PTM: A Technology Mapper for Pass-Transistor Logic

Nan Zhuang, Marcus v. Scotti and Peter Y.K. Cheung[1]

Abstract

Pass-Transistor Mapper (**PTM**), a logic synthesis tool specifically designed for pass-transistor based logic library that has only three basic cells, is reported. It exploits the close relationship between BDD representation of logic and the structure of pass-transistor logic cells to ensure efficient technology mapping. BDD variable order optimization is achieved through a genetic algorithm with dynamic parameters. Unlike **LEAP**, the only previously reported system for pass-transistor logic, **PTM** integrates both synthesis and logic optimization in one step and can be used for large logic functions. Results of using **PTM** on a large set of benchmarks are compared to that from Berkeley's **SIS** using the MCNC CMOS cell library and are found to be promising.

Indexing terms: *CMOS, Pass-Transistor, Logic synthesis, Technology Mapping*

1 Introduction

It has been demonstrated that CMOS pass-transistor based logic can often result in high-speed and high-density circuits. For example, by using pass-transistor logics, a $3.8ns$ $0.5\mu m$ CMOS 16×16-b multiplier was implemented in 1990 [1], a $1.5ns$ $0.25\mu m$ 32b CMOS ALU was developed in 1993 [2], and a $4.4ns$ $0.25\mu m$ CMOS 54×54-b multiplier was designed in 1995 [3].

The most important difference between conventional CMOS cells and pass-transistor cells is that in conventional logic the inputs can only be connected to the gate of transistors, and all inputs are symmetrical. In the pass-transistor logic inputs can be connected to both gates and drains,

[1]The authors are with the Dept. of Electrical and Electronic Engineering, Imperial College London, UK, SW7 2AZ

and changing the input configuration will correspond to different Boolean functions. It implies that pass-transistor based circuits are more flexible. Unfortunately, traditional state-of-the-art logic synthesis tools can no longer be used in pass-transistor logic design if the full potential of pass-transistor logic is to be exploited. Due to the lack of the automatic design tools for general logic functions, pass-transistor logic is currently mainly restricted to the design of arithmetic macros [1]–[4].

A research group at Hitachi proposed a synthesis package, **LEAP**(Lean Integration Pass-transistor), for pass-transistor logic [5]. In **LEAP** a very simple pass-transistor cell library is proposed as shown in Figure 1, where Y1, Y2, and Y3, which are essentially multiplexers, form the basic logic cells. Completing the library are simple inverters with different drive capabilities. The core idea of **LEAP** is (a) to express the required logic function with a reduced and shared BDD [6], and (b) to partition the BDD into the smaller trees which can be mapped into one of the basic cells in library. Although **LEAP** shows very positive results for small designs, it falls short in the following:

(1) it does not have an optimization capability;

(2) it can not handle large logic functions.

In this paper, we describe an improved pass-transistor synthesis tool called Pass-Transistor Mapper (**PTM**). In **PTM**, we use the same cell library as [5], where Y1, Y2, and Y3 are used as the essential logic cells, and the inverters are used to realize the negation operation of input variables. Compared with **LEAP**, **PTM** has the following improved features:

(1) it employs optimized ROBDD [7]–[9];

(2) it can handle large logic functions;

(3) it exploits state-of-the-art logic synthesis techniques for pass-transistor technology mapping.

PTM is a sophisticated package composed of a group of algorithms. The BDD ordering using Genetic Algorithm is explained in Section 2. Pass-transistor based technology mapping using Boolean matching and greedy covering algorithm is proposed in Section 3. The inverter optimization algorithm, netlist compiling algorithm, and **PTM** package implementation are described in Section 4. Section 5 discusses the test results for large set of MCNC benchmarks using Mentor Graphics' GDT environment for placement, routing and verification, and conclusions are given in Section 4.

2 BDD size minimization using Genetic Algorithm

2.1 Problem Formulation

It is well known that multiplexer based structures are directly related to BDDs, and finding the optimal data-select variable ordering of the multiplexer network is equivalent to finding the good variable ordering for BDDs [10, 11]. Since reduced ordered BDD (ROBDD) has been shown to be an efficient representation of Boolean functions for logic synthesis and verification [6], it is employed in our algorithm. The first step of our synthesis algorithm is to minimize the size of the ROBDD, and then the ROBDD is mapped to the pass-transistor cell library.

Definition 1: An OBDD is a rooted directed graph $G = (V, E)$. The vertex set V is composed of two kinds of vertex, non-terminal and terminal. Each non-terminal vertex has as attributes a pointer $index(v) \in \{1, 2, ..., n\}$ to an input variable in the set $\{x_1, x_2, ..., x_n\}$, and two children $low(v), high(v) \in V$. A terminal vertex v has as an attribute a value $value(v) \in \mathbf{B}(\mathbf{B} \in \{0, 1\})$. The edge set E is composed of negative edges and positive edges.

Definition 2: BDD size, $|BDD|$, is given by its number of non-terminal nodes. For a certain variable ordering, BDD size can be reduced with the following three rules:

Deletion Rule (R1): The node v with label x_i can be deleted, if and only if the two successors $low(v)$ and $high(v)$ of v represent the same subfunction of f, i.e. the subfunction represented a v does not depend essentially on x_i.

Merging Rule (R2): Nodes v and w with label x_i can be shared, if and only if they represented the same subfunction of f.

Complement Rule (R3): For nodes v and w with label x_i, w can be replaced with v', if and only if $v = w'$.

Definition 3.: A BDD is called a reduced ordered binary decision diagram(ROBDD) if no reductions can be achieved using the above rules.

ROBDD size is extremely sensitive to the choice of the variable ordering. The existing methods of finding a good variable ordering can be classified into three categories: heuristic methods [12, 13], exact methods [14], and Genetic Algorithms (**GA**) methods [7]–[9]. Previous research indicated that GA methods can produce better results with reasonable CPU time for most benchmark circuits. In this paper a novel GA [8] which uses dynamic parameters is employed to search for near optimal variable ordering.

Definition 4: For any Boolean function $f(x_1, x_2, ..., x_n)$, the decision variable ordering (from the root to the terminals of the data structure) can be defined with a vector $Order[n] = \{x_{k_1}, x_{k_2}, ..., x_{k_n}\}$.

For a certain $Order[n]$, the size of the ROBDD, $BDD_Size()$, is the number of the non-terminal vertex in the data structure. Our goal is to find a good variable ordering $\{x_{k_1}, x_{k_2}, ..., x_{k_n}\}$ for a given function $f(x_1, x_2, ..., x_n)$, such that $BDD_Size()$ is minimized.

2.2 The Dynamic Parameter Genetic Algorithm

The optimal variable ordering used in calculating the BDD is found with a Genetic Algorithm (GA), details of which can be found in [18]. A major factor that affects the performance of GAs

is the size of the population from which new generations are evolved. A small population size would tend to limit the search in a confined area. A large population size will spread the search area wider and reduce the chance of producing good offspring as result of the diversification of the parents.

Mutation is another important mechanism in GA which prevents a solution being trap in a local minima. Unfortunately, setting too high a mutation rate can cause the algorithm to escape from solution areas which are already close to the global optimum, while too low a rate might cause the algorithm to be trapped in local minima.

In the new algorithm, we propose to use dynamic population size and mutation rate as defined below:

```
if (improvement found in new generation)then
    decrease Population_Size by δ1;
    decrease Mutation_Rate by μ;
else
    increase Population_Size by δ2;
    increase Mutation_Rate by μ;
    Population_Size := max (MIN_SIZE,min(Population_Size, MAX_SIZE));
    Mutation_Rate := max (MIN_RATE, min (Mutation_Rate, MAX_RATE));
```

Here δ_1, δ_2 and μ are constants and are determined experimentally. Whenever improvements are found, the algorithm is exploring in an area which is already promising. Therefore it is reasonable to concentrate the search efforts and avoid increasing the exploration space. It is also reasonable to reduce the chance of escaping from this search area. On the other hand, if no improvement is found, it is worth enlarging the search area by increasing both the population size and mutation rate to provide a better chance of jumping out of local minima. This algorithm therefore avoids the use of too small or too large a search area and too many or too few random mutations.

In conventional GA implementations, the algorithm is terminated either after a fixed number of generations, or when no improvements are found in a fixed number of iterations. In our

algorithm, we allow the stop criteria of the algorithm to adapt to improvement as follow:

```
if (improvement found in new generation) then
    increase Generations_Remaining by γ1;
else
    decrease Generations_Remaining by γ2;
if (Population_Size ≥ MAX_SIZE) then
    decrease Generations_Remaining by γ3;
```

γ_1, γ_2 and γ_3 are empirical constants. While improvement is found, it is reasonable to spend more computing efforts in searching for a better solution by increasing the number of generations to evolve and vice versa. In addition, if *Population_Size* reaches a predetermined *MAX_SIZE*, it signifies that no improvement has been found for a number of iterations and the termination of the evolutionary loop is accelerated.

The pseudocode for the dynamic genetic algorithm for optimum variable ordering is given as follows:

```
Dynamic_Genetic_Algorithm(benchmark)
{
    for(K1 = 0, K1 <= MAX_POPULATION) {
        Random_Order();
        BDD_Size();
        Initial_Roulette_Wheel_Selection();
    }
    while(Generation_Remaining) {
        Crossover();
        Mutation();
        BDD_Size();
        random_number = rand()%100;
        if(random_number < HYBRID_RATE)Heuristic_Parent_Selection();
        else Roulette_Wheel_Parent_Selection();
        Elitism();
        if(improvement found in new generation) {
            decrease Population_Size by δ1;
            decrease Mutation_Rate by μ;
            increase Generation_Remaining γ1;
        } else {
```

```
        increase Population_Size by δ2;
        increase Mutation_Rate by μ;
        decrease Generation_Remaining by γ2;
      }
      if(Population_Size >= MAX_POPULATION)
        decrease Generation_Remaining by γ3;
      Population_Size = max(MIN_POPULATION, min(Population_Size,
                          MAX_POPULATION));
      Mutation_Rate = max(MIN_RATE, min(Mutation_Rate, MAX_RATE));
    }
    Copy_Best_Ordering();
}
```

3 Pass-transistor based Boolean Matching and Greedy Covering

Boolean matching, a key operation of the technology mapping process, is a technique to recognize logic equivalences between two functions, one representing a sub-network in the subject graph (called *cluster function*), and the other representing a library cell (named *pattern function*). In the subject graph each internal vertex has unit fanout.

Definition 1: A cluster function is denoted by: $\mathcal{F}(x_1, x_2, ...x_n)$. It has n inputs and one output. The phase of variable x_i is denoted by: $\phi \in \{0, 1\}$, where, $x_i^{\phi_i} = x_i$ for $\phi_i = 1$, and $x_i^{\phi_i} = \bar{x}_i$ for $\phi_i = 0$. A library is denoted by: $\xi : \{\mathcal{G}_1, \mathcal{G}_2, ..., \mathcal{G}_k\}$. The pattern function $\mathcal{G}_j$ is a multiple-inputs single-output function.

Definition 2: Given a cluster function $\mathcal{F}(x_1, x_2, ...x_n)$ and a pattern function $\mathcal{G}(y_1, y_2, ...y_m)$, find an ordering $\{o_1, o_2, ...o_n\}$ and a phase assignment $\{\phi_1, \phi_2, ..., \phi_n\}$ of the input variable of $\mathcal{F}$ such that $\mathcal{F}$ matches $\mathcal{G}$ if $\mathcal{F}(x_{o_1}^{\phi_{o_1}}, ..., x_{o_n}^{\phi_{o_n}}) = \mathcal{G}(y_1, y_2, ...y_m)$ is a tautology. In other words, Boolean matching addresses the question of whether an input variable Negation matrix $\mathbf{N}$, and an input variable Permutation matrix $\mathbf{P}$ exist such that $\mathcal{F}(X) = \mathcal{G}(\mathbf{PN}X)$ is a tautology.

There are three pattern functions corresponding to library cells Y1, Y2, and Y3 [5] as follows:

$$g_1 = (x_bar \cdot A + x \cdot B)' \quad (1)$$

$$g_2 = (x_bar \cdot A + x \cdot (y_bar \cdot B + y \cdot C))' \quad (2)$$

$$g_3 = (x_bar \cdot (y_bar \cdot A + y \cdot B) + x \cdot (y_bar \cdot C + y \cdot D))' \quad (3)$$

The relationship between a ROBDD and the cluster function it represents can be described as follow:

Corollary 1: An OBDD with root v denotes a cluster function $f_v : \mathbf{B}^n \rightarrow \mathbf{B}$ such that:

(1) If v is a terminal vertex with $value(v) = 1$, then $f_v = 1$;

(2) If v is a terminal vertex with $value(v) = 0$, then $f_v = 0$;

(3) If v is a non-terminal vertex and $index(v) = i$, then $f_v = x_i' \dot{f}_{low(v)} + x_i \dot{f}_{high(v)}$,

where, $\dot{f}_{low(v)} = f_{low(v)}$ if $edge(v, low(v))$ is positive, and $\dot{f}_{low(v)} = f'_{low(v)}$ if $edge(v, low(v))$ is negative. $\dot{f}_{high(v)}$ can be defined in a similar way.

In the normal Boolean matching algorithms [15] the upper bound on the number of variable permutation to be considered in the search of a matching is $\prod_{i=1}^{n} |C_i^b|! \cdot (|C_i| - |C_i^b|)!$, where n is the variable number of the cluster function, b is the number of the binate variables, and C_i is the symmetry class whose elements have cardinality i. In our package the matching is realized by traversing the OBDD. Because the multiplexer based cell library is strongly related to the BDD data structure, a sub-BDD can always be matched by one of the three cells in the library. The task of the matching algorithm, *pass_matching*(), is to find the input variable Negation matrix **N**, and Permutation matrix **P**. *pass_matching*() can be described as follows:

```
struct model {
    P *permu; /* permutation matrix */
    N *nega; /* negation matrix */
```

```
        char *cell; /* name of the library cell */
}
pass_matching(v, ξ)
{
        if(value(v) == 0 OR value(v) == 1) {
            model-> cell = constant;
            return 0;
        }
        if(fanout_num(low(v)) == 1 AND fanout_num(high(v)) == 1) {
            model-> cell = Y3;
            model-> permu = get_permutation_matrix(v, Y3);
            model-> nage = get_negation_matrix(v, Y3);
            array_insert_last(model*, matching_node_array, model);
        } else if(fanout_num(low(v)) == 1 OR fanout_num(high(v)) == 1) {
            model-> cell = Y2;
            model-> permu = get_permutation_matrix(v, Y2);
            model-> nage = get_negation_matrix(v, Y2);
            array_insert_last(model*, matching_node_array, model);
        }
        model-> cell = Y1;
        model-> permu = get_permutation_matrix(v, Y1);
        model-> nage = get_negation_matrix(v, Y1);
        array_insert_last(model*, matching_node_array, model);
        return 1;
}
```

$pass_matching(v, \xi)$ is called repeatedly during the traversal of the BDD. All matchings are stored in the array $matching_node_array$. The algorithm is of time complexity $O(|BDD|)$, which is much faster than the Boolean matching algorithm in [15]. The function $get_permutation_matrix()$ is quite straightforward. For example, the permutation matrix for the Y3 cell will be $(x_{index(v)}\ x_{index(low(v))}$ $low(low(v))\ low(high(v))\ high(low(v))\ high(high(v)))$. The negation matrix is determined by the negative edges in the BDD. For example, for the Y1 cell, if edge $(parent(v), v)$ is positive, edge $(v, low(v))$ is negative, and edge $(v, high(v))$ is positive, the Negation matrix will be (0 1). A greedy covering algorithm is employed in this package. The subject graph is not required to be a tree, but just a rooted DAG for Boolean covering. There is always at least one match for each vertex because the library contains a multiplexer, which is strongly related to the BDD

data structure. When multiple matches exist at a given vertex, the algorithm selects the biggest covering match. This is because according to the library circuits in Figure 1, the bigger the cell chosen, the more efficient it is in area and delay. The selection priority in the pass-transistor library is therefore Y3, Y2, and Y1. For example, for vertex j in Figure 2, there are three matches, where *match 1*, which related to Y1 cell in the library, is smaller than *match 2* and *match 3*. So, *match 1* is ignored. *Match 2* and *match 3* are with the same size. The algorithm will arbitrary select one of them. The complexity of the algorithm is linear with respect to the size of the subject graph.

4 PTM Implementation

After the covering algorithm, a gate interconnection netlist based on the pass-transistor library has to be generated. However, to further optimize the synthesised circuit and to provide a suitable interface with external physics design tools, such as Mentor Graphic's GDT, the following synthesis and interface algorithms are used.

4.1 Inverter optimization

During technology mapping described in the last Section, some redundant inverters would inevitably be introduced. We use the algorithm *red_inv_rem*() to remove these parallel and serial redundant inverters. If there are i ($i > 1$) inverters fanout from the same node, the algorithm will remove $i - 1$ parallel redundant inverters from the network. If an inverter's fan-in is also an inverter, the algorithm will remove both inverters from the network. The most significant difference between *red_inv_rem*() and *_reminv* in UC Berkeley SIS [16] is that *red_inv_rem*() will avoid removing buffer inverters associated with the pass-transistor library cells, because they are required to ensure correct drive capability in the circuit.

4.2 Phase optimization

The total number of inverters in the network can be further minimized via phase assignment algorithm. Here, we directly use *good_phase*() algorithm, which is available in UC Berkeley SIS [16], to perform phase optimization. *good_phase*() determines for each node whether to implement the function or its complement in order to reduce the total number of inverters.

4.3 Interface algorithm

To establish a suitable interface with the commercial physics design software, Mentor Graphic's GDT, we used the algorithm *interface_GDT*(). The algorithm can transfer the pass-transistor library based netlist into the netlist which can be accepted by GDT. It transforms a netlist format "cell node_1 node_2 ... node_n" into a router file format "cell_xx.node_yy, cell_xy.node_yz, ...,cell_zz.node_xz" used by the automatic place and route package *AutoCells*.

```
interface_GDT()
{
    Print_Header_Information() ;
    Print_Input_Nodes() ;
    Print_Output_Nodes();
    Print_Logic_Cell_Instances();
    while(cellsleft) {
        Read_Next_Cell(cell);
        if(cell == Any_Cell_Type) {
            Increase(cell.type.counter);
            while(type.node_counter <> 0) {
                Read_Cell_node(cell.node);
                if(cell.node == NEW) {
                    Start_New_List(cell.node, "cell.type, cell.type.counter, cell.node.type");
                }
                else {
                    Append_List(cell.node, "cell.type, cell.type.counter, cell.node.type");
                }
                Decrease(type.node_counter);
            }
            Reset(type.node_counter);
```

```
        }
    }
    Print_Full_List();
}
```

4.4 Verification

After synthesis, the network is usually changed from the original. It is necessary to verify the optimized network against the specification. We integrated Berkeley's formal verification algorithm $verify()$ in our package **PTM**.

The entire synthesis procedure for pass-transistor based CMOS circuits is depicted in Figure 3 **PTM** includes all algorithms within the rectangle, and has been integrated in UC Berkeley's SIS environment. It can accept the standard gate-level descriptions from HDL compilers, create the optimized netlist based on pass-transistor cells, and generate data for the layout tools.

5 Results

The numerous algorithms described in the previous sections were combined together as **PTM**. Since no other pass-transistor based synthesis tools which can test general benchmark circuits has been published, the performance of **PTM** is compared to state-of-the-art CMOS-based synthesis algorithms in UC Berkeley SIS. In SIS a gate-level description is initially optimized with the most robust technology independent synthesis script *script.rugged*. Then the optimized network is mapped into MCNC CMOS standard library, *mcnc.genlib*. *good_phase*() is also used to perform phase optimization. To make the comparison fair, both pass-transistor logic and standard CMOS logic libraries are designed for the TSMC 0.6μ process using 5V supply voltage. Since the technology allows only two metal layers metal1 is used for the cell design leaving metal2 for over-cell routing. The CMOS library has 19 different cells, the pass-transistor cell library only needed 7 different cells including inverters and buffers.

Although PTM does not use power dissipation as an optimisation parameter, a rough estimate

on the power dissipation of synthesised circuits were made. We concentrated on the capacitive switching power and neglected short-circuit, leakage and static power dissipation. Node activity for each input was assumed to be 50% and uncorrelated. Using the topology of the netlist the activity for each cell output was calculated. The capacitive load on this output was derived from a database containing input-capacitance information for every cell-type. These input-capacitance include the internal capacitances of the cells, which are derived from a statistical analysis of their hidden internal nodes. The average power dissipation can be calculated using the following formula [17]:

$$P_{avr} = a_{0->1} * C_L * Vdd^2 * f_{clk} \tag{4}$$

where $a_{0->1}$ is the probability of a '0' to '1' transition on the node, C_L is the load capacitance, V_{dd} and f_{clk} are the supply voltage and operation frequency respectively. Since V_{dd} and f_{clk} are constant for a given system, the average power for the entire circuit can be estimated by the activity-capacitance product:

$$C_{act} = \sum_{i=1}^{nodes} \left[a_i * \sum_{j=1}^{fanout_i} C_{ij} \right] \tag{5}$$

where C_{ij} is the input capacitance for the jth fanout of the ith node.

A large set of benchmark circuits were tested, and the results are shown in Table 1. Results for the pass-transistor implementation are represented as percentages of that for the CMOS implementation.

The CMOS values are used as reference, while **PTM** is given as a percentage to this value. Column "Area" indicates the area used in mm^2; "Delay" shows the critical path delay of the mapped and place-and-routed circuits in ns; "C-active" indicates the average node-activity multiplied by the load-capacitances (in pF) as mentioned above. "CPU" is the CPU time on a SUN Sparc 10 workstation in seconds. From Table 1, it can be seen that **PTM** designs are marginally larger in area when compared with standard CMOS. However, critical path delay is in general better, on average by a factor of 1.4, with occasional exceptions. Power dissipation is

also lower for PTM in most cases.

6 Conclusion

A CMOS pass-transistor based logic synthesis package, **PTM**, which can run in UC Berkeley's SIS environment is proposed in this paper. Since the cell library used is strongly related to the BDD data presentation, our algorithm is not only memory efficient, but also much faster than the Boolean matching method in [15]. To solve the NP-complete problem of BDD variable ordering, Genetic Algorithm with adaptive population size mutation rate and number of generations was used. Since mapping is achieved using a cell library consisting of only three logic cells and four buffer-inverters, adding new technologies is extremely easy. Comparing the area, speed and power of the benchmark circuits synthesis by **PTM** for pass-transistor logic and **SIS** for conventional CMOS logic, better area-power-speed product is achieved in the majority of the cases using **PTM**.

The current **PTM** only attempts to minimise the size of the mapped circuit. A future implementation will include power and timing in the optimisation cost function.

Acknowledgement

This project is supported by Mentor Graphics Corporation, USA, and Analog Devices.

References

[1] YANO K., YAMANAKA T., NISHIDA T., SAITO M., SHIMOHIGASHI K., SHIMIZU A., "A 3.8 ns CMOS 16 $\times$ 16 multiplier using complementary pass-transistor logic", *IEEE J. Solid State Circuits*, 1990, 25, (2), pp. 388–395

[2] SUSUKI M., OHKUBO N., YAMANAKA T., SHIMIZU A., SASAKI K., "A 1.5 ns 32 b CMOS ALU in double pass-transistor logic", *1993 IEEE Int. Solid-State Circuits Conf.*

Dig. Tech. papers, 1993, pp. 90

[3] OHKUBO N., SUSUKI M., SHINBO T., YAMANAKA T., SHIMIZU A., SASAKI K., NAKAGOME Y., "A 4.4ns CMOS 54 × 54-b multiplier using pass-transistor multiplexer", *IEEE J. Solid State Circuits*, 1990, 30, (3), pp. 251–257

[4] ZHUANG N., WU H., "A New Design of CMOS Full Adder", *IEEE J. of Solid-State Circuits*, 1992, 27, (5), pp. 452–456

[5] YANO K., SASAKI Y., RIKINO K., KEKI K., "Top-down pass-transistor logic design", *IEEE J. Solid State Circuits*, 1996, 31, (6), pp. 792–803

[6] BRYANT R., "Graph-based algorithms for Boolean function manipulation", *IEEE Trans. Comput.*, 1986, 35, (8), pp. 677–691

[7] ALMAINI A.E.A., ZHUANG N., BOURSET F., "Minimisation of multi-output Reed-Muller binary decision diagrams using hybrid genetic algorithm", *Electronics Letters*, 1995, 31, (20), pp. 1722-1723

[8] ZHUANG N., BENTEN M.S.T., CHEUNG P.Y.K., "Improved variable ordering of BDDs with novel genetic algorithm", *Proc. of IEEE Int. Symposium on Circuits and Systems*, May 1996, (III), pp. 414–417

[9] ALMAINI A.E.A., ZHUANG N., "Using genetic algorithm for the variable ordering of Reed-Muller binary decision diagrams", *Microelectronics Journal*, 1995, 26, (5), pp. 471–480

[10] SCHAFER I., PERKOWSKI M.A., "Synthesis of multilevel multiplexer circuits for incompletely specified multi-output Boolean functions with mapping to multiplexer based FPGA's", *IEEE Trans. on CAD*, 1993, 12, (11), pp. 1655–1664

[11] MURGAI R., SHENOY Y.N., BRAYTON R.K., SANGIOVANNI-VINCENTELLI A.L., "An improved synthesis algorithm for multiplexer-based PGAs", *Proc. of 29th ACM/IEEE Design Automat. Conf.*, 1992, pp. 380–386

[12] ISHIURA N., SAWADA H., YAJIMA S., "Minimization of binary decision diagrams based on exchanges of variable", *Proc. of IEEE Int. Conf. Comput.-Aided Design*, 1991, pp. 472–475

[13] RUDELL R., "Dynamic variable ordering for ordered binary decision diagrams", *Proc. of IEEE Int. Conf. Comput.-Aided Design*, Nov. 1993, pp. 42–47

[14] FRIENDMAN S.J., SUPOWIT K.J., "Finding the optimal variable ordering for binary decision diagrams", *IEEE Trans. on Comput.*, 1990, 39, (5), pp. 710–713

[15] MAILHOT F., DE MICHELI G., "Technology mapping with Boolean matching", *IEEE Trans. on CAD*, 1993, 12, (5), pp. 599–620

[16] SENTOVICH E.M., SINGH K.J., LAVAGNO L., MOON C., MURGAI R., SALDANHA A., H. Savoj, P.R. Stephan, and R.K. Brayton, "SIS: A system for sequential circuit synthesis", Technical report UCB/ERL M92/41, University of California, Berkeley, May 1992.

[17] CHANDRAKASAN A.P., BRODERSON R.W., "Low Power Digital CMOS Design", Kluwer Academic Publishers, 1995, ISBN 0-7923-9576-X

[18] MITCHELL M.,"An Introduction to Genetic Algorithms", MIT Press, 1996.

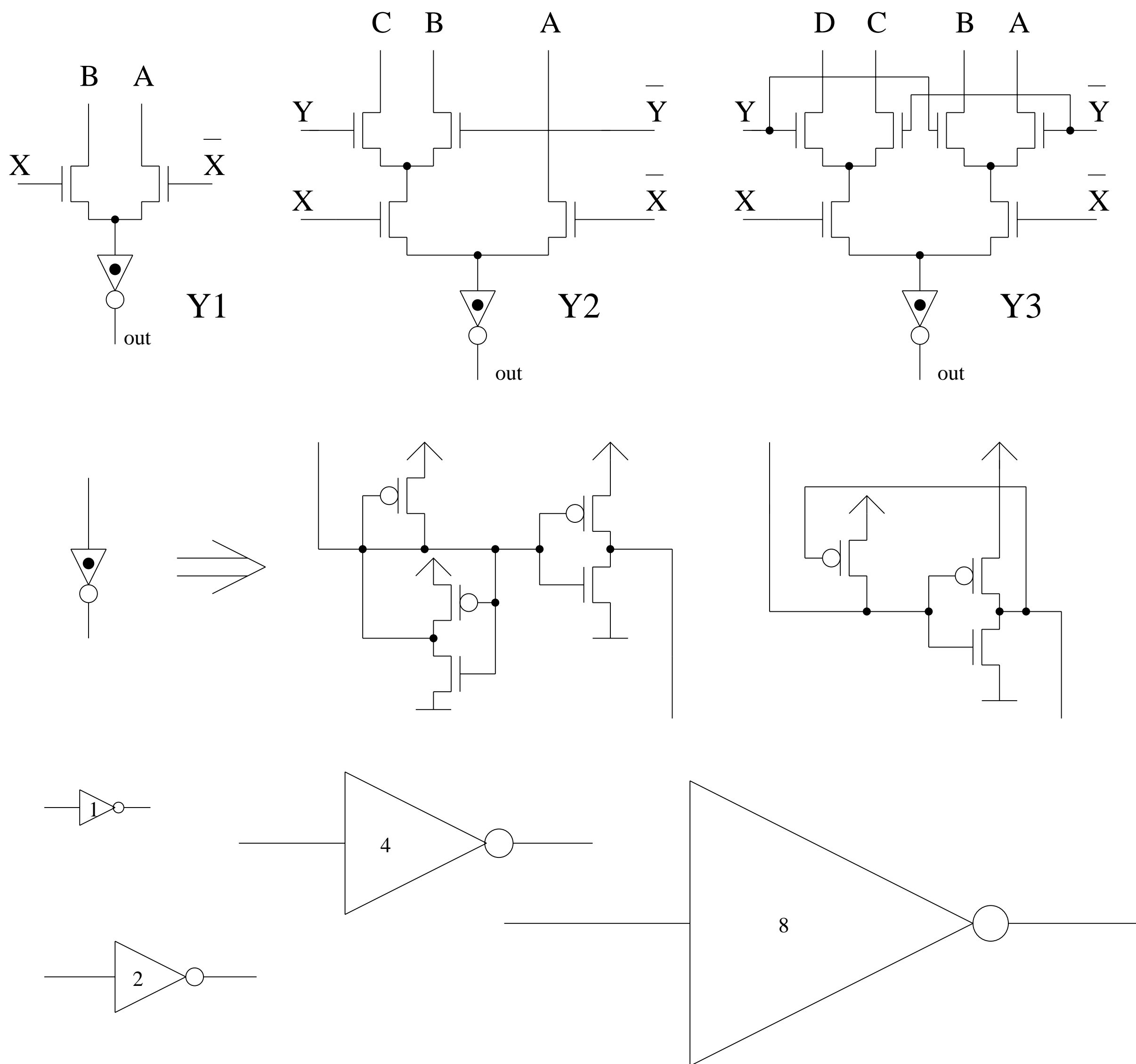

Figure 1: Pass-transistor cell library consisting of three basic cells Y1, Y2, Y3 and buffer/inverters of sizes 1,2,4 and 8

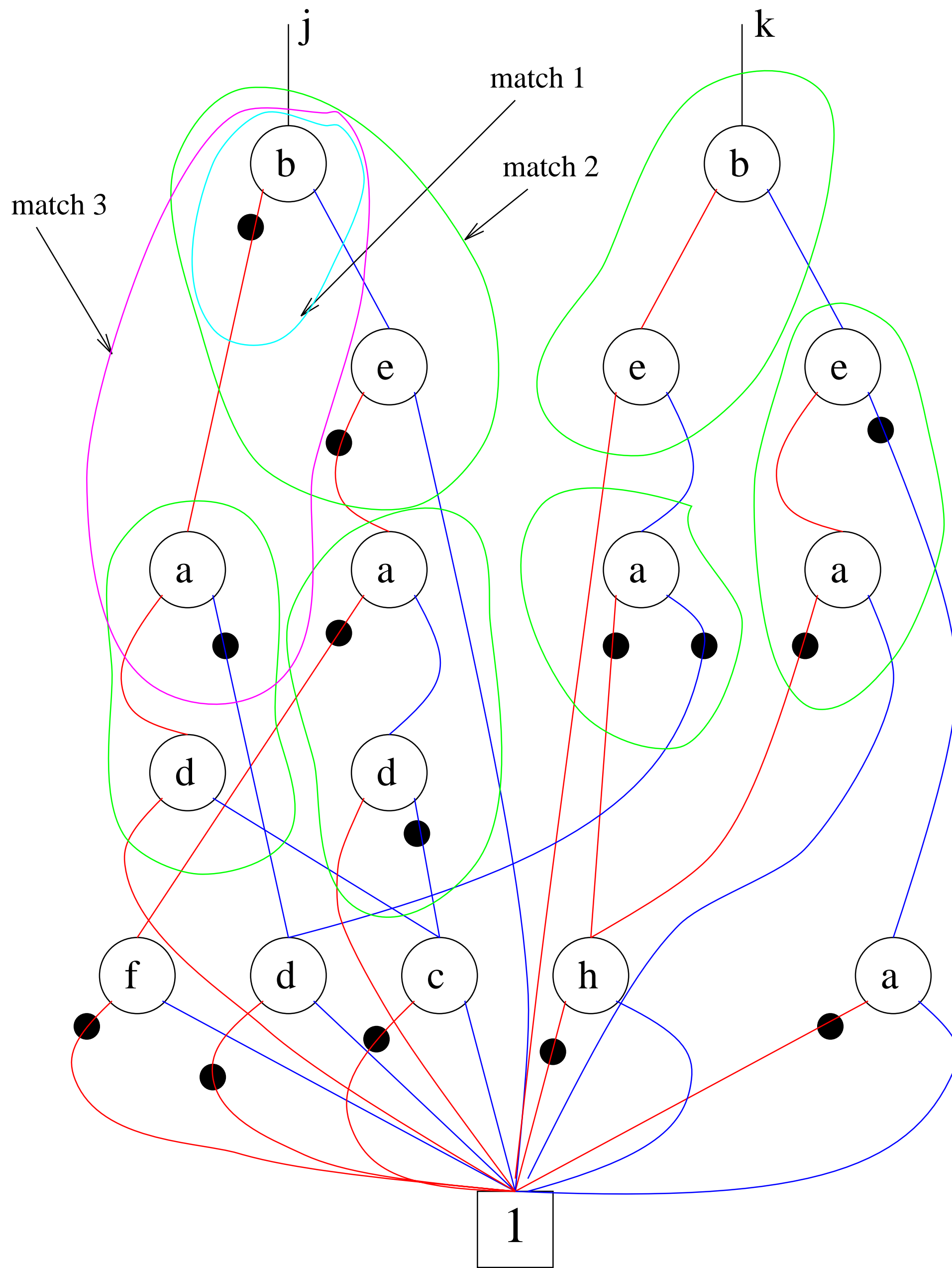

Figure 2: An example of the greedy covering (black dots depict negative edges)

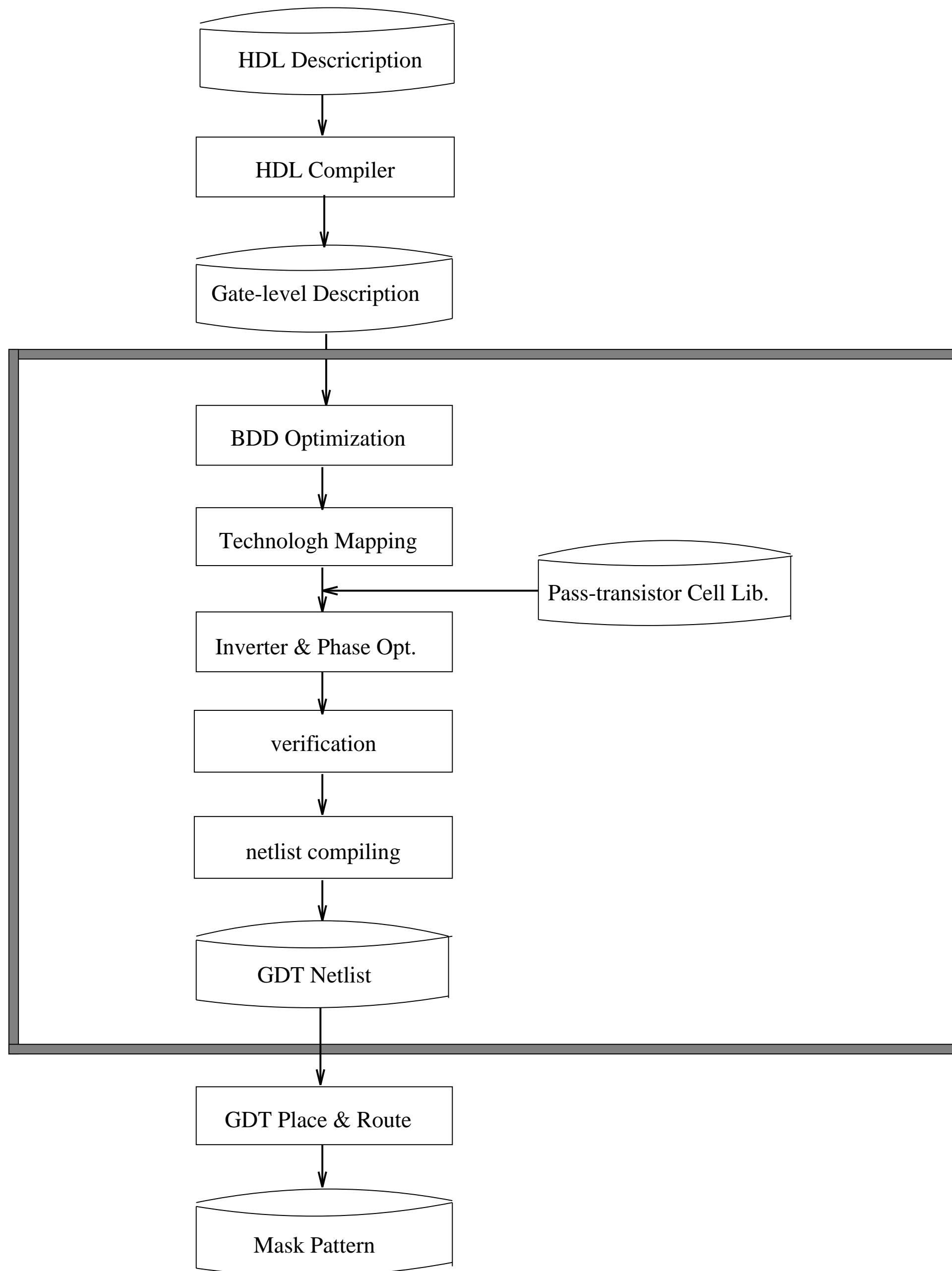

Figure 3: Schematic diagram of package **PTM** (outlined by box)

Circuits	Area (mm^2)		Delay (ns)		C-active (pF)		CPU (s)	
	CMOS	PTM (%)	CMOS	PTM (%)	CMOS	PTM (%)	CMOS	PTM (%)
5xp1	0.03610	68.1	11.61	45.0	4.264	43.5	12.400	88.7
9sym	0.07320	20.5	6.67	78.4	7.357	14.3	61.200	20.6
b12	0.02560	126.6	4.66	104.1	3.279	59.9	9.500	356.8
bw	0.05690	116.0	12.39	66.4	5.427	58.6	24.200	42.6
clip	0.03910	94.4	9.91	54.6	4.106	65.2	35.800	49.2
con1	0.00630	93.7	1.60	100.0	8.473	63.7	2.900	79.3
cordic	0.04850	69.1	4.25	316.7	2.877	70.4	2025.300	11.5
duke2	0.18330	128.8	10.30	145.9	10.197	73.1	97.700	80.2
inc	0.03310	115.1	22.42	18.1	3.545	63.7	14.300	49.7
misex1	0.01580	117.7	6.14	53.4	1.984	65.3	6.100	41.0
misex3	0.24840	123.8	12.52	111.6	15.841	77.7	916.200	65.5
rd53	0.00960	92.7	4.91	57.6	1.070	67.9	5.500	67.3
rd73	0.01790	101.7	7.82	58.6	1.970	67.2	12.700	34.6
rd84	0.04980	48.4	7.46	69.6	4.657	36.1	48.600	30.9
seq	0.96280	98.2	10.33	214.4	40.828	77.9	33606.600	10.4
sqrt8	0.02130	77.0	7.42	59.2	2.540	47.8	7.800	39.7
squar5	0.02100	79.5	5.53	55.2	2.163	52.2	7.500	94.7
table3	0.38980	150.1	46.21	29.0	14.958	102.7	1733.200	11.1
xor5	0.00560	75.0	3.39	61.1	0.353	112.0	3.600	33.3

Table 1: Results for LGSynth93 benchmarks (CMOS columns are results for conventional CMOS synthesis using SIS; PTM columns are results for pass-transistor logic synthesised using PTM expressed as percentages of CMOS)