Asynchronous BDD-like structures

Oleg Mayevsky[1], Andrey Mokhov[2], Danil Sokolov[2]

[1] oleg-maevsky@yandex.ru
[2] Newcastle University, United Kingdom
andrey.mokhov@ncl.ac.uk, danil.sokolov@ncl.ac.uk

Abstract. Another attempt to apply BDD-like structures to build digital systems. The close relationship of BDDs with elementary schemes of computational algorithms is used. A simple conceptual option for decomposition of asynchronous computational processes on sub-machines with BDD-like structure is seen in several examples.

Keywords: binary decision diagrams, asynchronous circuits

1 Introduction

An important property of Binary Decision Diagrams (BDD) is their ability to canonically and compactly represent Boolean functions, which is attractive in digital circuit design, and in some cases makes BDDs preferable to other representations [1]. A lot of research has been dedicated to the use of BDDs for synthesis and analysis of circuits in different technologies [2–4]. BDDs were typically used for implementing the combinational part of control automata in an 'orthogonal way' that came both with benefits, such as testability and race freedom, as well as drawbacks, such as the requirement for separate descriptions of the control and operational parts of the system, and, perhaps more importantly, the sequentiality of described processes. The sequentiality was caused by the fact that a single BDD could only represent a single Boolean function, which forced the designer to decompose the combinational part of control into a set of independently synthesised components that could not be used simultaneously.

In this work we use BDD-like diagrams for the compact representation of concurrent computations. Thanks to the orthogonality of obtained descriptions, it is possible to directly map them into efficient asynchronous controllers of large computation systems. We show that the size of such descriptions significantly depends on the decomposition between the control and operational parts of the system.

Section 2 demonstrates a correspondence between BDDs and simple logic computations on an example. To emphasise the correspondence we show how BDDs can be converted to *Orthogonal Canonical Parameterised Computation Graphs (OCPCGs)* that inherit orthogonality and canonicity – the two key properties of BDDs. We then discuss OCPCG-based control descriptions and show how they can be translated to Petri Nets. The rest of the section is dedicated to graph decomposition, concurrency and synchronisation issues.

Section 3 discusses ways of reducing the size of OCPCG descriptions by restricting the labellings of their elements. We use several examples to show how the size and properties of OCPCGs describing the control part of a system depend on the properties of its operational part, and study restrictions on OCPCG labellings that preserve the canonicity.

Section 4 concludes the paper with an example of using OCPCGs in the implementation of the control part of a 5-bit multiplier. It is known that the corresponding Boolean functions cannot be compactly represented by BDDs, however we show that OCPCGs admit a simple reformulation of this problem leading to significant savings in terms of the size of the obtained representations.

2 BDDs and logic computations

Consider a BDD in Fig. 1 describing a Boolean function $y = F(x_1, x_2, x_3, x_4, x_5)$ whose truth table is shown in Tab. 1. The *terminal* nodes 0 and 1 at the bottom of the BDD correspond to the value that the function has on a particular set of input variables. This diagram can also be thought of as a control part of a system, whose task is to execute a particular operation under the condition $y = 1$ and skip it when $y = 0$.

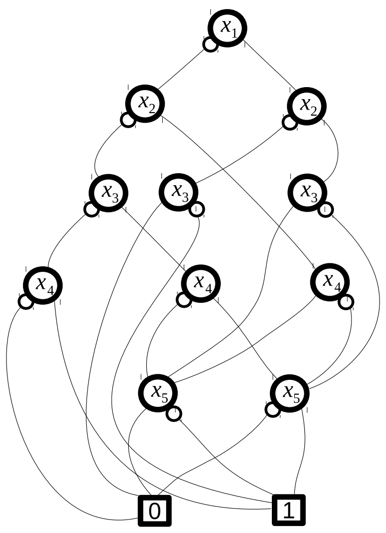


Fig. 1: BDD corresponding to the function from Tab. 1.

Let us modify the diagram in Fig. 1 by labelling the arcs leading to the two terminals by 0 and 1 according to their targets (0 labels may also be omitted on diagrams, as one can always infer them from remaining 1 labels). We then

x_1	x_2	x_3	x_4	x_5	y	x_1	x_2	x_3	x_4	x_5	y	x_1	x_2	x_3	x_4	x_5	y	x_1	x_2	x_3	x_4	x_5	y
0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	1	1	0	0	0	0
0	0	0	0	1	0	0	1	0	0	1	1	1	0	0	0	1	1	1	1	0	0	1	1
0	0	0	1	0	1	0	1	0	1	0	1	1	0	0	1	0	1	1	1	0	1	0	0
0	0	0	1	1	1	0	1	0	1	1	0	1	0	0	1	1	1	1	1	0	1	1	1
0	0	1	0	0	1	0	1	1	0	0	0	1	0	1	0	0	0	1	1	1	0	0	1
0	0	1	0	1	0	0	1	1	0	1	1	1	0	1	0	1	0	1	1	1	0	1	0
0	0	1	1	0	0	0	1	1	1	0	1	1	0	1	1	0	0	1	1	1	1	0	1
0	0	1	1	1	1	0	1	1	1	1	0	1	0	1	1	1	0	1	1	1	1	1	0

Table 1: Truth table of a Boolean function of 5 variables.

merge the terminals into a new node end, and add another node begin at the top of the diagram for symmetry. See the result in Fig. 2.

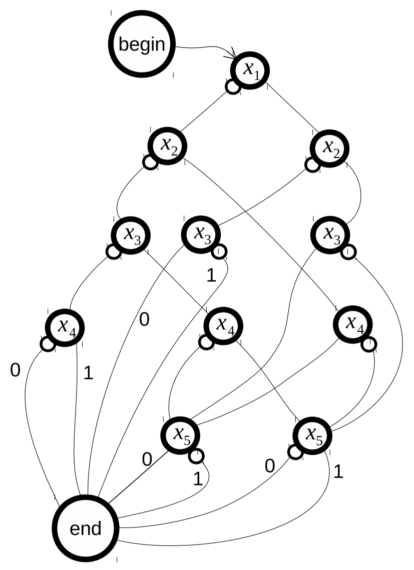


Fig. 2: A computation graph derived from the BDD in Fig. 1.

The newly added nodes begin and end have a different semantics from that of terminals 0 and 1 in BDDs: instead of denoting the final computed value, they simply indicate the start and end of a computation process. Note that the resulting computation graph in Fig. 2 may be *local*, that is, it may be a (sequential) part of a larger computation system, which may contain concurrency and/or cycles, as illustrated in Fig. 3.

All nodes of a computation graph can take part in the computation, similar to Petri Nets. As an example, we can place a *token* in the begin node, that

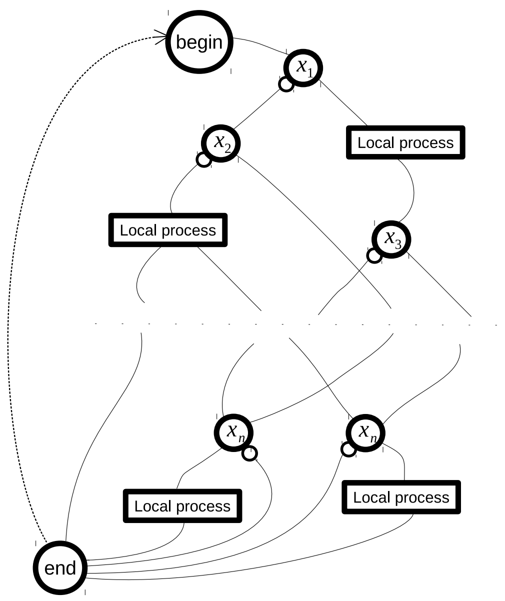


Fig. 3: A cyclic computation graph comprised of local processes.

will then travel along the arcs of the graph following the local *routing* decisions made in individual nodes according to the values of input variables. Each arc could in fact correspond to a Petri Net that consumes a single token from a node, performs a computation process according to usual Petri Net rules, and then signals about the completion by releasing the token to the next node. In fact, the whole computation graph can be modelled by a Petri Net if we create suitable Petri Net fragments corresponding to individual computation nodes, e.g. as shown in Fig. 4.

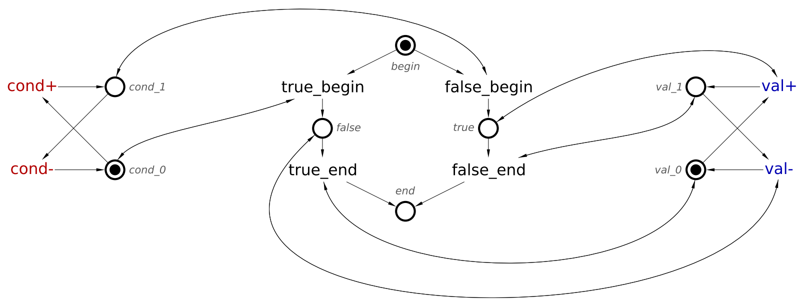


Fig. 4: An example fragment Petri Net.

3 Labellings

Let us come back to the graph in Fig. 2. The upper half of the graph is dedicated to the computation of the necessary logic conditions that are used by the lower half of the graph for actually executing the action associated with $y = 1$. The complexity of the computation part eventually determines the complexity of the hardware that implements it. It is well-known that BDDs often lead to overly complex circuits when used directly to implement logic computations, compared to conventional logic synthesis. However, one can consider alternative ways of labelling the elements of the computation graph in Fig. 2, which can lead to a reduction in the number of its vertices and therefore in a lower complexity of the resulting hardware.

As an example, let us change the semantics of the labels in our computation graph. The label of 1 will now correspond to adding 1 to the current value of y modulo 2, that is, $y \leftarrow y \oplus 1$. The label of 0 will correspond to $y \leftarrow y \oplus 0$, which is essentially a *no-op*. With this approach, one can imagine the token to start in the node begin with the initial value $y = 0$, and travel along the arcs of the graph, undergoing the transformations corresponding to the labels on route to the destination node end. When the token reaches end, the value of y becomes the final outcome of the computation. There may be several equivalent labellings that compute the same Boolean function. Fig. 5 gives an example of a labelling that is equivalent to the one shown in Fig. 2.

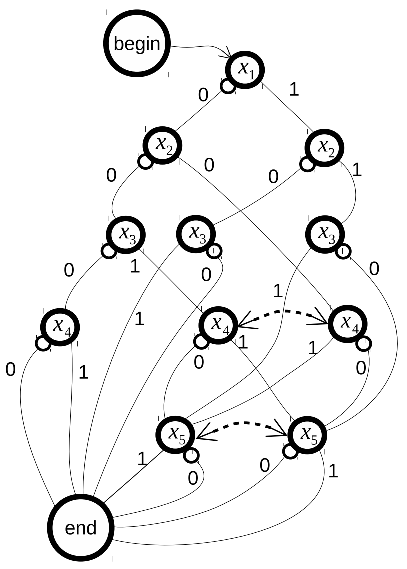


Fig. 5: An equivalent labelling of the graph from Fig. 2.

The existence of equivalent labellings means that our new computation graphs lost an important property inherited from BDDs – the canonicity of the representation of Boolean functions. Fortunately, it is possible to recover it.

Indeed, there are certain rules we can follow to relabel a computation graph in order to bring it back to a canonical form. For example, consider an arbitrary node x_n with arcs labelled a, b, c and d, as shown in Fig. 6(left).

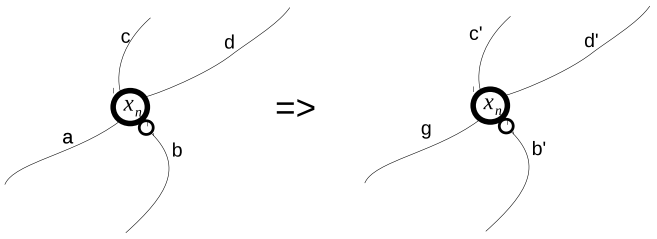


Fig. 6: Equivalent relabelling.

Let us try to change the label a to g. Labels b, c and d may need to be changed too in order to preserve the equivalence. To find out admissible new labels b', c' and d' we can solve the following system of equations:

$$\begin{cases} g \oplus c' = a \oplus c, \\ g \oplus d' = a \oplus d, \\ b' \oplus c' = b \oplus c, \\ b' \oplus d' = b \oplus d. \end{cases} \tag{1}$$

The equations above use modulo 2 addition $\oplus$, but in general any additive group can be used. If the set of labels does not form an additive group with respect to the chosen addition operator, then the resulting system of equations may have no solutions or multiple solutions, therefore limiting the freedom of graph relabelling and/or making the derivation of a canonical labelling more challenging. On the other hand, if the system of equations is guaranteed to always have a unique solution, as in the case of (1), then one can fix the label of a particular arc, e.g. fix the $x_n = 0$ branch to always have label 0, leading to a canonical labelling of the computation graph.

Once a canonical relabelling is performed, one can reduce the resulting graph by merging equivalent nodes, similar to the reduction of BDDs. Fig. 5 shows two pairs of equivalent nodes by dashed arrows; after merging them we obtain the reduced computation graph shown in Fig. 7.

The system of equations of the form (1) will be analogous for any additive group, including non-commutative ones, such as real or complex numbers, vectors, matrices, etc. The issue of commutativity in this context is related to concurrency. Indeed, if the labels do not commute then the computation order is important and the graph corresponds to a sequential computation process. On

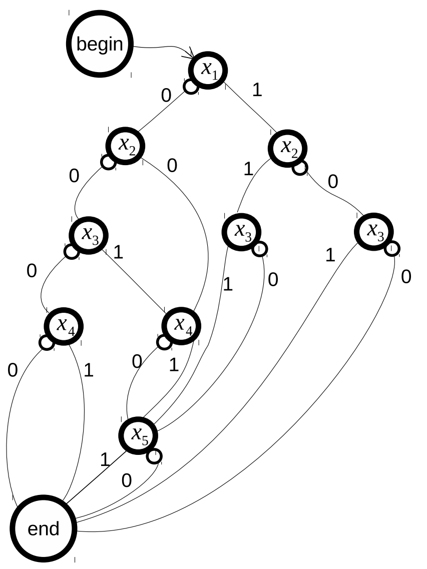


Fig. 7: Reduced computation graph from Fig. 5.

the other hand, if labels commute, e.g. $a + b = b + a$, then the order in which a and b are performed does not matter and they may be performed concurrently. Most real systems are mixed and involve both sequential and concurrent computations, which can be represented by general models such as Petri Nets. The following example illustrates how commutative labels can also be used for finding further reduction opportunities in computation graphs.

Let us remove the node conditions x_n from the graph in Fig. 7 and place them on the incoming arcs instead, as shown in Fig. 8. In other words, nodes no longer contain any conditions, and the latter actually become arc labels, which means variable comparisons are now performed while travelling along arcs. We therefore have a new set of arc labels $\{x_1, x_2, ...\}$ that do not interact in any way with existing labels $\{0, 1\}$, and therefore commute with them. However, we can no longer relabel arcs as described above, because we have not yet defined the group operation that acts on the new labels. Despite this, we can already identify new equivalent nodes that can be merged, as indicated by dashed arrows. The graph obtained by merging these nodes is shown in Fig. 9.

Let us refer to the new labels $\{x_1, x_2, ...\}$ as *condition labels*. Can we move these labels from one arc to another? Yes, we can! Below we give one possible relabelling method, which is not as simple as we would like, but does provide further intuition into labelled computation graphs.

Let us keep all condition labels in a queue, which is initially empty, and supports two operations: i) extracting a label at the front of the queue, ii) adding a new label to the queue, either combining it an with existing label corresponding to the same control variable (using a group operation defined below), or adding

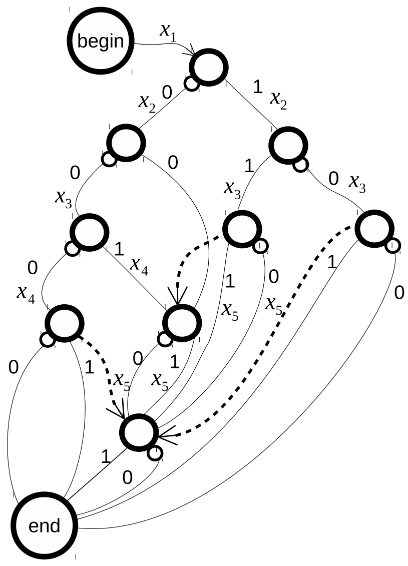


Fig. 8: Moving conditions from nodes to arcs.

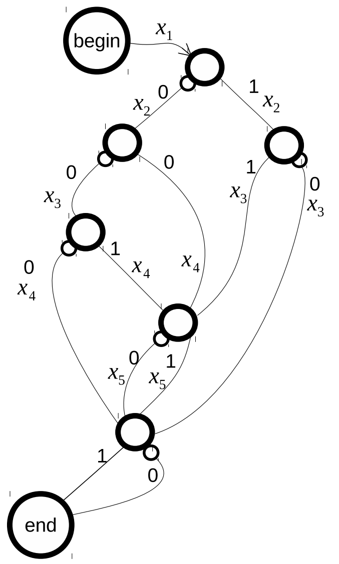


Fig. 9: Reducing the graph in Fig. 8.

it to the back of the queue if it is the first occurrence of the variable in the queue. Note that labels corresponding to different variables do not interact with each other, therefore it is sufficient to define an operation only on labels corresponding the the same variable.

Let x stand for a control variable. Then we can define a group operation $+$ for combining labels corresponding to this variable according to Tab. 2. This is an infinite commutative group.

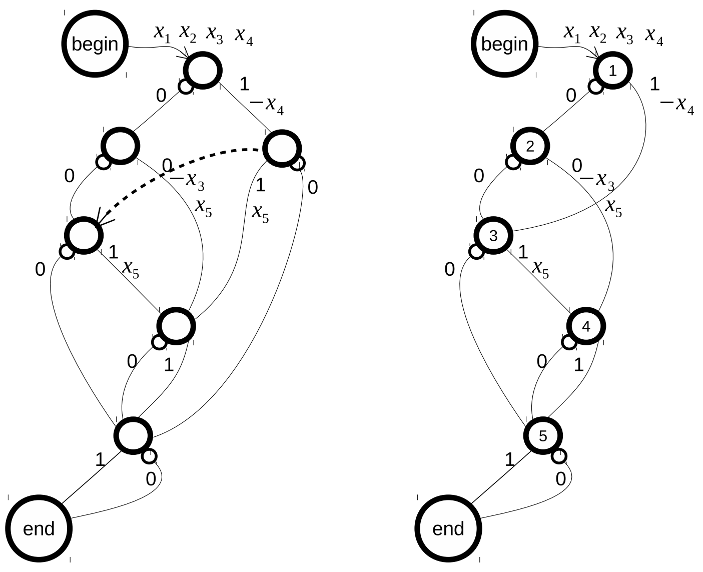
$+$	0	x	$-x$	$2 \cdot x$	$-2 \cdot x$	$\ldots$
0	0	x	$-x$	$2 \cdot x$	$-2 \cdot x$	$\ldots$
x	x	$2 \cdot x$	0	$3 \cdot x$	$-x$	$\ldots$
$-x$	$-x$	0	$-2 \cdot x$	x	$-3 \cdot x$	$\ldots$
$2 \cdot x$	$2 \cdot x$	$3 \cdot x$	x	$4 \cdot x$	0	$\ldots$
$-2 \cdot x$	$-2 \cdot x$	$-x$	$-3 \cdot x$	0	$-4 \cdot x$	$\ldots$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$

Table 2: Combining condition labels corresponding to variable x

Label x in this group means: *apply x as the branching condition at the current node.* Label $2 \cdot x$ (here the dot is not a group operation, it simply denotes the multiplicity of the label) means: *apply x as the branching condition at the current node as well as the next node.* Label $-x$ means: *skip x and use the next label in the queue as the branching condition.* This is a complex rule and will likely incur significant hardware cost in the implementation. Furthermore, to further reduce the size of the computation graph it may be necessary to allow multiple labels on each arc in the graph, which is also costly. Nevertheless, this approach may be practically beneficial for certain applications, where the aim is to minimise the number of branches. The graph obtained by labelling arcs using the group operations in Tab. 2 and its reduced variant are shown in Fig. 10.

To obtain the graph shown in Fig. 10(left) we use the following relabelling approach. Our end goal is to have 0 labels on all false arcs of the graph. To achieve that we first move variable x_1 to the topmost arc. We then remove label x_2 from the subsequent false arc by subtracting x_2 from both outgoing branches, and adding x_2 to the incoming arcs. This procedure is then repeated for all occurrences of x_3, until all false arcs are free from it, and so forth.

Let us now demonstrate how the resulting graph can be used for computation. Consider variable assignment 11010 (x_1 corresponds to the leftmost bit). Starting at the arc leaving node begin, we extract the first element from the queue and since $x_1 = 1$ we follow the true arc of the subsequent node of the graph (see Fig. 10b). The arc holds two labels: the first one tells us that we need to update the value of y by adding 1 to it: $y \leftarrow y \oplus 1 = 0 \oplus 1 = 1$, the second one tells us that condition variable x_4 will have to be skipped, therefore we erase it from the queue and proceed. These two markers do not interact and hence commute. Once both of them are handled, we arrive at the next node (denoted as 3 in Fig. 10b), and the next variable in the queue that we check is x_2. Since $x_2 = 1$ we travel along the arc leading to node 4, updating $y \leftarrow y \oplus 1 = 1 \oplus 1 = 0$



(a) Initial transformation. (b) Reduced graph.

Fig. 10: Graph Fig. 9 transformed using condition labels.

again and adding x_5 to the back of the control label queue. The next variable to check is $x_3 = 0$, hence we follow the false arc, which contains label 0 only (as a consequence of our relabelling algorithm). We do nothing and now check $x_5 = 0$ (remember, we erased x_4 from the queue), which leads us directly to node `end`, where we report the computation outcome: $y = 0$ (upon checking Tab. 1 we see that this is the correct value).

3.1 Labellings with two operations

The use of independent (commutative) labels allows us to model a system of Boolean functions with a single computation graph. However, the more independent labels we can have on an arc and the more different values a single label can take, the fewer opportunities for graph reduction will be available, because the number of 'unmergeable' nodes grows. As we have seen in the previous subsection, by introducing a richer algebraic structure to labels sometimes allows us to find new opportunities for graph reductions without sacrificing the canonicity of the representation. We can take this idea further, and consider labels that form richer algebraic structures with two operations, that still guarantee unique solutions to the system of equations arising in the process of graph relabelling. For finite sets of labels such algebras are *Galois fields*; for infinite sets – *division rings*.

An algebraic structure with two operations is a powerful tool for graph relabelling. We can associate pairs of labels with an arc of a computation graph: one for the additive component, and another for the multiplicative one. The additive component is added to the labels reachable during the computation, while the multiplicative one is multiplied by them.

The relabelling process is arranged in two stages. In the first stage we associate additive labels with all arcs making sure that false arcs contain 0 values (as before). In the second stage, we add multiplicative labels to all arcs except for those pointing to node `end`. We then normalise the pairs so that false arcs contain exactly the pair $(0, 1)$, the additive zero and the multiplicative identity.

Let us clarify the above using an example of a system of four Boolean functions of four input variables $x_1..x_4$ describing a 2-bit binary multiplier. Tab. 3 shows the truth table for all four functions $y_1..y_4$ of the multiplier.

x_1	x_2	x_3	x_4	y_1	y_2	y_3	y_4
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	0	1	0
0	1	1	1	0	0	1	1
1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	1	0	0
1	0	1	1	0	1	1	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	0	0	1	1	0
1	1	1	1	1	0	0	1

Table 3: Truth table of a 2-bit binary multiplier

Let us use 4-bit labels with addition $+$ being component-wise addition modulo 2, and multiplication $*$ defined modulo an irreducible polynomial $e^4 + e + I$ where $e = 0010$ is the generator. Below are all $2^4 = 16$ labels of the resulting algebraic structure:

- $e = 0010$ – the generator,
- $e^2 = 0100$,
- $e^3 = 1000$,
- $e^4 = e + I = 0011$,
- $e^5 = e^2 + e = 0110$,
- $e^6 = e^3 + e^2 = 1100$,
- $e^7 = e^3 + e + I = 1011$,
- $e^8 = e^2 + I = 0101$,

- $e^9 = e^3 + e = 1010$,
- $e^{10} = e^2 + e + I = 0111$,
- $e^{11} = e^3 + e^2 + e = 1110$,
- $e^{12} = e^3 + e^2 + e + I = 1111$,
- $e^{13} = e^3 + e^2 + I = 1101$,
- $e^{14} = e^3 + I = 1001$,
- $e^{15} = I = 0001$ – the identity element,
- $\emptyset = 0000$ – the zero element.

As the first step we build a computation graph according to the specification given in Tab. 3 using only additive labelling, that is using only the modulo 2 addition operation for combining labels along computation paths. The resulting graph is shown in Fig. 11.

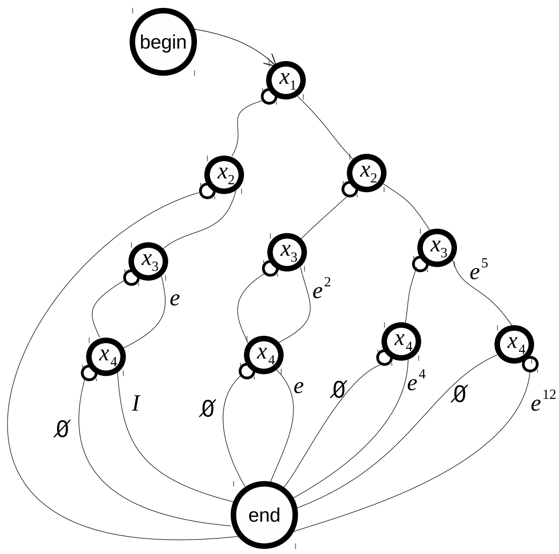


Fig. 11: Additive labelling.

There are no pairs of equivalent nodes, hence the obtained graph cannot be reduced. However, we can now relabel the graph using the multiplication operation, by factoring out common factors from outgoing arcs and moving them up to the incoming arcs in a canonical manner. After applying the relabelling to the lower layer of the graph we obtain a new graph shown in Fig. 12. We explain the used notation for new labels below.

We use so-called *Polish prefix notation* for compact representation of the effect (function) that additive and multiplicative labels have on values travelling along arcs. For example, label $+\emptyset * I$ should be interpreted as $y \leftarrow \emptyset + I * y$, where y represents the current value of the computation, just as in the examples

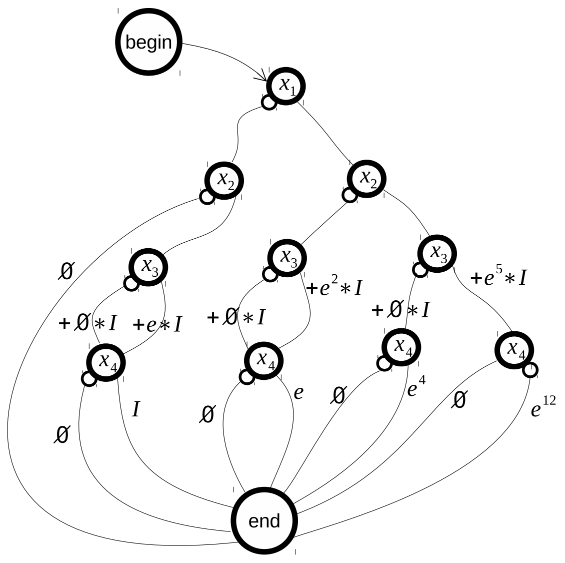


Fig. 12: Multiplicative labelling of the lower layer of the graph in Fig. 11.

before. The use of Polish notation allows us to avoid parentheses on the arcs of the graph. Alternatively, one can use *lambda calculus* for a more general and familiar (yet somewhat more verbose) representation.

Let us now define the relabelling rule using the properties of addition and multiplication of our Galois field. As before, consider a node x_n with incoming and outgoing arcs as shown in Fig. 13. If we would like to relabel the graph by changing a to g, how do we need to change the other labels?

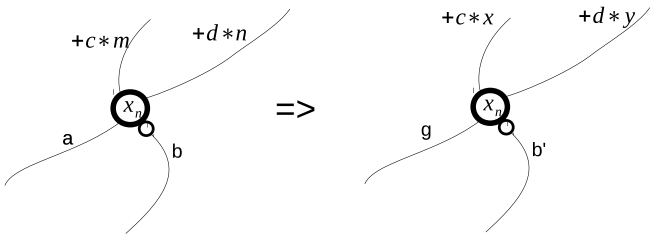


Fig. 13: Relabelling with two operations.

The following four equations need to be satisfied:

$$\begin{cases} c + x * g = c + a * m, \\ d + y * g = d + a * n, \\ c + x * b' = c + b * m, \\ d + y * b' = d + b * n. \end{cases} \tag{2}$$

Unique solutions of (2) have the following form:

$$\begin{cases} x = g^{-1} * a * m, \\ y = g^{-1} * a * n, \\ b' = g * a^{-1} * b. \end{cases} \tag{3}$$

Using the obtained solution we can complete the relabelling procedure of the lower layer of our example graph, as shown in Fig. 14.

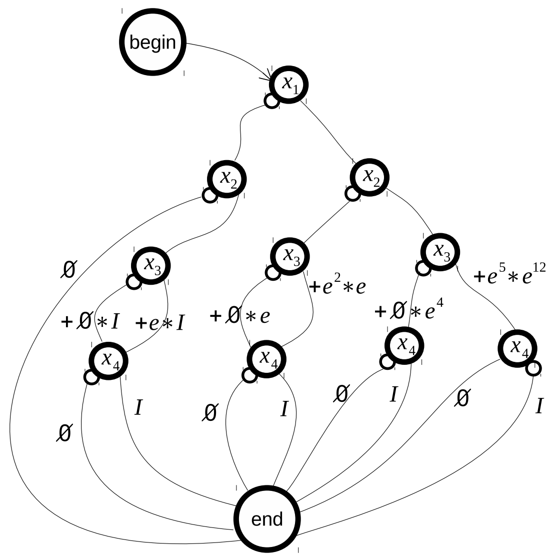


Fig. 14: Canonical relabelling of the lower layer of the graph in Fig. 12.

After the canonical relabelling all nodes of the lower layer become equivalent and can therefore be merged as shown in Fig. 15.

We continue the process by relabelling the previous (third) layer, leading the graph shown in Fig. 16 containing two new equivalent nodes that will further be merged. By proceeding analogously we complete the graph relabelling obtaining the graph shown in Fig. 17.

How do we use the obtained computation graph? We start at the node begin and traverse the graph according to the values of conditions $x_1..x_4$ until we reach

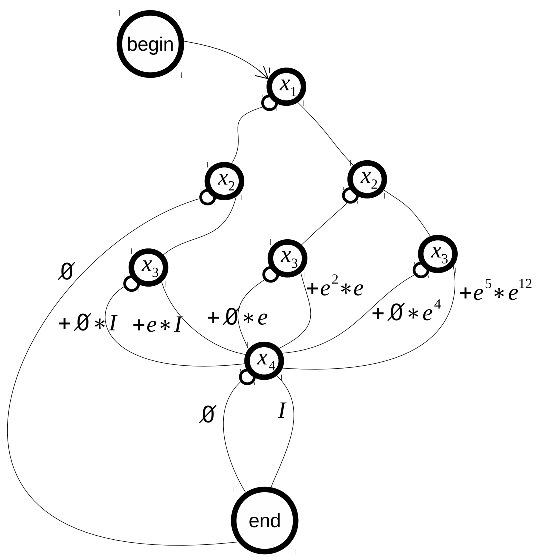


Fig. 15: Merging equivalent nodes of the lower layer in the graph in Fig. 14.

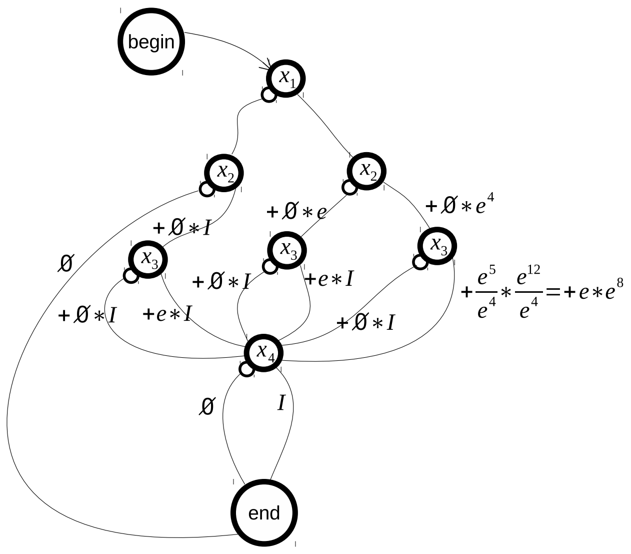


Fig. 16: Relabelling of the third layer.

the node end, writing down the computation result in the form of an expression in Polish prefix notation. We then evaluate the expression and interpret the result as a 4-bit Boolean vector $y_1..y_4$. For example, let $x_1 = x_2 = x_3 = x_4 = 1$, i.e. the input Boolean vector is 1111. This vector corresponds to the rightmost

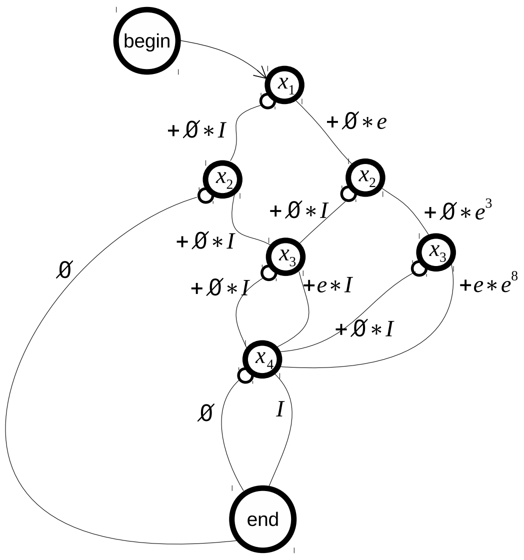


Fig. 17: Complete graph relabelling.

computation path in the graph in Fig. 17. The resulting expression is $(+\emptyset * e + \emptyset * e^3 + e * e^8)I$. We clarify the evaluation of the expression in Fig. 18. The resulting Boolean vector is 1001, which matches the specification in Tab. 3.

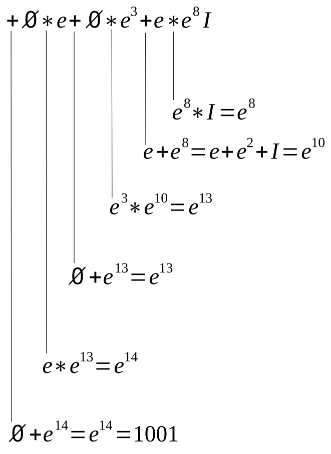

$+\emptyset * e + \emptyset * e^3 + e * e^8 I$
$e^8 * I = e^8$
$e + e^8 = e + e^2 + I = e^{10}$
$e^3 * e^{10} = e^{13}$
$\emptyset + e^{13} = e^{13}$
$e * e^{13} = e^{14}$
$\emptyset + e^{14} = e^{14} = 1001$

Fig. 18: Evaluating of the resulting expression for input 1111.

The presented approach to the reduction of the size of computation graphs is inspired by *differential BDDs* [5].

b1	b2	b3	b4	b5	Operation
0	0	0	0	0	$r_1; \emptyset; \emptyset; \emptyset; \emptyset; \emptyset; \emptyset; \emptyset; \emptyset; \emptyset$
				1	$r_1; r_2; \emptyset; \emptyset; \emptyset; \emptyset; \emptyset; \emptyset; \emptyset; \emptyset$
			1	0	$r_1; r_2; r_3; \emptyset; \emptyset; \emptyset; \emptyset; \emptyset; \emptyset; \emptyset$
				1	$r_1; r_2; r_3; r_2; \emptyset; \emptyset; \emptyset; \emptyset; \emptyset; \emptyset$
		1	0	0	$r_1; r_2; r_3; r_3; \emptyset; \emptyset; \emptyset; \emptyset; \emptyset; \emptyset$
				1	$r_1; r_2; r_3; r_3; r_2; \emptyset; \emptyset; \emptyset; \emptyset; \emptyset$
			1	0	$r_1; r_2; r_3; r_2; r_3; \emptyset; \emptyset; \emptyset; \emptyset; \emptyset$
				1	$r_1; r_2; r_3; r_2; r_3; r_2; \emptyset; \emptyset; \emptyset; \emptyset$
	1	0	0	0	$r_1; r_2; r_3; r_3; r_3; \emptyset; \emptyset; \emptyset; \emptyset; \emptyset$
				1	$r_1; r_2; r_3; r_3; r_3; r_2; \emptyset; \emptyset; \emptyset; \emptyset$
			1	0	$r_1; r_2; r_3; r_3; r_2; r_3; \emptyset; \emptyset; \emptyset; \emptyset$
				1	$r_1; r_2; r_3; r_3; r_2; r_3; r_2; \emptyset; \emptyset; \emptyset$
		1	0	0	$r_1; r_2; r_3; r_2; r_3; r_3; \emptyset; \emptyset; \emptyset; \emptyset$
				1	$r_1; r_2; r_3; r_2; r_3; r_3; r_2; \emptyset; \emptyset; \emptyset$
			1	0	$r_1; r_2; r_3; r_2; r_3; r_3; r_2; \emptyset; \emptyset; \emptyset$
				1	$r_1; r_2; r_3; r_2; r_3; r_2; r_3; r_2; \emptyset; \emptyset$
1	0	0	0	0	$r_1; r_2; r_3; r_3; r_3; r_3; \emptyset; \emptyset; \emptyset; \emptyset$
				1	$r_1; r_2; r_3; r_3; r_3; r_3; r_2; \emptyset; \emptyset; \emptyset$
			1	0	$r_1; r_2; r_3; r_3; r_3; r_2; r_3; \emptyset; \emptyset; \emptyset$
				1	$r_1; r_2; r_3; r_3; r_3; r_2; r_3; r_2; \emptyset; \emptyset$
		1	0	0	$r_1; r_2; r_3; r_3; r_2; r_3; r_3; \emptyset; \emptyset; \emptyset$
				1	$r_1; r_2; r_3; r_3; r_2; r_3; r_3; r_2; \emptyset; \emptyset$
			1	0	$r_1; r_2; r_3; r_3; r_2; r_3; r_2; r_3; \emptyset; \emptyset$
				1	$r_1; r_2; r_3; r_3; r_2; r_3; r_2; r_3; r_2; \emptyset$
	1	0	0	0	$r_1; r_2; r_3; r_2; r_3; r_3; r_3; \emptyset; \emptyset; \emptyset$
				1	$r_1; r_2; r_3; r_2; r_3; r_3; r_3; r_2; \emptyset; \emptyset$
			1	0	$r_1; r_2; r_3; r_2; r_3; r_3; r_2; r_3; \emptyset; \emptyset$
				1	$r_1; r_2; r_3; r_2; r_3; r_3; r_2; r_3; r_2; \emptyset$
		1	0	0	$r_1; r_2; r_3; r_2; r_3; r_2; r_3; r_3; \emptyset; \emptyset$
				1	$r_1; r_2; r_3; r_2; r_3; r_2; r_3; r_3; r_2; \emptyset$
			1	0	$r_1; r_2; r_3; r_2; r_3; r_2; r_3; r_2; r_3; \emptyset$
				1	$r_1; r_2; r_3; r_2; r_3; r_2; r_3; r_2; r_3; r_2$

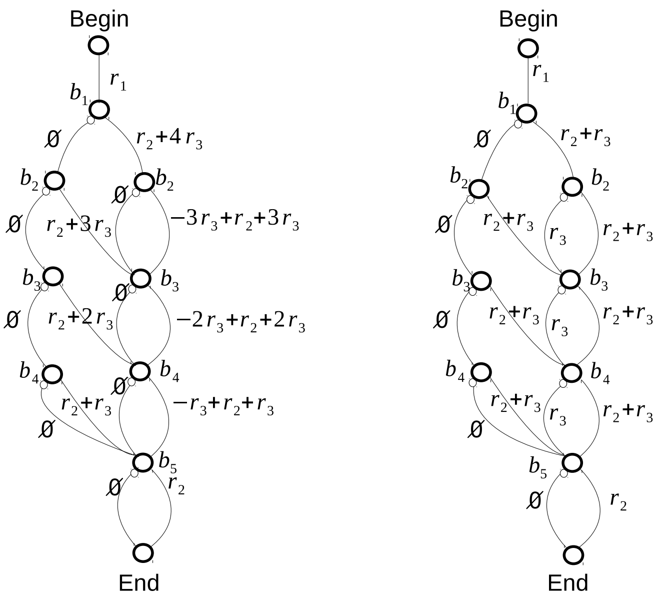
Table 4: Decision tree for a 5-bit sequential multiplier.

4 Example of a sequential computation process

Consider a sequential computation process corresponding to the multiplier of 5-bit non-negative integer numbers. We have a 10-bit asynchronous accumulating register Y that supports three operations: i) add a given 5-bit value A to its current content: $Y \leftarrow Y + A$, ii) double the currently stored value: $Y \leftarrow Y + Y$, and iii) reset to zero: $Y \leftarrow 0$. The complete binary decision tree for all input vectors is shown in Tab. 4; the table uses the following notation for brevity:

- $\emptyset$ corresponds to the no-op (doing nothing),
- r_1 corresponds to the reset operation: $Y \leftarrow 0$,
- r_2 corresponds to addition: $Y \leftarrow Y + A$,
- r_3 corresponds to doubling: $Y \leftarrow Y + Y$.

A list of operations separated by semicolons corresponds to the sequential execution of listed operations from left to right.



(a) Canonical. (b) Not canonical.

Fig. 19: Computation graphs for the 5-bit multiplier.

The leafs of the decision tree are all distinct, therefore the only way to achieve any reduction in the size of the computation graph is to introduce an algebraic structure on the arc labels. Let us first consider label r_1, which stands for the reset of the register. To define an additive group we use $\emptyset$ as the zero label, and introduce the negative label $-r_1$, which stands for *undo the reset* operation. Supporting such undo operations in hardware may be overly expensive, therefore we do not want them to appear the final computation graph, however we are still free to use them analytically in intermediate derivations. We extend the additive group on labels r_2 and r_3 in a similar manner, although their negative labels have more straightforward hardware implementations: $-r_2$ means *subtract A from Y*: $Y \leftarrow Y - A$, while $-r_3$ means *divide Y by 2*: $Y \leftarrow Y/2$. The resulting group is defined by the following equations:

- $r_1 + r_1 = 2 \cdot r_1,\ r_1 - r_1 = \emptyset, -r_1 - r_1 = -2 \cdot r_1,\ \ldots;$
- $r_2 + r_2 = 2 \cdot r_2,\ r_2 - r_2 = \emptyset, -r_2 - r_2 = -2 \cdot r_2,\ \ldots;$
- $r_3 + r_3 = 2 \cdot r_3,\ r_3 - r_3 = \emptyset, -r_3 - r_3 = -2 \cdot r_3,\ \ldots.$

Using the relabelling system of equations similar to (1), one can derive the canonical computation graph shown in Fig. 19a. We have not considered the optimality with respect to the register operations, however, and one can see that some of the resulting computation labels contain redundant shifting operations (doubling labels r_3 followed by immediate division by 2 labels $-r_3$). This is a cost of choosing a particular canonical relabelling. If canonicity can be sacrificed, it is possible to obtain a more efficient labelling shown in Fig. 19b, which avoids negative labels.

References

[1] Alex Semenov, Alex Yakovlev: "Combining Partial Orders and Symbolic Traversal for Efficient Verification of Asynchronous Circuits", 1995.

[2] "Binary Decision Diagram (BDD) adiabatic charging logic circuit", EP 1331738 A1, http://www.google.com.tr/patents/EP1331738A1?cl=en

[3] Aiqun Cao, Cheng-Kok Koh: "Non-crossing ordered BDD for physical synthesis of regular circuit structure", School of Electrical and Computer Engineering, Purdue University West Lafayette, IN 47907-1285, 2003.

[4] Robert Wille, Oliver Keszocze, Clemens Hopfmuller, Rolf Drechsler: "Revers BDD-based synthesys for splitter-free optical circuits", Institute of Computer Science, University of Bremen, Germany, Cyber Physical Systems.

[5] Anuchit Anuchitanukul, Zohar Manna, Toms E. Uribe: "Differential BDDs", In J. van Leeuwen, ed, Computer Science Today , Lecture Notes in Computer Science, vol. 1000, pp. 218-233, Springer-Verlag, 1995.