Admin

Assignment 4 due Monday at 11:59pm

Assignment 5 posted soon
  - due Friday March 11, at 5pm (before spring break)

Academic Honesty: Thanks!

Diving into your computer
One last note on CS41B

Encoding assembly instructions

What now?

Review: binary addition

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

```
  01010  
+ 01111  
  11001
```

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

```
  10  
+ 15  
  25
```

We saw before, that we can view this problem recursively. How?

SML: Binary addition

```
fun addAllListsBinary [] []      = []
| addAllListsBinary c [] []      = [c]
| addAllListsBinary c x l        = addAllListsBinary c x l []
| addAllListsBinary [] y l       = addAllListsBinary c [] y l
| addAllListsBinary (x::x3) (y::y3) =
  let
    val total = c + x + y
    in
      if total =< 2 then (* check if there’s a carry *)
        (total - 2)::addAllListsBinary x3 y3
      else
        total::addAllListsBinary x y
      end;
end;
```
SML: Binary addition

```sml
fun addAllListsBinary (x: int list) (y: int list) = ...

let total = c + x + y
in
  if total = 0 then (* check if there's a carry *)
    01010
  else
    total = addAllListsBinary 1111 y
    01111
end;

generate two pieces of information
- output bit
- carry bit
```

A recursive component

```
11 0
  + 01111
    11001
```

Adding with components

```
01010
+ 01111
```

Adding with components

```
01010
+ 01111
```
Adding with components

\[
\begin{array}{c}
0 \\
01010 \\
+01111 \\
\hline \\
1
\end{array}
\]

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

\[
\begin{array}{c}
1110 \\
01010 \\
+01111 \\
\hline \\
11001
\end{array}
\]
Implementing the component

What goes on inside the component?

Current implementation uses addition!

What are the outputs?

<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>0</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>1</td>
<td>0</td>
<td>1</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>1</td>
<td>0</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>0</td>
<td>1</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

fun addBinary @ [] [] = []
| addBinary @ x [] = x
| addBinary @ [] y = y
| addBinary @ (x::xs) (y::ys) =
  if x + y = 1 then
    addBinary (x XOR y) (xs ++ ys)
  else
    if (x = 1) THEN (y = 1) THEN
      addBinary (x OR y) (xs ++ ys)
    ELSE
      addBinary (x AND y) (xs ++ ys)
  END;

- 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>0</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 have inputs and outputs
- values are 0 or 1

They are hardware components!
Gates as hardware

Utilizing gates

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

Utilizing gates

Utilizing gates
Utilizing gates

When is this circuit 1?

A
B
A and B
A or B
not A
A nand B
A nor B
A xor B

Designing more interesting circuits

Design a circuit for this
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>0</td>
<td>1</td>
<td>1</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>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>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>

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>

Hint: solve each output bit independently

Design a circuit for this
Implementing a full adder

Implementing a full adder

Implementing 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

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

$$0010 + 1110$$
$$?$$

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

Throw away last carry bit

$$1110$$
$$0010 + 1110$$
$$0000$$

Is that right?
What numbers are these?
Signed addition

\[
\begin{array}{c}
1110 \\
0010 \\
+ 1110 \\
0000
\end{array}
\quad 2
\begin{array}{c}
0010 \\
1110 \\
2
0
\end{array}

Ripple carry adder will work for signed and unsigned numbers

Subtraction

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

How can we solve this with addition?

Subtraction

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

flip bits and add 1

0100

Do addition!

Ripple carry adder/subtractor

D = 0: addition
D = 1: subtraction

Why does this work?
Ripple carry adder/subtractor

If $D = 0$
- Carry in for first adder = 0
- $B_i \text{ XOR } 0 = B_i$

If $D = 1$
- Carry in for first adder = 1 (+1 to sum)
- $B_i \text{ XOR } 1 = \text{NOT } B_i$
  (flip all the bits of $B$)

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

```
0011
+0101
```

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

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

Yes. How do we detect it?

In general: if the sign bits are the same (of the numbers we end up adding), but the higher order bit of the result is different = overflow.

Detecting overflow

\[
\begin{array}{c}
0011 \\
0101 \\
-1010 \\
1000
\end{array}
\]

Yes. How do we detect it?

Yes. How do we detect it?

Detecting overflow

\[
\begin{array}{c}
0011 \\
-1001 \\
1010
\end{array}
\]

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

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

Python basics