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

### Formalize the problem

- Truth table
  - show don't cares
- Choose implementation target
  - if ROM, we are done
  - don't cares imply PAL/PLA may be attractive
- Follow implementation procedure
  - minimization using K-maps

<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>
<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>0</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>0</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>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Implementation as minimized sum-of-products

- 15 unique product terms when minimized individually

\[
\begin{align*}
C_0 &= A + B D + C + B' D' \\
C_1 &= C' D' + C D + B' \\
C_2 &= B + C' + D \\
C_3 &= B' D' + C D' + B C' D + B' C \\
C_4 &= B' D' + C D' \\
C_5 &= A + C' D' + B D' + B C' \\
C_6 &= A + C D' + B C' + B' C
\end{align*}
\]

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

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

Autumn 2003  

PAL implementation vs.  
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)
    \[ C_2 = B + C + D \]
    \[ C_2 = B' D + B C' D + C' D' + C D + B C D' \]
    \[ C_2 = B' D + B C' D + C' D' + W \]
    \[ W = C D + B C D' \]
  - decompose into multi-level logic (hopefully with CAD support)
  - find common sub-expressions among functions
    \[ C_0 = C_3 + A' B X' + A D Y \]
    \[ C_1 = Y + A' C_5 + C' D' C_6 \]
    \[ C_2 = C_5 + A' B' D + A' C D \]
    \[ C_3 = C_4 + B D C_5 + A' B' X' \]
    \[ C_4 = D' Y + A' C D' \]
    \[ C_5 = C' C_4 + A Y + A B X \]
    \[ C_6 = A C_4 + C C_5 + C' C_5 + A' B' C \]
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>$C_0$</th>
<th>$C_1$</th>
<th>$C_2$</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>$(A \cdot B)'$</td>
<td>logical NAND</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>$A \cdot B$</td>
<td>logical AND</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$A \oplus B$</td>
<td>logical XOR</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>$A + B$</td>
<td>logical NOR</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>always 0</td>
</tr>
</tbody>
</table>

Formalize the problem

<table>
<thead>
<tr>
<th>$C_0$</th>
<th>$C_1$</th>
<th>$C_2$</th>
<th>$A$</th>
<th>$B$</th>
<th>$F$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</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>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</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>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Choose implementation technology

- 5-variable K-map to discrete gates
- Multiplexor 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 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>
</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>don’t care</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>in spec</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>too long</td>
</tr>
</tbody>
</table>

logic implementation now straightforward
just use three 3-input AND gates

"too short" = AB'C'
(only first sensor tripped)

"in spec" = A B C'
(first two sensors tripped)

"too long" = A B C
(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

- Block diagram:

Choose implementation target
and perform mapping

- Discrete gates
  - 28 = m8’ m4’ m2 m1’ leap’
  - 29 = m8’ m4’ m2 m1’ leap
  - 30 = m8’ m4 m1’ + m8 m1
  - 31 = m8’ m1 + m8 m1’

- Can translate to S-o-P or P-o-S
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
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:
  \[ YT8' \ YT4' \ YT2' \ YT1' \cdot \ YO8' \ YO4' \ YO2' \ YO1' \]

- Divisible-by-400 combines the divisible-by-4 (applied to the thousands and hundreds digits) and divisible-by-100 circuits
  \[ (YM1' \ YH2' \ YH1' + YM1 \ YH2 \ YH1') \]
  \[ \cdot (YT8' \ YT4' \ YT2' \ YT1' \cdot \ YO8' \ YO4' \ YO2' \ YO1') \]

Combining to determine leap year flag

- Label results of previous three circuits: D4, D100, and D400
  \[
  \text{leap\_year\_flag} \quad = \quad D4 \ (D100 \cdot D400') '
  \]
  \[
  = \quad D4 \cdot D100' + D4 \cdot D400
  \]
  \[
  = \quad D4 \cdot D100' + D400
  \]
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
  - Rest represent the absolute value or magnitude
    - three low order bits: 0 (000) thru 7 (111)
  - Range for n bits
    - +/- 2n–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 \( N \) (its 1s complement or \( N' \)) is \( N' = (2^n - 1) - N \)
- 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

\[
\begin{align*}
0 \ 100 & = +4 \\
1 \ 011 & = -4
\end{align*}
\]
**2s complement**

- 1s complement with negative numbers shifted one position clockwise
  - only one representation for 0
  - one more negative number than positive numbers
  - high-order bit can act as sign bit

| 0 100 = +4 |
| 1 100 = -4 |

**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
    - \( 2^4 = 10000 \)
    - subtract 7 = 0111
    - \( 1001 = \) repr. of -7
  - example: 2s complement of -7
    - \( 2^4 = 10000 \)
    - subtract -7 = 1001
    - 0111 = repr. of 7

- shortcut: 2s complement = bit-wise complement + 1
  - 0111 -> 1000 + 1 -> 1001 (representation of -7)
  - 1001 -> 0110 + 1 -> 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

\[
\begin{align*}
4 & \quad 0100 & -4 & \quad 1100 \\
+3 & \quad 0011 & \quad +(-3) & \quad 1101 \\
7 & \quad 0111 & -7 & \quad 11001 \\
\end{align*}
\]

\[
\begin{align*}
4 & \quad 0100 & -4 & \quad 1100 \\
-3 & \quad 1101 & \quad +3 & \quad 0011 \\
1 & \quad 10001 & -1 & \quad 1111 \\
\end{align*}
\]

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
  - \(-M + N\) when \(N > M\):
    \[
    M^* + N = (2^n - M) + N = 2^n + (N - M)
    \]
    ignoring carry-out is just like subtracting \(2^n\)
  - \(-M + -N\) where \(N + M \leq 2^n - 1\)
    \[
    (-M) + (-N) = M^* + N^* = (2^n - M) + (2^n - N) = 2^n - (M + N) + 2n
    \]
    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 when carry into sign bit position is not equal to carry-out

<table>
<thead>
<tr>
<th>+1</th>
<th>+2</th>
<th>+3</th>
<th>+4</th>
<th>+5</th>
<th>+6</th>
<th>+7</th>
<th>+8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1101</td>
<td>1110</td>
<td>0001</td>
<td>0010</td>
<td>0101</td>
<td>0111</td>
<td>1011</td>
<td>1100</td>
</tr>
<tr>
<td>+1</td>
<td>+2</td>
<td>+3</td>
<td>+4</td>
<td>+5</td>
<td>+6</td>
<td>+7</td>
<td>+8</td>
</tr>
<tr>
<td>1101</td>
<td>1110</td>
<td>0001</td>
<td>0010</td>
<td>0101</td>
<td>0111</td>
<td>1011</td>
<td>1100</td>
</tr>
</tbody>
</table>

Overflow conditions

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

<table>
<thead>
<tr>
<th>+1</th>
<th>+2</th>
<th>+3</th>
<th>+4</th>
<th>+5</th>
<th>+6</th>
<th>+7</th>
<th>+8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1101</td>
<td>1110</td>
<td>0001</td>
<td>0010</td>
<td>0101</td>
<td>0111</td>
<td>1011</td>
<td>1100</td>
</tr>
<tr>
<td>+1</td>
<td>+2</td>
<td>+3</td>
<td>+4</td>
<td>+5</td>
<td>+6</td>
<td>+7</td>
<td>+8</td>
</tr>
<tr>
<td>1101</td>
<td>1110</td>
<td>0001</td>
<td>0010</td>
<td>0101</td>
<td>0111</td>
<td>1011</td>
<td>1100</td>
</tr>
</tbody>
</table>

5 + 3 = –8

Overflow conditions
Circuits for binary addition

- **Half adder** (add 2 1-bit numbers)
  - Sum = \( A_i' B_i + A_i B_i' = A_i \text{ xor } B_i \)
  - Cout = \( A_i B_i \)

- **Full adder** (carry-in to cascade for multi-bit adders)
  - Sum = \( C_i \text{ xor } A \text{ xor } B \)
  - Cout = \( B C_i + A C_i + A B = C_i (A + B) + A B \)

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

\[
\text{Cout} = A B + C_i (A \text{ xor } B) = A B + C_i \text{ xor } A \text{ xor } B
\]
Adder/subtractor

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

Ripple-carry adders

- Critical delay
  - the propagation of carry from low to high order stages
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

- Ripple-carry adders (cont’d)

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 with cascaded carry-lookahead logic

- 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**

### M = 0, logical bitwise operations

<table>
<thead>
<tr>
<th>S1 S0</th>
<th>Function</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>F = Ai</td>
<td>input Ai transferred to output</td>
</tr>
<tr>
<td>0 1</td>
<td>F = not Ai</td>
<td>complement of Ai transferred to output</td>
</tr>
<tr>
<td>1 0</td>
<td>F = Ai xor Bi</td>
<td>compute XOR of Ai, Bi</td>
</tr>
<tr>
<td>1 1</td>
<td>F = Ai xnor Bi</td>
<td>compute XNOR of Ai, Bi</td>
</tr>
</tbody>
</table>

### M = 1, C0 = 0, arithmetic operations

<table>
<thead>
<tr>
<th>S1 S0</th>
<th>Function</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>F = A</td>
<td>input A passed to output</td>
</tr>
<tr>
<td>0 1</td>
<td>F = not A</td>
<td>complement of A passed to output</td>
</tr>
<tr>
<td>1 0</td>
<td>F = A plus B</td>
<td>sum of A and B</td>
</tr>
<tr>
<td>1 1</td>
<td>F = (not A) plus B</td>
<td>sum of B and complement of A</td>
</tr>
</tbody>
</table>

### M = 1, C0 = 1, arithmetic operations

<table>
<thead>
<tr>
<th>S1 S0</th>
<th>Function</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>F = A plus 1</td>
<td>increment A</td>
</tr>
<tr>
<td>0 1</td>
<td>F = (not A) plus 1</td>
<td>twos complement of A</td>
</tr>
<tr>
<td>1 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>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>G</th>
<th>A1</th>
<th>B1</th>
<th>F1</th>
<th>Co</th>
<th>F2</th>
<th>Co1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- 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 S0 to complement Ai
S0 = 0 causes gate X1 to pass Ai
S0 = 1 causes gate X1 to pass Ai’
use S1 to block Bi
S1 = 0 causes gate A1 to make Bi go forward as 0
(don’t want Bi for operations with just A)
S1 = 1 causes gate A1 to pass Bi
use M to block Ci
M = 0 causes gate A2 to make Ci go forward as 0
(don’t want Ci for logical operations)
M = 1 causes gate A2 to pass Ci

other gates
for M=0 (logical operations, Ci is ignored)
Fi = S1 Bi xor (S0 xor Ai)
= S1 S0' (Ai') + S1 S0 (Ai') + S1 S0' (Ai Bi') + S1 S0 (Ai' Bi' + Ai Bi)

for M=1 (arithmetic operations)
Fi = S1 Bi xor ( (S0 xor Ai) xor Ci )
Ci+1 = Ci (S0 xor Ai) + S1 Bi ( (S0 xor Ai) xor Ci )

just a full adder with inputs S0 xor Ai, S1 Bi, and Ci

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 is in how negative numbers are represented
  - 2s 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