In which circumstance can overflow not occur?

- A: subtracting a positive number from a negative number
  \[ `-` + `-` \]

- B: subtracting a negative number from zero
  \[ 0 - (-8) = 8 \text{ overflow} \]

- C: adding two negative numbers
  \[ `-` + `-` \]

- D: subtracting a negative number from a positive number
  \[ `+` + `+` \]

- E: subtracting a negative number from a negative number
  \[ `-` + `+` \]
Building an ALU (Part 1)
An Arithmetic Logic Unit (ALU) is the primary manipulator of state information in computers.

Computer can do 2 things
1) Store state
2) Manipulate state (Combine arithmetic and logical operations into one unit)
233 in one slide!

- The class consists roughly of 4 quarters: (Bolded words are the big ideas of the course, pay attention when you hear these words)
  1. You will build a simple computer processor
     Build and create **state** machines with **data**, **control**, and **indirection**
  2. You will learn how high-level language code executes on a processor
     Time limitations create **dependencies** in the **state** of the processor
  3. You will learn why computers perform the way they do
     Physical limitations require **locality** and **indirection** in how we access **state**
  4. You will learn about hardware mechanisms for parallelism
     **Locality**, **dependencies**, and **indirection** on performance enhancing drugs

- We will have a SPIMbot contest!
Today’s lecture

- We start building our computer!
  - We’ll start with the arithmetic/logic unit (ALU)

- Adding single bits
  - Half Adders and Full Adder

- Multi-bit Arithmetic
  - Hierarchical design
  - Subtraction

- Building a Logic Unit
  - Multiplexors
The computation in a computer processor takes place in the arithmetic logic unit (ALU)

- Arithmetic Unit (AU) performs arithmetic operations
  - e.g., addition and subtraction
- Logic Unit (LU) performs bit-wise logical operations
  - e.g., AND, OR, NOR, XOR

- Typically these operations are performed on multi-bit words
  - The MIPS-subset processor we will build uses 32-bit words

In Lab 3 you will build a 32-bit ALU with the above operations
Binary Addition Review

\[
\begin{array}{cccccc}
1 & 1 & 1 & 0 & 0 & \text{Carries} \\
1 & 0 & 1 & 1 & & \text{Augend} \\
+ & 1 & 1 & 1 & 0 & \text{Addend} \\
\hline
1 & 1 & 0 & 0 & 1 & \text{Sum}
\end{array}
\]
First bit position receives two input bits to produce two output bits

Two input bits:
We’ll call them $x, y$

Two output bits: $c = \text{carry out}$
(from first column)

$s = \text{Sum}$
Specify the first bit position’s behavior with a truth table

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>C</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>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>0</td>
</tr>
</tbody>
</table>

Two input bits: We’ll call them x, y

\[ C = x \cdot y \]
\[ S = x' \cdot y + x \cdot y' \]

\[ 1 + 1 = 0 \quad \text{Carries} \]
\[ 1 + 1 = 1 \quad \text{Augend} \]
\[ 1 + 1 = 0 \quad \text{Addend} \]
\[ 1 + 1 + 0 = 1 \quad \text{Sum} \]

\[ c = \text{carry out} \quad \text{(from first column)} \]
\[ s = \text{Sum} \]
This truth table specifies a circuit we call a **half adder**

- Adds two input bits to produce a sum and carry out.

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>C</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>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>0</td>
</tr>
</tbody>
</table>

\[ C = XY \]
\[ S = X'Y + XY' = X \oplus Y \]

- The carry-out bit has twice the magnitude of the sum bit
Second bit position receives **three** input bits to produce two output bits
- (and every subsequent position)

Still two output bits:  
\[c = \text{carry out}\]  
\[s = \text{Sum}\]  
(from first column)
Remember the train module?
Specify the remaining bit positions’ behaviors with a truth table

- Adding 3 bits together to get a two bit number

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>C_in</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>0</td>
<td>0</td>
</tr>
<tr>
<td>0</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>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
This truth table specifies a circuit we call a **Full Adder**

- Adds three input bits to produce a sum and carry out.

\[
S = X \oplus Y \oplus C_{in} \\
C_{out} = XY + (X \oplus Y)C_{in}
\]

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>C_{in}</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>0</td>
<td>1</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>

- odd/even
We can use hierarchical design to build a full adder from a half adder

\[
S = X \oplus Y \oplus C_{in}
\]

\[
C_{out} = XY + (X \oplus Y)C_{in}
\]

Half Adder Equations

\[
C = XY
\]

\[
S = X \oplus Y
\]
We can use hierarchical design to build multi-bit adders

- Recall our discussion about hierarchical design
  - *(The stop lights to prevent train collisions...)*

- Example: 4-bit adder
An example of 4-bit addition

- Let’s try our initial example: \(A=1011\) (eleven), \(B=1110\) (fourteen).

What is the value of \(S1\)?

a) 0
b) 1

d) Overflow
Implementing Subtraction

- Subtraction is technically negating the second input and then adding
  \[ A - B = A + (-B) \]

- Negating in 2’s complement is inverting the bits and adding one
  \[ -B = \sim B + 1 \]

- Substituting in:

  \[ A - B = A + (-B) = \]

<table>
<thead>
<tr>
<th>Option</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>( A - \sim B + 1 )</td>
</tr>
<tr>
<td>B</td>
<td>( A + \sim B + 1 )</td>
</tr>
<tr>
<td>C</td>
<td>( A - (\sim B + 1) )</td>
</tr>
<tr>
<td>D</td>
<td>( A + \sim B - 1 )</td>
</tr>
<tr>
<td>E</td>
<td>none of the above</td>
</tr>
</tbody>
</table>
Let’s try an example: \( A = 0011 \) (three), \( B = 1110 \) (negative 2).

\[
A + \sim B + 1
\]

\[
\begin{array}{c}
0 \\
1 \\
0 \\
1
\end{array}
\]

What is the value of \( S_3 \)?

a) 0
b) 1
Use XOR gates to implement Addition + Subtraction in one circuit

- XOR gates let us selectively complement the B input.
  
  \[
  X \oplus 0 = X \quad X \oplus 1 = X'
  \]

- When \( \text{Sub} = 0 \), \( Y = B \) and \( \text{Cin} = 0 \). Result = \( A + B + 0 = A + B \).
- When \( \text{Sub} = 1 \), \( Y = \neg B \) and \( \text{Cin} = 1 \). Result = \( A + \neg B + 1 = A - B \).
We conceptually distinguish two types of signal in hardware: Data and Control

- **Datapath**
  - These generally carry the numbers we’re crunching
  - E.g., the X and Y inputs and the output S

- **Control**
  - These generally control how data flows and what operations are performed
  - E.g., the SUB signal.
Logical Operations

- In addition to ADD and SUBTRACT, we want our ALU to perform bit-wise AND, OR, NOR, and XOR.
- This should be straightforward.
  - We have gates that perform each of these operations.
Selecting the desired logical operation

- We need a control signal to specify the desired operation:
  - We’ll call that signal R
  - 4 operations means R is 2 bits

- We need a circuit to perform the selection:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>$G_i = X_i Y_i$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>$G_i = X_i + Y_i$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>$G_i = (X_i + Y_i)'$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>$G_i = X_i \oplus Y_i$</td>
</tr>
</tbody>
</table>
Multiplexors use **control** bits to select **data**

- A multiplexor is a circuit that (logically) selects one of its **data** inputs to connect to its **data** output.

- Consider a 2-to-1 multiplexor. It has:
  - 2 **data** input bits ($I_0, I_1$)
  - a 1-bit **control** input bit ($S$)
  - 1 **data** output bit ($Y$)

- The control input selects which data input is output:
  $$Y = S' \cdot I_0 + S \cdot I_1$$
  - $S=0$: $0 \cdot I_0 + 0 \cdot I_1 = 0 = I_0$
  - $S=1$: $0 \cdot I_0 + 1 \cdot I_1 = I_1$
Multiplexors, cont.

- In general, a multiplexor (mux) has:
  - $2^N$ data input bits ($I_0-I_{2^N-1}$)
  - an $N$-bit control input ($S$)
  - 1 data output bit ($Y$)
  - If $S = K$ then $Y = I_K$

Examples:

- 4-to-1 mux: 4 data input bits, 2-bit control input
  - $Y = S_1'S_0'I_0 + S_1'S_0I_1 + ____I_2 + S_1S_0I_3$

- 16-to-1 mux: 16 data input bits, 4-bit control input

- A: $S_1S_0$
- B: $S_2S_0$
- C: $S_1S_0'$
- D: $S_2S_0'$
- E: $S_1'S_0$
Complete 1-bit Logic Unit

<table>
<thead>
<tr>
<th>$R_1$</th>
<th>$R_0$</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>$G_i = X_i Y_i$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>$G_i = X_i + Y_i$</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>$G_i = (X_i + Y_i)'$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>$G_i = X_i \oplus Y_i$</td>
</tr>
</tbody>
</table>

The diagram shows a complete 1-bit logic unit with inputs $X$ and $Y$, and output $G$. The unit includes AND, OR, NOR, and XOR gates controlled by $R_0$ and $R_1$. The truth table provides the output $G_i$ for different combinations of $R_0$ and $R_1$.
Mux Hierarchical Design (operand width)

- What if we want to mux 2 2bit numbers?
Mux Hierarchical Design (more inputs)

- How do we build a mux with 4 inputs?

A:

B:

C: Either