ECE 270 Exam 2 Study Guide

July 30, 2025

1 SoP, PoS, K-Maps, and Timing Hazards (30%)

1.1 Sum of Products (SoP) and Product of Sums (PoS) Representations

- **Definition of Combinational Logic Circuit and Logic Function** :
  - A **combinational logic circuit** is defined as a circuit whose output depends solely on its current combination of input values.
  - A **logic function** is the assignment of a '0' or '1' to each possible combination of its input variables.
  - **Fundamental Terms** :
    - **Literal** : A variable or its complement.
      - *Example* : W, X, Y'.
    - **Product Term** : A single literal or a logical product (AND) of two or more literals.
      - *Example* : W · X' · Z.
    - **Sum Term** : A single literal or a logical sum (OR) of two or more literals.
      - *Example* : X + Y + Z'.
  - **Sum-of-Products (SoP) Expression** :
    - A logical sum (OR) of product terms.
    - This form is also referred to as AND-OR logic.
    - *Example* : X · Y' + W · Z.
  - **Product-of-Sums (PoS) Expression** :
    - A logical product (AND) of sum terms.
    - This form is also referred to as OR-AND logic.
    - *Example* : (X + Y) · (W + Z').
  - **Canonical Forms: Minterms and Maxterms** :
    - **Normal Term** : A product or sum term in which no variable appears more than once.
    - **n-variable Minterm** : A normal product term with 'n' literals.
      - Each minterm corresponds to a unique input combination (row in a truth table) where the function produces a '1' output.
      - For a minterm m_i, if the j-th bit of index i is 1, the j-th variable is uncomplemented; if it's 0, the variable is complemented.
      - *Example* : For X, Y, Z, m_3 (011) is $X' \cdot Y \cdot Z$.
    - **n-variable Maxterm** : A normal sum term with 'n' literals.
      - Each maxterm corresponds to a unique input combination (row in a truth table) where the function produces a '0' output.
      - For a maxterm M_i, if the j-th bit of index i is 0, the j-th variable is uncomplemented; if it's 1, the variable is complemented.
      - *Example* : For X, Y, Z, M_3 (011) is $X + Y' + Z'$.
  - **Canonical Sum (Sum of Minterms)** :
    - A logic function can be expressed as a sum of minterms that correspond to input combinations producing a '1' output.

    - **Notation** : $\sum_{X,Y,Z}(i,j,...)$ where $i,j,...$ are the decimal indices of the minterms (or rows in the truth table) for which the function is '1'.
- **Canonical Product (Product of Maxterms)** :
    - A logic function can be expressed as a product of maxterms that correspond to input combinations producing a '0' output.
    - **Notation** : $\prod_{X,Y,Z}(i,j,...)$ where $i,j,...$ are the decimal indices of the maxterms (or rows in the truth table) for which the function is '0'.
- **Shorthand Notations: ON Set and OFF Set** :
    - **ON set** : The minterm list that "turns on" an output function, representing all input combinations for which the function evaluates to '1'. It is equivalent to the canonical sum and uses the $\sum$ notation.
    - **OFF set** : The maxterm list that "turns off" an output function, representing all input combinations for which the function evaluates to '0'. It is equivalent to the canonical product and uses the $\prod$ notation.
    - The ON set and OFF set of a function are complementary; they cover all possible input combinations without overlap.
- **Conversion Between Representations (Truth Table to Expression)** :
    - **Truth Table to SoP / Canonical Sum / ON set** : For each row in the truth table where the output $F = 1$, identify its corresponding minterm. The logical OR of these minterms forms the canonical sum (SoP). The decimal indices of these rows form the ON set.
    - **Truth Table to PoS / Canonical Product / OFF set** : For each row in the truth table where the output $F = 0$, identify its corresponding maxterm. The logical AND of these maxterms forms the canonical product (PoS). The decimal indices of these rows form the OFF set.

Exam Style Problem & Explanation:

- **Problem** : Given the truth table for function $F(X,Y,Z)$:
-

Row #	X	Y	Z	F(X,Y,Z)
0	0	0	0	1
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	0
6	1	1	0	1
7	1	1	1	1

- Express $F(X,Y,Z)$ as:
    1. An ON set.
    2. An OFF set.
    3. A sum of minterms (canonical sum).
    4. A product of maxterms (canonical product).
- **Solution & Explanation**:
    1. **ON set** : Identify all rows where $F(X,Y,Z) = 1$.
        - Rows: 0 (000), 3 (011), 6 (110), 7 (111).
        - **Answer** : $\Sigma_{X,Y,Z}(0,3,6,7)$.
    2. **OFF set** : Identify all rows where $F(X,Y,Z) = 0$.
        - Rows: 1 (001), 2 (010), 4 (100), 5 (101).
        - **Answer** : $\Pi_{X,Y,Z}(1,2,4,5)$.
    3. **Sum of minterms (canonical sum)** : For each row in the ON set, write its corresponding minterm and logically OR them.
        - $m_0 = X' \cdot Y' \cdot Z'$
        - $m_3 = X' \cdot Y \cdot Z$
        - $m_6 = X \cdot Y \cdot Z'$
        - $m_7 = X \cdot Y \cdot Z$
        - **Answer** : $F(X,Y,Z) = X' \cdot Y' \cdot Z' + X' \cdot Y \cdot Z + X \cdot Y \cdot Z' + X \cdot Y \cdot Z$.

4. **Product of maxterms (canonical product)** : For each row in the OFF set, write its corresponding maxterm and logically AND them.
   * $M_1 = (X + Y + Z')$
   * $M_2 = (X + Y' + Z)$
   * $M_4 = (X' + Y + Z)$
   * $M_5 = (X' + Y + Z')$
   * **Answer** : $F(X, Y, Z) = (X + Y + Z') \cdot (X + Y' + Z) \cdot (X' + Y + Z) \cdot (X' + Y + Z')$.

—

2 Section 1b

(b) Karnaugh Maps (K-Maps)

Karnaugh Maps (K-Maps) provide a graphical method for simplifying Boolean expressions and designing minimal-cost combinational logic circuits. They are crucial for both Sum-of-Products (SoP) and Product-of-Sums (PoS) minimisation [1d, 1e].

– **Mapping a Truth Table or Expression to a K-Map** :
  * **Purpose** : A K-map represents a truth table in a way that allows for easy visual identification of adjacent minterms or maxterms.
  * **Structure** : K-maps are arranged such that adjacent cells (including cells that 'wrap around' the edges) differ by only one variable. This adjacency represents the Boolean algebra identity $X + X' = 1$ or $(X)(X') = 0$, which is key to simplification.
  * **Variable Order** : Variables on the map's axes follow a Gray code sequence (e.g., 00, 01, 11, 10) to ensure single-bit changes between adjacent cells.
  * **Placing Values** :
    · For **SoP expressions** or **Canonical Sums (ON sets)** : Place a '1' in the K-map cell corresponding to each minterm where the function's output is '1' [1c, 254, 271].
    · For **PoS expressions** or **Canonical Products (OFF sets)** : Place a '0' in the K-map cell corresponding to each maxterm where the function's output is '0' [1c, 255, 272].
    · **Don't Cares (X)** : For incompletely specified functions where certain input combinations cannot occur, an 'X' (or 'd') is placed in the corresponding cell [1c, 269, 276]. These 'don't care' terms can be treated as either a '0' or a '1' during grouping to achieve maximum simplification [1d, 1e, 269, 276].

– **Steps for Logic Simplification Using K-Maps** : The process involves three main steps: Grouping, Covering, and Expressing.

1. **Grouping (Generating Prime Implicants)** :
   * **Identify Groups** : Form rectangular groups of '1's (for SoP) or '0's (for PoS) on the K-map.
   * **Group Size** : Groups must contain 2^k cells (i.e., 1, 2, 4, 8, 16, etc.).
   * **Adjacency** : Cells within a group must be horizontally or vertically adjacent. Diagonal grouping is not allowed.
   * **Wrap-Around** : Groups can wrap around the edges of the K-map (e.g., the leftmost column is adjacent to the rightmost, and the top row is adjacent to the bottom).
   * **Largest Possible** : Always make each group as large as possible. These are called **prime implicants** (for SoP) or **prime implicates** (for PoS). A prime group is one not entirely contained within a larger group.
   * **Overlap Allowed** : Cells may be included in multiple groups.
   * **Don't Cares in Grouping** : Include 'don't care' terms ('X') in groups to make them larger, but a group cannot consist solely of 'don't cares'. Don't care cells do not *have* to be covered, only used if they help simplify.
2. **Covering (Selecting Minimal Set of Prime Implicants/Implicates)** :
   * **Essential Prime Implicants (EPIs)** : First, identify all **essential prime implicants/implicates**. An essential prime implicant is a group that covers at least one '1' (or '0' for PoS) that no other prime implicant covers. These must be included in the final minimal expression.
   * **Minimal Cover** : If the essential primes do not cover all '1's (or '0's), then select the minimum number of additional non-essential prime implicants to cover the remaining uncovered '1's (or '0's). The goal is the lowest possible cost (fewest terms, fewest literals). Multiple solutions of the same minimal cost may exist.
3. **Expressing (Writing the Minimal Expression)** :

    - **For SoP (Sum of Products)** :
        - For each selected group of '1's, write a product term.
        - Identify the variables whose values remain constant within the group (e.g., if X is '1' across the entire group, include X; if Y is '0', include Y').
        - Ignore variables that change their value (alternate) within the group.
        - The final expression is the logical OR of all the product terms.
    - **For PoS (Product of Sums)** :
        - For each selected group of '0's, write a sum term.
        - Identify the variables whose values remain constant within the group (e.g., if X is '0' across the entire group, include X; if Y is '1', include Y').
        - Ignore variables that change their value (alternate) within the group.
        - The final expression is the logical AND of all the sum terms.
- **Timing Hazards Detection with K-Maps** :
    - K-maps can also be used to detect static hazards in two-level SoP (AND-OR) or PoS (OR-AND) circuits.
    - A **static-1 hazard** (a momentary '0' output when the output should remain '1') occurs in an SoP circuit when two adjacent '1's are covered by different product terms and these two terms are not "connected" by a common product term. This means there's a path between them that is not covered by a single prime implicant.
    - To eliminate a static hazard, an extra product term (called a **consensus term**) is added to cover the hazardous input pair. This typically involves grouping the two adjacent '1's with an additional prime implicant, even if it's not essential.
    - A **static-0 hazard** is the dual case for PoS circuits.

Exam Style Problem & Explanation (SoP Minimization with Don't Cares):

- **Problem** : Find the minimal SoP form of the Boolean expression represented by the K-map below (X indicates a don't care term). [197, Q1]
PQ/RS	00	01	11	10
00	1	0	1	1
01	0	1	1	0
11	X	1	0	0
10	X	0	X	X
- **Solution & Explanation**:
    1. **Identify '1's and 'X's** :
        - '1's are at (PQ, RS): (00,00), (00,11), (00,10), (01,01), (01,11), (11,01)
        - 'X's are at: (11,00), (10,00), (10,11), (10,10)
    2. **Grouping - SoP (looking for 1s and using X's to expand groups)** :
        - **Group 1 (Green Circle)** : Group the '1' at (00,00), the 'X' at (10,00), and the '1' at (01,00) (this cell is actually 0, so rethinking the provided solution) [197, Q1]. Let's re-evaluate based on the common solution pattern. The provided solution to Exam2F23 Q1 (which is identical to exam2Spring24 Q1) suggests the answer 'QR'S + Q'S' + Q'R'. Let's derive it.
        - Looking at the map and aiming for the *minimal* SoP, we try to form the largest possible groups of 1s (using X's where beneficial).
        - * | PQ/RS | 00 | 01 | 11 | 10 |
            |---|---|---|---|---|
            | 00 | **1** | 0 | **1** | **1** |
            | 01 | 0 | **1** | **1** | 0 |
            | 11 | X | **1** | 0 | 0 |
            | 10 | X | 0 | X | X |
        - **Group A (Q'S')** : Consider the '1' at (00,00) and the 'X' at (10,00). They form a column where R=0, S=0. The P-values are 00 and 10. If we combine them with a group (00,00), (10,00), (00,01), (10,01), and (00,10), (10,10).
        - The '1' at (00,00) (Q'P'R'S'). Can be grouped with the 'X' at (10,00) (QP'R'S'). This is QR'S'. (No, P is 00 or 10, Q is 0 or 1.) Let's use the provided solution as a guide to work backwards.
        - Let the variables be P (MSB for rows), Q (LSB for rows), R (MSB for columns), S (LSB for columns).

* **Term Q'S'** : This would cover all cells where Q=0 and S=0.
  - (00,00) PQ=00, RS=00 → P'Q'R'S' → This is 1.
  - (00,10) PQ=00, RS=10 → P'Q'RS' → This is 1.
  - (10,00) PQ=10, RS=00 → PQ'R'S' → This is X.
  - (10,10) PQ=10, RS=10 → PQ'RS' → This is X.
  - This group of 4 covers three 1s and two X's (including the '1' at (00,00) and (00,10)). This is a valid group.
* **Term Q'R** : This would cover all cells where Q=0 and R=1.
  - (00,11) PQ=00, RS=11 → P'Q'RS → This is 1.
  - (00,10) PQ=00, RS=10 → P'Q'RS' → This is 1.
  - (10,11) PQ=10, RS=11 → PQ'RS → This is X.
  - (10,10) PQ=10, RS=10 → PQ'RS' → This is X.
  - This group of 4 covers the '1' at (00,11) and (00,10).
* **Term QR'S** : This would cover all cells where Q=1, R=0, S=1.
  - (01,01) PQ=01, RS=01 → P'QRS → This is 1. (No, P'Q R'S)
  - (11,01) PQ=11, RS=01 → PQRS → This is 1. (No, PQ R'S)
  - Let's check the terms given by the solution: A. QR'S + Q'S' + Q'R.
  - Let the K-map variables be P, Q (rows) and R, S (columns). So the function is F(P,Q,R,S).
* Let's re-do the K-Map with variables from the provided solution's options, which appear to use Q, R, S as inputs and P is implicitly gone or fixed, or the question is using Q,R,S as some of the variables and P as the others. The K-map shows PQ and RS. Let's assume P,Q,R,S are the 4 variables.

PQ/RS	00 (R'S')	01 (R'S)	11 (RS)	10 (RS')
00 (P'Q')	1	0	1	1
01 (P'Q)	0	1	1	0
11 (PQ)	X	1	0	0
10 (PQ')	X	0	X	X

  - **Group 1 (Essential Prime Implicant): P'Q'S' (covers (00,00), (00,10))**
  - The '1' at (00,00) is only covered by this group of 2: P'Q'R'S'.
  - The '1' at (00,10) is also covered by this group of 2: P'Q'RS'.
  - So, 'P'Q'S'' (P'Q'R'S' + P'Q'RS') is a group covering (00,00) and (00,10). This term simplifies to 'P'Q'S''. It's an EPI because (00,00) can only be covered by this and perhaps (10,00) (X).

3 Section 1c

(c) Timing Hazards in Combinational Circuits

Defining Timing Hazards

• **Hazards (glitches)** are undesirable momentary changes or short pulses in the output of a combinational logic circuit that occur when the steady-state analysis predicts the output should not change. They are caused by unequal gate propagation delays within the circuit.

• **Static Hazards** : The circuit's output momentarily changes from its correct, stable state before settling.

◦ **Static-1 Hazard** : Occurs when the output should remain at logic '1', but momentarily dips to '0'. This happens between two input combinations that: (a) differ in only one input variable, and (b) both produce a '1' output, where a momentary '0' output is possible during the transition of the differing input variable.

• *Example* : For a function F=X+X', if X transitions from 0 to 1, the output should ideally stay at 1. However, due to delays, a brief '0' glitch can occur.

◦ **Static-0 Hazard** : Occurs when the output should remain at logic '0', but momentarily spikes to '1'. This is the dual of a static-1 hazard, meaning it happens between two input combinations that: (a) differ in only one input variable, and (b) both produce a '0' output, where a momentary '1' output is possible during the transition of the differing input variable.

• **Dynamic Hazards** : The circuit's output changes multiple times (more than once) as a result of a single input transition, often described as a 'bounce'.

◦ **Dynamic 0-to-1 Hazard** : The output changes from 0 to 1, then back to 0, and then again to 1 ($0 \to 1 \to 0 \to 1$).

◦ **Dynamic 1-to-0 Hazard** : The output changes from 1 to 0, then back to 1, and then again to 0 ($1 \to 0 \to 1 \to 0$).

◦ Dynamic hazards can occur if there are multiple paths with differing propagation delays from the changing input to the output.

Identifying Hazards Using K-Maps

• K-maps can be used to detect static hazards in two-level Sum-of-Products (SoP) or Product-of-Sums (PoS) circuits.

• The occurrence of static hazards depends on the specific circuit design (realisation) of a logic function.

• A correctly designed two-level AND-OR (SoP) circuit will not have static-0 hazards but may exhibit static-1 hazards.

• **Detecting Static-1 Hazards on a K-map (for SoP circuits)** :

◦ A static-1 hazard is indicated on a K-map whenever two adjacent cells containing '1's are not covered by a single, common product term (group of 1s). This means that a transition between these two adjacent '1' states could cause a momentary '0' output because the terms covering them might turn off and on at slightly different times.

◦ *Example* : Consider the function F(X,Y,Z) = XZ' + YZ. In this K-map, the cells for (X=1, YZ=00) and (X=1, YZ=10) are adjacent '1's (representing minterms XZ' and XY'Z'). Also, the cells (X=0, YZ=11) and (X=1, YZ=11) are adjacent (representing minterms YZ and XYZ). Specifically, if Z transitions from 1 to 0 (e.g., from X=1, Y=1, Z=1 to X=1, Y=1, Z=0), the output should ideally remain '1'. However, the terms XZ' and YZ cover adjacent '1's but are covered by separate product terms, potentially leading to a momentary '0' glitch.

Designing Hazard-Free Circuits

• **Eliminating Static Hazards (using Consensus Terms)** :

◦ To eliminate a static hazard, an additional product term, known as a **consensus term** , should be included in the Boolean expression. This term is designed to cover the hazardous pair of input combinations.

◦ The consensus term ensures that during the transition between the hazardous states, at least one product term remains active, preventing the output from momentarily changing.

◦ *Example (continuing F(X,Y,Z) = XZ' + YZ)* :

• The consensus term for XZ' and YZ is XY. By adding this term, the function becomes F(X,Y,Z) = XZ' + YZ + XY. The added consensus term (XY, represented by the dashed green line) now directly covers the transition path between adjacent '1's that were previously only covered by separate terms.

• **Brute Force Method** : If circuit cost is not a primary concern, a hazard-free realisation can be achieved by using the **complete sum** , which means including all prime implicants in the design.

• **General Design Principle** : Functions with non-adjacent product terms are inherently hazardous when multiple inputs change simultaneously. For dynamic hazards, rearranging multi-level circuits into two-level designs can help reduce them.

Exam-Style Problem: The circuit below implements F(X) using an inverter and an XOR gate. [

] The circuit above exhibits the following type of hazard: (A) Static 0 hazard only when X transitions from 0 to 1 (B) Static 1 hazard when X transitions from 0 to 1 and also when X transitions from 1 to 0 (C) Static 1 hazard only when X transitions from 1 to 0 (D) Static 0 hazard when X transitions from 0 to 1 and also when X transitions from 1 to 0 (E) None of the above

Explanation: The circuit takes an input X, which is then inverted and fed into an XOR gate along with the original X input. Ideally, an XOR gate's output is '1' if its inputs are different, and '0' if they are the same. Since the inputs to this XOR gate are X and its complement X', the ideal output F should always be '1' (X XOR X' = 1).

However, due to the inherent propagation delay in the inverter, when the input X changes its value (e.g., from 0 to 1 or from 1 to 0), the output of the inverter (X') does not change immediately. This brief delay means that for a short period, both inputs to the XOR gate (X and the delayed X') might momentarily be the same value (either both '0' or both '1').

When the inputs to the XOR gate are momentarily identical, its output will momentarily switch to '0'. Since the ideal output should be '1', this temporary drop from '1' to '0' and back to '1' is characteristic of a **static-1 hazard**. This hazard occurs for transitions in both directions (X changing from 0 to 1, and X changing from 1 to 0) because the propagation delay causes the temporary input mismatch in both scenarios.

Answer: (B) Static 1 hazard when X transitions from 0 to 1 and also when X transitions from 1 to 0.

—

4 Section 2a

4.1 Decoders (15%)

4.2 Understanding Decoders

Definition and Role of n:2^n Decoders

• A **decoder** is a multiple-input, multiple-output logic circuit that converts coded inputs into coded outputs.

• The input code generally has fewer bits than the output code.

• In a one-to-one mapping, each input code word produces a different output code word.

• The most common decoder circuit is an **n-to-2^n decoder** or **binary decoder**.

◦ It has an n-bit binary input code and a 1-out-of-2^n output code.

◦ This means for n inputs, there are 2^n unique outputs, and only one of these outputs will be "active" at any given time, corresponding to the binary value of the inputs.

• **Role in Digital Circuits** : Decoders are used to activate exactly one of 2^n outputs based on an n-bit input value.

◦ *Analogy* : An electronically-controlled rotary selector switch.

◦ Decoders are often considered the "inverse" of encoders, as the roles of inputs and outputs are reversed.

◦ A device that routes an input to one of 2^n outputs is also typically referred to as a **(1-to-2^n) demultiplexer**.

4.3 Example: 2-to-4 (2:4) Decoder

• A 2:4 decoder has two select lines (S1, S0) and four outputs (Y0, Y1, Y2, Y3).

• The outputs correspond to the minterms of the inputs (e.g., Y0 for S1'S0', Y1 for S1'S0, etc.).

4.4 Comparing High-Active (Minterm-Based) vs. Low-Active (Maxterm-Based) Outputs

The activity level of a decoder's output defines whether a '1' or '0' logic level signifies the selected output.

• **High-Active Outputs (Minterm-Based)** :

◦ When an input combination is selected, its corresponding output goes to logic '1', while all other outputs remain at '0'.

◦ Each output of an n-to-2^n binary decoder represents a **minterm** of the n-variable Boolean function.

◦ Therefore, any arbitrary Boolean function of n-variables can be realized by OR-ing the needed high-active outputs.

◦ **Truth Table Example (2:4 Decoder with High-Active Outputs)** :

X1	X0	Y0	Y1	Y2	Y3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

◦ **Logic Expressions for High-Active Outputs** :

• $Y_0 = X_1' X_0'$

• $Y_1 = X_1' X_0$

• $Y_2 = X_1 X_0'$

• $Y_3 = X_1 X_0$

• **Low-Active Outputs (Maxterm-Based)** :

◦ When an input combination is selected, its corresponding output goes to logic '0', while all other outputs remain at '1'.

◦ Low-active decoder outputs represent **maxterms**. If the decoder outputs are active low, a NAND gate can be used to "OR" the minterms of the function (representing its ON set), or an AND gate can be used to "OR" the minterms of the complement function (representing its OFF set).

◦ Sometimes, active low outputs are indicated with overlines on the signals, or with both overlines and bubbles, which still means an inverted output.

◦ **Truth Table Example (2:4 Decoder with Low-Active Outputs)** :

X1	X0	Y0	Y1	Y2	Y3
0	0	0	1	1	1
0	1	1	0	1	1
1	0	1	1	0	1
1	1	1	1	1	0

◦ **Logic Expressions for Low-Active Outputs** :

• $Y_0 = (X_1' X_0')'$ (which simplifies to $X_1 + X_0$)

• $Y_1 = (X_1' X_0)'$ (which simplifies to $X_1 + X_0'$)

• $Y_2 = (X_1 X_0')'$ (which simplifies to $X_1' + X_0$)

• $Y_3 = (X_1 X_0)'$ (which simplifies to $X_1' + X_0'$)

4.5 Explaining the Use of Enable Signals

• An **Enable (EN) signal** is typically provided in decoders to control their operation.

• **Purpose** : The enable input allows for cascading decoders to create larger decoders (decoder trees) or to simply turn the decoder on or off.

• **Behavior when Disabled (EN = inactive)** :

◦ If the enable signal is inactive (e.g., low for an active-high enable, or high for an active-low enable), all outputs of the decoder will be in their inactive state, regardless of the select inputs.

◦ For high-active outputs, all outputs will be '0'.

◦ For low-active outputs, all outputs will be '1'.

• **Behavior when Enabled (EN = active)** :

◦ If the enable signal is active (e.g., high for an active-high enable, or low for an active-low enable), the decoder functions normally. The output corresponding to the binary input value will be active, and all other outputs will be inactive.

◦ For high-active enable with high-active outputs, the selected output is 'EN * minterm_i'.

◦ For high-active enable with low-active outputs, the selected output is '(EN * minterm_i)'' which is 'EN' + maxterm_i'.

• **Types of Enable Signals** : Decoders can have active-high enable(s) or active-low enable(s). Some complex decoders, like the 74HC138, might have multiple enable inputs with mixed polarities (e.g., one active-high, two active-low) which are all ANDed/NANDed together to produce a single internal enable signal.

Exam-Style Problem: (Source: Exam2SP24, Question 8) Consider the following 2:4 decoder circuit. What is the logic expression of F?

(A) C=D (B) (A=B).(D'+C) (C) (A=B)'.(D'+C) (D) (A=B).(D'+C') (E) None of the above

Explanation:

1. **Analyze the first decoder (left)** :

◦ Inputs: A, B. Enable is grounded (active-low enable, meaning it's always enabled if low). This decoder is implicitly active-high output since its outputs are fed directly into the inputs of the next decoder and then an OR gate without inversion.

◦ Outputs:

• D0 (left) = A'B'

• D1 (left) = A'B

• D2 (left) = AB'

• D3 (left) = AB

1. **Analyze the second decoder (right)** :

◦ Inputs: C, D. Enable is grounded (always enabled). This decoder is also implicitly active-high output.

◦ The connections are crucial:

• Input I0 (right) is connected to D3 (left) = AB

• Input I1 (right) is connected to D2 (left) = AB'

◦ So, the internal select lines of the right decoder are effectively (AB, AB').

◦ Outputs of the second decoder:

• D0 (right) = I0'(right) I1'(right) = (AB)' (AB')' = (A'+B)(A'+B')

• D1 (right) = I0'(right) I1 = (AB)' (AB)

• D2 (right) = I0 I1'(right) = (AB) (AB')'

• D3 (right) = I0 I1 = (AB) (AB')

1. Let's re-evaluate the connection based on the general form of a 2:4 decoder and the answer choices. The problem implies (A=B) and (D'+C) are related. The outputs of a 2:4 decoder are minterms of its inputs (I1, I0). So, for the second decoder, let's denote its inputs as I_1 = D2 (left) = AB' and I_0 = D3 (left) = AB. The outputs of the second decoder are:

◦ $Y_0 = I_1' I_0' = (AB')'(AB)' = (A' + B)(A' + B')$

◦ $Y_1 = I_1' I_0 = (AB')'(AB)$

◦ $Y_2 = I_1 I_0' = (AB')(AB)'$

◦ $Y_3 = I_1 I_0 = (AB')(AB) = 0$ (since AB' and AB are mutually exclusive)

1. This is not matching the options easily, which suggests a different interpretation of the problem's inputs or the typical convention. Let's look at the given solution. The solution is B: $(A = B).(D' + C)$. This implies the first part $(A = B)$ controls the enable of the second decoder, and $(D' + C)$ comes from the second decoder. However, the enable of both decoders is grounded.

1. Let's re-examine the image to confirm the connection of C and D. C and D are *inputs* to the second decoder, mapping to I0 and I1, respectively. So, the second decoder has inputs I0=C, I1=D. Its outputs are:

◦ D0 (right) = C'D'

◦ D1 (right) = C'D

◦ D2 (right) = CD'

◦ D3 (right) = CD

1. The OR gate combines four outputs from the second decoder. Let's see what outputs are connected:

◦ Input 1 to OR gate: D3 (right) = CD

◦ Input 2 to OR gate: D2 (right) = CD'

◦ Input 3 to OR gate: D1 (right) = C'D

◦ Input 4 to OR gate: D0 (right) = C'D'

5 Section 2b

(b) Designing Decoders

This section focuses on the practical aspects of designing decoders, including their implementation using logic gates and how to expand them into larger "decoder trees" [2c].

Logic Gate-Based Implementation of a Decoder

A decoder is a combinational logic circuit that converts a coded input, typically an n-bit binary code, into a coded output, usually a 1-out-of-2^n code. This means that for each unique input combination, a different, mutually exclusive output line is asserted.

– **Basic 2-to-4 Decoder (Active-High Outputs, Active-High Enable)** :

  * A common type is the n-to-2^n binary decoder, which has an n-bit binary input and a 1-out-of-2^n output code. It's used to activate exactly one of 2^n outputs based on the n-bit input value.

    * **Enable Signal:** Decoders often include an 'Enable' (EN) input. When this signal is inactive (e.g., low for an active-high enable), all outputs are typically de-asserted (e.g., all low for active-high outputs). When the enable signal is active, the decoder functions normally, asserting the output corresponding to the binary input.
    * **Truth Table** :
    * **Logic Expressions (Sum-of-Minterms)** : Each output (Y_i) corresponds to a minterm of the input variables (S1, S0), ANDed with the Enable signal (EN):
        · $Y_0 = EN \cdot S1' \cdot S0'$
        · $Y_1 = EN \cdot S1' \cdot S0$
        · $Y_2 = EN \cdot S1 \cdot S0'$
        · $Y_3 = EN \cdot S1 \cdot S0$
    * **Gate-Level Implementation:** Based on these expressions, a 2-to-4 decoder can be implemented using basic logic gates (AND gates for each output, and inverters for the complemented select lines).
- **Active-Low Outputs** :
    * Some decoders have active-low outputs, indicated by bubbles on the output lines or overlines in signal names.
    * For an active-low output decoder with an active-high enable, when enabled, exactly one output will be logic '0' (low), while all others are '1' (high).
    * These active-low outputs correspond to maxterms of the input variables.
    * An example is the 74HC138 chip, which is a 3-to-8 decoder with active-low outputs and multiple enable inputs (two active-low, one active-high).

Decoder Trees (Expanding Decoders)

Larger decoders can be built by combining smaller decoders in a hierarchical arrangement, often referred to as a "decoder tree" [2c, 303, 309].

- **Concept** :
    * The total set of input select lines is partitioned into two groups: "group-selects" (more significant bits) and "member-selects" (less significant bits).
    * The group-selects are used to enable one specific smaller decoder (the "group" decoder).
    * The member-selects are then routed to the data inputs of all the smaller decoders in the tree.
    * This structure allows a larger number of unique outputs to be generated than a single smaller decoder could provide.
- **Example: Building a 3:8 Decoder from 2:4 Decoders** :
    * To build a 3:8 decoder (3 input bits, 8 outputs), you could use a 1:2 decoder and two 2:4 decoders.
    * Let the 3 input bits be S2 (MSB), S1, S0 (LSB).
    * S2 can be used as the select line for a 1:2 decoder. The two outputs of the 1:2 decoder serve as enable signals for two separate 2:4 decoders.
    * Both 2:4 decoders would receive S1 and S0 as their common select inputs.
    * Only one of the 2:4 decoders will be enabled at any given time (based on S2), and that enabled decoder will then assert one of its four outputs based on S1 and S0, effectively generating 8 unique outputs.
- **Application: Memory Address Decoding** :
    * Decoders are fundamental in memory systems for selecting specific memory locations or chips.
    * In a larger memory system, higher-order address bits are often fed into a decoder (or a decoder tree) to select a particular memory chip or block.
    * The lower-order address bits are then used to specify the exact location within the selected chip or block. For example, a 3:8 decoder can select one of eight memory chips in a 32KB memory system using 15 address lines, where the top 3 lines go to the decoder and the bottom 12 go to the chips directly.

Exam Style Problem

- **Question:** What combinational device with outputs A-H is implemented by the left part of the circuit below? And which AND gate outputs should be connected to the OR gate to implement $Y = S2 \cdot S0' + S1 \cdot S0$? *(Refer to the circuit diagram provided in the original query, specifically the left portion with inputs EN, S2, S1, S0 connected to eight AND gates producing outputs A-H.)*
- **Explanation:**

* **Device Identification:** The left part of the circuit has three input lines (S2, S1, S0) serving as select lines and an Enable (EN) input. It produces eight distinct outputs (A through H). Each AND gate effectively decodes a unique combination of the select lines (minterm) when the enable is active. Therefore, this circuit implements a 3:8 Decoder. Since the AND gates require all their inputs to be high to produce a high output, the Enable signal (EN) must be an active-high enable.
* **Output Connection for $Y = S2 \cdot S0' + S1 \cdot S0$:**
  - To implement the boolean expression $Y = S2 \cdot S0' + S1 \cdot S0$, we need to identify the minterms that make the function true. The provided solution indicates that connecting outputs D, E, G, H to the OR gate produces the expression $Y = S2'S1S0 + S2S1'S0' + S2S1S0' + S2S1S0$.
  - This expression can be simplified as follows:
  - Group terms with $S2 \cdot S0'$: $S2S1'S0' + S2S1S0' = S2S0'(S1' + S1) = S2S0'$ (Outputs E, G).
  - Group terms with $S1 \cdot S0$: $S2'S1S0 + S2S1S0 = S1S0(S2' + S2) = S1S0$ (Outputs D, H).
  - Combining these, the resulting expression is $S2S0' + S1S0$, which matches the target function.
  - Therefore, the outputs D, E, G, and H should be connected as inputs to the OR gate.

—

6 Section 2c

(c) Using Decoders for Logic Function Implementation

Decoders are versatile combinational logic blocks whose primary outputs represent the minterms (or maxterms) of their inputs. This inherent property makes them suitable for implementing any arbitrary Boolean function [84, lec2.5Decoders1.pdf].

How Decoders Implement Arbitrary Boolean Functions

– **Decoder Outputs as Minterms:** An n-to-2^n binary decoder converts an n-bit binary input into a 1-out-of-2^n code output. This means for each unique input combination, exactly one output line is asserted.

  * When the outputs are **active-high** , each output (Y_i) corresponds directly to a specific minterm (m_i) of the decoder's select inputs [84, lec2.5Decoders1.pdf, slide 10].
  * When the outputs are **active-low** , each output ($\overline{Y_i}$) corresponds to the complement of a minterm ($\overline{m_i}$) of the decoder's select inputs [85, lec2.5Decoders1.pdf, slide 12].

– **Realising Boolean Functions:**

  * Any Boolean function can be expressed as a Sum-of-Minterms (ON set).
  * Since a decoder's outputs directly generate these minterms (or their complements), an additional logic gate can combine the required minterms to form the desired function.

Step-by-Step Examples of Using Decoders in Circuit Design

Example 1: Implementing a function using an Active-High Decoder

To implement a Boolean function F with active-high decoder outputs:

1. **Express the function as a Sum-of-Minterms (ON set)**. Identify all input combinations for which the function output is '1'.

1. **Select the corresponding decoder outputs:** For each minterm in the function's ON set, identify the corresponding active-high output line of the decoder.

1. **Combine with an OR gate:** Connect all the selected active-high decoder outputs to the inputs of a multi-input OR gate. The output of this OR gate will be the desired Boolean function F.

– **Illustration:** Consider implementing the function $F(X, Y, Z) = X'YZ + XY'Z'$ using a 3-to-8 active-high decoder.

  * The minterms in the ON set are $X'YZ$ (which is m_3) and $XY'Z'$ (which is m_4).
  * A 3-to-8 decoder with inputs X, Y, Z (where X is MSB, Z is LSB) will produce active-high outputs $Y_0, Y_1, \ldots, Y_7$, where Y_i corresponds to minterm m_i.

* Therefore, connect the decoder outputs Y_3 and Y_4 to a 2-input OR gate.
* The output of this OR gate is F.

Example 2: Implementing a function using an Active-Low Decoder

When using decoders with active-low outputs:

1. **Express the function as a Sum-of-Minterms (ON set)**.

1. **Select the corresponding decoder outputs:** For each minterm in the function's ON set, identify the corresponding active-low output line of the decoder.

1. **Combine with a NAND gate:** Connect all the selected active-low decoder outputs to the inputs of a multi-input NAND gate. The output of this NAND gate will be the desired Boolean function F.

- This works because if Y_i is an active-low output, it represents $\overline{m_i}$ (the complement of minterm i).
- Applying De Morgan's Law: $F = m_a + m_b + \cdots = \overline{\overline{m_a}} + \overline{\overline{m_b}} + \cdots = \overline{(\overline{m_a} \cdot \overline{m_b} \cdot \dots)}$.
- Thus, connecting $\overline{Y_a}, \overline{Y_b}, \dots$ to a NAND gate produces F.
- **Illustration:** Implement $F(X, Y, Z) = X'YZ + XY'Z'$ using a 3-to-8 active-low decoder.
  * The minterms in the ON set are $X'YZ$ (m_3) and $XY'Z'$ (m_4).
  * A 3-to-8 decoder with active-low outputs will produce $\overline{Y_0}, \overline{Y_1}, \dots, \overline{Y_7}$.
  * Connect the active-low decoder outputs $\overline{Y_3}$ and $\overline{Y_4}$ to a 2-input NAND gate.
  * The output of this NAND gate is F.

Exam Style Problem

- **Question (Adapted from Fall 2023 Exam 2, Q11****):** Which of the following is the correct representation of F using the OFF set, for a circuit with a 3-to-8 active-low decoder and a 5-input NAND gate? The outputs connected to the NAND gate correspond to minterms m_1, m_3, m_5, m_6, m_7.
  * A. $\Pi_{X,Y,Z}(0, 2, 4, 6)$
  * B. $\Pi_{X,Y,Z}(1, 3, 5, 7)$
  * C. $\Pi_{X,Y,Z}(0, 2, 4)$
  * D. $\Pi_{X,Y,Z}(1, 5, 6, 7)$
  * E. None of the above
- **Explanation:**
  1. **Identify the components:** The circuit consists of a 3-to-8 active-low decoder and a 5-input NAND gate.
  2. **Understand active-low outputs:** For an active-low decoder, an output Y_i being low means the input corresponds to minterm m_i. Thus, each output Y_i carries the logic $\overline{m_i}$.
  3. **Analyse the NAND gate operation:** The NAND gate takes several inputs and produces an output F. If the inputs to the NAND gate are A, B, C, D, E, the output is $(A \cdot B \cdot C \cdot D \cdot E)'$.
  4. **Apply De Morgan's Law:** When an active-low decoder's outputs ($\overline{m_i}$) are fed into a NAND gate, the function implemented is the OR of the corresponding minterms. That is, if outputs $\overline{Y_{i_1}}, \overline{Y_{i_2}}, \dots, \overline{Y_{i_k}}$ are connected to the NAND gate, the output F will be: $F = (\overline{Y_{i_1}} \cdot \overline{Y_{i_2}} \cdot \dots \cdot \overline{Y_{i_k}})' = (\overline{m_{i_1}} \cdot \overline{m_{i_2}} \cdot \dots \cdot \overline{m_{i_k}})' = \overline{\overline{m_{i_1}}} + \overline{\overline{m_{i_2}}} + \cdots + \overline{\overline{m_{i_k}}} = m_{i_1} + m_{i_2} + \cdots + m_{i_k}$.
  5. **Determine the ON set of F:** The problem states that the outputs connected to the NAND gate correspond to minterms m_1, m_3, m_5, m_6, m_7. Therefore, the ON set of function F is $\Sigma_{X,Y,Z}(1, 3, 5, 6, 7)$.
  6. **Determine the OFF set of F:** The OFF set includes all minterms that are *not* in the ON set. For a 3-input function, the possible minterms are $0, 1, 2, 3, 4, 5, 6, 7$.
     * Comparing the full set of minterms with the ON set: $0, 1, 2, 3, 4, 5, 6, 7 \setminus 1, 3, 5, 6, 7 = 0, 2, 4$.
     * The OFF set is thus $\Pi_{X,Y,Z}(0, 2, 4)$.
- **Answer:** C.

7 Section 3a

(a) Understanding Encoders

Encoders are combinational logic devices that perform the reverse operation of decoders, converting a large number of input lines into a smaller number of output lines, typically representing a binary code.

Definition: 2^n-to-n Binary-to-Decimal Encoder

- **Functionality:** A 2^n-to-n binary encoder converts a 2^n-bit input code, where typically only one input line is active at a time, into an n-bit binary output code.
- It acts as an "inverse decoder".
- The output is the binary representation of the activated input line's index.
- **Analogy:** Imagine a box with 16 push buttons that outputs a 4-bit binary pattern corresponding to the pressed button.
- **Assumption:** For a basic encoder, it is assumed that one and only one input line is active (HIGH) at any given time. If no input is active, the output might be all zeros (representing index 0), which can be ambiguous if input 0 is also active.
- **Truth Table Example (4:2 Binary Encoder):** Assume inputs W_3, W_2, W_1, W_0 and outputs Y_1, Y_0.

W_3	W_2	W_1	W_0	Y_1
0	0	0	1	0
0	0	1	0	0
0	1	0	0	1
1	0	0	0	1

This table is incomplete as it only shows cases where exactly one input is active. Other input combinations (e.g., all zeros or multiple ones) might lead to undefined or undesired outputs for a basic encoder .

Definition: Basic Encoder with Strobe Output

- **Purpose:** To address the ambiguity when no input is active, an encoder may include a "strobe" or "gate" output (often denoted as G or Z).
- **Functionality:**
- The strobe output (Z) is asserted (e.g., HIGH) if any input is active.
- If no input is asserted, the strobe output is de-asserted (e.g., LOW), indicating that the binary output ($Y_n \dots Y_0$) should be ignored.
- If multiple inputs are active simultaneously, a basic encoder with a strobe output will still produce an "unknown" or undefined binary code for the $Y_n \dots Y_0$ lines, while the strobe output will be active.
- **Truth Table Example (4:2 Binary Encoder with Strobe):**

W_3	W_2	W_1	W_0	Y_1	Y_0	Z
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	0	0	1	1
0	1	0	0	1	0	1
1	0	0	0	1	1	1

Note: 'X' indicates a don't care term, meaning the output is not meaningful in that state .

Definition: Priority Encoder

- **Problem Addressed:** The fundamental limitation of basic encoders is their inability to handle situations where more than one input is asserted simultaneously. A priority encoder is designed to solve this problem.
- **Functionality:** A priority encoder assigns a priority to each input line. When multiple inputs are asserted at the same time, the encoder will output the binary code corresponding to the **highest priority** input that is active. All other active inputs with lower priority are ignored.
- **Common Priority Assignment:** Typically, the input with the highest index (e.g., I_7 in an 8-to-3 encoder) is assigned the highest priority.
- **Strobe Output:** Priority encoders often include a strobe (or IDLE) output to indicate if any input is active, similar to basic encoders with strobe. This output allows distinguishing between the case where input 0 is active and the case where no input is active.

- **Truth Table Example (Reduced, 8:3 Priority Encoder with IDLE output):** *IDLE is 1 if all inputs are 0, otherwise IDLE is 0 and output is binary code of highest priority active input .*

I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0	A_2	A_1	A_0	IDLE
1	X	X	X	X	X	X	X	1	1	1	0
0	1	X	X	X	X	X	X	1	1	0	0
0	0	1	X	X	X	X	X	1	0	1	0
0	0	0	1	X	X	X	X	1	0	0	0
0	0	0	0	1	X	X	X	0	1	1	0
0	0	0	0	0	1	X	X	0	1	0	0
0	0	0	0	0	0	1	X	0	0	1	0
0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	X	X	X	1

Note: 'X' indicates a don't care term for lower priority inputs, as their state does not affect the output when a higher priority input is active .

Differences between Basic Encoders and Priority Encoders

The key distinction lies in how they handle multiple active inputs:

- **Basic Encoder:** If multiple inputs are active simultaneously, a basic encoder's output is undefined or produces an incorrect binary code. It relies on the assumption that only one input will be active at a time.
- **Priority Encoder:** It explicitly resolves the issue of multiple active inputs by implementing a priority scheme. Only the highest priority active input is encoded, ensuring a predictable and correct binary output even when multiple buttons are pressed.

Exam Style Problem

- **Question (Fall 2023 Exam 2, Q9):** Choose the combinational block that this logic diagram implements. *(Self-correction: The provided diagram in Q9 is for an 8:3 priority encoder, not a simple AND/OR gate diagram. I will use the description of its structure to explain the priority encoder behavior.)*
- The logic diagram shows an 8-input circuit with OR gates and inverters leading to 3 output lines (Y_2, Y_1, Y_0). Specifically, the outputs are formed by ORing inputs (or combinations of inputs and their complements) such that higher indexed inputs (e.g., D_7) affect the output bits, and lower indexed inputs are conditionally blocked. For example, for Y_2: it's an OR gate of D_7, D_6, D_5, D_4. For Y_1, it involves D_7, D_6, D_3, D_2 with some inversions. For Y_0, it involves D_7, D_5, D_3, D_1 with some inversions. The way the AND gates with inverters are structured ensures that for a given bit, no bits with higher priority are active.
- A. 3:8 Decoder
- B. 8:3 Priority encoder
- C. 8:3 Basic (Non-priority) encoder
- D. 2:4 decoder
- E. None of the others
- **Explanation:**

1. **Analyze the inputs and outputs:** The circuit has 8 inputs (D0 through D7) and 3 outputs (Q0, Q1, Q2). This immediately suggests an 8-to-3 encoder or decoder. Options A and D are decoders, which convert fewer inputs to more outputs (n to 2^n), so they are incorrect.

1. **Examine the internal logic:** The presence of OR gates and specific AND gates with NOT (inverters) on the input lines for outputs indicates a logic that selects based on conditions. For instance, the output Q_2 (or Y_2 in some notation) is formed by an OR gate that takes inputs I_4, I_5, I_6, I_7. This means if any of I_4, I_5, I_6, I_7 are active, Q_2 will be '1' regardless of lower inputs.

1. **Identify priority behavior:** The explanation states, "the AND gates with the nots ensure that for a given bit, no bits with higher priority are active". This directly describes the core functionality of a priority encoder: higher-indexed inputs override lower-indexed ones. For example, if D_7 is high, it dictates the output, regardless of $D_6, \ldots, D_0$. If D_7 is low but D_6 is high, D_6 then determines part of the output, overriding $D_5, \ldots, D_0$, and so on.

1. **Conclusion:** The circuit's behavior, where higher-indexed inputs override lower-indexed inputs to determine the output, matches the definition of a priority encoder.

- **Answer:** B.

8 Section 3b

(b) Designing Encoders

Designing encoders involves translating their specified functionality (whether basic or priority) into a combinational logic circuit using fundamental gates (AND, OR, NOT). This process typically begins with a truth table, from which Boolean expressions are derived, and then implemented using logic gates.

1. Implementing a Basic Binary Encoder

A basic binary encoder, such as a 2^n-to-n encoder, assumes that only one input line is active at any given time [lec2.6Encoders.pdf]. The design involves directly ORing the inputs that should contribute to a '1' for each output bit.

- **Example: 4:2 Binary Encoder**
- Inputs: W_0, W_1, W_2, W_3
- Outputs: Y_0, Y_1
- **Truth Table (assuming only one input is high):** [lec2.6Encoders.pdf, page 15]

W_3	W_2	W_1	W_0	Y_1	Y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

- **Boolean Expressions:**

  For Y_0: It is '1' when W_1 is active (encoding '01') or W_3 is active (encoding '11'). $Y_0 = W_1 + W_3$ [lec2.6Encoders.pdf, page 15 shows the logic for an 8:3 example where Y0 corresponds to I1, I3, I5, I7, which are ORed]

  For Y_1: It is '1' when W_2 is active (encoding '10') or W_3 is active (encoding '11'). $Y_1 = W_2 + W_3$

- **Circuit Diagram (conceptual, using OR gates):**
- **Limitation:** This basic design will produce an undefined or incorrect output if multiple inputs are active simultaneously [lec2.6Encoders.pdf]. For instance, if W_1 and W_2 are both high, $Y_0 = 1$ and $Y_1 = 1$, which encodes '11' (for W_3), not a meaningful representation of W_1 and W_2 simultaneously.

1. Implementing a Priority Encoder

A priority encoder resolves the issue of multiple active inputs by assigning a priority level to each input. When multiple inputs are asserted, the output corresponds to the highest priority active input [Supplement 2E Encoders.pdf, lec2.6Encoders.pdf].

- **Example: 4:2 Priority Encoder with Strobe Output (Z)**
- Inputs: W_0, W_1, W_2, W_3 (where W_3 has highest priority, W_0 lowest)
- Outputs: Y_0, Y_1, and Strobe (Z)
- **Functional Truth Table (reduced):** [lec2.6Encoders.pdf, page 19 shows a similar table for an 8:3 encoder]

W_3	W_2	W_1	W_0
0	0	0	0
0	0	0	1
0	0	1	X
0	1	X	X
1	X	X	X

  Note: 'X' indicates a don't care term. For example, if W_2 is high, the state of W_1 and W_0 does not matter as W_2 has higher priority.

- **Boolean Expressions (derived from K-maps or direct observation):**

  Strobe (Z): The strobe output is active if _any_ input is active. $Z = W_0 + W_1 + W_2 + W_3$

  Output Y_0: $Y_0 = W_1\bar{W}_2\bar{W}_3 + W_3$ (This simplifies to $Y_0 = W_1\bar{W}_2 + W_3$ if we consider the effect of W_3 having highest priority. For a complete SOP, it must cover conditions where W_3 is active and W_1 is active and W_2 is not. Based on the 8:3 priority encoder logic from slides, $A_0 = I_7 + I_6'I_5 + I_6'I_4'I_3 + I_6'I_4'I_2'I_1$. Applying this pattern to 4:2: $Y_0 = W_3 + W_2'W_1$.) Let's re-derive for clarity:

- $Y_0 = W_3 + W_2'W_1$
- $Y_1 = W_3 + W_2$
- $Z = W_3 + W_2 + W_1 + W_0$ This interpretation implicitly captures the priority; e.g., for Y_0, if W_3 is high, Y_0 is high. If W_3 is low, but W_2 is low and W_1 is high, Y_0 is high.

  From lec2.6Encoders.pdf (8:3 Priority Encoder adapted for 4:2): Using the pattern from the 8:3 encoder on page 19 of lec2.6Encoders.pdf, where I_7 is MSB, I_0 is LSB: $A_2 = I_7 + I_6 + I_5 + I_4$ $A_1 = I_7 + I_6 + I_5'I_4'I_3 + I_5'I_4'I_2$ $A_0 = I_7 + I_6'I_5 + I_6'I_4'I_3 + I_6'I_4'I_2'I_1$ Applying this to our 4:2 example (using W_3, W_2, W_1, W_0 for I_3, I_2, I_1, I_0 respectively, assuming W_3 highest priority): $Y_1 = W_3 + W_2$ $Y_0 = W_3 + W_2'W_1$ $Z = W_3 + W_2 + W_1 + W_0$ (which is $IDLE'$ in the slide, so $IDLE = (I_7 + ... + I_0)'$)
- **Circuit Diagram (conceptual, using AND, OR, NOT gates to implement priority):**

  Exam Style Problem
- **Question (Spring 2024 Exam 2, Q16):** Consider a 4:2 priority encoder with inputs I0 (LSB), I1, I2, I3 (MSB) and outputs A0 (LSB), A1 (MSB). Internally it is made up of basic logic gates. Which of the circuits shown below correctly implements the output A0?
- _(The question provides four circuit diagrams. We need to identify the one for A0.)_
- A. Circuit A (OR gate with I0, I1, I2, I3)
- B. Circuit B (AND gate with I0, NOT I1, NOT I2, NOT I3)
- C. Circuit C (OR gate with I1, and an AND gate of I0 and NOT I1)
- D. Circuit D (OR gate with I3, and an AND gate of I1 and NOT I2, and an AND gate of I0 and NOT I1 and NOT I2)
- E. None of the above
- **Reasoning:**

  1. **Understand Priority Encoding:** A priority encoder outputs the binary code of the highest priority active input. Here, I3 is the highest priority, followed by I2, I1, then I0 [lec2.6Encoders.pdf].

  1. **Determine Output A0 (LSB) logic:** We want A0 to be '1' when the encoded input (0, 1, 2, 3) has a '1' in its LSB position.

  Input 0 (I_0): Encodes 00 ($A_1A_0 = 00$). So, A0=0. (Only if I_3, I_2, I_1 are all 0).

  Input 1 (I_1): Encodes 01 ($A_1A_0 = 01$). So, A0=1. (Only if I_3, I_2 are all 0).

  Input 2 (I_2): Encodes 10 ($A_1A_0 = 10$). So, A0=0. (Only if I_3 is 0).

  Input 3 (I_3): Encodes 11 ($A_1A_0 = 11$). So, A0=1.

9 Section 3c

3c. Encoders and More

(c) Additional Concepts

This section delves into specific digital components and technologies that are crucial for implementing complex logic functions and understanding modern digital system design. These include Programmable Logic Devices (PLDs), Read-Only Memory (ROM), Look-Up Tables (LUTs), Field-Programmable Gate Arrays (FPGAs), Open-Drain Outputs, Tri-State Buffers, and Transmission Gates.

Programmable Logic Devices (PLDs)

- **Definition:** A PLD is an integrated circuit that can be configured by the user to implement specific logic functions.
- **Fundamental Principle:** PLDs are conceptually similar to a large breadboard where a grid of wires (blue and red) can be interconnected using programmable switches called **transmission gates**. These connections are controlled by configuration bits.
- **Logic Elements (LEs):** The internal "chips" within a PLD are not just simple gates but rather configurable logic elements (LEs). An LE can implement a full sum-of-products expression and typically includes a flip-flop.

- **Historical Development of PLDs:**
  - **Programmable Logic Arrays (PLAs) (1970s):** Early PLDs that primarily consisted of 3-4 two-level AND-OR trees. Configuration was achieved by blowing on-chip fuses.
  - **Generic Array Logic (GAL) (1980s):** Featured 8-12 LEs, complete with flip-flops and feedback capabilities. GALs were configured using internal E2PROM, making them non-volatile and rewritable.
  - **Complex Programmable Logic Devices (CPLDs) (1990s):** These devices scaled up significantly, containing dozens to thousands of macrocells (advanced LEs) on a single chip. Like GALs, CPLDs were configured using non-volatile, rewritable E2PROM.
  - **Field-Programmable Gate Arrays (FPGAs) (1990s):** Represent a major leap in complexity, housing hundreds to hundreds of thousands of LEs on a chip. A key distinction is their configuration storage: FPGAs use internal SRAM, which is volatile.

Read-Only Memory (ROM)

- **Functionality:** ROM is primarily used for storing programs and data in digital systems. Crucially, it can also serve as a method for realizing combinational logic circuits.
- **Non-Volatile Storage:** A defining characteristic of ROM is that its contents are preserved even when power is removed.
- **Combinational Circuit Element:** In the context of logic design, a ROM operates as a combinational circuit: given a specific address input, it produces a corresponding data output.
- **Structure:** A common notation for ROM size is $2^n \times b$, where 'n' represents the number of address lines (inputs) and 'b' represents the number of data outputs.
- **Verilog Implementation for Combinational Logic:** In Verilog, ROM can be implemented by defining a memory array (e.g., 'logic [3:0] ROM [7:0]', which represents an 8x4 memory array). The contents can be loaded, for instance, from an external file using '$readmemh'. Data is then accessed by indexing the array with the address, such as 'assign ROM_data = ROM[ROM_addr]'. This approach allows direct mapping of a truth table into memory, effectively implementing a combinational function.

Look-Up Tables (LUTs)

- **Core Component of LEs:** Modern logic elements found in PLDs, especially FPGAs, incorporate components known as Look-Up Tables (LUTs).
- **Structure and Inputs/Outputs:** A typical LUT can have between 4 and 12 inputs and usually produces a single output.
- **Implementation:** LUTs are implemented using multiplexers.
- **Functionality as a Truth Table:** A LUT is the fundamental building block of an FPGA, capable of implementing any logic function of 'N' Boolean variables. Essentially, it functions as a programmable truth table: each possible input combination is pre-programmed to produce either a '0' or '1' output.
- **Configuration in FPGAs:** When an FPGA is configured, the internal bits of the LUTs are loaded with ones or zeros, defining the desired truth table for each logic function.
- **Efficiency Considerations:** While versatile, using a LUT for very simple functions (e.g., a single inverter) can be inefficient, as an entire logic cell (containing the LUT) must be dedicated, and much of its capacity may be "wasted" because a LUT typically has only one output and cannot be partially used for other functions.

Field-Programmable Gate Arrays (FPGAs)

- **Architecture:** FPGAs are characterized by their extensive array of configurable logic cells, which include LUTs and flip-flops, interconnected by a configurable mesh of wires.
- **Configurable Connections:** The interconnections within an FPGA are established using **transmission gates**.
- **Volatile Configuration:** Unlike CPLDs, the programming configuration in FPGAs is stored in SRAM-based memory cells, making it volatile. This means that the FPGA's configuration is lost when power is removed.

- **Initialization/Boot Cycle:** Due to their volatile nature, FPGAs require their programming information to be loaded, typically from an external ROM chip, every time they are powered up. This process is referred to as the "initialization/boot" cycle.
- **Performance and Cost Comparison:**
  - **Speed & Power:** FPGAs are generally about half as fast and consume ten times more power compared to Application-Specific Integrated Circuits (ASICs).
  - **Development Cost:** FPGAs offer a significant advantage in terms of lower cost for testing and development.
  - **Unit Cost:** While initial development is cheaper, the unit cost of FPGAs is higher than that of ASICs. They are often used in low-volume production where the high initial investment of ASICs is not justified.
- **ECE 270 Lab FPGA:** The ECE 270 course specifically uses the Lattice iCE40HX-8K FPGA. This device features 256 pins and contains 7680 Logic Elements. Each of these LEs comprises a 4-bit LUT, an adder carry chain, and a flip-flop.

Open-Drain Outputs

- **Functionality:** In an open-drain General Purpose Input/Output (GPIO) output model, the output is pulled to a logic low (ground) by an internal NMOS transistor.
- **Contrast with PMOS:** It is important to note that the internal PMOS device in an open-drain configuration does *not* pull the output to Vdd (logic 1). This distinguishes it from push-pull output stages.

Tri-State Buffers

- **Functionality:** A tri-state buffer is a type of buffer that can exhibit three output states: logic 0, logic 1, and a high-impedance (Z) state.
- **High-Impedance State:** The buffer's output enters the high-impedance (floating) state when its active-high Enable signal is tied to 0. In this state, the output effectively disconnects from the circuit, allowing other devices to drive the line without contention.

Transmission Gates

- **Role in PLDs:** Transmission gates act as programmable switches that are used to connect wires together within programmable logic devices. They are controlled by configuration bits.
- **Delay Contribution:** These gates contribute to propagation delay, which can lead to timing hazards in circuits.
- **Application in Multiplexers:** Transmission gates are fundamental components in the CMOS implementation of multiplexers, enabling the selection of one input signal to be routed to the output based on select lines.

10 Multiplexers

10.1 Understanding Multiplexers

- **Definition and Function:**
  - A **multiplexer (mux)** is a digital switch that takes multiple input signals and forwards one of them to a single output line.
  - Its selection is controlled by a set of **select lines**.
  - For an $n:1$ multiplexer, where n is the number of data inputs, there are s select lines such that $n = 2^s$.
  - The fundamental motivation for multiplexers is to allow the communication of multiple sources of data over a single transmission line.
  - Multiplexers are implemented using logic gates, and in CMOS technology, they often utilise **transmission gates** as controlled switches.
- **Truth Tables and Logic Expressions:**
  - **2:1 Multiplexer:**

· A 2:1 multiplexer has two data inputs (I_0, I_1) and one select line (S). The output (F) is either I_0 or I_1 depending on S.

· **Truth Table:**

S	I_0	I_1	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Alternatively (simplified truth table) :

S	F
0	I_0
1	I_1

· **Logic Expression (Sum-of-Products):** $F = S' \cdot I_0 + S \cdot I_1$

∗ **4:1 Multiplexer:**

· A 4:1 multiplexer has four data inputs (D_0, D_1, D_2, D_3) and two select lines (S_1, S_0), with S_1 typically being the MSB.

· **Truth Table (Simplified):**

S_1	S_0	F
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

· **Logic Expression (Sum-of-Products):** $F = S_1'S_0'D_0 + S_1'S_0D_1 + S_1S_0'D_2 + S_1S_0D_3$ This expression indicates that the output is the logical OR of each data input, ANDed with the minterm formed by the select lines that correspond to that input's selection.

– **Relationship to Look-Up Tables (LUTs):**

∗ The way multiplexers function, selecting an output based on select line inputs, is analogous to how Look-Up Tables (LUTs) operate within Field-Programmable Gate Arrays (FPGAs).

∗ A LUT is a fundamental building block of an FPGA, capable of implementing any logic function of N Boolean variables by essentially acting as a programmable truth table.

Exam-Style Problem & Explanation

Problem: Choose the correct circuit that implements the 2:1 multiplexer as defined by the functional truth table below:

S	F
0	I0
1	I1

(A)

(B)

(C)

(D)

Solution & Explanation:

Answer: C

Reasoning: The functional truth table for a 2:1 multiplexer is $F = S' \cdot I_0 + S \cdot I_1$. We need to find the circuit that implements this Boolean expression. Let's analyze option C, which corresponds to the correct circuit:

1. The input S is fed directly to the second AND gate ($andA1$) and inverted (S') by the NOT gate to the first AND gate ($andA0$).

1. The input I_0 is connected to the second input of $andA0$. Therefore, the output of $andA0$ is $S' \cdot I_0$.

1. The input I_1 is connected to the second input of $andA1$. Therefore, the output of $andA1$ is $S \cdot I_1$.

1. The outputs of $andA0$ ($S' \cdot I_0$) and $andA1$ ($S \cdot I_1$) are fed into an OR gate (orF).

1. The final output F is $(S' \cdot I_0) + (S \cdot I_1)$.

This matches the Boolean expression for a 2:1 multiplexer. Other options can be eliminated by tracing their logic. For instance, option A uses OR gates, which would produce $S' + I_0$ and $S + I_1$ at the intermediate stage, not the required AND products. Option B and D use NOR gates, which would also not produce the correct sum-of-products form. This circuit implements the standard sum-of-products logic for a multiplexer, using an inverter for S', two AND gates for the product terms, and an OR gate for the sum.

11 Section 4b

(b) Designing Multiplexers

- **Designing a Multiplexer using Basic Logic Gates:**
  - **Derivation from Logic Expression:** The design of a multiplexer using basic logic gates directly follows from its Boolean expression [**?**, **?**].
    - For a $2^s : 1$ multiplexer with s select lines, the output F is a sum-of-products (SoP) expression, where each product term consists of a unique minterm of the select lines ANDed with the corresponding data input line [**?**, **?**].
    - **2:1 Multiplexer Example:**
    - Boolean Expression: $F = S' \cdot I_0 + S \cdot I_1$ [**?**, **?**].
    - This expression translates directly into an AND-OR circuit: two AND gates (one for $S' \cdot I_0$, one for $S \cdot I_1$) feeding into a single OR gate [**?**, **?**]. An inverter is needed for S'.
    - **Circuit Diagram for 2:1 Mux (AND-OR Implementation):**
    - **4:1 Multiplexer Example:**
    - Boolean Expression: $F = S_1'S_0'D_0 + S_1'S_0D_1 + S_1S_0'D_2 + S_1S_0D_3$ [Derived from 4(a) definition].
    - This requires two inverters (for S_1' and S_0'), four 3-input AND gates, and one 4-input OR gate.
  - **Implementation with Primitive Gates in Verilog (Structural Model):** This approach directly translates the gate-level design into Verilog code by instantiating primitive gates (e.g., 'and', 'or', 'not') and specifying their connections [**?**, **?**].
- **Cascading Multiple Multiplexers to Build Larger Multiplexer Systems (Mux-Trees):**

* **Concept:** Larger multiplexers (e.g., an 8:1 mux) can be constructed by cascading smaller multiplexers (e.g., 2:1 muxes or 4:1 muxes) [**?**, **?**]. This forms a "mux-tree" structure.

* **Underlying Principle - Shannon's Expansion:** This method is based on Shannon's Expansion theorem, which allows any Boolean function $f(w_1, w_2, \ldots, w_n)$ to be expanded around one or more variables. For a single variable w_1, the expansion is $f(w_1, w_2, \ldots, w_n) = w_1' \cdot f(0, w_2, \ldots, w_n) + w_1 \cdot f(1, w_2, \ldots, w_n)$ [**?**, **?**].

    · This directly maps to a 2:1 multiplexer where w_1 is the select line, and the data inputs are the "cofactors" $f(0, w_2, \ldots, w_n)$ and $f(1, w_2, \ldots, w_n)$ [**?**, **?**]. These cofactors themselves can be implemented with smaller logic circuits or further multiplexers.

* **Building a 4:1 Mux from 2:1 Muxes:**

    · A 4:1 mux can be built using three 2:1 muxes.

    · The two MSBs (S_1, S_0) serve as select lines. Let S_1 be the MSB and S_0 the LSB.

    · Two 2:1 muxes are used in the first stage, with S_0 as their common select line.

    · The first mux selects between D_0 and D_1. Its output is $F_0 = S_0'D_0 + S_0D_1$.

    · The second mux selects between D_2 and D_3. Its output is $F_1 = S_0'D_2 + S_0D_3$.

    · A third 2:1 mux takes F_0 and F_1 as its data inputs $(I_0 = F_0, I_1 = F_1)$ and uses S_1 as its select line.

    · The final output is $F = S_1'F_0 + S_1F_1 = S_1'(S_0'D_0 + S_0D_1) + S_1(S_0'D_2 + S_0D_3)$, which is the correct 4:1 mux expression.

    · **Mux-Tree Structure (4:1 from 2:1s):**

* **General Logic Function Implementation:** Any n-variable Boolean function can be implemented using a $2^{n-k} : 1$ multiplexer by connecting the first k variables to the select lines, and deriving the logic for the data inputs (D_i) based on the remaining $n - k$ variables [**?**, **?**]. The values connected to the data inputs can be 0, 1, or one of the remaining variables (or its complement), or a combination of them.

Exam-Style Problem & Explanation

Problem: What is the value of output F when ABC = 010? [**?**, **?**] Given the circuit below, which implements a logic function using two 4:1 Muxes and one 2:1 Mux:

(A) 1 (B) D' (C) 0 (D) D (E) None of the others

Solution & Explanation:

Answer: D'

Reasoning: We need to determine the output F when inputs A, B, C are '010'. Let's trace the signal flow through the multiplexers:

1. **Input to Muxes:**

- $A = 0$
- $B = 1$
- $C = 0$
- D is an independent input, so D or D' will be passed through.

1. **Evaluate the Top 4:1 Mux (Input Mux 1):**

- Select lines are $S_1 = B$ and $S_0 = C$.
- When $B = 1$ and $C = 0$, the select lines are $S_1, S_0 = 1, 0$, which selects input I_2.
- The input I_2 to this mux is D'.

– Therefore, the output of the top 4:1 Mux is D'.

1. **Evaluate the Bottom 4:1 Mux (Input Mux 2):**

– Select lines are $S_1 = C$ and $S_0 = B$. (Note: the select line order is different from the top mux's select line order in terms of which input variable maps to which select line, as depicted in the diagram).

– When $C = 0$ and $B = 1$, the select lines are $S_1, S_0 = 0, 1$, which selects input I_1.

– The input I_1 to this mux is D.

– Therefore, the output of the bottom 4:1 Mux is D.

1. **Evaluate the Final 2:1 Mux:**

– The final 2:1 Mux has the output of the top 4:1 Mux as its I_0 input and the output of the bottom 4:1 Mux as its I_1 input.

– So, $I_0 = D'$ and $I_1 = D$.

– The select line for the final 2:1 Mux is A.

– When $A = 0$, the 2:1 Mux selects its I_0 input.

– Therefore, the final output $F = I_0 = D'$.

The value of output F is D'. This corresponds to option (B).

12 Section 4c

(c) Using Multiplexers to Implement Logic Functions

– **Implementing Arbitrary Logic Functions using Multiplexers:**

* **Core Principle:** A $2^s : 1$ multiplexer can implement any Boolean function of $s + k$ variables. The s variables are connected to the select lines, and the remaining k variables (or their complements, or constants 0/1) are connected to the data inputs [lec2.3Mux.pdf, p. 24; Supplement 2F, p. 109]. This is the fundamental concept behind Look-Up Tables (LUTs) used in FPGAs [Supplement 2F, p. 109; lec2.4ROM_LUT.pdf, p. 21].

* **Methodology for an n-variable function using a $2^{n-1} : 1$ Mux:**

1. **Assign Select Lines:** Choose $n - 1$ of the function's variables to be the select lines $(S_{n-2}, \ldots, S_0)$ of the multiplexer. The remaining variable will be used for the data inputs.

2. **Construct a Modified Truth Table:** Create a truth table for the n-variable function, but group rows based on the combinations of the select variables.

3. **Determine Data Input Values:** For each unique combination of the select variables (which corresponds to a specific data input D_i of the multiplexer), observe the output of the function (F) as the remaining (unselected) variable changes. The data input D_i will then be one of four possibilities:

· '0' (if F is always 0 for that select combination).

· '1' (if F is always 1 for that select combination).

· The unselected variable (e.g., X) (if F directly follows the value of X).

· The complement of the unselected variable (e.g., X') (if F directly follows the complement of X).

* **Example: Implementing $F(X, Y, Z) = X \cdot Z + X' \cdot (Y \oplus Z)$ using an 8:1 Mux.**

· In this case, all three variables (X, Y, Z) are used as select lines $(S_2 = X, S_1 = Y, S_0 = Z)$. Each data input D_i corresponds to a minterm of X, Y, Z and is simply the desired output value (0 or 1) for that minterm [Supplement 2F, pp. 110-111].

· **Truth Table and Data Input Mapping:**

X	Y	Z	F	Minterm Index	D_i Connection
0	0	0	0	0	$D_0 = 0$
0	0	1	1	1	$D_1 = 1$
0	1	0	1	2	$D_2 = 1$
0	1	1	0	3	$D_3 = 0$
1	0	0	0	4	$D_4 = 0$
1	0	1	1	5	$D_5 = 1$
1	1	0	0	6	$D_6 = 0$
1	1	1	1	7	$D_7 = 1$

· **Equivalent Boolean Expression (for 8:1 Mux using $S_2 = X, S_1 = Y, S_0 = Z$):** $F = m_0 \cdot D_0 + m_1 \cdot D_1 + m_2 \cdot D_2 + m_3 \cdot D_3 + m_4 \cdot D_4 + m_5 \cdot D_5 + m_6 \cdot D_6 + m_7 \cdot D_7$ [Derived from Supplement 2F, p. 107]. Substituting the determined D_i values: $F = (X'Y'Z') \cdot 0 + (X'Y'Z) \cdot 1 + (X'YZ') \cdot 1 + (X'YZ) \cdot 0 + (XY'Z') \cdot 0 + (XY'Z) \cdot 1 + (XYZ') \cdot 0 + (XYZ) \cdot 1$ $F = X'Y'Z + X'YZ' + XY'Z + XYZ$ (This matches the minterms shown in Supplement 2F, p. 111, for the expanded SoP form).

· **Circuit Diagram (Conceptual for 8:1 Mux implementation):**

∗ **Example: Implementing $F(X, Y, Z) = X + Z'$ using an 8:1 Mux:**

· This function can be mapped directly to an 8:1 mux by setting the data inputs according to the function's truth table values [Supplement 2G, p. 120, 122]. The 'map' assignment in Verilog 'assign map = 8'b11110101;' directly represents the truth table output from minterm 7 down to 0, which corresponds to $F(111) = 1, F(110) = 1, F(101) = 1, F(100) = 1, F(011) = 0, F(010) = 1, F(001) = 0, F(000) = 1$. This is effectively using the multiplexer as a lookup table [Supplement 2G, p. 124].

– **Constructing a MUX-Tree (Cascading Multiplexers):**

∗ **Purpose:** To build a larger multiplexer from smaller multiplexer components [lec2.3Mux.pdf, p. 18]. This is useful when larger integrated muxes are not available or when a modular design is desired.

∗ **Example: Building an 8:1 Multiplexer from 2:1 Multiplexers.**

· An 8:1 multiplexer requires 3 select lines (e.g., S_2, S_1, S_0).

· **Stage 1:** Four 2:1 multiplexers are used. Each takes S_0 as its select input.

· The first 2:1 mux selects between D_0 and D_1.

· The second 2:1 mux selects between D_2 and D_3.

· The third 2:1 mux selects between D_4 and D_5.

· The fourth 2:1 mux selects between D_6 and D_7.

· **Stage 2:** Two more 2:1 multiplexers are used. Each takes S_1 as its select input.

· The first mux selects between the output of (D0/D1 mux) and (D2/D3 mux).

· The second mux selects between the output of (D4/D5 mux) and (D6/D7 mux).

· **Stage 3:** One final 2:1 multiplexer is used. It takes S_2 as its select input.

· This mux selects between the outputs of the two stage-2 multiplexers to produce the final output F.

· **MUX-Tree Diagram (8:1 from 2:1s):**

∗ **Verilog Implementation (Dataflow):** Can be very compact using direct indexing of data inputs 'D[S]' for a single multiplexer [Supplement 2F, p. 114], or through nested ternary operators to represent the mux-tree logic [Supplement 2F, p. 115]. The 'casez' statement in behavioral Verilog also provides a way to describe multiplexer behavior, often leading to synthesized muxes [lec2.3Mux.pdf, p. 32; lec2.6Encoders.pdf, p. 22].

Exam-Style Problem & Explanation

Problem: What logic expression does the following MUX circuit implement? [Exam2SP23-Key.pdf, Q11; Exam2SP24.pdf, Q11]

—

13 Section 5a

Here is the study guide section for ECE 270 Exam 2, focusing on Verilog Designs and Modeling Approaches:

13.1 Verilog Designs (25%)

13.2 Verilog Modeling Approaches

Verilog designs can be implemented using three primary modeling approaches: Structural, Dataflow, and Behavioral. These approaches offer different levels of abstraction for describing digital logic.

- **Structural Model**
  - **Definition:** Structural modeling describes a circuit by instantiating fundamental building blocks (like primitive gates or other user-defined modules) and manually connecting their inputs and outputs. It is akin to drawing a schematic diagram.
  - **Key Features:**
    - Relies on instantiating every module and connecting its inputs and outputs.
    - Module instantiation uses the syntax ‘module_name instance_name (signal_list);‘.
    - SystemVerilog does not require instance names for primitive gates (e.g., ‘and (output, input1, input2);‘).
    - Connections can be made **by order** (listing signals in the order of the module’s port declaration) or **by name** (explicitly mapping signals to port names using ‘.portName(signal);‘ syntax), with connection by name generally preferred for clarity and robustness.
    - Hierarchical designs are built by instantiating smaller modules within larger ones.
  - **Example (Structural Model with Primitive Gates):**
  - In SystemVerilog, primitive gate instances (like ‘and‘, ‘nand‘, ‘or‘, ‘xor‘) do not require explicit instance names. For example, ‘and (AB, A, B);‘ would be valid.
- **Dataflow Model**
  - **Definition:** Dataflow modeling describes a circuit’s functionality using continuous assignments (‘assign‘ statements) and bitwise operators. It expresses the logical relationships and flow of data between signals.
  - **Key Features:**
    - Uses ‘assign‘ statements to continuously connect the value of an expression to a signal.
    - Bitwise operators (‘∼‘ for NOT, ‘&‘ for AND, ‘|‘ for OR, ‘^‘ for XOR, ‘∼ ^‘ or ‘^ ∼‘ for XNOR) are used to specify operations on signals.
    - Operators act on all bits of a bus (vector) in parallel.
    - The concatenation operator ‘{}‘ can be used to group multiple signals into a bus.
  - **Example (Dataflow Model):**

* This is a concise way to model a multiplexer by selecting an element from an array ‘D‘ based on the select lines ‘S‘.

– **Behavioral Model**

* **Definition:** Behavioral modeling describes a circuit’s behavior using procedural blocks, primarily ‘always‘ blocks, which can change their behavior based on certain conditions or events. It allows for a higher level of abstraction, often resembling software programming constructs.

* **Key Features:**

  · Logic is implemented inside ‘always‘ blocks, specifically ‘always_comb‘ for combinational logic.

  · Uses procedural statements like ‘if-else‘ and ‘case‘ or ‘casez‘ statements to define logic.

  · It is possible to create non-synthesizable HDL code using behavioral models, meaning code that can simulate a digital system but not be realised as hardware.

* **Example (Behavioral Model):**

* This module ‘mux2‘ implements a 2:1 multiplexer using a behavioral model, where the output ‘out‘ is assigned ‘i1‘ if ‘s‘ is true, and ‘i0‘ otherwise.

Exam Style Problem:

Question: Given a SystemVerilog module that implements a 2:1 Multiplexer as shown below, choose the correct type of Verilog modeling that is shown in the module ‘mux2‘:

```
module mux2(input logic i0, i1, s, output logic out);
  always\_comb begin
    if(s)
      out = i1;
    else
      out = i0;
  end
endmodule
```

A. The module ‘mux2‘ is in behavioral modeling B. The module ‘mux2‘ is in dataflow modeling C. The module ‘mux2‘ is in structural modeling D. The module ‘mux2‘ is in functional programming modeling E. None of the above

Explanation:

The module ‘mux2‘ uses an ‘always_comb‘ block and ‘if-else‘ statements to define its logic. This approach, where logic is implemented inside procedural blocks that can change their behavior based on certain conditions, is characteristic of **behavioral modeling**. Dataflow modeling typically uses ‘assign‘ statements with operators, while structural modeling uses instantiated gates or modules.

Answer: A. The module ‘mux2‘ is in behavioral modeling.

—