A Quaternary Decision Diagram Machine and the Optimization of Its Code

Tsutomu Sasao [1], Hiroki Nakahara [1], Munehiro Matsuura[1]
Yoshifumi Kawamura [2], Jon T. Butler [3]
[1] Kyushu Institute of Technology, Iizuka 820-8502, Japan
[2] Renesas Technology Corp., Tokyo 100-0004, Japan
[3] Naval Postgraduate School, Monterey, CA 93943-5121, USA

Abstract

We show the advantage of Quarternary Decision Diagrams (QDDs) in representing and evaluating logic functions. That is, we show how QDDs are used to implement QDD machines, which yield high-speed implementations. We compare QDD machines with binary decision diagram (BDD) machines, and show a speed improvement of 1.28-2.02 times when QDDs are chosen. We consider 1-and 2-address BDD machines, and 3- and 4-address QDD machines, and we show a method to minimize the number of instructions.

1 Introduction

Branching program machines for BDDs have been used in control applications [2, 5, 7, 6]. Fast response is especially important in control applications in which there are usually hundreds of inputs. For such applications, a general purpose microprocessor (MPU) cannot meet the speed requirements. A branching program machine can be several times faster than an MPU: An ordinary MPU requires two or three machine instructions to read and test one input variable, while the branching program machine requires just one instruction [3].

In this paper, we present a Quarternary Decision Diagram (QDD) to implement a branching program machine. Although the QDD machine requires longer instruction words than the BDD machine, the QDD machine is $1.3-2.0$ times faster than the corresponding BDD machine. In the past, when the price of memory was high, 16-bit controllers were popular [13, 25]. However, nowadays, the price of memory is lower, and a 32-bit or wider architecture is often used to increase the performance of controllers. So, in this paper, we show a method to increase the performance by increasing the number of bits in a word.

The rest of this paper is organized as follows: Section 2 introduces a method to represent multi-output logic functions by multi-valued decision diagrams. Section 3 introduces branching program machines: It introduces both a 4-address QDD machine and a 3-address QDD machine. The 3-address QDD machine requires less memory than the 4-address QDD machine. Section 4 shows an optimization problem of codes for 3-address QDD machines. Section 5 shows the experimental results. And finally, Section 6 concludes the paper.

2 Representation of Multiple-Output Functions

2.1 Multi-Valued Decision Diagrams

An arbitrary n variable logic function can be represented by a binary decision diagram (BDD). Evaluation of a BDD requires n table look-ups. Fig. 2.1 shows an example of an MTBDD (multi-terminal binary decision diagram). In this case, many outputs can be evaluated at the same time. To further speed up the evaluation, a multiple-valued decision diagram (MDD) is used. In the MDD(k), k variables are grouped to form a 2^k-valued **super variable**. To evaluate the MDD(k), we need at most $\lceil \frac{n}{k} \rceil$ table look-ups [15, 19]. When the function is represented by an MDD(k), the evaluation of a logic function can be k times faster than the corresponding BDD [1]. Thus, a larger k yields a faster evaluation of the MDD(k). Unfortunately, the size of memory to represent a node for an MDD(k) is proportional to 2^k, as shown in Fig. 2.2. For many benchmark functions, the total size of the memory for an MDD(k) achieves its minimum when $k = 2$ [19]. Therefore, in logic evaluation, MDD(2)s are more suitable than BDDs. Since nodes in an MDD(2) have 4 branches, it is termed a **Quarternary Decision Diagram** (**QDD**).

[1]This is true only when the MDD(k) and the BDD are quasi reduced.

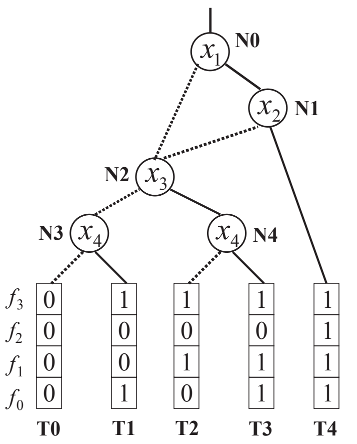
Figure 2.1. Example of an MTBDD.

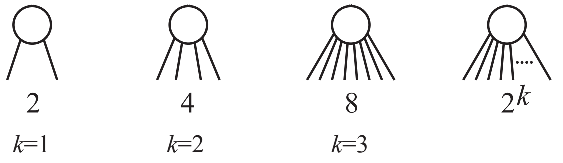
Figure 2.2. Nodes for MDD(k).

2.2 Optimization of MDDs

In an MDD(k), the evaluation of an n-variable logic function can be done by at most $\lceil \frac{n}{k} \rceil$ table look-ups. So, the major problem is the minimization of the number of nodes. In general, it is not so easy to obtain an MDD(k) with the minimum number of nodes. The following heuristic method is used to obtain near minimal MDDs:

1. Minimize the number of nodes of the BDD by a heuristic method [21].
2. Partition the input variables to generate an MDD(k) [22].

Fig. 2.3 shows an example of a conversion from a BDD into an MDD(2). In the above MDDs, we assume each group of variables has the same size. Such MDDs are **homogeneous MDDs**. When the groups have different sizes, the MDD is a **heterogeneous MDD**. For simplicity, in this paper, we consider only homogeneous MDDs.

3 Branching Program Machine

Special machines to evaluate MDDs have been developed [8, 9, 10]. Unfortunately, they are unsuitable for practical applications. Here, we consider a machine whose architecture is well-suited for evaluating MDDs, but is easily programmed.

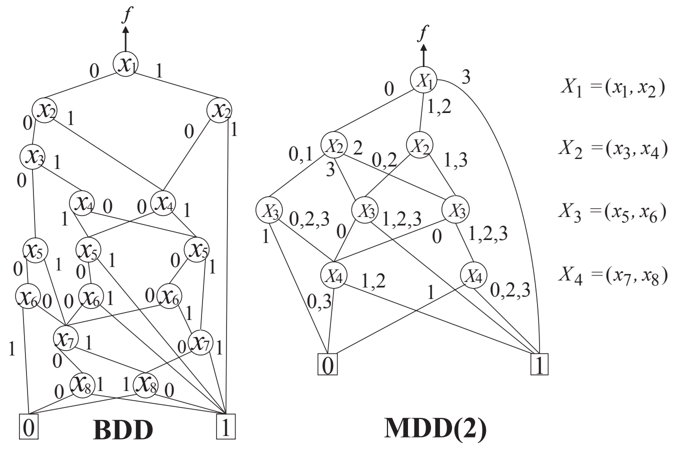
Figure 2.3. Conversion of BDD to MDD(2).

3.1 2-Address BDD Machine

A branching program for BDDs uses only two kinds of instructions:

```
B_Branch (ADDR0, ADDR1), INDEX
Output DATA, and GOTO ADDR.
```

The first one is the binary branch instruction that is similar to the computed GOTO statement of the FORTRAN language: If the value of the variable specified by INDEX is equal to 0, then go to ADDR0, otherwise goto ADDR1. The second one performs the output operation followed by an unconditional GOTO operation.

Example 3.1 *Consider the MTBDD shown in Fig. 2.1. The following code evaluates the MTBDD:*

```
N0:B_Branch(N2,N1), X1
N1:B_Branch(N2,T4), X2
N2:B_Branch(N3,N4), X3
N3:B_Branch(T0,T1), X4
N4:B_Branch(T2,T3), X4
T0:Output 0, and GOTO N0
T1:Output 9, and GOTO N0
T2:Output 10, and GOTO N0
T3:Output 11, and GOTO N0
T4:Output 15, and GOTO N0
```

In this example, DATA in Output DATA is the decimal equivalent of the function output values expressed in binary as f_3, f_2, f_1, f_0. *(End of Example)*

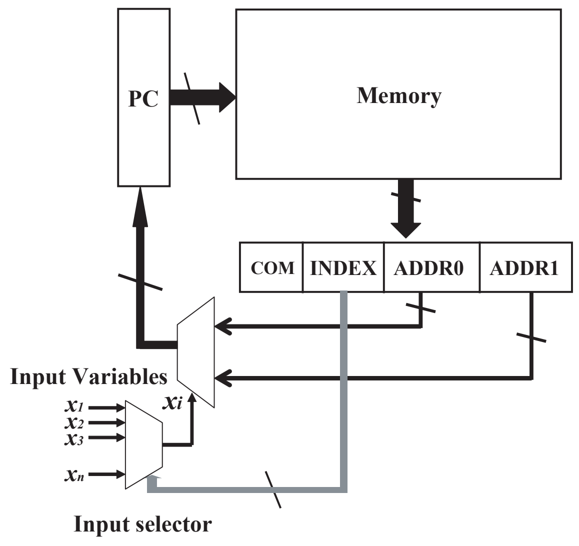

Figure 3.1. 2-address BDD Machine.

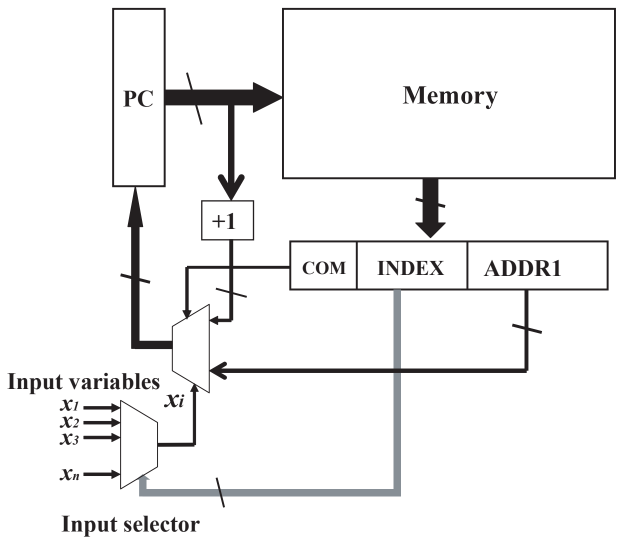

Figure 3.2. 1-address BDD Machine.

Fig. 3.1 shows the architecture of the 2-address BDD machine, where only the circuit for the branching operation is shown. The first field of the branching instruction specifies the branch command. The second field, INDEX, specifies the index i of the input variables x_i. It determines which variables to select. The input selector in Fig. 3.1 produces the value of the variable x_i selecting the next branch address. When $x_i = 0$, ADDR0 is selected. Otherwise, ADDR1 is selected. The selected address is then loaded into the program counter (PC). In this way, the next address is specified. To reduce the width of the instruction words, 1-address BDD machines shown in Fig. 3.2 have been developed [2, 6, 25, 13]. In this case, when the value specified by INDEX is 1, the machine works similarly to the case of the 2-address BDD machine. Otherwise, the content of the program counter (PC) is incremented by one, to access the next address. In this case, the size of the instruction word is reduced, but unconditional GOTO instructions are necessary, as shown later.

3.2 4-Address QDD Machine

By evaluating two binary variables and by increasing the number of branch addresses to four, we have a branch instruction for a 4-address QDD machine. Since it evaluates two binary variables at a time, it can reduce the evaluation time to half that of the 2-address BDD machine.

A branching program for 4-address QDD machines consists of two kind of instructions:

```
Q_Branch(ADDR0,ADDR1,ADDR2,ADDR3),INDEX
Output DATA, and GOTO ADDR
```

Fig. 3.4 shows the format for the branch instruction. Fig. 3.3 shows the architecture of the 4-address QDD machine, where only the circuit for the branching operation is shown. The first field of the branching instruction specifies the branch command. The second field, INDEX, specifies the index i of the input variable X_i. It determines which variables to select. In the case of a QDD, two consecutive binary variables are selected at a time. The input selector shown in Fig. 3.3 producesX_i. The upper multiplexer selects the variable. When $X_i = (0,0)$, ADDR0 is selected; when $X_i = (0,1)$, ADDR1 is selected; when $X_i = (1,0)$, ADDR2 is selected; and when $X_i = (1,1)$, ADDR3 is selected. The selected address is then loaded into the program counter (PC). In this way, the next address is specified as a function of INDEX i and the input variable X_i. Note that this instruction requires a rather long word, which would be expensive for embedded applications.

Fig. 3.5 shows the format for the output instruction. The left field specifies the instruction type: Output. The middle field contains the address to which this program should jump. The right field is the output value, as shown at the bottom of the QDD.

3.3 3-Address QDD Machine

Since the 4-address QDD instruction requires a long word, we developed a 3-address QDD machine. The branch instruction for the 3-address QDD machine contains only three address fields. For example, consider the instruction shown in Fig. 3.6. This instruction is symbolically denoted by

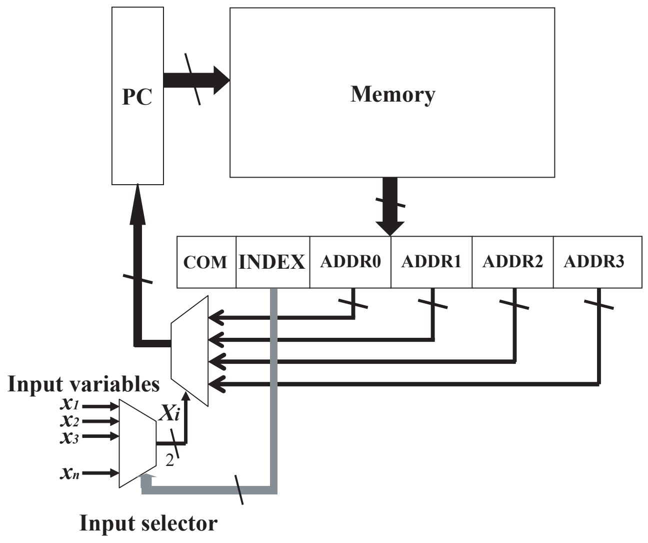

Figure 3.3. 4-address QDD Machine.

Branch	Index	ADDR0	ADDR1	ADDR2	ADDR3

Figure 3.4. Branch Instruction for 4-address QDD Machine.

```
Q_Branch(+1,ADDR1,ADDR2,ADDR3),INDEX.
```

In this instruction, ADDR1, ADDR2, and ADDR3 are specified, but ADDR0 is missing. ADDR0 is replaced by "+1", which shows the next address of the current instruction. This instruction performs the following operations:

- Let i be the value specified by INDEX. If $(i = 0)$ then goto the next address of the current instruction, else goto ADDRi.

Lemma 3.1 *An arbitrary QDD can be evaluated by a program consisting of the following instructions:*

```
Q_Branch(+1,ADDR1,ADDR2,ADDR3),INDEX
GOTO ADDR
Output DATA, and GOTO ADDR
```

For example, the instruction for the 4-address QDD machine

Output	Address	Output Values

Figure 3.5. Output Instruction for a QDD Machine.

Branch0	Index	ADDR1	ADDR2	ADDR3

Figure 3.6. Branch Instruction for a 3-address QDD Machine.

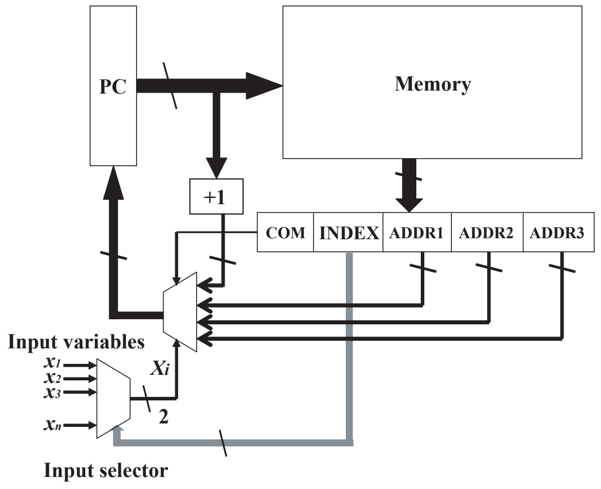

Figure 3.7. 3-address QDD Machine.

```
Q_Branch(ADDR0,ADDR1,ADDR2,ADDR3),INDEX
```

can be simulated by the pair of instructions:

```
Q_Branch(+1,ADDR1,ADDR2,ADDR3),INDEX
GOTO ADDR0
```

Note that the last instruction is an **unconditional GOTO statement**. As shown in the next section, the number of unconditional GOTO statements can be minimized by an optimization algorithm. Fig. 3.7 shows the architecture of the 3-address QDD machine, where only the circuit for branching operations is shown. Consider the instruction in Fig. 3.6. When the value specified by INDEX and the input variables is non-zero, the machine works similarly to the case of the 4-address QDD machine. When the value specified by INDEX and the input variables is equal to 0, the content of the program counter (PC) is incremented by one, to access the next address.

In the real system, we use four types of branch instructions shown in Fig. 3.8 . To distinguish four branch instructions, we use two additional bits in the instruction field. However, as shown in the experimental results, by using four branch instructions, we can reduce the number of instructions and the total bit size. So, the cost of these extra bits is fully compensated.

Branch0	**Index**	**ADDR1**	**ADDR2**	**ADDR3**
Branch1	**Index**	**ADDR0**	**ADDR2**	**ADDR3**
Branch2	**Index**	**ADDR0**	**ADDR1**	**ADDR3**
Branch3	**Index**	**ADDR0**	**ADDR1**	**ADDR2**

Figure 3.8. Four Types of Branch Instructions for 3-address QDD Machine.

4 Optimization of Codes for QDD Machines

In this section, we consider a method to reduce the number of instructions for QDD machines.

Definition 4.1 *Given the QDD and an order of the input variables (e.g. $x_1, x_2, \ldots$, and x_n), the code size CSIZE is the number of instructions needed to compute the Decision diagram on a given machine. Let 4aQDDM denote a 4-address QDD machine, and let 3aQDDM denote a 3-address QDD machine.*

Lemma 4.1 *Let N_N be the number of non-terminal nodes, and let N_T be the number of terminal nodes in a QDD. We have the following relation:*

$$CSIZE(4aQDDM) = N_N + N_T. \quad (4.1)$$

(Proof) In a 4-address QDD machine, a non-terminal node is represented by a branch instruction, and a terminal node is represented by an output instruction. (Q.E.D.)

Lemma 4.2 *Let N_N be the number of non-terminal nodes and let N_T be the number of terminal nodes in a QDD. Let N_U be the number of unconditional GOTO statements that are not part of output statements. Then, we have the following relations:*

$$CSIZE(3aQDDM) = N_U + N_N + N_T \quad (4.2)$$

$$0 \leq N_U \leq N_N \quad (4.3)$$

(Proof) In a 3-address QDD machine, a non-terminal node is represented by either a branch instruction or a pair consisting of a branch instruction and an unconditional GOTO statement. Also, a terminal node is represented by an output instruction. Thus, the number of unconditional GOTO statements is at most the number of non-terminal nodes. (Q.E.D.)

In the case of a 4-address QDD machine, there is no code optimization problem, i.e., the instructions can be generated in any order. However, in the case of a 3-address QDD machine, the length of the program depends on the order of instructions.

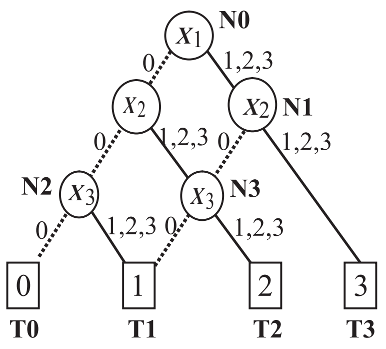

Figure 4.1. QDD for Example Function.

Example 4.1 *Consider the QDD shown in Fig. 4.1. It has five non-terminal nodes, and four terminal nodes. When the code is generated in the breadth-first order, i.e., in the order of X_1, X_2 and X_3, we have the following:*

```
/** Code with Unconditional GOTO **/
N0:Q_Branch(+1,N1,N1,N1),X1
   Q_Branch(+1,N3,N3,N3),X2
   GOTO N2
N1:Q_Branch(+1,T3,T3,T3),X2
   GOTO N3
N2:Q_Branch(+1,T1,T1,T1),X3
   GOTO T0
N3:Q_Branch(+1,T2,T2,T2),X3
   GOTO T1
T0:Output 0, and GOTO N0
T1:Output 1, and GOTO N0
T2:Output 2, and GOTO N0
T3:Output 3, and GOTO N0
```

Note that, the above program has four unconditional GOTO statements that are not part of output statements. However, when the code is generated in the depth-first order, it has no unconditional GOTO statements that are not part of output statements.:

```
/** Code without Unconditional GOTO **/
N0:Q_Branch(+1,N1,N1,N1),X1
   Q_Branch(+1,N3,N3,N3),X2
   Q_Branch(+1,T1,T1,T1),X3
T0:Output 0, and GOTO N0
N1:Q_Branch(+1,T3,T3,T3),X2
N3:Q_Branch(+1,T2,T2,T2),X3
T1:Output 1, and GOTO N0
T2:Output 2, and GOTO N0
T3:Output 3, and GOTO N0
```

Note that the first four instructions correspond to the left-most path from the root node to the terminal node T0.

The next three instructions correspond to the path from the the node N1, the node N3, and to the terminal node T1. (End of Example)

The code optimization problem for a 3-address QDD machine can be reduced to a graph covering problem as follows:

Definition 4.2 *A* **path cover** *of a QDD is a set of paths such that every node in the QDD belongs to exactly one path. A* **minimal path cover** *is a path cover with the fewest paths. A path in a QDD can consist of just one node.*

Theorem 4.1 *An optimal code for a 3-address QDD machine corresponds to a minimal disjoint path cover of the QDD.*

(Proof) A path in a QDD corresponds to a sequence of Q_Branch instructions followed by an output instruction. A sequence of Q_Branch instructions without an output instruction requires an unconditional GOTO statement. By Lemma 4.2, minimization of the number of unconditional GOTO statements minimizes the code size. (Q.E.D.)

5 Experiment and Observation

5.1 Benchmark Results

To see the effectiveness of QDDs over BDDs, and the effectiveness of the code optimization, we realized certain benchmark functions by BDDs and QDDs. First,we compare QDDs and BDDs with respect to the number of nodes. Then, we convert these into code for BDD and QDD machines, and compare QDD's and BDD's with respect to the number of instructions.

Table 5.1 shows the experimental results. *Func. name* denotes the name of the benchmark functions; *# Inp.* denotes the number of input variables; *# Out.* denotes the number of outputs; *BDD Nodes* denotes the number of nodes of the MTBDD including both terminal and non-terminal nodes; *Opt. Codes* under *BDD* denotes the number of instructions of the optimized code for the 1-address BDD machine (near optimal solution); *Term. Nodes* denotes the number of terminal nodes; *Aver. Inst.* under *BDD* denotes the average number of instructions to evaluate an input vector by a 1-address BDD machine; *QDD Nodes* denotes the number of nodes of the MTQDD including both terminal and non-terminal nodes, that is the same as the number of instructions for a 4-address QDD machine; *X=00 Codes* under *QDD* denotes the number of instructions in the code for 3-address QDD machine, when only the first type of instruction in Fig. 3.8 is used; *Opt. Codes* under *QDD* denotes the number of instructions of the optimized code for the 3-address QDD machine, when all four types of instructions in Fig. 3.8 are used to minimize the number of GOTO statements; $X = 00$ *GOTO* denotes the number of GOTO statements, when only one type of branching instruction is used; *Opt. GOTO*=(Opt. Codes -QDD. Nodes) under *QDD* denotes the number of GOTO statements, when four types branching instructions are used; *Aver. Inst.* in QDD denotes the average number of instructions to evaluate an input vector by a 3-address QDD machine; and *Ratio* denotes the value: (Aver. Inst. in 1-address BDD machine)/(Aver. Inst. in 3-address QDD machine).

5.2 Detail of the Experiment

Optimization of Decision Diagrams: First, the ordering that minimizes the size of the MTBDD is obtained. Then, the input variables are partitioned into groups of two variables in the natural order to obtain the MTQDDs.

Optimization of Codes: Theorem 4.1 shows how to minimize the number of GOTO statements. The algorithm given by [11] is only applicable to the program with nodes whose in-degrees and out-degrees are both two. So, we developed our own algorithm to obtain near optimal solutions for our more general case.

5.3 Observations

From the table, we can observe the following:

- The number of nodes in QDDs is smaller than that of BDDs.
- The number of instructions for the 3-address QDD machine can be considerably reduced by an optimization algorithm.
- For *C432, in3, misex2, misj*, and *risc*, the number of GOTO statements in the optimized QDD codes is zero. This means that optimal code is generated for these functions. Also, for these functions, optimal code for BDD machines are generated.
- *signet* requires many GOTO statements in both BDD and QDD machines. The number of GOTO statements for a BDD machine is given by (Opt. Codes)-(BDD Nodes)=8671-7347=1324.
- *Opt. Codes*, the number of instructions for a 3-address QDD machines is often larger than *QDD Nodes*, the number of instructions for a 4-address QDD machine. The column headed by *Opt. GOTO (=OPT. Codes - QDD. Nodes)* shows the extra GOTOs. Except for a few functions, the extra GOTOs are rather small.
- Consider the value: (Sum of X=00 Codes)-(Sum of Optimal Codes)=28535-24528=4007. This shows the total number of instructions reduced by using four types of branch instructions, instead of using only one type of branching instructions. However, to specify four types of instructions, we need two additional

Table 5.1. Number of Nodes and Code Sizes for BDD Machine and QDD Machine.

			BDD				QDD						
Func. Name	# Inp.	# Out.	BDD Nodes	Opt. Codes	Term. Nodes	Aver. Inst.	QDD Nodes	X=00 Codes	Opt. Codes	X=00 GOTO	Opt. GOTO	Aver. Inst.	Ratio
C432	36	7	1779	1779	128	19.10	1027	1408	1027	381	0	12.73	1.50
amd	14	24	206	206	84	5.63	164	171	164	7	0	3.47	1.62
apex2	39	3	335	363	8	6.66	231	332	265	101	34	4.99	1.33
apex4	9	19	749	750	319	8.24	600	639	601	39	1	4.61	1.79
chkn	29	7	220	241	28	7.01	157	215	172	58	15	5.16	1.36
duke2	22	29	636	637	255	6.36	546	594	547	48	1	4.09	1.55
gary	15	11	228	232	70	5.51	173	191	174	18	1	3.42	1.61
in0	15	11	195	200	52	5.02	145	170	148	25	3	2.92	1.72
in1	16	17	284	299	55	6.85	217	288	229	71	12	4.70	1.46
in2	19	10	291	296	73	3.98	219	262	225	43	6	2.60	1.53
in3	35	29	259	259	72	6.63	214	234	214	20	0	4.77	1.39
in4	32	20	607	611	178	4.69	491	569	495	78	4	3.44	1.36
in5	24	14	461	466	134	8.54	369	452	371	83	2	6.57	1.30
in6	33	23	4325	4338	1638	7.51	3546	3815	3555	269	9	5.88	1.28
in7	26	10	300	301	112	7.58	256	275	256	19	0	5.84	1.30
m181	15	9	222	222	84	6.80	196	217	196	21	0	4.71	1.44
misex2	25	18	113	113	35	4.97	91	96	91	5	0	3.60	1.38
misex3	14	14	2910	2975	1041	7.55	1773	2159	1773	386	0	4.05	1.86
misj	35	14	4656	4656	1408	14.12	3275	3828	3275	553	0	9.57	1.47
mlp6	12	12	5270	6062	1238	12.10	2582	2966	2694	384	112	5.98	2.02
risc	8	31	56	56	28	4.42	44	44	44	0	0	2.55	1.74
signet	39	8	7347	8652	128	18.23	5671	8374	6907	2703	1236	13.31	1.37
tial	14	8	697	790	49	12.05	388	552	466	164	78	6.37	1.89
vg2	25	8	131	135	24	7.65	89	110	91	21	2	5.62	1.36
x1dn	27	6	200	218	18	9.55	126	171	141	45	15	5.74	1.66
x6dn	39	5	214	231	28	4.14	159	215	177	56	18	2.74	1.52
x9dn	27	7	204	222	22	9.30	140	188	157	48	17	5.80	1.60

bits in the instruction field. Let w be the number of bits in a word in the 3-address QDD machine, where only one type of branching instruction is used. Then, the merit of using four types of instructions is accurately expressed as: (Sum of X=00 Codes)$\times w$-(Sum of Opt. Codes)$\times(w+2)$= $28535w - 24528(w+2) = 4007w - 49056$. Note that, in most cases, $w > 20$, so we can conclude that the use of four types of Q_Branch instructions reduces the total number of bits.

- The last column of the table shows that the 3-address QDD machine is $1.28 - 2.02$ times faster than the 1-address BDD machine. Note that for *MLP6*, the ratio is greater than 2.

6 Conclusions and Comments

In this paper, we considered a branching program machine to evaluate multiple-output logic functions. To increase the speed of evaluation, we used QDDs instead of BDDs. To reduce the memory size, we used 3-address QDD machines instead of 4-address QDD machines. We proposed the use of four types of branch instructions. Also, we considered a method to optimize codes for 3-address QDDs. This is different from existing methods to optimize the decision diagrams. For various benchmark functions, we optimized the codes, and showed the effectiveness of the approach.

To show the usefulness of QDD machines, we have developed a parallel branching program machine (PBM128) consists of 128 QDD machines and a programmable interconnection on the Altera's Stratix II FPGA. We realized many benchmark functions on PBM128, and compared its memory size and computation time with the Intel's Core2Duo microprocessor. PBM128 requires approximately a quarter of the memory for the Core2Duo, and is 21.4-96.1 times faster than the Core2Duo. Details are shown in [18].

Acknowledgments

This research is partly supported by The Japan Society for the Promotion of Science (JSPS) Grant in Aid for Scientific Research, and the Knowledge Cluster Initiative (the second stage) of MEXT (Ministry of Education, Culture, Sports, Science and Technology), and by the U.S. Air Force / CVAQ (D. Nussbaum). Discussions with Prof. Shigeki Iwata and Mr. Hisashi Kajiwara were quite helpful.

References

[1] P. Ashar and S. Malik, "Fast functional simulation using branching programs," *Proc. International Conference on Computer Aided Design*, pp. 408-412, Nov. 1995.

[2] R. T. Boute, "The binary-decision machine as programmable controller," *Euromicro Newsletter*, Vol. 1, No. 2, 1976, pp. 16-22.

[3] P. C. Baracos, R. D. Hudson, L. J. Vroomen, and P. J. A. Zsombor-Murray, "Advances in binary decision based programmable controllers," *IEEE Transactions on Industrial Electronics*, Aug 1988, Vol. 35, No.3, pp. 417-425.

[4] J. T. Butler, T. Sasao, and M. Matsuura, "Average path length of binary decision diagrams" *IEEE Transactions on Computers*, Vol. 54, No. 9, Sept. 2005, pp. 1041-1053.

[5] C. H. Clare, *Designing Logic Systems Using State Machines*, McGraw-Hill, New York, 1973.

[6] M. Davio, J.-P Deschamps, and A. Thayse, *Digital Systems with Algorithm Implementation*, John Wiley & Sons, New York , 1983, p. 368.

[7] D. Green,*Modern Logic Design*, Addison-Wesley Publishing Company,1986.

[8] Y. Iguchi, T. Sasao, and M. Matsuura, "Implementation of multiple-output functions using PROMDDs," *30th International Symposium on Multiple-Valued Logic*, Portland, Oregon, U.S.A., May 23 - 25, 2000, pp. 199-205.

[9] Y. Iguchi, T. Sasao, M. Matsuura, and A. Iseno,"A hardware simulation engine based on decision diagrams" *ASP-DAC 2000, (Asia and South Pacific Design Automation Conference 2000)*, Jan. 26-28, 2000, Yokohama, Japan.

[10] Y. Iguchi, T. Sasao, and M. Matsuura, "Evaluation of multiple-output logic functions using decision diagrams," *ASP-DAC 2003, (Asia and South Pacific Design Automation Conference 2003)*, Kitakyusu, Jan. 21 - 24, 2003, pp. 312-315.

[11] S. Iwata,"Programs with minimal goto statements," *Information and Control*, Vol. 37, No. 1, pp. 105-114, 1978.

[12] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni, " Multi-valued decision diagrams: theory and applications," *Journal of Multiple-Valued Logic*, Vol. 4, No.1-2, 1998, pp. 9-62.

[13] D. Mange, "A high-level-language programmable controller," *IEEE Micro*, Vol. 6, No. 1, pp. 25-41 (Part I), Feb/Mar, 1986, Vol. 6, No. 2, pp. 47-63 (Part II), Mar/Apr, 1986.

[14] S. Minato, N. Ishiura, and S. Yajima, "Shared binary decision diagram with attributed edges for efficient Boolean function manipulation," *Proc. 27th ACM/IEEE Design Automation Conf.*, pp. 52-57, June 1990.

[15] P. C. McGeer, K. L. McMillan, A. Saldanha, A. L. Sangiovanni- Vincentelli, and P. Scaglia, "Fast discrete function evaluations using decision diagrams," *International Conf. on Computer Aided Design*, Nov. 1995, pp. 402-407.

[16] R. Murgai, F. Hirose, and M. Fujita, "Logic synthesis for a single large look-up table," *Proc. International Conference on Computer Design*, pp. 415-424, Oct. 1995.

[17] H. Nakahara and T. Sasao, "A PC-based logic simulator using a look-up table cascade emulator," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, Vol. E89-A, No.12, Dec. 2006, pp. 3471-3481.

[18] H. Nakahara, T. Sasao, K. Matsuura, and Y. Kawamura, "Emulation of sequential circuits by a parallel branching program machine," *5th International Workshop on Applied Reconfigurable Computing (ARC2009)*, Karlsruhe, Germany, March 16-18, 2009, *Lecture Notes in Computer Science*, LNCS5443, pp. 261-267, March 2009.

[19] S. Nagayama, T. Sasao, Y. Iguchi, and M. Matsuura, "Area-time complexities of multi-valued decision diagrams," *IEICE Transactions on Fundamentals of Electronics*, Vol. E87-A, No.5, pp. 1020-1028, May, 2004

[20] S. Nagayama, and T. Sasao, "On the optimization of heterogeneous MDDs," *IEEE Transactions on CAD*, Vol. 24, No.11, Nov. 2005, pp. 1645-1659.

[21] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," *ICCAD-93*, pp. 42–47, 1993.

[22] T. Sasao and J. T. Butler, "A method to represent multiple-output switching functions by using multi-valued decision diagrams." *IEEE International Symposium on Multiple-Valued Logic*, Santiago de Compostela, Spain, May 29-31, 1996, pp. 248-254.

[23] T. Sasao, *Switching Theory for Logic Synthesis*, Kluwer Academic Publishers, 1999.

[24] C. Scholl, R. Drechsler, and B. Becker, "Functional simulation using binary decision diagrams," *ICCAD'97*, pp. 8-12, Nov. 1997.

[25] P. J. A. Zsombor-Murray, L. J. Vroomen, R. D. Hudson, Le-Ngoc Tho, and P. H. Holck, "Binary-decision-based programmable controllers, Part I-III" *IEEE Micro*, Vol. 3. No. 4, pp. 67-83 (Part I), July-Aug. 1983, Vol. 3. No. 5, pp. 16-26 (Part II), Oct. 1983, Vol. 3. No. 6, pp. 24-39 (Part III), Nov.-Dec. 1983.