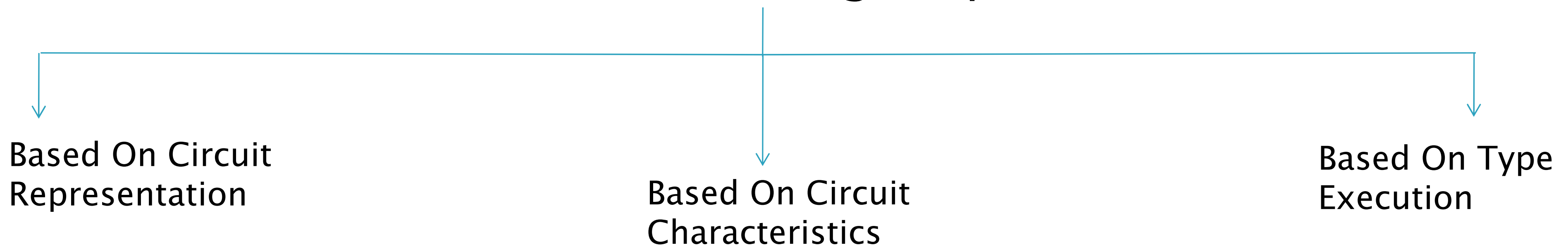
Multi-Level Minimization and Optimization.

- Logic optimization, a part of logic synthesis in electronics, is the process of finding an equivalent representation of the specified logic circuit under one or more specified constraints. Generally the circuit is constrained to minimum chip area meeting a prespecified delay.



- Based on circuit representation
  - a)Two-level logic optimization
  - b) Multi-level logic optimization
- Based on circuit characteristics
  - a)Sequential logic optimization
  - b)Combinational logic optimization
- Based on type of execution
  - a)Graphical optimization methods
  - b)Tabular optimization methods
  - c)Algebraic optimization methods

- Multi-level logic Means more than two-levels of logic representations and corresponds to multi-level logic circuits.
- If the specification is given in sum-of-products forms, the first step is to make it a more compact representation by introducing more logic levels in the representation. This is typically done by "factoring a logic expression" and "dividing a logic expression with another logic expression". For example, aef + bef + adf + bdf + ef + aeg + beg + adg + bdg + eg is an optimum twolevel logic representation. There are 26 literals appearing in the expression. It can be factored to ((a + b)(e + d) + e) (J + g), which needs only 7 literals. This factored logic expression corresponds to the multi-level circuit shown in Figure 2.1. It has 4 levels of logic and is much more compact than the two-level representation.

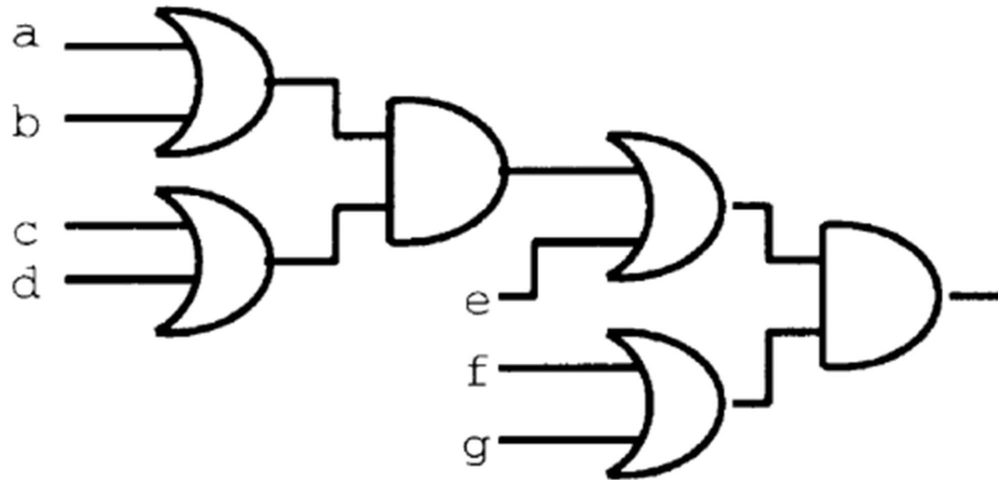


Figure 2.1. Multi-level logic circuit for aef + bef + adf + bdf + ef + acg + beg + adg + bdg + eg

- Multi-level logic optimization has a long history and early work can be found in the 60's. The simplest way to generate multi-level logic from twolevel logic representation is manual factoring. One can easily factor the expression above by hand. However, it is not easy to completely factor ab + ac + be. We can easily get the relation: ab+ae+be = a(b+e)+be = ab+(a+b)e. Further, these can be factored to ab + ae+ be = (a + b) (b+ e). In order to introduce the last expression, we need to use the relation bb = O. That is, we need to pay attention to the logic functions instead of just their representations. The optimization methods that consider logic functions as well as their representations are called "Boolean methods" , whereas the ones that consider only logic representations are called "Algebraic methods" . In other words, algebraic methods treat logic functions as polynomial expressions, whereas Boolean methods recognize Boolean algebra rules that can transform one logic representation to another. Generally speaking Boolean methods are much more powerful than algebraic methods but can be computationally expensive.

Mapping Algorithms

- Mapping Algorithm also known as Port Mapping.
- In this topic we are going to discuss how to make port mapping of a Sequential Circuit.
- As we know, In Sequential circuit there is clock pulse, a input port, output port and inout port.
- Let's take a exampal mapping algorithms for better understanding

```
library ieee;
use ieee. std_logic_1164.all;
use ieee. std_logic_arith.all;
use ieee. std_logic_unsigned.all;

entity SR_FF is
PORT( S,R,CLOCK: in std_logic;
Q, QBAR: out std_logic);
end SR_FF;

Architecture behavioral of SR_FF is
begin
PROCESS(CLOCK)
variable tmp: std_logic;
begin
if(CLOCK='1' and CLOCK'EVENT) then
if(S='0' and R='0')then
tmp:=tmp;
elsif(S='1' and R='1')then
tmp:='Z';
elsif(S='0' and R='1')then
tmp:='0';
else
tmp:='1';
end if;
end if;
Q <= tmp;
QBAR <= not tmp;
end PROCESS;
end behavioral;
```

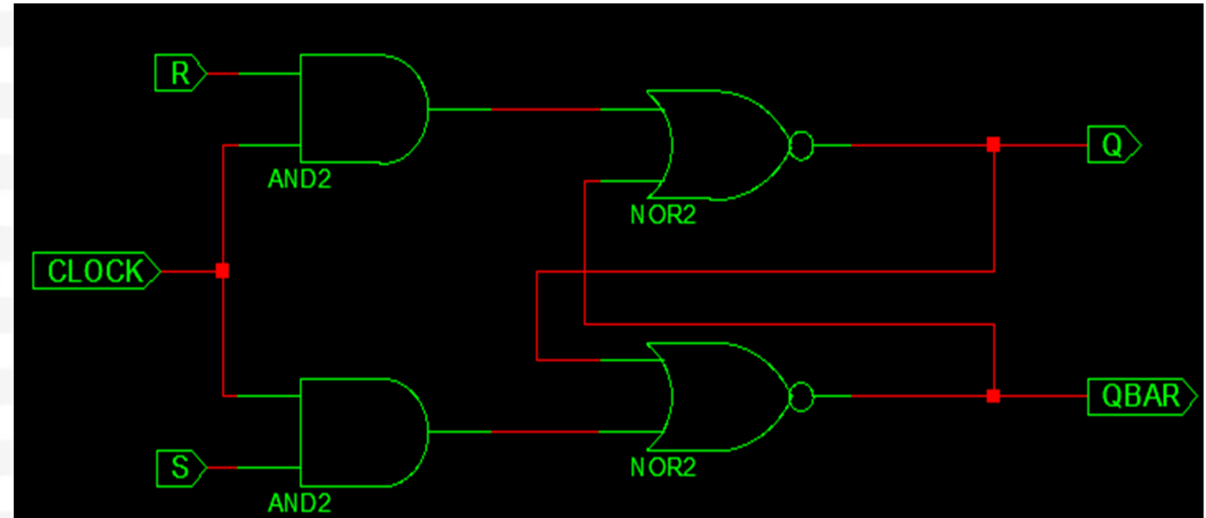


Fig. Logic diagram of SR Flip Flop

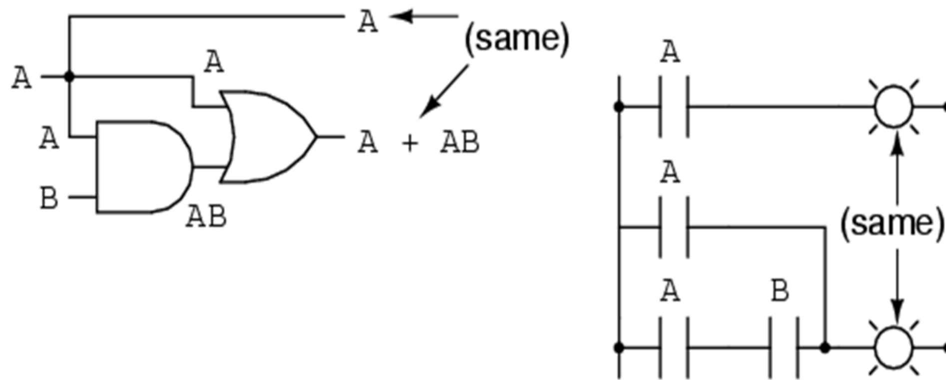
This is Mapping Algorithm of SR Flip Flop

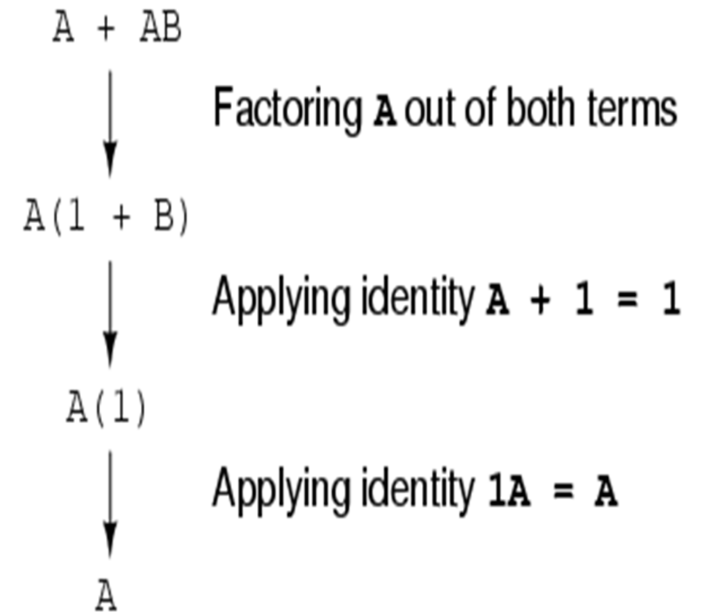
Factoring

- Boolean algebra finds its most practical use in the simplification of logic circuits.
- If we translate a logic circuit's function into symbolic (Boolean) form, and apply certain algebraic rules to the resulting equation to reduce the number of terms and/or arithmetic operations, the simplified equation may be translated back into circuit form for a logic circuit performing the same function with fewer components.

- Let's take a example of Factoring Method

A + AB = A





➢This rule may be proven symbolically by factoring an "A" out of the two terms, then applying the rules of A + 1 = 1 and 1A = A to achieve the final result:

Decomposition

- It's method of solving Boolean expression. In this method we are decompose the given expression after that simplify this
- let's take a example

Q.) $A'.B'+A'.B+A.B$

Sol.) $= A'.B'+A'.B+A.B$

$= A'.(B'+B)+A.B$

$= A'+A.B \quad \because (B'+B=1)$

$= A'+A'.A+A'.A+A.B \quad \because$(Decomposition of expression)

$= A'+ A'.A+A(A'+B)= (A'+A).(A'+ A'+B)= (A'+A).(A'+B)$

$= A'+B \quad \because(A'+A=1)$

Binary Decision Diagram(BDD)

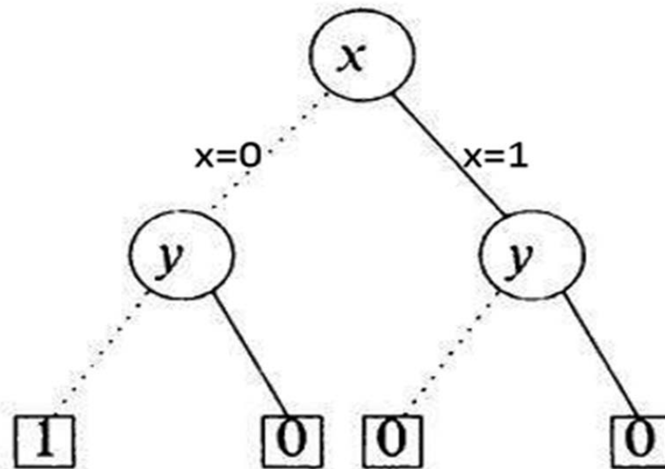
Introduction:-

- Boolean functions are mostly represented by truth tables and propositional formulas

$$f(x, y) \stackrel{\text{def}}{=} x \cdot (y + \overline{x}).$$

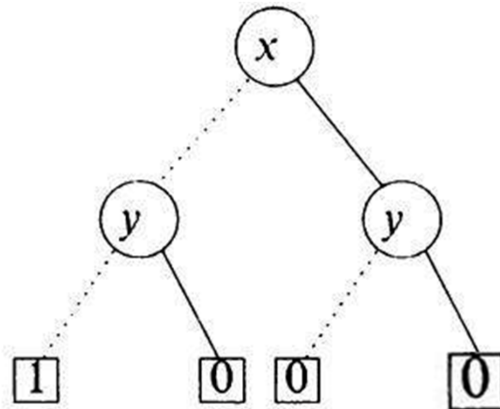
- Representation with truth table:
- Space inefficient. 100 variables results in 2100 lines.
- Checking satisfiability and equivalence is also inefficient.

- Another way of representing Boolean functions.
  - BDDs are trees with terminal nodes labeled with either 0 or 1 and non terminal nodes labeled with Boolean variables x, y, z…
  - Each non-terminal node has two edges
  - One dashed line and one solid line



Solid line: Corresponds when variable value = 1
Dashed line: Corresponds when variable value = 0

$f(x, y)=?$



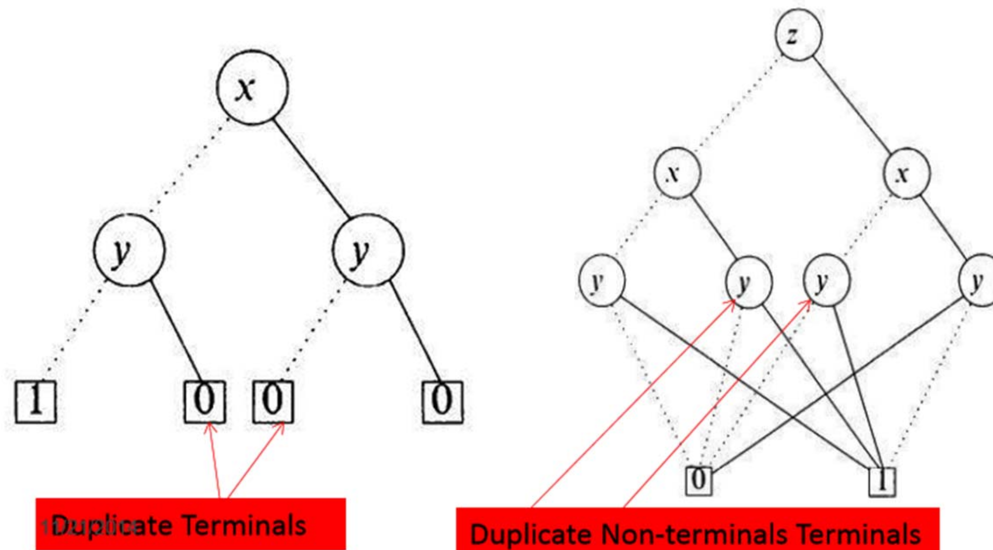
$f(x, y)= \overline{(x+y)}$

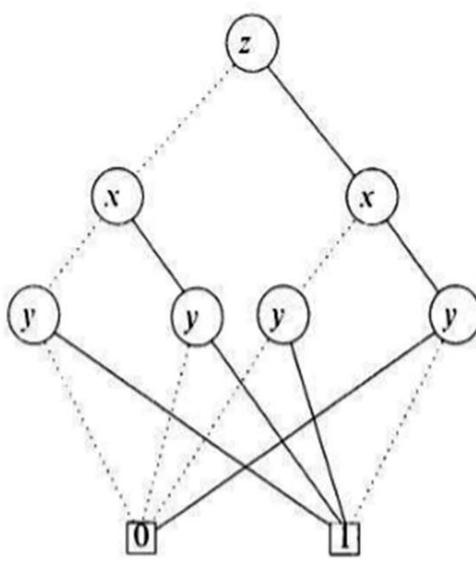
x	y	f(x,y)
0	0	1
0	1	0
1	0	0
1	1	0

- By observing this Binary Decision Diagram we can create the truth table of given logic circuit.
- Cons: f depending on n variables will have at least 2n+1 -1 nodes. This shows that this representation is not compact. But, we can exploit its redundancy in order to improve it.

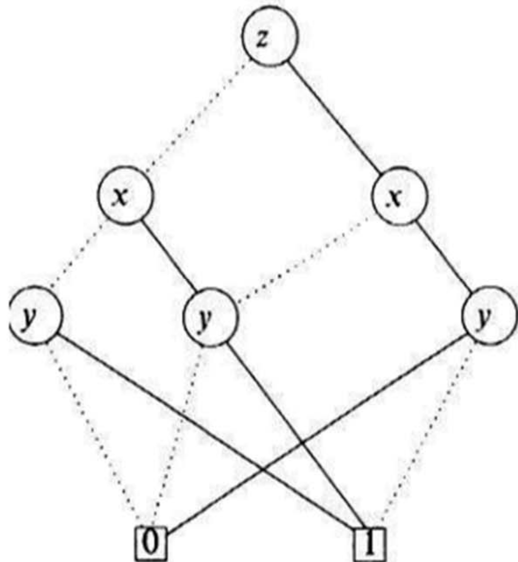
BDD Contraction

- [C1]: Removal of duplicate terminals
- [C2]: Removal of redundant tests
- [C3]: Removal of duplicate non-terminals

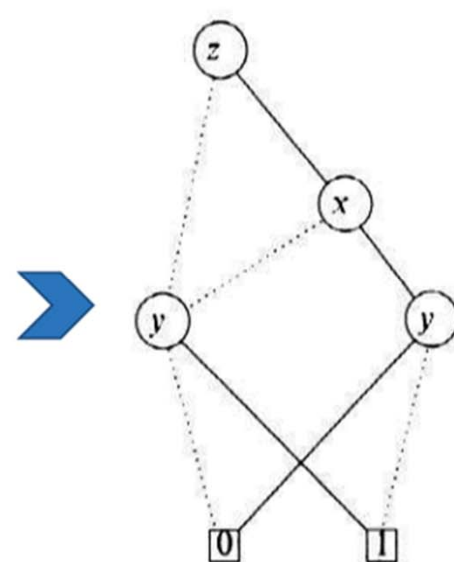




(a) Duplicated BDDs



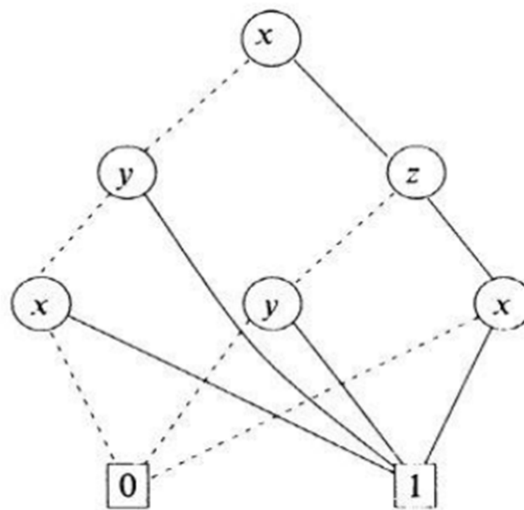
(b) Removal of duplicate *y*



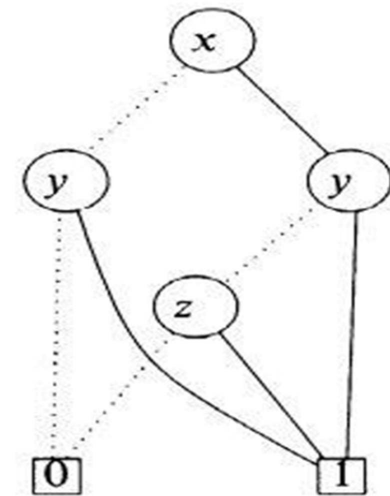
(c) Removal of redundant decision point *x*

BDD Cons

- BDDs with multiple occurrences of a Boolean variable along a path make it inefficient.
- Checking equivalence of two BDDs is still difficult. See figure (a) and (b)



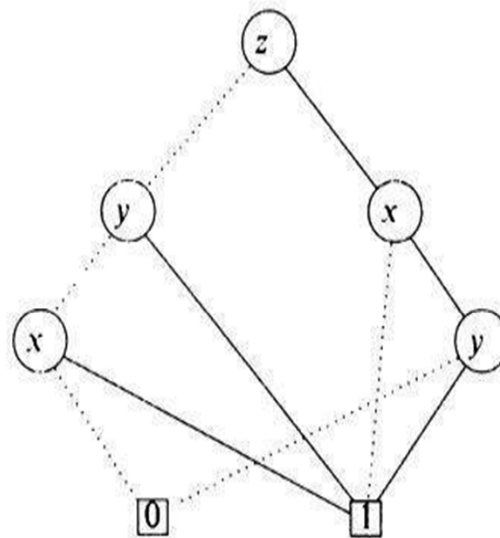
(a)



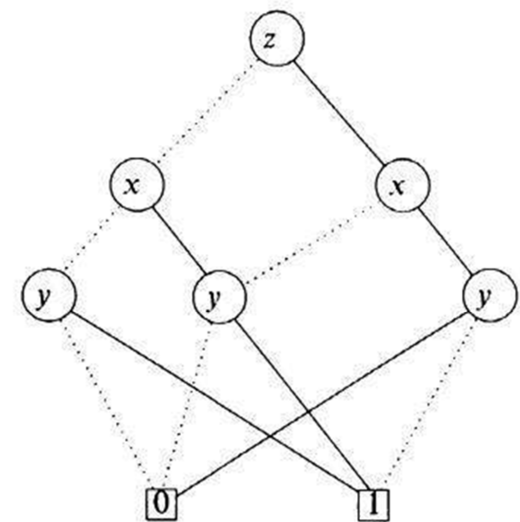
(b)

Ordered BDDs

- Ordered BDDs ensure the variables appear in the same order along all paths from the root to the leaves.
- There are not multiple occurrences of any variable along a path.



Unordered BDD



Ordered BDD

- With OBDDs we cannot have more than one BDD representing the same function, i.e., one function -> one OBDD. ▪ Follows that checking equivalence with OBDDs is immediate.
- OBDDS for f = (x1+x2). (x3+x4). (x5+x6)

Ordering: [x 1, x2, x3, x4, x5, x6]

Ordering: [x1, x3, x5, x2, x4, x6]

Thanks You