CS 240
Foundations of Computer Systems

WELLESLEY

Digital Logic

*Gate*way to computer science

transistors, gates, circuits, Boolean algebra

https://cs.wellesley.edu/~cs240/

Software
- Program, Application
- Programming Language
- Compiler/Interpreter
- Operating System

Instruction Set Architecture

Hardware
- Microarchitecture
- **Digital Logic**
- Devices (transistors, etc.)
- Solid-State Physics

Digital data/computation = Boolean

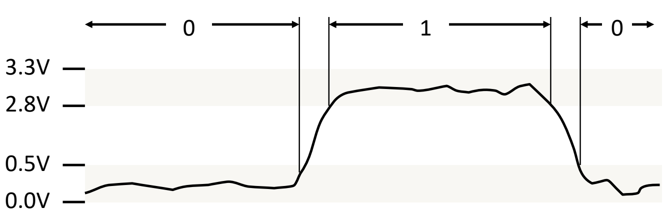
Abstraction!

Boolean value (***bit***): **0 or 1**

Boolean functions (AND, OR, NOT, ...)

Electronically:

bit = high voltage vs. low voltage

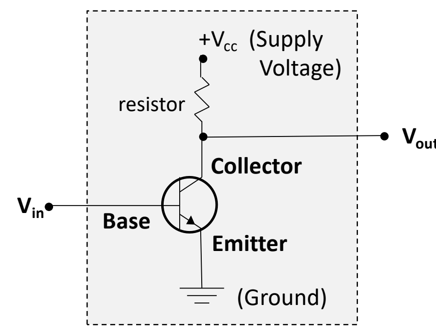


Boolean functions = logic gates, built from transistors

Transistors (more in lab)

If *Base* voltage is high:
Current may flow freely from *Collector* to *Emitter*.

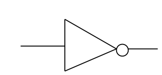
If *Base* voltage is low:
Current may not flow from *Collector* to *Emitter*.



Truth table

V_{in}	V_{out}		in	out		in	out
low	high	=	0	1	=	F	T
high	low		1	0		T	F

NOT gate



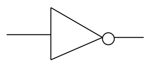
Abstraction!

Digital Logic Gates

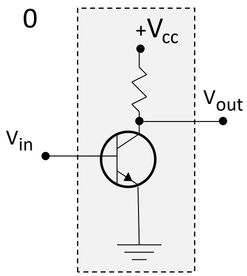
Abstraction!

ex

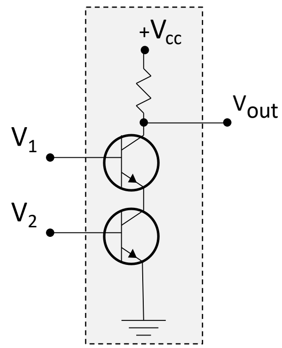
Tiny electronic devices that compute basic Boolean functions.

NOT 

V_{in}	V_{out}
0	1
1	0

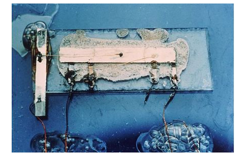


	V_2 0	1
V_1 0		
1		

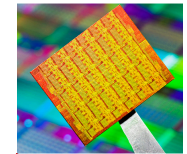


Integrated Circuits (1950s -)

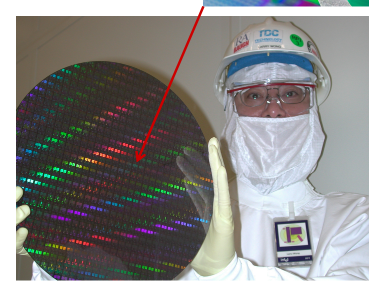
Early (first?) **transistor**



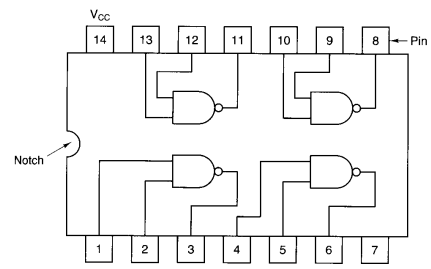
Chip



Wafer

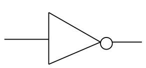


Small integrated circuit

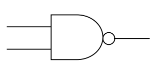


Five basic gates: define with truth tables

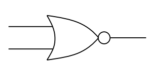
ex



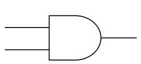
NOT	
0	1
1	0



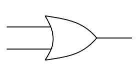
NAND	0	1
0	1	1
1	1	0



NOR	0	1
0		
1		



AND	0	1
0		
1		



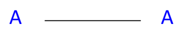
OR	0	1
0		
1		

Inspired by gates?

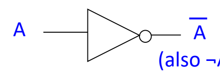
Simple Boolean Expressions

for combinational logic

inputs = *variables*
wires = *expressions*
gates = *operators/functions*
circuits = *functions*

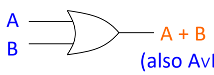


wire: identity



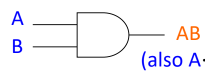
(also ¬A, ~A, A')

NOT: inverse or complement



(also A∨B)

OR = Boolean sum



(also A·B, A∧B, , A.B, A*B)

AND = Boolean product

Orange forms are most convenient in text editors.

A **boolean literal** is a variable or its complement.

E.g., A, A', B, B' are literal boolean expressions, but A + B, AB, and (AB)' are not.

General Boolean Expressions

Boolean expressions are generated by this context free grammar:

BE ::= *variable* | 0 | 1 | BE' | BE + BE | BE * BE | (BE)

Precedence: (...) > NOT > AND > OR

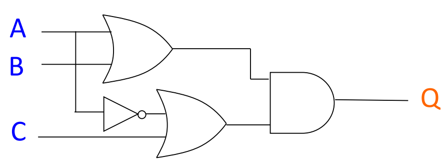
E.g., A'B + CD' means ((A')*B) + (C*(D'))

Circuits & Boolean Expressions

ex

Given input variables, **circuits** specify outputs as functions of inputs using wires & gates.

- Crossed wires touch *only if* there is an explicit dot.
- T intersections copy the value on a wire and don't need a dot.
- Doesn't make sense to wire together two inputs or two outputs; instead, combine two independent wires with a gate!



Each output can be translated to a boolean expression in terms of the input variables.

What is a boolean expression for Q in the above circuit?

What is the truth table for Q in the example circuit?

A	B	C	Q
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Translation Exercise

ex

Connect gates to implement these functions. Check with truth tables.

Use a direct translation -- it is straightforward and bidirectional.

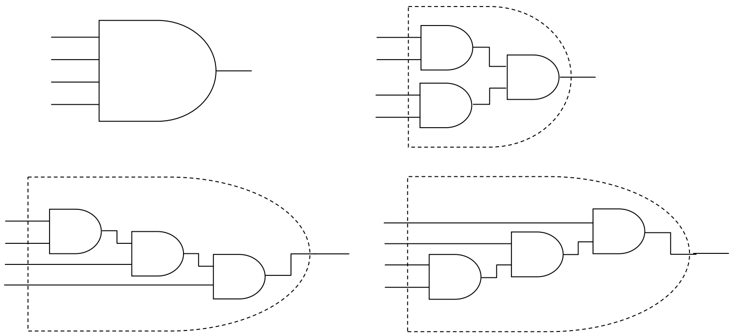
$F = (A\overline{B} + C)D$

$Z = \overline{W} + (X + \overline{WY})$

Larger gates

Using 2-input AND gates, it's easy to build an AND gate with more than 2 inputs.

E.g., Below are several ways to build a 4-input AND gate from three 2-input AND gates.



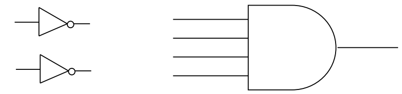
Multi-input OR gates with can be created analogously.

Multi-input NAND and NOR gates can be created by inverting the outputs of multi-input AND and OR gates.

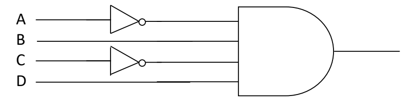
Circuit derivation: ***code detectors***

ex

A multi-input AND gate preceded by some inverters = **code detector** that recognizes **exactly one** input code (a specific combination of inputs).



E.g., here's a 4-input code detector that outputs 1 if ABCD = 0101, and 0 otherwise:



Design a 4-input code detector to accept two codes (ABCD=1001, ABCD=1101) and reject all others. (accept = 1, reject = 0)

Sum-of-products (SoP) Form

ex

A **sum-of-product (SoP)** form is a boolean expression for a circuit output that is expressed as a sum of **minterms**, one for each row whose output is 1.

A **minterm** for a row is a product of literals (variables or their negations) whose value is 1 for that row. Think of it as being a **code detector** for that row!

What is the sum-of-products expression for truth table below?

A	B	C	Q
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

How would you draw the circuit for this expression?

How is it related to **code detectors** from the previous slide?

Voting machines

ex

A majority circuit outputs 1 if and only if a majority of its inputs equal 1.

Design a majority circuit for three inputs. Use a sum of products.

A	B	C	Majority
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Triply redundant computers in spacecraft

- Space program also hastened Integrated Circuits.

Product-of-sums (PoS) Form

ex

A **product-of-sums (PoS)** form is a boolean expression for a circuit output that is expressed as a product of **maxterms**, one for each row whose output is 0.

A **maxterm** for a row is a sum of literals (variable or their negations) whose value is 0 for that row.

What is the product-of-sums expression for this truth table?

A	B	C	Q
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

How would you draw the circuit for this expression?

How can you relate it to the notion of **code detectors**?

Boolean Algebra: Simple laws

Boolean algebra laws can be proven by truth tables and used to show equivalences between boolean expressions.

For all laws in one place, see the Boolean Laws Reference Sheet

Name of Law / Theorem	Form	Equivalent/Dual form (interchange AND and OR, and 0 and 1)
Involution (or double negation)	$\overline{\overline{A}} = A$	none
Identity	$0+A = A$	$1*A = A$
Inverse (or Complements)	$A\overline{A} = 0$	$A+\overline{A} = 1$
Commutativity	$A+B = B+A$	$AB = BA$
Associativity	$(AB)C = A(BC)$	$(A+B)+C = A+(B+C)$
Idempotent	$A+A = A$	$AA = A$
Null (or Null Element)	$0*A = 0$ **(the Zero Law)**	$1+A = 1$ **(the One Law)**

Boolean Algebra: More Complex Laws

Distributive	$A+BC = (A+B)(A+C)$	$A(B+C) = AB+AC$
DeMorgan's	$\overline{A}+\overline{B}+\overline{C}+... = \overline{ABC...}$	$\overline{A+B+C+...} = \overline{A}\ \overline{B}\ \overline{C}...$
Absorption 1 (Covering)	$A+AB = A$	$A(A+B) = A$
Absorption 2	$A+\overline{A}B = A+B$	$A(\overline{A}+B) = AB$
Combining	$AB+A\overline{B} = A$	$(A+B)(A+\overline{B}) = A$
Consensus	$AB+\overline{A}C+BC = AB+\overline{A}C$	$(A+B)(\overline{A}+C)(B+C) = (A+B)(\overline{A}+C)$

You can use truth tables (or other Boolean laws) to convince yourself that these laws hold. (See the exercises on the following slides).

Boolean Algebra: Proving Laws by Truth Tables

ex

Distributive	$A+BC = (A+B)(A+C)$	$A(B+C) = AB+AC$

Complete the truth tables below to show that both distributive laws hold.

A	B	C	A+BC
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

A	B	C	(A+B)(A+C)
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

A	B	C	A(B+C)
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

A	B	C	AB + AC
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Notes:

- the law in the right column (distributing multiplication over addition) should be familiar from algebra of numbers.
- the law in the left column (distributing addition over multiplication) is **not** true in the algebra of numbers but **is** true for Boolean algebra!

Boolean Algebra: Proving Laws by Truth Tables

ex

DeMorgan's	$\overline{A}+\overline{B}+\overline{C}+\ldots=\overline{ABC\ldots}$	$\overline{A+B+C+\ldots}=\overline{A}\;\overline{B}\;\overline{C}\ldots$

Complete the truth tables below to show that both DeMorgan's laws hold for two variables.

A	B	C	(A')+(B')
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

A	B	C	(AB)'
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

A	B	C	(A+B)'
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

A	B	C	(A')(B')
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Boolean Algebra: Proving Laws by Algebra

Absorption 1 (Covering)	$A+AB=A$	$A(A+B)=A$

Each step has a **redex** = the subexpression to which the law is applied.

Here, both the redex and the applied law are highlighted in the same color.

But redexes can also be highlighted by boxing, underlining, etc.

The explicit *s highlight the duality between * and +, but can be replaced by juxtaposition.

Step-by-step derivation

explicit Commutativity

A + A*B
= 1*A + A*B [Identity (*)]
= A*1 + A*B [Commutativity (*)]
= A*(1 + B) [Distributivity (*/+)]
= A*1 [One law]
= 1*A [Commutativity (*)]
= A [Identity]

implicit Commutativity

A + A*B
= A*1 + A*B [Identity]
= A*(1 + B) [Distributivity]
= A*1 [One law]
= A [Identity]

***Dual** step-by-step derivation*
*(Swap * ⇔ +, 0 ⇔ 1)*

A*(A + B)
= (0 + A)*(A + B) [Identity (+)]
= (A + 0)*(A + B) [Commutativity (+)]
= A + 0*B [Distributivity (+/*)]
= A + 0 [Zero law]
= 0 + A [Commutativity (+)]
= A [Identity]

A*(A + B)
= (A + 0)*(A + B) [Identity (+)]
= A + 0*B [Distributivity (+/*)]
= A + 0 [Zero law]
= A [Identity]

Boolean Algebra: Proving Laws by Algebra

ex

Absorption 2	$A+\overline{A}B=A+B$	$A(\overline{A}+B)=AB$

Boolean Algebra: Proving Laws by Algebra

ex

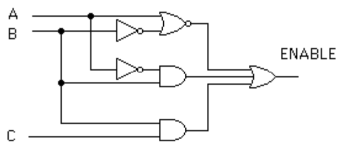
Combining	$AB+A\overline{B}=A$	$(A+B)(A+\overline{B})=A$

Circuit simplification

Why simplify?

ex

Is there a simpler circuit that performs the same function?

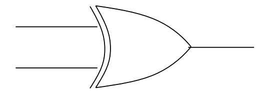


Start with an equivalent Boolean expression, then simplify with algebra, and convert the simplified expression back to a circuit.

F(A, B, C) =

Check the answer with a truth table.

XOR: Exclusive OR



ex

Output = 1 if exactly one input = 1.

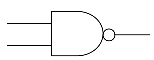
Truth table:

XOR	0	1
0		
1		

Build from earlier gates (start with SoP or PoS):

Often used as a one-bit comparator.

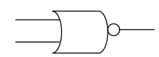
NAND is *universal*



ex

All Boolean functions can be implemented using only NANDs.
Build NOT, AND, OR, NOR, using only NAND gates.

NOR is also *universal*

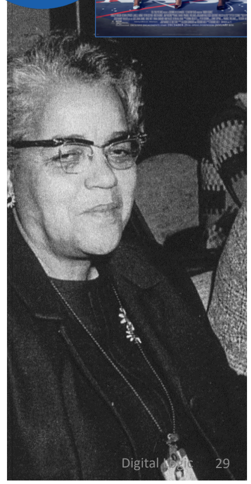


ex

All Boolean functions can also be implemented using only NORs.
Build NAND using only NOR gates; then since NAND is universal, NOR must be too! (Why?)

Computers

- Manual calculations
- powered all early US **space** missions.
- Facilitated transition to digital computers.



Katherine Johnson

- Supported Mercury, Apollo, Space Shuttle, ...

Mary Jackson

- NASA's first black female engineer
- Studied air around airplane via wind tunnel experiments.

Dorothy Vaughan

- First black supervisor within NACA
- Early self-taught FORTRAN programmer for NASA move to digital computers.

Early pioneers in reliable computing

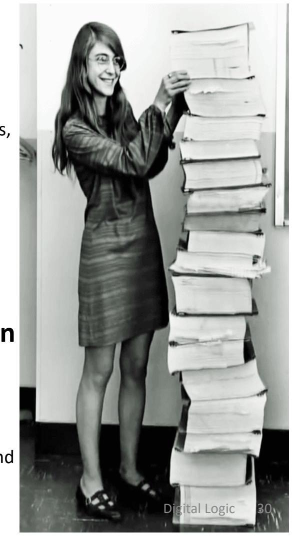
Katherine Johnson

- Calculated first US human space flight trajectories
- Mercury, Apollo 11, Space Shuttle, …
- Reputation for accuracy in manual calculations, verified early code
- Called to verify results of code for launch calculations for first US human in orbit
- Backup calculations helped save Apollo 13
- Presidential Medal of Freedom 2015

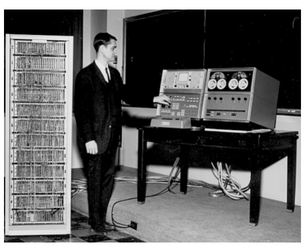
Margaret Hamilton

- Led software team for Apollo 11 Guidance Computer, averted mission abort on first moon landing.
- Coined "software engineering", developed techniques for correctness and reliability.
- Presidential Medal of Freedom 2016

Apollo 11 code print-out



Wellesley Connection: Mary Allen Wilkes '59



Created LAP operating system at MIT Lincoln Labs for Wesley A. Clark's LINC computer, widely regarded as the first personal computer (designed for interactive use in bio labs). Work done 1961—1965.

Created first interactive keyboard-based text editor on 256 character display. LINC had only 2K 12-bit words; (parts of) editor code fit in 1K section; document in other 1K.

In 1965, she developed LAP6 with LINC in Baltimore living room. First home PC user!

Early versions of LAP developed using LINC simulator on MIT TX2 compute, famous for GUI/PL work done by Ivan and Bert Sutherland at MIT.

Later earned Harvard law degree and headed Economic Crime and Consumer Protection Division in Middlesex (MA) County District Attorney's office.