# Digital circuits

$323 + 189 = 512$

$213 - 139 = 184$

sign 21 for compliments

- analog is continous and variable
- digital is characterized by two continous voltages (two states are defined by the circuit being above or below a certain value)
  - immune to noise, just need to be above threshold

to represent this signal level we only need to use 1 or 0

we commonly use base 10 / for binary

$1024_{10} = 1\cdot10^3 + 0\cdot10^2 + 2\cdot10^1 + 4\cdot10^0$

$1011_2 = 1\cdot2^4 + 0\cdot2^3 + 1\cdot2^2 + 1\cdot2^1 + 0\cdot2^0 = 22_{10}$

adding and subtracting → in base 10 if we do 9+4 [we have to add 1 to get to 10 then add 3 → 13] (base)

so if you add 1+1 in binary you get $10_2$ (passed the base)

adding →

```
  1 1 1
  1001
+ 1111
------
 11000
```

put 0 and carry the 1

subtracting

```
  1 1 1 10
- 1 0 0 1
---------
  1 0 1
```

10 - 1 = 9, 3 - 2 = 1

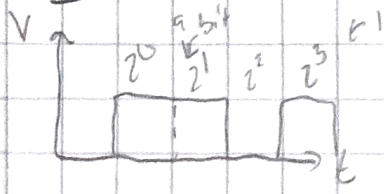
→ a binary digit is a "bit"? How can we express all these digits electronically?

- parallel - each digit or bit of a number is represented simultaneously by a voltage in the circuit

(ex) digital circuit with inputs $i^3, i^2$ and outputs $i^3, i^2, i^1, i^0$

the more precision you want in a number, ie # of digits you need more lines

- serial - sends a time sequence of highs and lows




1011

in order for it to work the sender and reciever need to agree on 4 things

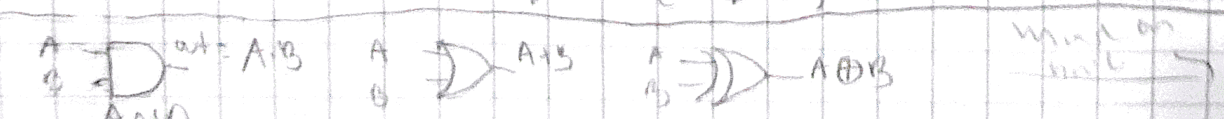
1. how many bits are going to be sent
2. what digital low corresponds to one bit
3. the interval between bits
4. how will the start of the number be recognized

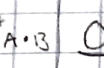
## Logic gates and boolean algebra

And gate - high only when all of inputs are high $(A \cdot B)$

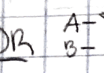
OR gate - high when any input is high $(A + B)$

XOR gate - exclusive OR gate, high when either input is high but not when both are high, $(A \oplus B)$

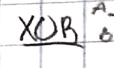


And  $A \cdot B$

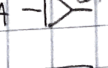
| A | B | out |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 1 | 1 |

OR  $A+B$

| A | B | out |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 1 |

XOR  $A \oplus B$

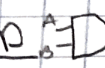
| A | B | out |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 0 |

 $out = A$

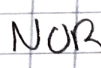
| A | out |
|---|---|
| 0 | 0 |
| 1 | 1 |

 $\overline{A}$

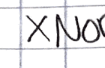
| A | out |
|---|---|
| 0 | 1 |
| 1 | 0 |

NAND  $\overline{A \cdot B}$

| A | B | out |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 0 |

NOR  $\overline{A+B}$

| A | B | out |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 1 | 0 |

XNOR  $\overline{A \oplus B}$ or $A \odot B$

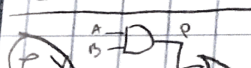
| A | B | out |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 1 | 1 |

All these are the same as the ones above except the outputs are reversed

XOR is also $A\overline{B} + \overline{A}B$ or $A'$ is used as inverse
XNOR is also $AB + \overline{A}\overline{B}$

NAND = 1 unless A and B are 1
NOR = 0 unless A and B are 0

XOR = 1 when A and B are different
XNOR = 0 when A and B are different

ex 

[for # of inputs use $2^n$ $n^{th}$ inputs]

and's out

| A | B | C | P | Q |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |

out

1 when P or C or both or is 1

0 except for when bot A and B are 1

Karnaugh map

| C \ AB | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

combos of AB

combos of C

only one digit at a time is changes as we write various combos (arbitrary) ↓start w/ 00 then change to 01 etc..

ex: when AB are 00, C is 0, out is what

after you make it → look for groups of 2, 4, 8, 16 → only horizontal and vertical no diagonal

| C \ AB | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

this group is always when C=1 for any A or B

"or"

when C=0 the output is only 1 when A & B are 1

$Q = (A \cdot B) + C$

obvious from circuit but useful tool for when it gets complex

copied karnaugh map

# Quick tutorial on how computers figure out sign

↳ the compliments

**definitions:** ($n$ = # of digits of machine)

$$\bar{N}_{2c} \equiv 2^n - N \quad \text{(two's compliment)}$$

$$\bar{N}_{1c} \equiv 2^n - 1 - N = \bar{N}_{2c} - 1 \quad \text{(1's compliment)}$$

for number $-N$:

$\bar{N}_{2c} \equiv 2^n - N \rightarrow N = 2^n - \bar{N}_{2c}$

$\bar{N}_{1c} \equiv 2^n - 1 - N \quad N = 2^n - \bar{N}_{1c} - 1$

$N_{1c} - N_{2c} = -1$

ex) $N = 0010\ 1101_2$, $n = 8$

(1's) $\bar{N}_{1c} = 2^n - 1 - N$ easy calculate trick → invert 1's and 0's → $1101\ 0010_2$

(2's) $\bar{N}_{2c} = \bar{N}_{1c} + 1$ → add 1 to $\bar{N}_{1c}$: $1101\ 0010 + 0000\ 0001 = 1101\ 0011_2$

to calculate $A - B$ a computer instead $A - B = A + \bar{B}_{2c} = A + \bar{B}_{1c} + 1$

a sign bit doesn't make the math work out when adding binary

## Ben Eater Video

[ones compliment] ← this instead inverts all of bits to be negative

↑ adding 1 is one off the and there's +0 and -0

1000 -7
1001 -6
1010 -5
1011 -4
1100 -3
1101 -2
1110 -1
1111 -0
0000 +0
0001 +1
0010 +2
0011 +3
0100 +4
0101 +5
0110 +6
0111 +7

1110 -1 [missed one whoops]

0101 5 + 1010 (-5) = 1111 = -0

0011 3 + 1100 (-3) = 1111

0101 5 + 1100 (-3) = 1 0001 = 1

extra bit forget it

1 off so they invented the two's compliment

[two's compliment] * no +0 and -0 and it adds correctly

1000 -8
1001 -7
1010 -6
1011 -5
1100 -4
1101 -3
1110 -2
1111 -1
0000 0
0001 1
0010 2
0011 3
0100 4
0101 5
0110 6
0111 7

0101 5 + 1011 (-5) = 0000

0110 6 + 1110 (-2) = 0100 4

-8 4 2 1

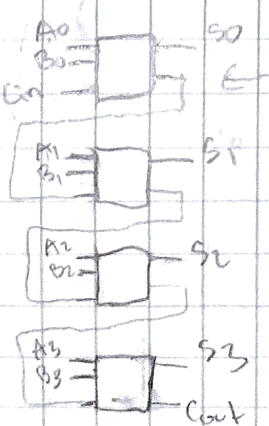
also ex → 1010 -6 (negative 8's place) -8 + 2 = -6

kinda of a sign bit but not really

* to make a number negative
ex 0101 +5
invert 1010
+1 1011 -5

* -0 + 1 = 0

Full Adder ex




Carry out here goes into the next one like you are doing it by hand.

* like you can add 2 4bit numbers

# Adder Circuits

refresh on adding:

$0+0=0$ ; $0+1=1$ ; $1+0=1$ ; $1+1=10$ (sum 0, carry the 1)

A, B → [ ] → Sum, Carry

A + B = (carry) Sum

Pretty much can only do 1+1

| XOR | A | B | Out |
|---|---|---|---|
| | 0 | 0 | 0 |
| | 1 | 0 | 1 |
| | 0 | 1 | 1 |
| | 1 | 1 | 0 |

→ looks like sum!

Now how do we do carry?

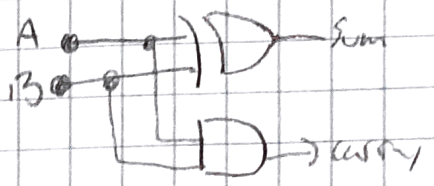
$00+00=00$ ; $00+01=01$ ; $01+00=01$ ; $01+01=10$

adding 1 bit you need 2 bits

So when both A and B are 1 we need that to be our carry

and

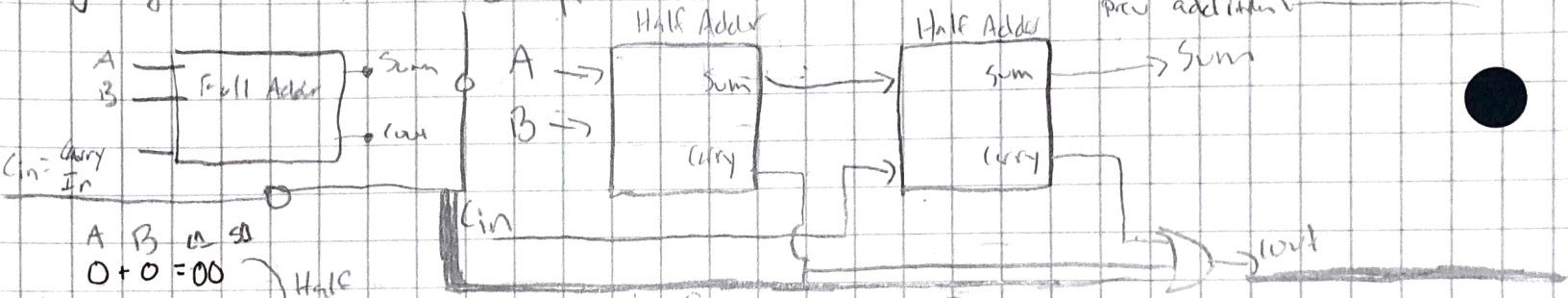
thus a half adder looks like




---

Addition on 3 binary digits

- you give it A and B plus whatever its carry is from prev addition




| A | B | C | S |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

Half Adder

| Cin | | | | |
|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

↑ inverted sum bit

Cout ↑ Sum

Step 1: draw half adder

Step 2: $S_2$ is just inverse of $S_1$ if there is a Cin

→ use another XOR → look at truth table, when (B = Cin) = 1, A is inverted and that's what we want

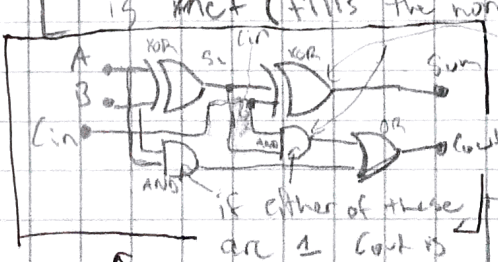
Sum bit is inverted when Cin = 1

(even though it works) xor gate does this

1 + 0 + 0 = 0
1 + 0 + 1 = 0
1 + 1 + 0 = 0
1 + 1 + 1 = 1

Step 3 for carry: if A or B is 1 while Cin = 1 we need to output 1

→ also carry out is 1 when this condition or the normal and gate is met (fills the normal half adder conditions for carry)




Full adder

when A or B is 1 while Cin is 1 we want to output 1 for carry

if either of these are 1, Cout is 1

In order to do a full adder you need to take a carry bit in, then you can add n-bits by putting full adders together since they can handle carrying as you down the add

An adder for a column like

1 1 1 0 0 ← can carry like this

1 0 1 1 0

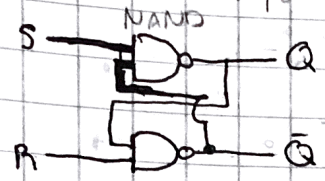
0 1 1 0

for multiple bits you can stack these together!

# Flip flops

| A | B | NAND |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

7:10 USED to STORE SINGLE BITS OF DATA




- needs output for input let's say R=S=1 for now and look at different starting Q's
- it remembers which input was low last by looking at Q's and S-R

[keeping R and S high then switching it]

| | R | S | Q | $\bar{Q}$ | |
|---|---|---|---|---|---|
| 1 start | 1 | 1 | 0 | 1 | |
| 1 end | 1 | 0 | 1 | 0 | ← making S low causes Q's to flip |
| 2 start | 1 | 1 | 1 | 0 | ← flip Q's from case 1 |
| 2 end | 1 | 0 | 1 | 0 | ← making S low has no effect |
| 3 start | 1 | 1 | 1 | 0 | |
| 3 end | 0 | 1 | 0 | 1 | ← making reset low flips |
| 4 start | 1 | 1 | 0 | 1 | |
| 4 end | 0 | 1 | 0 | 1 | ← making R low stays |

- if both S and R are high no change in Q's
- if S is high and R is low Q->0 and $\bar{Q}$->1
- if S is low and R is high Q->1 and $\bar{Q}$->0
- if both are low Q->1 and $\bar{Q}$-> which is bad for bit storage (idk why)

## S'-R-Latch

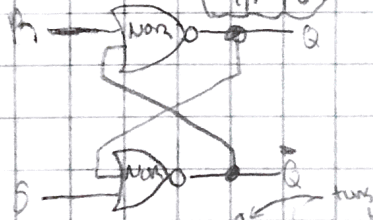

| A | B | out |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | ? |

- if everything starts off low LED=out=0
- if you turn A on out turns on so 10 -> 1 then out turns B on so 11 -> 1
- if you turn A off LED will stay ON! cus B is on

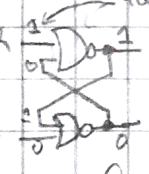
first quick look at
- it will stay on unless you disconnect power so lets look at SR latch now

## SR latch

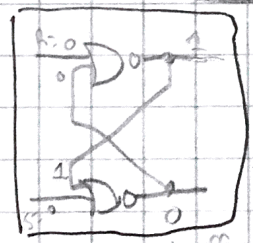
| A | B | out |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |




now lets make R=1

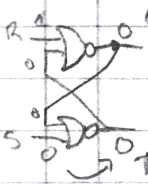



① start like this

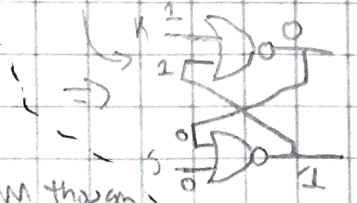



0,0 outputs a 1 which feeds into bottom NOR

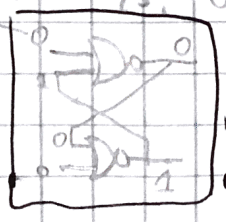
0,1 outputs a 0 so it stays off




inputs into R are not.




if you release the gate it stays! 0,1 -> 0

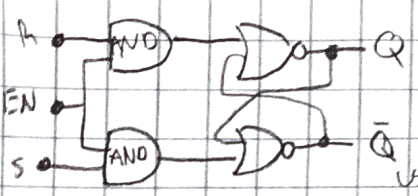



top LED off now bottom ON!

if you turn this to 1 no affect 1,1 -> 0 still

- if you press R and S at same type both go low invalid state
- both start 0,0 so they want to turn on but one will be slightly faster and output 1 which turns off the other.
- if you have nail on it holds

# SR Latch w/ Enable



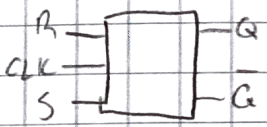

Slight modification

| R | S | EN | Q | $\bar{Q}$ | |
|---|---|---|---|---|---|
| 0 | 0 | 1 | Last state | | ← last state if no button is pressed |
| 1 | 0 | 1 | 0 | 1 | |
| 0 | 1 | 1 | 1 | 0 | |
| 1 | 1 | 1 | 0 | 0 | |
| 0 | 0 | 0 | Last state | | always outputs 0 when EN = 0 causing stay |
| 0 | 1 | 0 | Last state | | |
| 1 | 0 | 0 | Last state | | |
| 1 | 1 | 0 | Last state | | |

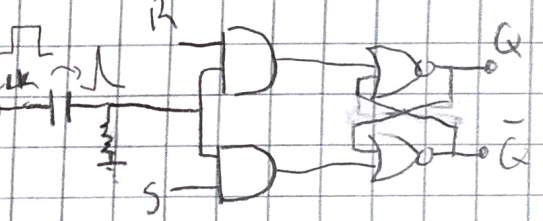
Usually invalid state →

[otherwise w/ enable ON it acts like normal]

# S-R Flip Flop




- Latch will change whenever input changes
- Flip Flop the output only changes w/ a clock pulse
- Quick pulse enables gates to change for a short amount of time

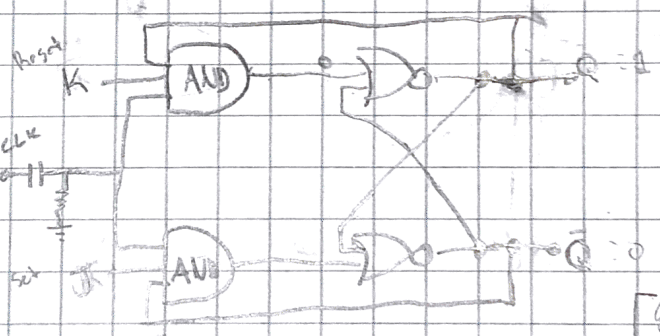



| R | S | CLK | Q | $\bar{Q}$ | |
|---|---|---|---|---|---|
| 0 | 0 | ∫ | last state | | |
| 1 | 0 | ∫ | 0 | 1 | |
| 0 | 1 | ∫ | 1 | 0 | |
| 1 | 1 | ∫ | 0 | 0 | ? Random when it releases |
| X | X | X | Last state | | CLK 0 |

Invalid state

# J-K flip flop

→ makes this state a little more predictable




| J | K | CLK | Q | $\bar{Q}$ | |
|---|---|---|---|---|---|
| 0 | 0 | ∫ | last state | | |
| 0 | 1 | ∫ | 0 | 1 | |
| 1 | 0 | ∫ | 1 | 0 | |
| 1 | 1 | ∫ | Toggle | | |
| X | X | X | Last state | | |

- if J and K are 0 AND's will be 0, [goes to Latch and just stays on Last State]

Case latch set ← and want to RESET

So Q = 1 and $\bar{Q}$ = 0

↳ Q = 1, K = 1, CLK = 1 → [AND = 1 goes to Preset signal] → turns Q = 0 then $\bar{Q}$ = 1

[if you try to RESET = K it while it's already in RESET Mode → Q: 0, $\bar{Q}$ = 1, AND gates would turn on due to the feedback which is ok]

[or if in reset it will set if in set it will reset]

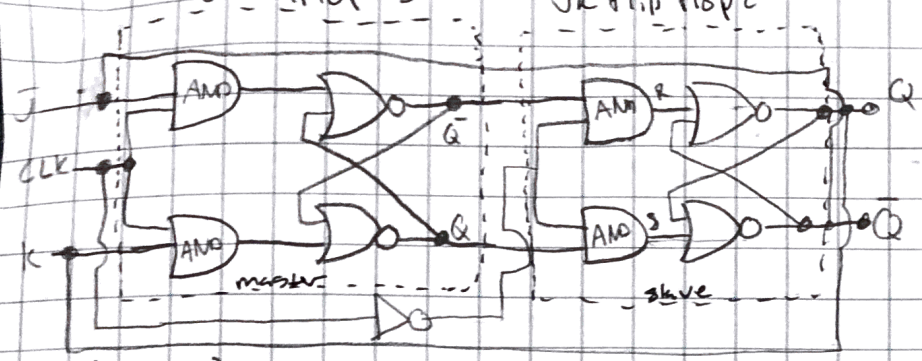
- if both are 1 → it will toggle if from whatever state it was in

↳ if clock pulse is too long, (for toggling) it can cause a feed back and make toggling inconsistent, output will race and switch fast (Racing)

(pulse is from HL constant)

Fix w/ master slave flip flop

# Master - Slave JK flip flop




- 2 JK flip flops together
- an inverter of clock from JKFF1 to JKFF2

*note if Op gets passed it * STAYS

(Master)
- JKFF1 isnt active unless clock is high [J will pass through and set / K will pass through and Reset]

(Slave)
- Inputs for JKFF2 come from JKFF1 and JKFF2's clock is the inverted one of Master
  - ↳ slave is only active when clock is low

setting master slave

ex 4 If you set J high while clock is high [master is activated]
- ↳ master activated causes its Q and $\bar{Q}$ to flip (Q -> 1 and $\bar{Q}$ -> 0)

2 when clock goes low slave is activated
- ↳ the masters Q's get passed into slave latch
- ↳ the high Q sets and makes Q go high and $\bar{Q}$ low

so by making J go high and clock pulses Q turns on

works directly back

flips Q and $\bar{Q}$

- Resetting is same but flipped
- Toggling -> the feedback of Q and $\bar{Q}$ still cause it to toggle

J sets [Q -> 1 or $\bar{Q}$ -> 0]
K resets [Q -> 0 or $\bar{Q}$ -> 1]

J makes the Q -> 1 normally but feedback [makes Q -> 1 = $\bar{Q}$ for J / makes Q -> 0 = $\bar{Q}$ for K]

Feed back does is K makes the Q -> 0

[Clock goes high to set Master latch and when its low it goes to Slave, Thus no racing problem! (for it to output the Q, it must do clock goes low)]

↳ only toggles once per clock cycle

[With these you can build a binary counter and then with logic gates and configurations you can build a computer
↳ clock speed on these things is very important]

more