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></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:**

$$Y = D_0 \overline{S} + D_1 S$$

<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**

**Tristates**

**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

![Diagram of multiplexers with truth tables and minterm expressions.](image-url)
Logic using Multiplexers

• Example of 2:1 mux implementation

\[
Y = AB
\]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></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>

\[
A \quad Y
\]

\[
A \quad B \quad Y
\]

\[
A \quad 0 \quad 0
\]

\[
A \quad 0 \quad 0
\]

\[
A \quad 1 \quad B
\]

\[
A \quad 1 \quad 1
\]

\[
A \quad 1 \quad Y
\]

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

A. Yes

B. No
Mux as general-purpose logic

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

Selects between one of $N$ inputs to connect to the output. 
$log_2 N$-bit select input – control input
Demultiplexers (opposite of Mux)

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

\[ y_i = 0 \text{ otherwise} \]
Decoder

- $N$ inputs, $2^N$ outputs
- One-hot outputs: only one output HIGH at a time when enable signal is 1 (EN=1)

<table>
<thead>
<tr>
<th>$A_1$</th>
<th>$A_0$</th>
<th>$Y_3$</th>
<th>$Y_2$</th>
<th>$Y_1$</th>
<th>$Y_0$</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>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Sources: TSR, Katz, Boriello & Vahid
• 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

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

- $F(A,B,C) = \Pi M(0,2,4)$
Example as general-purpose logic

\[ F_1 = A'BC'D' + A'B'CD + ABCD \]
\[ F_2 = ABC'D' + ABC \]
\[ F_3 = (A' + B' + C' + D') \]
Decoder applications

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

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

\( n \) to \( 2^n \) decoder function:

- \( I_0 \)
- \( I_1 \)
- \( I_2 \)

\( 0 \)\hspace{1cm} \( 1 \)\hspace{1cm} \( 2 \)
\( 0 \hspace{1cm} 1 \hspace{1cm} 2 \hspace{1cm} 3 \hspace{1cm} 4 \hspace{1cm} 5 \hspace{1cm} 6 \hspace{1cm} 7 \)

\( n \) inputs
\( n = 3 \)

\( 2^n \) outputs
\( 2^3 = 8 \)
Implement a $6-2^6$ decoder with $3-2^3$ decoders.
At most one $I_i = 1$.

$(y_{n-1}, \ldots, y_0) = i$ 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

En

y_0

I_1
I_3
I_5
I_7

En

I_2
I_3
I_6
I_7

En

y_1

I_4
I_5
I_6
I_7

En

y_2

I_0
I_1
: 
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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C_{out}</th>
<th>S</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>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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

Full Adder

<table>
<thead>
<tr>
<th>C_{in}</th>
<th>A</th>
<th>B</th>
<th>C_{out}</th>
<th>S</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>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</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>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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

Types of multi-bit adders

- Ripple-carry (slow)
- Carry-lookahead (faster)

Symbol

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 \text{ xor } Bi \text{ xor } Ci
  - Ci+1 = Ai Bi + Ai Ci + Bi Ci
    = Ai Bi + Ci (Ai \text{ xor } Bi)
    = Gi + Ci Pi

Increasingly complex logic for carries
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. 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 -> 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: $y = 4 - 7$

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></th>
<th>a3</th>
<th>b3</th>
<th>s3</th>
</tr>
</thead>
<tbody>
<tr>
<td>overflow (a)</td>
<td>1 1 1 1</td>
<td>0 0 0 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>overflow (b)</td>
<td>1 1 1 1</td>
<td>1 0 0 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>no overflow (c)</td>
<td>1 0 0 0</td>
<td>0 1 1 1</td>
<td></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

\[
\begin{array}{c}
A \quad B \\
N \quad N \\
- \quad - \\
Y \\
N \\
\end{array}
\]

Implementation

\[
\begin{array}{c}
A \quad B \\
N \quad N \\
+ \\
Y \\
N \\
\end{array}
\]

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

A. True
B. False
More ALU Components
Comparator: Equality

Symbol

\[ A_4 \quad B_4 \]

Implementation

\[ A_3 \quad B_3 \]
\[ A_2 \quad B_2 \]
\[ A_1 \quad B_1 \]
\[ A_0 \quad B_0 \]

Equal

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

A < B
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

Sources: TSR, Katz, Boriello & Vahid
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

\[
\begin{array}{cccc}
  a_3 & a_2 & a_1 & a_0 \\
  \times & b_3 & b_2 & b_1 & b_0 \\
  \hline
  b_0a_3 & b_0a_2 & b_0a_1 & b_0a_0 & \text{(pp1)} \\
  b_1a_3 & b_1a_2 & b_1a_1 & b_1a_0 & 0 & \text{(pp2)} \\
  b_2a_3 & b_2a_2 & b_2a_1 & b_2a_0 & 0 & 0 & \text{(pp3)} \\
  + b_3a_3 & b_3a_2 & b_3a_1 & b_3a_0 & 0 & 0 & 0 & \text{(pp4)} \\
  \hline
  p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}
\]

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

```
+ (5-bit)
+ (6-bit)
+ (7-bit)
```

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

\[
\begin{array}{cccccc}
  b_0a_3 & b_0a_2 & b_0a_1 & b_0a_0 \\
  b_1a_3 & b_1a_2 & b_1a_1 & b_1a_0 & 0 \\
  b_2a_3 & b_2a_2 & b_2a_1 & b_2a_0 & 0 & 0 \\
  b_3a_3 & b_3a_2 & b_3a_1 & b_3a_0 & 0 & 0 & 0 \\
\end{array}
\]

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

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

- Repeated subtraction
  - Set quotient to 0
  - Repeat while dividend >= 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 -</td>
<td>1 +</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

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)**
  - 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

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

<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**

The diagram illustrates a 32-bit arithmetic logic unit (ALU) with inputs and outputs labeled. The ALU processes binary operations on inputs `a` and `b`, which are 32-bit numbers. The outputs include `Result`, `CarryOut`, and flags for `Less`, `Equal`, and `Greater`. The inputs include `Binvert`, `CarryIn`, and `Operation`.

### Table: What signals accomplish ADD?

<table>
<thead>
<tr>
<th></th>
<th>Binvert</th>
<th>CIn</th>
<th>Oper</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>NONE OF THE ABOVE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sign bit (adder output from bit 31): This flag indicates the sign of the result, with a value of 0 for positive and 1 for negative.
Full 32-bit ALU

what signals accomplish OR?

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

Little more intense – can you get this?

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

<table>
<thead>
<tr>
<th>F2:0</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>A &amp; B</td>
</tr>
<tr>
<td>001</td>
<td>A</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; ~B</td>
</tr>
<tr>
<td>101</td>
<td>A</td>
</tr>
<tr>
<td>110</td>
<td>A - B</td>
</tr>
<tr>
<td>111</td>
<td>Not used</td>
</tr>
</tbody>
</table>

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-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>

Sources: TSR, Katz, Boriello & Vahid