IE1204 Digital Design:

CMOS Implementation of Boolean Logic

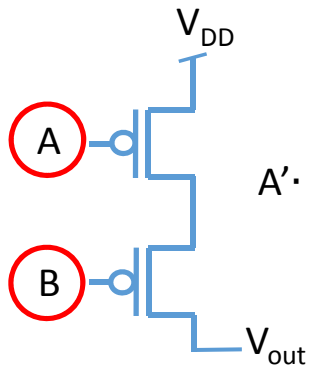
Ahmed Hemani

KTH/ICT/ES

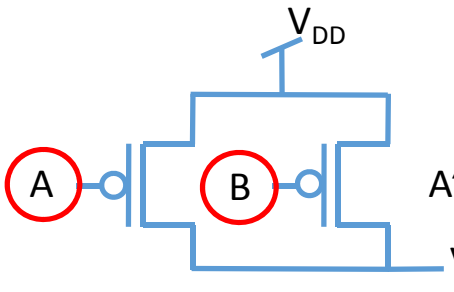
hemani@kth.se

The Pull Up Primitives

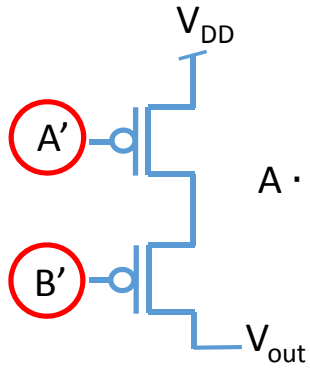
Careful: The Pull Up Primitives invert the literals when used as inputs of the PMOS transistors
Why ? Because the PMOS transistor has a buit in inversion.



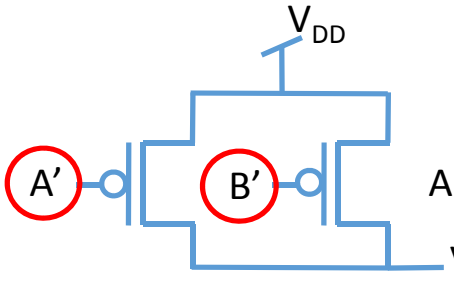
$A' \cdot B'$ → The Serial Pull Up Primitive



$A' + B'$ → The Parallel Pull Up Primitive

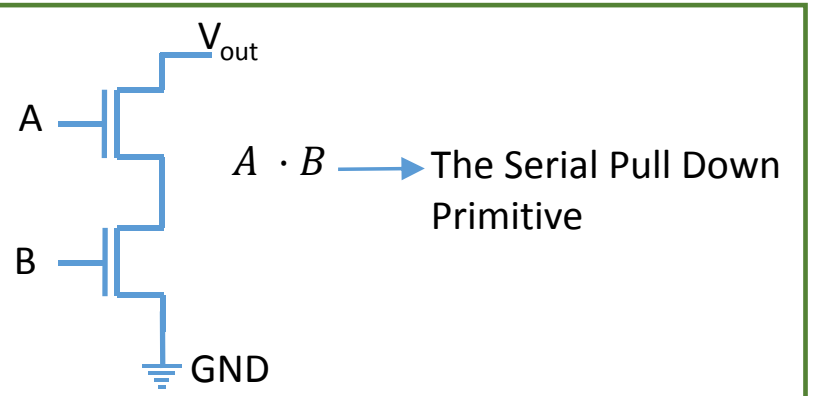


$A \cdot B$ → The Serial Pull Up Primitive

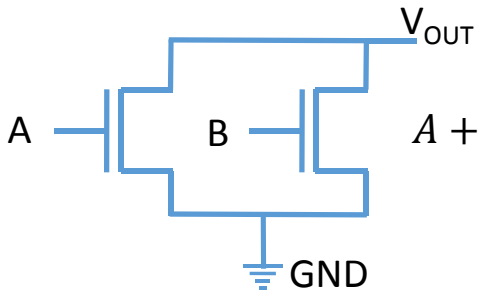


$A + B$ → The Parallel Pull Up Primitive

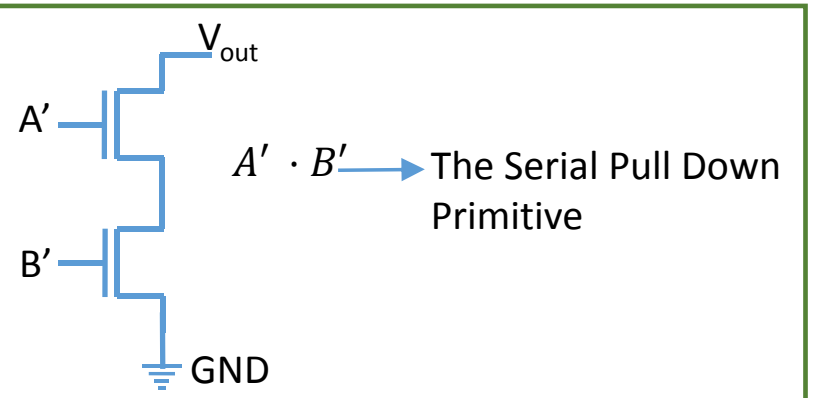
The Pull Down Primitives



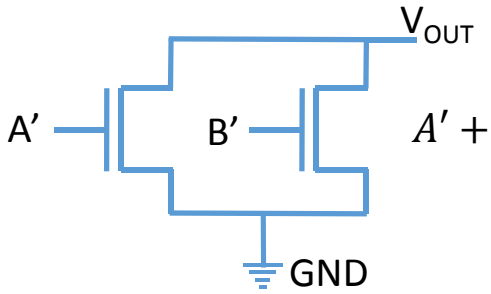
$A \cdot B$ → The Serial Pull Down Primitive



$A + B$ → The Parallel Pull Down Primitive



$A' \cdot B'$ → The Serial Pull Down Primitive



$A' + B'$ → The Parallel Pull Down Primitive

Recipe for implementing arbitrary Boolean function in CMOS

Step 1: Simplify the function – ***f*** , so that it is composed of ONLY the serial and parallel Pull Up and Pull Down primitive $A + B$, $A \cdot B$, $A' + B'$, $A' \cdot B'$,
Note that it is not allowed to complement products (NANDs) or complement sums (NORs). Though inversion of literals is OK. This restriction is because we want the function *f* to be expressed in terms of our Pull Up and Pull Down primitives that allows only the above four forms.

Let us call this simplified function as ***fs****. Note that* ***f*** *and* ***fs*** *are functionally equivalent. They have the same truth table.*

We create two versions of ***fs****:* fs_{ON} *to implement the ON set and* fs_{OFF} *to implement the OFF set*

Step 2: fs_{ON} is the same as ***fs***. Just **remember to invert the literals**. This is done because fs_{ON} is implemented by Pull Up PMOS network. And PMOS has built in inversion of the literals.

Step 3*:* fs_{OFF} *is created by inverting the fs and again ensuring that it is in terms of the primitives. This is done because by definition* fs_{OFF} *implements the complement of* ***f*** *– the OFF set.*

NAND Gate derived by the Recipe

$$f = (A \cdot B)'$$

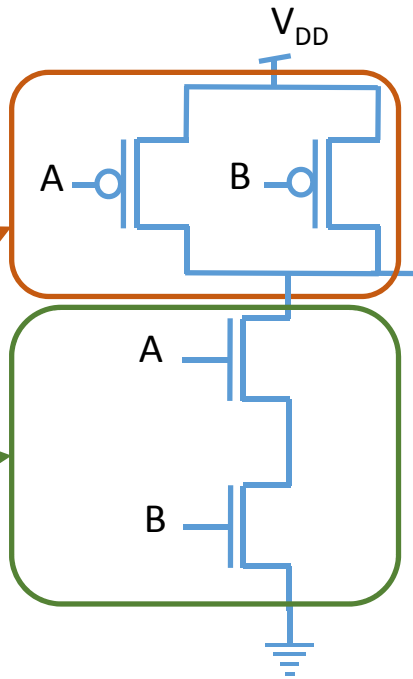
Step 1: Transform *f* to be in terms of the primitives.

$f_s = A' + B'$

Step 2: Create fs_{ON} by inverting the literal when used as inputs to the gates of PMOS Pull Up Network

Step 3: Create fs_{OFF} by inverting *f* and again simplifying it in terms of the primitives. fs_{OFF} is implemented by the Pull Down Network composed of NMOS transistors

$fs_{OFF} = f' = A \cdot B$



This Pull Up Network implements $fs_{ON} = A' + B'$

A NAND B

This Pull Down Network implements $fs_{OFF} = A \cdot B$

NOR Gate derived by the Recipe

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

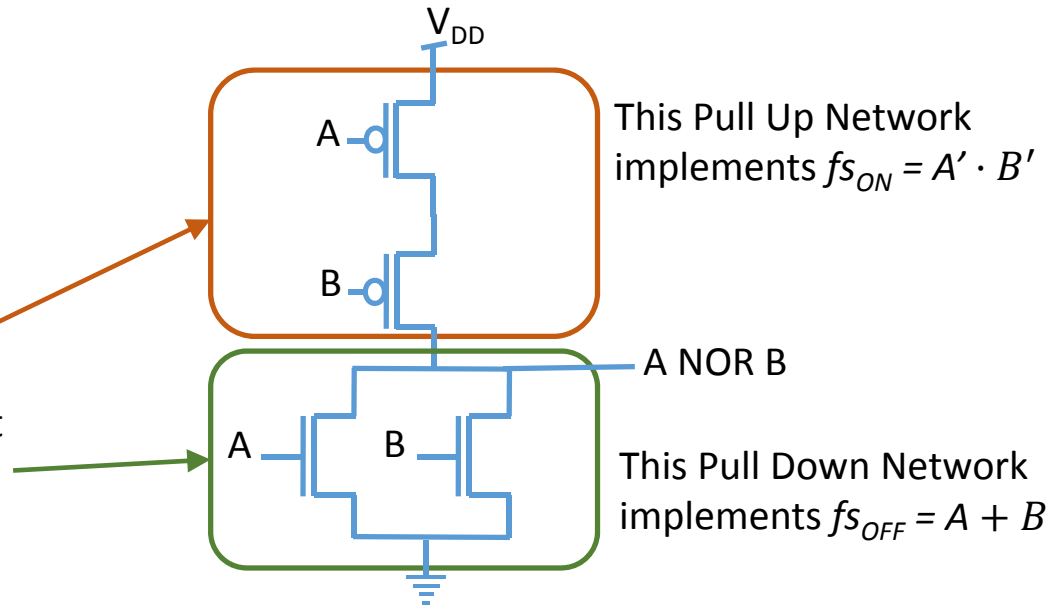
Step 1: Transform *f* to be in terms of the primitives.

$$f_s = A' \cdot B'$$

Step 2: Create fs_{ON} by inverting the literal when used as inputs to the gates of PMOS Pull Up Network

Step 3: Create fs_{OFF} by inverting *f* and again simplifying it in terms of the primitives.

$fs_{OFF} = f' = A + B$



Recipe applied to arbitrary boolean function

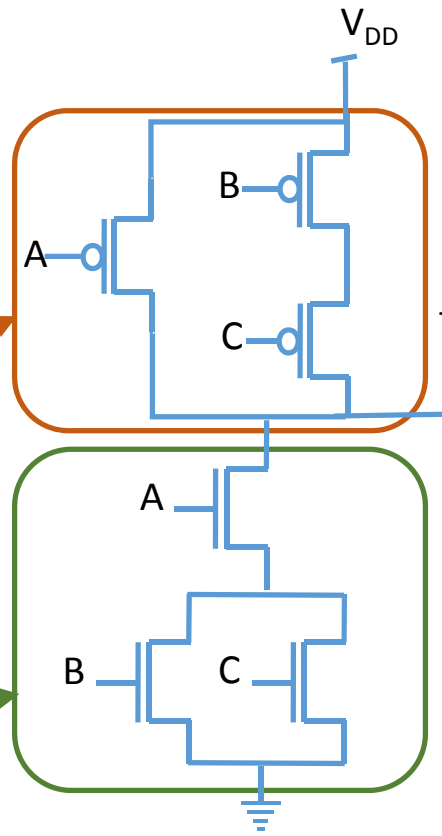
$f = [A \cdot (B + C)]'$

Step 1: Transform *f* to be in terms of the primitives.

$f_s = A' + (B' \cdot C')$

Step 2: Create fs_{ON} by inverting the literal when used as inputs to the gates of PMOS Pull Up Network
Notice the *fs* has two primitives. One is a serial primitive *B·C* and the other is a parallel primitive combining *B·C* and *A*

Step 3: Create fs_{OFF} by inverting *f* and again simplifying it in terms of the Pull Down NMOS primitives.
$fs_{OFF} = f = A \cdot (B + C)$



This Pull Up Network implements $fs_{ON} = A' + (B' \cdot C')$

A NOR B

This Pull Down Network implements $fs_{OFF} = A \cdot (B + C)$

Recipe applied to XOR Gate

$$f = a' \cdot b + a \cdot b'$$

Step 1: Transform *f* to be in terms of the primitives. The XOR gate function is already in form of primitives

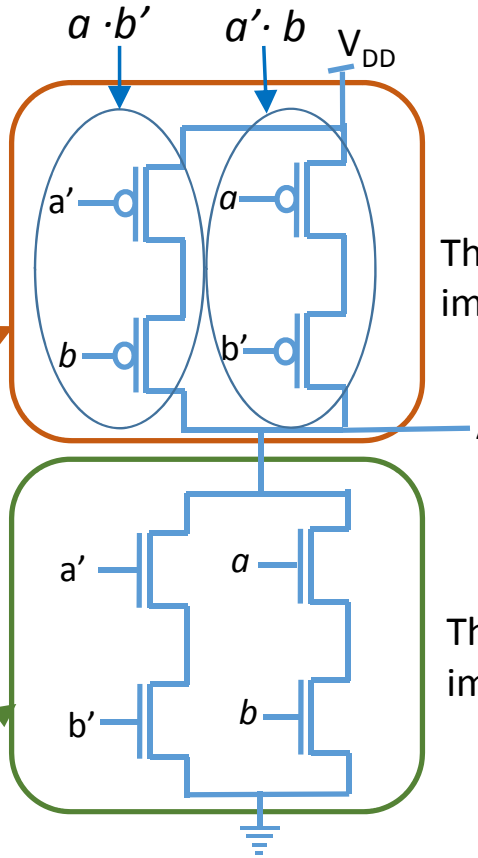
$f_s = f$

Step 2: Create fs_{ON} by inverting the literal when used as inputs to the gates of PMOS Pull Up Network.

Notice the *fs* has three primitives. Two are serial primitives a'·*b* and *a*·*b'* the third is a parallel primitive combining the two serial primitives.

Step 3: Create fs_{OFF} by inverting *f* and again simplifying it in terms of the Pull Down NMOS primitives.

$fs_{OFF} = f' = a' \cdot b' + a \cdot b$



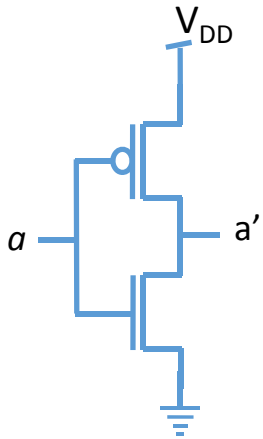
This Pull Up Network implements $fs_{ON} = a' \cdot b + a \cdot b'$

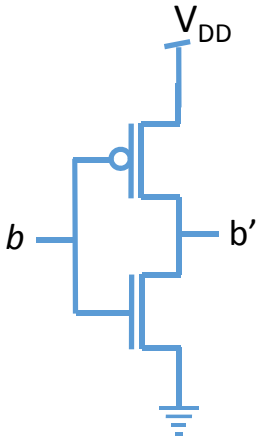
A XOR B

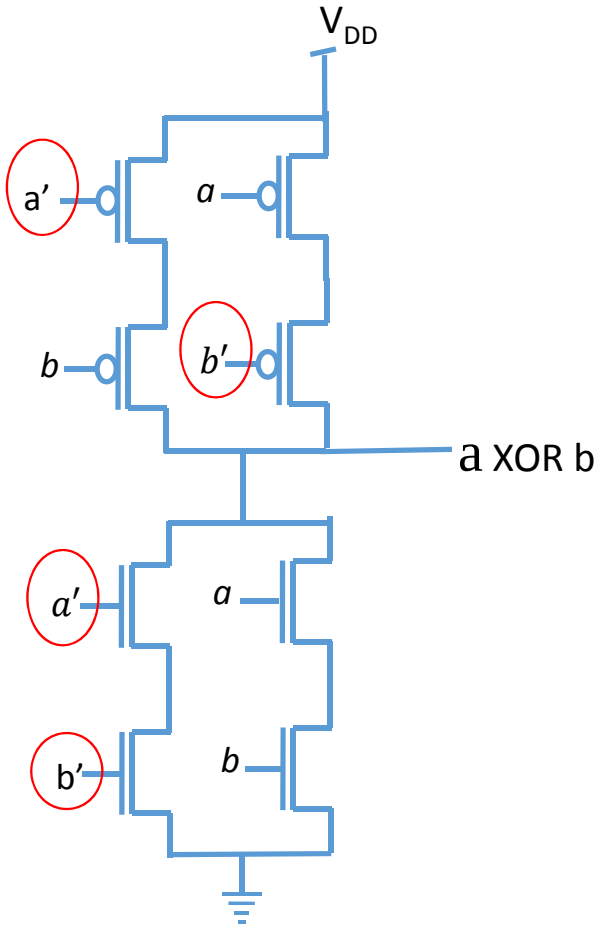
This Pull Down Network implements $fs_{OFF} = a' \cdot b' + a \cdot b$

Who provides the inverted literals ?

12 Transistors Needed

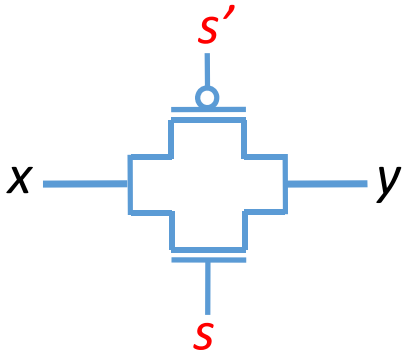




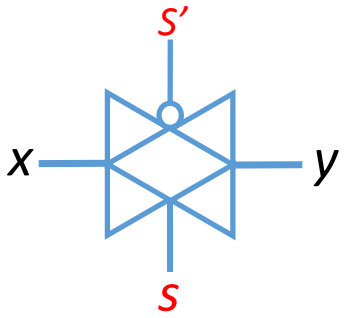


CMOS Transmission Gates

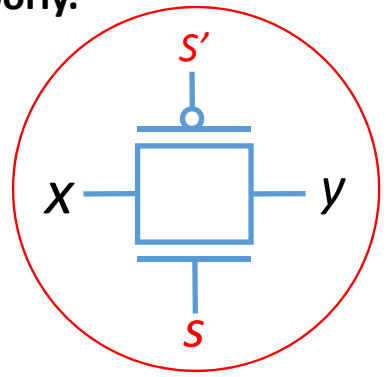
CMOS Transmission Gates (TGs)



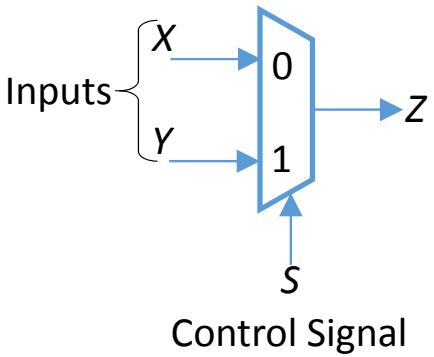
1. **Transmission Gates are also constructed from PMOS and NMOS transistors**
2. **They act as a switch whose operation can be controlled by a signal *s***
3. **When *s = 1*, both the NMOS and PMOS transistors are ON.**
   ***y = x*, i.e., *y* is connected to *x* or the resistance between *y* and *x* is negligibly low.**
   We say *x* is transmitted to *y* or *y* is driven by *x*
4. **When *s = 0*, *y* is disconnected from *x*, i.e., there is a high impedance between *y* and *x*.**
   The voltage of *y* has no corelation to the voltage at *x*.
5. **Why do we have two complementary transistors (NMOS and PMOS) ?**
   Because NMOS is good at transmitting 0 (0 V/GND) and PMOS is transmitting 1(V_{DD}). If we had only one transistor PMOS or NMOS, it would still function as a switch but it will transmit 0 poorly or 1 poorly.



Transmission Gate Symbols



Multiplexor

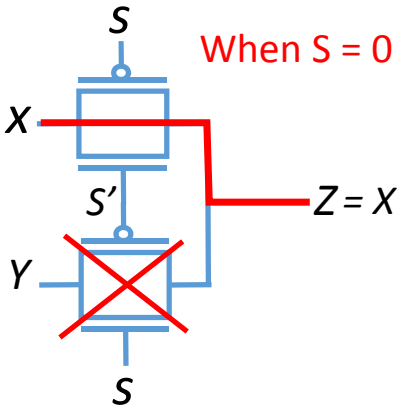


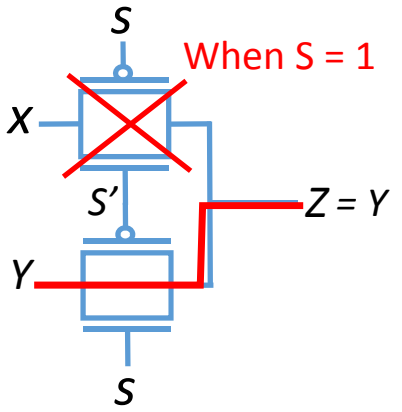
1. It is a universal building block like NAND or NOR gate because any arbitrary boolean function can be built using Muxes. We will study this in a later lecture on combinatorial circuits
2. Multiplexor are often abbreviated to Mux, plural form - Muxes

X	***Y***	***S***	***Z***
x	-	0	*x*
-	*y*	1	*y*

```
Z  <= X when S = 0 ELSE
      Y;
```

Mux Using Transmission Gate





X	Y	S	Z	
0	-	0	0	The upper TG is ON and passes X to Z
1	-	0	1	The lower TG is OFF and blocks Y
-	0	1	0	The lower TG is ON and passes Y to Z
-	1	1	1	The upper TG is OFF and blocks X

'-' symbol means don't care.

The output *Z* is unaffected by what ever value of *X* or *Y* in the rows where their value is shown as '-'

Homework Exercise:

1. Draw the truth table of two to 1 multiplexor
2. Design the Multiplexor using NAND gates. Use Karnaugh Map
3. Draw the corresponding CMOS transistor schematic
4. Show the flow current for different rows of the truth table

Tri State

Section 3.8 and Section 3.9

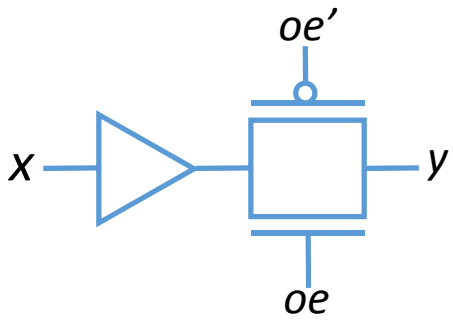
1. All nodes (input and output) of digital logic occupy, ideally logic 1 or logic 0 state.
2. These states imply that the node in logic is driven by V_{DD} or connected to GND (logic 0)
3. However, in some cases, a node in the logic may be not connected to either V_{DD} or GND.
4. This may sound as an anomaly (abnormality), however in reality this is desirable and is very useful.
5. This *third state is often called the high impedance state* because there is a high impedance between the node and the VDD / GND
6. This third state is represented by *Z*
7. Logic gates that can intentionally drive its output to the *Z* state are called tri-stated

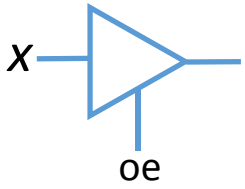
oe: output enable

x	oe	y
x	1	x
-	0	Z

y = x when oe = 1

y = Z when oe = 0; y is disconnected from x





y = x when oe = 1
else Z

Inverting Tri State

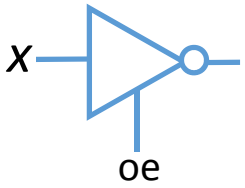
Section 3.8 and Section 3.9

oe: output enable

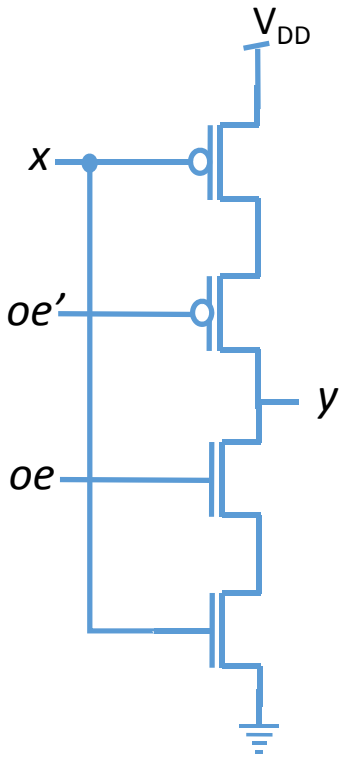
x	oe	y
x	1	x’
-	0	Z

y = x’ when oe = 1

y = Z when oe = 0; y is disconnected from x



y = x when oe = 1
else Z

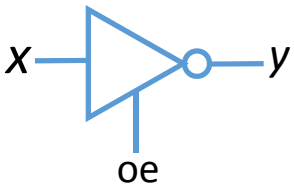


Tri State Variants

Section 3.8 and Section 3.9

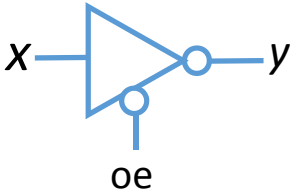
oe: output enable

x	oe	y
x	1	$\overline{x}$
-	0	Z

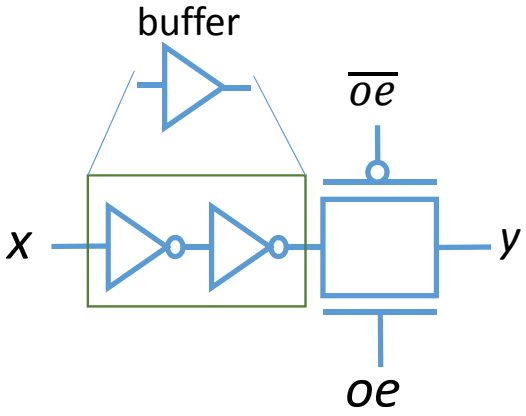


$y = \overline{x}$ *when oe = 1*
else Z

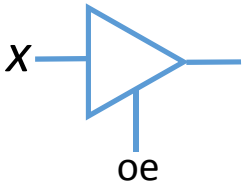
x	oe	y
x	0	$\overline{x}$
-	1	Z



$y = \overline{x}$ *when oe = 0*
else Z

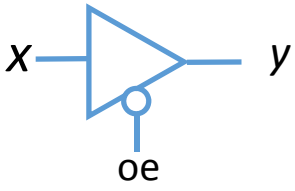


x	oe	y
x	1	x
-	0	Z



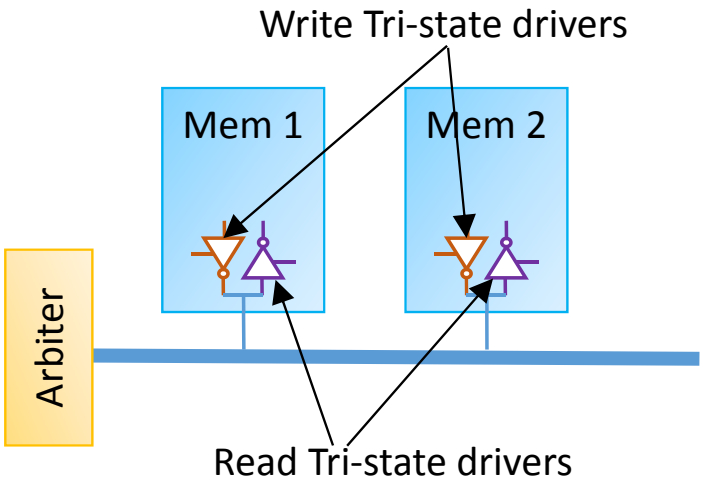
y = x when oe = 1
else Z

x	oe	y
x	0	x
-	1	Z



y = x when oe = 0
else Z

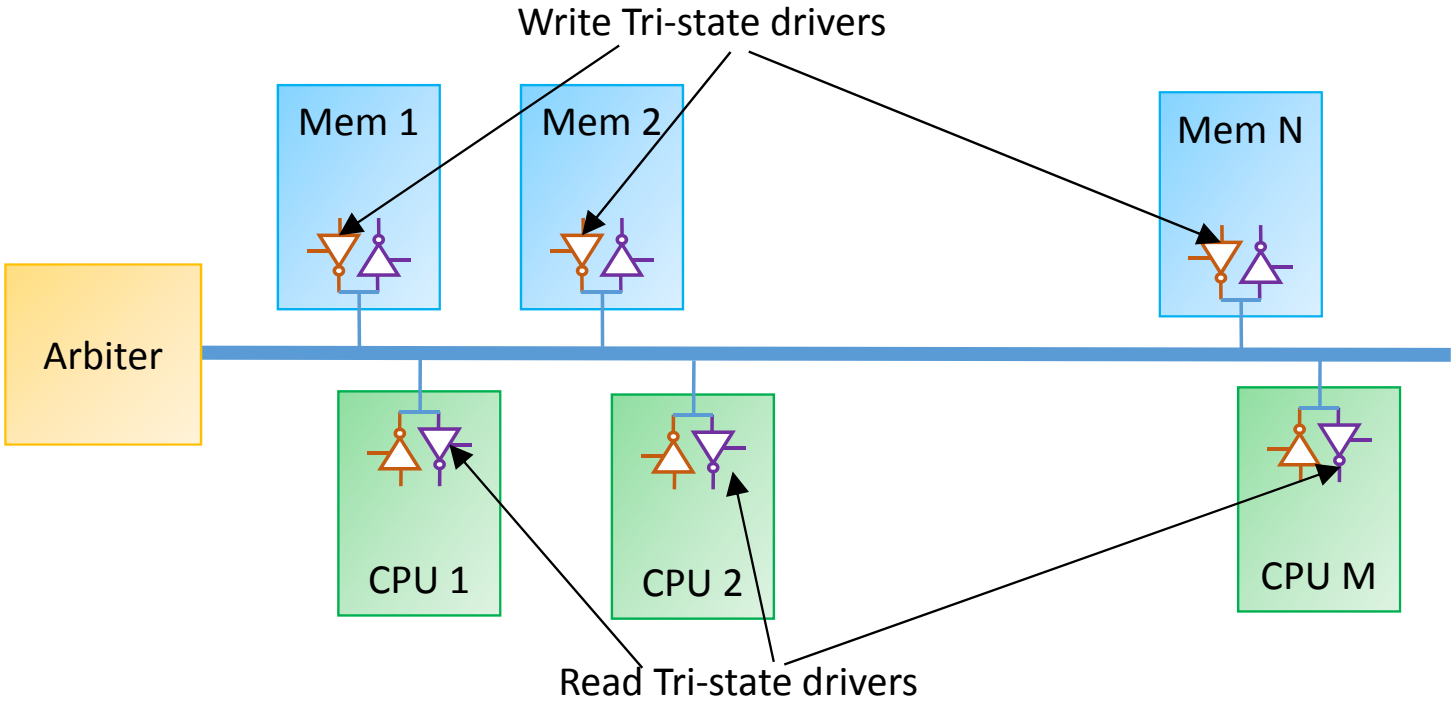
What are tri state buffers/drivers used for ?



1. Let us say two Memory Devices share a bus (a bundle of wires)
2. Both memory devices can read from the bus at the same time
3. Both memory devices cannot however write to the bus at the same time
   One could be wanting to write logic 1 and ther other could be wanting write logic 0. This would create a short circuit between VDD and GND
4. Tri-state buffers / drivers help because we can disable one of the driver, while the other is writing
5. Usually there is a device called arbiter that grants access to the bus before it can write to it.
6. If all devices can read, why do we need ***read*** tri-state drivers ?
7. This is because if Mem 1 device is writing to the bus the read tri-state driver in Mem 1 is disabled. What Mem 1 is writing is meant for external devices.

What are tri states used for ?

Tri States are used when we need bi-directional connections
When a device or a sub-system what to read and write from the same port



Only one device is allowed to drive, i.e., write to the bus. The output enable of only one output tri-state driver is activated at a time. All others are inactivate
Multiple devices can read.