Admin

Assignment 4 due Monday at 11:59pm

Assignment 5 posted soon
   ▶ due Friday Oct. 23rd, at 5pm

CS lunch today!

Midterm

Average: 23.25

Top quartile: 26
Top half (median): 24.6
Bottom quartile: 21.4
Diving into your computer

Normal computer user

After intro CS

After 5 weeks of cs52
What now?

One last note on CS41B

Encoding assembly instructions

What now?
Review: binary addition

\[
\begin{array}{c}
\text{01010} \\
+ \text{01111} \\
\hline \\
\text{11001}
\end{array}
\]

Do the binary addition, making sure to keep track of the carries.
Assume unsigned numbers for now.

Just to be sure, what are these numbers in decimal?

\[
\begin{array}{c}
\text{1110} \\
\text{01010} \\
+ \text{01111} \\
\hline \\
\text{11001}
\end{array}
\]

We saw before, that we can view this problem recursively. How?
handle a digit at a time

A recursive component

Adding with components
Adding with components

\[
\begin{array}{c}
01010 \\
+ 01111 \\
\hline
10100
\end{array}
\]

Adding with components

\[
\begin{array}{c}
0 \\
+ 01111 \\
\hline
1
\end{array}
\]

Adding with components

\[
\begin{array}{c}
10 \\
01010 \\
+ 01111 \\
\hline
01
\end{array}
\]

Adding with components

\[
\begin{array}{c}
110 \\
01010 \\
+ 01111 \\
\hline
001
\end{array}
\]
Adding with components

```
1110
01010
+01111
11001
```

What goes on inside the component?

Current implementation uses addition!

Implementing the component

```
let total = c + x + y
if total >= 2 then (* check if there's a carry *)
    total = total - 2
end
```

What are the outputs?
Implementing the component

<table>
<thead>
<tr>
<th>in1</th>
<th>in2</th>
<th>carry-in</th>
<th>out</th>
<th>carry-out</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>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<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>1</td>
<td>1</td>
<td>0</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>0</td>
<td>0</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>

Another implementation

```python
for addMultBinary (ML ML) ML = ML
  addMultBinary (ML mL) ML = addMultBinary (ML ML) mL
  addMultBinary (ML mL) mL = addMultBinary (ML mL) y1
  addMultBinary (ML mL) y1 = addMultBinary (ML mL) y
  if x = 1 (addMultBinary (ML mL) mL) then
    1:(addMultBinary (ML mL) y)
  else if x = 1 (addMultBinary (ML mL) mL) then
    Cx = 1 (addMultBinary (ML mL) mL) Cx = 1
  else if x = 1 (addMultBinary (ML mL) mL) then
    x = 1 (addMultBinary (ML mL) mL) x = 1
  else
    0:(addMultBinary (ML mL) y)
```

- Don’t use addition anymore
- Translated the problem into a boolean logic problem

What are some boolean operators?

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A and B</th>
<th>A or B</th>
<th>not A</th>
</tr>
</thead>
<tbody>
<tr>
<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>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</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>0</td>
</tr>
</tbody>
</table>

What are some boolean operators?

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A and B</th>
<th>A or B</th>
<th>not A</th>
</tr>
</thead>
<tbody>
<tr>
<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>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</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>0</td>
</tr>
</tbody>
</table>
Gates

Gates have inputs and outputs
- values are 0 or 1

Are hardware components!

Utilizing gates

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A and B</th>
<th>A xor B</th>
<th>A nand B</th>
<th>A nor B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<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>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</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>

Gates as hardware

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

Utilizing gates

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A and B</th>
<th>A xor B</th>
<th>A nand B</th>
<th>A nor B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</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>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</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>

10/8/15
Utilizing gates

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A and B</th>
<th>A or B</th>
<th>not A</th>
<th>A nand B</th>
<th>A nor B</th>
<th>A xor B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</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>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<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>
</tbody>
</table>

When is this circuit 1?

A

B

A and B

A or B

not A

A nand B

A nor B

A xor B

Utilizing gates

Designing more interesting circuits

<table>
<thead>
<tr>
<th>in1</th>
<th>in2</th>
<th>in3</th>
<th>OUT</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>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>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Design a circuit for this
Designing more interesting circuits

<table>
<thead>
<tr>
<th>in1</th>
<th>in2</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</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>

Back to addition...

<table>
<thead>
<tr>
<th>in1</th>
<th>in2</th>
<th>carry-in</th>
<th>carry-out</th>
<th>sum</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>0</td>
<td>0</td>
<td>1</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>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

A half-adder: no carry-in

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>carry</th>
<th>sum</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>

A half-adder: no carry-in

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>carry</th>
<th>sum</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>

Design a circuit for this

Hint: solve each output bit independently
A half-adder: no carry-in

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>carry</th>
<th>sum</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>

Implementing a full adder

Implementing a full adder

Can I ever get a carry from both half adders?

Implementing the component

What goes on inside the component?
Implementing the component

Ripple carry adder

To implement an n-bit adder, we chain together n full-adders, each adder handles one bit position.

\[ A = A_3 A_2 A_1 A_0 \]
\[ B = B_3 B_2 B_1 B_0 \]

Adder for adding 4-bit numbers

Signed addition

0010
+ 1110
\[ ? \]

Do the binary addition, making sure to keep track of the carry.
Assume signed numbers for now.
Signed addition

\[ \begin{array}{c}
\text{throw away last carry bit} \\
1110 \\
0010 \\
+ 1110 \\
0000
\end{array} \]

Is that right?
What numbers are these?

Signed addition

\[ \begin{array}{c}
1110 \\
0010 \\
+ 1110 \\
0000
\end{array} \]

\[
\begin{array}{c}
2 \\
-2 \\
0
\end{array}
\]

Ripple carry adder will work for signed and unsigned numbers

Subtraction

\[ \begin{array}{c}
0010 \\
- 1110 \\
\end{array} \]

We can solve this doing addition

Subtraction

\[ \begin{array}{c}
0010 \\
- 1110 \\
\end{array} \]

\[ \begin{array}{c}
0010 \\
0010 \\
\end{array} \]

flip bits and add 1

\[ \begin{array}{c}
0100
\end{array} \]

Do addition!
Ripple carry adder/subtractor

If $D = 0$
- Carry in for first adder = 0
- $B_i \oplus 0 = B_i$

If $D = 1$
- Carry in for first adder = 1 (+1 to sum)
- $B_i \oplus 1 = \overline{B_i}$ (flip all the bits of $B_i$)

C, N, Z and V bits

In addition to the sum, we often also calculate some other useful information:
- **C**: carry out bit of the adder
- **Z**: 1 if the total result is zero, 0 otherwise
- **N**: sign bit of the result
- **V**: if there was “signed overflow”; the result cannot be represented with the number of bits we’re using

What are the cases where signed overflow can occur?

V bit

- $V$: if there was “signed overflow”; the result cannot be represented with the number of bits we’re using
- Adding two positive numbers (too big positive)
- Subtracting a negative number from a positive number (too big positive)
- Adding two negative numbers (too big negative)
- Subtracting a positive number from a negative number (too big negative)
Detecting overflow

Add these (as signed numbers).
Does overflow occur?

\[
\begin{array}{c}
0011 \\
+ 0101 \\
\hline
1000 \\
\end{array}
\]

Yes. How do we detect it?

Detecting overflow

Subtract these (as signed numbers).
Does overflow occur?

\[
\begin{array}{c}
111 \\
0011 \\
+ 0101 \\
\hline
1000 \\
\end{array}
\]

- Added two positive numbers and got a negative
- In general: if the sign bits are the same (of the numbers we end up adding), but the higher order bit of result is different = overflow
Detecting overflow

<table>
<thead>
<tr>
<th>000</th>
</tr>
</thead>
<tbody>
<tr>
<td>0011</td>
</tr>
<tr>
<td>- 1001</td>
</tr>
<tr>
<td>1010</td>
</tr>
</tbody>
</table>

Yes. How do we detect it?

- Subtracted a negative number from a positive, should have been positive
- In general: if the sign bits are the different (of the numbers we end up subtracting), but the higher order bit of result is different = overflow

Python basics

- Subtracted a negative number from a positive
- In general: if the sign bits are the same (of the numbers we end up adding), but the higher order bit of result is different = overflow