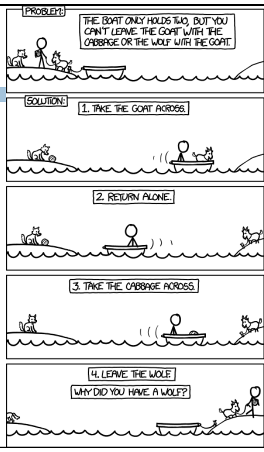


http://xkcd.com/1134/

DIGITAL CIRCUITS

David Kauchak
CS52 – Fall 2015

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

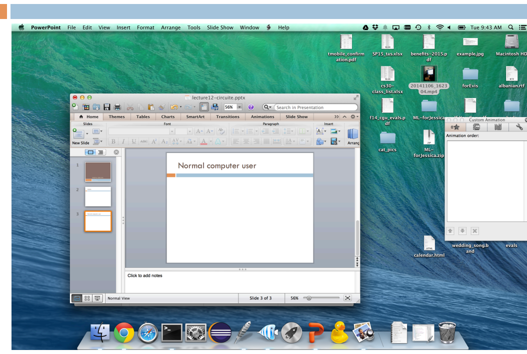
Top quartile: 26

Top half (median): 24.6

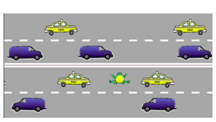
Bottom quartile: 21.4

Diving into your computer

Normal computer user



After intro CS



```java
import objectdraw.*;

public class Frog {
    // Height of the frog image
    private static final double FROG_HEIGHT = 48;

    // This should refer to the image of the frog. Note that it is not
    // initialize by
    // the code we have provided.
    private VisibleImage frogImage;

    public Frog() {

    }

    public boolean overlaps(VisibleImage vehicleImage) {
        return false; // YOU NEED TO CHANGE THIS!
    }

    public void kill() {

    }

    public void reincarnate() {

    }

    public void hopToward(Location point) {

    }

    public boolean isAlive() {
        return false; // YOU NEED TO CHANGE THIS!
    }

}
```

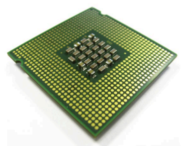
After 5 weeks of cs52

```java
import objectdraw.*;

public class Frog {
    // Height of the frog image
    private static final double FROG_HEIGHT = 48;

    // This should refer to the image of the frog. Note that it is not
    // initialize by
    // the code we have provided.
    private VisibleImage frogImage;

    public Frog() {

    }

    public boolean overlaps(VisibleImage vehicleImage) {
        return false; // YOU NEED TO CHANGE THIS!
    }

    public void kill() {

    }

    public void reincarnate() {

    }

    public void hopToward(Location point) {

    }

    public boolean isAlive() {
        return false; // YOU NEED TO CHANGE THIS!
    }

}
```

```
        loa r1 r0 0         ; get a value for a
        loa r2 r0 0         ; get a value for b
        add r3 r0 r0        ; result = 0;

        ble r1 r0 endloop   ; test if a <= 0
loop
        add r3 r3 r2        ; result += b;
        sbc r1 r1 1         ; a--;
        blt r0 r1 loop      ; return for another iteration

endloop
        sto r0 r3 0         ; write the value of product
        hlt                 ; halt
        end
```

What now?

```
        loa r1 r0 0         ; get a value for a
        loa r2 r0 0         ; get a value for b
        add r3 r0 r0        ; result = 0;

        ble r1 r0 endloop   ; test if a <= 0
loop
        add r3 r3 r2        ; result += b;
        sbc r1 r1 1         ; a--;
        blt r0 r1 loop      ; return for another iteration

endloop
        sto r0 r3 0         ; write the value of product
        hlt                 ; halt
        end
```



One last note on CS41B

```
Instruction View
0000 : I/O
0002 : 9400  loa r1 r0
0004 : 9800  loa r2 r0
0006 : cc00  add r3 r0 r0
0008 : 7106  bge r0 r1 0010
000a : cf80  add r3 r3 r2
000c : f501  sbc r1 r1 1
000e : 61fa  blt r0 r1 000a
0010 : 8300  sto r0 r3
0012 : 1000  hlt
```

memory address | binary representation of code | instructions (assembly code)

How do we get this?

Encoding assembly instructions

4	2	2	2	6
opcode	dest	src0	src1	unused

4	2	2	8
opcode	dest	src0	argument

```
Instruction View
0000 : I/O
0002 : 9400  loa r1 r0
0004 : 9800  loa r2 r0
0006 : cc00  add r3 r0 r0
0008 : 7106  bge r0 r1 0010
000a : cf80  add r3 r3 r2
000c : f501  sbc r1 r1 1
000e : 61fa  blt r0 r1 000a
0010 : 8300  sto r0 r3
0012 : 1000  hlt
```

1001 0100 0000 0000

1001 1000 0000 0000

1100 1100 0000 0000

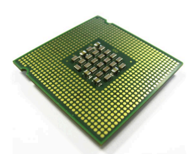
opcode dest src0 src1

What now?

1001 0100 0000 0000

1001 1000 0000 0000

1100 1100 0000 0000



Review: binary addition

```
 01010
+01111
------
   ?
```

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

Review: binary addition

```
 1110
 01010
+01111
------
 11001
```

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

Review: binary addition

```
 1110
 01010     10
+01111     15
------
 11001     25
```

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

```
fun addAsListsBinary 0 nil     nil     = nil
  | addAsListsBinary c nil     nil     = [c]
  | addAsListsBinary c xl      nil     = addAsListsBinary c xl [0]
  | addAsListsBinary c nil     yl      = addAsListsBinary c [0] yl
  | addAsListsBinary c (x::xs) (y::ys) =
    let
        val total = c + x + y
    in
        if total >= 2 then (* check if there's a carry *)
            (total - 2)::addAsListsBinary 1 xs ys
        else
            total::addAsListsBinary 0 xs ys
    end;
```

```
fun addAsListsBinary 0 nil    nil    = nil
  | addAsListsBinary c nil    nil    = [c]
  | addAsListsBinary c xl     nil    = addAsListsBinary c xl [0]
  | addAsListsBinary c nil    yl     = addAsListsBinary c [0] yl
  | addAsListsBinary c (x::xs) (y::ys) =
    let
        val total = c + x + y
    in
        if total >= 2 then (* check if there's a carry *)
            (total - 2)::addAsListsBinary 1 xs ys
        else
            total::addAsListsBinary 0 xs ys
    end;
```

```
   1 1 1 0
   01010
 + 01111
 -------
   11001
```

handle a digit at a time

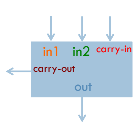
```
fun addAsListsBinary 0 nil    nil    = nil
  | addAsListsBinary c nil    nil    = [c]
  | addAsListsBinary c xl     nil    = addAsListsBinary c xl [0]
  | addAsListsBinary c nil    yl     = addAsListsBinary c [0] yl
  | addAsListsBinary c (x::xs) (y::ys) =
    let
        val total = c + x + y
    in
        if total >= 2 then (* check if there's a carry *)
            (total - 2)::addAsListsBinary 1 xs ys
        else
            total::addAsListsBinary 0 xs ys
    end;
```

```
   1 1 1 0
   01010
 + 01111
 -------
   11001
```

generate two pieces of information
- output bit
- carry bit

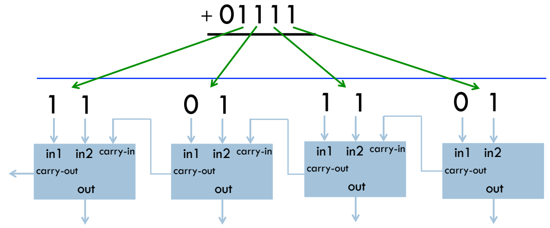
A recursive component

```
   1 1 1 0
   01010
 + 01111
 -------
   11001
```



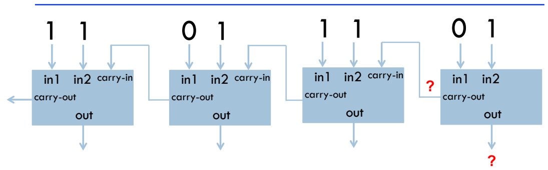
Adding with components

```
   01010
 + 01111
```



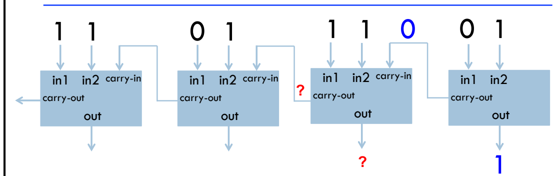
Adding with components

```
 01010
+01111
```



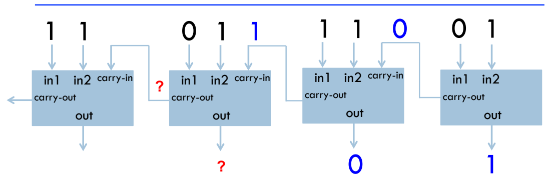
Adding with components

```
    0
 01010
+01111
     1
```



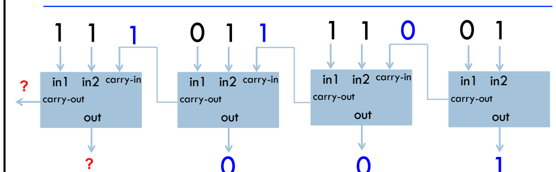
Adding with components

```
   10
 01010
+01111
   01
```



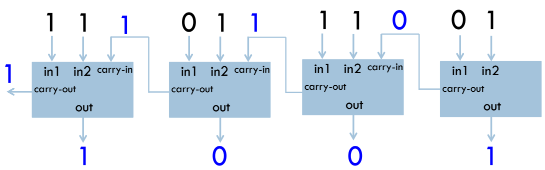
Adding with components

```
  110
 01010
+01111
  001
```

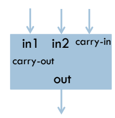


Adding with components

```
  1110
  01010
+ 01111
  11001
```

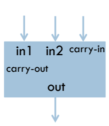


Implementing the component



What goes on inside the component?

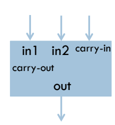
Implementing the component




```
let
    val total = c + x + y
in
    if total >= 2 then (* check if there's a carry *)
        (total - 2)::addAsListsBinary 1 xs ys
    else
        total::addAsListsBinary 0 xs ys
end;
```

Current implementation uses addition!

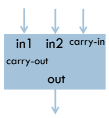
Implementing the component



in1	in2	carry-in	out	carry-out
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

What are the outputs?

Implementing the component



in1	in2	carry-in	out	carry-out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Another implementation

```
fun addAsListsBinary 0 nil     nil     = nil
  | addAsListsBinary c nil     nil     = [c]
  | addAsListsBinary c xl      nil     = addAsListsBinary c xl [0]
  | addAsListsBinary c nil     yl      = addAsListsBinary c [0] yl
  | addAsListsBinary c (x::xs) (y::ys) =
    if x = 1 andalso y = 1 andalso c = 1 then
       1::(addAsListsBinary 1 xs ys)
    else if (x = 1 andalso y = 1) orelse
            (x = 1 andalso c = 1) orelse
            (y = 1 andalso c = 1) then
       0::(addAsListsBinary 1 xs ys)
    else if x = 1 orelse y = 1 orelse c = 1 then
       1::(addAsListsBinary 0 xs ys)
    else
       0::(addAsListsBinary 0 xs ys);
```

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

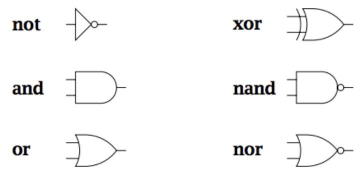
What are some boolean operators?

A	B	A and B	A or B	not A
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

What are some boolean operators?

A	B	A and B	A or B	not A	A nand B	A nor B	A xor B
0	0	0	0	1	1	1	0
0	1	0	1	1	1	0	1
1	0	0	1	0	1	0	1
1	1	1	1	0	0	0	0

Gates

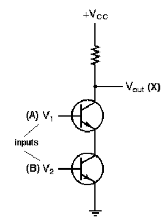


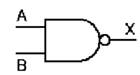
Gates have inputs and outputs

- values are 0 or 1

Are **hardware** components!

Gates as hardware

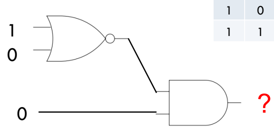


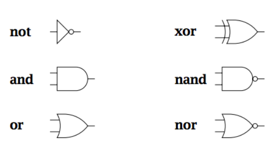


A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

Utilizing gates

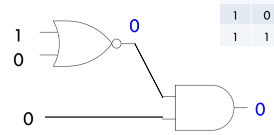
A	B	A and B	A or B	not A	A nand B	A nor B	A xor B
0	0	0	0	1	1	1	0
0	1	0	1	1	1	0	1
1	0	0	1	0	1	0	1
1	1	1	1	0	0	0	0

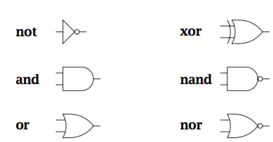




Utilizing gates

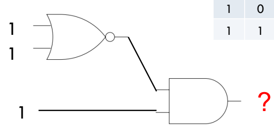
A	B	A and B	A or B	not A	A nand B	A nor B	A xor B
0	0	0	0	1	1	1	0
0	1	0	1	1	1	0	1
1	0	0	1	0	1	0	1
1	1	1	1	0	0	0	0

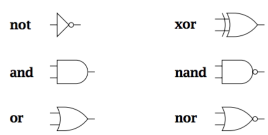




Utilizing gates

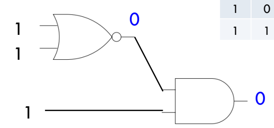
A	B	A and B	A or B	not A	A nand B	A nor B	A xor B
0	0	0	0	1	1	1	0
0	1	0	1	1	1	0	1
1	0	0	1	0	1	0	1
1	1	1	1	0	0	0	0



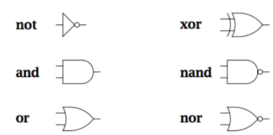


Utilizing gates

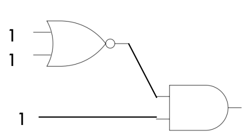
A	B	A and B	A or B	not A	A nand B	A nor B	A xor B
0	0	0	0	1	1	1	0
0	1	0	1	1	1	0	1
1	0	0	1	0	1	0	1
1	1	1	1	0	0	0	0



When is this circuit 1?



Utilizing gates



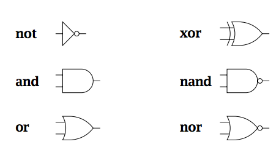
in1	in2	in3	OUT
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Designing more interesting circuits

A	B	A and B	A or B	not A	A nand B	A nor B	A xor B
0	0	0	0	1	1	1	0
0	1	0	1	1	1	0	1
1	0	0	1	0	1	0	1
1	1	1	1	0	0	0	0

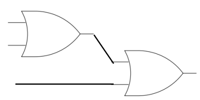
in1	in2	in3	OUT
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Design a circuit for this



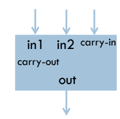
Designing more interesting circuits

in1	in2	in3	OUT
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1



Back to addition…

in1	in2	carry-in	carry-out	sum
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



A half-adder: no carry-in

A	B	carry	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

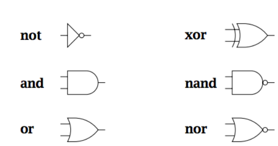
A half-adder: no carry-in

A	B	A and B	A or B	not A	A nand B	A nor B	A xor B
0	0	0	0	1	1	1	0
0	1	0	1	1	1	0	1
1	0	0	1	0	1	0	1
1	1	1	1	0	0	0	0

A	B	carry	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

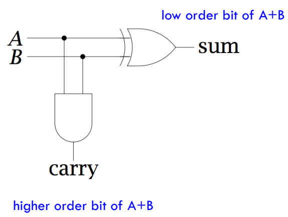
Hint: solve each output bit independently

Design a circuit for this

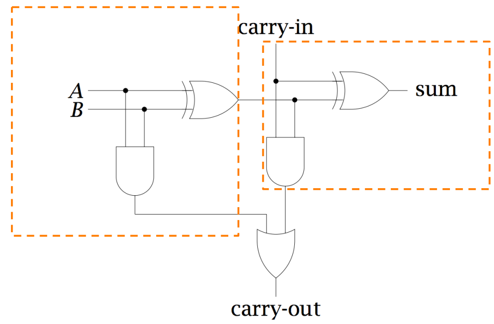


A half-adder: no carry-in

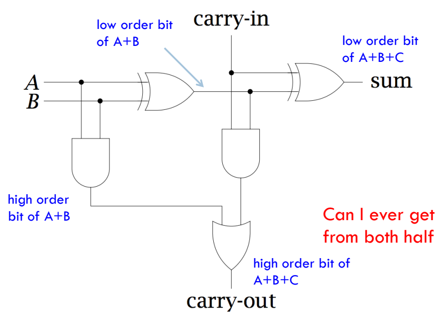
A	B	carry	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



Implementing a full adder

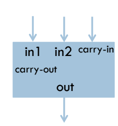


Implementing a full adder



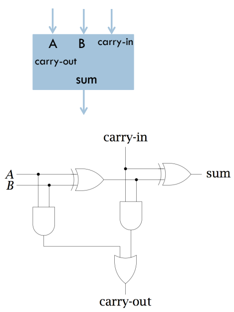
Can I ever get a carry from both half adders?

Implementing the component



What goes on inside the component?

Implementing the component



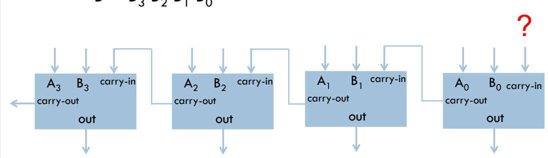
Ripple carry adder

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



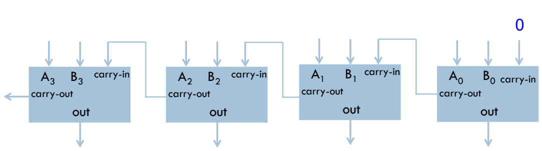
Ripple carry adder

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

$$
\begin{array}{r}
0010 \\
+\ 1110 \\
\hline
?
\end{array}
$$

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

Signed addition

throw away last carry bit → 1110

```
  0010
+ 1110
  ----
  0000
```

Is that right?
What numbers are these?

Signed addition

```
  1110
  0010     2
+ 1110    -2
  ----
  0000     0
```

Ripple carry adder will work for signed and unsigned numbers

Subtraction

```
  0010
- 1110
  ----
```

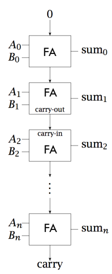
We can solve this doing addition

Subtraction

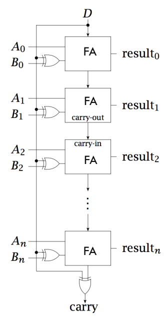
```
  0010              0010
- 1110    ⇒         0010
  ----  flip bits   ----
        and add 1   0100
```

Do addition!

Ripple carry adder/subtractor



D = 0: addition
D = 1: subtraction



Why does this work?

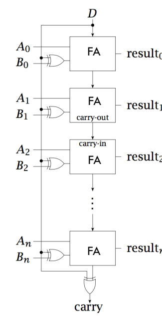
Ripple carry adder/subtractor

If D = 0

- Carry in for first adder = 0
- B_i XOR 0 = B_i

If D = 1

- Carry in for first adder = 1 (+1 to sum)
- B_i XOR 1 = 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

```
  111
   0011
 + 0101
 ------
   1000
```

Yes. How do we detect it?

Detecting overflow

```
  111
   0011
 + 0101
 ------
   1000
```

- 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

```
   0011
 - 1001
 ------
```

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

Detecting overflow

```
  000
  0011
- 1001
  ----
  1010
```

Yes. How do we detect it?

Detecting overflow

```
  000
  0011
- 1001
  ----
  1010
```

- 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

Detecting overflow

```
  100
  0011
- 1101
  ----
  0110
```

- 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

Python basics