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

- processor
- video
- Ethernet
- memory

Sources: TSR, Katz, Boriello & Vahid
2:1 Multiplexer or Mux

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

![2:1 Mux Diagram]

<table>
<thead>
<tr>
<th>$S$</th>
<th>$D_1$</th>
<th>$D_0$</th>
<th>$Y$</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>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Logic gates

<table>
<thead>
<tr>
<th>$S$</th>
<th>$D_0$</th>
<th>$D_1$</th>
<th>$Y$</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>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Tristates

<table>
<thead>
<tr>
<th>$S$</th>
<th>$Y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$D_0$</td>
</tr>
<tr>
<td>1</td>
<td>$D_1$</td>
</tr>
</tbody>
</table>

Pass gates

$Y = D_0S + D_1S$
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:** \( Z = A'B'C'I_0 + A'B'Cl_1 + A'BC'I_2 + A'BCl_3 + AB'C'I_4 + AB'Cl_5 + ABC'I_6 + ABCl_7 \)

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

![Multiplexer Diagram](image)
## Logic using Multiplexers

- Example of 2:1 mux implementation

$$Y = AB$$

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

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

Sources: TSR, Katz, Boriello & Vahid
This multiplexer implements the same functionality for Y as the truth table

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

Y = AB

A. Yes
B. No
Mux as general-purpose logic

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

  $Z(0, 1, 0) = \phi + B + \phi$

  $Z(0, 1, 1) = \phi + \phi + B$

  $Z(1, 1, 0) = \phi + B + \phi$

  $Z(1, 1, 1) = \phi + B + \phi$
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>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 \oplus B)</td>
<td>logical XOR</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>(A \oplus B)'</td>
<td>logical XNOR</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>(A \cdot 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>

**Diagram:**

- 3x2 MUX configuration
- Inputs: A, B, \(S_2, S_1, S_0\)
- Outputs: F

Sources: TSR, Katz, Boriello & Vahid
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 $En=1$, $S=10_2$, $D[3:0]=A_{16}$?

A. 0
B. 1
C. Z
D. X
E. None of the above
Demultiplexers (opposite of Mux)

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

- \( O_0 = G \cdot S' \)
- \( O_1 = G \cdot S \)

2:4 Decoder:

- \( O_0 = G \cdot S_1' \cdot S_0' \)
- \( O_1 = G \cdot S_1' \cdot S_0 \)
- \( O_2 = G \cdot S_1 \cdot S_0' \)
- \( O_3 = G \cdot S_1 \cdot S_0 \)

3:8 Decoder:

- \( O_0 = G \cdot S_2' \cdot S_1' \cdot S_0' \)
- \( O_1 = G \cdot S_2' \cdot S_1' \cdot S_0 \)
- \( O_2 = G \cdot S_2' \cdot S_1 \cdot S_0' \)
- \( O_3 = G \cdot S_2' \cdot S_1 \cdot S_0 \)
- \( O_4 = G \cdot S_2 \cdot S_1' \cdot S_0' \)
- \( O_5 = G \cdot S_2 \cdot S_1' \cdot S_0 \)
- \( O_6 = G \cdot S_2 \cdot S_1 \cdot S_0' \)
- \( O_7 = G \cdot S_2 \cdot S_1 \cdot S_0 \)
Logic Using Decoders

- OR minterms

\[ Y = AB + \overline{AB} \]
\[ = A \oplus B \]

Sources: TSR, Katz, Boriello & Vahid
Another example

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

F1 = A'B'C'D + A'B'CD + ABCD
F2 = ABC'D' + ABC
F3 = (A' + B' + C' + D')

F3 = ABCD
Decoder Applications

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

n to $2^n$ decoder function:

$y_i = 1$ if $E = 1$ & $(I_2, I_1, I_0) = i$

$y_i = 0$ otherwise

n inputs

n= 3

2^n outputs

$2^3 = 8$
Implement a 6-2^6 decoder with 3-2^3 decoders.
At most one $I_i = 1$.

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

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

\[A = 1 \text{ if } En = 1 \text{ and one } i \text{ s.t. } I_i = 1\]

\[A = 0 \text{ otherwise.}\]
Encoder: Logic Diagram

I_1, I_3, I_5, I_7 → y_0

I_2, I_3, I_6, I_7 → y_1

I_4, I_5, I_6, I_7 → y_2

I_0, I_1, I_5, I_6, I_7 → A

Sources: TSR, Katz, Boriello & Vahid
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**

\[
\begin{array}{c|c|c|c}
A & B & C_{\text{out}} & S \\
0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 \\
1 & 1 & 1 & 0 \\
\end{array}
\]

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

**Full Adder**

\[
\begin{array}{c|c|c|c|c|c|c|c}
C_{\text{in}} & A & B & C_{\text{out}} & S \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 & 1 \\
0 & 1 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 & 1 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

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

**Symbol**

Sources: TSR, Katz, Boriello & Vahid
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 \oplus Bi \oplus Ci
    - Ci+1 = Ai Bi + Ai Ci + Bi Ci
      = Ai + Ci (Ai \oplus Bi)
      = Gi + Ci Pi

Increasingly complex logic for carries
**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

---

**Examples:**
- \( C0 \):
- \( C1 @ 3 \)
- \( C2 @ 3 \)
- \( C3 @ 3 \)
- \( C4 @ 3 \)

---

**Sources:** TSR, Katz, Boriello & Vahid
Carry-Lookahead Adder

• **Example:** 4-bit blocks \((G_{3:0} \text{ and } P_{3:0})\):
  \[
  G_{3:0} = G_3 + P_3 (G_2 + P_2 (G_1 + P_1 G_0))
  
  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 (G_{i-1} + P_{i-1} (G_{i-2} + P_{i-2} G_j))
    
    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. 16-bit sections of ripple-carry connected with carry-lookahead

\[ C_{out} = G_o + C_i \cdot P_o \]
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 -> 1000 + 1 = 1001 (-7)
  - -7* is 7: 1001 -> 0110 + 1 = 0111 (7)
Subtraction

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

A. 1011
B. 0011
C. 1101
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’

<table>
<thead>
<tr>
<th>sign bits</th>
<th>a3</th>
<th>b3</th>
<th>s3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>+</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>overflow</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sign bits</th>
<th>a3</th>
<th>b3</th>
<th>s3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(b)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>+</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>overflow</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sign bits</th>
<th>a3</th>
<th>b3</th>
<th>s3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>+</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>no overflow</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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: $\text{overflow} = c_3 \oplus c_4 = c_3 c_4' + c_3' c_4$

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

Symbol

Implementation

Sources: TSR, Katz, Boriello & Vahid
In this schematic addition occurs when Sel signal is:
A. True
B. False
More ALU Components
Comparator: Equality

Symbol

\[ \begin{array}{c}
A_3 \\
\downarrow 4 \\
\hline \\
A_2 \\
\downarrow 4 \\
\hline \\
A_1 \\
\downarrow \\
\hline \\
A_0 \\
\downarrow \\
\hline \\
B_3 \\
\downarrow 4 \\
\hline \\
B_2 \\
\downarrow \\
\hline \\
B_1 \\
\downarrow \\
\hline \\
B_0 \\
\downarrow \\
\hline \\
= \\
\end{array} \]

Implementation

Equal

\[ \begin{array}{c}
A_3 \\
\downarrow \\
\hline \\
A_2 \\
\downarrow \\
\hline \\
A_1 \\
\downarrow \\
\hline \\
A_0 \\
\downarrow \\
\hline \\
B_3 \\
\downarrow \\
\hline \\
B_2 \\
\downarrow \\
\hline \\
B_1 \\
\downarrow \\
\hline \\
B_0 \\
\downarrow \\
\hline \\
\end{array} \]
Comparator: Less Than

A < B

A - B

3 - 7 = -4

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

\[
A_{3:0} \quad \overset{4}{\downarrow} \quad \overset{4}{\downarrow} \quad Y_{3:0}
\]

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

- **Generalized representation of multiplication by hand**

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

\[
\begin{array}{cccccccc}
  b_0 a_3 & b_0 a_2 & b_0 a_1 & b_0 a_0 & \quad (pp1) \\
  b_1 a_3 & b_1 a_2 & b_1 a_1 & b_1 a_0 & 0 & \quad (pp2) \\
  b_2 a_3 & b_2 a_2 & b_2 a_1 & b_2 a_0 & 0 & 0 & \quad (pp3) \\
  + b_3 a_3 & b_3 a_2 & b_3 a_1 & b_3 a_0 & 0 & 0 & 0 & \quad (pp4) \\
\end{array}
\]

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

For demo see: [http://courses.cs.vt.edu/~cs1104/BuildingBlocks/multiply.010.html](http://courses.cs.vt.edu/~cs1104/BuildingBlocks/multiply.010.html)
Multiplier – Array Style

- Multiplier design – array of AND gates

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

\[ \begin{array}{cccccccc}
  b_{0a3} & b_{0a2} & b_{0a1} & b_{0a0} & (pp1) \\
  b_{1a3} & b_{1a2} & b_{1a1} & b_{1a0} & 0 & (pp2) \\
  b_{2a3} & b_{2a2} & b_{2a1} & b_{2a0} & 0 & 0 & (pp3) \\
  b_{3a3} & b_{3a2} & b_{3a1} & b_{3a0} & 0 & 0 & 0 & (pp4) \\
\end{array} \]

\[ + \begin{array}{cccccc}
  b_3 & b_2 & b_1 & b_0 \\
\end{array} \]

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

Block symbol
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

\[
\begin{array}{ccc}
\text{Dividend} & \text{Quotient} \\
5 & 0 & + \\
\downarrow & \uparrow & \\
2 & 1 & \\
\downarrow & \uparrow & \\
3 & 0 & + \\
\downarrow & \uparrow & \\
1 & 1 & \\
\end{array}
\]

For demo see: [http://courses.cs.vt.edu/~cs1104/BuildingBlocks/Binary.Divide.html](http://courses.cs.vt.edu/~cs1104/BuildingBlocks/Binary.Divide.html)
ALU: Arithmetic Logic Unit
Designing an Arithmetic Logic Unit

• **ALU Control Lines (ALUop)**

<table>
<thead>
<tr>
<th>ALUop</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>And</td>
</tr>
<tr>
<td>001</td>
<td>Or</td>
</tr>
<tr>
<td>010</td>
<td>Add</td>
</tr>
<tr>
<td>110</td>
<td>Subtract</td>
</tr>
<tr>
<td>111</td>
<td>Set-on-less-than</td>
</tr>
</tbody>
</table>
A One Bit ALU

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

1-bit ALU

32-bit ALU

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?

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

**Selection** | **Choices**
---|---
A | 1 alone
B | Both 1 and 2
C | Both 3 and 4
D | Both 2 and 5
E | None of the above
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></td>
<td></td>
</tr>
</tbody>
</table>

E  NONE OF THE ABOVE

sign bit (adder output from bit 31)
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

<table>
<thead>
<tr>
<th>$F_{2:0}$</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>$A &amp; B$</td>
</tr>
<tr>
<td>001</td>
<td>$A \mid B$</td>
</tr>
<tr>
<td>010</td>
<td>$A + B$</td>
</tr>
<tr>
<td>011</td>
<td>Not used</td>
</tr>
<tr>
<td>100</td>
<td>$A &amp; \neg B$</td>
</tr>
<tr>
<td>101</td>
<td>$A \mid \neg B$</td>
</tr>
<tr>
<td>110</td>
<td>$A - B$</td>
</tr>
<tr>
<td>111</td>
<td>Not used</td>
</tr>
</tbody>
</table>
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-A</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>A-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 xor B</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>A or B</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>A and B</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>