C BASED VLSI ANSWERS

1. **What are the steps involved in HLS?**

Ans:

• Preprocessing: Intermediate representation (CDFG) construction, data-dependency, live variable analysis, compiler optimization.

• Scheduling: Assigns control step to the operations of the input behaviour.

• Allocation: Computes minimum number of functional units and registers.

• Binding: Variables are mapped to registers, operation to functional units, data transfers to the interconnection units.

• Data path & Controller design: controller is designed based on inter connections among the data path elements, data transfer required in different control steps.

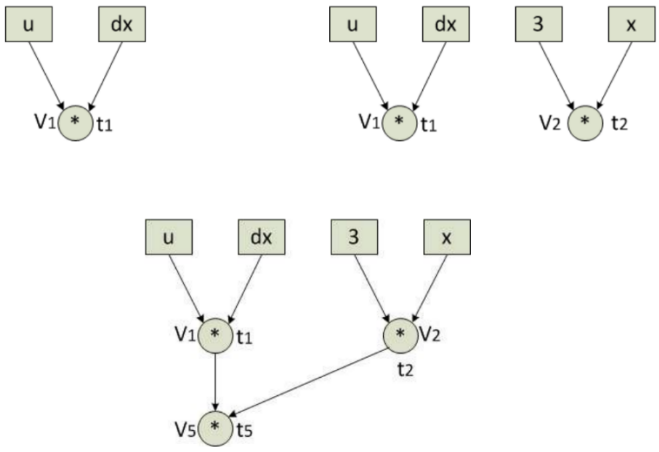
2. **For the 2nd order differential equation solver given below –**

Diffeq: (x, dx, u, a, clock, y)
input: x, dx, u, a, clock;
output: y
while(x < a)
u1 = u-(3*x*u*dx)-(3*y*dx);
y1 = y+(u*dx);
x1 = x+dx;
x = x1, y = y1, u = u1;
end, Perform Preprocessing and obtain the Data Dependency Graph

Ans:

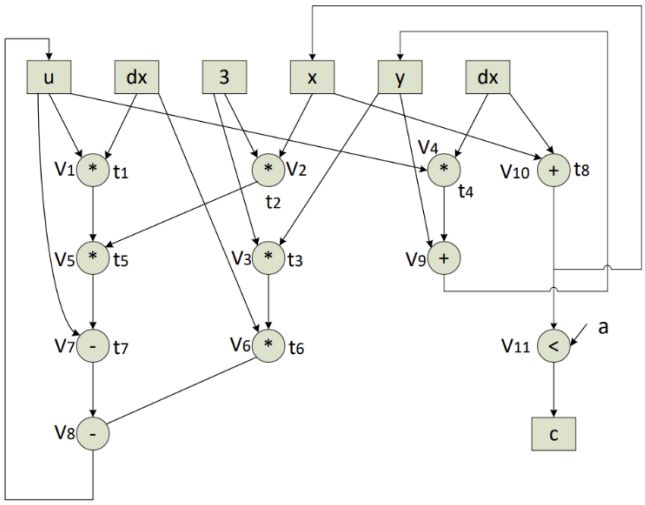
Preprocessing

B_1
$V_1 : t_1 = u * dx$
$V_2 : t_2 = 3 * x$
$V_3 : t_3 = 3 * y$
$V_4 : t_4 = u * dx$
$V_5 : t_5 = t_1 * t_2$
$V_6 : t_6 = t_3 * dx$
$V_7 : t_7 = u - t_5$
$V_8 : u = t_7 - t_6$
$V_9 : y = y + t_4$
$V_{10} : x = x + dx$
$V_{11} : c = x < a$



Preprocessing

B_1
$V_1 : t_1 = u * dx$
$V_2 : t_2 = 3 * x$
$V_3 : t_3 = 3 * y$
$V_4 : t_4 = u * dx$
$V_5 : t_5 = t_1 * t_2$
$V_6 : t_6 = t_3 * dx$
$V_7 : t_7 = u - t_5$
$V_8 : u = t_7 - t_6$
$V_9 : y = y + t_4$
$V_{10} : x = x + dx$
$V_{11} : c = x < a$

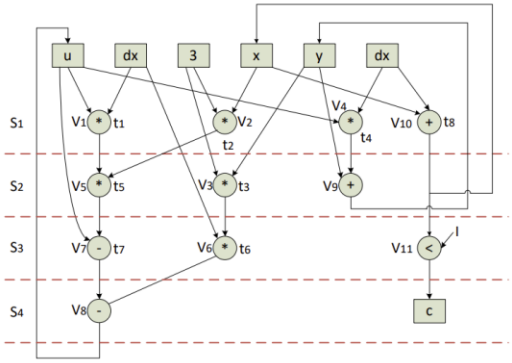


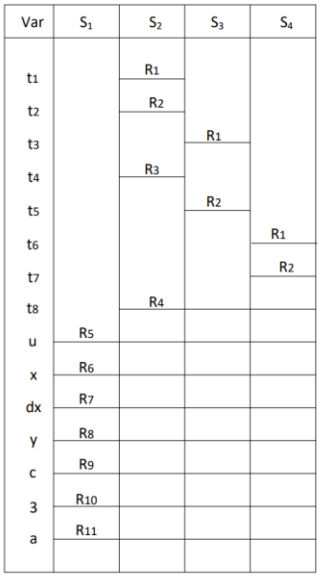
Date dependency graph

3. For Question No 2 perform Register allocation and Binding

Ans:

Register Allocation and Binding





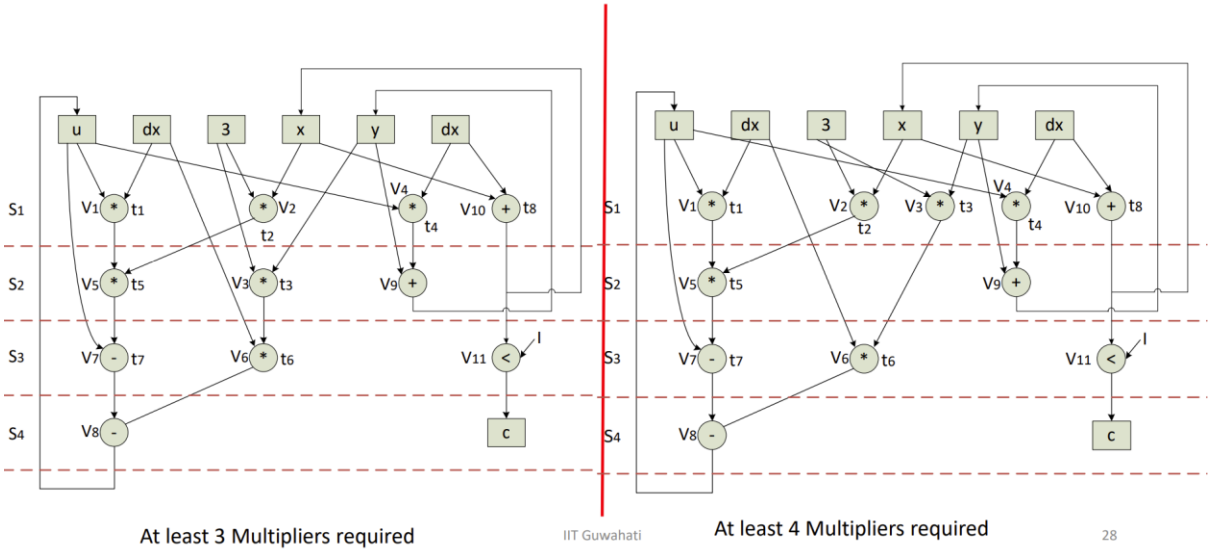
R_1: t_1, t_3, t_6
R_2: t_2, t_5, t_7
R_3: t_4
R_4: t_8
R_5: u
R_6: x
R_7: dx
R_8: y
R_9: c
R_{10}: 3
R_{11}: a

Interval graph

4. What are the scheduling possibilities for Question No 2?

Ans:

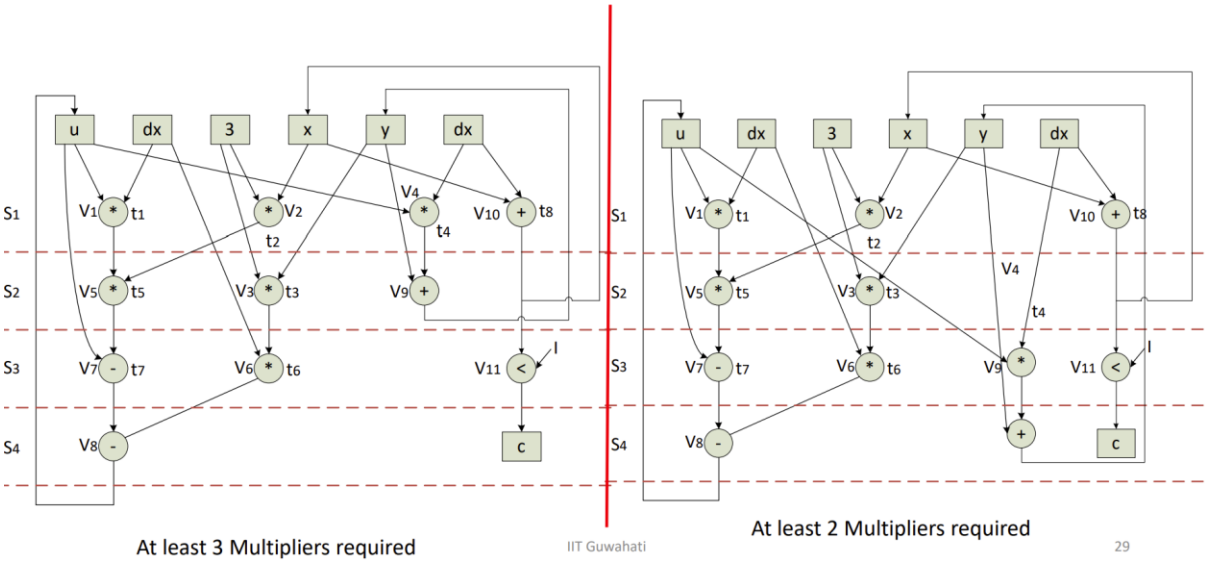
Scheduling Possibilities



At least 3 Multipliers required

At least 4 Multipliers required

Scheduling Possibilities



At least 3 Multipliers required

At least 2 Multipliers required

5. List the problems related to Scheduling.

Ans:

Scheduling Problems

- Minimum Latency Unconstrained minimum-latency scheduling problem (Unconstraint)
- Minimum latency under resource constraints (MLRC)
- Minimum resource under latency constraints (MRLC)

6. **Brief on –**
    1. **Loop fusion: Combine multiple loops that iterate over the same range into a single loop.**

Example (C):

```c
for (i=0; i<N; i++) a[i] = b[i] + 1;
for (i=0; i<N; i++) c[i] = a[i] * 2;
```

⬇️

```c
for (i=0; i<N; i++) {
    a[i] = b[i] + 1;
    c[i] = a[i] * 2;
}
```

    2. **Loop inversion: Move the loop condition from entry to exit (convert while to do–while).**

Example:

```c
while (i < N) { body; i++; }
```

⬇️

```c
if (i < N)
do { body; i++; } while (i < N);
```

    3. **Loop Interchange: Swap inner and outer loops.**

Example:

```c
for (i=0;i<N;i++)
  for (j=0;j<M;j++)
    A[i][j]++;
```

⬇️

```c
for (j=0;j<M;j++)
  for (i=0;i<N;i++)
    A[i][j]++;
```

    4. **Loop-Invariant Code Motion (LICM): Move computations that don't change across loop iterations outside the loop.**

```c
for (i=0;i<N;i++)
    y[i] = x[i] * c;
```

⬇

```c
temp = c;
for (i=0;i<N;i++)
    y[i] = x[i] * temp;
```

5. **Loop Nest Optimization: Optimizing multiple nested loops together. Includes:**

Interchange

Fusion

Tiling

Unrolling

6. **Loop Unrolling: Replicate loop body multiple times.**

Example (unroll ×2):

```c
for (i=0;i<N;i+=2) {
    a[i]   = b[i] + 1;
    a[i+1] = b[i+1] + 1;
}
```

7. Loop Splitting (Loop Fission): Split one loop into multiple loops.

Example:

```c
for (i=0;i<N;i++) {
    a[i] = b[i] + 1;
    c[i] = d[i] * 2;
}
```

⬇

```c
for (i=0;i<N;i++) a[i] = b[i] + 1;
for (i=0;i<N;i++) c[i] = d[i] * 2;
```

8. Loop Unswitching: Move loop-invariant conditions outside the loop.

```c
for (i=0;i<N;i++) {
    if (flag) a[i]++;
    else a[i]--;
}
```

```c
if (flag)
  for (i=0;i<N;i++) a[i]++;
else
  for (i=0;i<N;i++) a[i]--;
```

9. Software Pipelining: Overlap execution of different loop iterations.

10. Loop Tiling (Blocking): Break loops into smaller blocks.

```c
for (ii=0; ii<N; ii+=T)
  for (i=ii; i<ii+T; i++)
    ...
```

7. **Brief on (i) Constant Propagation & (ii) Variable Propagation.**

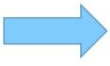
Ans:

Constant Propagation or Constant Folding

- Constant propagation consists of detecting constant operands and pre-computing the value of the operation with that operand.
- Since the result may be again a constant, the new constant can be propagated to those operations that use it as input.

```
int x = 14;
int y = 7 - x / 2;
return y * (28 / x + 2);
```

```
int x = 14;
int y = 7 - 14 / 2;
return y * (28 / 14 + 2);
```

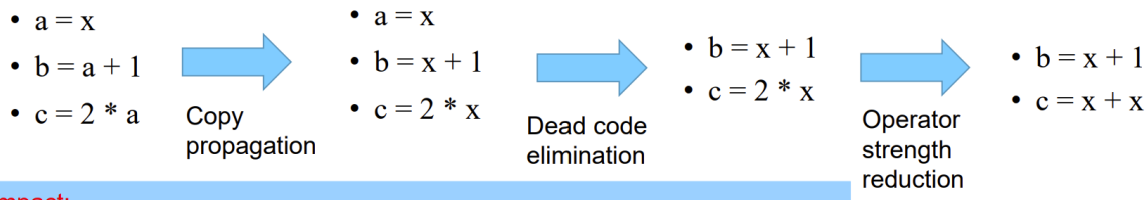


```
int x = 14;
int y = 0;
return 0;
```

Impact:
Reduces number of operations to be executed
- computation time
- hardware resources

Variable Propagation or Copy Propagation

- Variable propagation consists of detecting the *copies* of variables, i.e., the assignments like **x** = ***y,*** and using the right-hand side in the following references in place of the left-hand side.
- Data-flow analysis permits the identification of the statements where the transformation can be done.
- The propagation of ***a*** cannot be done after a different reassignment to **x**



Impact:
- Reduces number of variables and hence number of registers in hardware.
- May enable other optimizations

8. Brief on Tree Height Reduction.

Ans:

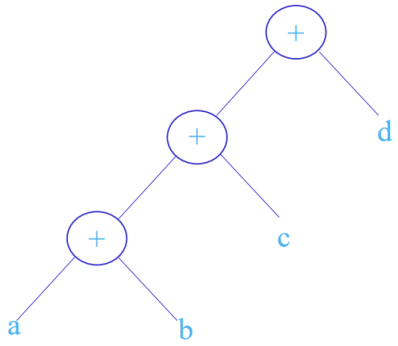
Tree Height Reduction

- This transformation applies to the compound arithmetic statement.
- The expression splits into two-operand expressions, so that the parallelism available in hardware can be exploited at best.

- x = a + b + c + d

x = a + b
x = x + c
x = x + d

- Three Addition in series
- Need one adder
- Three cycles needed

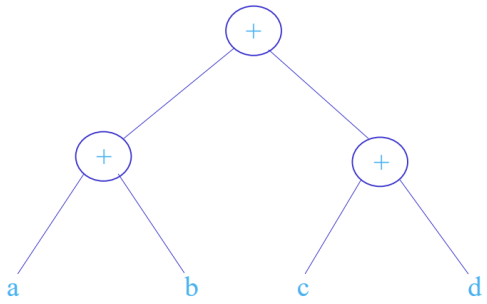


Tree Height Reduction (cont'd)

- Two Additions can be done in parallel.
- Need two adders and two cycles.

Goals:

- The goal is balancing the expression tree as much as possible.
- Tree-height reduction exploits some properties of the arithmetic operations.



Impact: The height is proportional to a lower bound on the overall computation time

9. List the unsupported C constructs in Hardware Efficient C Coding

Ans:

Unsupported C Constructs

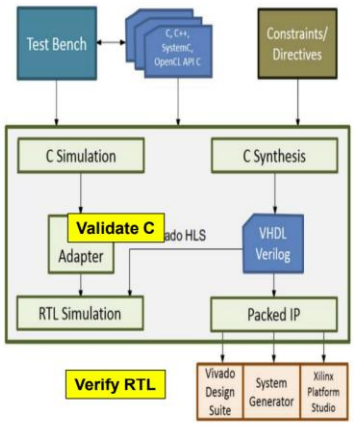
- System calls
- Dynamic Memory allocation
- Recursion
- Standard template library

10. Brief on the two important steps involved in verifying the design

Ans:

C Validation and RTL Verification

- There are two steps to verifying the design
  - Pre-synthesis: C ***Validation***
    - Validate the algorithm is correct
  - Post-synthesis: RTL ***Verification***
    - Verify the RTL is correct
- C validation
  - A **HUGE** reason users want to use HLS
    - Fast, free verification
  - Validate the algorithm is correct **before** synthesis
- RTL Verification
  - Check if the generated RTL is correct
  - Vivado HLS can co-simulate the RTL with the original test bench



11. You have only one Dual port RAM of 2K size 32bits width, execute the below code using one dual port RAM of 2K size 32bits width.

```
int A1[1000] , A2[500] ;
for ( i = 1 to 1000 ) {
 A1[i] = value1;
 if (i<500)
A2[i]=value2;}
```

Ans:

Solution

```
int A1[1000] , A2[500] ;
for ( i = 1 to 1000 )
{
    A1[i] = value1;
    if (i<500)
        A2[i] = value2;
}
```

```
int A[1500] ;
for ( i = 1 to 1000 )
{
    A[i] = value1;
    if (i<500)
      A[i+1000] = value2;
}
```

Here two writes possible due to availability of dual port.

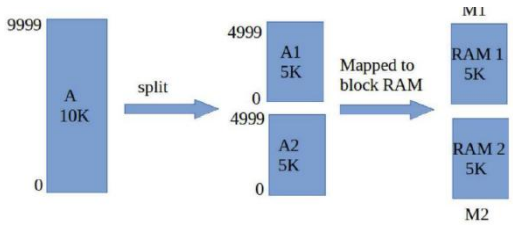
12. Brief on Array Partitioning.

Ans:

- Array to be used will be larger than standard RAM size or it has many access.
- it can be partitioned into multiple smaller arrays if it has nonoverlapping accesses.

Array partitioning has the following advantages:

- Results in RTL with multiple small memories or multiple registers instead of one large memory.
- Effectively increases the amount of read and write ports for storage.
- Potentially improves the throughput of the design



- In Vivado, the general syntax for array partitioning is as follows:
  - #pragma HLS array_partition variable=<name> <type> factor=<int> dim=<int>
- **variable=<name>:** the array variable to be partitioned.
- **<type>**
- Optionally specifies the partition type. The default type is complete.

- **cyclic**

• Cyclic partitioning creates smaller arrays by interleaving elements from the original array. The array is partitioned cyclically by putting one element into each new array before coming back to the first array to repeat the cycle until the array is fully partitioned. For example, if factor=3 is used:

• Element 0 is assigned to the first new array

• Element 1 is assigned to the second new array.

• Element 2 is assigned to the third new array.

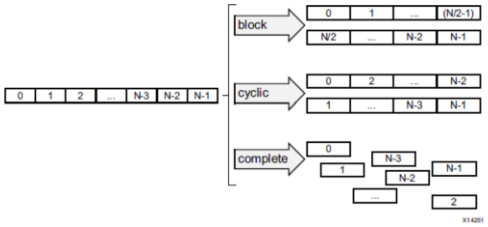
• Element 3 is assigned to the first new array again.

• **block**

• Block partitioning creates smaller arrays from consecutive blocks of the original array. This effectively splits the array into N equal blocks, where N is the integer defined by the factor= argument

• **complete**

• Complete partitioning decomposes the array into individual elements. For a one-dimensional array, this corresponds to resolving a memory into individual registers. This is the default <type>.



- **factor=<int>**
  - Specifies the number of smaller arrays that are to be created. For complete type partitioning, the factor is not specified. For block and cyclic partitioning, the factor is required.
- **dim=<int>**
  - Specifies which dimension of a multi-dimensional array to partition. Specified as an integer from 0 to <N>, for an array with <N> dimensions:
    - If a value of 0 is used, all dimensions of a multi-dimensional array are partitioned with the specified type and factor options.
    - Any non-zero value partitions only the specified dimension. For example, if a value 1 is used, only the first dimension is partitioned.

13. Brief on Force-Directed Scheduling with an example. Also, mention the two types of Force.

Ans:

Force-Directed Scheduling is a resource-constrained HLS scheduling technique that minimizes hardware usage by distributing operations across time steps using force calculations.

Heuristic scheduling methods [by Paulin and Knight]:

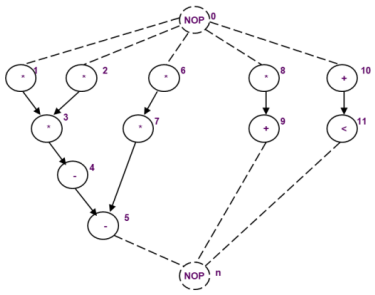
• Min latency subject to resource bound

Force-directed scheduling for MLRC

• Min resource subject to latency bound

Force-directed scheduling for MRLC

Example



- *v1* has zero mobility. p1(l) = 1, p1(2) = p1(3) = pl(4) = 0.
- p2(1) = 1, p2(2) = p2(3) = p2(4) = 0
- *v6* time frame is [1, 2]. p6(l) = p6(2) = 0.5 and p6(3) = p6(4) = 0.
- Operation *v8* timeframe is [1, 3]. Hence p8(l) = p8(2) = p8(3) = 0.3 and p8(4) = 0.

- Distribution graphs for multiplier and ALU

Force-directed scheduling

- Force is as *priority* function in during operation selection
- Force is related to concurrency:
  - The larger the force, the larger the concurrency
- Two types of force
  - Self force (relating operation to a time step)
  - Predecessor/successor force (related to dependencies)

14. Why is Logic Locking/Obfuscation needed? Brief on Logic Obfuscation.

Ans: Hardware Trojans in IC design flow

Malicious modifications to designs • Objective: Control, modify, disable, or monitor content of a design • Scenarios: • Malicious 3PIP vendor • Malicious foundry • Real-life example: Syrian radars were turned off by hardware trojans

Counterfeiting

Forgery or imitation of original components • Objective: Commercial benefit and subvert mission-critical systems • Scenarios: • Malicious test facility • Malicious assembly unit • Malicious sellers • Real-life example: Counterfeit components in US military systems

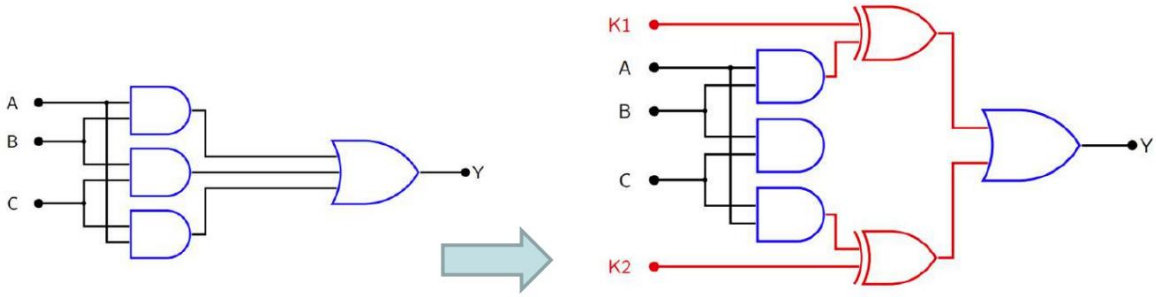
Reverse Engineering

Identifying device technology, functionality, design • Objective: Enables Trojan insertion and piracy • Scenarios: • Malicious SoC integration house • Malicious foundry • Malicious user • Real-life example: Chipworks reverse engineers ICs to detect piracy

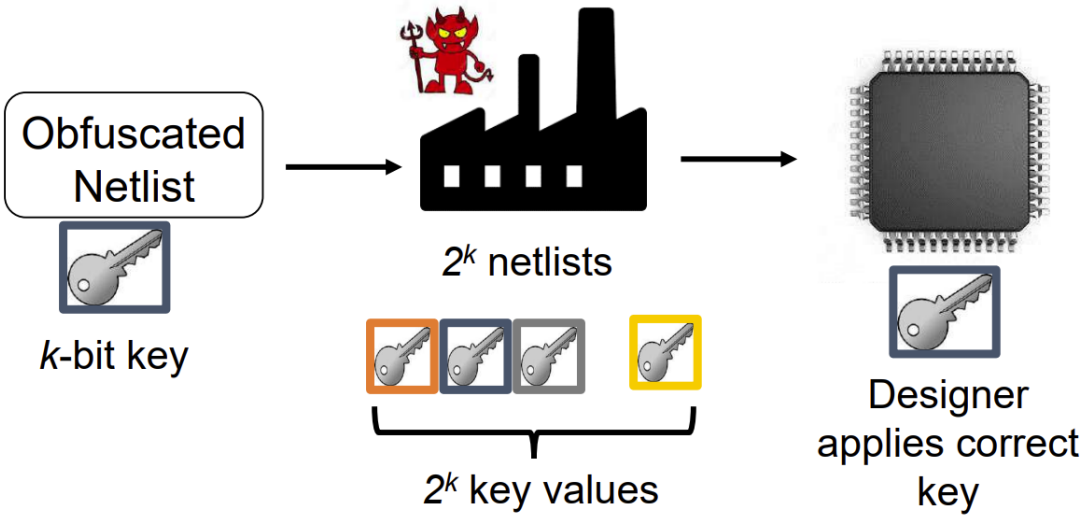
IC/IP Piracy and Overbuilding

Steal & claim ownership of an IC and/or illegal use • Scenarios: • Malicious SoC integration house • Malicious foundry • Real-life examples: • $4,000,000,000 loss per year to IC industry • ARM detected IP piracy in 2000

Solution Logic Locking/Obfuscation



Logic obfuscation



Logic Obfuscation is a hardware security technique where **additional logic and secret keys** are inserted into a design so that:

- The circuit works **correctly only when the right key is applied**
- With an incorrect key, the output is **wrong, unpredictable, or degraded**

Basic Idea

- Insert **key-controlled gates** (XOR/XNOR, MUXes, LUTs)
- Correct key → normal operation
- Wrong key → incorrect outputs

15. f = abc + abd + bcd, Find all the kernels for the function f.

Ans:

Find all Kernels

- f = abc + abd + bcd
- Find all kernels

Divider cube d	F = d.Q + k	Is Q a kernel of F
1	(abc + abd + bcd) + 0	No, here cube b as factor.
a	a(bc + bd) + bcd	No. b is a factor
b	b (ac + ad + cd) + 0	Yes, kernel (ac + ad + cd) is cube-free
ab	ab (c + d) + bcd	Yes, kernel (c + d) is cube-free

16. List the various methods of logic transformations in MultiLevel Optimization

Ans:

Multi-level Logic Optimizations

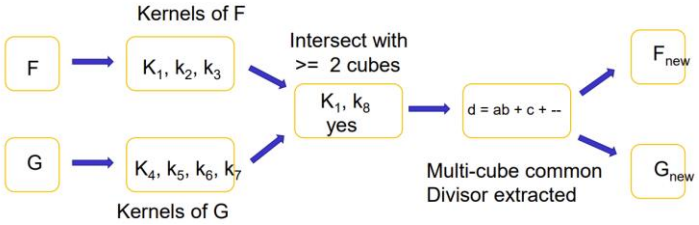
- There are various methods of logic transformations. The methods are as follows:
  1. Factoring
  2. Decomposition
  3. Extraction
  4. Substitution
  5. Elimination.

17. List the steps involved in Algebraic Method (MultiLevel Optimization)

Ans:

The Algebraic Method

1. Find kernels of F and G
2. Find kernels in intersections of K(F) and K(G)
3. Extract multi-cube common divisor D
4. Rewrite F and G using D



Example

- F = ae + be + cde + ab
- G = ad + ae + be + bc

K(F)	Co-kernal	K(G)	Co-kernal
a + b + cd	e	a + b	e
b + c	a	d + e	a
a + e	b	d + e + c	b
ac + be + cde + ab	1	ab + ac + be + bc	1

(a + b + cd) ∩ (a + b) = (a + b)
So, this is workable multi-cube divisor of F and G.

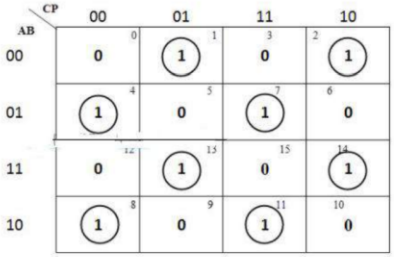
18. What is the need for Multilevel Logic Optimization

Ans:

Need for **Multi-level Logic Optimization**

- In some cases SoP representation is impractical
  - Example: Parity bit generator, multiplication

4-bit received message				Parity error check C_p
A	B	C	P	
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0



nential number of implicants in SoP form for parity bit generator

Need for **Multi-level Logic Optimization**

- **Advantage:**
  - Multi-level logic circuits require less area and delay compared to the corresponding two-level realizations and hence are more practical.
  - More than one functions can be optimized together.
- **Disadvantage:**
  - It is difficult to obtain provably optimal multi-level realizations because of the much larger design space available for exploration.

Phases Of Multi-level Logic Synthesis

- Technology Independent Synthesis
- Technology Dependent Synthesis

Objectives

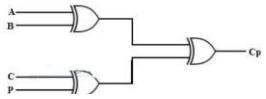
- Reduces the number of literal in expression
  - No of transistors is twice of no of literals
- Identifying common sub-expression within an Boolean Expression (for single output) or between multiple Boolean expressions (for multiple outputs)

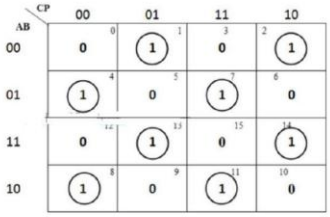
$$PEC = \bar{A}\,\bar{B}\,(\bar{C}\,D + C\,\bar{D}) + \bar{A}\,B\,(\bar{C}\,\bar{D} + C\,D) + A\,B\,(\bar{C}\,D + C\,\bar{D}) + A\,\bar{B}\,(\bar{C}\,\bar{D} + C\,D)$$

$$= \bar{A}\,\bar{B}\,(C \oplus D) + \bar{A}\,B\,(\overline{C \oplus D}) + A\,B\,(C \oplus D) + A\,\bar{B}\,(\overline{C \oplus D})$$

$$= (\bar{A}\,\bar{B} + A\,B)\,(C \oplus D) + (\bar{A}\,B + A\,\bar{B})\,(\overline{C \oplus D})$$

$$= (A \oplus B) \oplus (C \oplus D)$$





19. What are the different forms to represent a Boolean function?

Ans:

Representations of Boolean Functions

Forms to represent Boolean Functions

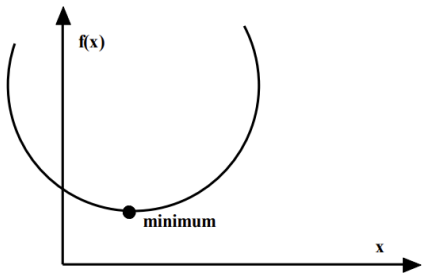
- Truth table
- List of cubes: Sum of Products, Disjunctive Normal Form (DNF)
- List of conjuncts: Product of Sums, Conjunctive Normal Form (CNF)
- Binary Decision Tree, Binary Decision Diagram
- Boolean formula
- Boolean network
- Cube form

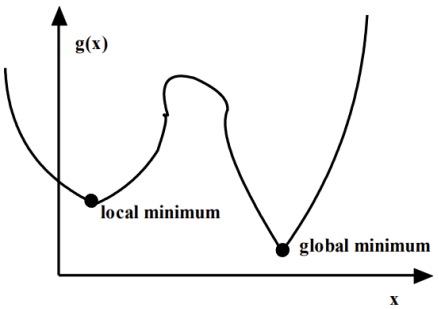
20. Brief on the Methods involved in Two-Level Logic Optimizations.

Ans:

Two-level Logic Optimizations: Methods

- Find prime implicants and try to cover them using minimal number of covers
  - Karnaugh Map based method
  - Quine–McCluskey method
  - Heuristic based approach
    - ESPRESSO
- ESPRESSO is based on simulated annealing method





ESPRESSO

- Start from initial cover
  - May be the minterm itself
- Modify cover
  - Make it prime and irredundant,
  - Reduce, expand, irredundant operations
- Stop when no further improvement is possible or timed out