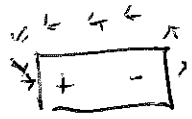
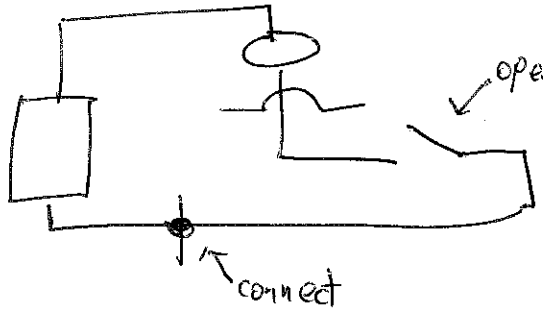
Class notes

3/30



some things conduct electrons ⇒ circuits with wires.

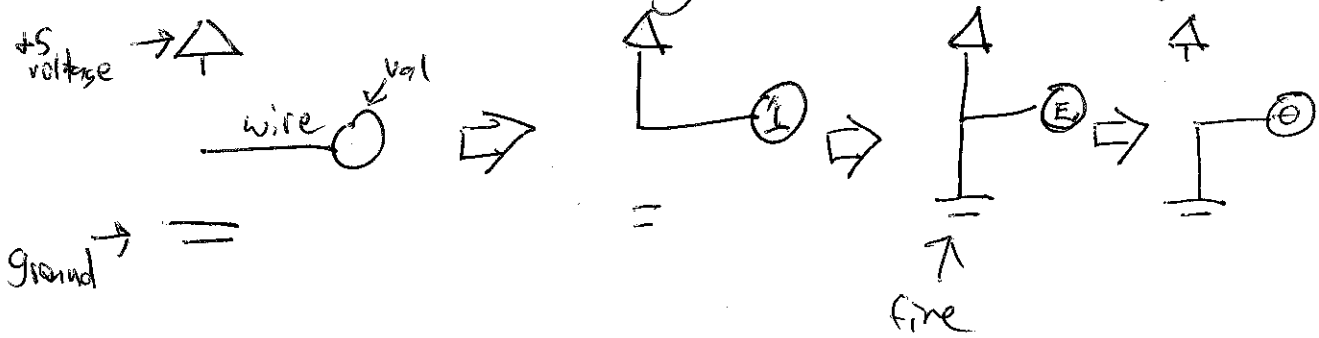


electrons can flow in a circuit.
electrons circle through.

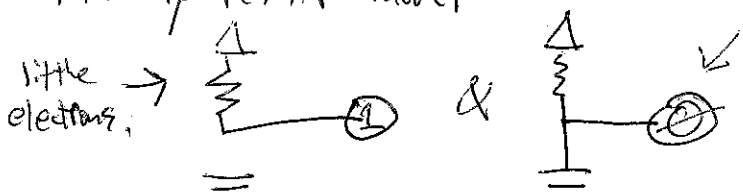
resistor (resists electrons flowing through).

At a low level, you want to think about bits.
we're going to use prop.s of electricity to make computation machines ⇒ how does electricity represent bits?

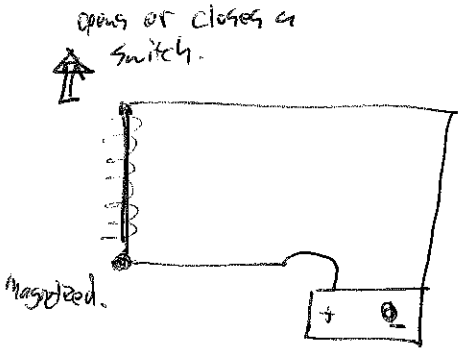
If you have a wire carrying voltage @ +5, that's a 1!
If it's connected to the ground, that's a 0.



Pull-up resistor model



lots of electrons flow but not enough to melt



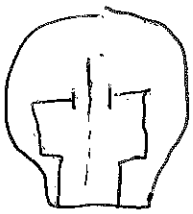
relay

BUT moth can insulate. ☹

⇨ Grace

This is old school.

Consumes power, slow, noisey

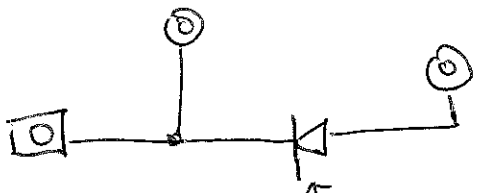


← vacuum tubes

Improvement, but takes time to warm up.

Still limited.

THEN, 3 people made a transistor!



↙ diode

↑ semi conductor

THEN, to transistor p-type & n-type

Difference b/t circuit & subcircuit?

3/30 p.2

Elect. → 1's & 0's

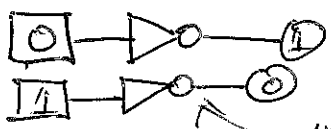
T, F (boolean)

⇒ we want boolean operations

eg NOT ~A (NOT 1 = 0)

NOT	
A	Q
1	0
0	1

-AKA an inverter — we'll see demos of this



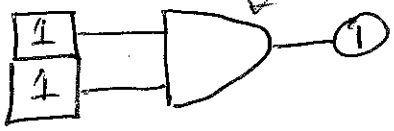
Not Gate

Truth table

AND		Q
0	0	0
0	1	0
1	0	0
1	1	1

A ^ B

⇒

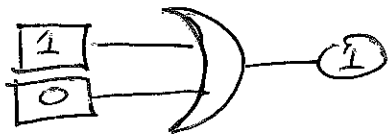


and gate

OR		Q
0	0	0
0	1	1
1	0	1
1	1	1

A ∨ B

⇒

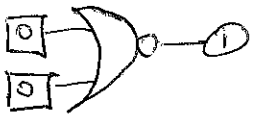


Let's combine

Not OR		
0	0	1
0	1	0
1	0	0
1	1	0

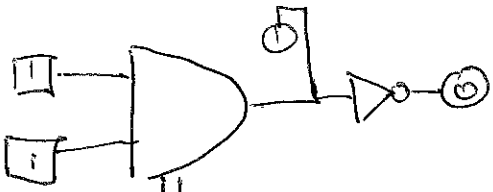
= NOR gate

⇒



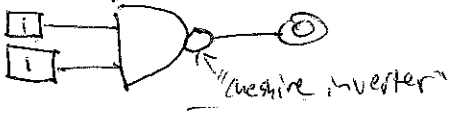
NOT AND		
0	0	1
0	1	1
1	0	1
1	1	0

⇒



= NAND gate

⇓ =



Is 0 negative charge or positive

Post-Class Notes

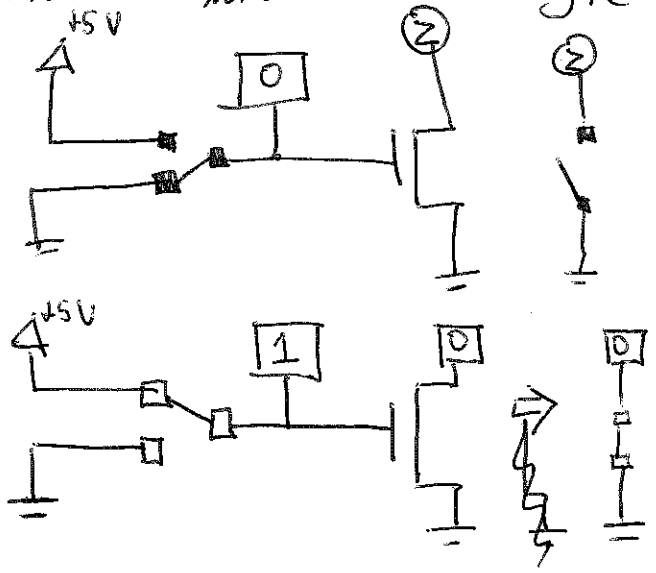
Diodes:

Transistors:

Physically how does this shit work?... (? V_{DS}, V_{GS})

n-type MOS transistor:

- closes the "switch" when the gate has a sufficiently positive voltage.

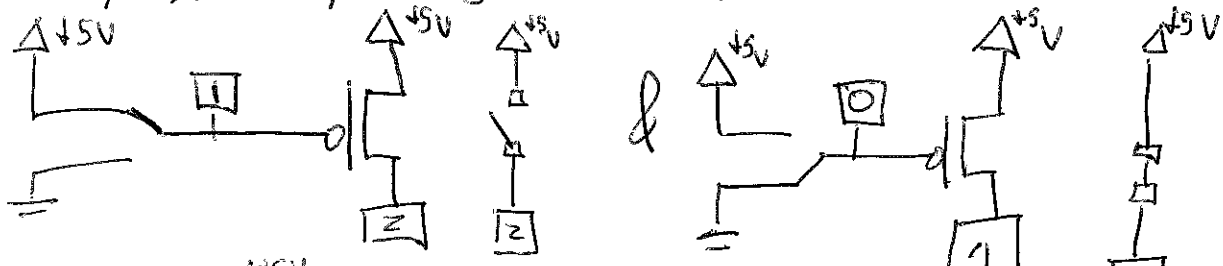


- n-type connected to ground to begin with
- Adding positive to the n-trans. ⇨ the 0 from the ground can go through.

⇨ just one of many families & types of transistors.

P-type MOS transistor:

- closes w/ sufficiently negative voltage



- connected to +5V to begin with ⇨ adding the 0 (neg) to the transistor ⇨ the 1 can go through.

CS51 4/1 Class Notes

of inputs to a gate = the fan-in of a gate

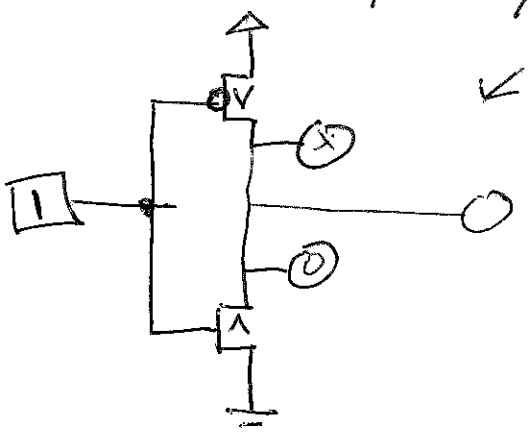
In Logisim XORs, it's 1 iff it's exactly 1.
- sometimes XOR gates do very different things

"Avoid the Dark Corners"

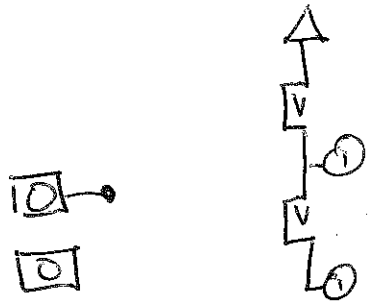
<u>Building Blocks:</u>
- n-type (needs 1 to open)
- p-type (needs 0 to open)
- pull-up resistor — open or zero → one or zero

- CMOS (complimentary MOS transistor)

NOT transistor



It's ok to connect 0 to an X
Not ok to connect 1 to an X

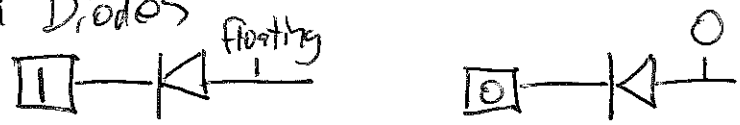


NOR

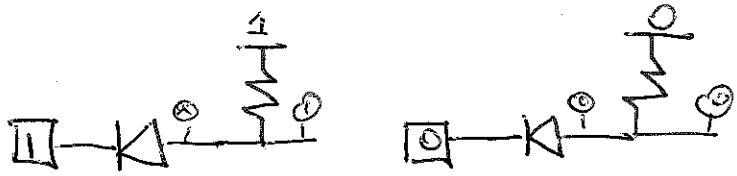
A	B	Q
0	0	1
0	1	0
1	0	0
1	1	0

use that CMOS set up.

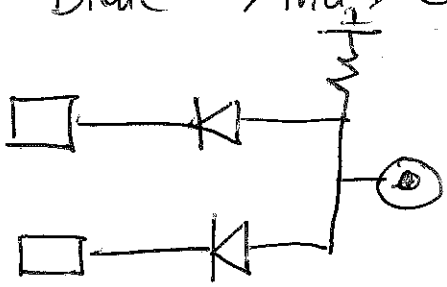
Diodes



Pull-up Resistor ⇒



⇒ Diode And Gate



Demorgan's Law:

Not (A and B) ⇒ (not A) or (not B)

Class Notes 4/2

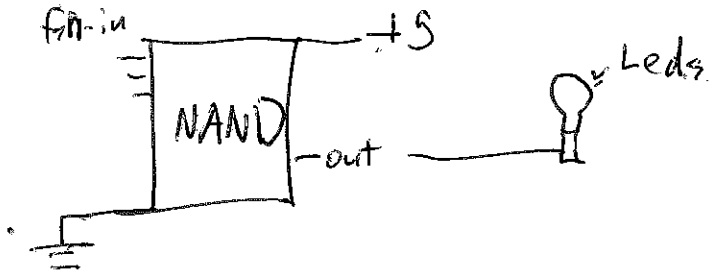
- In the real world, things take time
  - what if a not gate takes a little while?
  - This can mess things up big time.

- eg Demorgan's law:

- In terms of power consumption, zeros cost a lot more in positive logic. So you often flip to negative logic (active 0 or active low),
  you get a cheshire inverter in decater tools
  Traditionally
  1 = simple positive voltage (low current)
  0 = powerful (high current) ground.

You end up seeing "active low" inputs / outputs

To build real circuits from discrete logic IC's,
1. look up the chip layout,
2. hook up power & ground.
3. play



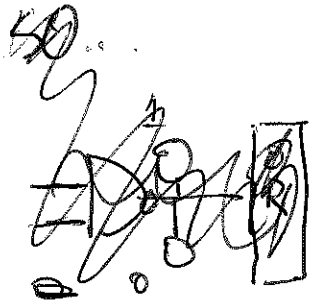
Pros:
- expensive, fragile ⇒ wire wrapper (dope as fucks & cheaper & durable, but physics ⇒ not fast & dies eventually)

Next level = print circuit board.
wires = printed ink that conduct.

Pre: Discrete transistors ⇒ Integrated circuits ⇒ bread boards ⇒ wire wrapped ⇒ print circuit board.

Later you can even do a FPGA: "field-programmable gate array" or ASIC "Application-specific IC" (see CS56) or ENGS 31 ↑ upload your logic to the IC.

To be real, for complex processors:
don't do it by hand!
Program it in Verilog or VHDL, and it will get compiled into a circuit.



When something goes wrong, read the fine print.
eg. the NAND overdrive.

Slashdot.org USB stick of death
You can use power analysis to wreck security.
- even measuring electromagnetism w/ antenna.

4/6/2015 Class notes: Binary

Base 10: "$d_3 d_2 d_1 d_0$" $= d_3 \cdot 10^3 + d_2 \cdot 10^2 + d_1 \cdot 10^1 + d_0 \cdot 10^0$

Binary = Base 2: "$d_3 d_2 d_1 d_0$" $= d_3 \cdot 10^3 + d_2 \cdot 10^2 + d_1 \cdot 10^1 + d_0 \cdot 10^0$

eg binary 1111 $= 1 \cdot 2^3 + 2^2 + 2^1 + 1 \cdot 2^0 = 8 + 4 + 2 + 1 = 15$

Formula $\sum_{i=0}^{n} d_i \cdot 2^i$

Binary coded decimal (BCD):

$\boxed{1}\boxed{5}$ (break into 4 digit bin. decis)

~~0010 1010~~ 0001 0101

a nibble

Usually we talk bout bin. #'s by breaking them into 4-bit chunks & think them into 4 bit chunks & think about each as a hex digit.

"$d_3 d_2 d_1 d_0$" $= d_3 \cdot 16^3 + d_2 \cdot 16^2 + d_1 \cdot 16^1 + d_0 \cdot 16^0$

15 = F

2's complement for signed integers

- use most significant bit as a sign

$\Rightarrow$ 0001 = 1 ~~1001 = 1~~

- es. 3 bit #'s

signed: $\sum_{i=0}^{n} d_i \cdot 2^i$

$-d_{n-1} B^{n-1} + \sum_{i=0}^{n-2}$

	U	S
000	0	0
001	1	1
010	2	2
011	3	3
100	4	-4
101	5	-3
110	6	-2
111	7	-1

U = 0 → 7

S = -4 → 3

$\rightarrow -1 \cdot 2^2 + \sum$

$-4 + 2 + 1 =$

Modular arithmetic

- As you add with signed numbers, it might wrap.
  If you add too many large bin. values, you can get overflow
  ⇒ need ~~too~~ more many bits than you have

Binary (we don't write up)

0x~~F~~FFF = 0111 1111 1111 1111

1's complement has also been used. Dig floating point representation.

Tools:

- Multibit pin
- splitter (takes multiple inputs too)
- Hex Value render

Building Comparators.

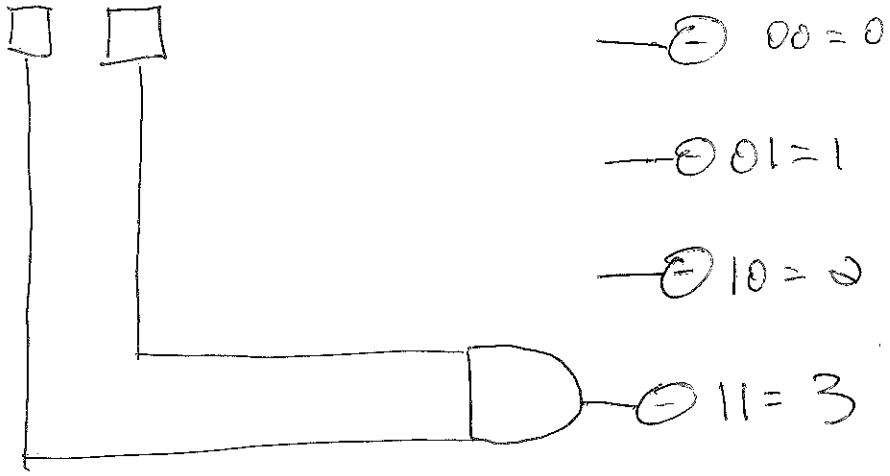
- use Ands
- XNOR each bit together!

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

4/6/2015 p.2

Functionally Complete:
1. And & Not
2. NAND
3. NOR

How to do things w/ Binary numbers:
Decoder: turns on one of these output lines depending on 2 bits



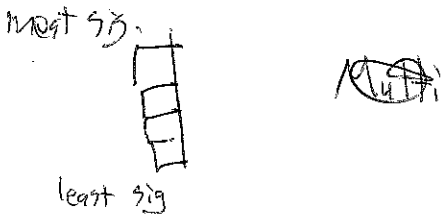
Generalizing?

CS91 4/8/2015

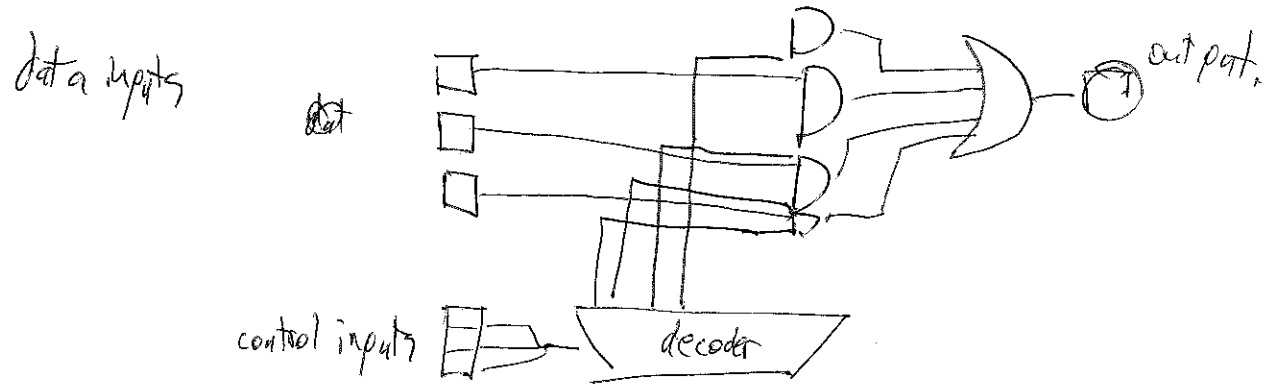
Decoder: take inputs, get 2^n outputs
- Seven-segment LED Display

- Input BCD, output turning on the right segments

inputs:	S_1	. . . . .	S_7
000	0		
001	0	0 . . . 1	1 . . . 0
. . .			
. . .			
111	0 . . . 1	0 . . . 1	1 . . . 0



Multiplexers:
we want some control inputs to decide what data input goes through.



Demultiplexers:
Symetric operation?
Multiple outputs to 1 input

Adders:

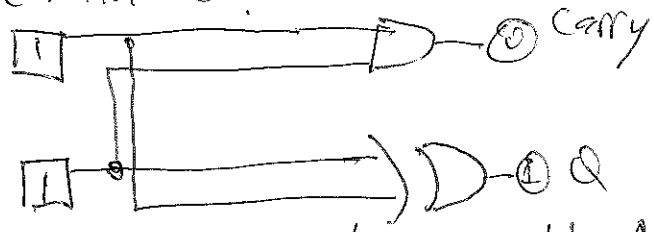
$$\begin{array}{r} 4 \\ +\ 6 \\ \hline 10 \end{array}$$

Truth Table

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1
		↑ XOR	↑ AND

A XOR B = S
A AND B = C

This a half-adder circuit. But what if I want more than 2?



But you may need an n-adder that does 3 bits! (for multiple bits)

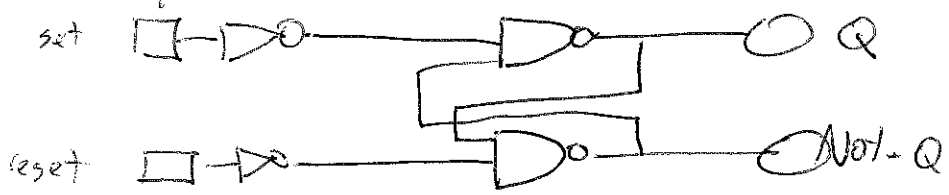
A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
1	0	0	1	0
0	1	1	0	1
1	1	0	0	1
1	0	1	0	1
1	1	1	1	1

Can make with 4 decoder
&

Class Notes 4/15/2015

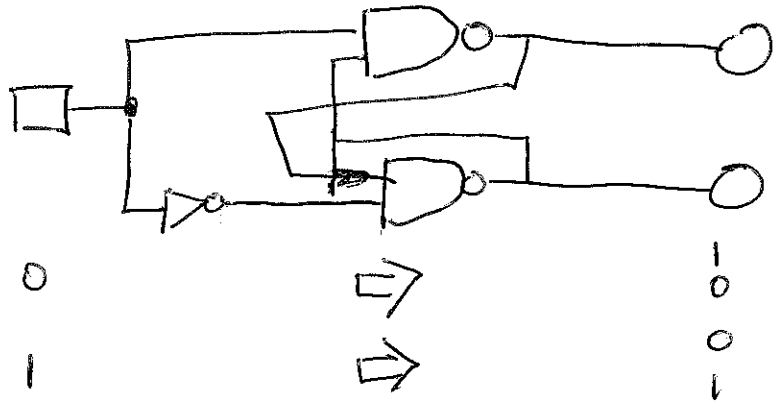
We've been working so far w/ DAGs.
- combinatorial logic. Outputs depend on current inputs
- cyclic circuits ⇒ functions that aren't well-defined

Q = bit
set = put to 1
reset = put to 0

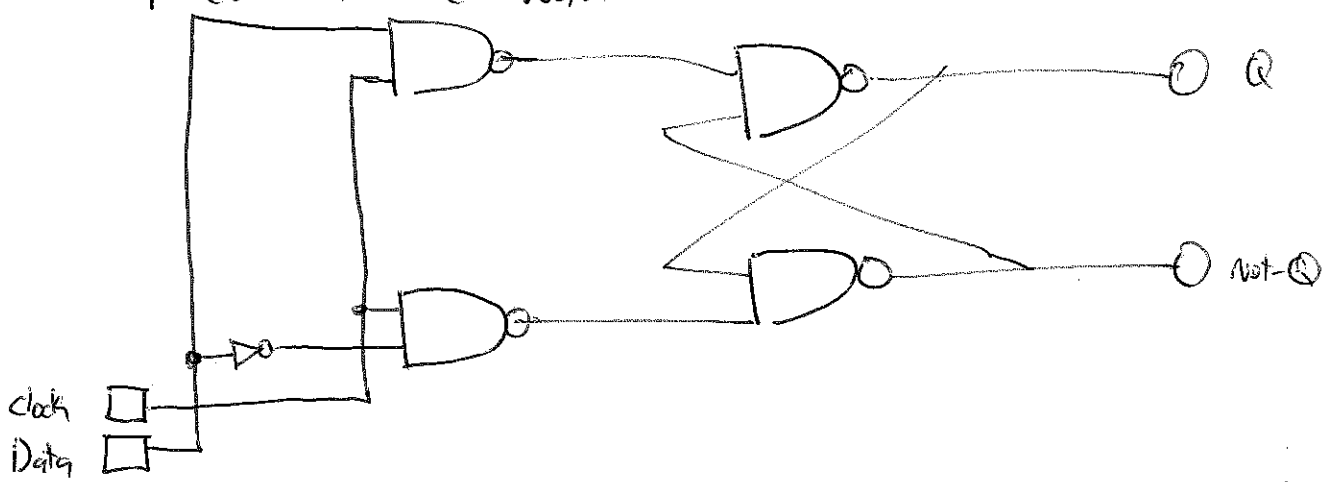


This is a "SR Latch"
- trying to use 1 switch to controll an SR-latch
⇒ oscillation

Latch attempt:



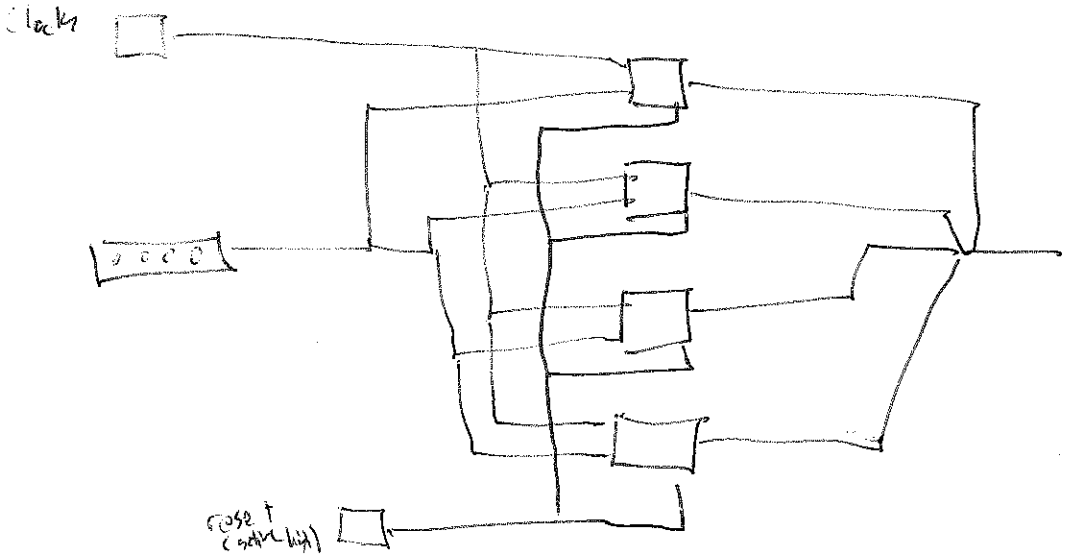
Here's what we want



When clock = 0, the two nand gates
⇒ ignore input (feed in ones)
If clock = 1,
Q stores the data.
- we just built static memory bit!

Logisim calls this the D flip-flop.
- but configured w/ a level-triggered clock.

How about a 4 bit rememberer?



4/10/2015

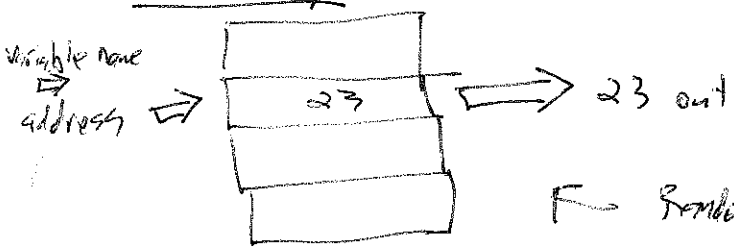
Class Notes

latch vs. flip flop

latch → level-triggered → while the clock is 1, you're sampling

flip flop → rising edge → only samples at the moment you switch clock from 0 to 1

Memory:



you can store & get things back...

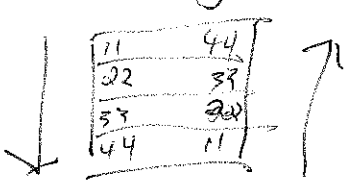
← Random access Memory

computers usually have a "natural word size"
- was 32-bits, now it's 64
you can read or write to slots in any order

In practice, you have 8-bit slots, so you give such a "word" @ 4 slots

what order do you store the bytes?

0x11223344 (which end goes at the bottom)
(little endian)

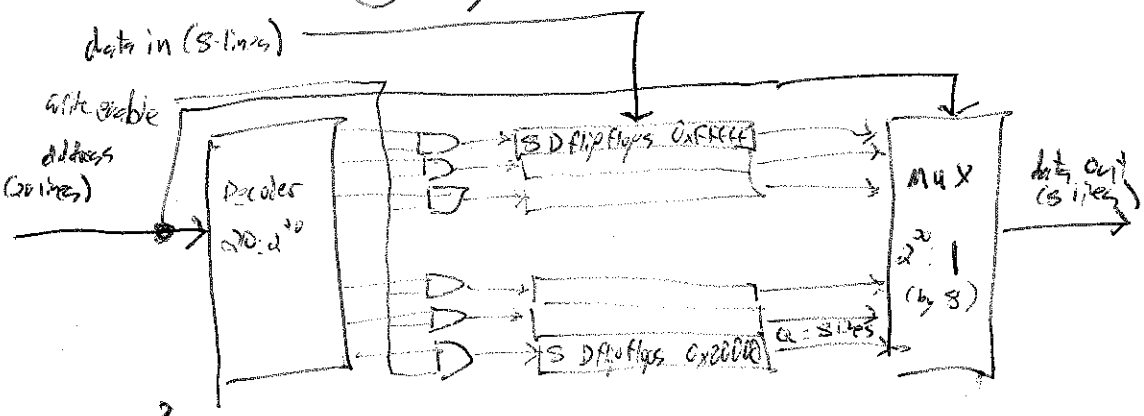


or

Also, some alignment requirements (e.g. 4-bit word better start at a multiple of 4).

little ambien, we're going to ignore alignment issues

⇨ a Megabyte of RAM



This is hard & expensive.

⇨ Dynamic Ram

using capacitors to build trickier but "sparser"

Dynamic RAM (DRAM) ⇨ greater densities

SO Reality!

DRAM: w/ capacitors (dense, higher power consumption)

SRAM: w/ flip flops (less dense, but lower power).

but reading it destroys it. Must write it again when you read it.

4/20 Class Notes

Traffic light = FSM approach
we have finite state machine approach
w/ combinatorial logic & memory to store state
controls output & determines next state

- expresses a "process" as an FSM & build logic circuit to make it happen
  "state number" is stored in edge-trigger register
  combo logic reads state and spits output / next state
  (to be clocked)

Micro code
we need to think about the "instructions"

At step 0:
turn N Red off & turn N. green On

Step 1:
Turn N. Green off & turn North yellow On

Step 2:
Turn N. Yellow off & turn N. Red on

Step 3: Turn E. red off & E. green On
4: E green off & E. yellow on
5: E yellow off & E. red on
go to 0.

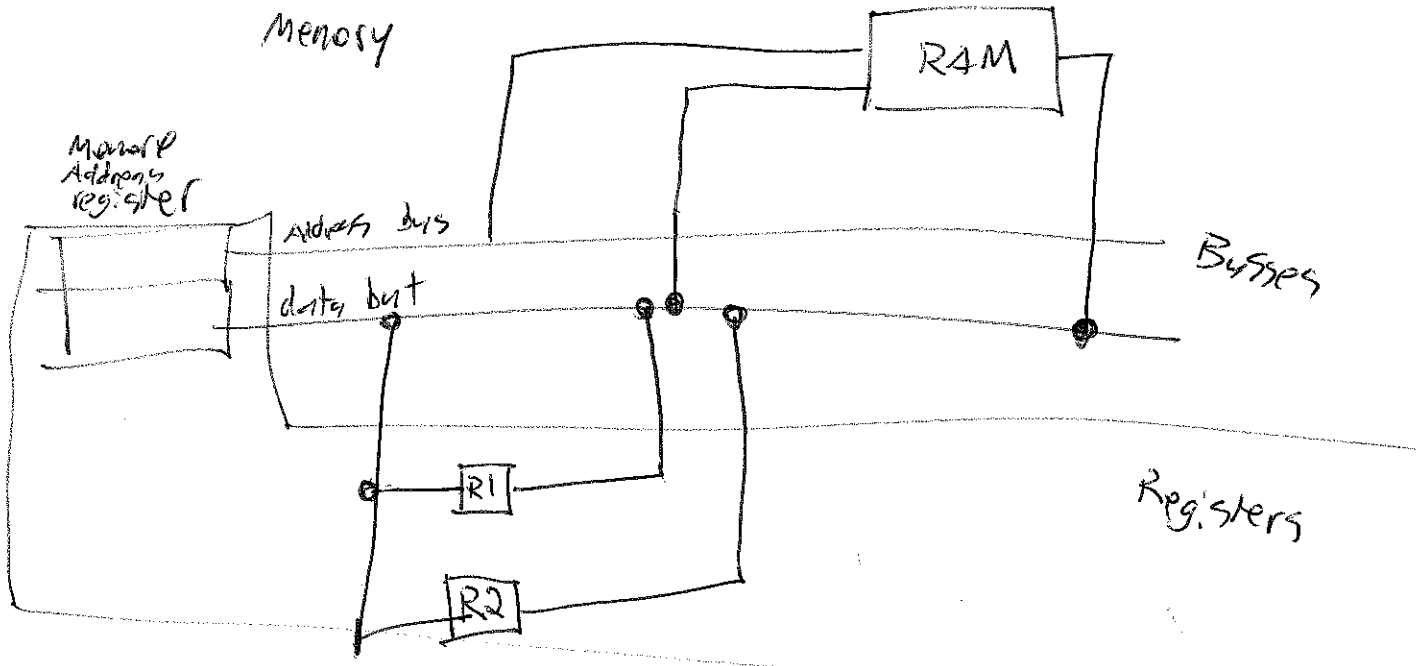
13 instructions...
can view as just 13 bits

what if instead, we had a magic box
001 ⇒ the 13 bits corresponding to step 1...
010 ⇒ the 13 bits corresponding to step 2...

Since we only read it, we can abstract this to ROM

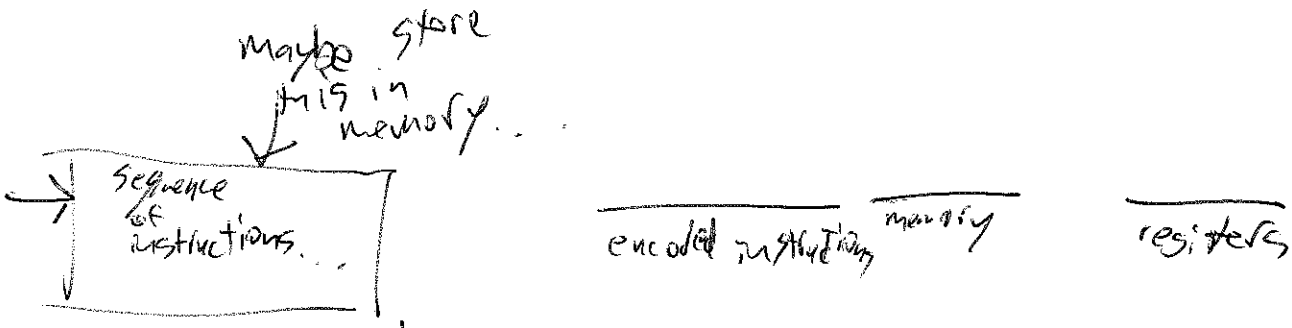
<u>Abstracting Ram:</u>

Class Notes 4/22/2015



but, we don't really manna wiggle the values.
let's do a traffic light type of thing...

state 7 w/ n.
Maybe I can express the op. w/ a binary number.
pt. 1 = we're moving from address to a ~~number~~ register
pt. 2 = address.



need some way to indicate where we're at "PC" program counter, "IP" Instruction pointer.

where does the program ~~memory~~ sequence get stored.
Von Neumann = same place as data memory
Harvard architecture = different places.
Von Neumann won w/ x86. But Harvard better for security.

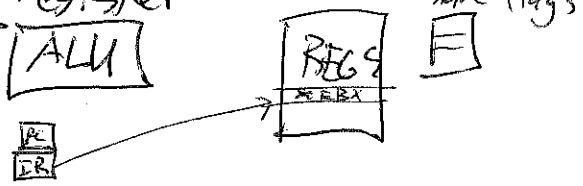
Class Notes 4/24

The ISA is what you play with to write a machine language program on a processor

IR = Instruction Register (in ebx)

~~Reg~~ Addressing Modes:

1) Register



In x86, rrmovl rA, rB ⇒ move rA into rB

In the instruction register, the register containing the operand is referenced.

PC = IP (instruction pointer)

eg rrmovl rA, rB

2. ~~Immediate addressing~~

The value directly follows the instruction

irmovl V, rB

3	0	F	rB	V

3) Absolute

- In instruction, we get an immediate value we treat as an address to a value (in Data memory) we care about

rmmovl rA, D

4	0	rA	F	D

4-byte address

~~DE~~
~~DE~~
~~BE~~
~~AD~~
~~DE~~

4. Indirect:

we point to a register, use that as its address

rmmovl rA, (rB) | 4 | 0 | rA | rB | 00 00 00 00 |

means Memory ~~indirect~~ location in

move value in rA to memory location stored in rB

5. Base + Displacement

Address in memory = Base pointer + offset
(eg. looping through lists)

rmmove rA, D(rB)
⇒ add D to value in rB

egs.

reg. addressing
rmovl %ecx, %edx 20 12

indirect addressing
irmovl target2, %ebx
rmmovl %ecx, (%ebx) ⇒ 401 30000000

little endian

absolute
rmmovl %ecx, 0x50 ⇒ 4 0 1 F ~~00 00 00 50~~ 50 00 00 00
rmmovl %ecx, 4(%ebx) ⇒ 401304 00 00 00

Class Notes 4/29/2019

Assembly = 1-step above binary
Textual representation w/ each line as a binary instruction.

Directives:
- .pos = where to start (always have one first)
- .align 16 = start at multiple of 16
- .long prefixes a 4-byte word
- .word = 2-bytes
- .byte

Immediates start w/ $

<u>Basic Instructions:</u>
nop = do nothing
halt = stop processor
4 move operations
jmp

biggest positive # = 0x7fffffff
biggest neg # = $0x 80 00 00 00

Conditional codes ⇒ conditional jumps & moves
under what conditions do you do the jump/move?
jmp $\boxed{7|0}$ = unconditional
jle, jne, jge, jl, jg, je
~~je when~~
C-B ⇒ ZF ⇒ they're equal
C-B ⇒ SF

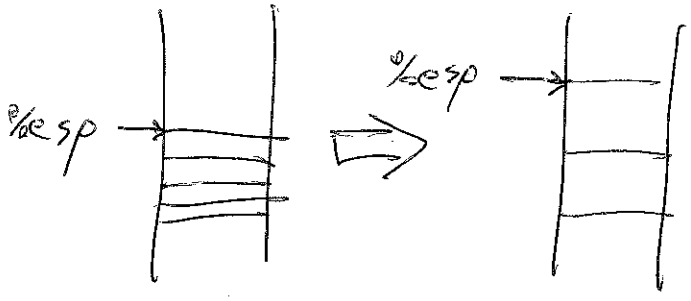
⇒ C<B ⇒ SF XOR OF
C = B ⇒ ZF
C > B ⇒ ¬(SF ∨ OF ∨ ZF) ∨ (SF ∧ OF)
(see list on cheat sheet)

Real architectures have a "cmp" instruction that pretends to do the op. but doesn't destroy

The stack! 2 instructions: push & pop
pushl rA ⇒ push value in rA onto stack
popl rB ⇒ remove val on stack ⇒
stack pointer!
Points to the last used spot.

Class Notes 4/30/2015

- The stack grows to lower addresses



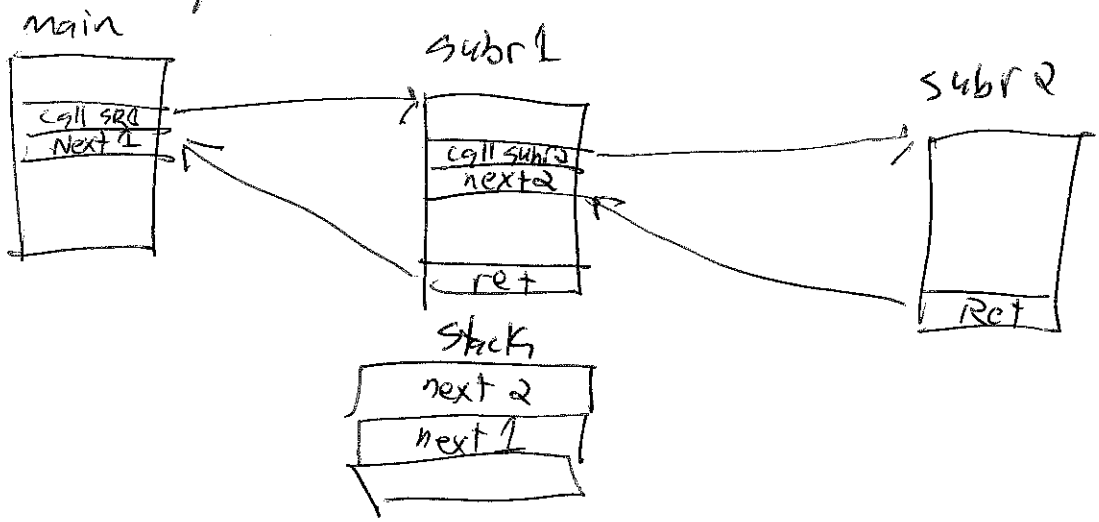
If you pop an empty stack, what happens?

<u>Subroutines</u>

For recursion purposes, need a stack

call pushes return address (next instruction

ret ~~returns~~ pops return address



%esp is always stack pointer.

Have to set it up!

.pos 0

irmovl stack, %esp

stack pointer

.pos 0x10C

dummy value

stack:

.long 0xFFFFFFFF # Top of stack

Problem, though. when you make a subroutine, shit we got overwritten w/ deadbeef
we must find a way to make our local vars not get trashed when we call a func.

Idea ~~0~~:
let's save and restore mem
push whole state onto the stack (ret. location & registers you care about!)

Idea 1: who saves the data?
Caller can do it, or callee can do it!

Some architectures have it so call & ret. do all the pushing / popping for you!