Combinational logic design case studies

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

General design procedure for combinational logic

- 1. Understand the problem
  - What is the circuit supposed to do?
  - Write down 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
  - MUX, 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)
BCD to 7-segment display controller

**Understanding the problem**
- Input is a 4 bit BCD digit (A, B, C, D)
- Output is the control signals for the display (7 outputs C0 – C6)

**Block diagram**

```
<table>
<thead>
<tr>
<th></th>
<th>C0</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
</tr>
</thead>
<tbody>
<tr>
<td>C5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

00239

**Formalize the problem**

**Truth table**
- Show don't cares

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>C0</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Choose implementation target**
- If ROM, we are done
- Don't cares imply PAL/PLA may be attractive

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>C0</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Follow implementation procedure**
- Minimization using K-maps

1 1 - - - - - - - -
Implementation as minimized sum-of-products

**15 unique product terms when minimized individually**

\[ C_0 = A + B \bar{D} + C + B' \bar{D}' \]
\[ C_1 = C' \bar{D} + C \bar{D} + B' \]
\[ C_2 = B + C' + D \]
\[ C_3 = B' \bar{D}' + C \bar{D}' + B C' \bar{D} + B' C \]
\[ C_4 = B' \bar{D}' + C \bar{D}' \]
\[ C_5 = A + C' \bar{D}' + B \bar{D}' + B C \]
\[ C_6 = A + C D' + B C' + B' C \]

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

\[ C_0 = A + B D + C + B' D' \]
\[ C_1 = C' D' + C D + B' \]
\[ C_2 = B + C' + D \]
\[ C_3 = B' D' + C \bar{D}' + B C' \bar{D} + B' C \]
\[ C_4 = B' D' + C \bar{D}' \]
\[ C_5 = A + C' D' + B D' + B C \]
\[ C_6 = A + C D' + B C' + B' C \]
**PLA implementation**

![Diagram of PLA implementation]

**PAL implementation/Discrete gate implementation**

- **Limit of 4 product terms per output**
  - decomposition of functions with larger number of terms
  - do not share terms in PAL anyway
    (although there are some with some shared terms)
    
    \[
    C_0 = C_3 + A' B' X' + A D Y
    \]
    
    \[
    C_1 = Y + A'C_5' + C' D' C_6
    \]
    
    \[
    C_2 = C_5 + A' B' D + A' C D
    \]
    
    \[
    X = C' + D'
    \]
    
    \[
    C_3 = C_4 + B D C_5 + A' B' X'
    \]
    
    \[
    Y = B' C'
    \]
    
    \[
    C_4 = C' Y + A' C' D'
    \]
    
    \[
    C_5 = C' C_4 + A Y + A' B X
    \]
    
    \[
    C_6 = A C_4 + C C_5 + C_4' C_5 + A' B' C
    \]
    
    \[
    W = C D + B C D'
    \]
    
    **decompose into multi-level logic (hopefully with CAD support)**
    
    - find common sub-expressions among functions
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>C0</th>
<th>C1</th>
<th>C2</th>
<th>Function</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>always 1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>A + B</td>
<td>logical OR</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>(A • B)'</td>
<td>logical NAND</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>A xor B</td>
<td>logical xor</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>A xnor B</td>
<td>logical xnor</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>A • B</td>
<td>logical AND</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>(A + B)'</td>
<td>logical NOR</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>always 0</td>
</tr>
</tbody>
</table>

Formalize the problem

choose implementation technology
5-variable K-map to discrete gates
multiplexor implementation

Autumn 2000 CSE370 - V - Combinational Logic Case Studies
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 other side
  - rods that are too short stay on belt
  - 3 light barriers (light source + photocell) 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**

- show don't cares

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Function</th>
<th>Logic implementation now straightforward</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>do nothing</td>
<td>just use three 3-input AND gates</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>do nothing</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>do nothing</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>do nothing</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>too short</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>don't care</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>in spec</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>too long</td>
<td></td>
</tr>
</tbody>
</table>

"too short" = ABC (only first sensor tripped)

"in spec" = ABC (first two sensors tripped)

"too long" = ABC (all three sensors tripped)

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**

```c
integer number_of_days ( month, leap_year_flag ) { switch (month) { case 1: return (31); case 2: if (leap_year_flag == 1) then return (29) else return (28); case 3: return (31); case 4: return (30); case 5: return (31); case 6: return (30); case 7: return (31); case 8: return (31); case 9: return (30); case 10: return (31); case 11: return (30); case 12: return (31); default: return (0); } }
```
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

<table>
<thead>
<tr>
<th>Month</th>
<th>Leap</th>
<th>28</th>
<th>29</th>
<th>30</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0001</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0010</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0011</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0100</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0101</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0110</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0111</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1000</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1001</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1010</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1011</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1100</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1101</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>111-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Block diagram:**

```
+---+---+---+---+
|   |   |   |   |
+---+---+---+---+
|   |   |   |   |
+---+---+---+---+
|   |   |   |   |
+---+---+---+---+
| 28 | 29 | 30 | 31 |
```

Choose implementation target and perform mapping

**Discrete gates**

- \( 28 = m_8' m_4' m_2 m_1' \text{leap}' \)
- \( 29 = m_8' m_4' m_2 m_1' \text{leap} \)
- \( 30 = m_8' m_4 m_1' + m_8 m_1 \)
- \( 31 = m_8' m_1 + m_8 m_1' \)

**Can translate to S-o-P or P-o-S**

<table>
<thead>
<tr>
<th>Month</th>
<th>Leap</th>
<th>28</th>
<th>29</th>
<th>30</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0001</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0010</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0011</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0100</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0101</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0110</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0111</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1000</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1001</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1010</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1011</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1100</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1101</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>111-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Leap year flag

- Determine value of leap year flag given the year
  - For years after 1582 (Gregorian calendar reformation),
  - leap years are all the years divisible by 4,
  - except that years divisible by 100 are not leap years,
  - but years divisible by 400 are leap years.

- Encoding the year:
  - binary – easy for divisible by 4,
    - but difficult for 100 and 400 (not powers of 2)
  - BCD – easy for 100,
    - but more difficult for 4, what about 400?

- Parts:
  - construct a circuit that determines if the year is divisible by 4
  - construct a circuit that determines if the year is divisible by 100
  - construct a circuit that determines if the year is divisible by 400
  - combine the results of the previous three steps to yield the leap year flag

Activity: divisible-by-4 circuit

- BCD coded year
  - YM8 YM4 YM2 YM1 – YH8 YH4 YH2 YH1 – YT8 YT4 YT2 YT1 – YO8 YO4 YO2 YO1

- Only need to look at low-order two digits of the year
  - all years ending in 00, 04, 08, 12, 16, 20, etc., are divisible by 4
  - if tens digit is even, then divisible by 4 if ones digit is 0, 4, or 8
  - if tens digit is odd, then divisible by 4 if the ones digit is 2 or 6.

- Translates into the following Boolean expression
  (where YT1 is the year's tens digit low-order bit,
  YO8 is the high-order bit of year's ones digit, etc.):

  \[\begin{align*}
  & \text{YT1}' (YO8' YO4' YO2 YO1' + YO8' YO4 YO2 YO1' + YO8 YO4 YO2' YO1') \\
  & + \text{YT1} (YO8' YO4' YO2 YO1' + YO8 YO4 YO2' YO1')
\end{align*}\]

- Digits with values of 10 to 15 will never occur, simplify further to yield:

  \[\text{YT1} YO2' YO1' + YT1 YO2 YO1'\]
Divisible-by-100 and divisible-by-400 circuits

Divisible-by-100 just requires checking that all bits of two low-order digits are all 0:

\[ YT8' YT4' YT2' YT1' \]

- \[ YO8' YO4' YO2' YO1' \]

Divisible-by-400 combines the divisible-by-4 (applied to the thousands and hundreds digits) and divisible-by-100 circuits

\[ (YM1' YH2' YH1' + YM1 YH2 YH1') \]

- \[ (YT8' YT4' YT2' YT1' • YO8' YO4' YO2' YO1') \]

Combining to determine leap year flag

Label results of previous three circuits: \( D4, D100, \) and \( D400 \)

\[ \text{leap\_year\_flag} = D4 (D100 \cdot D400')' \]

\[ = D4 \cdot D100' + D4 \cdot D400 \]

\[ = D4 \cdot D100' + D400 \]
Implementation of leap year flag

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 lookahead 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
  - 1s complement
  - 2s 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)
  - sign: 0 = positive (or zero), 1 = negative
  - 0 100 = +4
  - 1 100 = -4
- Rest represent the absolute value or magnitude
  - three low order bits: 0 (000) thru 7 (111)
- Range for n bits
  - +/- 2^(n-1) – 1 (two representations for 0)
- Cumbersome addition/subtraction
  - must compare magnitudes to determine sign of result
1s complement

**If N is a positive number, then the negative of N (1s complement or N’) is N’ = (2n-1) - N**

- example: 1s complement of 7

\[
\begin{align*}
2^4 & = 10000 \\
1 & = 00001 \\
2^4 - 1 & = 1111 \\
7 & = 0111 \\
1000 & = -7 \text{ in 1s complement form}
\end{align*}
\]

**shortcut: simply compute bit-wise complement (0111 -> 1000)**

1s complement (cont’d)

**Subtraction implemented by 1s complement and then addition**

**Two representations of 0**

- causes some complexities in addition

**High-order bit can act as sign bit**

0 100 = +4

1 011 = -4
2s complement

- 1s 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

\[
\begin{align*}
0 \ 100 &= +4 \\
1 \ 100 &= -4
\end{align*}
\]

2s complement (cont’d)

- If \( N \) is a positive number, then the negative of \( N \) (its 2s complement or \( N^* \)) is \( N^* = 2^n - N \)
- example: 2s complement of 7
  \[
  4 = 10000 \\
  \text{subtract } 7 = \underline{0111}
  \]
- example: 2s complement of -7
  \[
  4 = 10000 \\
  \text{subtract } -7 = \underline{1001}
  \]
- shortcut: 2s complement = bit-wise complement + 1
  - \(0111 \rightarrow 1000 + 1 \rightarrow 1001 \) (representation of -7)
  - \(1001 \rightarrow 0110 + 1 \rightarrow 0111 \) (representation of 7)
2s complement addition and subtraction

Simple addition and subtraction

Simple scheme makes 2s complement the virtually unanimous choice for integer number systems in computers

<table>
<thead>
<tr>
<th></th>
<th>0100</th>
<th>1100</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td>-4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0111</th>
<th>1101</th>
</tr>
</thead>
<tbody>
<tr>
<td>+3</td>
<td></td>
<td>+(-3)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0111</th>
<th>-7</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td></td>
<td>1101</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0100</th>
<th>1100</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td>-4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1101</th>
<th>0011</th>
</tr>
</thead>
<tbody>
<tr>
<td>-3</td>
<td></td>
<td>+3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>10001</th>
<th>-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>1111</td>
</tr>
</tbody>
</table>

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 be ignored

\[-M + N \text{ when } N > M:\]

\[M^* + N = (2n - M) + N = 2n + (N - M)\]

ignoring carry-out is just like subtracting \(2n\)

\[-M + -N \text{ where } N + M \leq 2n-1\]

\[(-M) + (-N) = M^* + N^* = (2n - M) + (2n - N) = 2n - (M + N) + 2n\]

ignoring the carry, it is just the 2s complement representation for \(- (M + N)\)
Overflow in 2s complement addition/subtraction

**Overflow conditions**

- add two positive numbers to get a negative number
- add two negative numbers to get a positive number

Overflow conditions

**Overflow when carry into sign bit position is not equal to carry-out**

<table>
<thead>
<tr>
<th></th>
<th>0111</th>
<th>0100</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0101</td>
<td>-7</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>-2</td>
</tr>
<tr>
<td>-8</td>
<td>1000</td>
<td>7</td>
</tr>
<tr>
<td>overflow</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0000</th>
<th>1111</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0101</td>
<td>1110</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>1011</td>
</tr>
<tr>
<td>-8</td>
<td>1111</td>
<td>1100</td>
</tr>
<tr>
<td>no overflow</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Circuits for binary addition

**Half adder (add 2 1-bit numbers)**
- Sum = Ai' Bi + Ai Bi' = Ai \oplus Bi
- Cout = Ai Bi

**Full adder (carry-in to cascade for multi-bit adders)**
- Sum = Ci \oplus A \oplus B
- Cout = B Ci + A Ci + A B = Ci (A + B) + A B

<table>
<thead>
<tr>
<th>Ai</th>
<th>Bi</th>
<th>Si</th>
<th>Cout</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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

Late arriving signal:
- Two gate delays to compute \( \text{Cout} \)
Ripple-carry adders (cont'd)

- Critical delay
  - the propagation of carry from low to high order stages
  - 1111 + 0001 is the worst case addition
  - carry must propagate through all bits

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 Cout can be re-expressed in terms of generate/propagate:
  - \( S_i = A_i \oplus B_i \oplus C_i \)
  - \( = P_i \oplus C_i \)
  - \( C_i + 1 = A_i B_i + A_i C_i + B_i C_i \)
  - \( = A_i B_i + C_i (A_i + B_i) \)
  - \( = A_i B_i + C_i (A_i \oplus B_i) \)
  - \( = G_i + C_i P_i \)
**Carry-lookahead logic (cont’d)**

- Re-express the carry logic as follows:
  - \( C_1 = G_0 + P_0 C_0 \)
  - \( C_2 = G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_0 \)
  - \( C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0 \)
  - \( C_4 = G_3 + P_3 C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0 \)

- 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

- Increasingly complex logic for carries
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

Carry-lookahead adder

- 4 four-bit adders with internal carry lookahead
- Second level carry lookahead unit extends lookahead to 16 bits
**Carry-select adder**

- **Redundant hardware to make carry calculation go faster**
  - Compute two high-order sums in parallel while waiting for carry-in
  - One assuming carry-in is 0 and another assuming carry-in is 1
  - Select correct result once carry-in is finally computed

---

**Arithmetic logic unit design specification**

<table>
<thead>
<tr>
<th>M = 0, logical bitwise operations</th>
<th>S1 S0</th>
<th>Function</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0</td>
<td>F = Ai</td>
<td>Input Ai transferred to output</td>
</tr>
<tr>
<td>0 1</td>
<td>0</td>
<td>F = not Ai</td>
<td>Complement of Ai transferred to output</td>
</tr>
<tr>
<td>1 0</td>
<td>0</td>
<td>F = Ai xor Bi</td>
<td>Compute XOR of Ai, Bi</td>
</tr>
<tr>
<td>1 1</td>
<td>1</td>
<td>F = Ai xnor Bi</td>
<td>Compute XNOR of Ai, Bi</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>M = 1, C0 = 0, arithmetic operations</th>
<th>S1 S0</th>
<th>Function</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0</td>
<td>F = A</td>
<td>Input A passed to output</td>
</tr>
<tr>
<td>0 1</td>
<td>1</td>
<td>F = not A</td>
<td>Complement of A passed to output</td>
</tr>
<tr>
<td>1 0</td>
<td>0</td>
<td>F = A plus B</td>
<td>Sum of A and B</td>
</tr>
<tr>
<td>1 1</td>
<td>1</td>
<td>F = (not A) plus B</td>
<td>Sum of B and complement of A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>M = 1, C0 = 1, arithmetic operations</th>
<th>S1 S0</th>
<th>Function</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0</td>
<td>F = A plus 1</td>
<td>Increment A</td>
</tr>
<tr>
<td>0 1</td>
<td>1</td>
<td>F = (not A) plus 1</td>
<td>Two's complement of A</td>
</tr>
<tr>
<td>1 0</td>
<td>0</td>
<td>F = A plus B plus 1</td>
<td>Increment sum of A and B</td>
</tr>
<tr>
<td>1 1</td>
<td>1</td>
<td>F = (not A) plus B plus 1</td>
<td>B minus A</td>
</tr>
</tbody>
</table>

Logical and arithmetic operations

Not all operations appear useful, but "fall out" of internal logic
Arithmetic logic unit design (cont’d)

**Sample ALU – truth table**

<table>
<thead>
<tr>
<th>M</th>
<th>S1</th>
<th>S0</th>
<th>Ci</th>
<th>A</th>
<th>B</th>
<th>Fi</th>
<th>Cb+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Arithmetic logic unit design (cont’d)

**Sample ALU – multi-level discrete gate logic implementation**

12 gates
Arithmetic logic unit design (cont'd)

Sample ALU – clever multi-level implementation

- **first-level gates**
  - Use $S_0$ to complement $A_i$:
    - $S_0 = 0$ causes gate $X_1$ to pass $A_i$
    - $S_0 = 1$ causes gate $X_1$ to pass $A_i'$
  - Use $S_1$ to block $B_i$:
    - $S_1 = 0$ causes gate $A_1$ to make $B_i$ go forward as 0 (don't want $B_i$ for operations with just $A$)
    - $S_1 = 1$ causes gate $A_1$ to pass $B_i$
  - Use $M$ to block $C_i$:
    - $M = 0$ causes gate $A_2$ to make $C_i$ go forward as 0 (don't want $C_i$ for logical operations)
    - $M = 1$ causes gate $A_2$ to pass $C_i$

- **other gates**
  - For $M=0$ (logical operations, $C_i$ is ignored):
    - $F_i = S_1 B_i \oplus (S_0 \oplus A_i)$
    - $F_i = S_1 S_0' (A_i) + S_1 S_0 (A_i') + S_1 S_0' (A_i B_i) + S_1 S_0 (A_i' B_i)$
  - For $M=1$ (arithmetic operations):
    - $F_i = S_1 B_i \oplus ((S_0 \oplus A_i) \oplus C_i)$
    - $C_{i+1} = C_i + S_1 (S_0 \oplus A_i) + S_1 B_i ((S_0 \oplus A_i) \oplus C_i)$
    - Just a full adder with inputs $S_0 \oplus A_i, S_1 B_i,$ and $C_i$

Summary for examples of combinational logic

- **Combinational logic design process**
  - **Formalize problem**: encodings, truth-table, equations
  - **Choose implementation technology**: ROM, PAL, PLA, discrete gates
  - **Implement by following the design procedure for that technology**

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

- **Circuits for binary addition**
  - **Basic half-adder and full-adder**
  - **Carry lookahead logic**
  - **Carry-select**

- **ALU Design**
  - **Specification, implementation**