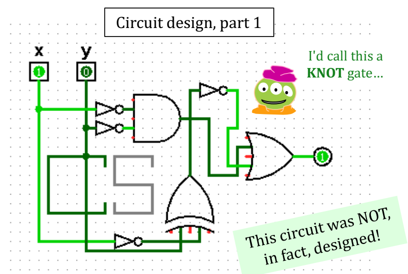
More *bits* of CS

Reading:
Sections 4.2.3 thru 4.2.6

Too many bits? *Compress!*

Below binary: ***physical circuits***



Hw #4 due Monday

- *pr0 (reading)* A bug and a crash!
- *pr1 (lab)* binary ~ decimal
- *pr2* conversion + compression
- *extra* image processing...

Lots of tutoring hrs - join in... !

P/F vs T/F

Bits' big idea

Aha! This can be implemented *just with wiring!*

Take-home Concept	Python	Bitwise reason (purely mechanical)
left-shifting by 1 ***doubles*** a value	`42 << 1` `84`	`'101010'` 42 in binary, columns double in value leftward `'1010100'` 84
right-shifting by 1 ***halves*** a value	`42 >> 1` `21`	`'101010'` 42 in binary, columns halve in value rightward `'10101'` 21

Do I halve to remember this?

I hope I don't have to remember L vs R!

No - *it falls out!*

Lab Debriefing &
`hw4pr2.py`

in
42

```
ntb(42)
ntb(21) + '0'
ntb(10) + '1'
ntb(5) + '0'
ntb(2) + '1'
ntb(1) + '0'
ntb(0) + '1'
''
```

'101010'
out

```
def numToBin(N):
    """Converts a decimal int to a binary string
    """
    if N == 0:      return ''
    elif N%2 == 0:  return numToBin(N//2) + '0'
    elif N%2 == 1:  return numToBin(N//2) + '1'
```

these are awfully similar...

How high can we count...?

I can see some patterns here – even *with one eye closed!*

with	**1 bit**	1	1
	2 bits	11	3
	3 bits	111	7
	4 bits	1111	15
	7 bits	1111111	127
	8 bits	11111111	255
	N bits		
	31 bits		

Insight Ancient Egyptian Multiplication

halver	dbler	
21	× 6	(ans. should be 126)

halver	dbler
21	6

Example

AEM/RPM algorithm

Write the factors in two columns.

Repeatedly **halve** the LEFT and **double** the RIGHT. (toss remainders...)

Pull out the RIGHT values where the LEFT values are ***odd***.

Sum *those values for the answer!*

Why does this work?

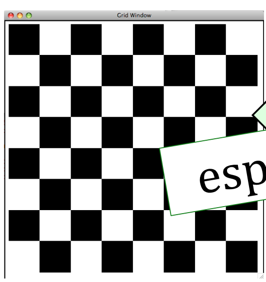
a.k.a. *Russian Peasants' Multiplication*

Insight AEM algorithm

Decimal		Binary	
21	6	110_6	
		x 10101_{21}	
21	6	110_6	6
10	12	0000_{12}	
5	24	11000_{24}	24
2	48	000000_{48}	
1	96	+ 1100000_{96}	+ 96
		1111110_{126}	126

Although in ancient Egypt the concept of base 2 did not exist, the algorithm is essentially the same algorithm as long multiplication after the multiplier and multiplicand are converted to binary. The method as interpreted by conversion to binary is therefore still in wide use today as implemented by binary multiplier circuits in modern computer processors.

Hw4: *images are just bits, too*!



10101010
01010101
1010...
...01010
01010101
10101010
01010101

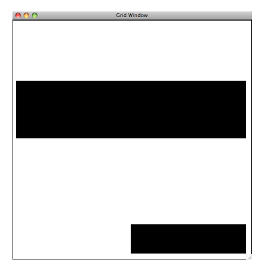
especially ***binary*** images

Binary Image

Encoding as raw bits
one big string of 64 characters

"1010101001010101101010100101010110101010010101011010101001010101"

Hw4: *lossless* binary image compression



00000000
00000000
11111111
11111111
00000000
00000000
00000000
00001111

same-data streaks

Binary Image

Encoding as raw bits
one big string of 64 characters

If our images tend to have long streaks of unchanging data, how might we represent it more efficiently, ***but still in binary***?

compress
uncompress

"0000000000000000111111111111111100000000000000000000000000001111"

If you use ***7 bits*** to hold the # of consecutive repeats, what is the largest number of bits that *one block can represent*?

00010000

1 bit: the initial pixel | 7 bits: # of repeats

8-bit total data block

7 bits?

B bits?

*What if you need a **larger** # of repeats?*

hw4 pr2

a binary image

"0000000000000000111111111111111100000000000000000000000000001111"

16 zeros | 16 ones | 28 zeros | 4 ones

```
def compress(I):
    """Returns the RLE of the
       given binary image I"""
```

What helper function might be useful here?

the "compressed" image:

"00010000100100000001110010000100"

16 | 16 | 28 | 4

```
def uncompress(CI):
    """Returns the binary image I
       from the run-length-encoded,
       "compressed" argument CI"""
```

back to the original binary image

"0000000000000000111111111111111100000000000000000000000000001111"

16 zeros | 16 ones | 28 zeros | 4 ones

Try it!

`frontNum(S)` should return the number of times the first element of the argument **S** appears consecutively ***at the start*** of S:

Try writing the recursive function, `frontNum(S)`

Examples…

```
>>>   frontNum('1111010')
4
>>>   frontNum('00110010')
2
```

or 2 base cases:

1 base case:

```
def frontNum(S):

  if  len(S) <= 1:    len(S)== 0:
                      len(S)== 1:
    return

  elif  S[0] == ______ :
    return

  else:
    return
```

shortest longest

What are the **BEST** / **WORST** compression results you can get for an 8x8 input image (64 bits)?

EXTRA! Can you change our algorithm so that compressed images are always smaller than the originals?

It's all bits!

images, text, sounds, data, ...

even the string 'forty*two' is represented
as a sequence of bits...

'forty*two'

011001100110111101110010011101000111100100101010011101000111011101101111

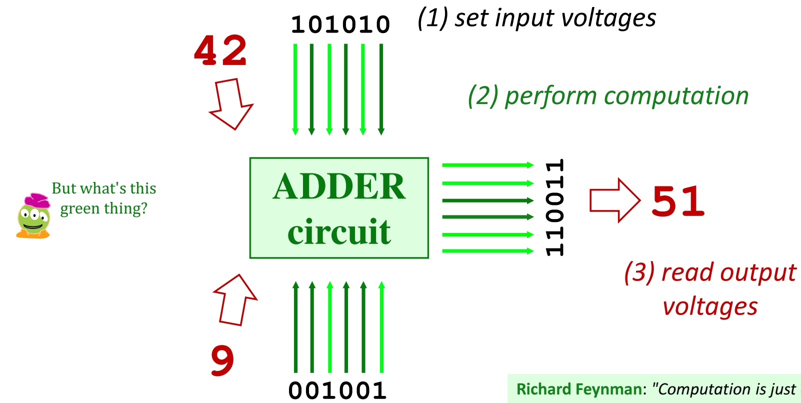
9 ASCII characters
8 bits each

9*8 == 72 bits total

All computation boils down to manipulating bits!

In a computer, each bit is represented as a <u>voltage</u>
(**1** is +3v and **0** is 0v)

Computation is simply the **deliberate combination** of those voltages!



Richard Feynman: *"Computation is just a physics experiment that always works!"*

All computation

is simply ***functions of bits***

binary inputs **A** and **B** → output, **A+B**

A	B	A+B
00	00	000
00	01	001
00	10	010
00	11	011
01	00	001
01	01	010
01	10	011
01	11	100
10	00	010
10	01	011
10	10	100
10	11	101
11	00	011
11	01	100
11	10	101
11	11	110

This week: you'll build this in Python.

bitwise addition function

addB

Next week: you'll design this with *wires*.

Carrying on...

hw4: **addB**

S: '23'   T: '19'

```
def add10(S, T):
    """Adds the *strings* S and T
       as decimal numbers
    """
    if len(S) == 0: return T
    if len(T) == 0: return S
    eS = S[-1]
    eT = T[-1]
    if   eS == '0' and eT == '1': return  add10(S[:-1], T[:-1]) + '1'
    elif eS == '1' and eT == '1': return  add10(S[:-1], T[:-1]) + '2'
    elif eS == '2' and eT == '1': return  add10(S[:-1], T[:-1]) + '3'
    elif eS == '3' and eT == '1': return  add10(S[:-1], T[:-1]) + '4'
    # what if we have to carry to the next column?
    elif eS == '3' and eT == '9':
       return
```

eS ~ the "end of S"   **eT** ~ the "end of T"

S '23'
T '19'

Notice that this code doesn't "understand" addition at all!

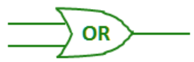
Our building blocks: ***logic gates***

AND outputs 1 only if **ALL** inputs are 1

AND

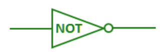
OR outputs 1 if **ANY** input is 1

OR



NOT reverses its input

NOT



These circuits are ***physical*** functions of bits…

… and *all* mathematical functions can be built from them!

Name(s) ______________________

Quiz

Ancient Egyptian Multiplication!

halver	dbler	(ans. should be 126)
21 ×	6	
21	6	6
10	12	
5	24	24
2	48	
1	96	+ 96
		126

Example

AEM algorithm

Write the factors in two columns.

Repeatedly **halve** the LEFT and **double** the RIGHT. (toss remainders...)

Pull out the RIGHT values where the LEFT values are ***odd***.

***Sum** those values for the answer!*

halver dbler

11 × 15 (ans. ~ 165)

Try it!

halver dbler

12 × 20 (ans. ~ 240)

Extra: *Why does this always work?* **Hint**: it's binary!