

adding numbers in binary

\[ 317 + 422 = (100111101)_2 + (110100110)_2 \]

```
\[ \begin{array}{cccc}
1 & 0 & 0 & 1 \\
1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 & 1 \\
1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\
\end{array} \]
```

1-bit binary adder

- Inputs: \( A, B, \text{Carry-in} \)
- Outputs: \( \text{Sum}, \text{Carry-out} \)

```
\begin{array}{cccccc}
A & B & \text{Cin} & S & \text{Cout} \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 & 1 \\
0 & 1 & 1 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 & 1 \\
1 & 1 & 0 & 1 & 0 \\
1 & 1 & 1 & 0 & 0 \\
\end{array}
```

apply theorems to simplify expressions

The theorems of Boolean algebra can simplify expressions
- e.g., full adder's carry-out function

\[
\text{Cout} = A' \text{B Cin} + A \text{B' Cin}' + A \text{B Cin}' + A \text{B Cin}
\]

adding extra terms creates new factoring opportunities

1-bit adder

```
\begin{array}{cccccc}
A & B & \text{Cin} & \text{S} & \text{Cout} \\
0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 \\
0 & 1 & 1 & 1 & 1 \\
1 & 0 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 \\
\end{array}
```

\[
\text{S} = A' \text{B' Cin} + A' \text{B Cin}' + A' \text{B' Cin'} + A \text{B Cin}
\]

\[
\text{Cout} = A' \text{B Cin} + A \text{B' Cin} + A \text{B Cin'} + A \text{B Cin}
\]

cse 311: foundations of computing

Spring 2015
Lecture 5: Canonical forms and predicate logic

I love quantifying

2-bit ripple-carry adder

```
\begin{array}{cccccc}
A & B & A_1 & B_1 & A_2 & B_2 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 1 & 0 \\
1 & 1 & 0 & 0 & 1 & 1 \\
\end{array}
```

```
\begin{array}{cccccc}
\text{Cin}_0 & \text{Cin}_1 & \text{Cout}_1 & \text{Cout}_2 & \text{Sum}_0 & \text{Sum}_1 & \text{Sum}_2 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 1 & 1 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 1 & 1 \\
\end{array}
```
mapping truth tables to logic gates

Given a truth table:
1. Write the Boolean expression
2. Minimize the Boolean expression
3. Draw as gates
4. Map to available gates

A
B
C
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

\[ F = A'BC + A'BC' + ABC + ABC' \]
\[ = A'(BC + C') + (A + B)C' \]
\[ = A'B + AC' \]

Truth table is the unique signature of a Boolean function
The same truth table can have many gate realizations
- we've seen this already
- depends on how good we are at Boolean simplification

Canonical forms
- standard forms for a Boolean expression
- we all come up with the same expression

canonical forms

• Truth table is the unique signature of a Boolean function
• The same truth table can have many gate realizations
  - we've seen this already
  - depends on how good we are at Boolean simplification
• Canonical forms
  - standard forms for a Boolean expression
  - we all come up with the same expression

sum-of-products canonical form

- also known as Disjunctive Normal Form (DNF)
- also known as minterm expansion

sum-of-products canonical form

Product term (or minterm)
- ANDed product of literals – input combination for which output is true
- each variable appears exactly once, true or inverted (but not both)

\[ F = \sum \text{minterms} \]
\[ F(A, B, C) = \sum (A'B'C', A'BC', AB'C', ABC', ABC) \]

\[ F = (A + B + C) (A + B' + C) (A' + B + C) \]

product-of-sums canonical form

- Also known as Conjunctive Normal Form (CNF)
- Also known as maxterm expansion

Product of sums (or maxterm)
- ORed product of literals – input combination for which output is false
- each variable appears exactly once, true or inverted (but not both)

\[ F = \Pi \text{maxterms} \]
\[ F(A, B, C) = \Pi (A'B'C, A'BC, AB'C, ABC, ABC') \]

\[ F = (A + B + C) (A + B' + C) (A' + B + C) \]

s-o-p, p-o-s, and de Morgan’s theorem

Complement of function in sum-of-products form:
- \( F' = A'B'C + A'BC' + ABC' \)
Complement again and apply de Morgan’s and get the product-of-sums form:
- \( (F')' = (A'B'C' + A'BC' + ABC')' \)
- \[ F = (A + B + C) (A + B' + C) (A' + B + C) \]
product-of-sums canonical form

Sum term (or maxterm)
- ORed sum of literals – input combination for which output is false
- each variable appears exactly once, true or inverted (but not both)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Maxterms</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A+B+C</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>A+B+C'</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>A+B+C</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>A+B+C'</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>A+B+C</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>A+B+C'</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>A+B+C</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>A+B+C'</td>
</tr>
</tbody>
</table>

F in canonical form:
- F(A, B, C) = (A + B + C) (A + B' + C) (A' + B + C)
- canonical form = minimal form
- F(A, B, C) = (A + B + C) (A + B' + C) (A' + B + C)
- F(A, B, C) = (A + B + C) (A + B' + C) (A' + B + C)
- F(A, B, C) = (A + C) (B + C)

 predicate logic

- Propositional Logic
  - If Pikachu doesn’t wear pants, then he flies on Bieber’s jet unless Taylor is feeling lonely.

- Predicate Logic
  - If x, y, and z are positive integers, then $x^3 + y^3 \neq z^3$.

predicate logic

Predicate or Propositional Function
- A function that returns a truth value, e.g.,
  - "x is a cat"
  - "x is prime"
  - "student x has taken course y"
  - "x > y"
  - "x + y = z" or Sum(x, y, z)
  - "5 < x"

Predicates will have variables or constants as arguments.

domain of discourse

We must specify a "domain of discourse", which is the possible things we’re talking about.

- "x is a cat"
  (e.g., mammals)
- "x is prime"
  (e.g., positive whole numbers)
- student x has taken course y
  (e.g., students and courses)

quantifiers

∀x P(x)
P(x) is true for every x in the domain
read as "for all x, P of x"

∃x P(x)
There is an x in the domain for which P(x) is true
read as "there exists x, P of x"

statements with quantifiers

- ∃x Even(x)
- ∀x Odd(x)
- ∀x (Even(x) Æ Odd(x))
- ∃x (Even(x) Æ Odd(x))
- ∀x Greater(x+1, x)
- ∃x (Even(x) Æ Prime(x))

Domain:
Positive Integers

Even(x) (Odd(x) (Prime(x) Greater(x,y) (or "x>y") Equal(x,y) (or "x=y") Sum(x,y,z))

I love Pokémon
• \( \forall x \exists y \, \text{Greater}(y, x) \)
  - Domain: Positive Integers

• \( \forall x \exists y \, \text{Greater}(x, y) \)

• \( \forall x \exists y \,(\text{Greater}(y, x) \land \text{Prime}(y)) \)

• \( \forall x \, (\text{Prime}(x) \rightarrow (\text{Equal}(x, 2) \lor \text{Odd}(x))) \)

• \( \exists x \exists y \,(\text{Sum}(x, 2, y) \land \text{Prime}(x) \land \text{Prime}(y)) \)

Domain of quantifiers is important!

English to predicate logic

• "Red cats like tofu"

• "Some red cats don’t like tofu"

negations of quantifiers

• \( \forall x \, \text{PurpleFruit}(x) \)
  - "All fruits are purple"
  - What is \( \neg \forall x \, \text{PurpleFruit}(x) \)?
  - "Not all fruits are purple"

• How about \( \exists x \, \text{PurpleFruit}(x) \)?
  - "There is a purple fruit"
  - If it’s the negation, all situations should be covered by a statement and its negation.
  - Consider the domain (Orange): Neither statement is true!
  - No.

• How about \( \exists x \, \neg \text{PurpleFruit}(x) \)?
  - "There is a fruit that isn’t purple"
  - Yes.

• not every positive integer is prime

• some positive integer is not prime

• prime numbers do not exist

• every positive integer is not prime

de Morgan’s laws for quantifiers

\[ \neg \forall x \, P(x) \equiv \exists x \, \neg P(x) \]
\[ \neg \exists x \, P(x) \equiv \forall x \, \neg P(x) \]
de Morgan's laws for quantifiers

\[ \neg \forall x \ P(x) \equiv \exists x \ \neg P(x) \]
\[ \neg \exists x \ P(x) \equiv \forall x \ \neg P(x) \]

“There is no largest integer.”
\[ \neg \exists x \ \forall y \ (x \geq y) \equiv \forall x \ \neg \forall y \ (x \geq y) \equiv \forall x \ \exists y \ (x \geq y) \equiv \forall x \ \exists y \ (y > x) \]

“For every integer there is a larger integer.”

scope of quantifiers

example: Notlargest(x) = \exists y \text{ Greater} (y, x) = \text{ \exists z Greater} (z, x)

truth value:
- doesn’t depend on y or z “bound variables”
- does depend on x “free variable”

quantifiers only act on free variables of the formula they quantify
\[ \forall \ x \ (\exists \ y \ (P(x, y)) \rightarrow \forall \ x \ Q(x, y)) \]

scope of quantifiers

\[ \exists x \ (P(x) \wedge Q(x)) \quad \text{vs.} \quad \exists x \ P(x) \wedge \exists x \ Q(x) \]