162] Ch. J. Kappler, "The MusiCAL Lambda Reduction Machine", in *FPL '92.*

[163] R.M. Karp, "Reducibility Among Combinatorial Problems". Complexity of Computer Computation, *Plenum Press,* ed. Miller, pp. 85-103, New York, 1972.

[164] V.I. Kashirov, et al., "Problems in Realizing the L-Machine", *Dokl. Rev. Ustr. Kon. Avt,* p. 138, 1962.

b37

[165] E. Katona, "Cellular Processing", Chapter 6 in "Fuzzy, Holographic, and Parallel Intelligence, By B. Soucek and the IRIS Group, pp. 215-229, John Wiley & Sons, 1992.

b38

[166] T. Kean, "Using CAL to Accelerate Maze Routing of CAL Designs", in *FPL '92.*

[167] P. Kerntopf, and A. Michalski, "Selected Problems in Synthesis of Combinatorial Logic Circuits," *PWN*, Warsaw, 1972 (in Polish).

[168] L. Kida, "Documentation of Cube Calculus Machine", PSU EE Dept. Report, 1989

b39

[169]

[170] L. Kida, and M.A. Perkowski, "The Cube Calculus Machine: A Ring of Asynchronous Automata to Process Multiple-Valued Boolean Functions", *Proc. of the ISCAS'92, International Symposium on Circuits and Systems,* Sheratons on Harbor Island, San Diego, CA, May 10-13, 1992, pp. 807-810.

[171] L. Kida, "Associative Processing Implemented with Content-Addressable Memories", M.S. Thesis, PSU, 1991.

[172] S. Kleene, *"Introduction to Mathematical Logic",* New York, John Wiley, 1967.

[173] G.J. Klir, "Introduction to the Methodology of Switching Circuits", *D. Van Nostrand Co.,* New York, 1972.

[174] Koestler, A., *"The Art of Creation",* New York: Dell, 1966.

[175] Kung, H.T., "Why Systolic Architectures", *IEEE Computer,* pp. 37-46, January 1982.

[176] Kung, H. T., and P.L. Lehman, "Systolic (VLSI) arrays for relational database operations," *International Conference on Management of Data,* May 1980.

[177] H.T. Kung, and C.E. Leiserson, "Systolic Arrays", in "Introduction to VLSI Systems", C.A. Mead and L.A. Conway, *Addison-Wesley.*

[178] Kung, H.T., "Special Purpose Devices for Signal and Image Processing: An Opportunity in Very Large Scale Integration (VLSI)", *SPIE*, Vol.241, Real Time Signal Processing III, pp.74-84, 1980.

[179] S.Y. Kung, "On Supercomputing with Systolic/Wavefront Array Processors", *Proc. IEEE*, Vol.72, pp.867-884, July 1984.

[180] Y.S. Kuo, "Generating Essential Primes for a Boolean Function with Multiple-Valued Inputs", *IEEE Trans. on Computers*, Vol. C-36, pp. 356-359, March 1987.

LLLLLLLLLLLLL

[181] T. Lang, and H.S. Stone, "A Shuffle-Exchange Network with Simplified Control", *IEEE Transactions on Computers*, C-25, 1, pp. 55-65, 1976.

[182] V.G. Lazarev, and Yu. L. Sagalovich, "The Switchboard of a Machine for Synthesizing Contact Networks", *Prob. Pered. Infor.,* Vol. 4, pp. 124-132, 1959.

[183] V.G. Lazarev, and P.P. Parkhomenko, "A Machine for the Process of Analysis and Synthesis of the Structure of Relay Devices", *Proc. of the First Intern. Congress of the Intern. Federation of Automatic Control*, Moscow, 1960, (Butterworth, London, 1961), Vol. 2, pp. 853-859.

[184] V.G. Lazarev, O. A. Oganov, and V.N. Roginskii, "Fundamentals in the Design of Contactless Machines for the Synthesis of Relay-Contact Networks", *Prob. Pered. Infor.,* Vol. 8, pp. 5-19, 1961.

b41

[185] H. V. D. Le, and M. A. Perkowski: "A New General Purpose Systolic Architecture for Matrix Computations", *Proc. of the Intern. Conf. on Computer and Information Systems,* Toronto, Ontario, Canada, May 23-29, 1989, pp. 182-185.

[186] H.V.D. Le, and M.A. Perkowski, "Size Independent Implementation of Matrix Operations on TASA - A Two-Dimensional Array Matrix Architecture", *Proc. of the IEEE International Phoenix Conference on Computers and Communication,* Scottsdale, Arizona, March 1990, pp. 889-890.

[187] H. V. D. Le, and M. A. Perkowski, "Realization of Extensions to Faddev Algorithm on Array of SIMD Processors", *Proc. of the ISCAS'90, International Symposium on Circuits and Systems,* New Orleans, 1-3 May 1990, pp. 2312-2315.

[188] R.M. Lea, "SCAPE: a Single-chip Array Processing Element for Signal and Image Processing", *IEE Proceedings,* Pt. E., Vol. 133, No. 3., pp. 145-151, May 1986.

[189] R.M. Lea, "VLSI and WSI Associative String Processor for Structured Data Processing", *IEE Proceedings,* Pt. E., Vol. 133, No. 3., pp. 153-162, May 1986.

[190] R.M. Lea, "ASP: A Cost-effective Parallel Microcomputer", *IEEE Micro,* Vol. 8, No. 5, pp. 10-29, October 1988.

[191]

[192] E. B. Lee, and M. Perkowski, "A new approach to structural synthesis of automata". *University of Minnesota, Department of Electrical Engineering,* preprint, 1982.

[193]

[194] E. B. Lee, and M. Perkowski, "Concurrent Minimization and State Assignment of Finite State Machines", *Proceedings of the 1984 Intern. Conf. on Systems, Man, and Cybernetics,* IEEE, Halifax, Nova Scotia, Canada, October 9 - 12, 1984.

[195] C.Y. Lee, "Intercommunicating Cells, Basis for a Distributed-Logic Computer", *Proc. of 1962 Fall Joint Computer Conference.*

[196] C.Y. Lee, and M.C. Paul, "A Content-Addressable Distributed-Logic Memory with Applications to Information Retrieval", *IEE Proceedings*, 51, pp. 924-932, 1963.

[197] R.S. Ledley, "Mathematical Foundations and Computational Methods for a Digital Logic Machine", *J. Ops. Res. Soc. Amer.,* Vol. 2, pp. 249-274, 1954.

[198] R.S. Ledley, "Digital Computational Methods in Symbolic Logic, with Examples in Biochemistry", *Proc. Nat'l. Acad. Sci.,* Vol. 41., pp. 498-511, July 1955.

[199] R.S. Ledley, "Logical Aid to Systematic Medical Diagnosis (and Operational Simulation in Medicine)", *J. Ops. Res. Soc. Amer.,* Vol. 4, No.3., p. 392, Aug. 1956.

[200] R.S. Ledley, "Reasoning Foundations of Medical Diagnosis", *Science,* Vol. 130, No. 3366, pp. 9-21, 3 July 1959.

[201] R.S. Ledley, "Digital Computer and Control Engineering", *McGraw-Hill Book Co.,* New York, 1960.

[202] R.S. Ledley, "Use of Computers in Biology and Medicine", *McGraw-Hill Book Co.,* New York, 1960.

[203] G. Leibniz, *"On the Art of Combination",* in "Parkinson Logical Papers".

[204] Ch.E. Leiserson, "FAT-TREES: Universal Networks for Hardware-Efficient Supercomputing", *1985 International Conference on Parallel Processing,* IEEE Computer Society, August 1985.

b42

[205] A. Linde, T. Nordstroem, and M. Taveniku, "Using FPGAs to Implement a Reconfigurable Highly Parallel Computer", in *FPL '92.*

[206] L. Loewenheim, "Uber die Aufloesung von Gleichungen im Logischen Gebietekalkul", *Math. Ann.* Vol. 68., 1910, pp. 169-207. *Translation:* "The Solution of Equations in the Calculus of Logic", AFCRL-69-0149, Air Force Cambridge Research Laboratories, April 1969.

[207] D. Luckham, "The Resolution Principle in Theorem-Proving", *Machine Intelligence I,* (N.L. Collins and D. Michie, Eds.), *Oliver & Boyd,* Edinburgh & London, 1967.

b43

[208] W. Luk, and I. Page, "Parametrising Designs for FPGAs", in *FPGAs,* pp. 285-295.

[209] Raymon Lullus, (a.k.a. Ramon Llull), "Ars Major", "Ars Generalis", "Ars Veritatis Inventiva",

MMMMMMMMMMMMMM

[210] G. Mago, "A Network of Microprocessors to Execute Reduction Languages, Part I", *Intern. Journal of Computer and Information Sciences*, 8, 5, pp. 349-385.

[211] J. Maluszynski, and H.J. Komorowski, "Unification-free Execution of Logic Programs", *Proc. of the 1985 Symposium on Logic Programming,* July 15-18, Boston, MA, pp. 78-86, 1985,

[212] E. Marczewski, "Independence in Algebras of Sets and Boolean Algebras", *Fundamenta Mathematicae,* Vol. 48, pp. 135-145, 1960.

b44

[213] M.A. Marin, "Investigation of the Field of Problems for the Boolean Analyzer", *Ph.D. Dissertation,* Univ. of California, Los Angeles, 1971. early version in Report No. 68-28.

[214] A. Marquand, "A Logical Diagram for N terms", *Philosophical Magazine,* Vol. 12., pp. 266-270, 1881.

[215] A.J. Martin, "A Synthesis Method for Self-Timed VLSI Circuits", *Proc. 15th Annual International Symposium of Computer Architecture,* May 30-June 2, 1988, Honolulu, Hawaii, pp. 224-229.

[216] A.J. Martin, S.M. Burns, T.K. Lee, D. Borkovic, and P.J. Hazewindus, "The First Asynchronous Microprocessor: The Test Results", *Proc. 16th Annual International Symposium of Computer Architecture,* May 28-June 1, 1989, Jerusalem, Israel, pp. 95-110.

[217] U. Martin, and T. Nipkow, "Unification in Boolean Rings", *Journal of Automated Reasoning,* Vol. 4., pp. 381-396, 1988.

[218] U. Martin, and T. Nipkow, "Boolean Unification - the Story so Far", *Journal of Symbolic Computation,* Vol. 7., pp. 275-293, 1989.

[219] W. Mays, and D. G. Prinz, "A Relay Machine for the Demonstration of Symbolic Logic", *Nature,* 165, 1950, p.197.

[220] W. Mays, "Note on the Exhibition of Logical Machines", *Mind,* Vol. 60, April 1951, p. 262.

[221] W. Mays, and D. Henry, "Exhibition of the Work of W. Stanley Jevons", *Nature,* 170, 1952, pp.696-697.

[222] W. Mays, and D. Henry, "Jevons and Logic", *Mind,* LXII, 1953, pp.484-505.

[223] H. McCall, "Symbolic Reasoning", *Mind,* 1880, p. 46.

b45

[224] J.T. McCall, J.G. Tront, F.G. Gray, R.T. Haralick, and W.M. McCormack, "Parallel Computer Architectures and Problem Solving Strategies for the Consistent Labeling Problem", *IEEE TC.,* Vol. C-34, No. 11, Nov. 1985.

[225] C. Mead, and L. Convay, "Introduction to VLSI Systems". *Addison-Wesley*, 1980.

[226] Mentor, Patent for emulation machine, 1992.

[227] R.E. Michalski, and R.E. Flick, "Automated Construction of Classifications: Conceptual Clustering Versus Numerical Taxonomy", *Pattern Matching and Machine Intelligence*, July 1983, pp. 410-416, 1983.

[228] M. Minsky, "Computation: Finite and Infinite Machines", *Prentice-Hall*, 1967.

[229] M. Minsky, and S. Papert, "Perceptrons", MIT Press, second edition, 1972.

[230] M. Minsky, "K-Lines: A Theory of Memory", Massachusetts Institute of Technology Artificial Intelligence Laboratory Memo 514, Reprinted in Cognitive Science, 1980, pp. 117-133.

[231] D. Mithani, "Implementation of NAND Synthesis Using Implicants Containing Output Variables", *M.S. Thesis,* Dept. Electr. Engng., Univ. of Wisconsin, 1977.

[232] E.F. Moore, "Sequential Machines: Selected Papers", *Addison-Wesley*, 1964.

b47

[233] W. Moore, and W. Luk, "FPGAs," Edited from the Oxford 1991 International Workshop on Field Programmable Logic and Applications, *Abingdon EE & CS Books,* Abingdon England,1991.

[234] P. and E. Morrison, *"Calculating Engines",* (WHERE TO FIND IT?)

[235] P. Morrison, and J. Morrison, (Eds.), *"Charles Babbage and His Calculating Engines",* New York: Dover Publications, 1961.

NNNNNNNNNNNN

[236] T.A. Nodes, J.L. Smith, and R. Hecht-Nielsen, "A Fuzzy Associative Memory Module and its Applications to Signal Processing", *Proc. Intern., Conf. on Acoustics Speech and Signal Processing, ICASSP,* pp. 1511-1514, New York, 1985.

[237] A. Newell, J.C. Shaw, and H.A. Simon, "Empirical Explorations with the Logic Theory Machine: A Case Study in Heuristics", *Proceedings of the Western Joint Computer Conference,* 15, 1957, in Feigenbaum and Feldman, "Computers and Thought", p. 115.

[238] A. Newell, J.C. Shaw, and H.A. Simon, "The Logic Theory Machine", in Feigenbaum and Feldman, *"Computers and Thought",* pp. 121 - ??.

[239] A. Newell, and H.A. Simon, "GPS, A Program that Simulates Human Thought", in Feigenbaum and Feldman, *"Computers and Thought",* p. 286.

[240] L. Nguyen, M. Perkowski, and N. Goldstein: "PALMINI - Fast Boolean Minimizer for Personal Computers", *Proc. of the 24th Design Automation Conference,* June 28 - July 1, 1987, Miami, Florida, Paper 33.3, pp. 615-621.

[241] J. Nievergelt, J.C. Farrar, and E.M. Reingold, *"Computer Approaches to Mathematical Problems",* Englewood Cliffs, N.J.: Prentice Hall, 1974.

OOOOOOOOOOOOOOOoo

[242] S.E. Orcutt, "Implementation of Permutation Functions in ILLIAC IV-Type Computers", *IEEE Trans. on Computers,* C-25, 9, pp.929-936, 1976.

[243] J.V. Oldfield, "Logic Programms and an Experiment Architecture for their Execution", *IEE Proc. Pt.E.,* Vol. 133, No. 3, pp. 163-167, May 1986.

[244] J.V. Oldfield, Ch. Storman, and M. Brule, "The Application of VLSI Content-Addressable Memories to the Acceleration of Logic Programming Systems", *Proc. of VLSI and Computers, First Intern. Conf. on Computer Technology, Systems, and Applications,* pp. 27-30, IEEE, Hamburg, May 11-15, 1987.

[245] J.V. Oldfield, R.D. Williams, and N.E. Wiseman, "Content-Addressable Memories for Storing and Processing Recursively Subdivided Images and Trees", *Electronic Letters,* Vol. 23, No. 6, pp. 262-263, 12 March 1987.

[246] O'Keefe, R.A., "A Comment on "A Hardware Unification Unit: Design and Analysis", *Computer Architecture News,* Vol. 14, No. 1, pp. 2-3, January 1986.

[247] S.A. Ozkarahan, S.A. Schuster, and K.C. Sevcik, "A Data Base Processor", Tech. Rep. CSRG-43, *Computer Systems Research Group,* University of Toronto, 1974.

[248] S.A. Ozkarahan, S.A., Schuster, and K.C. Sevcik, "Performance Evaluation of a Relational Associative Processor", *ACM Trans. Database Systems*, 2, 2, 1977.

PPPPPPPPPPPPPP

b48

[249] I. Page, and W. Luk, "Compiling OCCAM into FPGAs", in *FPGAs,* pp. 271-283.

[250] Ch.A. Papachristou, "Content-Addressable Memory Requirements for Multivalued Logic", *11th Int. Symp. on Multi-Valued Logic,* pp. 62-72, 1981.

[251] B. Parhami, "Rapid: A Rotating Associative Processor for Information Dissemination", UCLA-ENG-7213, *University of California*, Los Angeles.

[252] P.P. Parkhomenko, "Automatizing the Process of Analyzing Relay Contact Networks", *Vses. Mos. Sov. Teorii Ust. Rel. Deist.,* p. 70, 1957.

[253] P.P. Parkhomenko, "A Large Capacity Logic Machine for Relay Circuit Analysis", *Izv. A.N. O.T.N. E i A.,* No. 3., pp. 179-180, 1959.

[254] P.P. Parkhomenko, "Machine Analysis of Relay Networks", *Avt. i Telem.* Vol. 20, No. 4., pp. 486-497, 1959.

[255] P.P. Parkhomenko, "The Principles of Mechanization of the Analysis of Relay-Contact Networks", *Dokl. A.N.,* Vol. 124, No. 1., pp. 83-86, 1959.

[256] P.P. Parkhomenko, "A Large Capacity General-Purpose Computer for the Analysis of Relay Networks", *Teoriya i Prim. Disk. Avt. Sistem,* 1960.

[257] C.S. Peirce, ed., "Studies in Logic", *By Members of the Johns Hopkins University, Little Brown & Co.,* Boston, 1883.

[258] C.S. Peirce, "Logical Machines", *Amer. J. Psychology,* Vol. 1. pp. 165-170, 1887.

[259] M. Perkowski, "Relational-structure languages and their application in the system for automatic design of block

synthesis of digital systems of automatic control". *Proceedings of the 6th National Conference on Automatic Control,* Poznan, Poland, 9-11 September 1974, Vol. 1, pp. 695-707 (in Polish).

[260] M. Perkowski, "An example of heuristic programming application in the three-level combinational logic design". Proceedings of the 3rd Symposium on heuristic methods. Polish Cybernetical Society Warsaw, 25 September 1976, Vol. 1. pp. 105-132.

[261] M. Perkowski, "Synthesis of multioutput three level NAND networks". Proc on the Seminar on Computer Aided Design. Budapest 3-5 November 1976, pp.238-265.

[262] M. Perkowski, "Some concepts on reasoning by analogy. The heuristic programming approach". *Proceedings of IVth International Symposium on Heuristic Methods,* Warsaw, 24 September, 1977, Vol. 2, pp. 25-119.

[263] M. Perkowski, A. Rydzewski, and P. Misiurewicz, "Theory of Logic Circuits, Selected Problems". *Publishers of the Technical University of Warsaw,* Ed. 1 - 1977, Ed. 2 - 1978, Ed. 3 - 1984 (in Polish).

[264] M. Perkowski, "An application of general problem-solving methods in computer-aided design: the MULTICOMP system and its problem-oriented source language". *Proceedings of IVth International Symposium on Heuristic Methods,* Warsaw, 24 September, 1977, Vol. 3, pp. 55-102.

[265] M. Perkowski, "The state-space approach to the design of multipurpose problem-solver for logic design". *"Artificial Intelligence and Pattern Recognition in Computer-Aided Design",* J. C. Latombe (ed.), North Holland, Amsterdam, pp. 124-140, 1978. (book chapter).

[266] M. Perkowski, "The state-space approach to the design of multipurpose Conference "Artificial Intelligence and Pattern Recognition in Computer-Aided Design". Grenoble, France, 17-19 March 1978, J. C. Latombe (ed.) North Holland, Amsterdam, pp. 124-140, 1978.

[267] M. Perkowski, and M. Gluch, "Knowledge Engineering Language Using Strategies (KELUS)", *Institute of Automatic Control, Technical University of Warsaw,* 1979.

[268] M. Perkowski, and A. Zasowska, "Minimal area MOS asynchronous automata". Proceedings of the International Symposium on Applied Aspects of Automata Theory, Warna, Bulgaria, 14-19 May 1979, pp. 284-298.

[269]

[270] M. Perkowski, "The method of solving combinatorial problems in the automatic design of digital systems". Institute of Automatic Control, Technical University of Warsaw, Ph.D. Thesis, 1980, (in Polish).

[271] M. Perkowski, "Multistrategical Problem Solver," Proceedings of the 2nd International Meeting on Intelligent Robotics and Knowledge

[272] b49

[273] M. Perkowski, "General methods for solving combinational problems". Chapter 4 in A. Goralski (ed.) "Problem, method, solution", Vol. 4 Scientific-Technical Publishers, Poland, Warszawa 1982.

[274] M. Perkowski, A Goralski, and G. Zielinski, "Elements of Artificial Intelligence." Book in Polish. Preprint 1982.

[275] M. Perkowski, B. Goldstein, "A new algorithm for multi-output Boolean functions minimization based on reduction to graph coloring." University of Minnesota., Department of Electrical Engineering, report.

[276] M. Perkowski, "Fast Approach to Multiple-Valued Constraints Problem: Systolic VLSI Realization of System MULTICOMP", *PSU Report*, December 1984.

[277] M. Perkowski, "Design of Finite State Machines with SuperPeg", Seminar of Department of Electrical Engineering, Oregon State University, Corvallis, Oregon, February 12, 1985.

[278]

[279] M. Perkowski, "A Systolic Processor for Approximate Solutions to NP-Complete Combinatorial Problems". Symposium on Complexity of Approximately Solved Problems. Computer Science Department, Columbia University, New York, NY 10027, April 17 -19 , 1985.

[280]

[281]

[282] M.A. Perkowski, "Systolic Architecture for the Logic Design Machine". *Proceedings of the IEEE and ACM International Conference on Computer* Aided Design - ICCAD 85, *Santa Clara, 19-21 November 1985, pp. 133-135.*

[283] M. Perkowski, "Reduction of Problems for the Logic Design Machine". Portland State University, report.

[284] Perkowski, M., Goldstein, B., Nguyen, L. "PLA with tail gives estimation from lower bound and has a decreased area". Preprint.

[285] M. Perkowski, and N. Nguyen, "Minimization of Finite State Machines in SuperPeg". Proceedings of the Midwest Symposium on Circuits and Systems, *Luisville, Kentucky, 22-24 August 1985.*

[286] M.A. Perkowski, and J. Brandenberg, "Solving Basic Boolean Algebra Problems on a Hypercube Computer", Report PSU, *1989.*

[287] M.A. Perkowski, "A Universal Logic Machine", Proc. of the IEEE ISMVL'92, the 21st International Symposium on Multiple-Valued Logic, *Sendai, Japan, May 27-29, 1992, pp. 262-271, invited address.*

[288] M.A. Perkowski, "The Generalized Orthonormal Expansion of Functions with Multiple-Valued Inputs and Some of its Applications", Proc. of the ISMVL'92, the 21st IEEE International Symposium on Multiple-Valued Logic, *Sendai, Japan, May 27-29, 1992, pp. 442-450.*

[289] M. Perkowski, "On the Reduction of Combinatorial Problems and Parallel Computers for Solving Generic Combinatorial Problems", System Science Seminar PSU, *February 11, 1986.*

bitemperkowski M.A. Perkowski, T. Luba, S. Grygiel, P. Burkey, M. Burns, N. Iliev, M. Kolsteren, R. Lisanke, R. Malvi, Z. Wang, H. Wu, F. Yang, S. Zhou, and J.S. Zhang, "Unified Approach to Functional Decompositions of Switching Functions," Technical Report, Portland State University, 1995.

[290] M. Perkowski, "Logic Design Machine", Carnegie Mellon University, *Department of Electrical Engineering, Invited Lecture, 1986.*

[291] M. Perkowski, D. Smith, and R. Krzywiec: "Logic Simulation/Design/Verification Environment in Prolog", Proc. of the 17th

Annual Pittsburgh Conference on Modelling and Simulation, *24-25 April, 1986, University of Pittsburgh, Pittsburgh, Pennsylvania, pp. 945-958.*

[292] M. Perkowski, "Parallel Programs for Tautology, Satisfiability, and Complementation of Boolean Functions", INTEL Scientific Computers, *Report, September 1986.*

[293] M. Perkowski, and J. Liu, "A System for Fast Prototyping of Logic Design Programs", Proc. of the 30th Midwest Symposium on Circuits and Systems, *Syracuse, New York, August 16-18, 1987, paper TA 2.6.3.*

[294] M. Perkowski, and J.E. Brown, "An Unified Approach to Designs Implemented with Multiplexers and to the Decomposition of Boolean Functions", Proc. of the 1988 ASEE National Conference, *Portland, Oregon, June 19-23, 1988, pp. 1610-1619.*

[295] M.A. Perkowski, M. Helliwell, and P. Wu, "Minimization of Multiple-Valued Input, Multi-Output Mixed-Radix Excllusive Sums of Products for Incompletely Specified Boolean Functions", Proc. of the 19th ISMVL, International IEEE Symposium on Multi-Valued Logic, *Guangzhou, People's Republic of China, May 1989, pp. 256-263.*

[296] M. Perkowski, J. Liu, and J. Brown, "Quick Software Prototyping: CAD Design of Digital CAD Algorithms", In G. Zobrist (ed) "Progress in Computer Aided VLSI Design", *Vol. 1., Ablex Publishing Corp., 1989, pp. 353-401.*

[297] M.A. Perkowski, P. Wu, and K.A. Pirkl, "KUAI-EXACT: A New Approach for Multi-Valued Logic Minimization in VLSI Synthesis", Proc. of the 1989 ISCAS - International Symposium on Circuits and Systems, *May 9-11, 1989, pp. 401-404.*

[298] M. Perkowski, M. Helliwell, and P. Wu, "Minimization of Multiple-Valued Input Multi-Output Mixed-Radix Exclusive Sums of Products for Incompletely Specified Boolean Functions", Proc. ISMVL-89,

[299] M.A. Perkowski, P. Dysko, and B.J. Falkowski, "Two Learning Methods for a Tree-Search Combinatorial Optimizer", Proceedings of IEEE International Phoenix Conference on Computers and Communication, *Scottsdale, Arizona, March 1990, pp. 606-613.*

[300] M.A. Perkowski, and J. Liu, "Generation of Finite State Machines from Parallel Program Graphs in DIADES", Proc. of the ISCAS'90,

International Symposium on Circuits and Systems, *New Orleans, 1-3 May 1990, pp. 1139-1142.*

[301] M.A. Perkowski, and J. Liu, "A Program for Synthesis of Three Level NAND Networks", Proc. of the ISCAS'90, International Symposium on Circuits and Systems, *New Orleans, 1-3 May 1990, pp. 1118-1121.*

b62

[302] M.A. Perkowski, and M. Chrzanowska-Jeske, "An Exact Algorithm to Minimize Mixed-Radix Exclusive Sums of Products for Incompletely Specified Boolean Functions", Proc. of the ISCAS'90, International Symposium on Circuits and Systems, *New Orleans, 1-3 May 1990, pp. 1652-1655.*

[303] M.A. Perkowski, and L. Jozwiak, "Two-Dimensional State Minimization of Finite State Machines", planned to be submitted.

[304] M.A. Perkowski, A.Sarabi, and I. Schaefer, "Aplication of Orthogonal Transforms in Image Processing", Proc. of Northcon'91, *Portland, 1-3 October 1991, Session S1, pp. 303-309.*

[305] M. A. Perkowski, and P. Johnson, "Canonical Multivalued-Input Reed-Muller Trees Expressions", Proc. of the Third NASA Symposium on VLSI Design, *Moscow, Idaho, October 30-31, 1991.*

[306] M.A. Perkowski and students, "The Multiple Valued Cube Calculus Machine, CCM2", Portland State University report

[307] M.A. Perkowski and Pan Wu, "A new Approach To Exact Minimization Of Boolean Functions With Multiple Valued Inputs", Diades research group, Portland State University

[308] M.A. Perkowski, Martin Helliwell, Pan Wu, "Minimization Of Multiple Valued Input Multi-Output Mixed-Polarity Generalized Reed-Muller Forms For Incompletely Specified Boolean Functions", Portland State University report.

[309] M.A. Perkowski, "ULM-Lisp, the tutorial.", 1997.

[310] C. Peterson, "iWARP", HOT Chips Symposium Record, *Santa Clara, CA, August 20-21, 1990.*

b54

[311]

[312] S.R. Petrick, "On the Minimization of Boolean Functions", Proc. Symp. on Switch. Theory, *IFIP, Paris, June 1959.*

[313] PiE Design, company broschures, 1992.

[314] P. Poretsky, "On the Methods for Solving Logical Equations and on the Inverse Method for Mathematical Logic", (In Russian), Bull. de la Soc. Physico-Mathematique de Kasan, *Vol. 2., pp. 161-230, 1884.*

[315] G.N. Povarov, "A Method for the Synthesis of Computing and Control Contact Networks", Avt. i Telem., *Vol. 18., No 2., pp. 145-162, 1957.*

[316] D.K. Pradhan, Fault-Tolerant Computing. Theory and Techniques. Vol. I., *Prentice-Hall, 1987.*

[317] Pratt, V., "Thinking Machines. The Evolution of Artificial Intelligence", Basil Blackwell, Inc., *Oxford, U.K., 1987.*

[318] Proceedings of the Workshop on FPGA custom architectures, *Napa Valey, California, April 1993.*

[319] Proceedings of ACM FPGA'92 Workshop, *Hotel Durant, Berkeley, 16-18 Febr. 1992.*

[320] Proceedings of Workshop of Field-Programmable Logic and Applications, *FPL'92, Vienna, Austria, 31 August - 2 September 1992.*

[321] Proceedings of ACM FPGA'92 Workshop, *Hotel Durant, Berkeley, 16-18 Febr. 1992.*

[322] Proceedings of Workshop of Field-Programmable Logic and Applications, *FPL'92, Vienna, Austria, 31 August - 2 September 1992.*

QQQQQQQQQQQQQ

[323] Quickturn company materials, 1991.

[324] M.R. Quillian, "Semantic Memory", in "Semantic Information Processing", M. Minsky (ed.), MIT Press, *pp. 227-270, 1968.*

[325] J.R. Quinlan, "C4.5: Programs for machine learning," San Mateo, CA: Morgann Kaufmann.

RRRRRRRRRRRRR

[326] M.O. Rabin, "Probabilistic Algorithms". In J.Traub (ed.) Algorithms and Complexity: New Directions and Recent Results, Academic Press, New York, 1976, pp.21-39.

[327] B. Randell, "The Origins of Digital Computers", *Springer-Verlag, Berlin, 1975.*

[328] Randell, "On Alan Turing and the Origins of Digital Computers", in B, Melzer and D. Michie (eds), Machine Intelligence *7, Edinburgh University Press, Edinburgh, 1972.*

[329] C.V. Ramamoorthy, J.L. Turner, and B. W. Wah, "A Design of Fast Cellular Associative Memory for Ordered Retrieval", IEEE Trans. on Computers, *Vol. C-27, No. 9, pp. 800-815, September 1978.*

[330] A.P. Reeves, "Parallel Computer Architectures for Image Processing", Proc. of the Intern. Conf. on Parallel Processing, *pp. 199-206, 1981.*

[331] J.C. Ribeiro, C.D. Stormaon, J.V. Oldfield, and M.R. Brule, "Content-Addressable Memories Applied to Execution of Logic Programs" IEE Proceedings, *Vol. 136, Pt. E., No. 5, pp. 383-388, September 1989.*

[332] C. Rieger, "ZMOB: A Mob of 256 Cooperative Z80A-Based Microcomputers", Computer Science Tech. Rep., *Series TR-825, University of Maryland, College Park, MD, 1979.*

[333] C. Rieger, J. Bane, and R. Trigg, "ZMOB: A Highly Parallel Multiprocessor", Tech. Rep. TR-911, Dept. Comp. Sci, *University of Maryland, College Park, MD, 1980.*

b59 bibitem7 D. Rine, "Computer Science and Multiple-Valued Logic. Theory and Applications", North-Holland, *1984.*

[334]

[335] B. Reusch, and L. Detering, "On the Generation of Prime Implicants", Annales Societatis Mathematicae Polonae, *Series IV: Fundamenta Informaticae II, pp. 167-186, 1979.*

[336] J.A. Robinson, "A Machine Oriented Logic Based on the Resolution Principle", J. of the Association for Computing Machinery, *Vol. 12., No. 1., pp. 23-41, January 1965.*

[337] P. Robinson, "The SUM: an AI Coprocessor", BYTE, *pp. 169-180, June 1985.*

[338] V.I. Rodin, "An Electronic Analyzer for Contact Networks", Avt. i Telem., *Vol. 18, No. 5., pp. 437-443, 1957.*

[339] *V.N. Roginskii, "A Machine for the Design of Relay Networks",* Vestnik Akad. Nauk S.S.S.R., *No. 10, pp. 71-73, 1958.*

[340] *T.D. Ross, M.J. Noviskey, T.N. Taylor, and D.A. Gadd, "Pattern Theory: An Engineering Paradigm for Algorithm Design,"* Final Technical Report WL-TR-91-1060, *Wright Laboratories, USAF, WL/AART/WPAFB, OH 45433-6543, August 1991.*

[341] *P. Roth,* "Computer Logic, Testing and Verification", *Rockville, MD: Computer Science, 1980.*

[342] *S. Rudeanu, "Boolean Equations and their Applications to the Study of Bridge Circuits I",* Bull. Math. Soc. Math. Phys. R. P. Roumaine, *Vol. 3., pp. 445-473, 1959.*

[343] *S. Rudeanu, "Boolean Functions and Equations",* North-Holland Publ. Co. & American Elsevier, *New York, 1974.*

[344] *R.L. Rudell, and A.L. Sangiovanni-Vincentelli: "ESPRESSO-MV: algorithms for multiple-valued logic minimization,* Proc. IEEE Custom Integrated Circuits Conf., *1985.*

[345] *R. Rudell,* "Multiple-Valued Logic Minimization for PLA Synthesis", *M.S. Report, June 5, 1986. University of California, Berkeley California 94720.*

[346] *R.L. Rudell, and A.L. Sangiovanni-Vincentelli, "Multiple-valued minimization for PLA optimization,"* Proc. Intern. Symp. on Multiple-Valued Logic, *pp. 198-208, May 26-28, Boston, MA, 1987.*

[347] *B. Russell,* "An Outline of Philosophy", *Allen and Unwin, London, 1927.*

[348] *R.A. Rutenbar, T.N. Mudge, and D.E. Atkins, "A Class of Cellular Architectures to Support Physical Design Automation",* IEEE TCAD, *Vol CAD-3, No.4., October 1984, pp.264-278.*

SSSSSSSSSSSSS

[349] *A. Sarabi, and M.A. Perkowski, "Fast Exact and Quasi-Minimal Minimization of Highly Testable Fixed-Polarity AND/EXOR Canonical Networks",* Proc. DAC'92.

[350] *T. Sasao, "An application of multiple-valued logic to a design of Programmable Logic Arrays",* Proc. 8th Intern. Symp. on Multiple-Valued Logic (ISMVL), *1978.*

[351] T. Sasao, and H. Terada, "Multiple-Valued Logic and the Design of Programmable Logic Arrays with Decoders", Proc. of 9th International Symposium on Multiple-Valued Logic, *Bath, England, pp. 27 - 37, 1979.*

[352] T. Sasao, "Multiple-valued decomposition of generalized boolean functions and the complexity of programmable logic arrays," IEEE Trans. Comput., *Vol. C-30, pp. 635-643, Sept. 1981.*

b60

[353]

[354] T. Sasao, "Input variable assignment and output phase optimization of PLA's," IEEE Trans. Comput., *Vol. C-33, pp. 879 - 984, Oct. 1984.*

b61

[355]

[356] T. Sasao, "HART: A Hardware for Logic Minimization and Verification", Proc. ICCD'85, *pp. 713-718, Oct. 7-10, 1985.*

[357] T. Sasao, "MACDAS: Multi-level AND-OR circuit synthesis using two-variable function generators", 23-rd Design Automation Conference, *Las Vegas, pp. 86-93, June 1986.*

[358] T. Sasao, and P. Besslich, "On the Complexity of MOD-2 Sum PLA", Institute of Electronics and Communication Engineers of Japan, *FTS86-17, pp. 1-8, Nov. 17, 1986.*

[359] T. Sasao, "EXMIN: A Simplification Algorithm for Exclusive-OR-Sum-of-Products Expressions for Multiple-Valued Input Two-Valued Output Functions", Proc. of 20th Int. Symp. on Multiple-Valued Logic, *pp. 128-135, May 1990.*

[360] J.M. Saul, "An Improved Algorithm for the Minimization of Mixed Polarity Reed-Muller Representations", Proc. ICCD'90, *pp. 372-375, Sept. 1990.*

[361] I. Schaefer, "An Effective Cube Comparison Method for Discrete Spectral Transformations of Logic Functions", M. Sc., Thesis, *May 1990.*

[362] I. Schaefer, B.J. Falkowski, and M.A. Perkowski, "An Efficient Computer Algorithm for the Calculation of Walsh Transform for Completely and Incompletely Specified Multiple-Valued Input Binary Functions", Proc. of the 34th IEEE Midwest Symposium on Circuits and Systems, *Monterey, CA, May 1991. PAGES?*

[363] I. Schaefer, B.J. Falkowski, and M.A. Perkowski, "A Fast Computer Implementation of Adding and Arithmetic Multi-Polarity Transforms for Logic Design", Proc. of the 34th IEEE Midwest Symposium on Circuits and Systems, *Monterey, CA, May 1991. PAGES?*

[364] I. Schaefer, B.J. Falkowski, and M.A. Perkowski, "A Fast Computer Implementation of Adding and Arithmetic Multi-Polarity Transforms for Logic Design", International Journal of Electronics. *Pages?*

[365] I. Schaefer, and M.A. Perkowski, "Multiple-Valued Input Generalized Reed-Muller Forms", Proc. ISMVL'91, *Victoria, British Columbia, May 1991. PAGES?*

[366] I. Schaefer, and M.A. Perkowski, "Multiple-Valued Input Generalized Reed-Muller Forms", IEE Journal, *???.*

[367] I. Schaefer, and M.A. Perkowski, "An Algorithm to Find the Minimal Multiple-Valued Input Kronecker Reed-Muller Form", ???

[368] I. Schaefer, and M.A. Perkowski, "Synthesis of Multi-Level Multiplexer Circuits for Incompletely Specified Multi-Output Boolean Functions with Mapping Multiplexer Based FPGAs", IEEE Transactions on Computer Aided Design, *????*

[369] J.T. Schwartz, "Ultracomputers", ACM Transactions on Programming Languages and Systems, *2, 4, pp. 484-521, 1980.*

[370] J.T. Schwartz, "A Taxonomic Table of Parallel Computers, Based on 55 Designs", Courant Institute, *New York University, 1983.*

[371] S. Shankar, "A Hierarchical Associative Memory Architecture for Logic Programming Unification" Logic Programming, *Proc. of the 5th Intern. Conf. and Symp. Vol. 2., pp. 1428-1447, MIT Press, 1988.*

[372] Y. Shotobatake, and H. Aiso, "A Unification Processor Based on Uniformly Structured Cellular Hardware", 13th Intern. Symp. on Computer Architecture, *pp. 140-148, 1986.*

[373] C.E. Shannon, and E.F. Moore, "Machine Aid for Switching Circuit Design", Proc. IRE, *Vol. 41., pp. 1348-1351, Oct. 1953.*

[374] *D.E. Shaw, "The NON-VON Supercomputer,* Department of Computer Science, *Columbia University, 1982.*

b64

[375] *D. Shumake, "The MOS Boolean Analyzer",* M.Sc. Thesis, *UCLA, 1971.*

[376] *H.J. Siegel, L.J. Siegel, F.C. Kemmerer, P.T. Mueller, Jr., H.E. Smalley Jr., and S.D. Smith, "PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition",* IEEE Trans. on Computers, *C-30, 12, pp. 934-946, 1981.*

[377] *J. Siekmann, "Unification Theory",* J. of Symbolic Computation, *Vol. 7, pp. 207-274, 1989.*

[378] *J.R. Slagle, et al, "A New Algorithm for Generating Prime Implicants",* IEEE Trans. on Computers, *Vol. C-19, pp. 304-310, 1970.*

[379] *R. Spickelmier, (ed.), "OCT Tools Distribution 2.1.",* Electronics Research Laboratory, *University of California, Berkeley, March 25, 1988.*

[380] *A. Srinivasan, T. Kam, S. Malik, and R.K. Brayton, "Algorithms for Discrete Function Manipulation",* IEEE Proc. of IEEE Int. Conf. on CAD, *ICCAD'90, pp. 92-95, 1990.*

b65

[381] *A. Stern, "Matrix Logic",* North-Holland, *1988.*

[382] *Stolfo, S.J., and D.E. Shaw, "DADO: A Tree-Structured Machine Architecture for Production Systems",* Department of Computer Science, *Columbia University, 1982.*

[383] *C.D. Stormaon, M.R. Brule, J.V. Oldfield, and J.C. Ribeiro, "An Architecture Based on Content-Addressable Memory for the Rapid Execution of Prolog",* Logic Programming: Proc. of the 5th Intern. Conference and Symposium, *ed. Kenneth A. Bowen, Vol. 2., pp. 1449-1473, MIT Press, 1988.*

[384] *M.D. Strugala, D. Tavangarian, K. Waldschmidt, and G. Roll, "An Associative Processor as a Design Rule Check Accelerator",* Proc. of VLSI and Computers First Intern. Conf. on Computer Technology, Systems and Applications, *pp. 426-431, IEEE, Hamburg, May 11-15, 1987.*

b67

[385]

[386] Su, S.Y.H., and P.T. Cheung, "Computer Minimization of Multivalued Switching Functions", IEEE Trans. on Comp., *Vol. C-21, No. 9, p. 995, September 1972.*

[387] P. Suppes, "Introduction to Logic", *New York: Van Nostrand Reinhold, 1957.*

b66

[388] A. Svoboda, "Boolean Analyzer", Proc. Information Processing 68, *Amsterdam, North-Holland, 1969, pp. 824-830.*

b68

[389] A. Svoboda, "Parallel Processing in Boolean Algebra", IEEE TC, *Vol. C-22, No. 9, pp. 848-851, Sept 1973.*

[390] A. Svoboda, and D.E. White, "Advanced Logical Circuit Design Techniques", Garland STPM Press, *New York, 1979.*

TTTTTTTTTTT

[391] M.A. Tapia, J.H. Tucker and A.W. Bennett, "Boolean Integration", Proc. IEEE Southeast-Con, *Clemson, SC, April 1976.*

[392] M.A. Tapia, "Boolean Differentiation and Integration using Karnaugh Map", Proc. IEEE Southeast-Con., *1977.*

[393] M.A. Tapia, "Application of Boolean Calculus to Digital System Design", Proc. IEEE Southeast-Conf., *Nashville, Tenn., 14-16 April 1980.*

[394] M.A. Tapia, and J.H. Tucker, "Complete Solution of Boolean Equations", IEEE Trans. on Comp. *Vol. C-29, No. 7., pp. 662-665, July 1980.*

[395] M.A. Tapia, "Boolean Integral Calculus for Digital Systems", IEEE Trans. on Comp. *Vol. C-34, No. 1., pp. 78-81, Jan. 1985.*

[396] D.K. Taylor, "Analyzing Relational Databases using Propositional Logic", M.S. Thesis, *Department of Electrical Engineering, University of Kentucky, December, 1981.*

[397] A. Thayse, "Boolean Differential Calculus", Philips Res. Reports, *Vol. 26., pp. 229-246, 1971.*

[398] A. Thayse, and M. Davio, "Boolean Differential Calculus and its Applications in Switching Theory", IEEE Trans. on Computers, *Vol. C-22., pp. 409-420, 1973.*

[399] B.L. Timofeev, "A Machine for Minimizing Relay Circuits in the Disjunctive Normal Form Category", Avt. i Vyc. Tekhn., *Vol. 2., 1959.*

[400] B.L. Timofeev, "Mechanization of the Process of Determining Minimal Variants of Relay Systems in Disjunctive Normal Forms", Voprosy Sbornik Avtomat. Upravlenie, *pp. 345-353, 1960.*

[401] B.L. Timofeev, "The Minimization of Logical Functions as Disjunctive Normal Forms, with the Aid of a Specialized Computer", Dokl. Rel. Ustr. Kon. Avt., *1962.*

[402] P. Tison, "Generalization of Consensus Theory and Application to the Minimization of Boolean Functions", IEEE Trans. Electronic Computers, *Vol. EC-16, pp. 446-456, 1967.*

[403] D. Tavangarian, "Flag-Algebra: A New Concept for the Realisation of Fully Parallel Associative Architectures", IEE Proc., *Vol. 136, Pt. E, No. 5, pp. 357-365, September 1989.*

b69

[404] T. Toffoli, and M. Margolus, "Cellular Automata Machines", MIT Press, *Cambridge, Mass., 1987.*

[405] P.C. Traleaven, and G.F. Moll, "A Multi-Processor Reduction Machine for User-Defined Reduction Languages", Seventh Annual Symposium on Computer Architecture, La Baule, France, pp. 121-130, 1980.

[406] Ph.C. Treleaven, D.R. Brownbridge, and R. P. Hopkins, "Data-Driven and Demand-Driven Computer Architecture", Computing Surveys, *Vol. 14, No. 1, pp. 93-142, March 1982.*

[407] Ph.C. Treleaven, "The New Generation of Computer Architecture", SIGARCH Newsletter, *Vol. 11, No. 3, pp. 402-416, 1983.*

[408] T.T. Tsukanov, "A Matrix Analyzer of Relay-Contact Networks", Vses. Mos. Sov. Teorii Ust. Rel. Deist., *p. 72, 1957.*

[409] T.T. Tsukanov, "Problems in Mechanization of the Analysis of Relay-Contact Circuits", Trudy Tomsk. Elekt.Inst. Inzhen. Zheleznykh Dorog, *Vol. 23, pp. 129-149, 1957.*

[410] T.T. Tsukanov, "Some Instances of Application of the Matrix Analyzer" Trudy Tomsk. Elekt.Inst. Inzhen. Zheleznykh Dorog, *Vol. 23, pp. 150-116, 1957.*

[411] S. Turing, "Alan M. Turing", *Cambridge, U.K.: W. Heffer & Sons, 1959.*

UUUUUUUUUUUUUU

[412] T. Uehara, and N. Kawato, "Logic Circuit Synthesis using Prolog", New Generation Computing, *Vol. 1., No. 2., 1983.*

[413] Ullman, J.D., "Computational Aspects of VLSI", Computer Science Press, *1984.*

b70

[414] M.E. Ulug, and B.A. Bowen, "A Unified Theory of the Algebraic Topological Methods for the Synthesis of Switching Systems", IEEE TC, *pp. 255-267, March 1974.*

b71

[415] M.E. Ulug, "VLSI Knowledge Representation Using Predicate Logic and Cubical Algebra", Proc. IEEE Intern. Conference on Computers and Communications, *Arizona, 1985, pp. 292-297.*

[416] M.E. Ulug, "Application of Cubical Array Operators to a Relational Database", Proc. of the Minnowbrook Workshop, *July 23-26, 1985, Blue Mountain Lake, New York.*

[417] M.E. Ulug, "A Real-Time AI System for Military Communications", Proc. of the Third IEEE Conference on Artificial Intelligence Applications, *February 1987, Orlando, Florida.*

[418] Usher, *(NAME??)* "Mechanical Inventions", *(EDITOR YEAR?)*

VVVVVVVVVVV

[419] G.Y. Vichniac, Boolean derivatives on cellular automata, Physica D, *45, pp. 63-74, 1990.*

b34

[420] Virtual ASIC, "Concept Silicon partitions your design onto multiple FPGAs", INCA company materials, *1992.*

WWWWWWWWWWWWW

b72

[421]

[422] B.W. Wah, Y.W.E. Ma, "MANIP - A Multicomputer Architecture for Solving Combinatorial Extremum-Search Problems", IEEE TC, *Vol. C-33, No. 5, pp. 377 - 390, May 1984.*

[423] H. Wang, "Towards Mechanical Mathematics", IBM J. for Research and Development, *4, 1960, pp. 2-22.*

[424] J. Weissman, "Boolean Algebra, Map Coloring, and Interconnections", Amer. Math. Monthly, Vol. 69., pp. 606-613, 1962.

[425] I. Williams, "Using FPGAs to Prototype New Computer Architecures", in FPGAs, *pp. 373-382.*

[426] T.A. Welch, "An Investigation of Descriptor Oriented Architecture", Computer Architecture News, *Vol. 4, No. 4, pp. 141-146, January 1976.*

[427] J. Werner, " Japanese trends in system design. Part 2. Advanced Computer-Aided Design Tools". VLSI Design, July 1984, pp.36-42.

[428] N.S. Woo, "A Hardware Unification Unit: Design and Analysis", 12th International Symposium on Computer Architecture, *pp. 198-205, 1985.*

[429] W.S. Wojciechowski, and A.S. Wojcik, "Multiple-Valued Logic Design by Theorem Proving", Proc. 9-th Intl. Symp. on Multiple-Valued Logic, *Bath, England, 1979, pp. 196-199.*

[430] W.S. Wojciechowski, "Multiple-Valued Combinational Logic Design using Theorem Proving", Dissertation, Ill. Inst. of Tech., *University Microfilms, No. KRA80-2162, May 1980.*

[431] W.S. Wojciechowski, and A.S. Wojcik, "Automated Design of Multiple-Valued Logic Circuits by Automated Theorem Proving Techniques", IEEE Trans. on Computers, *Vol. C-32, pp. 785-798, Sept. 1983.*

[432] S. Wolfram, "Cellular Automata as Models of Complexity", Nature, *311,4, pp. 419-424, 1984.*

[433] S. Wolfram, Theory and Applications of Cellular Automata, World Scientific Press, *Singapore, 1986.*

[434] L. Wos, R. Overbeek, E. Lusk, and J. Boyle, "Automated Reasoning: Introduction and Applications", Prentice-Hall, *Englewood Cliffs, N.J., 1984.*

[435] L-.F. Wu, and M.A. Perkowski, "Minimization of Permuted Reed-Muller Trees for Cellular Logic Programmable Gate Arrays", Proc. of the 2nd Intern. Workshop on Field-Programmable Logic and Applications, *FPL'92, Vienna, Austria, August 31-September 2, 1992, pp. 7/4.1-7/4.4.*

[436] Wu, S.B., and M. T. Liu, "A Cluster Structure as an Interconnection Network for Large Multicomputer Systems", IEEE Trans. on Computers, *C-30, 4., pp. 254-264, 1981.*

XXXXXXXXXXXXXXXxx

[437] XILINX, Inc., "The Programmable Gate Array Data Book", *1991.*

YYYYYYYYYYYYYYYYYY

[438] K. Yamada, and K. Yoshida, "An Application of Boolean Algebra in Practical Situations", Hitotsubashi J. Arts & Sciences, *Vol. 5., pp. 41-57, 1965.*

[439] K. Yamada, H. Nakada, A. Tsutsui, T. Fujii, and N. Ohta, "FPGA Design for Digital Telecommunication Circuits using a High Level Design System", Report.

[440] H. Yasura, T. Tsujimoto, K. Tamaru, "Parallel Exhaustive Search For Several NP-Complete Problems Using Content Addressable Memories", Proc. IEEE Intern. Conference on Circuits and Systems, *ISCAS'88, pp. 333- 336.*

[441] T. Yuba, T. Shimada, Y. Yamaguchi, K. Hiraki, and S. Sakai, "Dataflow Computer Development in Japan", ACM SIGARCH News, *Vol. 18, No. 3, pp. 140-147, Sept. 1990.*

ZZZZZZZZZZZZZZZZZ

[442] A.D. Zakrevskii, "A Machine for the Solution of Logical Problems of Switching Synthesis Type", Dokl. Rel. Ustr. Kon. Avt., *1962.*

[443] A.D. Zakrevskii, "A Universal System for the Solution of Problems Similar to Contact Network Synthesis", Vyc. Tekh. Avt. Teorii. Inf., *No. 42., pp. 9-37, 1963.*

[444] H. Zemanek, "Central European Prehistory of Computing", in Metropolis et al. "History of Computing". *(PUBLISHER??).*

[445] I.I. Zhegalkin, "On the Calculation of Propositions in Symbolic Logic", (in Russian), Math. Sbornik, *Vol. 34., pp. 9-28, 1927.*

[446] B. Zupan, and M. Bohanec, "Experimental Evaluation of Three Partition Selection Criteria for Decision Table Decomposition," Research Report, *October 19, 1996, Department of Intelligent Systems, Jozef Stefan Institute, 1000 Ljubljana, Slovenia.*

[447] B. Zupan, and M. Bohanec, "Learning Concept Hierarchies from Examples by Function Decomposition," Technical Report, Department of Intelligent Systems, *Jozef Stefan Institute, Ljubljana, Slovenia, September 1996.*

[448] Zwicky (Name??), "INVENTION ??""

Chapter 7

APPENDIX 1: XACT DEVELOPMENT SYSTEM TUTORIAL

The tutorial presented in this appendix is an introduction to the design flow and development tools for configuring Xilinx LCAs. The tutorial consists of a simple design for a seven-segment decoder that will be downloaded to the provided demonstration board supplied by Xilinx. It encompasses the following:

1. The viewing of the created design in the FutureNet DASH schematic capture package.

2. The conversion of the drawing files to Xilinx netlist files.

3. The mapping of the netlist files to logic cell array files.

4. The viewing of the LCA design and layout.

5. The downloading and operation of the design in the Xilinx LCA.

The Xilinx tools reside on the 486 PC in Dr. Perkowski's lab downstairs in the front room of the lab. The FutureNet DASH and Xilinx manuals also reside in the front room and Xilinx download cable and Demo board are stored in the back room.

7.1 CREATING THE DESIGN

The design used in this tutorial was created using the supplied schematic capture application FutureNet DASH [?]. The dashdice design is a two-level schematic, with the top-level design calling a lower-level schematic design which holds the operation of the included macro.

1. To view the Dashdice example, you must first enter the FutureNet DASH Schematic Capture tools. To invoke DASH, type:

   ```
   C:>\\ "fn 2"
   ```

2. This automatically calls the macro library for the XC2000 family of LCA's. This family is selected since the demonstration board was designed for use with a Xilinx 2064 LCA device. The schematic capture package will be invoked.

3. Load the XC2000 macros.

4. Load the current profile, then give yourself control of the program.

   The mouse is setup as such:

   ```
   left button = select, middle button = execute, right button = menu
   ```

5. From the command prompt, type 'dir' to view the current directory that you are located, which should be

   ```
   C:>\\XACT\\DESIGNS.
   ```

   You will need to change directories to

   ```
   C:>\\XACT1_30\\TUTORIAL.
   ```

   This can be done by typing

   ```
   'cd \\XACT1_30\\TUTORIAL"
   ```

   at the command prompt. Then type

```
'dir'
```

to view the contents of this directory. There should be at least two files: dashdice.dwg and dice7seg.dwg for this tutorial to work properly.

6. If you are currently located in the menu of DASH move to the far right of the menu and select 'Load'. The drawing screen will then appear with a command line at the bottom containing the word load.

7. Type in 'dashdice' and press return. The drawing will be loaded into the full screen for you to view.

8. Move the cursor around the screen observing the dashed box that follows the cursor. This is the zoom window. Move the cursor so the box is encompassing the area in which you wish to zoom in upon. To zoom in and out so details of the drawing can be better seen, use the 'Pageup' and 'Pagedown' keys from the keyboard. Features that you might wish to look at are the input/output buffers and pins, and how they are represented in symbolic terms within the drawing. The labels given to drawing are called attributes. The attributes define the labels given to certain macros such as I/O pins, logic gates, and the signals running between the two. As the mouse is moved across the attributes, the values are displayed to the right of the viewing area giving greater insight to their definition. These attributes are quite important to the final LCA design through their ability to direct outputs of the design to the actual device pins, and give a macro in a design an actual call to an additional drawing design.

9. Now zoom out to the full-size drawing picture and place the zoom window around the large rectangular box in the lower right portion of the drawing (not the title box). Zoom in to that area and notice that the box at the top contains the title 'dice7seg'. This box actually represents a lower level schematic named 'dice7seg.dwg' that is implemented as a user-created macro in the top-level design 'dashdice.dwg'. To view this drawing, go back to the menu by pressing the right mouse button. Move to the far right side of the Menu again and select 'Load'. The viewing screen will once again appear with the command line at the bottom.

10. Type 'dice7seg' and hit ¡RETURN¿. The drawing of the macro dice7seg will be loaded for you to view.

11. If you wish to bypass the Menu portion of this process, just type 'load dice7seg' and it will do the same job as selecting load from the Menu.

This section was meant to let a user see the schematic description of the design in which you will soon be downloading to the device. No changes should have been made at this stage of the tutorial, otherwise the final LCA design may no function correctly. You now want to quit out of the schematic capture package by one of two ways:

1. Go back in the menu and select 'quit'.

2. Type 'quit' or 'q' from where you currently are located.

3. You should now be back to the DOS prompt, and should be in the directory:

   ```
   C:>\\XACT1_30\\TUTORIAL>
   ```

   This is the default directory where DASH places all of its design files. It is necessary to be located in this directory for now, since you will be converting the design files to Xilinx netlist files. Please refer to the FutureNet DASH manual Vol. I for an overview of the available operations in the schematic capture tools. Also, refer to Vol. II of the DASH manuals for definitions of each command and the user selected options stored in the application's profiles.

7.2 CONVERTING THE DRAWING FILES

1. The first step that a user will make is to prepare the drawings that you have just viewed for post-processing before conversion into a pinlist file. Follow Figure 2 in Chapter II to see the basic design flow. At the DOS prompt, type:

   ```
   C:\\XACT1_30\\TUTORIAL> dcm dashdice.dwg
   ```

   then,

```
C:\\XACT1_30\\TUTORIAL> dcm dice7seg.dwg
```

This will create a total of four files: dashdice.der, dice7seg.der, dashdice.dcm, and dice7seg.dcm. The two .der files are a record of the conversion of the drawing files to .dcm files and are not needed for further processing. The two files that we are concerned with are the .dcm files. DCM is a pre-processor program used in the translation from DASH-LCA schematics to an XNF file. These pre-processor files are the first step in converting the drawing files to pinlist files. The pinlist files contain a list of the connections within and electronic design, consisting of the names of the pins of all symbols in the entire design, and the names of the signals to which each pin connects. To convert the .dcm files to .pin files, type:

```
C:\\XACT1_30\\TUTORIAL> pinc dashdice.dcm
```

then,

```
C:\\XACT1_30\\TUTORIAL> pinc dice7seg.dcm
```

2. This will create the two files: dashdice.pin and dice7seg.pin. PINC is a pin-list generator program used to translate the pre-processed dcm files to a FutureNet compatible PIN file. For a detailed explanation of the PINC command, refer to Vol. II of the Xilinx XACT Programmable Gate Array manual in Chapter 2.2.

3. The next step is to convert the .pin files into Xilinx Netlist Files (XNF) with pin2xnf. The netlist file is a common Xilinx format for logic input to the development system, regardless of the source (i.e. schematic, boolean equations, state machine language).

4. To make the conversion, type:

```
C:\\XACT1_30\\TUTORIAL> pin2xnf -P 2064PC68-50 dashdice.pin
```

5. The -P option allows the user to select the target Xilinx device that will be used (in this case the XC2064PC). The trailing -50 is the speed grade of the device. The trailing

filename is the pinlist file that you just created. As you can observe, you only had to convert the top level design to a netlist file. Since the design dice7seg is included in dashdice, the XNF conversion process searches the top level pinlist file (dashdice.pin) and automatically converts all files contained within it (dice7seg.pin).

7.3 CREATING THE LOGIC CELL ARRAY FILE

1. Now that you have converted the entire design to the gate-level netlist file, you must now convert it to a Configurable Logic Block and Input/Output Block level. This is called a Logic Cell Array (LCA) file. This file will later on be placed and routed into the targeted device. To make this conversion, type:

   ```
   C:\\XACT1_30\\TUTORIAL> xnfcvt dashdice.xnf dash.xnf
   ```

   This command provides the conversion from older XNF files to the most recent version. This is needed since the pin2xnf conversion is from the older software. Refer to the Xilinx User Guide and Tutorials manual for the best explanation of the xnfcvt command. The XNF filename was changed, so we now have two versions of the same XNF file.

2. At this point the user needs to partition the design logic into CLB and IOB resources with the xnfmap program. To map the dashdice design, type:

   ```
   C:\\XACT1_30\\TUTORIAL> xnfmap -p 2064pc68-50 dash.xnf
   ```

   When xnfmap has been executed, two files should have been created. The file dash.crf is a cross-reference report showing how xnfmap partitioned the logic. The file dash.map contains the partitioned logic from the XNF file. The dash.map file must then be converted to an LCA file.

3. To convert the partitioned logic into an LCA file, type:

   ```
   C:\\XACT1_30\\TUTORIAL> map2lca -p 2064pc68-50 dash
   ```

4. After the execution of the map2lca program, three files should have been created. The first file, dash.aka, lists the signal prefixes to let the user know which functional block the signal comes from. The file dash.lca is an LCA design file that has yet to be placed or routed. The dash.scp file is a schematic constraints file containing information that assists APR in placement and routing, such as pin and block locations, and net flags from the schematic files. Chapter 5.4 of Vol. II of the Xilinx XACT Programmable Gate Array Development System manual gives a detailed explanation of the map commands.

5. The final step is to place and route your LCA file into the target device. The Automatic Place and Route procedure (APR), reads an LCA file, generates a new block placement, routes the nets of the design, and writes the result to another LCA file.

   In APR, there are four steps:

   1) Partitioning phase where logic is partitioned into pieces that can be implemented with LCA logic blocks (i.e. CLB's, IOB's, etc...);

   2) Placement phase where blocks are assigned to a location with the LCA;

   3) Routing phase where the signals connecting the blocks are routed using the routing resources of the LCA;

   4) Programming phase where the design information is converted into a bit stream which is then loaded into an LCA.

   The number of possible block placements is the matrix size (# of CLB's). For the XC2064, this is greater than 10^{89} possible placements, not including I/O blocks.

6. A "simulated annealing" algorithm is used to determine the optimal block placement. Annealing refers to the crystallization process in which a metal is melted and then slowly cooled back to freezing in order to form highly-ordered crystals with few defects. Simulated annealing is an algorithm for finding good solutions to complicated optimization problems in a manner analogous to the physical crystallization

of a metal. During each iteration of the simulated annealing placement algorithm, two or more blocks are exchanged at random, and the "routability" of the resulting placement is calculated [?]. This routability is expressed in terms of a "routing score", calculated using a formula that includes the length of the routes, the net weighting, and the number of available routing channels as factors.

The simulated annealing placement algorithm has two phases: the annealing phase and the quenching phase. In the annealing phase, a new placement that results in better routability is always accepted. However, if the new placement is worse, there is still a probability that that it will be accepted, depending on how much worse it is and the current "temperature". The worse the placement, the less likely it will be accepted, although the higher the "temperature", the greater the chance of its acceptance. As the algorithm proceeds, the design is slowly "cooled" by lowering the temperature. As the design cools, blocks move less freely and tend to settle into place. Once the design has sufficiently cooled, the quenching phase is entered. At this stage, only placements of better optimization are accepted.

By default, APR automatically calculates and assigns a starting temperature sufficient to "melt" the design, or in other words, a temperature high enough to allow total scrambling of the initial placement. The algorithm then determines the rate at which the temperature is lowered and when the design has cooled enough to allow quenching. All of this information can be seen as its happening upon invoking the APR program.

7. To place and route your LCA design, type:

```
C:\\XACT1_30\\TUTORIAL> apr dash.lca dashapr.lca
```

This step will take several minutes to place and route this design. Larger designs such as the design for the Iterative Logic Unit (ILU) of the Cube Calculus Machine 2.0 (CCM2) may take up to several hours to do the APR step. As you can see, the syntax of the APR statement has the unrouted LCA file, and the placed and routed LCA file which will be distinguished as dashapr.lca. Refer to Chapter 3 of the Xilinx XACT Programmable Gate Array Development System Vol. II manual.

7.4 VIEWING THE LOGIC CELL ARRAY FILE

1. You now have a complete logic cell array design that is placed and routed. To view your placed and routed design, using the XACT Design Editor by typing at the prompt:

   ```
   C:\\XACT\\DESIGNS> xde
   ```

2. This brings you into the XACT development system, allowing you to view, change and download your design to the device. Move the cursor to the Designs menu and select Design. Another menu will pop up and allow you to choose the design which you wish to work with. If you followed the exact syntax that I used for the APR step, the design you want to work with is dashapr.lca. Select this file, and wait.

3. Once the file is loaded and ready, move the cursor to the Programs Menu and select Editlca. The editing screen will appear, and you will be able to view the layout of the dashdice design. Move the cursor onto the layout area and click down on the left mouse bottom and drag the mouse around. This allows you to scan across the layout while being able to see a global picture of the layout in the lower right corner of the screen.

4. Now, move to where you have a good view of the routing within your local window and let up on the mouse. The global window disappears and you will be able to see the routing from CLB's through switching matrices to other CLB's and I/O Blocks. Move the cursor and point to a pin on any CLB or I/O block, and watch the information at the bottom of the screen. It will tell you the current pin being pointed at, what it corresponds to in your design layout, and its net connections.

5. Once you have looked around here, move the cursor to the Misc Menu and select Exit. You will now be out at the Main Menu of XACT and ready for downloading the dashapr.lca design to the demo board.

7.5 DOWNLOADING THE LOGIC CELL ARRAY FILE

1. First, install an XC2064 in the demo board's LCA socket and connect the power source to the jumper J3 (pin 2-GND, pin 1-Vcc).

2. Next install the download cable. One end is connected to the parallel port of the computer which has the Xilinx key connected to it. Fit the connector to the key and the other end to the demo board's J1 jumper. Connection to J1 is obvious due to the keyed pin fitting. When the demo board has been properly prepared, invoke Makebits from the Programs Menu to access the bitstream and downloading commands.

3. If prompted, select the design file you wish to download (dashapr.lca). Once the Makebits screen appears, use the Misc Menu's Port command to specify which port the download cable is connected to. For our setup, select LPT1. If the cable is connected properly, a message saying that Port LPT1 is initiated will appear at the bottom of the screen.

4. Next, invoke the Config Menu's Makebits command, creating a bitstream file that is present in memory and can be downloaded into an LCA.

5. At this junction, select the Tie option which will internally tie all floating I/O pins low.

6. Then select Done, and the XACT system will create the bitstream file dashapr.bit for downloading. Also created is dashapr.mbo, a record of the Makebits options that were selected in the process of creating dashapr.bit.

7. Next, invoke the Writebits command in Config Menu to save the bitstream file to disk. Hit ¡CR¿ for the default filename, which should be dashapr.bit.

8. Then, use the Download Menu's Download command to transfer the bitstream into the LCA on the demo board. If prompted "Reset the LCA", then press the demo board's Reset switch once. The bottom of the screen should show that the Reset pin has gone low.

9. Then hit ¡CR¿ to start the downloading process. Downloading should take 1 to 20 seconds depending upon the speed of the PC.

10. If the download is successful, the message "Done Signal Went High" will appear at the bottom of the screen. If downloading fails, first check the cable connection between the demo board and the PC for proper connection. If download is successful but the demo board doesn't function correctly, check the user switch settings SW1 on the demo board, noting the correct positions of ON/OFF switches.

For this design, a 3-bit Johnson counter is decoded into a seven segment display of 1-6. For proper operation, the seven segment display on the demo board should be rapidly counting from 1-6 until the clock is inhibited by setting SW1 to 0/OFF, effectively stopping the counter at a random number.

Chapter 8

APPENDIX 2: VHDL CODE OF CCM2.

```
%%%%%%%%%%%
% file clock.vhdl_6
%%%%%%%%%%%



--*****************************************************
-- clock genertor
--
-- During reset drive clk to  '1' until reset
-- is deasserted. This allows synchronization of bus
-- unit interface and bus clock to the cpu clock.
--
-- cycle time is arbitrarily set to 100ns.
--*****************************************************

ENTITY clock IS
   PORT (reset: IN BIT; clk: INOUT BIT);
END clock;

ARCHITECTURE clock0 OF clock IS
BEGIN
clk <= '1' WHEN (reset = '1')
```

```
          ELSE NOT clk AFTER 50ns;
END clock0;

%%%%%%%%%%%%%%%%%%%%
% file flip_flop.vhdl_12
%%%%%%%%%%%%%%%%%%%%%


--*****************************************************
-- Edge triggered, D flip-flop, with synchronous reset
--*****************************************************

ENTITY flip_flop IS
   PORT (clk, enable, reset, din: IN BIT; dout: OUT BIT);
END flip_flop;

ARCHITECTURE msff OF flip_flop IS
   SIGNAL temp0, temp1, temp2: BIT;
   SIGNAL slct: BIT_VECTOR (1 DOWNTO 0);
   BEGIN
      slct  <= (enable & reset);
      WITH slct SELECT
         temp0 <= '0' WHEN "01",
                  din WHEN "10",
                  temp2 WHEN OTHERS;
      temp1 <= temp0 WHEN (clk = '0')
               ELSE temp1;
      temp2 <= temp1 WHEN (clk = '1')
               ELSE temp2;
      dout  <= temp2;
   END msff;

%%%%%%%%%%%%
% file mux4to1.vhdl_7
%%%%%%%%%%%%

--********************************************************
-- 4 to 1 mux, used in this design to implement any one of
-- the 16 possible two-input binary functions.
-- The 4 bit input is the truth values of the binary function,
-- and the two select lines are the two
-- input bits to the binary function.
```

```
--***********************************************************

ENTITY mux4to1 IS
   PORT (in0, in1, in2, in3, sel0, sel1: IN BIT;
         out0: OUT BIT);
END mux4to1;

ARCHITECTURE zmux0 OF mux4to1 IS

   SIGNAL sel: BIT_VECTOR (1 DOWNTO 0);

   BEGIN
      sel <= (sel1 & sel0);
      WITH sel SELECT
         out0 <= in0 WHEN "00",
                 in1 WHEN "01",
                 in2 WHEN "10",
                 in3 WHEN "11";
END zmux0;




%%%%%%%%%%%
% file counter.vhdl_21
%%%%%%%%%%


--**************************************************
-- 5 bit counter, used to count the number of output
-- cubes of a sequential cube operation.
--**************************************************


USE work.all;
ENTITY counter IS
 PORT (clk,        -- Clock from clock generator

      count,      -- From control unit state machine.
                  -- Enables a count by 1.

      ld_ir,      -- From bus interface units. Loads
```

```
                    -- instruction register, and clears
                    -- the counter for the next operation.

      reset,        -- External CPU reset.

      stall:        -- Register file stalls the CPU when it is full.

       IN BIT;

      count_out:  -- Counter output to the bus unit.
                    -- Provides status on the number of output
                    -- cubes from a particular instruction.

               OUT BIT_VECTOR (4 DOWNTO 0));
END counter;


ARCHITECTURE counter0 OF counter IS

   SIGNAL temp, value, carry: BIT_VECTOR (4 DOWNTO 0);
   SIGNAL rst_count, count_en: BIT;

   COMPONENT flip_flop
             PORT (clk, enable, reset, din: IN BIT;
                   dout: OUT BIT);
   END COMPONENT;

   BEGIN

      rst_count <= ld_ir OR reset;
               -- Reset the counter to 0 on the
               -- start of new instruction execution
               -- indicated by ld_ir going active high,
               -- or when cpu reset is asserted.


      -- carry chain for the counter
      --
      carry(0)  <= '1';
      carry(1)  <= value(0) AND carry(0);
      carry(2)  <= value(1) AND carry(1);
      carry(3)  <= value(2) AND carry(2);
      carry(4)  <= value(3) AND carry(3);

      temp      <= value XOR carry;
      count_en  <= count AND NOT stall;
               -- Enable the counter when count
               -- is asserted by the control unit
               -- state machine, and the register
               -- file is not full. Count is asserted
               -- when a request is made by the CU
               -- state machine, and the ILU responds
               -- that it has not completed all
               -- cube computations (wait_out high).
               -- Stall causes the CU state machine to
               -- spin waiting for a register file
               -- entry to become available, so counting
               -- should also be disabled in such case.

      msff0: flip_flop
             PORT MAP (clk, count_en, rst_count, temp(0), value(0));
      msff1: flip_flop
             PORT MAP (clk, count_en, rst_count, temp(1), value(1));
      msff2: flip_flop
             PORT MAP (clk, count_en, rst_count, temp(2), value(2));
      msff3: flip_flop
             PORT MAP (clk, count_en, rst_count, temp(3), value(3));
      msff4: flip_flop
             PORT MAP (clk, count_en, rst_count, temp(4), value(4));

      count_out <= value;
END counter0;



%%%%%%%%%%%%%%%%%%
% file reg16.vhdl_5
%%%%%%%%%%%%%%%%%%%

--*********************************************************
-- 16 bit register, output triggered on rising clk
--*********************************************************

ENTITY reg16 IS
```

```
   PORT (ld_reg,     -- write enable

         clk:        -- global CPU clock

              IN BIT;

         reg_in:     -- register input bus

              IN BIT_VECTOR (15 DOWNTO 0);

         reg_out:    -- register output bus

              OUT BIT_VECTOR (15 DOWNTO 0));
END reg16;

ARCHITECTURE reg16_0 of reg16 IS

   SIGNAL reg_wr_en: BIT;
   SIGNAL temp1, temp2: BIT_VECTOR (15 DOWNTO 0);

   BEGIN
      reg_wr_en <= ld_reg AND NOT clk;
      temp1     <= reg_in WHEN (reg_wr_en = '1')
                   ELSE temp1;
      temp2     <= temp1 WHEN (clk = '1')
                   ELSE temp2;
      reg_out   <= temp2;
END reg16_0;

%%%%%%%%%%%%%%%%%%
% file reg32.vhdl_4
%%%%%%%%%%%%%%%%%%%



--**************************************************************
-- 32 bit register, output triggered on rising clk
--**************************************************************

ENTITY reg32 IS
   PORT (ld_reg,     -- write enable
        clk:        -- global CPU clock

            IN BIT;

        reg_in:     -- register input bus

            IN BIT_VECTOR (31 DOWNTO 0);

        reg_out:    --- register output bus

            OUT BIT_VECTOR (31 DOWNTO 0));
END reg32;

ARCHITECTURE reg32_0 of reg32 IS

   SIGNAL reg_wr_en: BIT;
   SIGNAL temp1, temp2: BIT_VECTOR (31 DOWNTO 0);

   BEGIN
      reg_wr_en <= ld_reg AND NOT clk;
      temp1     <= reg_in WHEN (reg_wr_en = '1')
                    ELSE temp1;
      temp2     <= temp1 WHEN (clk = '1')
                    ELSE temp2;
      reg_out   <= temp2;
END reg32_0;
```

```
%%%%%%%%%%%%%%%%%%%%%%%%
% file regfile.vhdl_7
%%%%%%%%%%%%%%%%%%%%%%%%

--*****************************************************************
-- This functional block is the output cube queue. A queue has
-- been the design choice for implementing the register file in
-- order to simplify the central control unit state machine, and
-- to allow buffering between the ILU computation, and a bus
-- interface DMA like control. Computation can progress at a rate
-- different from the DMA rate, and the buffer smoothes out
-- variations in speed. The queue stalls the CPU when it is full
-- and cannot accept new results. It signals to the bus interface
-- unit when it is empty, and has not data for the DMA to read.
--
-- This block consists of the following logic:
--
--    1. Six 32-bit entries to hold result cubes, and a valid bit
--       for each entry. The number of the queue entries have been
--       arbitrarily chosen. It should be optimized for best
--       area-performance trade-off when a target main CPU is
--       selected, and a bus interface unit is designed to
--       inetrafce with the main CPU.
--
--    2. A shift carry chain, which detects if there is any non
--       valid entry ahead in the queue. If there is, then all
--       previous entris are shifted one position ahead in the
--       queue in one clock. The queue is therefore is implemented
--       as 33 6-bit shift registers (32 data, and one valid bit),
--       and the shift register control is local.
--
--    3. A head carry chain which detects the head of the queue,
--       i.e. the oldest entry which has valid data. When the
--       bus interface unit reads the queue, the head carry chain
--       automatically controls the read cycle, and enables
--       reading the oldest entry in the queue.
--
--    4. Logic to clear the valid bit associated with the head of
--       the queue when the bus interface unit reads an entry. For
--       best performance, a read and a shift can happen in the
--       same cycle, and the correct valid bit in this case is
--       cleared.
--
--    5. Output cubes from the ILU are always written in entry 0
--       of the queue. The ILU output is therefore the input to
--       the queue shift registers.
--
--    6. Logic which detects full and empty states. When the queue
--       is full, it stalls the CPU, and when it is empty it
--       signals this condition to the bus interface unit to make
--       it aware that there is no valid data yet for it to read.
--*****************************************************************
```

```
USE work.all;

ENTITY regfile IS
   PORT (clk,           -- Global CPU clock

         reset,         -- CPU reset

         wr_en,         -- External input which enables writing a
                        -- new entry in the queue.

         rd_en:         -- External input which enables reading
                        -- the head of the queue.

                IN BIT;

         reg_file_in:   -- Input data to write into the queue

                IN BIT_VECTOR (31 DOWNTO 0);

         empty,         -- Queue is empty

         full:          -- Queue is full

               OUT BIT;

         reg_file_out:  -- Output data from the queue

               OUT BIT_VECTOR (31 DOWNTO 0));
END regfile;

ARCHITECTURE regfile0 OF regfile IS

   SIGNAL head, head_carry, read, write, regvclr,
          regv, shift_carry: BIT_VECTOR (5 DOWNTO 0);
   SIGNAL reg_out0, reg_out1, reg_out2, reg_out3,
          reg_out4, reg_out5: BIT_VECTOR (31 DOWNTO 0);

   COMPONENT reg32
             PORT (ld_reg, clk: IN BIT;
                   reg_in: IN BIT_VECTOR (31 DOWNTO 0);
                   reg_out: OUT BIT_VECTOR (31 DOWNTO 0));
```

```
END COMPONENT;

COMPONENT flip_flop
          port (clk, enable, reset, din: IN BIT;
                dout: OUT BIT);
END COMPONENT;

BEGIN

   -- instantiate the queue's six 32-bit registers
   --
   reg0: reg32
         PORT MAP (write(0), clk, reg_file_in, reg_out0);
   reg1: reg32
         PORT MAP (write(1), clk, reg_out0, reg_out1);
   reg2: reg32
         PORT MAP (write(2), clk, reg_out1, reg_out2);
   reg3: reg32
         PORT MAP (write(3), clk, reg_out2, reg_out3);
   reg4: reg32
         PORT MAP (write(4), clk, reg_out3, reg_out4);
   reg5: reg32
         PORT MAP (write(5), clk, reg_out4, reg_out5);

   -- instantiate the queue's valid bits
   --
   valid0: flip_flop
           PORT MAP (clk, write(0), regvclr(0), wr_en, regv(0));
   valid1: flip_flop
           PORT MAP (clk, write(1), regvclr(1), regv(0), regv(1));
   valid2: flip_flop
           PORT MAP (clk, write(2), regvclr(2), regv(1), regv(2));
   valid3: flip_flop
           PORT MAP (clk, write(3), regvclr(3), regv(2), regv(3));
   valid4: flip_flop
           PORT MAP (clk, write(4), regvclr(4), regv(3), regv(4));
   valid5: flip_flop
           PORT MAP (clk, write(5), regvclr(5), regv(4), regv(5));

   -- Shift carry chain: when one or more entries ahead in the
   -- queue is empty, a shift carry is generated for the entries
   -- before which causes all previou entries in the queue to
-- be shifted by one position, therefore allowing a new cube
-- to be written from the ILU into the tail entry of the
-- queue (entry 0).
--
shift_carry(5) <= (NOT regv(5));
shift_carry(4) <= shift_carry(5) OR (NOT regv(4));
shift_carry(3) <= shift_carry(4) OR (NOT regv(3));
shift_carry(2) <= shift_carry(3) OR (NOT regv(2));
shift_carry(1) <= shift_carry(2) OR (NOT regv(1));
shift_carry(0) <= shift_carry(1) OR (NOT regv(0));
write <= shift_carry;

-- Head carry chain: signal for an entry that there is a
-- valid entry ahead of it in the queue. The head carry chain
-- is used to read the oldest entry in the queue, when the
-- bus interface unit reads an entry from the queue.
--
head_carry(5) <= '0';
head_carry(4) <= head_carry(5) OR regv(5);
head_carry(3) <= head_carry(4) OR regv(4);
head_carry(2) <= head_carry(3) OR regv(3);
head_carry(1) <= head_carry(2) OR regv(2);
head_carry(0) <= head_carry(1) OR regv(1);

head <= regv AND NOT head_carry;

read(0)  <= head(0) AND rd_en;
read(1)  <= head(1) AND rd_en;
read(2)  <= head(2) AND rd_en;
read(3)  <= head(3) AND rd_en;
read(4)  <= head(4) AND rd_en;
read(5)  <= head(5) AND rd_en;

-- Clear the valid bit of the entry read by the bus
-- interface unit. Make sure to clear the valid bit
-- of the next entry in the queue, and not the entry
-- being read when a read and a shift occur simultaneously.
--
regvclr(0) <= reset OR
              (read(0) AND NOT write(0));
regvclr(1) <= reset OR
              (read(1) AND NOT write(1)) OR
              (read(0) AND write(1));
regvclr(2) <= reset OR
              (read(2) AND NOT write(2)) OR
              (read(1) AND write(2));
regvclr(3) <= reset OR
              (read(3) AND NOT write(3)) OR
              (read(2) AND write(3));
regvclr(4) <= reset OR
              (read(4) AND NOT write(4)) OR
              (read(3) AND write(4));
regvclr(5) <= reset OR
              (read(5) AND NOT write(5)) OR
              (read(4) AND write(5));

-- Output the entry read from the queue
--
WITH read SELECT
   reg_file_out <= reg_out0 WHEN "000001",
                   reg_out1 WHEN "000010",
                   reg_out2 WHEN "000100",
                   reg_out3 WHEN "001000",
                   reg_out4 WHEN "010000",
                   reg_out5 WHEN OTHERS;

-- Detect when the queue is full.
--
full <= regv(0) AND
        regv(1) AND
        regv(2) AND
        regv(3) AND
        regv(4) AND
        regv(5);

-- Detect when the queue is empty.
--
empty <= NOT regv(0) AND
         NOT regv(1) AND
         NOT regv(2) AND
         NOT regv(3) AND
         NOT regv(4) AND
         NOT regv(5);
```

```
END regfile0;



%%%%%%%%%%%%%%%%%%%%%%%%%%
% file custm.vhdl_11
%%%%%%%%%%%%%%%%%%%%%%%%%%


--*********************************************************
-- Control unit state machine. It has three states:
--
--    1. Fetch state: CPU is done processing previous
--       and is waiting for the bus interface unit to load
--       a new instruction in the instruction register.
--    2. Clear state: CPU signals to the iterative logic
--       unit to clear its iterative state machines, and to
--       compute the relation function and detect which
--       variables, if any, are active.
--    3. Request state: CPU requests a cube output from the
--       iterative unit. CPU keeps on asserting this signal
--       until the iterative unit has computed all its
--       output cubes.
--*********************************************************


USE work.all;
ENTITY custm IS
   PORT (clk,       -- Clock from clock generator

         ld_ir,     -- From bus interface unit. Loads instruction
                    -- register, and signals to the state machine to
                    -- start next instruction execution.

         wait_out,  -- When asserted by ILU, indicates that the
                    -- ILU still has more output cubes to compute.

         reset,     -- External CPU reset.

         stall,     -- Register file is empty, so CPU should stall
                    -- until a register file entry becomes available
                    -- for the ILU to store another cube result.
         prime:     -- Opcode bit which indicates that the operation
                    -- is simple or complex operation (one cube result)
                    -- and not a sequential operation. this bit forces
                    -- a one cycle count pulse for prime operations to
                    -- make the counter counts to one, the number of
                    -- output cubes for prime operations.

                IN BIT;

         clear,     -- signal to ILU to clear its sequential state
                    -- machines and identify active variables.

         request,   -- signal to ILU to compute next cube result.

         count,     -- signal to the counter to count one more cube,
                    -- and for the register file to write the cube
                    -- currently output by the ILU.

         fetch:     -- signal to the bus unit to fetch the next
                    -- instruction.

                OUT BIT);
END custm;

ARCHITECTURE custm0 OF custm IS

   SIGNAL fetch_state, clear_state, request_state,
          next_state_fetch, next_state_request,
          next_state_clear, update_state, count_in: BIT;
   SIGNAL next_state, state: BIT_VECTOR (1 DOWNTO 0);
   SIGNAL slct: BIT_VECTOR (2 DOWNTO 0);

   COMPONENT flip_flop
             PORT (clk, enable, reset, din: IN BIT;
                   dout: OUT BIT);
   END COMPONENT;

   BEGIN

      -- decode current state
      --
```

```
fetch_state   <= '1' WHEN (state = "00")
                 ELSE '0';
clear_state   <= '1' WHEN (state = "01")
                 ELSE '0';
request_state <= '1' WHEN (state = "10")
                 ELSE '0';

-- compute next state
--
next_state_fetch   <= request_state AND
                      NOT wait_out AND
                      NOT stall;
next_state_clear   <= fetch_state AND
                      ld_ir;
next_state_request <= clear_state;

-- drive current outputs
--
clear   <= clear_state;
request <= request_state;
fetch   <= fetch_state;

slct <= (next_state_fetch & next_state_clear &
         next_state_request);

-- encode next state, and latch into D flip-flops.
--
WITH slct SELECT
   next_state <= "00" WHEN "100",
                 "01" WHEN "010",
                 "10" WHEN "001",
                 state WHEN OTHERS;

update_state <= '1';
count_in     <= request_state AND
                (wait_out OR prime OR stall);

msff0: flip_flop
       PORT MAP (clk, update_state, reset, count_in, count);
msff1: flip_flop
       PORT MAP (clk, update_state, reset, next_state(0), state(0));
msff2: flip_flop
```

```
            PORT MAP (clk, update_state, reset, next_state(1), state(1));

END custm0;



%%%%%%%%%%%%%%%%%%%%%%
% file ident_main.vhdl_10
%%%%%%%%%%%%%%%%%%%%%%


--*******************************************************************
-- This functional block contains the logic which identifies if
-- a variable is "specific" during a complex or sequential operation.
--
-- The block contains 4 carry chains. Two AND carry chains, and
-- two OR carry chains, with one of the two pairs of chains used
-- for a particular instruction to determine which variables are
-- "specific". A bit in the instruction (and_or) determines if the
-- "AND" or "OR" carry chains are to be used. If the sequential
-- operation is of the "AND" type, then a variable is "specific" if
-- all its bit positions meet the "specific" relation specified in
-- "rel" subfield of the instruction. If the sequential operation is
-- of the "OR" type, then a variable is specific if one or more of
-- its bit positions satisfy the specific relation for the
-- instruction.
--
-- Each pair of carry chains consist of a carry right, and carry
-- left chains. The use of symmetrical chains one to the left, and
-- one to the right, instead of a one directional chain, and a
-- confirm chain reduces the carry propagation delay by half. With
-- a confirm chain implementation, the carry has to propagate all
-- the way in one direction and then the confirm, in the worst
-- case, has to propagate afterwards all the way in the opposite
-- direction before a variable is identified. In this two chain
-- implemenatation, propagation right, and propagation left happens
-- simultaneously, therefore cutting down the worst case propagation
-- delay by half without adding extra hardware.
--*******************************************************************
```

```
ENTITY ident_main IS
   PORT (and_carry_r_in,  -- "AND" right carry input from the cell to the left

         and_carry_l_in,  -- "AND" left carry input from the cell to the right

         or_carry_r_in,   -- "OR" right carry input from the cell to the left

         or_carry_l_in,   -- "OR" left carry input from the cell to the right

         left_edge,       -- An input to each iterative cell which comes from
                          -- right edge register bit for the the cell on its left.
                          -- This bit is set to '1' at the higher hierarchical
                          -- level (ILU level) for cell 15.
                          --
                          -- It is very important to set the right edge bit to the
                          -- left cell of the leftmost non-water (enabled) cell,
                          -- if the leftmost cell is not cell 15. Carry generation
                          -- requires proper identification of the left edge of
                          -- each variable. So though a cell to the left of a left
                          -- edge cell may be watered, the edge register for
                          -- a particular instruction still need to be set for
                          -- watered variables for correct operation of the carry
                          -- cahinas.

         right_edge,      -- edge register bit which indicates that a particular
                          -- cell in an operation is the rightmost position of
                          -- a variable. As explained above, this same bit also
                          -- indicates to the cell to the right that it is the
                          -- leftmost cell of a variable.

         water,           -- bit from the "water" register, which indicates if
                          -- a cell shouldd be enabled in a particular
                          -- instrcution.

         rel0,            -- The output of an instruction's "specific" relation
                          -- on bit 0 of an iterative cell.

         rel1,            -- The output of an instruction's "specific" relation
                          -- on bit 1 of an iterative cell.

         and_or:          -- Instruction bit which determines if the specific
                          -- relation of an instruction is of the "AND" or "OR"
                          -- type.

                  IN BIT;

         and_carry_r_out, -- "AND" carry output to the cell to the right

         and_carry_l_out, -- "AND" carry output to the cell to the left

         or_carry_r_out,  -- "OR" carry output to the cell to the right

         or_carry_l_out,  -- "OR" carry output to the cell to the left

         var:             -- set when a cell is identified to be contained
                          -- within a "specific" variable.

                  OUT BIT);
END ident_main;

ARCHITECTURE ident_main0 OF ident_main IS

   SIGNAL gen_and_carry_l, gen_and_carry_r, prop_and_carry,
          gen_or_carry, prop_or_carry_l, prop_or_carry_r,
          temp1, temp2, temp3, temp4: BIT;

   BEGIN

      -- AND carry chains logic
      --
      prop_and_carry  <= (rel0 AND rel1) OR water; -- propagate AND carry if the
                                                   -- specific "AND" relation
                                                   -- results in high output on
                                                   -- both bit positions of an
                                                   -- iterative cell.

      gen_and_carry_r <= left_edge AND       -- generate "AND" carry at the right
                         prop_and_carry AND  -- edge position of a variable, if
                         NOT water;          -- the right cell meets the
                                             --  "specific" relation.

      gen_and_carry_l <= right_edge AND      -- generate "AND" carry at the left
                         prop_and_carry AND  -- edge position of a avariable, if
                         NOT water;          -- the left cell meets the
```

```
                                        -- "specific" relation.

-- propagate the carry from the left and the carry from the right if
-- a cell meets the "specific" relation, and the cell is not an edge
-- cell. If the cell is an edge cell, then drive an active carry only
-- if the edge cell meets the "specific" relation.
--
temp1           <= (and_carry_r_in AND prop_and_carry)
                   OR gen_and_carry_r;
temp2           <= (and_carry_l_in AND prop_and_carry)
                   OR gen_and_carry_l;
and_carry_r_out <= temp1;
and_carry_l_out <= temp2;

-- "OR" carry chains logic:
--
-- generate a carry to the left and to the right if a cell meets the
-- "specific" relation. If a cell is an edge cell do not propagate the
-- carry from outside the variabel. If a cell is not an edge cell
-- perform the "OR" carry function by propagating the left and right
-- carry even when the cell does not meet the "specific" relation.
--
gen_or_carry    <= (rel0 OR rel1) AND NOT water;
prop_or_carry_r <= (NOT left_edge) OR water;
prop_or_carry_l <= (NOT right_edge) OR water;
temp3           <= (or_carry_r_in AND prop_or_carry_r)
                   OR gen_or_carry;
temp4           <= (or_carry_l_in AND prop_or_carry_l)
                   OR gen_or_carry;
or_carry_r_out  <= temp3;
or_carry_l_out  <= temp4;

-- now that the carry out to right and to left is computed, identify
-- if a cell is within a specific variable. If the carry chain function
-- is "AND", a cell is within a variable if both carry right and
-- carry left are active. From the carry logic above it can easily be
-- seen that both "AND" chain outputs active implies that this cell, all
-- cells to the left, and all cells to the right meet the specific
-- relation.
--
-- If the carry chain function is "OR" than from the "OR" carry logic
-- above, this cell, one or more of the cells to the right, or one or
```

```
      -- more of the cells to the left meet the specific relation, i.e. the
      -- variable is specific. So var in this case is simply the OR of the
      -- left and right carry outputs.
      --
      var <= (temp1 AND temp2) WHEN (and_or = '1') ELSE
             (temp3 OR temp4);

END ident_main0;




%%%%%%%%%%%%%%%%%%%%%%%
% psm.vhdl_11
%%%%%%%%%%%%%%%%%%%%%%%


--*************************************************************************
-- Iterative cell state machine.
--
-- The iterative logic unit in this cube calculus architecture uses
-- distributed control to execute sequential cube calculus efficently.
-- Each iterative cell in the ILU contains a small state machine which
-- with the help of simple global control from the CPU control unit,
-- determine the state during execution of the iterative cell, and the
-- type of funvtion the iterative cell executes. i.e. the state of the
-- state machine determines if the iterative cell computes the "before",
-- "active" or "after" functions, based on the time in the execution
-- cycle, and whether the iterative cell belongs to a specific variable,
-- or not.
--
-- The state machine contaions the following logic component:
--
-- psmcst: consists of two D edge triggered flip-flops associated with
--         the two state bits of the iterative cell. Signals from the
--         control unit state machine resets or latches the inputs to
--         these flip-flops. A stall signal from the output data queue
--         stalls the state machine when the queue is full and cannot
--         accept any more output cubes.
--
-- psmnst: combinatorial logic which computes the next state based on
--         inputs to the state machine and the current state.
```

```
--
-- psmout: combinatorial logic which outputs the iterative control wait
--         carry chain, and the control which selects the function to
--         apply the the cube operand bits for a particular execution
--         cycle.
--*************************************************************************

USE work.all;

ENTITY psm IS
   PORT (clk,          -- global CPU clock

         request,      -- Control unit requests another sequential output
                       -- cube. This signals clocks the state flip-flops,
                       -- and latches the next state value in the state
                       -- flip_flops, unless "stall" is active.

         stall,        -- Input from the output data queue which indicates
                       -- that the queue is full. Stalls the iterative
                       -- control state machine until the output queue has
                       -- empty entries to accept new output cubes.

         clear,        -- Control unit signal which triggers the start of
                       -- the ILU execution. It resets all iterative cells
                       -- state to the "before" state, and allows a cycle
                       -- time for the iterative cells to identify if they
                       -- belong to a specific variable.

         wait_in,      -- Iterative carry chain input. When asserted, it
                       -- indicates to a cell that there is a specific
                       -- variable to the left for which the corresponding
                       -- output cube has not been yet computed. This
                       -- signal keeps the cell from making a transition
                       -- into an active or after state until all specific
                       -- variables to its left has made their active
                       -- computation.

         var,          -- Active when a cell identifies itself as part of
                       -- a specific variable.

         right_edge,   -- The cell is the rightmost of a variable.
         water,       -- From the water register. Remove a cell from a
                      -- computation when set to '1'

         prime:       -- Instruction bit which determines if the operation
                      -- is simple/complex (one output cube) or sequential
                      -- (0, 1 or more output cubes).

             IN BIT;

         wait_out:    -- Iterative carry chain output. Asserted by a
                      -- specific variable until it has made its active
                      -- computation.

             OUT BIT;

         slct:        -- Control signal which selects if the function
                      -- executed by the iterative cell is "before",
                      -- "active" or "after".

             OUT BIT_VECTOR (1 DOWNTO 0));
END psm;

ARCHITECTURE psm0 OF psm IS

   SIGNAL next_state, state: BIT_VECTOR (1 DOWNTO 0);

   COMPONENT psmcst
             PORT (next_state: IN BIT_VECTOR (1 DOWNTO 0);
                   clk, request, clear, stall: IN BIT;
                   state: OUT BIT_VECTOR (1 DOWNTO 0));
   END COMPONENT;

   COMPONENT psmnst
             PORT (wait_in, var: IN BIT;
                   state: IN BIT_VECTOR (1 DOWNTO 0);
                   next_state: OUT BIT_VECTOR (1 DOWNTO 0));
   END COMPONENT;

   COMPONENT psmout
             PORT (wait_in, right_edge, var, water, prime: IN BIT;
                   state: IN BIT_VECTOR (1 DOWNTO 0);
                   wait_out: OUT BIT;
```

```
                 slct: OUT BIT_VECTOR (1 DOWNTO 0));
   END COMPONENT;

   BEGIN

      -- next state logic component
      --
      nst0: psmnst
            PORT MAP (wait_in, var, state, next_state);

      -- state flip-flops component
      --
      cst0: psmcst
            PORT MAP (next_state, clk, request, clear, stall,  state);

      -- output logic, and iterative control chain component
      --
      out0: psmout
            PORT MAP (wait_in, right_edge, var, water, prime,
                         state, wait_out, slct);
END psm0;


%%%%%%%%%%%%%%%%%%%%%%%%
% file psmnst.vhdl_5
%%%%%%%%%%%%%%%%%%%%%%%%



--*****************************************************************
-- This functional block determines the next state the iterative
-- state machine should transit into, if control unit state
-- machine "request" a new cube computation, and the output
-- data queue does not "stall" because it is full.
--*****************************************************************

ENTITY psmnst IS
   PORT (wait_in,        -- Iterative carry chain input. When asserted,
                         -- it indicates to a cell that there is a
                         -- specific variable to the left for which the
                         -- corresponding output cube has not yet been
                         -- computed. This signal keeps the cell from
```

```
                        -- making a transition into an active or after
                        -- state until all specific variables to its
                        -- left has made their active computation.

         var:           -- Active when a celll identifies itself as
                        -- part of a specific variable.

               IN BIT;

         state:         -- iterative cell state variable

               IN BIT_VECTOR (1 DOWNTO 0);

         next_state:    -- next state into which to make a transition.

               OUT BIT_VECTOR (1 DOWNTO 0));
END psmnst;

ARCHITECTURE psmnst0 OF psmnst IS

   SIGNAL slct: BIT_VECTOR (1 DOWNTO 0);
   SIGNAL next_state_act, next_state_aft,
          bef_state, act_state, aft_state: BIT;

   BEGIN

      -- decode current state
      --
      bef_state <= '1' WHEN (state = "00")
                   ELSE '0';
      act_state <= '1' WHEN (state = "01")
                   ELSE '0';
      aft_state <= '1' WHEN (state = "10")
                   ELSE '0';

      -- go to active state if the cell is a specific variable,
      -- and all specific cells to the left have made their
      -- active computation (indicated by an inactive wait carry
      -- input).
      --
      next_state_act <= bef_state AND
                        var AND
```

```
                       NOT wait_in;

      -- go to sfter state if the cell is not a specific
      -- variable, and all specific cells to the left have made
      -- their active computation (indicated by an inactive wait
      -- carry input).
      --
      next_state_aft <= (bef_state AND
                         NOT var AND
                         NOT wait_in) OR act_state;

      -- encode into 2 bits the next state value
      --
      slct <= (next_state_act & next_state_aft);
      WITH slct SELECT
         next_state <= "01" WHEN "10",
                       "10" WHEN "01",
                       state WHEN OTHERS;
END psmnst0;



%%%%%%%%%%%%%%%%%%%%%%%%%%
% psmout.vhdl_5
%%%%%%%%%%%%%%%%%%%%%%%%%%



--******************************************************************
-- Iterative carry chain and output logic:
--
-- The "wait" carry chain is asserted by an iterative cell until
-- it has had its turn, in case it is a specific variable, to do
-- its active computation, and produce its output cube. This
-- carry chain propagates from left to right. As long as the wait
-- carry is high, an iterative cell is in the "before" state, and
-- performs the "before" binary function of the instruction. When
-- wait carry becomes low into a cell, the cell makes either
-- makes a transition into the active state for one cycle to
-- compute its active cube output, or makes an immediate
-- transition into an agter state, and propagates a 0 wait carry.
```

```
--*****************************************************************

ENTITY psmout IS
   PORT (wait_in,     -- Iterative control carry chain input

         right_edge,  -- From the edge register. Indicates that the
                      -- cell is the rightmost of a variable.

         var,         -- The cell has identified itself as a specific
                      -- variable.

         water,       -- Water register bit which disables a cell
                      -- from the computation when active.

         prime:       -- Instruction register bit which determines if
                      -- an instruction is simple/complex (one cube
                      -- output) or sequential (0, 1, or more cube
                      -- outputs)

              IN BIT;

         state:       -- Current iterative state

              IN BIT_VECTOR (1 DOWNTO 0);

         wait_out:    -- Output of the iterative control carry chain

              OUT BIT;

         slct:        -- Control output which selects the function
                      -- to use (before, active or after) in a
                      -- computation cycle.

              OUT BIT_VECTOR (1 DOWNTO 0));
END psmout;

ARCHITECTURE psmout0 OF psmout IS

   SIGNAL bef_state, act_state, aft_state, gen_wait: BIT;
   SIGNAL temp1: BIT_VECTOR (1 DOWNTO 0);

   BEGIN
```

```
      -- decode the current state
      --
      bef_state <= '1' WHEN (state = "00")
                   ELSE '0';
      act_state <= '1' WHEN (state = "01")
                   ELSE '0';
      aft_state <= '1' WHEN (state = "10")
                   ELSE '0';

      -- Generate a wait carry when a specific variable  cell is in
      -- a before state (has not done the active computation), the
      -- cell is enabled (water bit is '0'), and the instruction is
      -- sequential.
      --
      gen_wait  <= bef_state AND
                   right_edge AND
                   var AND
                   NOT water AND
                   NOT prime;

      -- Propagate the wait carry, or drive an active wait, if the
      -- cell has generated a wait carry.
      --
      wait_out  <= wait_in OR gen_wait;

      -- Output control which selects the function to execute in
      -- a particular cycle, dependoing on the state and the type
      -- of operation (prime vs. sequential).
      --
      temp1     <= (prime & var);
      WITH temp1 SELECT
         slct <= "00" WHEN "10",
                 "01" WHEN "11",
                 state WHEN OTHERS;
END psmout0;



%%%%%%%%%%%%%%%%%%%%%%%%%%
% file psmcst.vhdl_15
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%


--*******************************************************************
-- This block contains 2 D edge triggered flip-flops which contains
-- the current state for the iterative cell.
--*******************************************************************

USE work.all;

ENTITY psmcst IS
   PORT (next_state:    -- Next state to latch into the flip-flops

              IN BIT_VECTOR (1 DOWNTO 0);

         clk,           -- Global CPU clock

         request,       -- Control signal from the control unit state. Causes
                        -- iterative cell state transition by enabling the
                        -- state flip-flip.

         clear,         -- Control signal from the control unit which fires the
                        -- iterative cells sequential execution. Resets the state
                        -- machine to the "before" (before active i.e.) state.

         stall:         -- Stall signal from the output data queue

              IN BIT;

         state:         -- Output of the state flip-flops

              OUT BIT_VECTOR (1 DOWNTO 0));
END psmcst;

ARCHITECTURE psmcst0 OF psmcst IS

   SIGNAL update_state: BIT;

   COMPONENT flip_flop
            PORT (clk, enable, reset, din: IN BIT;
                  dout: OUT BIT);
   END COMPONENT;
   BEGIN
      update_state <= request AND NOT stall;
      ms1: flip_flop
           PORT MAP (clk, update_state, clear, next_state(1), state(1));

      ms0: flip_flop
           PORT MAP (clk, update_state, clear, next_state(0), state(0));

END psmcst0;
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%
% file  itcell.vhdl_7
%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
--*************************************************************************
-- This is the iterative cell block. It is the most primitive
-- cell of the cube calculus execution unit, named in this design
-- the ILU. Each iterative cell operates on two bits of the two
-- input cubes for a particulat instruction. These two bits can
-- represent the four possible values of a binary variable
-- using positional notaation encodings (01 for truth value 0, 10
-- for value 1, 11 for don't care, and 00 for contradiction or
-- empty set). If multi-valued calculus is needed, adjacent iterative
-- cells can be combined together to represent a variable. For
-- example, two adjacent iterative cells can represent in postitional
-- notation a four-valued logic variable. The edge register define
-- the boundaries of the variables within the 16-itcell ILU. the
-- water register disables unused iterative cells, if an instruction
-- is executed on cubes which contain less number of variables than
-- can fit within a 32-bit operand.
--
-- The functions of the iterative cell are:
--
--    1. Identify if the cell is within a specific variable. A
--       specific variable, is a variable position within the input
--       cubes which meets the "specific" relation for the executed
--       instruction. the specific relation is defined in "rel"
--         subfield of the instruction. There are two types of specific
--         relations that can be defined using the and_or bit of the
--         instruction. An "AND" relation requires a specific variable
--         to meet the relation at each bit position within the
--         variable. An "OR" relation requires a specific variable to
--         meet the relation at one or more bit positions within the
--         variable.
--
--      2. Each iterative cell has its own embedded iterative control
--         state machine. the state machine within a cell is used to
--         execute sequential cube calculus operations. In sequential
--         operations, each iterative cell compute one of three possible
--         functions on its bits. These functions are called before,
--         after, and active, and are defined in the "bef", "act" and
--         "aft" 4-bit subfields of the instruction. The number of
--         output cubes for a sequential operation is equal to the number
--         of specific variables within the input cubes. Each variable
--         contribute to an output cube its after, active or before
--         result depending on its position relative to the specific
--         variable corresponding to this output cube. The state machine
--         has three states which determine if a cell is currently
--         executing the before, after or active function. The state
--         machine output controls a three to one mux which select the
--         before, after, or active function subfield from the executed
--         instruction.
--
--      3. Each cell has logic associated with the iterative control
--         carry chain. This carry chain is called the "wait" chain.
--         A wait signal propagates from one specific variable to another,
--         in the left to right direction, and is used to control the
--         state transitions of the iterative cell state machine.
--
--      4. Each cell propagates "AND" and "OR" carries, in both directions
--         which are used to determine if a variable is specific, in case
--         the cell is within a multi-valued logic variable. If a cell
--         identifies itself within a specific variable, it asserts "var"
--         which is also used to control its state machine transitions.
--************************************************************************

USE work.all;

ENTITY itcell IS
```

```
PORT (rel, bef, act, aft:    -- relation, before, active and after subfields
                             -- from the instruction register

           IN BIT_VECTOR (3 DOWNTO 0);

      clk,                   -- CPU clock

      water,                 -- Water register bit, disables a cell when high

      left_edge,             -- Indicates cell is leftmost in a variable

      right_edge,            -- Indicates cell is rightmost in a variable

      and_or,                -- From instruction: determines "AND" or "OR"
                             -- type specific relation.

      a0, b0, a1, b1,        -- Input bits from the input operands

      wait_in,               -- Input of iterative control carry chain

      request,               -- Input of control unit state machine. It is
                             -- a request for another cube result, so causes
                             -- iterative state machine transition in the
                             -- current active cell, next active cell, and all
                             -- cells in between (left to right).

      stall,                 -- Register file is full and cannot take any
                             -- new result, so CPU should stall.

      clear,                 -- From control unit state machine. It clears
                             -- the state of the iterative cell, to allow
                             -- the iterative cell to identifiy itcelf, and
                             -- set its state machine to start executing
                             -- another operation.

      prime,                 -- Indicates if an operation is simple/comples
                             -- (one output cube), or sequential (0 or more
                             -- output cubes.

      and_carry_r_in, and_carry_l_in,   -- "AND" carry chain inputs

      or_carry_r_in, or_carry_l_in:     -- "OR" carry chain inputs
```

```
              IN BIT;

         wait_out,                         -- iterative control chain output

         and_carry_r_out, and_carry_l_out, -- "AND" carry chain outputs

         or_carry_r_out, or_carry_l_out,   -- "OR" carry chain outputs

         c0, c1:                           -- output cube bits

              OUT BIT);
END itcell;

ARCHITECTURE itcell0 OF itcell IS

   SIGNAL rel0, rel1, var: BIT;
   SIGNAL slct: BIT_VECTOR (1 DOWNTO 0);
   SIGNAL fun: BIT_VECTOR (3 DOWNTO 0);

   COMPONENT mux4to1
      PORT (in0, in1, in2, in3, sel0, sel1: IN BIT;
            out0: OUT BIT);
   END COMPONENT;

   COMPONENT funmux
      PORT (slct: IN BIT_VECTOR (1 DOWNTO 0);
            bef, act, aft: IN BIT_VECTOR (3 DOWNTO 0);
            fun: OUT BIT_VECTOR (3 DOWNTO 0));
   END COMPONENT;

   COMPONENT ident_main
      PORT (and_carry_r_in, and_carry_l_in, or_carry_r_in,
            or_carry_l_in, left_edge, right_edge, water,
            rel0, rel1, and_or: IN BIT;

            and_carry_r_out, and_carry_l_out, or_carry_r_out,
            or_carry_l_out, var: OUT BIT);
   END COMPONENT;

   COMPONENT psm
      PORT (clk, request, stall, clear, wait_in, var,
         right_edge, water, prime: IN BIT;

         wait_out: OUT BIT;

         slct: OUT BIT_VECTOR (1 DOWNTO 0));
END COMPONENT;

BEGIN

   -- Two 4 to 1 muxes, used to perform any one of 16 possible binary functions.
   -- The 4 bit input represent the values in the truth table for the binary
   -- function. This input comes from the relation subfield of the
   -- instruction. the two select bits are the input operand bits, and the
   -- output is the result of the relation function specified in the 4 bit
   -- input, on the two operand bits.
   --
   mux0: mux4to1
         PORT MAP (rel(0), rel(1), rel(2), rel(3),
                   b0, a0, rel0);
   mux1: mux4to1
         PORT MAP (rel(0), rel(1), rel(2), rel(3),
                   b1, a1, rel1);

   -- A mux to select "bef", "act" or "aft" 4-bit function encodings from the
   -- instruction. The select lines are controlled by the iterative state machine.
   -- A 4-bit function encoding is simply the values of the truth table for
   -- the binary function.
   --
   mux2: funmux
         PORT MAP (slct, bef, act, aft, fun);

   -- Two 4 to 1 muxes used to execute the "before", "active" or "after" functions
   -- on the two cube operand bits.
   --
   mux3: mux4to1
         PORT MAP (fun(0), fun(1), fun(2), fun(3),
                   b0, a0, c0);
   mux4: mux4to1
         PORT MAP (fun(0), fun(1), fun(2), fun(3),
                   b1, a1, c1);

   -- This component identified is the cell is within a specific variable
```

```
      --
      ident0: ident_main
              PORT MAP (and_carry_r_in, and_carry_l_in, or_carry_r_in,
                        or_carry_l_in, left_edge, right_edge, water,
                        rel0, rel1, and_or, and_carry_r_out,
                        and_carry_l_out, or_carry_r_out, or_carry_l_out,
                        var);

      -- This is the iterative state machine
      --
      stm0: psm
            PORT MAP (clk, request, stall, clear, wait_in, var,
                      right_edge, water, prime, wait_out, slct);

END itcell0;



%%%%%%%%%%%%%%%%%%%%%%%%%%%
% file ilu.vhdl_13
%%%%%%%%%%%%%%%%%%%%%%%%%%%


--*************************************************************************
-- This is the iterative logic unit. It is the main execution unit for
-- the cube calculus CPU. It is simply an array of 16 iterative cells.
-- Each iterative cell executes the specified cube calculus operation on
-- two bits of a teo 32-bit cubes. Therefore, if binary variables only
-- are used in the cube operands, the ILU executes any cube calculus
-- operation on a cube which can contain up to 16 literals.
--
-- To understand the IN, OUT and internal signals of the ILU, see the
-- comments in the "itcell" component, and the logic components inside
-- the iterative cell.
--
-- Carry signals are input and output from the ILU iterative cell chains.
-- this makes it possible to combine two or more ILUs together to build
-- a CPU which can execute operations on cubes that have more than 16
-- binary variables.
--**************************************************************************

USE work.all;
```

```
ENTITY ilu IS

   PORT (water, right_edge: IN BIT_VECTOR (15 DOWNTO 0);
         rel, bef, act, aft: IN BIT_VECTOR (3 DOWNTO 0);
         cube0, cube1: IN BIT_VECTOR (31 DOWNTO 0);
         wait_in, and_carry_r_in, and_carry_l_in,
         or_carry_r_in, or_carry_l_in, and_or, prime,
         request, stall, clear, ilu_left_edge, clk: IN BIT;
         out_cube: OUT BIT_VECTOR (31 DOWNTO 0);
         and_carry_r_out, and_carry_l_out, or_carry_r_out,
         or_carry_l_out, wait_out, ilu_right_edge: OUT BIT);
END ilu;

ARCHITECTURE ilu0 OF ilu IS

   SIGNAL and_carry_r, and_carry_l, or_carry_r, or_carry_l,
          wait_carry, edge: BIT_VECTOR (16 DOWNTO 0);

   COMPONENT itcell
             PORT (rel, bef, act, aft: BIT_VECTOR (3 DOWNTO 0);

                   clk, water, left_edge, right_edge, and_or,
                   a0, b0, a1, b1, wait_in, request, stall, clear,
                   prime, and_carry_r_in, and_carry_l_in,
                   or_carry_r_in, or_carry_l_in: IN BIT;

                   wait_out, and_carry_r_out, and_carry_l_out,
                   or_carry_r_out, or_carry_l_out, c0, c1: OUT BIT);
   END COMPONENT;

   BEGIN

      -- set the ILU edge carry chain, and left edge signals.
      --
      and_carry_r(16) <= and_carry_r_in;
      and_carry_l(0)  <= and_carry_l_in;
      and_carry_r_out <= and_carry_r(0);
      and_carry_l_out <= and_carry_l(16);
      or_carry_r(16)  <= or_carry_r_in;
      or_carry_l(0)   <= or_carry_l_in;
      or_carry_r_out  <= or_carry_r(0);
```

```
or_carry_l_out  <= or_carry_l(16);
wait_carry(16)  <= wait_in;
wait_out        <= wait_carry(0);
edge            <= ilu_left_edge & right_edge;
ilu_right_edge  <= right_edge(0);

-- place and connect 16 iterative cells to form one 16 binary variable
-- ILU.
--
it0:  itcell
      PORT MAP (rel, bef, act, aft, clk, water(0), edge(1), edge(0),
                and_or, cube0(1), cube1(1), cube0(0), cube1(0),
                wait_carry(1), request, stall, clear,
                prime, and_carry_r(1),
                and_carry_l(0), or_carry_r(1), or_carry_l(0),
                wait_carry(0), and_carry_r(0), and_carry_l(1),
                or_carry_r(0), or_carry_l(1), out_cube(1),
                out_cube(0));

it1:  itcell
      PORT MAP (rel, bef, act, aft, clk, water(1), edge(2), edge(1),
                and_or, cube0(3), cube1(3), cube0(2), cube1(2),
                wait_carry(2), request, stall, clear,
                prime, and_carry_r(2),
                and_carry_l(1), or_carry_r(2), or_carry_l(1),
                wait_carry(1), and_carry_r(1), and_carry_l(2),
                or_carry_r(1), or_carry_l(2), out_cube(3),
                out_cube(2));

it2:  itcell
      PORT MAP (rel, bef, act, aft, clk, water(2), edge(3), edge(2),
                and_or, cube0(5), cube1(5), cube0(4), cube1(4),
                wait_carry(3), request, stall, clear,
                prime, and_carry_r(3),
                and_carry_l(2), or_carry_r(3), or_carry_l(2),
                wait_carry(2), and_carry_r(2), and_carry_l(3),
                or_carry_r(2), or_carry_l(3), out_cube(5),
                out_cube(4));

it3:  itcell
      PORT MAP (rel, bef, act, aft, clk, water(3), edge(4), edge(3),
                and_or, cube0(7), cube1(7), cube0(6), cube1(6),
                  wait_carry(4), request, stall, clear,
                  prime, and_carry_r(4),
                  and_carry_l(3), or_carry_r(4), or_carry_l(3),
                  wait_carry(3), and_carry_r(3), and_carry_l(4),
                  or_carry_r(3), or_carry_l(4), out_cube(7),
                  out_cube(6));

it4:  itcell
      PORT MAP (rel, bef, act, aft, clk, water(4), edge(5), edge(4),
                  and_or, cube0(9), cube1(9), cube0(8), cube1(8),
                  wait_carry(5), request, stall, clear,
                  prime, and_carry_r(5),
                  and_carry_l(4), or_carry_r(5), or_carry_l(4),
                  wait_carry(4), and_carry_r(4), and_carry_l(5),
                  or_carry_r(4), or_carry_l(5), out_cube(9),
                  out_cube(8));

it5:  itcell
      PORT MAP (rel, bef, act, aft, clk, water(5), edge(6), edge(5),
                  and_or, cube0(11), cube1(11), cube0(10), cube1(10),
                  wait_carry(6), request, stall, clear,
                  prime, and_carry_r(6),
                  and_carry_l(5), or_carry_r(6), or_carry_l(5),
                  wait_carry(5), and_carry_r(5), and_carry_l(6),
                  or_carry_r(5), or_carry_l(6), out_cube(11),
                  out_cube(10));

it6:  itcell
      PORT MAP (rel, bef, act, aft, clk, water(6), edge(7), edge(6),
                  and_or, cube0(13), cube1(13), cube0(12), cube1(12),
                  wait_carry(7), request, stall, clear,
                  prime, and_carry_r(7),
                  and_carry_l(6), or_carry_r(7), or_carry_l(6),
                  wait_carry(6), and_carry_r(6), and_carry_l(7),
                  or_carry_r(6), or_carry_l(7), out_cube(13),
                  out_cube(12));

it7:  itcell
      PORT MAP (rel, bef, act, aft, clk, water(7), edge(8), edge(7),
                  and_or, cube0(15), cube1(15), cube0(14), cube1(14),
                  wait_carry(8), request, stall, clear,
                  prime, and_carry_r(8),
                  and_carry_l(7), or_carry_r(8), or_carry_l(7),
                  wait_carry(7), and_carry_r(7), and_carry_l(8),
                  or_carry_r(7), or_carry_l(8), out_cube(15),
                  out_cube(14));

it8:  itcell
      PORT MAP (rel, bef, act, aft, clk, water(8), edge(9), edge(8),
                  and_or, cube0(17), cube1(17), cube0(16), cube1(16),
                  wait_carry(9), request, stall, clear,
                  prime, and_carry_r(9),
                  and_carry_l(8), or_carry_r(9), or_carry_l(8),
                  wait_carry(8), and_carry_r(8), and_carry_l(9),
                  or_carry_r(8), or_carry_l(9), out_cube(17),
                  out_cube(16));

it9:  itcell
      PORT MAP (rel, bef, act, aft, clk, water(9), edge(10), edge(9),
                  and_or, cube0(19), cube1(19), cube0(18), cube1(18),
                  wait_carry(10), request, stall, clear,
                  prime, and_carry_r(10),
                  and_carry_l(9), or_carry_r(10), or_carry_l(9),
                  wait_carry(9), and_carry_r(9), and_carry_l(10),
                  or_carry_r(9), or_carry_l(10), out_cube(19),
                  out_cube(18));

it10: itcell
      PORT MAP (rel, bef, act, aft, clk, water(10), edge(11), edge(10),
                  and_or, cube0(21), cube1(21), cube0(20), cube1(20),
                  wait_carry(11), request, stall, clear,
                  prime, and_carry_r(11),
                  and_carry_l(10), or_carry_r(11), or_carry_l(10),
                  wait_carry(10), and_carry_r(10), and_carry_l(11),
                  or_carry_r(10), or_carry_l(11), out_cube(21),
                  out_cube(20));

it11: itcell
      PORT MAP (rel, bef, act, aft, clk, water(11), edge(12), edge(11),
                  and_or, cube0(23), cube1(23), cube0(22), cube1(22),
                  wait_carry(12), request, stall, clear,
                  prime, and_carry_r(12),
                  and_carry_l(11), or_carry_r(12), or_carry_l(11),
                  wait_carry(11), and_carry_r(11), and_carry_l(12),
                or_carry_r(11), or_carry_l(12), out_cube(23),
                out_cube(22));

it12: itcell
      PORT MAP (rel, bef, act, aft, clk, water(12), edge(13), edge(12),
                and_or, cube0(25), cube1(25), cube0(24), cube1(24),
                wait_carry(13), request, stall, clear,
                prime, and_carry_r(13),
                and_carry_l(12), or_carry_r(13), or_carry_l(12),
                wait_carry(12), and_carry_r(12), and_carry_l(13),
                or_carry_r(12), or_carry_l(13), out_cube(25),
                out_cube(24));

it13: itcell
      PORT MAP (rel, bef, act, aft, clk, water(13), edge(14), edge(13),
                and_or, cube0(27), cube1(27), cube0(26), cube1(26),
                wait_carry(14), request, stall, clear,
                prime, and_carry_r(14),
                and_carry_l(13), or_carry_r(14), or_carry_l(13),
                wait_carry(13), and_carry_r(13), and_carry_l(14),
                or_carry_r(13), or_carry_l(14), out_cube(27),
                out_cube(26));

it14: itcell
      PORT MAP (rel, bef, act, aft, clk, water(14), edge(15), edge(14),
                and_or, cube0(29), cube1(29), cube0(28), cube1(28),
                wait_carry(15), request, stall, clear,
                prime, and_carry_r(15),
                and_carry_l(14), or_carry_r(15), or_carry_l(14),
                wait_carry(14), and_carry_r(14), and_carry_l(15),
                or_carry_r(14), or_carry_l(15), out_cube(29),
                out_cube(28));

it15: itcell
      PORT MAP (rel, bef, act, aft, clk, water(15), edge(16), edge(15),
                and_or, cube0(31), cube1(31), cube0(30), cube1(30),
                wait_carry(16), request, stall, clear,
                prime, and_carry_r(16),
                and_carry_l(15), or_carry_r(16), or_carry_l(15),
                wait_carry(15), and_carry_r(15), and_carry_l(16),
                or_carry_r(15), or_carry_l(16), out_cube(31),
                out_cube(30));
```

```
END ilu0;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% cpu.vhdl_17
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


--***********************************************************
-- Cube calculus CPU. Consists of:
--
--   1. Register file in which output cubes are written
--      by the ILU, and read by the bus interface unit.
--      The register file is imlemented as a queue.
--   2. Instruction register in which commands are written
--      by the bus interface unit.
--   3. Water register which is used to disable operations
--      on unused variables in the operands.
--   4. Edge register which indicates to the ILU the bit
--      positions of differnet operand variables.
--   5. A 5 bit counter, which counts the number of output
--      cube from a particular operation. Bus interface unit
--      has a direct access to a 5 bit bus output from this
--      counter. How exactly the bus unit would use this
--      value is left as an open implementation issue for
--      the bus unit designer.
--   6. Two 32 bit operand registers.
--   7. A control unit state machine.
--   8. An iterative logic unit, which consists of an array
--      of 16 iterative cells, capable of performing simple,
--      complex, or sequential cube calculus operations on
--      two 32 bit operands. The two operands can represent
--      a cube consisting of a product of up to 16 binary
--      variables.
--***********************************************************

USE work.all;

ENTITY cpu IS
   PORT (reset,        -- External reset pin
         ld_ir,        -- Asserted by bus interface unit to
                       -- write a new instruction into the opcode
                       -- register. Triggers the start of execution.

         ld_water,     -- Asserted by bus unit to write water register.

         ld_edge,      -- Asserted by bus unit to write edge register.

         rd_reg_file,  -- Asserted by bus unit to read head of
                       -- data output queue (register file).

         ld_cube0,     -- Asserted by bus unit to write operand 0 register.

         ld_cube1:     -- Asserted by bus unit to write operand 1 register.

              IN BIT;

         ibus:         -- Instruction bus, used by bus unit to load
                       -- instruction, water, edge, operand0 and operand1
                       -- registers.

              IN BIT_VECTOR (31 DOWNTO 0);

         dbus:         -- Data bus, used by the bus unit to read output
                       -- cubes from the register file queue.

              OUT BIT_VECTOR (31 DOWNTO 0);

         cube_num:     -- Output from 5 bit counter. Has the value of  the
                       -- number of cubes generated by a particular operation.

              OUT BIT_VECTOR (4 DOWNTO 0);

         fetch,        -- Indicates  that the CPU is done with the previous
                       -- instruction, and is ready for another one.

         empty,        -- Indicates that the data output queue is empty.

         clkout:       -- Clock output from the CPU to the bus interface unit,
                       -- and external bus.
              OUT BIT);
END cpu;
```

```
ARCHITECTURE cubecalc OF CPU IS

   SIGNAL water, right_edge: BIT_VECTOR (15 DOWNTO 0);
   SIGNAL rel, bef, act, aft: BIT_VECTOR (3 DOWNTO 0);
   SIGNAL cube0, cube1, out_cube,
          opcode: BIT_VECTOR (31 DOWNTO 0);
   SIGNAL wait_in, and_carry_r_in, and_carry_l_in,
          or_carry_r_in, or_carry_l_in, and_or, prime,
          request, clear, and_carry_r_out, and_carry_l_out,
          or_carry_r_out, or_carry_l_out, wait_out, stall,
          count, clk, ilu_left_edge, ilu_right_edge: BIT;

   COMPONENT ilu
             PORT (water, right_edge: IN BIT_VECTOR (15 DOWNTO 0);
                   rel, bef, act, aft: IN BIT_VECTOR (3 DOWNTO 0);
                   cube0, cube1: IN BIT_VECTOR (31 DOWNTO 0);
                   wait_in, and_carry_r_in, and_carry_l_in,
                   or_carry_r_in, or_carry_l_in, and_or, prime,
                   request, stall, clear, ilu_left_edge, clk: IN BIT;
                   out_cube: OUT BIT_VECTOR (31 DOWNTO 0);
                   and_carry_r_out, and_carry_l_out, or_carry_r_out,
                   or_carry_l_out, wait_out, ilu_right_edge: OUT BIT);
   END COMPONENT;

   COMPONENT custm
             PORT (clk, ld_ir, wait_out, reset, stall, prime: IN BIT;
                   clear, request, count, fetch: OUT BIT);
   END COMPONENT;

   COMPONENT counter
             PORT (clk, count, ld_ir, reset, stall: IN BIT;
                   count_out: OUT BIT_VECTOR (4 DOWNTO 0));
   END COMPONENT;

   COMPONENT reg16
             PORT (ld_reg, clk: IN BIT;
                   reg_in: IN BIT_VECTOR (15 DOWNTO 0);
                   reg_out: OUT BIT_VECTOR (15 DOWNTO 0));
   END COMPONENT;

   COMPONENT reg32
```

```
          PORT (ld_reg, clk: IN BIT;
                reg_in: IN BIT_VECTOR (31 DOWNTO 0);
                reg_out: OUT BIT_VECTOR (31 DOWNTO 0));
END COMPONENT;

COMPONENT regfile
          PORT (clk, reset, wr_en, rd_en: IN BIT;
                reg_file_in: IN BIT_VECTOR (31 DOWNTO 0);
                empty, full: OUT BIT;
                reg_file_out: OUT BIT_VECTOR (31 DOWNTO 0));
END COMPONENT;

COMPONENT clock
          PORT (reset: IN BIT; clk: INOUT BIT);
END COMPONENT;

BEGIN

   -- set carry chain and edge inputs to the ILU
   --
   and_carry_r_in <= '0';
   and_carry_l_in <= '0';
   or_carry_r_in  <= '0';
   or_carry_l_in  <= '0';
   wait_in        <= '0';
   ilu_left_edge  <= '1';

   -- iterative logic unit
   --
   ilu0: ilu
         PORT MAP (water, right_edge, rel, bef, act, aft,
                   cube0, cube1, wait_in, and_carry_r_in,
                   and_carry_l_in, or_carry_r_in, or_carry_l_in,
                   and_or, prime, request, stall, clear,
                   ilu_left_edge, clk, out_cube, and_carry_r_out,
                   and_carry_l_out, or_carry_r_out, or_carry_l_out,
                   wait_out, ilu_right_edge);

   -- control unit state machine
   --
   custm0: custm
           PORT MAP (clk, ld_ir, wait_out, reset, stall, prime,
                   clear, request, count, fetch);

-- 5 bit counter
--
counter0: counter
          PORT MAP (clk, count, ld_ir, reset, stall, cube_num);

-- water register
--
water_reg: reg16
           PORT MAP (ld_water, clk, ibus(15 DOWNTO 0), water);

-- edge register
--
edge_reg: reg16
          PORT MAP (ld_edge, clk, ibus(15 DOWNTO 0), right_edge);

-- instruction register
--
ireg: reg32
      PORT MAP (ld_ir, clk, ibus, opcode);

-- set differnet control fields from the instruction register
--
rel    <= opcode(3 DOWNTO 0);
bef    <= opcode(7 DOWNTO 4);
act    <= opcode(11 DOWNTO 8);
aft    <= opcode(15 DOWNTO 12);
and_or <= opcode(16);
prime  <= opcode(17);

-- operand 0 register
--
cube0reg: reg32
          PORT MAP (ld_cube0, clk, ibus, cube0);

-- operand 1 register
--
cube1reg: reg32
          PORT MAP (ld_cube1, clk, ibus, cube1);

-- register file (output data queue)
```

```
    --
    outq: regfile
          PORT MAP (clk, reset, count, rd_reg_file,
                    out_cube, empty, stall, dbus);

    -- clock generator
    --
    globclk: clock
             PORT MAP (reset, clk);

    clkout <= clk;

END cubecalc;
```

Chapter 9

APPENDIX 3: SYSTEMATIC LISTING OF CLASSES OF APPLICATIONS OF UNIVERSAL LOGIC MACHINE

In this chapter we will systematically present all known to us problems that can be efficiently solved on ULM.

In decision problems the solution is binary: yes/no.

In all other problems the solution is an array (clist) of cubes, with one or more cubes, representing:

- subset,
- mapping,
- Switching function,
- learned neural network,
- relation.

The ULM system includes the host with software and DEC-PERLE-1 board. DEC PERLE-1 board includes (programmable) modules referenced above.

The groups of problems will include:

1. Tautology problems,
2. Satisfiability problems,
3. Finding one solution to decision function problems,
4. Finding best solution to decision function problems,
5. Finding all solution to decision function problems,
6. Finding all solution to decision relation problems,
7. Covering problems,
8. Graph Theory problems,
9. Games and Logic Puzzles,
10. Switching function manipulation and Logic Synthesis problems,
11. Problems related to Finite State Machines and Microprogramming,
12. Problems related to High-Level Synthesis, Scheduling and Allocation,
13. Problems related to testing, verification and analysis,
14. Problems related to physical synthesis.
15. Boolean Equations and Multiple-Valued Equations,
16. Boolean Relations and Multiple-Valued Relations.
17. General Purpose Consistent Labeling Problems.
18. Data-Base Problems,
19. Cryptography Problems.
20. Morphological Image Processing Problems,
21. High-Level Computer Vision,

22. Automatic Theorem Proving,

23. Applications of Logic in Humanities,

24. Integer Programming,

25. Pseudo-Boolean Programming.

9.1 TAUTOLOGY PROBLEMS.

These are all decision problems with answer Yes/No.

POS is a Product of Sums of mv literals, single output switching function.

SOP is an Inclusive Sum of Products of mv literals, single output switching function.

ESOP is a Exclusive Sum of Products of mv literals, single output switching function.

POSP is a Product of Sums of Products mv literals, single output switching function.

SOPS is a Sum of Products of Sums mv literals, single output switching function.

Problem TAUT1.

(For All $X \in XX$) [POS(X) = 0]

Application in automatic theorem-proving.

Problem TAUT2.

(For All $X \in XX$) [SOP(X) = 1]

Application in PLA minimization (Sasao - HART).

Problem TAUT3.

(For All $X \in XX$) [ESOP(X) = 1]

Problem TAUT4.

(For All $X \in XX$) [ESOP(X) = 0]

Problem TAUT5.

(For All $X \in XX$) [POSP(X) = 1],

Problem TAUT6.

(For All $X \in XX$) [POSP(X) = 0],

Problem TAUT7.

(For All $X \in XX$) [SOPS(X) = 1],

Problem TAUT8.

(For All $X \in XX$) [SOPS(X) = 0].

Applications in automatic theorem-proving, logic synthesis, solving Boolean equations.

Problem TAUT9.

(For All $X \in XX$) [SOP(X) = SOP(X)]

Problem TAUT10.

(For All X $\in$ XX) [SOP(X) = POS(X)]

Problem TAUT11.

(For All X $\in$ XX) [POS(X) = POS(X)]

Problem TAUT12.

(For All X $\in$ XX) [ESOP(X) = ESOP(X)]

Applications in automatic theorem-proving, logic synthesis, solving Boolean equations.

9.2 SATISFIABILITY PROBLEMS.

These are all decision problems with answer Yes/No.

Problem SAT1.

(Exists X $\in$ XX) [POS(X) = 1]

Application in automatic theorem-proving.

Problem SAT2.

(Exists X $\in$ XX) [SOP(X) = 0]

Problem SAT3.

(Exists X $\in$ XX) [ESOP(X) = 1]

Problem SAT4.

(Exists X $\in$ XX) [ESOP(X) = 0]

Problem SAT5.

(Exists X $\in$ XX) [POSP(X) = 1],

Problem SAT6.

(Exists X $\in$ XX) [POSP(X) = 0],

Problem SAT7.

(Exists X $\in$ XX) [SOPS(X) = 1],

Problem SAT8.

(Exists X $\in$ XX) [SOPS(X) = 0],

Applications in automatic theorem-proving, logic synthesis, solving Boolean equations.

Problem SAT9.

(Exists X $\in$ XX) [SOP(X) = SOP(X)]

Problem SAT10.

(Exists X $\in$ XX) [SOP(X) = POS(X)]

Problem SAT11.

(Exists X $\in$ XX) [POS(X) = POS(X)]

Problem SAT12.

(Exists X $\in$ XX) [ESOP(X) = ESOP(X)]

Applications in automatic theorem-proving, logic synthesis, solving Boolean equations.

9.3 FIND SATISFIABLE SOLUTION TO DECISION FUNCTION.

This is to find one solution X that satisfies all constraints (i.e. the function).

Problem SATSOL1.

(Find_one X $\in$ XX) [POS(X) = 1]

Application in automatic theorem-proving.

Problem SATSOL2.

(Find_one X $\in$ XX) [SOP(X) = 0]

Problem SATSOL3.

(Find_one X $\in$ XX) [ESOP(X) = 1]

Problem SATSOL4.

(Find_one X $\in$ XX) [ESOP(X) = 0]

Problem SATSOL5.

(Find_one X $\in$ XX) [POSP(X) = 1],

Problem SATSOL6.

(Find_one X $\in$ XX) [POSP(X) = 0],

Problem SATSOL7.

(Find_one X $\in$ XX) [SOPS(X) = 1],

Problem SATSOL8.

(Find_one X $\in$ XX) [SOPS(X) = 0],

Applications in automatic theorem-proving, logic synthesis, solving Boolean equations.

Problem SATSOL9.

(Find_one X $\in$ XX) [SOP(X) = SOP(X)]

Problem SATSOL10.

(Find_one X $\in$ XX) [SOP(X) = POS(X)]

Problem SATSOL11.

(Find_one X $\in$ XX) [POS(X) = POS(X)]

Problem SATSOL12.

(Find_one X $\in$ XX) [ESOP(X) = ESOP(X)]

Applications in automatic theorem-proving, logic synthesis, solving Boolean equations.

9.4 FIND BEST SATISFIABLE SOLUTION TO DECISION FUNCTION.

This is to find one solution X that satisfies all constraints, and is best with respect to some cost function COST(X).

Problem BESTSOL1.

(Find_best X $\in$ XX) [POS(X) = 1]

Application in automatic theorem-proving.
Problem BESTSOL2.
(Find_best X $\in$ XX) [SOP(X) = 0]
Problem BESTSOL3.
(Find_best X $\in$ XX) [ESOP(X) = 1]
Problem BESTSOL4.
(Find_best X $\in$ XX) [ESOP(X) = 0]
Problem BESTSOL5.
(Find_one X $\in$ XX) [POSP(X) = 1],
Problem BESTSOL6.
(Find_one X $\in$ XX) [POSP(X) = 0],
Problem BESTSOL7.
(Find_best X $\in$ XX) [SOPS(X) = 1],
Problem BESTSOL8.
(Find_best X $\in$ XX) [SOPS(X) = 0],
Applications in automatic theorem-proving, logic synthesis, solving Boolean equations.
Problem BESTSOL9.
(Find_best X $\in$ XX) [SOP(X) = SOP(X)]
Problem BESTSOL10.
(Find_best X $\in$ XX) [SOP(X) = POS(X)]
Problem BESTSOL11.
(Find_best X $\in$ XX) [POS(X) = POS(X)]
Problem BESTSOL12.
(Find_best X $\in$ XX) [ESOP(X) = ESOP(X)]
Applications in automatic theorem-proving, logic synthesis, solving Boolean equations.

9.5 FIND ALL SATISFIABLE SOLUTIONS TO DECISION FUNCTION.

This is to find one solution X that satisfies all constraints, and is best with respect to some cost function COST(X).
Problem ALLSOL1.
(Find_all X $\in$ XX) [POS(X) = 1]
Application in automatic theorem-proving.
Problem ALLSOL2.
(Find_all X $\in$ XX) [SOP(X) = 0]
Problem ALLSOL3.
(Find_all X $\in$ XX) [ESOP(X) = 1]
Problem ALLSOL4.
(Find_all X $\in$ XX) [ESOP(X) = 0]

Problem ALLSOL5.
(Find_one X $\in$ XX) [POSP(X) = 1],
Problem ALLSOL6.
(Find_one X $\in$ XX) [POSP(X) = 0],
Problem ALLSOL7.
(Find_all X $\in$ XX) [SOPS(X) = 1],
Problem ALLSOL8.
(Find_all X $\in$ XX) [SOPS(X) = 0],
Applications in automatic theorem-proving, logic synthesis, solving Boolean equations.
Problem ALLSOL9.
(Find_all X $\in$ XX) [SOP(X) = SOP(X)]
Problem ALLSOL10.
(Find_all X $\in$ XX) [SOP(X) = POS(X)]
Problem ALLSOL11.
(Find_all X $\in$ XX) [POS(X) = POS(X)]
Problem ALLSOL12.
(Find_all X $\in$ XX) [ESOP(X) = ESOP(X)]
Applications in automatic theorem-proving, logic synthesis, solving Boolean equations.

9.6 FIND ALL SATISFIABLE SOLUTIONS TO DECISION RELATION.

This is to all solutions X that satisfy all constraints of a decision relation.
Problem ALLREL1.
(Find_all X $\in$ XX) [POS(X) = 1]
Application in automatic theorem-proving.
Problem ALLREL2.
(Find_all X $\in$ XX) [SOP(X) = 0]
Problem ALLREL3.
(Find_all X $\in$ XX) [ESOP(X) = 1]
Problem ALLREL4.
(Find_all X $\in$ XX) [ESOP(X) = 0]
Problem ALLREL5.
(Find_one X $\in$ XX) [POSP(X) = 1],
Problem ALLREL6.
(Find_one X $\in$ XX) [POSP(X) = 0],
Problem ALLREL7.
(Find_all X $\in$ XX) [SOPS(X) = 1],
Problem ALLREL8.

(Find_all X $\in$ XX) [SOPS(X) = 0],

Applications in automatic theorem-proving, logic synthesis, solving Boolean equations.

Problem ALLREL9.

(Find_all X $\in$ XX) [SOP(X) = SOP(X)]

Problem ALLREL10.

(Find_all X $\in$ XX) [SOP(X) = POS(X)]

Problem ALLREL11.

(Find_all X $\in$ XX) [POS(X) = POS(X)]

Problem ALLREL12.

(Find_all X $\in$ XX) [ESOP(X) = ESOP(X)]

Applications in automatic theorem-proving, logic synthesis, solving Boolean equations.

9.7 COVERING PROBLEMS.

MAXIMUM INDEPENDENT SET.

Reduce to:

MAXIMUM INDEPENDENT SET.

Reduce to:

MAXIMUM CLIQUE.

Reduce to:

SET COVERING.

Reduce to Petrick function, reduce to GPF, reduce to:

COVERING/CLOSURE.

Equivalent to Binate Covering, reduce to:

EDGE COVERING PROBLEM.

9.8 GRAPH THEORY PROBLEMS.

PROPER GRAPH COLORING.

(Maghout Method) Reduce to finding all Maximum Cliques and next Covering nodes with them.

COMPATIBLE GRAPH COLORING.

CDEC-COMPATIBLE GRAPH COLORING.

EDGE COLORING PROBLEM.

9.9 GAMES AND LOGIC PUZZLES.

Games and Logic Puzzles are described in [?, ?, ?, ?, ?].

9.10 SWITCHING FUNCTION MANIPULATION AND LOGIC SYNTHESIS PROBLEMS.

Switching function manipulation include: Complementation, sharp, intersection, generation of all primes, supercube, consensus, etc. SOP/POS and POS/SOP transformations,

Logic Synthesis problems include PAL/PLA minimization, two-level, three-level, four-level logic minimizatin, ESOP, GRM, CRMP, minimization.
SATISFIABILITY Every NP-hard problem can be reduced to it [?, ?, ?]. Many recent research papers from U.C. Berkeley and U. Colorado are based on tautology. Approach based on unification is presented in [?]. Very many other reductions to operations efficiently realized in our machine can be found in recent DAC, ICCD, ICCAD, ISMVL, ISCAS, ICASSP conferences and Logic Synthesis workshops, and there is even no space here to list them all.

There are two representation methods for switching functions used in logic synthesis programs: "(Binary or Multiple-Valued) cube calculus" and "(Binary or Multiple-Valued) Decision Diagrams". In this paper only the first one will be discussed, however the new variant of our machine (TP) allows to deal also with data other than the "multiple-valued input cubes" presented here, and includes the MDDs as well.
COMPLEMENTATION OF BOOLEAN FUNCTION.
"SOP-TO-POS" AND "POS-TO-SOP" TRANSFORMATIONS.

9.11 PROBLEMS RELATED TO FINITE STATE MACHINES AND MICROPROGRAMMING.

Problems related to Finite State Machines and Microprogramming include: Minimization of FSM, Concurrent State and Input Minimization of FSM, Concurrent State Assignment and State Minimization of FSM, State Assignment of FSM, Decomposition of FSM, Splitting of states of FSM,

9.12 PROBLEMS RELATED TO HIGH-LEVEL SYNTHESIS, SCHEDULING AND ALLOCATION.

Problems related to High-Level Synthesis, Scheduling and Allocation include: Data Path Scheduling, Data Path Allocation, Memory Scheduling, Memory Allocation, Multi_processor Scheduling, Multi-processor Allocation,

9.13 PROBLEMS RELATED TO TESTING, VERIFICATION AND ANALYSIS.

Problems related to testing, verification and analysis include:

9.14 PROBLEMS RELATED TO PHYSICAL SYNTHESIS.

Problems related to physical synthesis include:

9.15 BOOLEAN EQUATIONS AND MULTIPLE-VALUED EQUATIONS.

Boolean Equations and Multiple-Valued Equations include:
SOLVING BOOLEAN EQUATIONS.
SOLVING MULTIPLE-VALUED EQUATIONS.

9.16 BOOLEAN RELATIONS AND MULTIPLE-VALUED RELATIONS.

Boolean Relations and Multiple-Valued Relations include:
Boolean relation of inputs x_1, x_2, x_3, x_4, x_5 and outputs y_1, y_2, y_3, y_4 is represented in the form of two cubes:
10-11-01-01-11=1011
01-11-01-01-11=0110
where = separates input from output variables. This means:
$\overline{x_1}$ x_3 x_4 -¿ 1011, 0110
or
$\overline{x_1}$ x_3 x_4 -¿ y_1 $\overline{y_2}$ y_3 y_4, $\overline{y_1}$ y_2 y_3 $\overline{y_4}$.
Another notation may be:
RELATION ($\overline{x_1}$, x_3 , x_4, y_1, $\overline{y_2}$, y_3, y_4)
RELATION ($\overline{x_1}$, x_3, x_4 , $\overline{y_1}$, y_2, y_3, $\overline{y_4}$.)
Generalization of the above notation to multiple-valued input and multiple-valued output variables looks like this:
1000-1111-10101-11001-111=1011-0101
1000-1111-10101-11001-111=1000-1110
1000-1111-10101-11001-111=1011-1010
SOLVING BOOLEAN RELATIONS.
SOLVING MULTIPLE-VALUED RELATIONS.

9.17 GENERAL PURPOSE CONSISTENT LABELING PROBLEMS.

General Purpose Consistent Labeling Problems include:
ENUMERATE CARTESIAN PRODUCT WITH CONSTRAINTS.

This is morphological methods of solving problems [?, ?].
SOLVING GENERAL-PURPOSE CONSISTENT LABELING PROBLEM.

See problems solved by MULT-II, Multicomp, all versions.

1. Use Multicomp,
2. Use CCM model,
3. Use CCM instructions,
4. use pipelining, parallelism.
5. use Multicomp problem description language,
6. adapt old papers: multicomp, book AI, analogy, relational languages.

9.18 DATA-BASE PROBLEMS.

Data-Base Problems include: Ulug [417, 417, ?] proposed an extended

9.19 CRYPTOGRAPHY PROBLEMS.

Cryptography Problems include:

9.20 MORPHOLOGICAL IMAGE PROCESSING PROBLEMS.

Morphological Image Processing Problems include:

9.21 HIGH-LEVEL COMPUTER VISION PROBLEMS.

High-Level Computer Vision problems include:

9.22 AUTOMATIC THEOREM PROVING.

Automatic Theorem Proving includes: resolution-based theorem proving, resolution/unification-based theorem proving, theorem proving in higher-order logics, Analogy-based reasoning.

9.23 APPLICATIONS OF LOGIC IN HUMANITIES.

Applications of Logic in Humanities include:

9.24 INTEGER PROGRAMMING.

Integer Programming includes:

9.25 PSEUDO-BOOLEAN PROGRAMMING.

Pseudo-Boolean Programming includes [?].

9.26 APPLICATIONS REQUIRING LONG WORD ARITHMETIC OPERATIONS.

9.27 FUZZY LOGIC

Fuzzy logic. Although it is just a one more multiple-valued logic, we pay special attention to it since it finds recently several practical applications.

such as "fuzzy chips" [?, ?, ?] find use.

ROUGH SETS

STACK FILTERS FOR IMAGE PROCESSING.

NEURAL NETWORKS.

LOGIC PROGRAMMING.

OPERATIONS IN LINEAR SYSTEMS

Modulo-p fields for encoding, testing, etc.

CAM-BASED

In paper [440] a CAM-based architecture has been proposed that is able to solve by exhaustive search the "Traveling Salesman", "Set Covering", and several other combinatorial problems close to those of logic synthesis.

Kida thesis.

RHINE book.

LOGIC SIMULATION

Several machines for logic simulation have been successfully build and are used in practice in USA and Japan.

CONTROLLERS TO EVALUATE BDD DIAGRAMS.

paper from IEEE Trans. on Industrial Electronics.

It is also useful in programs using the other efficient representation of Boolean functions: Binary Decision Diagrams and Multiple-valued Decision Diagrams. Multiple-valued Cube Calculus seems then to be one of the most general currently known

internal representation of data in propositional and predicate logic, logic synthesis, logic programming, logic simulation and sequential evaluation of combinational logic, data-bases, image processing and several areas of AI and problem-solving. Methods based on it have been successfully applied in practically realized products.
EQUIVALENCE TO TURING MACHINES, COMPUTERS, PURE LISP.

our machine should be a general-purpose computer, it means it must be Turing-machine equivalent and must include the most common and all necessary instructions of general-purpose computers.
APPLICATIONS OF GAPP, TRANSPUTER, THINKING MACHINES.

the machine must be a hardware accelerator board for PC, but at the same time the chip must be also a building block of various massively parallel architectures (such as it is the case with GAPP or Transputer). Thinking Machines Connection Machine. Data-Programming.
APPLICATIONS BASED ON TERNARY AND OTHRE N-COUNTING.

two-bit IT cell to represent a part of any mv-literal, flexible number of values in literals, and iterative circuit of FSMs with information flowing in two directions. This was also the first machine which emphasized cube calculus operations, and the first which did not use the concept of counting in any way: it based algorithms such as tautology on cube operations such as sharp, intersection or crosslink.
VARIOUS PROBLEM SOLVING

What all "logic machines" have in common is that they use special hardware to do some kind of processing of Boolean or other switching functions. These tasks include: evaluation, Boolean operations such as intersection or complementation, checking for tautology or satisfiability, verification, solving logic equations, optimizing decision functions (like Petrick or its generalizations), performing resolution and unification, operating on binary images and quadtrees represented as switching functions, stack filters, data-base operators, inferring facts in logic, and many other.

Many well-known problem reductions exist that support our approach both theoretically (reductions of NP-complete problems [105]) and practically (several CAD and Operations Research algorithms).

It was observed that all NP-complete problems can be reduced to few generic problems such as graph coloring, maximum clique, shortest path, SOP to POS transformation, etc. Those in turn are reducible to basic binary logic problems such as satisfiability [?]. Solving the Satisfiability problem can be next reduced to using some subset of the cube calculus operations such as sharp, absorption, intersection and other.
ROBOTICS
OPERATIONS RESEARCH
The binary "cube calculus" [?] has been extended for a logic with multiple-valued inputs [?, ?, ?, ?, ?] by Sasao, and is called a *positional cube notation.* This calculus has been used for many two-, three- and many level Boolean minimizers, tautology and satisfiability checkers, verifiers, programs for complementation of Boolean functions, synthesis of mixed and fixed generalized Reed-Muller forms, generation of prime, minimal and disjoint implicants, spectral transforms (Walsh, Reed-Muller, Arithmetic), and many other [?, 73, 1, 145, 169, ?, 355, 385, ?, ?, ?, ?, ?, ?, ?].
REAL-TIME AI Implementing a hardware processor to operate in this calculus will have wide applications, including real-time AI and optimization, where recent devices

9.28 THE TUPLE PROCESSOR.

In addition to operations on sets and numbers, known from CCM-2, our interest in solving discrete combinatorial prolems is in *finite formal systems* that have a finite (and from practical point of view - also limited) number of symbols as well as a limited number of operations. We assume that all operations on symbols can be decomposed to simple operations, which in turn can be described as tables (two-dimensional matrices). We are already familiar with such matrices describing sharp, consensus or other binary operators. This assumption allows to encode efficiently symbols as binary strings, and implement efficiently the matrices using ROMs, RAMs, CAMs, PLDs or other standard binary logic. Such systems will be called Limited Finite Systems (LFS). Let us observe that integer arithmetics is not an LFS since truth-table description of the addition operation is impractical for hardware realization, while (assuming small value of integer n) the modulo n Galois Field is an LFS, since such matrix description can be used to design the logic of modulo n operations.

9.29 MULTI-DIMENSIONAL SOLID ALGEBRAS.

MATRIX LOGIC

9.30 TRULY MULTIPLE-VALUED LOGICS.

9.31 TUPPLE PROCESSOR: AREAS OF INTEREST AND FUTURE APPLICATIONS

Each word of TP will represent

1. a number (as in CCM-2),
2. a pair of numbers (an interval),
3. a set (as in CCM-2),
4. a symbol.

Each symbol can have various meanings, it can be simple or complex. Simple symbol can be associated with a symbol of multiple-valued algebra, modal algebra, or any discrete system (such as consistent labeling), in which the number of such systems is limited. In TP, which will be constructed using FPGA technology, particularly XILINX, we take into account the limits of technology. Since the cell of Xilinx has five inputs and can realize any cell of five variables and some functions of 6 variables, we restrict ourselves to symbols on three bits. This allows to use one or two cells for all two argument operations in algebras having up to 8 values ($2^3 = 8$). This is sufficient for most known algebras, except for 16-valued algebras used in simulation. However, we can still use complex symbols composed of simple symbols in the same way as the set operations in literals are composed in CCM-2 from binary operations in ITs.

Our machine introduces the dynamically modifiable size; the operations on all words of an N-tuple will be done in parallel.

Our goal of introducing this computer will be achieved step-by-step by introducing a number of simpler architectures and algorithms for classes of problems. We will show and discuss all steps leading to our ultimate goal. Particularly, we will first have to understand the Cube Calculus Machine [?] and next we will generalize it.

The TP is a comprehensive generalization of ideas and architectures used in several areas:

1. *Multiple-valued input "cube calculus",* an algebraical model popularly used to process and minimize Boolean functions. The TP uses a "positional cube representation" which finds many applications in logic synthesis and combinatorial problem solving. It supports all operations including sharp, consensus, supercube and crosslink, as well as many new operations.

2. Multiple-valued notations for *truly multiple-valued* (both input and output) logics. Operations of many multiple-valued logics such as the *modal logic, matrix logic, quantum logic* and other will be realized.

ASSOCIATIVE PROCESSING

Associative N-tuples used in Artificial Intelligence and Image Processing to represent and process knowledge.

DATA-BASE ALGEBRAS.

Data-Base Algebras used for hardware realization of relational data bases.

DIGITAL SIMULATION ALGEBRAS.

Digital Simulation algebras used to simulate computers and digital systems in special hardware accelerators [].

MECHANISMS OF HARDWARE REALIZATIONS OF PROLOG, LISP AND EQUATIONAL LOGIC.

Search, evaluation and inference mechanisms of *relational, functional and logical languages* such as Lisp, Prolog, and Equational Logic.

MECHANIZMS TO SOLVE COMBINATORIAL PROBLEMS.

General formalisms to represent and solve combinatorial problems [?, ?, ?, ?, ?].

ARCHITECTURES FOR COMPUTER VISION.

Formalisms and architectures for *computer image processing and image recognition.*

SPECTRAL TRANSFORMS.

Spectral transforms (Fourier, Walsh, Haar, Arithmetic, Adding, Hough) and signal processing ideas (convolution, digital filters).

NEURAL-LIKE LEARNING NETWORKS.

Neural Nets, Genetic Programming, Simulated Annealing, Cellular neural networks.

Neural-like learning networks and recognizers. Let us observe that the taught binary neural network is the same as multi-input, multi-output Boolan function. Learning of the network is the process of constructing this functions (possibly by modifying some initial function). This can be generalized to arbitrary discrete neural nets, which correspond to multiple-valued input, multiple-valued output switching functions. It is possible that using the cube calculus operations the learning process will be faster, think for instance about the n-input EXOR function, which is very hard to learn by neural nets, and is trivial to learn using for instance crosslink operator.

PROGRAMMABLE XILINX ARCHITECTURES. DYNAMIC PROCESSORS.

Programmable architectures like those base od on Xilink chips - [?]. Svetlana Kartashev.

TREE SEARCHING ARCHITECTURES

Paper how to map tree searching to pipes (Information Processing Letters?).

PROBLEMS GOOD FOR SYSTOLIC AND PIPELINED ARCHITECTURES.

Transputer, GAPP, systolic processors, iWARP and DSP architectures (AIM Sharp).

USING MV CUBE CALCULUS TO SOLVE COMBINATORIAL PROBLEMS.

Brayton's approach to using mv CC to solve combinatorial problems.

9.32 CELLULAR AUTOMATA.

Cellular Automata are a class of nonlinear mathematical systems characterized by discretness (in space, time, and state values), determinism, and local interaction.

There are some narrower classes and some extensions discussed in the literature.

There can be:

1. one-dimensional, two-dimensional, more dimensional.
2. regular, irregular.
3. If regular, triangle, rectangle, hexagon.
4. linear, non-linear.

Cellular automata have attracted substantial interest in recent years. Theyt are used as simple models for complex physical and biological phenomena [119, 433].

9.33 COMPRESSION AND DECOMPRESSION. IMAGE PROCESSING.

Our efforts will be on developing system/logic methods that will allow for fast prototyping of any compression/decompression algorithms. Our approach is being tested on several algorithms of various nature. This should give good chance that it will also work on proprietary algorithms of our customers.

We are interested in data, speech and image compression, but as an example we will discuss the last one only.

Discrete Cosine Transform (DCT) is used in many compression standards. In particular 8 * 8 DCT is now used widely in image coding, and it will be perhaps used in HDTV. Recently several new systolic, and pipelined solutions to 8 * 8 DCT have been proposed based on very regular decompositions. Some of them are multiplier-free. Some use distributed arithmetic. There are solutions that use Fast Fourier Transform (FFT) or Fast Walsh Transform (FWT). Characteristic to all those solutions are the following properties, advantageous from the point of view of our approach.

1. Very high regularity of blocks. Short connections between blocks. Regular patterns of connections.

2. Multi-level regular hierarchy of regular blocks.

3. Much use of fast RAM memory.

4. Use of adders, shifters, counters, ROMs, EXOR gates.

5. Variants investigate trade-offs between time and space, which is, speed and cost.

6. It results from our preliminary calculations that about ten boards, each of them 6 * 6 CLI chips should allow for realization of even the most sophisticated hardware algorithms for HDTV pixel rates, even if progressive scan (144 MSamples/s) is neded.

7. Several architectures are very similar for DCT, sine, Fourier, Walsh, Hadamard, and other transforms. Similar aproaches can be used also for inverse transforms. This should be a proof of generality of the approach.

8. **The techniques are very good for prefilter and postfilter operations used in the entire image coding system. In essence, any kind of digital filter is easily realizable.**

9. **Convolution is easily realizable so as many algorithms based on it.**