Combinational Logic Design Case Studies

- General design procedure
- Examples
  - Calendar subsystem
  - BCD to 7-segment display controller
  - Process line controller
  - Logical function unit
- Arithmetic
  - Integer representations
  - Addition/Subtraction
  - Arithmetic/logic units

Calendar Subsystem

- Determine number of days in a month (to control watch display)
  - Used in controlling the display of a wrist-watch LCD screen
  - Inputs: month, leap year flag
  - Outputs: number of days
  - Use software implementation to help understand the problem

Choose Implementation Target and Perform Mapping

- Discrete gates
  - Table:
    | Month | Leap | 28 | 29 | 30 | 31 |
    |-------|------|----|----|----|----|
    | 0001  | 0    | 0  | 0  | 0  | 0  |
    | 0010  | 1    | 0  | 0  | 0  | 0  |
    | 0110  | 0    | 0  | 0  | 0  | 0  |
    | 1010  | 0    | 0  | 0  | 0  | 0  |
    | 1110  | 0    | 0  | 0  | 0  | 0  |
  - Can translate to S-o-P or P-o-S

General Design Procedure for Combinational Logic

1. Understand the Problem
   - What is the circuit supposed to do?
   - What are the inputs (data, control) and outputs
   - Draw block diagram or other picture
2. Formulate the Problem using a Suitable Design Representation
   - Truth table or waveform diagram are typical
   - May require encoding of symbolic inputs and outputs
3. Choose Implementation Target
   - ROM, PAL, PLA
   - Array, decoder and OR-gate
   - Discrete gates
4. Follow Implementation Procedure
   - K-maps for two-level, multi-level
   - Design tools and hardware description language (e.g., Verilog)

Formalize the Problem

- Encoding:
  - Binary number for month: 4 bits
  - 4 wires for 28, 29, 30 and 31
  - one-hot — only one true at any time
- Block diagram:

Choose Implementation Target and Perform Mapping

- Discrete gates:
  - Table:
    | Month | Leap | 28 | 29 | 30 | 31 |
    |-------|------|----|----|----|----|
    | 0001  | 0    | 0  | 0  | 0  | 0  |
    | 0010  | 1    | 0  | 0  | 0  | 0  |
    | 0110  | 0    | 0  | 0  | 0  | 0  |
    | 1010  | 0    | 0  | 0  | 0  | 0  |
    | 1110  | 0    | 0  | 0  | 0  | 0  |
  - Can translate to S-o-P or P-o-S

BCD to 7-segment display controller

- Understanding the problem
  - Inputs: a 4-bit code (A, B, C, D)
  - Output is the control signals for the display (7 outputs: 0-6)
- Block diagram:

Combinational Logic Design Case Studies
Formalize the problem

- Truth table
  - Show don’t care

- Choose implementation
  - Insert X to represent don’t care
  - Do any of Xs have to be used?
  - Don’t cares imply PAL/PLA may be attractive

- Follow implementation procedure
  - Minimization using K-maps

Implementation as Minimized Sum-of-Products

- 15 unique product terms when minimized individually

Implementation as Minimized S-o-P (cont’d)

- Can do better
  - 9 unique product terms (instead of 15)
  - Share terms among outputs
  - Each output not necessarily in minimized form

 PLA implementation

- Production Line Control
  - Rods of varying length (+/-10%) travel on conveyor belt
  - Mechanical arm pushes rods within spec (+/-5%) to one side
  - Second arm pushes rods too long to the other side
  - Rods that are too short stay on belt
  - 3 light barriers (light source + photosensitive) as sensors
  - Design combinational logic to activate the arms

- Understanding the problem
  - Inputs are three sensors
  - Outputs are two arm control signals
  - Assume sensor reads "1" when tripped, "0" otherwise
  - Call sensors A, B, C
Sketch of Problem

- Position of Sensors
  - A to B distance specification = 5%
  - A to C distance specification = 5%

Formalize the Problem

- Truth Table
  - Shows don't cares

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>do nothing</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>do nothing</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>do nothing</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>do nothing</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>too short</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>too long</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>too long</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>too long</td>
</tr>
</tbody>
</table>

Logic implementation now straightforward
o Direct from Input AND gates

Logical Function Unit

- Multi-purpose Function Block
  - 3 control inputs to specify operation to perform on operands
  - 2 data inputs for operands
  - 1 output of the same bit-width as operands

<table>
<thead>
<tr>
<th>B</th>
<th>C</th>
<th>A</th>
<th>Function</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>A + B</td>
<td>Logic OR</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>A + B</td>
<td>Logic NAND</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>A + B</td>
<td>Logic AND</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>A + B</td>
<td>Logic OR</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>A + B</td>
<td>Logic AND</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>A + B</td>
<td>Logic OR</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>A + B</td>
<td>Logic AND</td>
</tr>
</tbody>
</table>

Formalize the Problem

- choose implementation technology

Arithmetic Circuits

- Excellent Examples of Combinational Logic Design
  - Time vs. Space Trade-offs
    - Doing things fast may require more logic and thus more space
    - Example: carry look-ahead logic
  - Arithmetic and Logic Units
    - General-purpose building blocks
    - Critical components of processor datapaths
    - Used within most computer instructions

Number Systems

- Representation of positive numbers is the same in most systems
- Major differences are in how negative numbers are represented
- Representation of negative numbers come in three major schemes
  - Sign and magnitude
  - 1's complement
  - 2's complement
- Assumptions
  - We'll assume a 4-bit machine word
  - 16 different values can be represented
  - Roughly half are positive, half are negative
Sign and Magnitude

- One bit dedicate to sign (positive or negative)
- 0 sign: 0 is positive or zero, 1 is negative

- Rest represent the absolute value of magnitude
- Three lowest order bits: 0000-7 (011)

- Range for n bits
- \( n = 2^{n-1} \) (two representations for 0)

- Cumbersome addition/subtraction
- Must compare magnitudes to determine sign of result

1's Complement

- If \( N \) is a positive number, then the negative of \( N \) (its 1's complement or \( N' \)) is \( N' = 2^n - N \)
- Example: 1's complement of 7
  \[
  2^3 = 0000 \\
  1 = 0001 \\
  2^2 - 1 = 1111 \\
  7 = 0111 \quad \rightarrow \text{is 1's complement form}
  \]

- Shortcut: simply compute bit-wise complement (0111 \( \rightarrow \) 0000)

2's Complement

- 2's complement with negative numbers shifted one position clockwise
- Only one representation for 0
- One more negative number than positive number
- High-order bit can act as sign bit


2's Complement Addition and Subtraction

- Simple Addition and Subtraction
- Simple scheme makes 2's complement the virtually unanimous choice for integer number systems in computers

- Example: 2's complement of 7
  \[
  2^3 = 0000 \\
  7 = 0111 \\
  \text{subtract } 0001 = \text{rep of } 7
  \]

- Example: 2's complement of -7
  \[
  2^3 = 0000 \\
  7 = 0111 \\
  \text{subtract } 1001 = \text{rep of } 7
  \]

- Shortcut: 2's complement = bit-wise complement + 1
  \[
  0011 = 0000 + 1 = 0011 \quad \text{(representation of -7)} \\
  0000 + 0110 = 0111 \quad \text{(representation of 7)}
  \]
Why Can the Carry-out be Ignored?

- Can't ignore it completely
- Needed to check for overflow (see next two slides)
- When there is no overflow, carry-out may be true but can't be ignored

\[ M + N \quad \text{when } N > M: \]
\[ M^* + N = (2^n - M) + N = 2^n - (N - M) \]

ignoring carry-out is just like subtracting 2^n

\[ M = -N \quad \text{where } N > M \leq 2^n - 1 \]
\[ (-M)^* + (-N)^* + N^* = (2^n - M) + (2^n - N) = 2^n - (M + N) + 2^n \]

ignoring the carry, it is just the 2's complement representation for \(-M^* N^*\)

Overflow in 2's Complement Addition/Subtraction

- Overflow conditions
  - Add two positive numbers to get a negative number
  - Add two negative numbers to get a positive number

\[ \begin{array}{c|c|c|c|c|c}
    \text{Overflow} & \text{Add A} & \text{Add B} & \text{Sum} & \text{Carry} \\
    \hline
    \text{No} & 0 & 0 & 0 & 0 \\
    \text{Yes} & 1 & 1 & 1 & 1 \\
\end{array} \]

Circuits for Binary Addition

- Half adder (add 2 1-bit numbers)
  - Sum = \( A' B + A B' = A \oplus B \)
  - Carry = \( A B \)

- Full adder (carry-in to cascade for multi-bit adders)
  - Sum = \( G + A C + B G + (A + B) \cdot A B \)
  - Carry = \( G + C A + B G + (A + B) \cdot A B \)

Full adder implementations

- Standard approach
  - 6 gates
  - 2 XORs, 2 ANDs, 2 ORs

- Alternative implementation
  - 5 gates
  - Half adder is an XOR gate and AND gate
  - 2 XORs, 2 ANDs, 1 OR

Adder/Subtractor

- Use an adder to do subtraction thanks to 2's complement representation
  - \( A - B = A + (-B) = A + B' + 1 \)
  - Control signal selects \( B \) or 2's complement of \( B \)
Ripple-Carry Adders

Critical Delay
- The propagation of carry from low to high order stages

![Diagram of Ripple-Carry Adders](image)

Ripple-Carry Adders (cont'd)

Critical delay
- The propagation of carry from low to high order stages
- $\begin{array}{ll} \text{III} & \text{XXXX} \end{array}$ is the worst case addition
- Carry must propagate through all bits

![Graph of Ripple-Carry Adders](image)

Carry-Lookahead Logic

Carry generate: $G_i = A_i B_i$
- Must generate carry when $A = B = 1$

Carry propagate: $P_i = A_i \oplus B_i$
- Carry-in will equal carry-out here

Sum and Carry can be re-expressed in terms of generate/propagate:
- $G = A_i \oplus B_i \oplus G_i$
- $P = A_i \oplus B_i \oplus (A_i \oplus B_i) = A_i \oplus B_i \oplus G_i$

Carry-Lookahead Logic (cont'd)

Re-express the carry logic as follows:
- $C_1 = \overline{C_0}$
- $C_2 = \overline{C_1} + P_1 + P_0 C_0$
- $C_3 = \overline{C_2} + P_2 C_2 + P_2 C_1 + P_2 P_0 C_0$
- $C_4 = \overline{C_3} + P_3 C_3 + P_3 C_2 + P_3 P_2 C_2 + P_3 P_2 C_1 + P_3 P_2 C_0 + P_3 P_2 P_1 C_0$

Each of the carry equations can be implemented with two-level logic:
- All inputs are now directly derived from data inputs and not from intermediate carries
- This allows computation of all sum outputs to proceed in parallel

Carry-Lookahead Implementation

Adder with propagate and generate outputs

Carry-Lookahead Implementation (cont'd)

Carry-lookahead logic generates individual carries:
- Sums computed much more quickly in parallel
- However, cost of carry logic increases with more stages

![Diagram of Carry-Lookahead Implementation](image)
Summary for Examples of Combinational Logic

- Combinational logic design process
  1. Formalize problem encoding: truth-table, equations
  2. Choose implementation tech (COIL, PAL, PLA, discrete gates)
  3. Implement by following the design procedure for that technology

- Binary number representation
  1. Positive numbers the same
  2. Difference is in how negative numbers are represented
  3. 2's complement easiest to handle: one representation for zero, slightly complicated implementation simple addition

- Circuits for binary addition
  1. Basic half-adder and full-adder
  2. Carry look-ahead logic
  3. Carry-select

- ALU Design
  1. Specification, implementation