INVITED PAPER *Special Section on Multiple-Valued Logic and VLSI Computing*

A Quaternary Decision Diagram Machine: Optimization of Its Code*

Tsutomu SASAO[†a)], **Hiroki NAKAHARA**[†], **Munehiro MATSUURA**[†], **Yoshifumi KAWAMURA**[††], ***and* Jon T. BUTLER**[†††], ***Members***

SUMMARY This paper first reviews the trends of VLSI design, focusing on the power dissipation and programmability. Then, 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.
key words: *quarternary decision diagram, branching program machine*

1. Trends of VLSI Design

1.1 Explosion of Complexity

With the growth of multimedia and other applications, the demand for high-performance processors has increased. In the past, Moore's Law solved this problem. Moore's Law states that the number of transistors on a chip doubles every 18 months.

In the process of miniaturization, the scaling down of transistor size and chip area has reduced power dissipation. That is, by scaling down the transistor size in LSIs, chip area, delay, and power dissipation can be reduced at the same time. However, in the future, the number of transistors on a chip is expected to fall short of that predicted by Moore's Law.

1.2 Power Dissipation

As transistor size decreases, supply voltage must also scale down to keep the electric field in the integrated circuit constant [32]. However, as the supply voltage decreases, subthreshold leakage current increases. Nowadays, power dissipation due to leakage current accounts for about 40% of the total power dissipation in a microprocessor [5]. Therefore, as supply voltage is reduced, *power density* is a limiting factor. With an increase of the power density, the temperature of chip may become too high. To make matters worse, leakage current increases exponentially with temperature [3]. When a transistor produces more heat than the heatsink can dissipate, thermal runaway occurs. Therefore, cooling is very important. In the past, reduction of chip area was the main design issue. However, nowadays, the reduction of power dissipation is the primary design issue. In mobile applications, battery size is limited, so the use of low power devices is crucial.

1.3 Multi-Core and Parallel Processing

Power dissipation of a CMOS gate is approximately

$$P = \alpha \times V_{dd}^2 \times f,$$

where α is a constant, V_{dd} is the supply voltage, and f is the clock frequency.

Reduction of the supply voltage without changing transistor dimensions requires a reduction in clock frequency f [4]. Assume that the power supply voltage is reduced by 30%, and that the clock frequency is reduced by 50%. In this case, we have

$$\alpha \times (0.7V_{dd})^2 \times 0.5f = 0.25\alpha V_{dd}^2 f.$$

Consider a dual core version of this, as shown in Fig. 1. In this case, a reduction by half of the frequency is compensated by an increase by two times of the number of processors, yielding nearly equal throughput. That is, this change has resulted in a reduction by half of the power with no change in the system throughput.

In personal computers, many threads are running at the same time. Thus, many computers can benefit from multicores. In this sense, chip area is increased to reduce power

Manuscript received November 9, 2009.
[†]The authors are with Kyushu Institute of Technology, Iizuka-shi, 820–8502 Japan.
[††]The author is with Renesas Electronics Corp., Tokyo, 100–0004 Japan.
[†††]The author is with the Naval Postgraduate School, Monterey, CA 93943–5121, USA.
[*]A preliminary version of this paper was presented at ISMVL-2010 [30].
a) E-mail: sasao@cse.kyutech.ac.jp
DOI: 10.1587/transinf.E93.D.2026

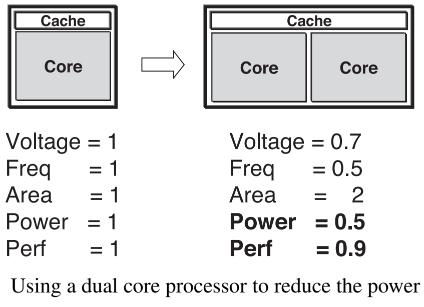


Fig. 1 Using a dual core processor to reduce the power by half.

dissipation. Increasing the number of cores increases the chip cost, but the reduction of power dissipation is more important.

By reducing power, cooling fans can be often eliminated [2]. Also, reliability will be enhanced because of lower temperatures. Excessively high temperature can burn out the chip. Even if the temperature is low enough so that this does not occur, high temperature can cause cumulative damage.

In multi-core systems, unused cores can be turned off to further reduce power dissipation. Unfortunately, developing efficient software for multi-core is not so easy. Most existing software is single-threaded. In a single core processor, various methods are used to increase the performance without increasing the clock frequency, including pipelining, super scalar, super pipeline architecture, and very long instruction word processors (VLIWs). Unfortunately, even if the chip area of a single-core processor is doubled to increase the performance, the resulting performance is increased only by 1.4 times, as predicted by Pollack's rule [4].

1.4 Programmable Device

With the miniaturization of chips, the cost of masks for VLSI has increased drastically. Since the number of transistors has increased, VLSI design is now very complicated. As transistors become smaller, variability of the threshold voltage of transistors increases. Therefore, achieving consistent switching becomes difficult. As a result, design and test cost has also increased [9]. Due to this, custom chips are feasible only for mass-production products, such as games and cellular phones. In addition, the life of today's products is short: every few months, new products are developed. Thus, the number of newly developed VLSIs has been reduced. Instead, microprocessors, application specific standard products (ASSPs), and field programmable gate arrays (FPGAs) are used to implement electronic appliances. These can be customized by writing programs.

2. Introduction of Branching Program Machines

In the rest of this paper, we focus on branching program machines, which are suitable for control applications. They are programmable, since major parts consist of memories. Because memory is involved, reliability can be improved by using traditional techniques, such as error correcting codes (ECC).

Branching program machines for BDDs have been used in control applications [6], [10]–[12]. 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 [7].

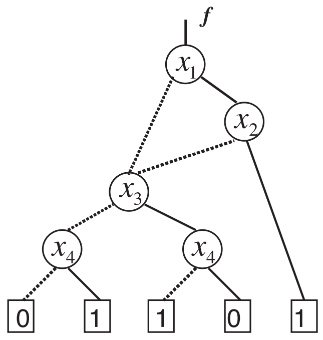


Fig. 2 An example of BDD.

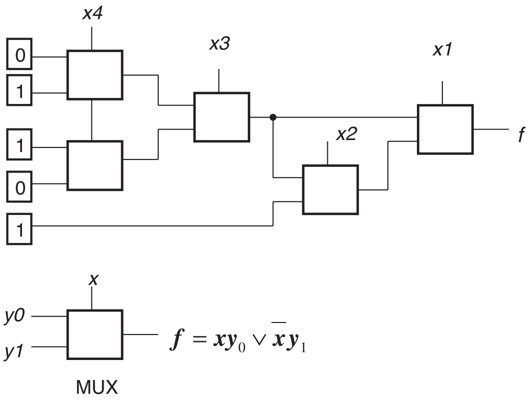


Fig. 3 MUX circuit.

Parallelization can be implemented by multi-way branching programs. Thus, performance can be improved without increasing the clock frequency.

2.1 Conversion from a Circuit to a Branching Program Machine

Consider the implementation of a given logic function. This can be represented by a binary decision diagram (BDD). Figure 2 shows the BDD of an example function, $f(x_1, x_2, x_3, x_4) = x_1x_2 \vee (x_3 \oplus x_4)$. In this diagram, dotted lines (left lines) correspond to $x_i = 0$ and solid lines (right lines) correspond to $x_i = 1$. By replacing each non-terminal node of a BDD with a multiplexer (MUX), we have a circuit, at the top of Fig. 3, that realizes the given logic function whose BDD is shown in Fig. 2.

However, such implementation requires dedicated interconnections and expensive masks. A branching program machine is a sequential circuit that emulates the MUX circuit. In this case, the interconnections are programmed in a memory. Thus, by using a branching program machine, a logic function is implemented by logic and memory. Since it has no instruction fetch, it is faster and dissipates less power than a general purpose microprocessor.

Unfortunately, a branching program machine is slower than the original logic circuit, since it emulates the circuit sequentially. A straightforward method to increase the speed is to increase the clock frequency. However, this is difficult in most cases. To increase processing speed without increasing the clock frequency, we use a Multi-valued Decision Diagram (MDD). For example, when two variables are evaluated at the same time, the decision diagram has four-way branches; this is called a Quarternary Decision Diagram (QDD). In this way, performance is increased without increasing the clock frequency. Such an idea is used in VLIW processors [21], where branch instructions are multiway.

2.2 Optimization of Branching Program Machine

A Quarternary Decision Diagram (QDD) machine is *up to* two times faster than a BDD machine. However, instruction words for the QDD machine require four address fields, i.e., instructions with many bits are necessary. This increases the power dissipation, which is proportional to the number of bits in the instruction words.

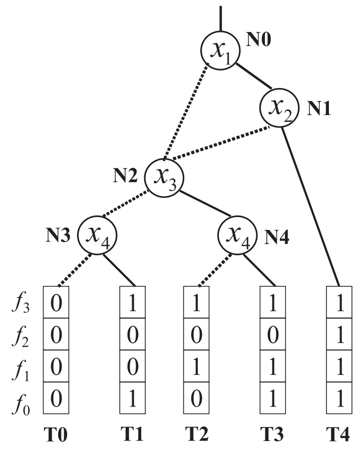
Optimization of code for a QDD machine can be treated as an optimization of a 4-valued logic circuit. A multi-core system of 128 QDD machines was implemented on an FPGA [24]. This is up to 96 times faster than the microprocessor (Core2Duo, 1.2 GHz, U7600), even though the QDD machine runs at 100 MHz, while the microprocessors run at 1.2 GHz. Further, the power dissipation of 128 QDD machine is only a quarter of the microprocessor.

The rest of this paper is organized as follows: Section 3 introduces a method to represent multi-output logic functions by multi-valued decision diagrams. Section 4 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 5 shows an optimization problem of codes for 3-address QDD machines. Section 6 shows the experimental results. And finally, Sect. 7 concludes the paper.

3. Representation of Multiple-Output Functions

3.1 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. Figure 4 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 [20], [25]. When the function is represented by an MDD(k), the evaluation of a logic function can be k times faster than the corresponding BDD[†]. 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. 5. For many benchmark functions, the total size of the memory for an MDD(k) achieves its minimum when $k = 2$ [25]. Therefore, in logic evaluation, MDD(2)s



	T0	T1	T2	T3	T4
f_3	0	1	1	1	1
f_2	0	0	0	0	1
f_1	0	0	1	1	1
f_0	0	1	0	1	1

Fig. 4 Example of an MTBDD.

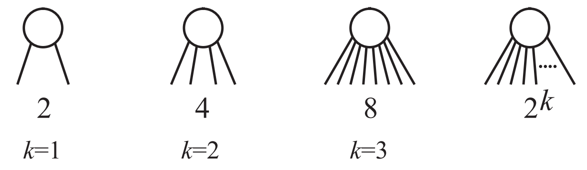


Fig. 5 Nodes for MDD(k).

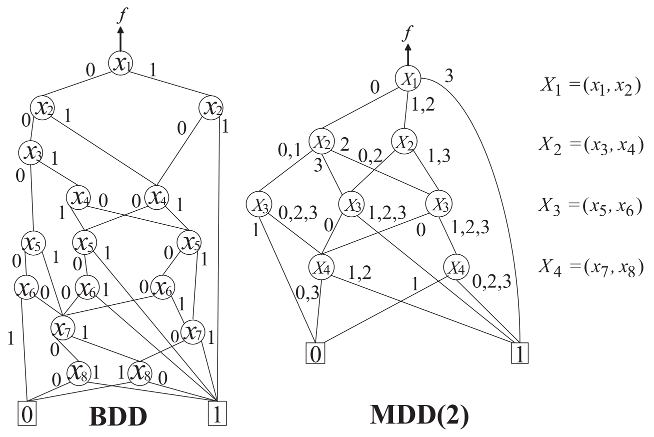


Fig. 6 Conversion of BDD to MDD(2).

are more suitable than BDDs. Since nodes in an MDD(2) have 4 branches, it is termed a Quarternary Decision Diagram (QDD).

3.2 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 nodes of the BDD by a heuristic method [27].

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

2. Partition the input variables to generate an MDD(k) [28].

Figure 6 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.

4. Branching Program Machine

Special machines to evaluate MDDs have been developed [13]–[15]. Unfortunately, they are unsuitable for practical applications. Here, we consider a machine whose architecture is well-suited for evaluating MDDs, but is easily programmed.

4.1 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 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 4.1: Consider the MTBDD shown in Fig. 4. 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 7 shows the architecture of the 2-address BDD machine, where only the circuit for the branching operation is shown. The first field, COM, 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. 7 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

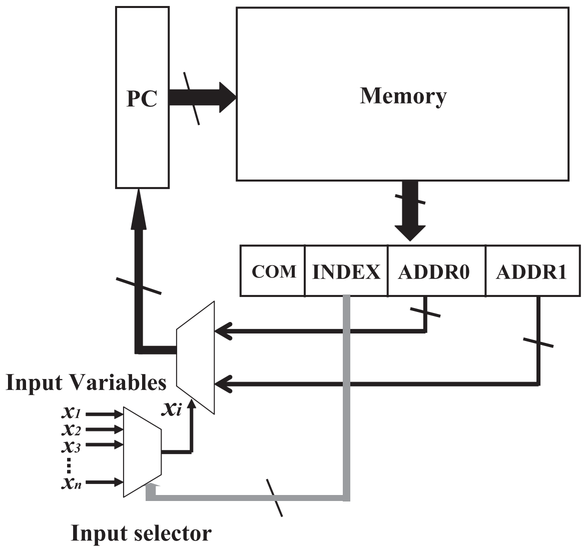


Fig. 7 2-address BDD machine.

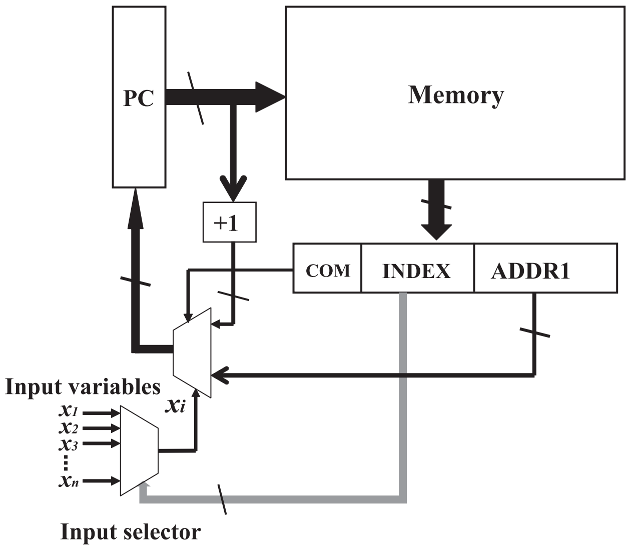


Fig. 8 1-address BDD machine.

is specified. To reduce the width of the instruction words, 1-address BDD machines shown in Fig. 8 have been developed [6], [11], [18], [33]. In this case, when the value 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.

4.2 4-Address QDD Machine

By simultaneously 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

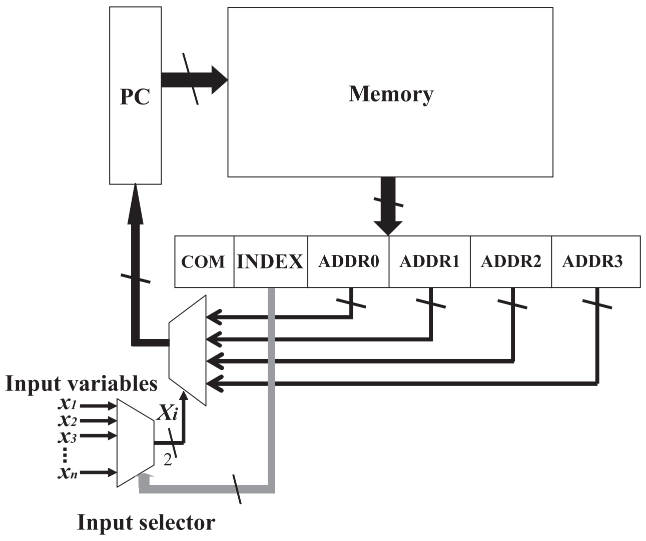


Fig. 9 4-address QDD machine.

Branch	INDEX	ADDR0	ADDR1	ADDR2	ADDR3

Fig. 10 Branch instruction for 4-address QDD machine.

Output	Address	Output Values

Fig. 11 Output instruction for a QDD machine.

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
```

Figure 10 shows the format for the branch instruction. Figure 9 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. 9 produces X_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.

Figure 11 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.

Branch0	INDEX	ADDR1	ADDR2	ADDR3

Fig. 12 Branch instruction for a 3-address QDD machine.

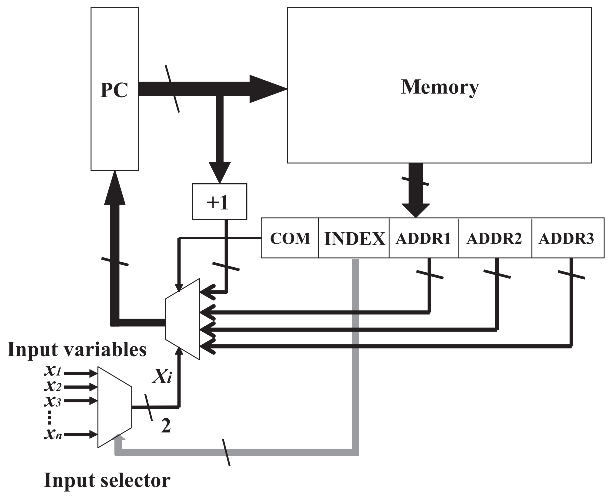


Fig. 13 3-address QDD machine.

4.3 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. 12. This instruction is symbolically denoted by

```
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 corresponds to the next address of the current instruction. This instruction performs the following operations:

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

Lemma 4.1: 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

```
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. Figure 13 shows the architecture of the

Branch0	INDEX	ADDR1	ADDR2	ADDR3
Branch1	INDEX	ADDR0	ADDR2	ADDR3
Branch2	INDEX	ADDR0	ADDR1	ADDR3
Branch3	INDEX	ADDR0	ADDR1	ADDR2

Fig. 14 Four types of branch instructions for 3-address QDD machine.

3-address QDD machine, where only the circuit for branching operations is shown. Consider the instruction in Fig. 12. When the value of INDEX and the input variables are non-zero, the machine is like 4-address QDD machine. When the value of INDEX and the input variables are equal to 0, the program counter (PC) is incremented by one, to access the next address.

In our hardware implementation, we use the four types of branch instructions shown in Fig. 14. 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.

5. Optimization of Codes for QDD Machines

In this section, we consider a method to reduce the number of instructions for QDD machines. Interestingly, this is solved by minimizing the number of unconditional GOTO statements.

Definition 5.1: Given the QDD and an order of the input variables (e.g. $x_1, x_2, \ldots,$ andx_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 5.2: 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:

$$CSIZE(4aQDDM) = N_N + N_T. \tag{1}$$

(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 5.3: 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:

$$CSIZE(3aQDDM) = N_U + N_N + N_T \tag{2}$$

$$0 \leq N_U \leq N_N \tag{3}$$

(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.

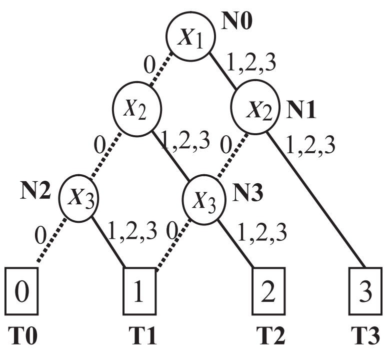


Fig. 15 QDD for example function.

Example 5.2: Consider the QDD shown in Fig. 15. It has five non-terminal nodes, and four terminal nodes. When the code is generated in 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 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 node N1, node N3, and 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 5.2: 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 5.1: 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 5.3, minimization of the number of unconditional GOTO statements minimizes the code size. (Q.E.D.)

6. Experiment and Observation

6.1 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 numbers of nodes. Then, we convert these into code for BDD and QDD machines, and the number of instructions.

Table 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. 14 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. 14 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).

Table 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

6.2 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 Code: Theorem 5.1 shows how to minimize the number of instructions by minimizing the number of GOTO statements. The algorithm given by [16] 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.

6.3 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 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) × w − (Sum of Opt. Codes) × $(w + 2)$ = $28535w - 24528(w + 2) = 4007w - 49056$. 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. This is due to GOTO statements. If we compared the average numbers of instructions in a 2-address BDD machine and a 4-address QDD machine, the ratio is at most 2.

6.4 Hardware Implementation

To show the usefulness of multi-core QDD machines, we have developed a parallel branching program machine (PBM128) consisting of 128 QDD machines and a programmable interconnection on Altera's Stratix II FPGA. We realized many benchmark functions on the PBM128, and compared its memory size and computation time with Intel's Core2Duo microprocessor. PBM128 requires approximately one quarter of the memory required by the Core2Duo, and is 21.4-96.1 times faster than the Core2Duo. Details are shown in [24].

7. Conclusions

In this paper, first, we review the trends of VLSI design, focusing on the power dissipation and programmability. Then, 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. We show that the minimization of the number of instructions corresponds to minimizing the number of unconditional GOTO statements. For various benchmark functions, we optimized the codes, and showed the effectiveness of the approach.

Acknowledgements

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] B. Beavers, "The story behind the Intel Atom processor success," IEEE Des. Test Comput., vol.26, no.2, pp.8–13, March-April 2009.
[3] S. Borkar, "Design challenges of technology scaling," IEEE Micro, vol.19, no.4, pp.23–29, July-Aug. 1999.
[4] S. Borkar, "Thousand core chips: A technology perspective," Proc. 44th annual Design Automation Conference, DAC-2007, pp.746–749, June 2007.

[5] S. Borkar, N.P. Jouppi, and P. Stenstrom, " Microprocessors in the era of terascale integration," Proc. Conference on Design, Automation and Test in Europe, (DATE-2007), pp.237–242, April 2007.

[6] R.T. Boute, "The binary-decision machine as programmable controller," Euromicro Newsletter, vol.1, no.2, pp.16–22, 1976.

[7] P.C. Baracos, R.D. Hudson, L.J. Vroomen, and P.J.A. Zsombor-Murray, "Advances in binary decision based programmable controllers," IEEE Trans. Ind. Electron., vol.35, no.3, pp.417–425, Aug. 1988.

[8] J.T. Butler, T. Sasao, and M. Matsuura, "Average path length of binary decision diagrams" IEEE Trans. Comput., vol.54, no.9, pp.1041–1053, Sept. 2005.

[9] R.K. Brayton, "The future of logic synthesis and verification," in H. Soha and T. Sasao eds., Logic Synthesis and Verification, Kluwer Academic Publishers, 2002.

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

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

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

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

[14] 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), Yokohama, Japan, Jan. 2000.

[15] 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), pp.312–315, Kitakyusu, Jan. 2003.

[16] S. Iwata, "Programs with minimal goto statements," Information and Control, vol.37, no.1, pp.105–114, 1978.

[17] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni, "Multi-valued decision diagrams: Theory and applications," J. Multiple-Valued Logic, vol.4, no.1-2, pp.9–62, 1998.

[18] D. Mange, "A high-level-language programmable controller," IEEE Micro, vol.6, no.1, pp.25–41 (Part I), Feb./March, 1986, vol.6, no.2, pp.47–63 (Part II), March/April, 1986.

[19] 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.

[20] 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, pp.402–407, Nov. 1995.

[21] S.M. Moon and S.D. Carson, "Generalized multiway branch unit for VLIW microprocessors," IEEE Trans. Parallel Distrib. Syst., vol.6, no.8, pp.850–862, Aug. 1995.

[22] 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.

[23] H. Nakahara and T. Sasao, "A PC-based logic simulator using a look-up table cascade emulator," IEICE Trans. Fundamentals, vol.E89-A, no.12, pp.3471–3481, Dec. 2006.

[24] 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 2009, Lect. Notes Comput. Sci., LNCS5443, pp.261–267, March 2009.

[25] S. Nagayama, T. Sasao, Y. Iguchi, and M. Matsuura, "Area-time complexities of multi-valued decision diagrams," IEICE Trans. Fundamentals, vol.E87-A, no.5, pp.1020–1028, May 2004.

[26] S. Nagayama and T. Sasao, "On the optimization of heterogeneous MDDs," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol.24, no.11, pp.1645–1659, Nov. 2005.

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

[28] 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, pp.248–254, Spain, May 1996.

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

[30] T. Sasao, H. Nakahara, M. Matsuura, Y. Kawamura, and J.T. Butler, "A quaternary decision diagram machine and the optimization of its code," 39th International Symposium on Multiple-Valued Logic (ISMVL 2009), pp.362–369, May 2009.

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

[32] N. Weste and K. Eshraghian, Principles of CMOS VLSI Design: A Systems Perspective, Addison-Wesley, 1980.

[33] 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.

Tsutomu Sasao received the B.E., M.E., and Ph.D. degrees in Electronics Engineering from Osaka University, Osaka Japan, in 1972, 1974, and 1977, respectively. He has held faculty/research positions at Osaka University, Japan, IBM T. J. Watson Research Center, Yorktown Height, NY and the Naval Postgraduate School, Monterey, CA. He has served as the Director of the Center for Microelectronic Systems at the Kyushu Institute of Technology, Iizuka, Japan. Now, he is a Professor of Department of Computer Science and Electronics, His research areas include logic design and switching theory, representations of logic functions, and multiple-valued logic. He has published more than 8 books on logic design including, Logic Synthesis and Optimization, Representation of Discrete Functions, Switching Theory for Logic Synthesis, and Logic Synthesis and Verification, Kluwer Academic Publishers 1993, 1996, 1999, 2001 respectively. He has served Program Chairman for the IEEE International Symposium on Multiple-Valued Logic (ISMVL) many times. Also, he was the Symposium Chairman of the 28th ISMVL held in Fukuoka, Japan in 1998. He received the NIWA Memorial Award in 1979, Takeda Techno-Entrepreneurship Award in 2001, and Distinctive Contribution Awards from IEEE Computer Society MVL-TC for papers presented at ISMVLs in 1986, 1996, 2003 and 2004. He has served an associate editor of the IEEE Transactions on Computers. He is a Fellow of the IEEE.

Hiroki Nakahara received the B.E., M.E., and Ph.D. degrees in computer science from Kyushu Institute of Technology, Fukuoka, Japan, in 2003, 2005, and 2007, respectively. He has a research position at Kyushu Institute of Technology, Iizuka, Japan. His research interests include logic synthesis, reconfigurable architecture, embedded system, and high-level synthesis. He is a member of the IEEE.

Munehiro Matsuura was born in 1965 in Kitakyushu City, Japan. He studied at the Kyushu Institute of Technology from 1983 to 1989. He received the B.E. degree in Natural Sciences from the University of the Air, in Japan, 2003. He has been working as a Technical Assistant at the Kyushu Institute of Technology since 1991. He has implemented several logic design algorithms under the direction of Professor Tsutomu Sasao. His interests include decision diagrams and exclusive-OR based circuit design.

Yoshifumi Kawamura graduated from Electrical Engineering of Miyagi Technical College. In 1981, he entered the Semiconductor Division of Hitachi, Ltd., and engaged in the development of Telecommunication devices. In 1993, he transferred Graphics Communication Laboratories, and engaged in the research and development of MPEG2 System for four years. In 1997, he transferred to the Semiconductor Div. Hitachi Ltd., and engaged in development of LSI for GSM Cell phone. In 2003, he transferred to the Corporate Strategy Planning Division, Renesas Technology Corporation. Now, he is a senior engineer of the System Core Technology Division. And, he is involved in the development of programmable and reconfigurable technology.

Jon T. Butler received the BEE and MEngr degrees from Rensselaer Polytechnic Institute, Troy, New York, in 1966 and 1967, respectively. He received the PhD degree from The Ohio State University, Columbus, Ohio, in 1973. Since 1987, he has been a professor at the Naval Postgraduate School, Monterey, California. From 1974 to 1987, he was at Northwestern University, Evanston, Illinois. During that time he served two periods of leave at the Naval Postgraduate School, first as a National Research Council Senior Postdoctoral Associate (1980-1981) and second as the NAVALEX Chair Professor (1986-1987). He served one period of leave as a foreign visiting professor at the Kyushu Institute of Technology, Iizuka, Japan. His research interests include logic optimization, multiple-valued logic, and reconfigurable computing. He has served on the editorial boards of the IEEE Transactions on Computers, Computer, and the IEEE Computer Society Press. He has served as the editor-in-chief of Computer and the IEEE Computer Society Press. He received the Award of Excellence, the Outstanding Contributed Paper Award, and a Distinctive Contributed Paper Award for papers presented at the International Symposium on Multiple-Valued Logic. He received the Distinguished Service Award, two Meritorious Awards, and nine Certificates of Appreciation for service to the IEEE Computer Society. He is a fellow of the IEEE.