Unit 8: Multiplexers

CSE 220: System Fundamental I

Stony Brook University

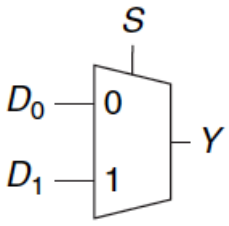
Joydeep Mitra

Multiplexer Overview

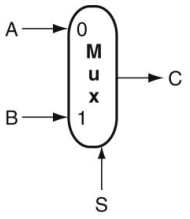
- What is a multiplexer?
  - A multiplexer (also MUX) is a larger building block used to choose a data input from several data inputs based on one or more select/control signals
- Why use a multiplexer?
  - To build complex combinational circuits (an application of abstraction)
- Example: A 2:1 MUX with 2 data inputs, a select signal, and an output signal
  - The MUX chooses between D_0 and D_1 based on the select/control signal S

S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Truth table of 2:1 MUX



Schematic of 2:1 MUX



Alternative schematic; "MUX" label is optional

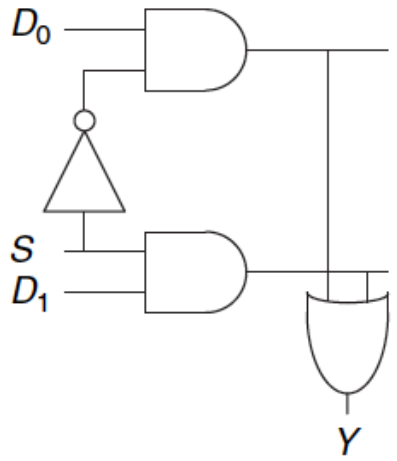
Edit: *In lecture, I mistakenly say that 1 stands for select signal. The 1 stands for the output signal and not the select signal.*

Implementing A Multiplexer

- A multiplexer can be easily implemented with sum-of-products logic

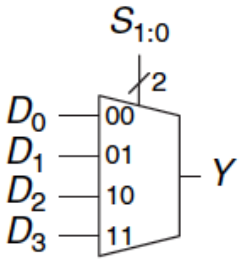
Y / S \ $D_{1:0}$	00	01	11	10
0	0	1	1	0
1	0	0	1	1

$$Y = D_0\bar{S} + D_1S$$

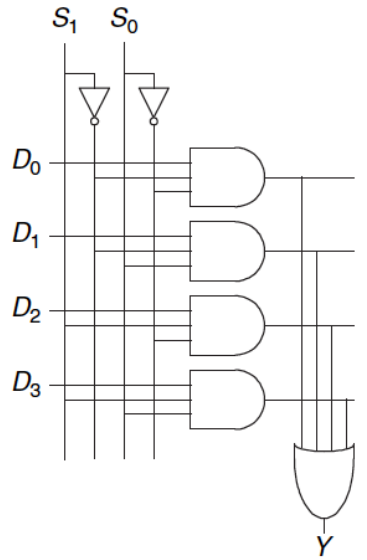


Wider Multiplexers

- A 4:1 MUX has 4 data inputs, 2 control/select signals, and 1 output
- 4:1 MUX can also be implemented using SOP logic
- Similarly, 8:1 MUX will have 8 data inputs, 3 control signals, and 1 output
- In the same vein, 16:1 MUX will have 16 data inputs, 4 control signals, and 1 output
- In general, an N:1 MUX will have N data inputs, $\log_2 N$ control signals, and 1 output



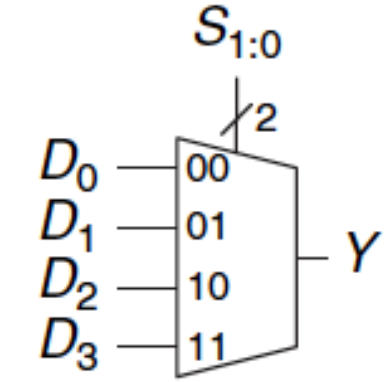
Schematic of a 4:1 MUX

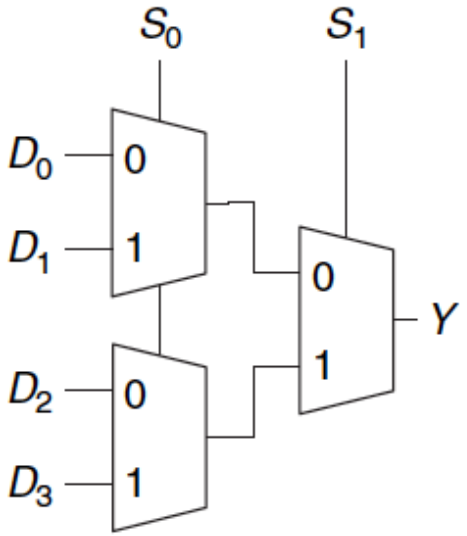


A 4:1 MUX implementation with SOP logic

Wider Multiplexers

- A wider multiplexer can be implemented with narrower multiplexers. How?
  - Create a hierarchy of the available narrower MUXes
  - E.g., 4:1 MUX with two layers of three 2:1 MUXes
- Which is better for implementing MUX: narrower MUXes or SOP logic?
  - Depends on the target technology
  - Availability vs scalability tradeoffs are involved





Multiplexers As Minimal Complete Sets

- A **minimal complete set** is a set with the least number of unique gates that can be used to express any Boolean function
  - {AND, NOT}
  - {NAND} and {NOR}
  - {AND, NOT, OR} is complete but not minimal. Why?
    - For {AND, NOT}, OR is redundant; for {NOT, OR} AND is redundant
- *Multiplexers are a minimal complete set* as they can be used to express any Boolean function
- In general, a 2^N-input multiplexer can be programmed to perform any N-input logic function

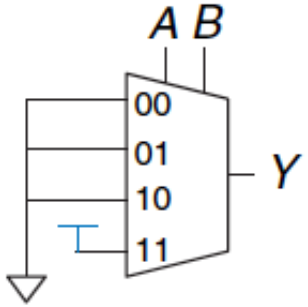
Edit: *In lecture, I mistakenly stated that NAND and NOR gates are not minimal complete sets. They are. You can express any Boolean function in terms NAND or NOR gates.*

AND Gate With Multiplexer

- `Y(A,B) = AB` is a logic function with 2 inputs
- We can implement this with a 4:1 MUX
  - A and B are control signals
  - 4 inputs mapped to either 0 or 1 based on Y's value in truth table
  - 0 is indicated by ground (triangle), and 1 is indicated by high voltage (flat bar)
- Can we do better?

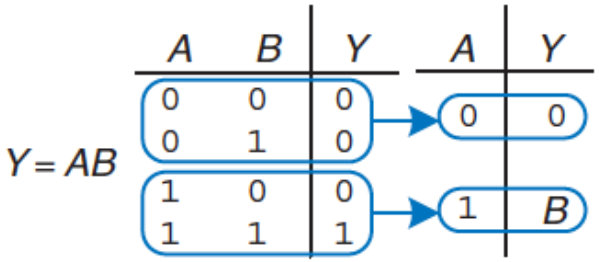
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

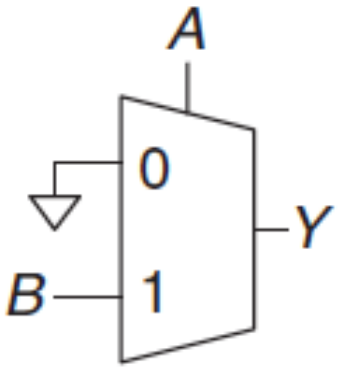
$Y = AB$



AND Gate With Multiplexer

- A MUX implementation of any logic function with N inputs can be cut into half by selecting one of the inputs to be the control signal
  - 2^N:1 MUX becomes 2^{N-1}:1
- General strategy:
  - Start with the truth table
  - Combine pairs of rows to eliminate the right-most input variable by expressing output in terms of this variable
- E.g., `Y(A,B) = AB`
  - When A = 0, Y = 0 regardless
  - When A = 1, Y = 0 if B = 0, and Y = 1 if B = 1
    - So, Y = B when A = 1





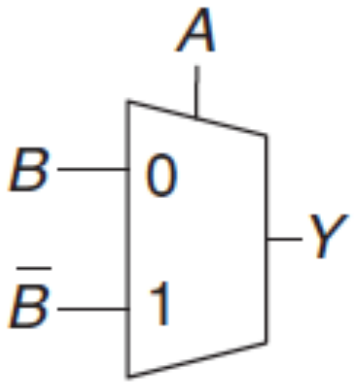
XOR Gate With Multiplexer

- An XOR gate can be easily implemented using a 2:1 MUX

$Y = A \oplus B$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

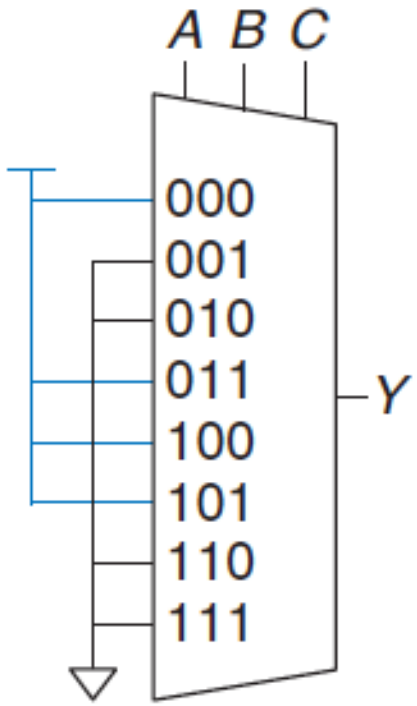
A	Y
0	B
1	$\overline{B}$



Example #1

- Implement the function Y = AB′ + B′C′ + A′BC. We only have 8:1 multiplexers

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

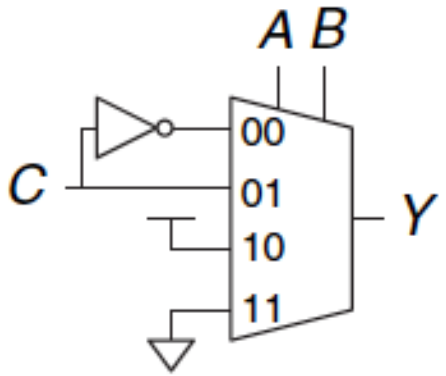


Example #2

- Implement the function Y = AB′ + B′C′ + A′BC. We only have 4:1 multiplexers
- We reduce the truth table to 4 rows by basing the output on C

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

A	B	Y
0	0	$\overline{C}$
0	1	C
1	0	1
1	1	0

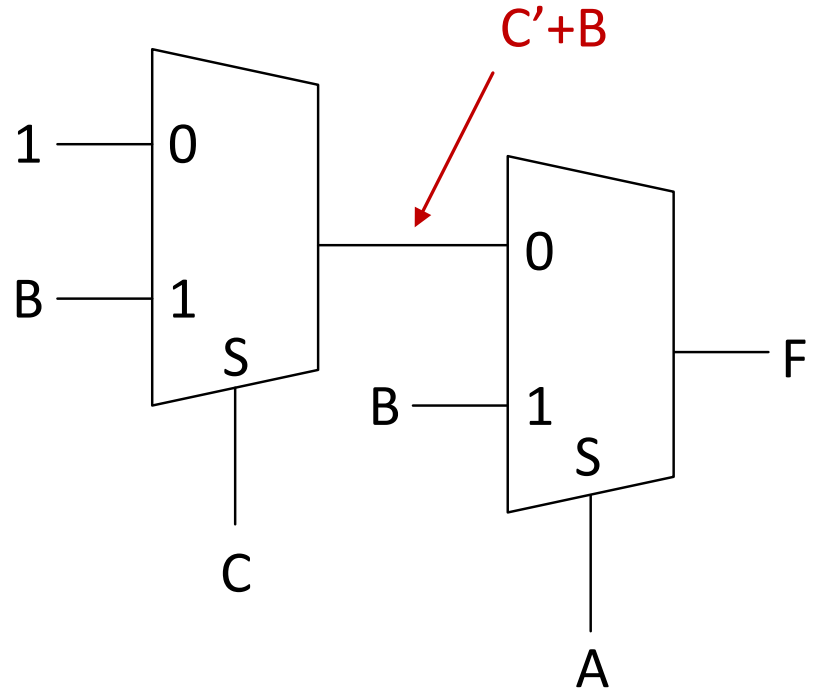


Multiplexers From Boolean Equations

- In addition to the truth table approach, we can use Boolean algebra to design our circuits with muxes
- We need a Boolean equation in SOP form
- If not, then use Boolean algebra to re-write to SOP form (similar to K-maps)
- Key insight
  - An equation of the form `PA + P'B` can be converted to a MUX where P is the selector/control variable, and A and B are the input signals
  - A and B may be the output of multiplexers as well

Example #3

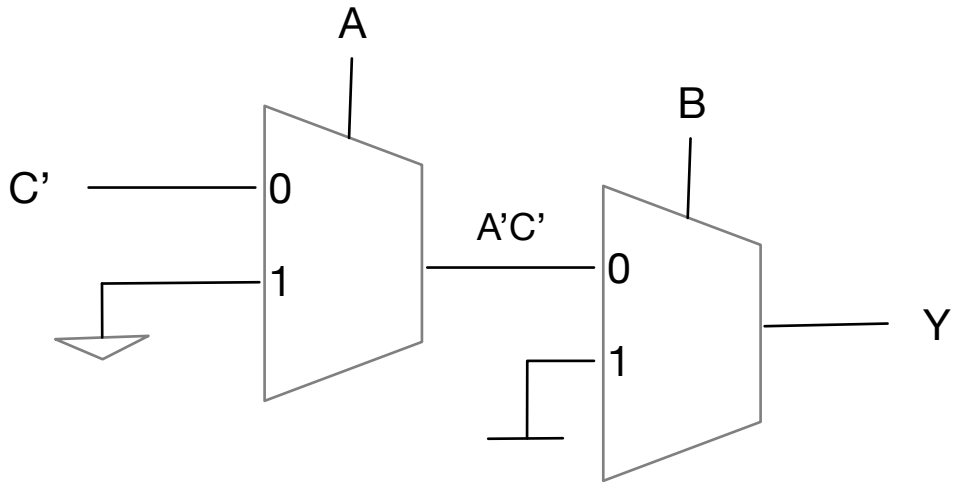
- Implement the function F = ABC′ + A′BC′ + A′B′C′ + BC using only 2:1 muxes
- Let's convert to SOP form
  - F = ABC′ + A′BC′ + A′B′C′ + BC
    = ABC′ + A′BC′ + A′B′C′ + (A+A′)BC
    = ABC′ + A′BC′ + A′B′C′ + ABC + A′BC
- We pick A as the control variable
  - F = ABC′ + A′BC′ + A′B′C′ + ABC + A′BC
    = A(BC′ + BC) + A′(BC′ + B′C′ + BC)
    = A(B) + A′(C′1 + BC)
- We will need two 2:1 muxes
  - With control variable A for main expression in F
  - With control variable C for the expression (C′1 + BC)



Example #4

- Let's try the same example with selector/control signal B
  - F = ABC′ + A′BC′ + A′B′C′ + ABC + A′BC

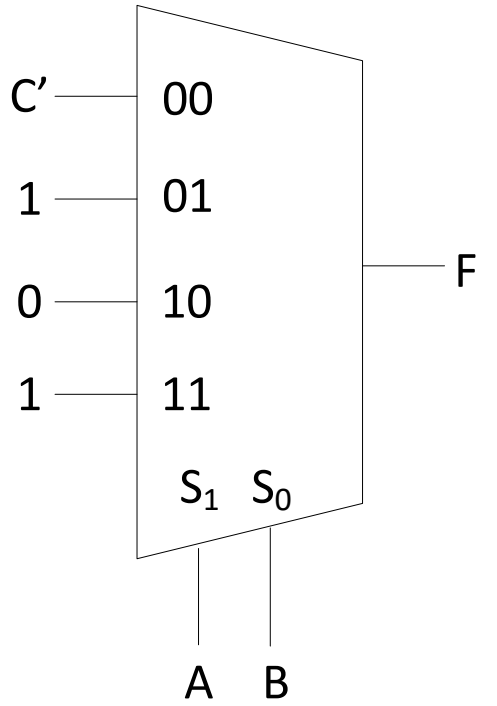
    = B(AC′ + A′C′ + AC + A′C) + A′B′C′
    = B(C′ + C) + B′(A′C′)
    = B(1) + B′A′C′ + B′A0
    = B(1) + B′(A′C′ + A0)



- We will need two 2:1 muxes
  - With control variable B for main expression in F
  - With control variable A for the expression (A′C′ + A0)

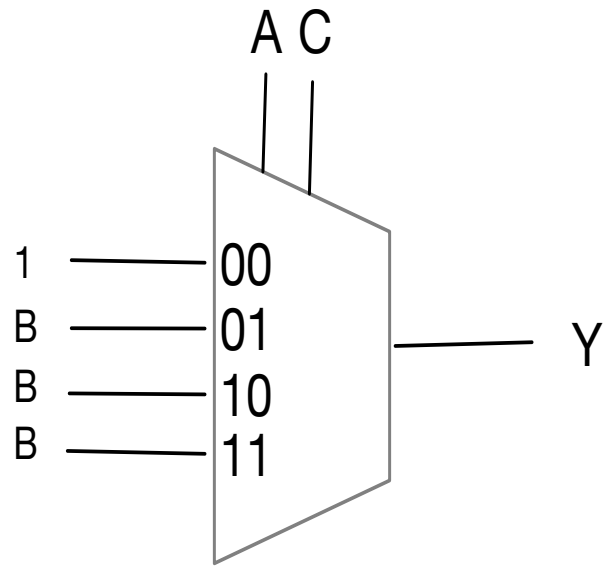
Example #5

- Implement the function F = ABC′ + A′BC′ + A′B′C′ + BC using only 4:1 muxes
- The SOP form
  - F = ABC′ + A′BC′ + A′B′C′ + ABC + A′BC
- 4:1 MUX, so we need 2 control/selector variables. We pick AB
  - F = ABC′ + A′BC′ + A′B′C′ + ABC + A′BC
    = AB(C + C′) + A′B(C′ + C) + A′B′C
    = AB1 + A′B1 + A′B′C
    = AB1 + A′B1 + A′B′C′ + AB′0



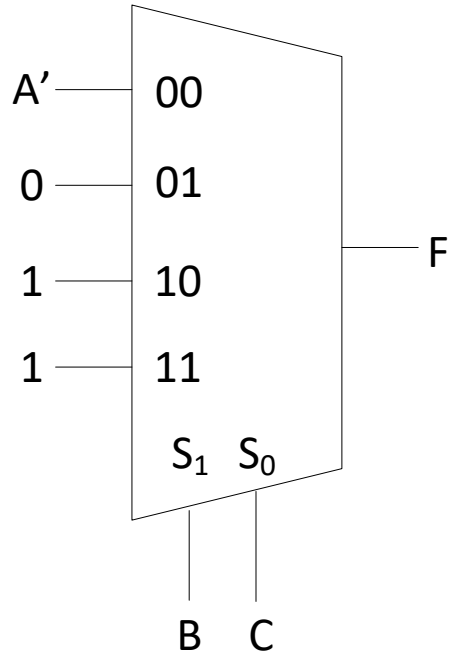
Example #6

- Implement the function F = ABC′ + A′BC′ + A′B′C′ + BC using only 4:1 muxes
- The SOP form
  - F = ABC′ + A′BC′ + A′B′C′ + ABC + A′BC
- 4:1 MUX, so we need 2 control/selector variables. We pick AC this time
  - F = ABC′ + A′BC′ + A′B′C′ + ABC + A′BC
    = AC′(B) + A′C′(B+B′) + AC(B) + A′C (B)
    = AC′(B) + A′C′(1) + AC(B) + A′C (B)



Example #7

- Implement the function F = ABC′ + A′BC′ + A′B′C′ + BC using only 4:1 muxes
- The SOP form
  - F = ABC′ + A′BC′ + A′B′C′ + ABC + A′BC
- 4:1 MUX, so we need 2 control/selector variables. We pick BC this time
  - F = ABC′ + A′BC′ + A′B′C′ + ABC + A′BC
    = BC′(1) + B′C′(A′) + BC(1)
    = BC′(1) + B′C′(A′) + BC(1) + B′C(0)



Points To Note

- Selecting different selector/control variables will result in different circuits
- The circuits may use different gates
- Which circuit to pick will ultimately depend on the target technology, the availability and cost of the gates involved
- As practice try implementing the following gates with MUXes
  - NOT
  - OR
  - NAND
  - NOR
  - XNOR