

Propositional Logic

CSC 173

- Propositional logic – mathematical model (or algebra) for reasoning about the truth of logical expressions (propositions)
- Logical expressions – propositional variables or logical expressions connected with logical operators (not, and, or)
- Uses
 - design of digital circuits
 - Composition of logical expressions in programs
 - Automatic reasoning systems
 - Model of computation (programming language Prolog)

Logical Expressions

- Propositional variables (whose value is TRUE or FALSE) and the propositional constants TRUE and FALSE are logical expressions
- If LE1 and LE2 are logical expressions, then LE1 AND LE2 is a logical expression, whose value is TRUE if both LE1 and LE2 have the value TRUE, and is FALSE otherwise
- If LE1 and LE2 are logical expressions, then LE1 OR LE2 is a logical expression, whose value is TRUE if either LE1 or LE2 have the value TRUE, and is FALSE otherwise
- If LE1 is a logical expression, then NOT LE1 is a logical expression, whose value is TRUE if LE1 has the value FALSE, and is FALSE otherwise

Algebraic Laws for Logical Expressions

- AND and OR are commutative
- AND and OR are associative
- AND is distributive over OR; OR is distributive over AND
- TRUE is the identity for AND; FALSE is the identity for OR
- FALSE annihilates AND; TRUE annihilates OR
- AND and OR are idempotent ($p \text{ AND } p \equiv p$ OR $p \equiv p$)
- Subsumption
 - $(p \text{ OR } (p \text{ AND } q)) \equiv (p \text{ AND } (p \text{ OR } q)) \equiv p$
- DeMorgan's Laws:
 - $\text{NOT}(p \text{ AND } q) \equiv (\text{NOT } p) \text{ OR } (\text{NOT } q)$
 - $\text{NOT}(p \text{ OR } q) \equiv (\text{NOT } p) \text{ AND } (\text{NOT } q)$

Logic Minimization

- Essence of simplification
 - Repeatedly find two-element sub-sets of true values in which only one variable changes its value while the other variables do not
 - Apply the Unifying Theorem to eliminate the single varying variable –
 - $\text{FUNC} = A.B + A.\bar{B}$
 - $\text{FUNC} = A.(\bar{B}+B)$ – apply the Distributive law of Boolean Algebra
 - $F = A$ – apply the Inverse law of Boolean Algebra

CNF and DNF

- Disjunctive normal form (DNF) – sum of products
- Conjunctive normal form (CNF) – product of sums

Construct logical expressions from truth tables using either DNF or CNF

Karnaugh Maps

- A graphical representation of the truth table – boolean cube in n-dimensional space where n is the number of input variables
- Entry for each combination of input variables specifying the value of the output function
- Uses Gray code encoding – advancing from 1 index to the next changes the value of only a single input variable/bit
- Multi-dimensional table with logical adjacency along a dimension
 - And two adjacent elements (horizontal or vertical) are distance one apart
 - Adjacencies provide clues about whether uniting theorem can be applied

Goal: find a minimum cover of the 1's using rectangles or squares containing a power of 2 number of 1's

In essence, a mechanical method to find the *don't cares* in the truth table

Completeness of NAND

- $p \text{ AND } q \equiv ((p \text{ NAND } q) \text{ NAND } \text{TRUE})$
- $p \text{ OR } q \equiv ((p \text{ NAND } \text{TRUE}) \text{ NAND } (q \text{ NAND } \text{TRUE})) \equiv (\text{NOT } p) \text{ NAND } (\text{NOT } q)$
- $(\text{NOT } p) \equiv (p \text{ NAND } \text{TRUE}) \equiv (p \text{ NAND } p)$

- Laws of implication
- Reasoning with Propositional logic
 - Deductive proofs
 - Take as given a set of premises (or hypotheses) that are known to be true and attempt to prove a conclusion valid by a sequence of steps, termed inferences
 - Each inference follows from the premises or a previous inference by application of an inference rule

Laws of Implication

- $p \rightarrow q \equiv \text{NOT } p \text{ OR } q$
- $(p \rightarrow q) \text{ AND } (q \rightarrow p) \equiv (p \equiv q)$
- $(p \equiv q) \rightarrow (p \rightarrow q)$
- $(p_1 \text{ AND } p_2 \text{ AND } \dots p_n \rightarrow q) \equiv (\text{NOT } p_1 \text{ OR } \text{NOT } p_2 \text{ OR } \dots \text{NOT } p_n \text{ OR } q)$
- $(p \rightarrow q) \rightarrow (\text{NOT } q \rightarrow \text{NOT } p)$
(contrapositive law)
- $((p \rightarrow q) \text{ AND } (\text{NOT } p \rightarrow q)) \equiv q$

Reasoning with Propositional Logic

- Given premises (or inferences) $P_1 \dots P_n$, we can infer expression E if $P_1 \text{ AND } P_2 \text{ AND } \dots P_n \rightarrow E$ is a tautology
 - Whenever E is a tautology, $P_1 \text{ AND } P_2 \text{ AND } \dots P_n \rightarrow E$ is a tautology
 - Given two premises P_1 and P_2 , we can infer $P_1 \text{ AND } P_2$
 - If P_1 and $(P_1 \rightarrow P_2)$ are given or inferred, then we can infer P_2 by the rule of "modus ponens" $((p \text{ AND } (p \rightarrow q)) \rightarrow q)$
 - If $\text{NOT } P_2$ and $(P_1 \rightarrow P_2)$ are given or inferred, then we can infer $\text{NOT } P_1$ by the rule of "modus tollens"
 - If P_1 and $(P_1 \equiv P_2)$ are given or inferred, we can infer P_2
- Techniques
 - Prove tautologies using inference
 - Deductive proof

Proof with Resolution

- Resolution tautology - If we know $p \text{ OR } q$ and $p \rightarrow r$ then we can deduce $q \text{ OR } r$
 - $(p \text{ OR } q) \text{ AND } (\text{NOT } p \text{ OR } r) \rightarrow q \text{ OR } r$
- In order to apply resolution in a proof:
 - Express hypotheses and conclusion as a product of sums (conjunctive normal form), such as those that appear above
 - Each maxterm in the CNF of the hypothesis becomes a clause of the proof
 - Apply the resolution tautology to pairs of clauses, producing new clauses
 - Prove by producing all clauses of the conclusion OR prove by contradiction using resolution

Circuit Design

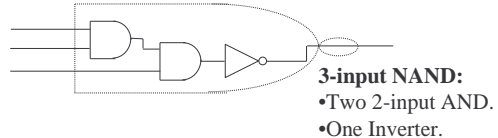
- Gate:
 - Basic electronic device.
 - Computes a Boolean function.



- AND, OR, NOT, NAND:
 - Easy to implement
 - Conceptually any number of inputs
 - Used in practice

Circuit Design

- Circuit:
 - A combination of gates
 - Output of some gates are the input of others.
 - Has one or more inputs:
 - These are inputs to the gates in the circuit.
 - May have one or more outputs.



Combinational & Sequential Circuits

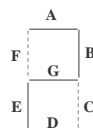
- Combinational:
 - **Output** is a **Boolean** function of **input** values.
 - Are **Acyclic**:
 - No cycles between inputs of a gate and its outputs.
 - **No memory**:
 - Cannot remember previous inputs or outputs.
 - Example of use:
 - Decode instructions and perform arithmetic.

Combinational & Sequential Circuits (2)

- Sequential:
 - Output depends on the **current** input values and the **previous** sequence of input values.
 - Are **Cyclic**:
 - Output of a gate feeds its input at some future time.
 - **Memory**:
 - Remember results of previous operations
 - Use them as inputs.
 - Example of use:
 - Build registers and memory units.

Combinational Circuit: Encoder for a 7-Segment Display

- Goal: Design a circuit...
 - With 10 inputs: $i_0, i_1, i_2, \dots, i_9$.
 - Each one corresponds to the decimal digits (0-9).
 - Lights up the display segments **A, B, C, ..., G**.
 - As needed to display the digit specified by the input.
 - Total: 7 outputs.



Number 2:

- Input $i_2 = 1$.
- $i_0, i_1, i_3, \dots, i_9 = 0$.
- Outputs:
 - $A=B=D=E=G=1$
 - $C=F=0$

Encoder for a 7-Segment Display (2)

- Boolean expression for the outputs:

$$A = i_0 + i_2 + i_3 + i_5 + i_7 + i_8 + i_9$$

$$B = i_0 + i_1 + i_2 + i_3 + i_4 + i_7 + i_8 + i_9$$

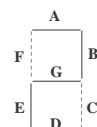
$$C = i_0 + i_1 + i_3 + i_4 + i_5 + i_6 + i_7 + i_8 + i_9$$

$$D = i_0 + i_2 + i_3 + i_5 + i_6 + i_8$$

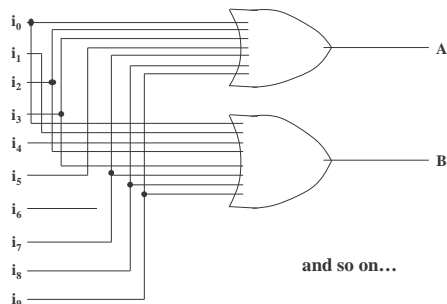
$$E = i_0 + i_2 + i_6 + i_8$$

$$F = i_0 + i_4 + i_5 + i_6 + i_8 + i_9$$

$$G = i_2 + i_3 + i_4 + i_5 + i_6 + i_8 + i_9$$
- Build the circuit with 7 OR gates:
 - One for each segment of the display



Encoder for a 7-Segment Display (3)



Constraints on Circuit Design

- Numerous constraints impact:
 - The **speed** and **cost** of a circuit.
- Speed:
 - Every gate in a circuit introduces a small delay.
 - Circuit delay depends on the number of gates between inputs and outputs
 - Depends on fan-in and fan-out of a gate

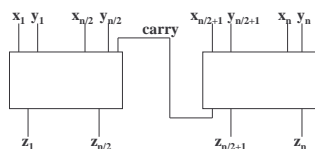
Constraints on Circuit Design

- Size limitations:
 - More gates lead to larger circuits.
 - Large circuits are more **expensive**
 - Higher failure rate.
 - And **slower**.
 - Signals must propagate from one end to the other.
- Fan-in and Fan-out:
 - Number of inputs and outputs of a gate.
 - Large fan-in makes a gate slower.

Divide and Conquer Adder

- Already seen *Ripple-Carry* adder
- Need:
 - Adder with a smaller delay for larger words.
- Solution:
 - Use a divide and conquer strategy.
 - Use two $N/2$ -bit adders and combine results.
 - Left and right halves added in parallel.

Divide and Conquer Adder (2)



- Carry: Not known in advance:
 - How can the adders operate in parallel?
 - Compute to sums for the upper half.
 - One assuming there is a carry.
 - One assuming there is NO carry.
 - Use additional circuit to select the correct sum.

Design of an N-adder

- Assume two N -bit operands: $x_1 \dots x_N$ & $y_1 \dots y_N$.
- Design N -adder that computes:
 - Sum **without** carry-in: $s_1 \dots s_N$.
 - Sum **with** carry-in: $t_1 \dots t_N$.
 - The **carry-propagate** bit, p :
 - It is 1 if there is a carry-out assuming there is carry-in.
 - The **carry-generate** bit, g :
 - It is 1 if there is a carry-out even if there is NO carry-in.
 - **NOTE:** if g is one then p will be one too (**g implies p**).
- First, build an 1-bit adder.

A 1-bit Adder

Boolean Functions

| x | y | s | t | p | g |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 1 |

Logical Expressions

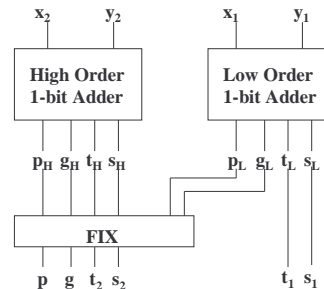
$$s = \bar{x}y + x\bar{y}$$

$$t = \bar{x}y + xy$$

$$p = x + y$$

$$g = xy$$

A 2-bit Adder from 1-bit Adders



“FIX” Circuit

- **Carry-propagate bit:** $p = p_H p_L + g_H$
 - If there is a carry-in **p** is 1 if:
 - Both the low and high order part propagate a carry ($p_H p_L$).
 - Or: The high order part generates a carry (g_H).
- **Carry-generate bit:** $g = g_H + g_L p_H$
 - If there is **NO** carry-in **g** is 1 if:
 - If the high order part generates a carry (g_H).
 - Or if there is a carry from the low part and the high part propagate that carry ($g_L p_H$).

“FIX” Circuit (2)

- **High order sum, NO carry-in:**
 - It is: $s_2 = s_H \bar{g}_L + t_H g_L$
 - s_H if there is no carry from low order part ($\sim g_L$).
 - t_H if there is carry from low order part (g_L).
- **High order sum, with carry-in:**
 - It is: $t_2 = s_H \bar{p}_L + t_H p_L$
 - t_H if there is a carry from the low order part.
 - s_H otherwise.

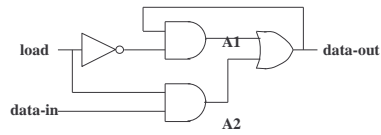
Sequential Circuits for Memory Elements.

- **Memory element:**
 - A collection of gates capable of producing its last input as output.
 - They are **sequential** circuits.
 - Their behavior depends on current and past inputs.
- **Flip-flop:**
 - A 1-bit memory element.
 - Typical flip-flop:
 - Takes two inputs (load and data-in).
 - Produces one output (data-out).

Flip-Flops

- **Load==0:**
 - The circuit produces the stored value as output.
- **Load==1:**
 - The circuit stores the value **data-in** and
 - Produces it as output.

Flip-Flop Circuit



- Load=1 \Rightarrow A1=0 \Rightarrow data-out=A2=data-in.
- Load=0 \Rightarrow A2=0 \Rightarrow data-out=A1
 - Which is the previously stored value.