CSE140: Components and Design Techniques for Digital Systems

Decoders, adders, comparators, multipliers and other ALU elements

Tajana Simunic Rosing
Mux, Demux
Encoder, Decoder
Transmission Gate:  
Mux/Tristate building block

- nMOS are on when gate=1  
  - pass 1’s poorly from source to drain
- pMOS are on when gate=0  
  - pass 0’s poorly from source to drain
- Transmission gate is a better switch  
  - passes both 0 and 1 well
- When $EN = 1$, the switch is ON:  
  - $EN = 0$ and $A$ is connected to $B$
- When $EN = 0$, the switch is OFF:  
  - $A$ is not connected to $B
Floating: Z, Tristate Buffer and Tristate Busses

- Floating, high impedance, open, high Z
  - Disconnected
- Floating nodes are used in tristate busses
  - many different drivers, but only one is active at once

Tristate Buffer

<table>
<thead>
<tr>
<th>E</th>
<th>A</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Z</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Z</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Tristate Bus
**2:1 Multiplexer or Mux**

- Selects between one of \( N \) inputs to connect to output
- \( \log_2 N \)-bit select input – control input
- **Example:**

```
\begin{array}{cccc|c}
S & D_1 & D_0 & Y \\
0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 \\
0 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 \\
1 & 1 & 1 & 1 \\
\end{array}\\
```

**Logic gates**

\[ Y = D_0 S + D_1 S \]

**Tristates**

\[ Y = D_0 S + D_1 S \]

**Pass gates**

Sources: TSR, Katz, Boriello & Vahid
Multiplexers

- **2:1 mux:** \( Z = A'I_0 + AI_1 \)
- **4:1 mux:** \( Z = A'B'I_0 + A'BI_1 + AB'I_2 + ABI_3 \)
- **8:1 mux:** \( Z = A'B'C'I_0 + A'B'CI_1 + A'BC'I_2 + A'BCI_3 + AB'C'I_4 + AB'CI_5 + ABC'I_6 + ABCI_7 \)

- In general: \( Z = \sum_{k=0}^{2^n-1} (m_k I_k) \) – in minterm shorthand form for a 2\(^n\):1 Mux
Logic using Multiplexers

- Example of 2:1 mux implementation

\[ Y = AB \]
This multiplexer implements the same functionality for $Y$ as the truth table:

A. Yes
B. No

\[
\begin{array}{ccc}
A & B & Y \\
0 & 0 & 0 \\
0 & 1 & 0 \\
1 & 0 & 0 \\
1 & 1 & 1 \\
\end{array}
\]

\[Y = AB\]
Mux as general-purpose logic

- Example: $Z(A,B,C) = AC + BC' + A'B'C$

\[
\begin{align*}
A(B+B')C &+ (A+A')BC' \\
ABC + AB'BC &+ A'B'C \\
AB(C+C') &+ AB'C + ABC'
\end{align*}
\]
Function $Z(A,B,C)$ implemented by 2:1 Muxes above is:

A. $A'B'C' + ABC + BC'$
B. $(A'+AC)B + B'C'$
C. $A'B + B'C + BC'$
D. $A'+AC + BC'$
E. None of the above
Mux example: Logical function unit

<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>0</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>

Mux example:

8:1 MUX

S2 S1 S0

F

Sources: TSR, Katz, Boriello & Vahid
CSE140: Components and Design Techniques for Digital Systems

Decoders, adders, comparators, multipliers and other ALU elements

Tajana Simunic Rosing
Welcome to CSE 140!

• What we covered last time:
  – Switches & logic gates, Boolean algebra & representations, K-Maps, algorithm for simplification, mux

• Where we are going today:
  – Mux, demux, encoder, decoder, adder etc.

• Deadlines:
  – HW2 due, HW 3 assigned
  – Zybook

• Discussion session:
  – Wednesday at 7pm, Center Hall 113
Selects between one of $N$ inputs to connect to the output.

$\log_2 N$-bit select input – control input

What is the output of the mux if $\text{En} = 1$, $S = 10_2$, $D[3:0] = A_{16}$?

A. $0$
B. $1$
C. $Z$
D. $X$
E. None of the above

$D_2 = 0$

$D_0$

$D_1$

$D_3$

$S_1$

$S_0$

$\text{En} = 1$

$S_2 = 01_2$

$y = D_1 = 1$

Sources: TSR, Katz, Boriello & Vahid
Demultiplexers (opposite of Mux)

\[ y_i = x \text{ if } i = (S_{n-1}, \ldots, S_0) \& En = 1 \]
\[ y_i = 0 \text{ otherwise} \]

Control Input

\[ \begin{array}{cc|ccccc}
S_1 & S_0 & y_0 & y_1 & y_2 & y_3 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 \\
\end{array} \]
• \( N \) inputs, \( 2^N \) outputs

• One-hot outputs: only one output HIGH at a time when enable signal is 1 (EN=1)
### Decoder: logic equations & implementation

#### Decoders/demultiplexers
- Control inputs (called “selects” (S)) represent binary index of output to which the input is connected.
- Data input usually called “enable” or G in equations.

#### 1:2 Decoder:
\[
\begin{align*}
O_0 &= G \cdot S' \\
O_1 &= G \cdot S
\end{align*}
\]

#### 2:4 Decoder:
\[
\begin{align*}
O_0 &= G \cdot S_1' \cdot S_0' \\
O_1 &= G \cdot S_1' \cdot S_0 \\
O_2 &= G \cdot S_1 \cdot S_0' \\
O_3 &= G \cdot S_1 \cdot S_0
\end{align*}
\]

#### 3:8 Decoder:
\[
\begin{align*}
O_0 &= G \cdot S_2' \cdot S_1' \cdot S_0' \\
O_1 &= G \cdot S_2' \cdot S_1' \cdot S_0 \\
O_2 &= G \cdot S_2' \cdot S_1 \cdot S_0' \\
O_3 &= G \cdot S_2' \cdot S_1 \cdot S_0 \\
O_4 &= G \cdot S_2 \cdot S_1' \cdot S_0' \\
O_5 &= G \cdot S_2 \cdot S_1' \cdot S_0 \\
O_6 &= G \cdot S_2 \cdot S_1 \cdot S_0' \\
O_7 &= G \cdot S_2 \cdot S_1 \cdot S_0
\end{align*}
\]
Logic Using Decoders

- OR minterms

\[ Y = A \oplus B \]

\[ Y = AB + \overline{AB} \]

\[ = A \oplus B \]

Sources: TSR, Katz, Boriello & Vahid
Another example

- \( F(A,B,C) = \prod M(0,2,4) = \Sigma m(1,3,5,7) \)
Example as general-purpose logic

\[ F_1 = A'B'C'D + A'B'CD + ABCD \]
\[ F_2 = ABC'D' + ABC \]
\[ F_3 = (A' + B' + C' + D') = \frac{1}{ABCD} \]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Decoder Applications

Decoder converts a binary address to the assertion of the addressed device

\[ y_i = 1 \text{ if } E = 1 \& (I_2, I_1, I_0) = i \]
\[ y_i = 0 \text{ otherwise} \]

- **n** inputs
  - \( n = 3 \)
- **2^n outputs**
  - \( 2^3 = 8 \)
Implement a $6-2^6$ decoder with $3-2^3$ decoders.

Tree of Decoders
At most one $I_i = 1$.

$(y_{n-1}, \ldots, y_0) = i$ if $I_i = 1$ & $En = 1$

$(y_{n-1}, \ldots, y_0) = 0$ otherwise.

$A = 1$ if $En = 1$ and one $i$ s.t. $I_i = 1$

$A = 0$ otherwise.
Encoder: Logic Diagram

\[\begin{align*}
\text{I}_1 & \quad \text{I}_3 & \quad \text{I}_5 & \quad \text{I}_7 \\
\text{I}_4 & \quad \text{I}_5 & \quad \text{I}_6 & \quad \text{I}_7 \\
\text{I}_0 & \quad \text{I}_1 & \quad \text{I}_6 & \quad \text{I}_7 \\
\end{align*}\]
Decoder, Encoder, Mux, Demux

Decoder: Decode the address to assert the addressed device
Mux: Select the inputs according to the index addressed by the control signals
Adders
1-Bit & Multi-bit Adders

**Half Adder**

\[ S = A \oplus B \]
\[ C_{out} = AB \]

**Full Adder**

\[ S = A \oplus B \oplus C_{in} \]
\[ C_{out} = AB + AC_{in} + BC_{in} \]

**Types of multi-bit adders**
- Ripple-carry (slow)
- Carry-lookahead (faster)

**Symbol**
Ripple-Carry Adder

- Chain 1-bit adders together
- Carry ripples through entire chain
- Disadvantage: slow

Ripple-carry adder delay

\[ t_{\text{ripple}} = N t_{FA} \]

where \( t_{FA} \) is the delay of a full adder
Carry-lookahead adders

- Adder with propagate (P) and generate (G) outputs:
  - Evaluate Sum and Ci+1
    - Sum = Ai \text{xor} Bi \text{xor} Ci
    - Ci+1 = Ai Bi + Ai Ci + Bi Ci
      = Ai Bi + Ci (Ai \text{xor} Bi)
      = Gi + Ci Pi

\[
C_1 = G_0 + C_0 P_0 = A_0 B_0 + G_0 (A_0 \text{xor} B_0)
\]

\[
C_2 = G_1 + C_1 P_1 = G_1 + (G_0 + C_0 P_0) P_1
\]

\[
= G_1 + G_0 P_1 + C_0 P_0 P_1
\]
Carry-Lookahead Adder

• Example: 4-bit blocks ($G_{3:0}$ and $P_{3:0}$):

$$G_{3:0} = G_3 + P_3 \left( G_2 + P_2 \left( G_1 + P_1 G_0 \right) \right)$$
$$P_{3:0} = P_3 P_2 P_1 P_0$$

• Generally:
  - **Step 1:** Compute $G_i$ and $P_i$ for all columns
  - **Step 2:** Compute $G$ and $P$ for $k$-bit blocks
  - **Step 3:** $C_{in}$ propagates through each $k$-bit propagate/generate block

$$G_{i:j} = G_i + P_i \left( G_{i-1} + P_{i-1} \left( G_{i-2} + P_{i-2} G_j \right) \right)$$
$$P_{i:j} = P_i P_{i-1} P_{i-2} P_j$$
$$C_i = G_{i:j} + P_{i:j} C_{i-1}$$
Adders: CLA vs. Ripple

You are designing a 64-bit adder. To get the best performance, would you design:

A. A 64-bit ripple-carry adder
B. A 64-bit carry-lookahead adder
C. 8-bit sections of carry-lookahead with ripple carry connecting them
D. 32-bit sections of ripple-carry connected with carry-lookahead
Subtractors
2s complement

- If $N$ is a positive number, then the negative of $N$ (its 2s complement or $N^*$) is bit-wise complement plus 1
  - $7^*$ is $-7$: $0111 \rightarrow 1000 + 1 = 1001$ ($-7$)
  - $-7^*$ is 7: $1001 \rightarrow 0110 + 1 = 0111$ (7)
Subtraction

If you are using 4 bit number, what is the result of the following equation in 2s complement: \( y = 4 - 7 \approx -3 \)

A. 1011
B. 0011
C. 1101  \( \text{Correct Answer} \)
D. 1100
E. None of the above
Detecting Overflow: Method 1

- Assuming 4-bit two’s complement numbers, one can detect overflow by detecting when the two numbers’ sign bits are the same but are different from the result’s sign bit
  - If the two numbers’ sign bits are different, overflow is impossible
    - Adding a positive and negative can’t exceed the largest magnitude positive or negative
- Simple circuit
  - overflow = a3'b3's3 + a3b3s3'

```
+ 1 1 1 1 + 1 1 1 1 + 1 0 0 0
\hline
0 1 1 1 0 1 0 0 0 1 0 1 1 1
\hline
\frame{1}{1}{1}{1}
\frame{1}{1}{1}{0}
\frame{1}{0}{0}{1}
```

If the numbers’ sign bits have the same value, which differs from the result’s sign bit, overflow has occurred.
Detecting Overflow: Method 2

- Detect a difference between carry-in to sign bit and carry-out from it
- Yields a simpler circuit: overflow = c3 \text{ xor } c4 = c3 c4' + c3' c4

If the carry into the sign bit column differs from the carry out of that column, overflow has occurred.
## Subtractor

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Symbol" /></td>
<td><img src="image2" alt="Implementation" /></td>
</tr>
</tbody>
</table>

Sources: TSR, Katz, Boriello & Vahid
In this schematic addition occurs when \textit{Sel} signal is:

A. True

B. False
More ALU Components
Comparator: Equality

Symbol

Implementation

\[ \begin{array}{c}
A_3 \\
B_3 \\
A_2 \\
B_2 \\
A_1 \\
B_1 \\
A_0 \\
B_0 \\
\end{array} \]

Equal

Sources: TSR, Katz, Boriello & Vahid
Comparator: Less Than

A < B

Sources: TSR, Katz, Boriello & Vahid
Shifters

- **Logical shifter**: shifts value to left or right and fills empty spaces with 0’s
  - Ex: $11001 >> 2 = 00110$
  - Ex: $11001 << 2 = 00100$

- **Arithmetic shifter**: same as logical shifter, but on right shift, fills empty spaces with the old most significant bit
  - Ex: $11001 >>> 2 = 11110$
  - Ex: $11001 <<< 2 = 00100$

- **Rotator**: rotates bits in a circle, such that bits shifted off one end are shifted into the other end
  - Ex: $11001$ ROR 2 = 01110
  - Ex: $11001$ ROL 2 = 00111
General Shifter Design

$A_{3:0} \rightarrow \text{shamt}_{1:0} \rightarrow Y_{3:0}$
Multiplication of positive binary numbers

- Generalized representation of multiplication by hand

\[
\begin{array}{cccc}
  a_3 & a_2 & a_1 & a_0 \\
  \times & b_3 & b_2 & b_1 & b_0 \\
\end{array}
\]

\[
\begin{array}{cccccc}
  b_0a_3 & b_0a_2 & b_0a_1 & b_0a_0 & \text{(pp1)} \\
  b_1a_3 & b_1a_2 & b_1a_1 & b_1a_0 & 0 & \text{(pp2)} \\
  b_2a_3 & b_2a_2 & b_2a_1 & b_2a_0 & 0 & 0 & \text{(pp3)} \\
  b_3a_3 & b_3a_2 & b_3a_1 & b_3a_0 & 0 & 0 & 0 & \text{(pp4)} \\
\end{array}
\]

\[
\begin{array}{cccccccc}
  p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
  1010 & \times & 1100 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
  0000 & 0000 & 0000 & 0000 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
  10100000 + 10100000 & \text{\color{red} \underline{\hspace{10cm}}} \\
\end{array}
\]

\[
\begin{array}{cccccccc}
  11110000 \\
\end{array}
\]

For demo see:
http://courses.cs.vt.edu/~cs1104/BuildingBlocks/multiply.010.html

Sources: TSR, Katz, Boriello & Vahid
Multiplier – Array Style

- Multiplier design – array of AND gates

```
<table>
<thead>
<tr>
<th>a3</th>
<th>a2</th>
<th>a1</th>
<th>a0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>b3</td>
<td>b2</td>
<td>b1</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>b0a3</td>
<td>b0a2</td>
<td>b0a1</td>
<td>b0a0</td>
</tr>
<tr>
<td>b1a3</td>
<td>b1a2</td>
<td>b1a1</td>
<td>b1a0</td>
</tr>
<tr>
<td>b2a3</td>
<td>b2a2</td>
<td>b2a1</td>
<td>b2a0</td>
</tr>
<tr>
<td>+ b3a3</td>
<td>b3a2</td>
<td>b3a1</td>
<td>b3a0</td>
</tr>
</tbody>
</table>
```

Block symbol

Sources: TSR, Katz, Boriello & Vahid
Division of positive binary numbers

• Repeated subtraction
  – Set quotient to 0
  – Repeat while dividend $\geq$ divisor
    • Subtract divisor from dividend
    • Add 1 to quotient
  – When dividend < divisor:
    • Reminder = dividend
    • Quotient is correct

Example:
• Dividend: 101; Divisor: 10

<table>
<thead>
<tr>
<th>Dividend</th>
<th>Quotient</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>0 +</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>11 - 10</td>
<td>1 + 1</td>
</tr>
<tr>
<td>1 2</td>
<td>10 2</td>
</tr>
</tbody>
</table>

For demo see: http://courses.cs.vt.edu/~cs1104/BuildingBlocks/Binary.Divide.html
CSE140: Components and Design Techniques for Digital Systems

Decoders, adders, comparators, multipliers and other ALU elements

Tajana Simunic Rosing
Welcome to CSE 140!

• What we covered last time:
  – Mux, demux, encoder, decoder, add, sub, mul, div, cmp, shift

• Where we are going today:
  – Review of combinational components
  – ALU design

• Deadlines:
  – HW 3 due Th
  – Zybook daily – see course schedule

• Discussion session:
  – Wednesday at 7pm, Center Hall 113
Programmable Shifter

\[ A_{3:0} \xrightarrow{4} \text{shamt}_{1:0} \xrightarrow{4} Y_{3:0} \]

- **00** → no shift
- **01** \( A_3 \rightarrow Y_2, Y_3 = \Phi \)
- **10** \( A_3 \rightarrow Y_1, Y_2 = \Phi \)
- **11** \( A_3 \rightarrow Y_0, Y_3 = \Phi \)

Sources: TSR, Katz, Boriello & Vahid
Mux, Demux

**Mux**

\[
\begin{array}{c}
\text{En} \\
S_1 \quad S_0 \\
0 \quad 1 \quad 2 \quad 3 \\
D_0 \quad D_1 \quad D_2 \quad D_3 \\
y
\end{array}
\]

**Demux**

\[
\begin{array}{c}
\text{En} \\
S_1 \quad S_0 \\
0 \quad 1 \quad 2 \quad 3 \\
C_0 \quad C_1 \quad C_2 \quad C_3 \\
w
\end{array}
\]

\[
\begin{array}{c|cccc}
S_1 S_0 & C_3 & C_2 & C_1 & C_0 \\
\hline
00 & \varnothing & \varnothing & \varnothing & \varnothing \\
01 & 6 & \varnothing & w & \varnothing \\
10 & \varnothing & w & \varnothing & \varnothing \\
11 & w & \varnothing & \varnothing & \varnothing
\end{array}
\]

EN = 1

Sources: MSR, Katz, Boriello & Vahid
Adder, Subtractor & Comparator

A < B

A ≥ B

A = B

SUB

Operation

0

+ 1

- 1

2's Comp & Subtract

If SUB = 1

Sources: TSR, Katz, Boriello & Vahid
ALU: Arithmetic Logic Unit
Designing an Arithmetic Logic Unit

- **ALU Control Lines (ALUop)**
  - 000: And
  - 001: Or
  - 010: Add
  - 110: Subtract
  - 111: Set-on-less-than
A One Bit ALU

- This 1-bit ALU performs AND, OR, and ADD
A 32-bit ALU

1-bit ALU

32-bit ALU

Operation

CarryIn

a

b

0

1

2

Result

CarryOut

ALU bit slice

CarryIn

CarryOut

a0

b0

CarryIn

CarryOut

a1

b1

CarryIn

CarryOut

a2

b2

CarryIn

CarryOut

a31

b31

Result0

Result1

Result2

Result31

Sources: TSR, Katz, Boriello & Vahid
Subtract – We’d like to implement a means of doing A-B (subtract) but with only minor changes to our hardware. How?

\[ B - A > 0 \] 

1. Provide an option to use bitwise NOT A
2. Provide an option to use bitwise NOT B
3. Provide an option to use bitwise A XOR B
4. Provide an option to use 0 instead of the first CarryIn
5. Provide an option to use 1 instead of the first CarryIn

<table>
<thead>
<tr>
<th>Selection</th>
<th>Choices</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1 alone</td>
</tr>
<tr>
<td>B</td>
<td>Both 1 and 2</td>
</tr>
<tr>
<td>C</td>
<td>Both 3 and 4</td>
</tr>
<tr>
<td>D</td>
<td>Both 2 and 5</td>
</tr>
<tr>
<td>E</td>
<td>None of the above</td>
</tr>
</tbody>
</table>
Full 32-bit ALU

what signals accomplish ADD?

<table>
<thead>
<tr>
<th>Binvert</th>
<th>CIn</th>
<th>Oper</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>NONE OF THE ABOVE</td>
<td></td>
</tr>
</tbody>
</table>

sign bit (adder output from bit 31)
Full 32-bit ALU

what signals accomplish OR?

<table>
<thead>
<tr>
<th>Binvert</th>
<th>CIn</th>
<th>Oper</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>NONE OF THE ABOVE</td>
<td></td>
</tr>
</tbody>
</table>

sign bit (adder output from bit 31)
Full 32-bit ALU

what signals accomplish SUB?

<table>
<thead>
<tr>
<th>Binvert</th>
<th>CIn</th>
<th>Oper</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>NONE OF THE ABOVE</td>
<td></td>
</tr>
</tbody>
</table>

sign bit (adder output from bit 31)
### Arithmetic Logic Unit – Example 2

**Diagram:**

- **ALU**
- **Input:** A, B
- **Output:** Y
- **Carry Out:** C<sub>out</sub>
- **Sign:** S
- **Zero Extend:** [N-1]
- **Functions:**
  - F<sub>2:0</sub>  |  Function
  - 000  |  A & B
  - 001  |  A | B
  - 010  |  A + B
  - 011  |  Not used
  - 100  |  A & ~B
  - 101  |  A | ~B
  - 110  |  A - B
  - 111  |  Not used

**Sources:** TSR, Katz, Boriello & Vahid
ALU Design Example 3

<table>
<thead>
<tr>
<th>S2</th>
<th>S1</th>
<th>S0</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>B\cdot A</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>A\cdot B</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>A+B</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>A \text{xor} B</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>A \text{ or} B</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>A \text{ and} B</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
\[
\text{min SOP}(\overline{A}B\overline{C}\overline{D}) = \overline{A}D' + \overline{D'}\overline{C'} + \overline{A}B + \overline{A}'D C
\]

\[
\overline{A}\overline{B}\overline{C}\overline{D} \quad F(\overline{A}\overline{B}\overline{C}\overline{D}) = \Sigma m(\ldots)
\]