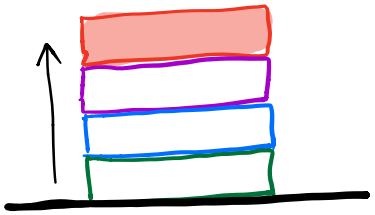
Week 1

Abstraction is the critical concept!

If you can break a system into independent layers, each with a well-defined **interface**, you only need to implement one layer at a time.

You don't have to worry about the details of **how** the below layers are implemented – you only need to understand **what** they do.

Abstraction and Implementation



Boolean Algebra

↗ also known as "canonical form"

"Disjunctive Normal form" → used to construct a Boolean expression from a truth table by describing each row with a result of "1" with ANDs, then OR-ing those equations together.

A Boolean function:

x	y	z	f
0	0	0	1
0	0	1	0
0	1	0	1
⋮	⋮	⋮	⋮

$$f = \begin{cases} (\bar{x} \cdot \bar{y} \cdot \bar{z}) \leftarrow \\ + \\ (\bar{x} \cdot y \cdot \bar{z}) \leftarrow \\ + \\ \vdots \end{cases}$$

From here, expression can be simplified using laws of Boolean Algebra (see book or video lecture)

Some fundamental theorems:

1) Every Boolean function can be represented using only OR, AND, and NOT operations

↓

$x \text{ OR } y = \text{NOT}(x) \text{ AND } \text{NOT}(y)$ De Morgan's Law

↓

2) Every Boolean function can be represented using only AND and NOT gates

↓

Construct a new NAND operator.

$x \text{ NAND } y = \text{NOT}(x \text{ AND } y)$

Prove you can do NOT and AND with NAND

$\text{NOT}(x) = x \text{ NAND } x$

$x \text{ AND } y = \text{NOT}(x \text{ NAND } y)$

$= (x \text{ NAND } y) \text{ NAND } (x \text{ NAND } y)$

↓

3) Every Boolean function can be represented using only NAND gates!

Note: 1-3 can also be done with OR gates to prove that any Boolean function can be represented using only NOR gates.

Unlike functions on e.g. integers, there are a finite number of Boolean functions for a given number of input variables. This allows us to completely describe a Boolean function using a truth table and construct it using only AND, OR, and NOT gates. 2^{2^n}

Gate Logic

One abstraction can have many different implementations!

Some may be better than others in terms of power, elegance, cost, efficiency, etc...

This course does not deal with physical implementations!

HDL → Hardware Description Language

Simulation

How do you generate the compare file (.cmp)?

You can generate the correct output using the idea of Behavioral simulation

↳ write the functionality/implementation of the chip in some high-level language!

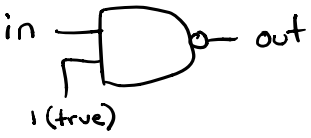
Specify the behavior of the chip, then compare it to the actual implementation!

Hardware construction projects

- System architects → creates API, specification, requirements for each component
- Developers → go and implement each component

Constructing primitive gates

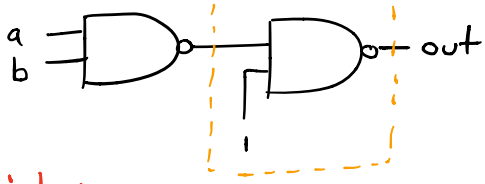
NOT:



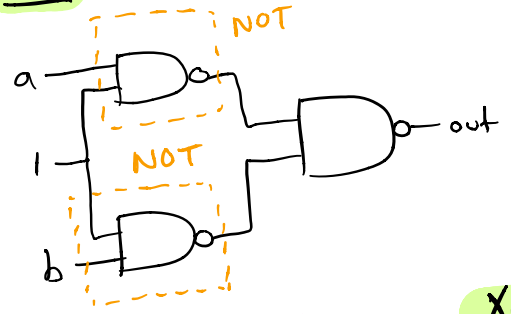
Alternatively:

AND:

simply invert NAND



OR: $f = A + B = \overline{(\bar{A} \cdot \bar{B})}$ De Morgan's Law



XOR:

(least composite gates)

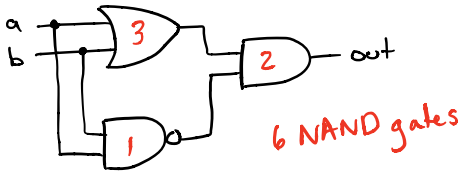
Implementation #1

a	b	f	OR	NAND	OR · NAND
0	0	0	0	1	0
0	1	1	1	1	1
1	0	1	1	1	1
1	1	0	1	0	0

can be implemented using only 3 composite gates!

$$f = (\bar{a} \cdot b) + (a \cdot \bar{b})$$
$$= (\bar{a}+a) \cdot (\bar{a}+\bar{b}) \cdot (b+a) \cdot (b+\bar{b}) \quad \text{Distributive Law}$$
$$1 \cdot (\bar{a}+\bar{b}) \cdot (b+a) \cdot 1$$
$$= (\bar{a}+\bar{b}) \cdot (b+a)$$
$$= \overline{(a \cdot b)} \cdot (b+a) \quad \text{De Morgan's Law}$$

NAND OR



6 NAND gates

(least primitive gates)

Implementation #2

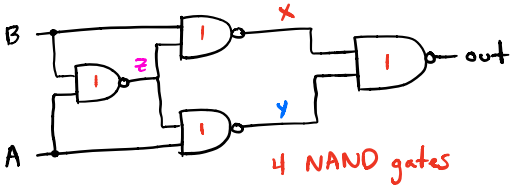
$$f = (\bar{a} \cdot b) + (a \cdot \bar{b})$$
$$= \overline{(\overline{(\bar{a} \cdot b)} \cdot \overline{(a \cdot \bar{b})})} \quad \text{De Morgan's Law}$$
$$= \overline{(a+\bar{b}) \cdot (\bar{a}+b)} = X \text{ NAND } Y$$

X Y

$$X = a + \bar{b}$$
$$= \overline{(\bar{a} \cdot b)}$$
$$= \overline{(\bar{a} \cdot b) + (\bar{b} \cdot b)}$$
$$= \overline{(\bar{a}+\bar{b}) \cdot b}$$
$$= \overline{(a \cdot \bar{b}) \cdot b} \quad z$$
$$= B \text{ NAND } (A \text{ NAND } B)$$

$$y = \bar{a} + b$$
$$= \overline{(a \cdot \bar{b})}$$
$$= \overline{(a \cdot \bar{b}) + (a \cdot \bar{a})}$$
$$= \overline{(\bar{b}+\bar{a}) \cdot a}$$
$$= \overline{(b \cdot a) \cdot a} \quad z$$
$$= A \text{ NAND } (A \text{ NAND } B)$$

$$f = (B \text{ NAND } (A \text{ NAND } B)) \text{ NAND } (A \text{ NAND } (A \text{ NAND } B))$$
z z



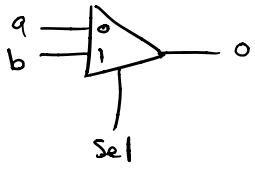
4 NAND gates

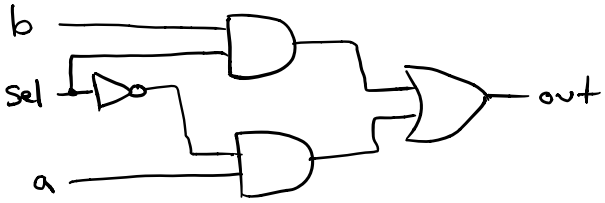
The above example with XOR shows one example of the optimizations that matter in practice but are not covered as part of the course. Implementation #2 is better for speed, power, area, and cost, but arriving at it is non-obvious. Further optimizations can be made at the transistor level.

Mux:

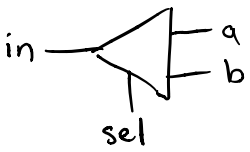
sel	a	b	out
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

$(a \cdot \overline{sel}) + (b \cdot sel)$





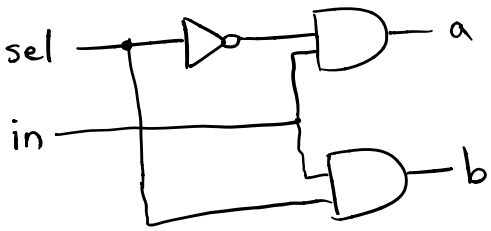
DMux:



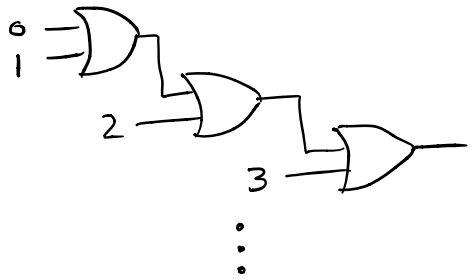
in	sel	a	b
0	0	0	0
0	1	0	0
1	0	1	0
1	1	0	1

$b = in \cdot sel$

$a = in \cdot \overline{sel}$



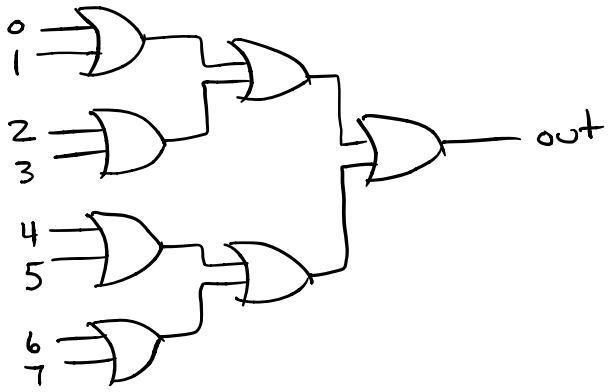
8 way Or:



7 OR gates

(n-1) OR gates

7 gates of delay



7 OR gates

Stages in tree = $\log_2 n$

Gates = $2^0 + 2^1 + \dots + 2^{(\log_2 n - 1)}$

$= \sum_{i=0}^{\log_2 n - 1} 2^i = 2^{(\log_2 n)} - 1$

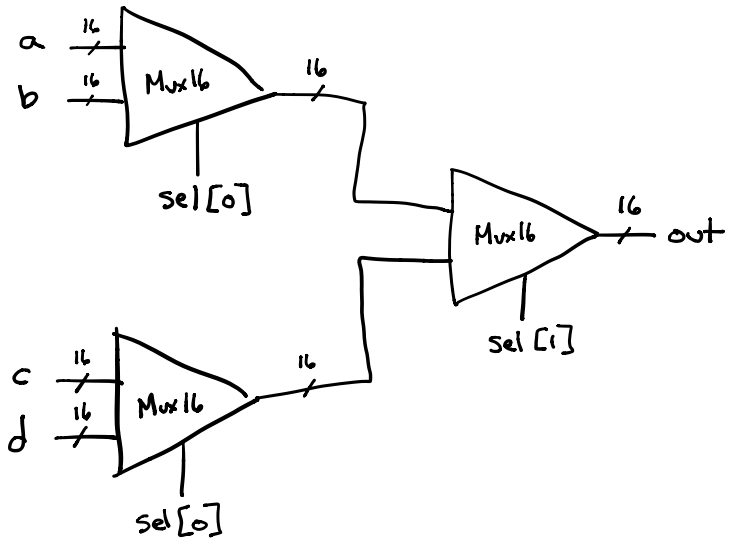
$= n - 1$

3 gates of delay

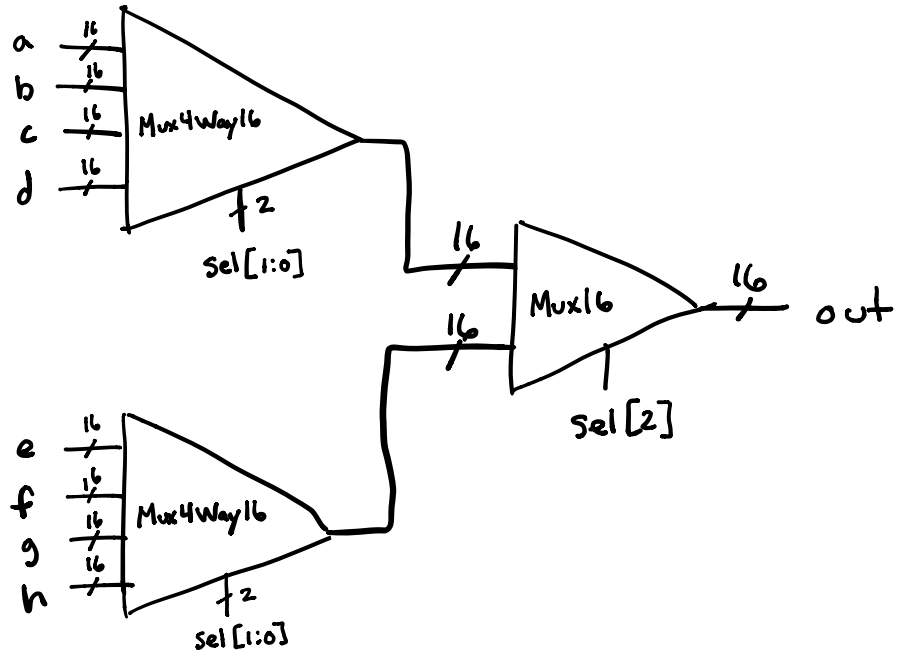
The 8-way OR provides another example of how multiple different implementations of the same abstraction may have different performance characteristics.

They both use the same number of gates, but the "tree" design on the right is "faster" (in terms of maximum propagation delay).

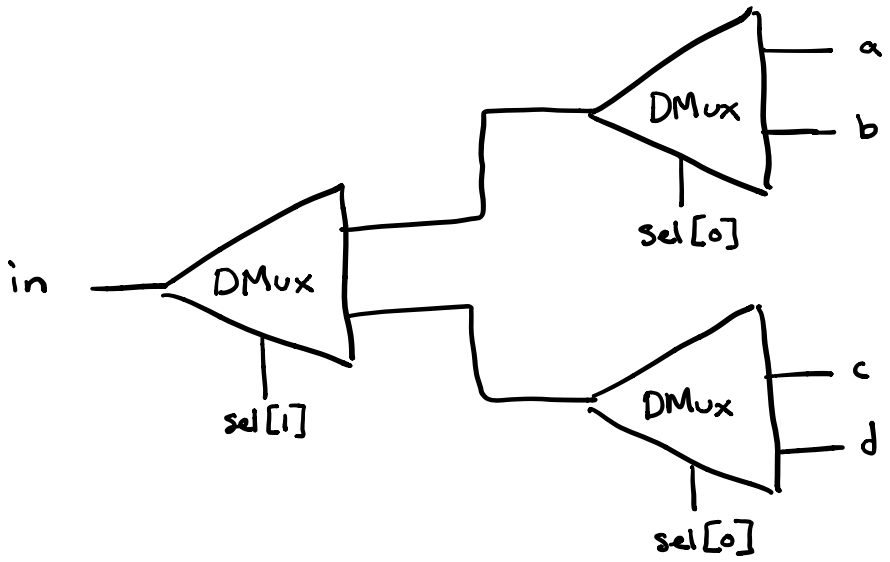
Mux4Way16:



Mux8Way16:



DMux4Way :



DMux8Way :