No.

Date

LOGIC GATES

Important terms in physics

positive, negative, voltage (potential difference), current, conductor, ~~insulb~~ insulator, ground, switch

Semiconductors and Doping

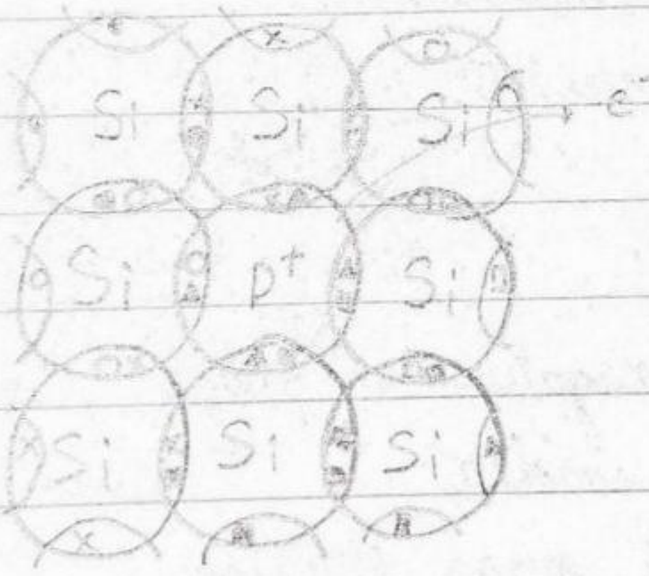
semiconductor (pure silicon) does not conduct electricity

⇒ introduce impurities to increase number of free charge carriers

n-type

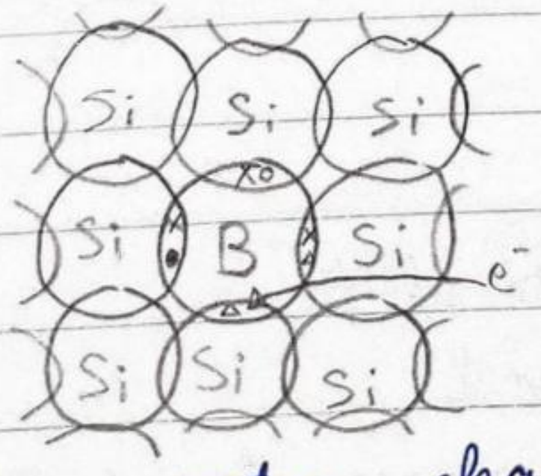
adding phosphorus / arsenic ~~that~~ which has 5 outermost shell electrons

so the ~~th~~ extra electron is free to move



p-type

adding boron which has 3 electrons in outermost shell so an electron is missing,



Both are neutral (no extra charge) ↑ all covalent bonds

Putting p and n together, free electrons in n fill up holes in p

⇒ current called diffusion current ↓ p-n Junction

Thin layer called depletion layer formed at boundaries of p-n.

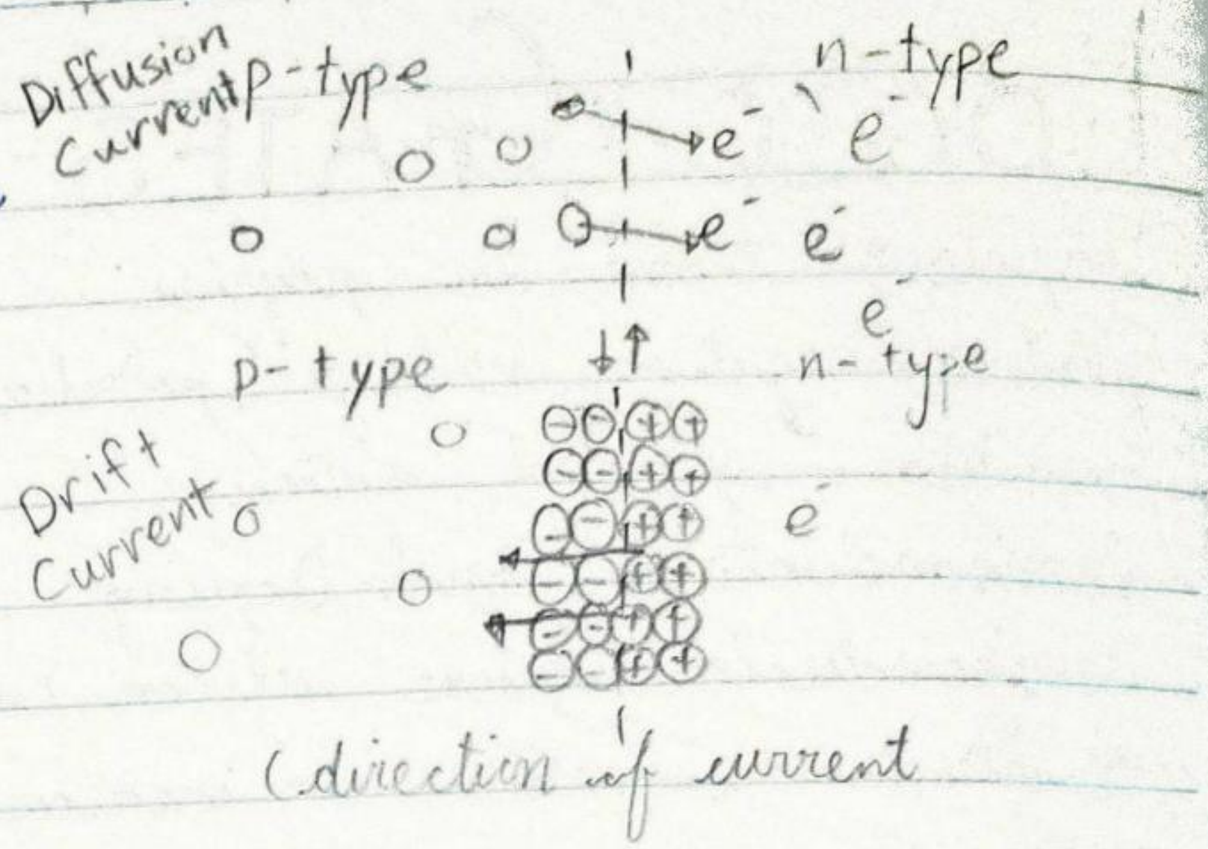
↳ electric field and causes electrons to flow back to original position

No.

Date

↳ drift current

At rest, both currents in equilibrium

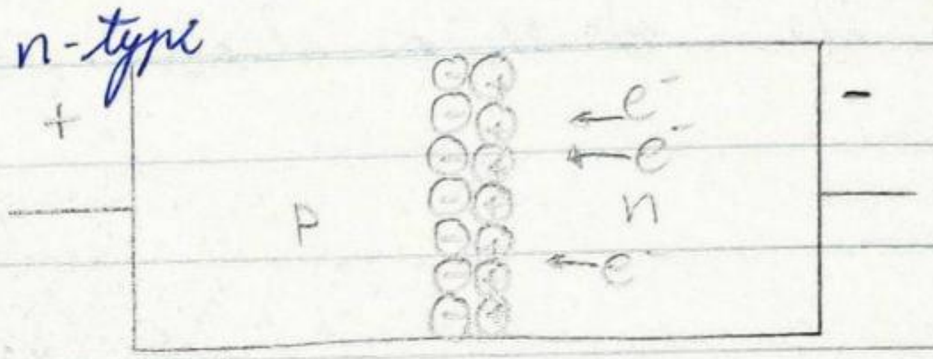


Forward Bias

positive on p-~~typ~~ type and negative on n-type

depletion layer becomes narrower

⇒ increase diffusion current

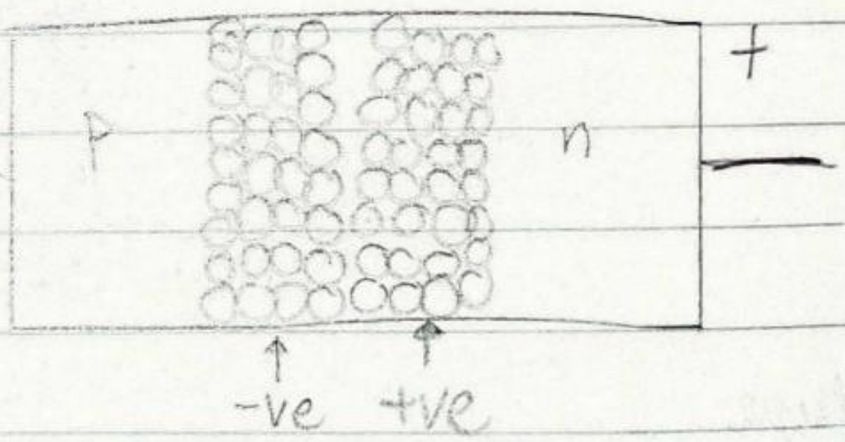


Reverse Bias

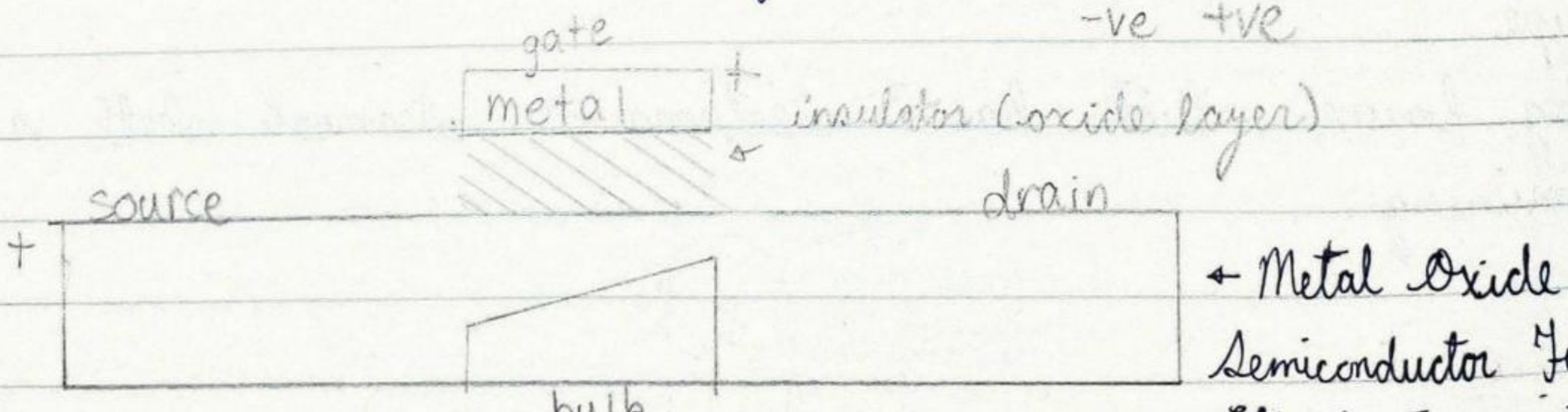
positive on n-type and negative on p-type

depletion layer becomes wider

⇒ little to no current pass through



MOSFET



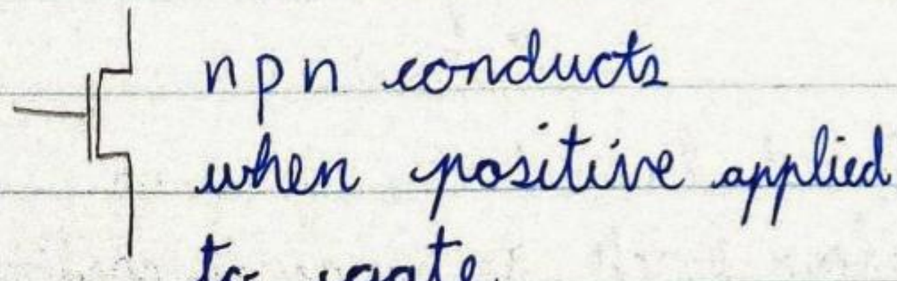
← Metal Oxide Semiconductor Field Effect Transistor

positive charge on gate attracts electrons

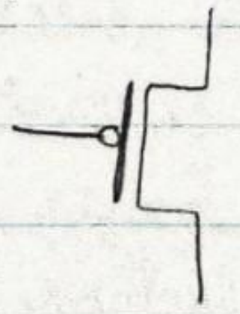
electrons form an n-type channel

current can flow from ~~s~~ source to drain

nMOS vs pMOS



npn conducts when positive applied to gate

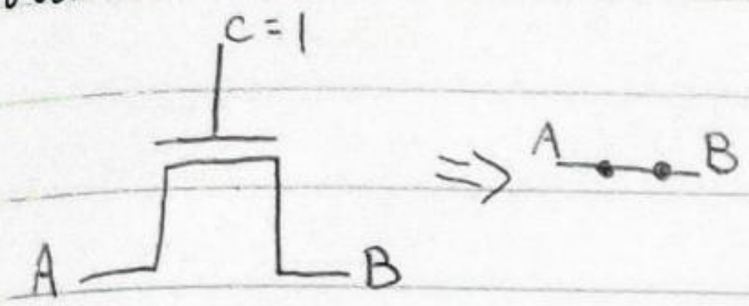


pnp is a closed switch when voltage is logic-zero

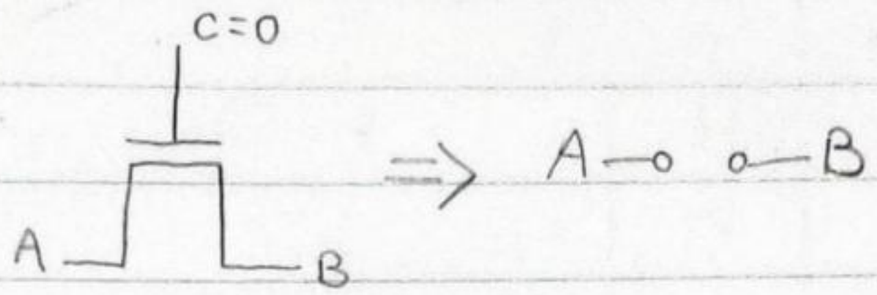
No.

Date

transistors



value in gate high
A and B are connected



value in gate low, A and B are disconnected

CMOS

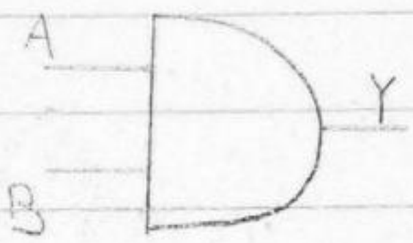
complementary MOSFET ⇒ either logic-0 or logic-1, never both, never neither

AND

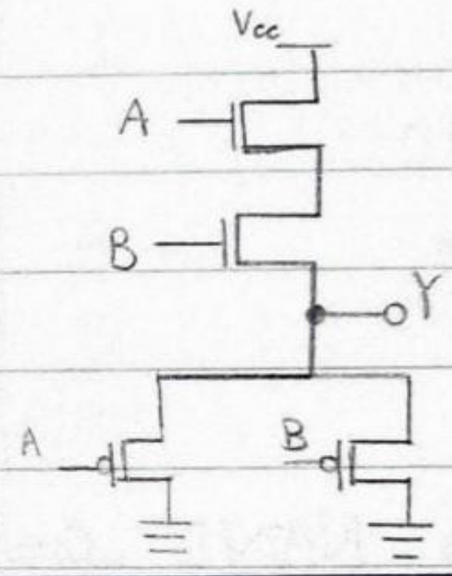
TRUTH TABLE

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

SYMBOL

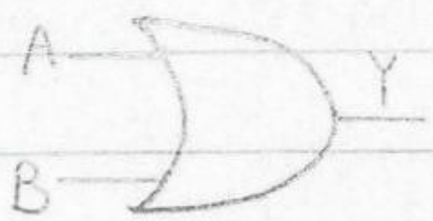


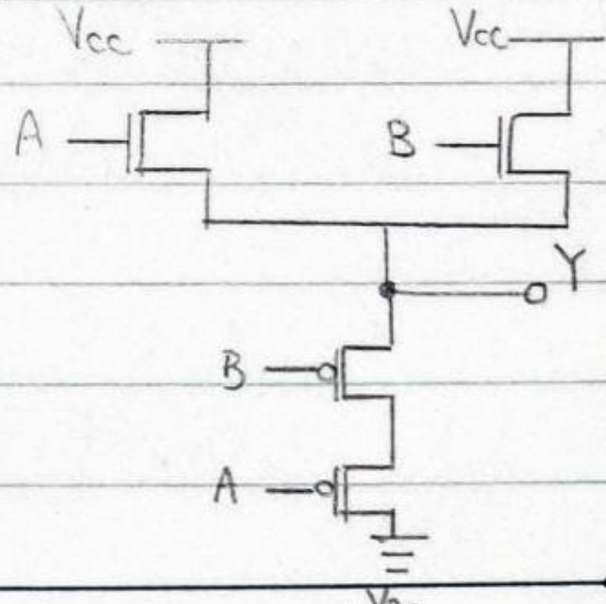
CIRCUIT



OR

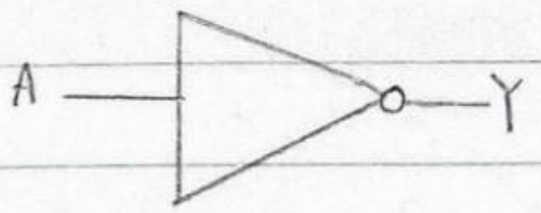
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

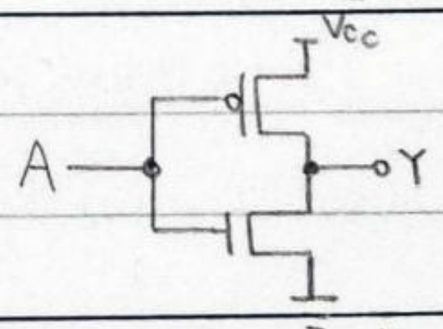




NOT

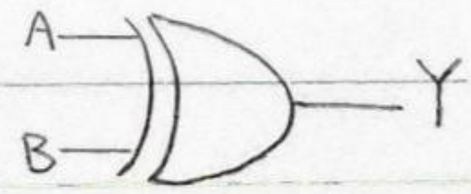
A	Y
0	1
1	0

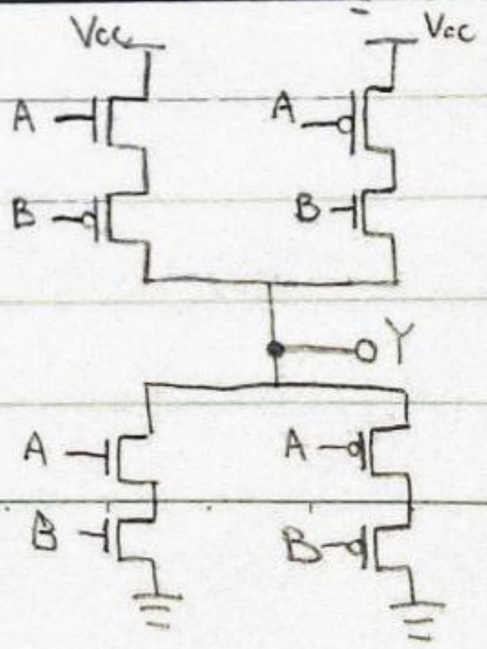




XOR

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



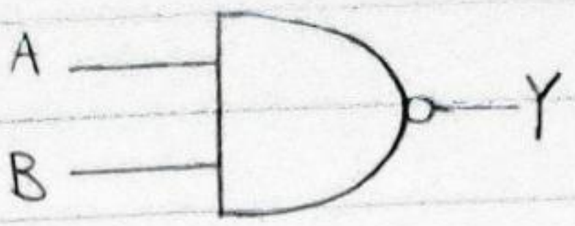


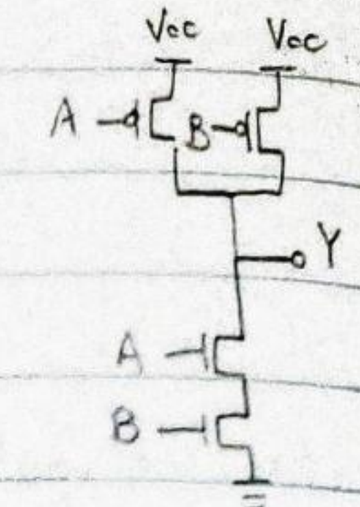
No.

Date

NAND

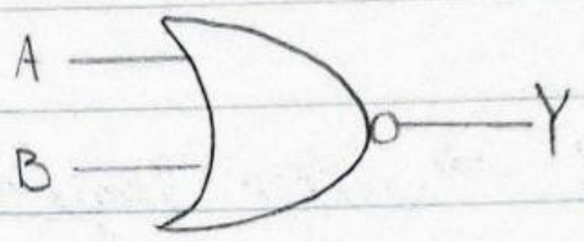
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

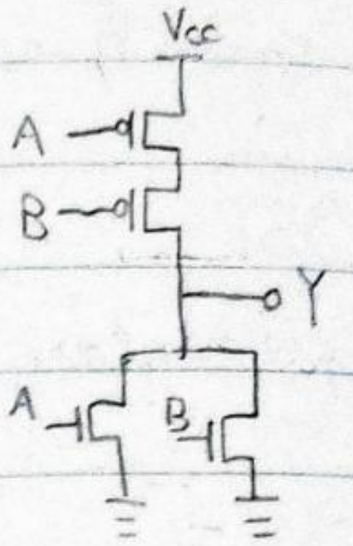




NOR

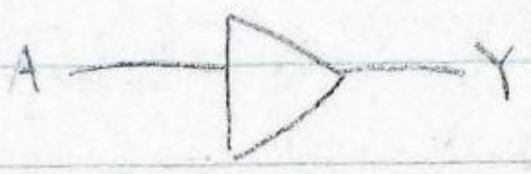
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0





Buffer

A	Y
0	0
1	1

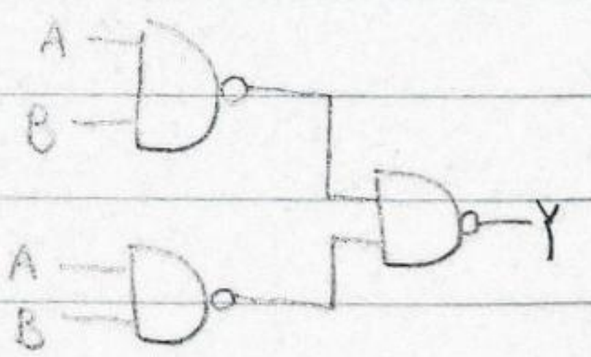


Using NAND Gate to build other gates

AND

(A NAND B) NAND (A NAND B)

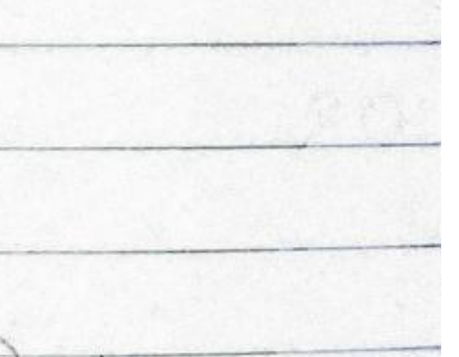
0	1	0	0	0	1	0	
0	1	1	0	0	1	1	
1	1	0	0	1	1	0	
1	0	1	1	1	0	1	



XOR

(A NAND (A NAND B)) NAND (B NAND (A NAND B))

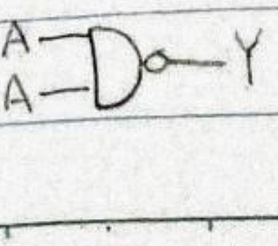
0	1	0	1	0	0	0	1	0	1	0	
0	1	0	1	1	1	1	0	0	1	1	
1	0	1	1	0	1	0	1	1	1	0	
1	1	1	0	1	0	1	1	1	0	1	



NOT

A NAND A

0	1	0
1	0	1

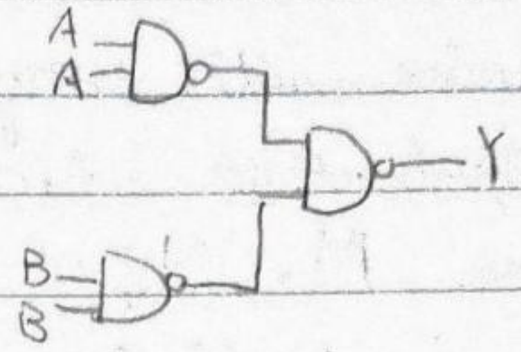


No.

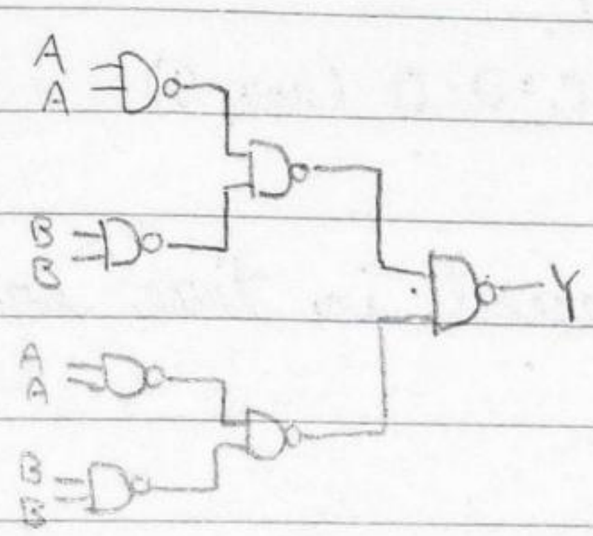
Date

OR (A NAND A) NAND (B NAND B)

0	1	0	0	0	1	0	
0	1	0	1	1	0	1	
1	0	1	1	0	1	0	
1	0	1	1	1	0	1	



NOR

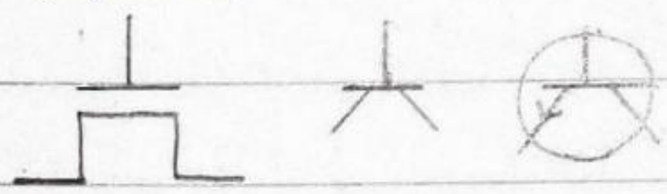


14/01/2020

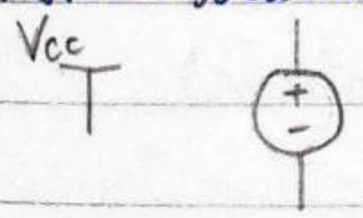
BUILDING CIRCUITS

NOTATIONS

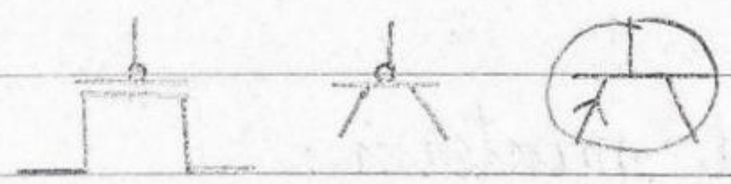
Transistors



NPN transistor



Vcc; high



PNP transistor

ground; low

~~Bob~~ Boolean Logic

AND: A·B·C / ABC / A*B*C ≈ A∧B∧C

OR: A+B+C ≈ A∨B∨C

NOT: !A / ~A / $\bar{A}$ / A` / ¬A

XOR: A⊕B

CREATING COMPLEX LOGIC

1. Create truth tables
2. Express as Boolean expression
3. Convert to gates

No.

Date

MAXTERMS

an OR expression with every input present in true (A) or complemented ($\bar{A}$) form
rows of ~~output that~~ which output is 0
expressed as $M_0, M_1, \cdots M_x$
e.g. given 4 inputs (A, B, C, D), $A+B+\bar{C}+D$
from $M_0(A+B+C+D)$ to $M_{15}(\bar{A}+\bar{B}+\bar{C}+\bar{D})$
M_0 is A or B or C or D $\Rightarrow M_0=0 \leftrightarrow A=B=C=D=0$ (row 0) and 1 elsewhere

MINTERMS

an AND expression with every input present in true or complemented form
rows of which output is 1
expressed as $m_0, m_1, \cdots m_x$
e.g. given 3 inputs (A, B, C). $A \cdot \bar{B} \cdot C$
from $m_0(\bar{A} \cdot \bar{B} \cdot \bar{C})$ to $m_7(A \cdot B \cdot C)$
m_0 is $\bar{A}$ and $\bar{B}$ and $\bar{C} \Rightarrow m_0=1 \leftrightarrow A=B=C=0$ (row 0) and 0 elsewhere

nice thing about minterm:
$m_4 \Rightarrow A \cdot \bar{B} \cdot \bar{C}$
1 0 0 ← which is 4 in binary

(maxterms can be done using negation of minterm)

SUM-OF-MINTERMS (SOM)

combined high outputs = union of these minterms
e.g. $m_2+m_6+m_7+m_{10}$ tells ~~us~~ us where the 1s are
AKA Sum-of-Products

PRODUCT-OF-MAXTERMS (POM)

combined low outputs ~~are~~ = intersection of these maxterms
e.g. $M_3 \cdot M_5 \cdot M_7 \cdot M_{10} \cdot M_{14}$ tells us where the 0s are
AKA Product-of-Sums

No.

Date

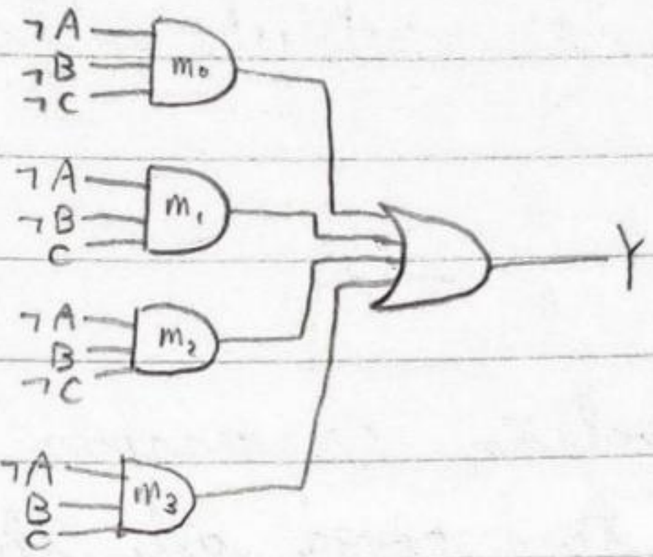
Converting SOM to gates

find which rows the minterms correspond to in truth table

& AND for each row and OR the outputs

$m_0 + m_1 + m_2 + m_3$

$= \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + \bar{A}B\bar{C} + \bar{A}BC \Rightarrow$



BOOLEAN ALGEBRA

Axioms

$0 \cdot 0 = 0$ $\quad 0 \cdot 1 = 1 \cdot 0 = 0$

$1 \cdot 1 = 1$ $\quad$ if $x = 1$, $\bar{x} = 0$

Furthermore

$x \cdot 0 = 0$ $\quad x + 1 = ~~\not\Rightarrow~~ 1$

* $x \cdot 1 = x$ $\quad$ * $x + 0 = x$

$x \cdot x = x$ $\quad x + x = x$

$x \cdot \bar{x} = 0$ $\quad x + \bar{x} = 1$

$\bar{\bar{x}} = x$

Absorption Law

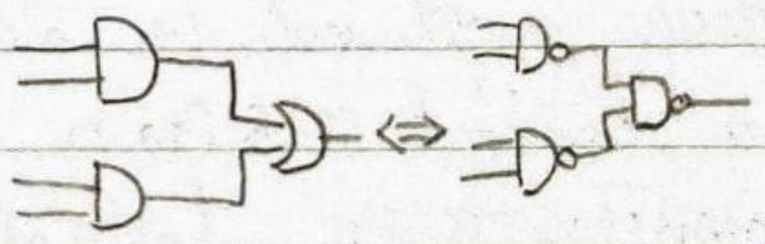
$x \cdot (x + y) = x$ $\quad x + (x \cdot y) = x$

De Morgan's Law

$\bar{x} \cdot \bar{y} = \overline{x + y}$ $\quad \bar{x} + \bar{y} = \overline{x \cdot y}$

Converting to NAND gates

Because of De Morgan's Law, Sum-Of-Products circuit can be converted into equivalent NAND gates



GATE COST

G → cost of gate * # of gates

GN → ~~cost~~ G + cost of not gate + # of not gates

Commutative Law

$x \cdot y = y \cdot x$ $\quad x + y = y + x$

Associative Law

$x \cdot (y \cdot z) = (x \cdot y) \cdot z$ $\quad x + (y + z) = (x + y) + z$

~~$x + (y + z)$~~ Distributive

$x \cdot (y + z) = x \cdot y + x \cdot z$ $\quad x + (y \cdot z) = (x + y) \cdot (x + z)$

Simplification Law

$x + (\bar{x} \cdot y) = x + y$ $\quad x \cdot (\bar{x} + y) = x \cdot y$

Consensus Law extra

$x \cdot y + \bar{x} \cdot z + y \cdot z = ~~x + y~~ \; x \cdot y + \bar{x} \cdot z$

No.

Date . .

KARNAUGH MAPS / K-MAPS

2D grid of ~~a~~ minterms where adjacent minterm locations in grid differ by a single literal

Values on grid are output ~~a~~ for that minterm

	$\bar{B}\cdot\bar{C}$	$\bar{B}C$	BC	$B\bar{C}$
$\bar{A}$	0	0	1	0
A	1	0	1	1

Simplifying boolean expression using k-maps

draw boxes to cover all the 1s (as few as possible)

- must be rectangles, ▭ or ▯ or □
- size ~~be~~ must be powers of 2
- may overlap
- may wrap across edges of map

within the box, create boolean expression using ~~inputs~~ common inputs

e.g. ▯ = $B\cdot C$ ▯ = $A\cdot\bar{C}$ $\Rightarrow$ $Y = B\cdot C + A\cdot\bar{C}$

MAX-TERM CASE

	$B+C$	$B+\bar{C}$	$\bar{B}+\bar{C}$	$\bar{B}+C$
A	M_0	M_1	M_3	M_2
$\bar{A}$	M_4	M_5	M_7	M_6

group zeros instead ~~or~~ of ones

BUILDING DEVICES

21/01/2020

k-maps might not be able to simplify expressions sometimes

e.g. 3-input XOR Gates

COMBINATIONAL CIRCUITS

- outputs strictly rely on inputs

No.

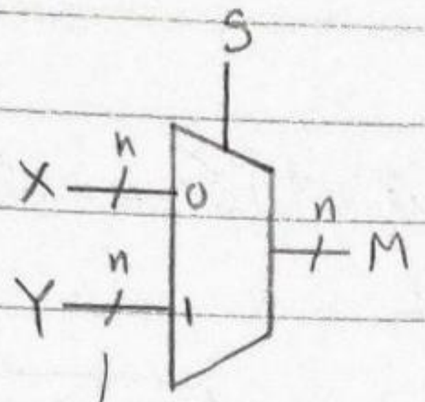
Date

MULTIPLEXERS (MUX)

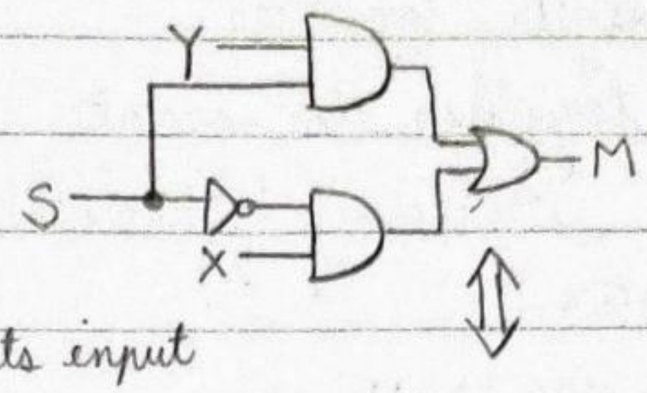
S - select input

X and Y - data inputs

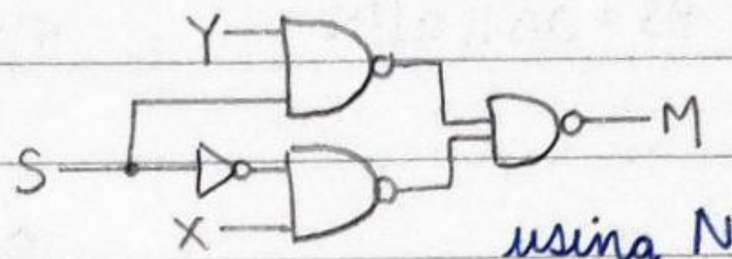
Output = X if S is 0
Y if S is 1



$M = Y \cdot S + X \cdot \bar{S}$



	$\bar{Y}\cdot\bar{S}$	$\bar{Y}\cdot S$	$Y\cdot S$	$Y\cdot\bar{S}$
$\bar{X}$	0	0	1	0
X	1	0	1	1



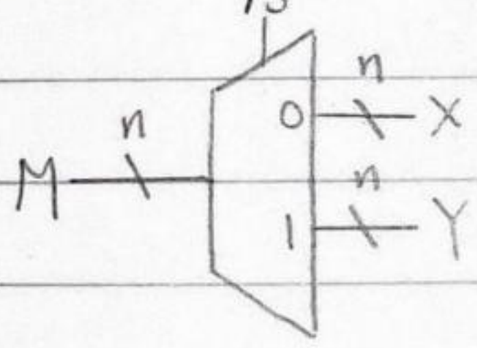
using NAND

DECODERS

binary translators

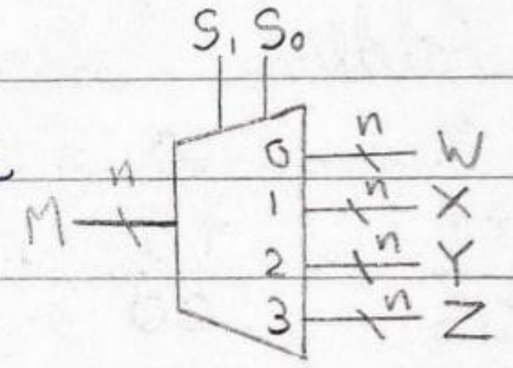
provides a mapping from a binary number to another encoding

Demultiplexers



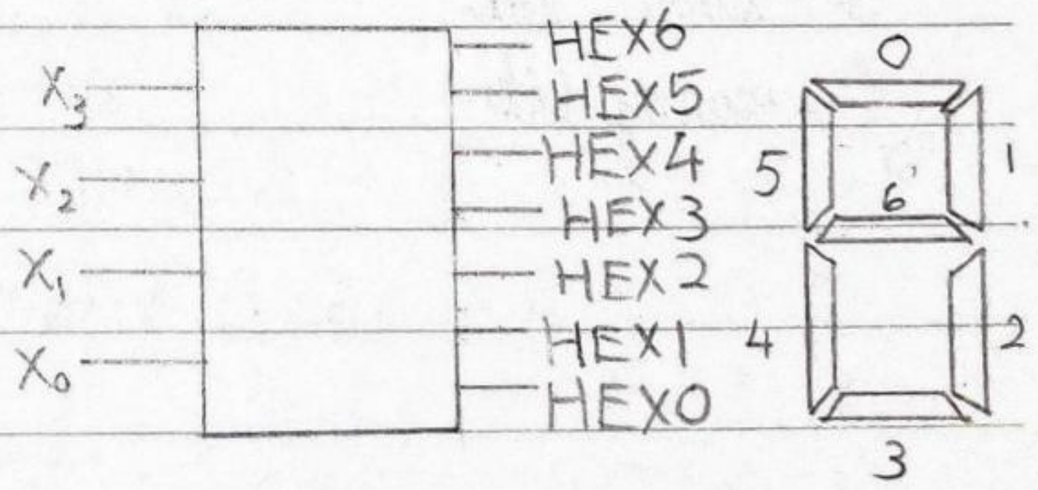
the selected output would be M,

like mux, but reverse



7-segment decoder

- translate 4-digit binary number to seven segments of a digital display
- create boolean logic for each output segment ~~to~~



* Active-low - turns on when wire / output is low

e.g. HEX0

	$\bar{X}_1\cdot\bar{X}_0$	$\bar{X}_1\cdot X_0$	$X_1\cdot X_0$	$X_1\cdot\bar{X}_0$
$\bar{X}_3\cdot\bar{X}_2$	0	1	0	0
$\bar{X}_3\cdot X_2$	1	0	0	0
$X_3\cdot X_2$	x	x	x	x
$X_3\cdot\bar{X}_2$	0	0	x	x

x = don't care (value can be 1 or 0 depending if what helps to simplify the circuit)

$\Rightarrow HEX0 = \bar{X}_3\cdot\bar{X}_2\cdot\bar{X}_1\cdot X_0 + X_2\cdot\bar{X}_1\cdot\bar{X}_0$

No.

Date . .

ADDERS

- adds 2 digits together
- combined together to create iterative combinational circuits

Unsigned Binary Addition

e.g. 27+53

27 = 00011011

53 = 00110101

```
  1 1 1 1
  0 0 0 1 1 0 1 1
+ 0 0 1 1 0 1 0 1
-----------------
  0 1 0 1 0 0 0 0
```

side: 1+1=10 ↑ moves to next digit

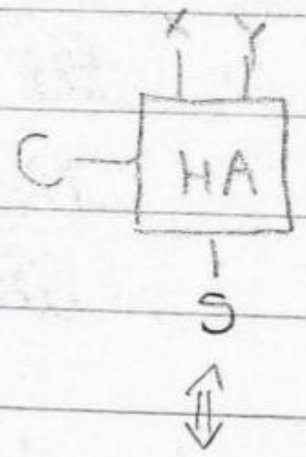
$0\times2^7 + 1\times2^6 + 0\times2^5 + 1\times2^4 + 0\times2^3 + 0\times2^2 + 0\times2^1 + 0\times2^0$

$= 64 + 16 = 80$

* problem: with 8 bits, unsigned numbers max 255
~~adding~~ a sum that exceeds 255 ~~with~~ will cause an overflow

Half Adders

X	0	0	1	1
+Y	+0	+1	+0	+1
CS	00	01	01	10



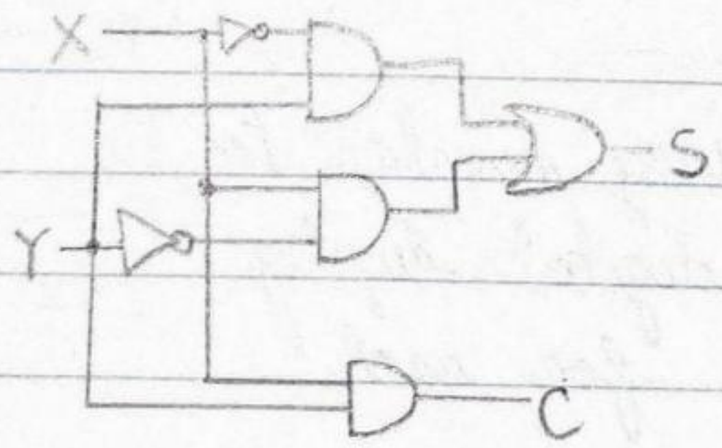
S - sum bit

C - carry bit

X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$\rightarrow S = X \oplus Y$

$C = X \cdot Y$



Full Adders

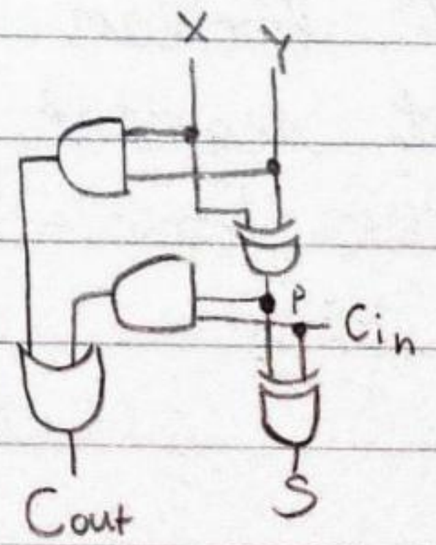
half adder with an extra carry-in bit

when z = 0, see above; when z = 1

X	1	1	1	1
+Y	+0	+1	+0	+1
+Z	+1	+1	+1	+1
CS	01	10	10	11

$\therefore C = X \cdot Y + X \cdot Z + Y \cdot Z \Leftrightarrow C = X \cdot Y + (X \oplus Y) \cdot Z$

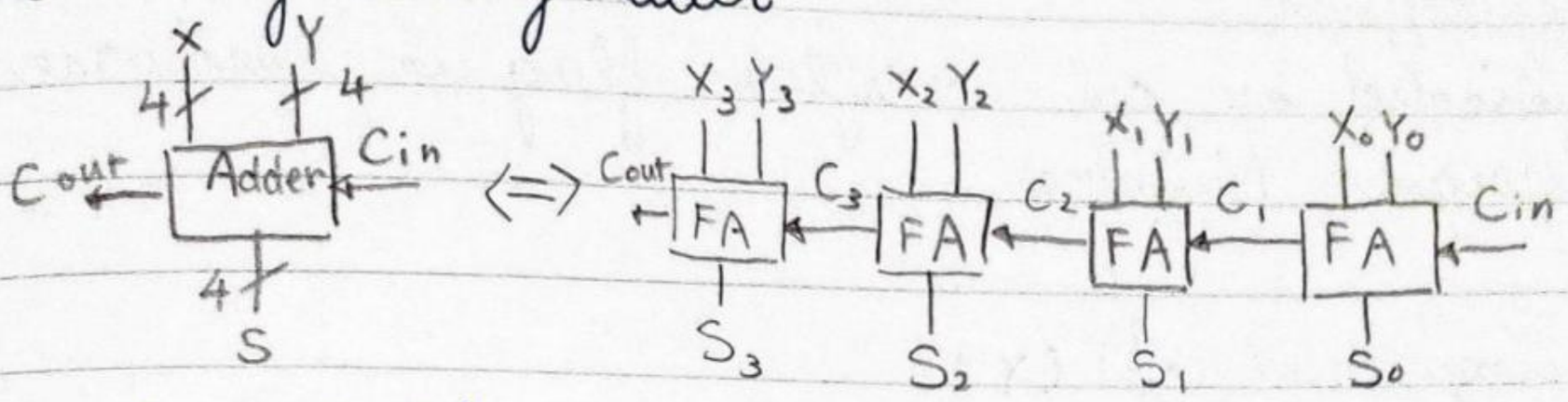
$S = X \oplus Y \oplus Z$



No.

Date

Ripple-Carry Binary Adder



this work just like a unsigned binary addition

SUBTRACTORS

- perform addition on a negative number

negative binary numbers

- Sign-and-magnitude e.g. $-18 = \overset{\text{-ve}}{\downarrow}\ 1\ 10010$

  uses a sign bit (to represent +ve / -ve)

- Signed (2's complement) e.g. $-18 = \overset{-2^5}{\downarrow}\ 10\ 1110$

  most significant bit (MST) has negative value

  1. invert each bit i.e. 1 → 0 , 0 → 1 (1's complement)
  2. +1 to last bit (2's complement)

e.g. 18 = 010010

-18 : 101101

+ 1

101110

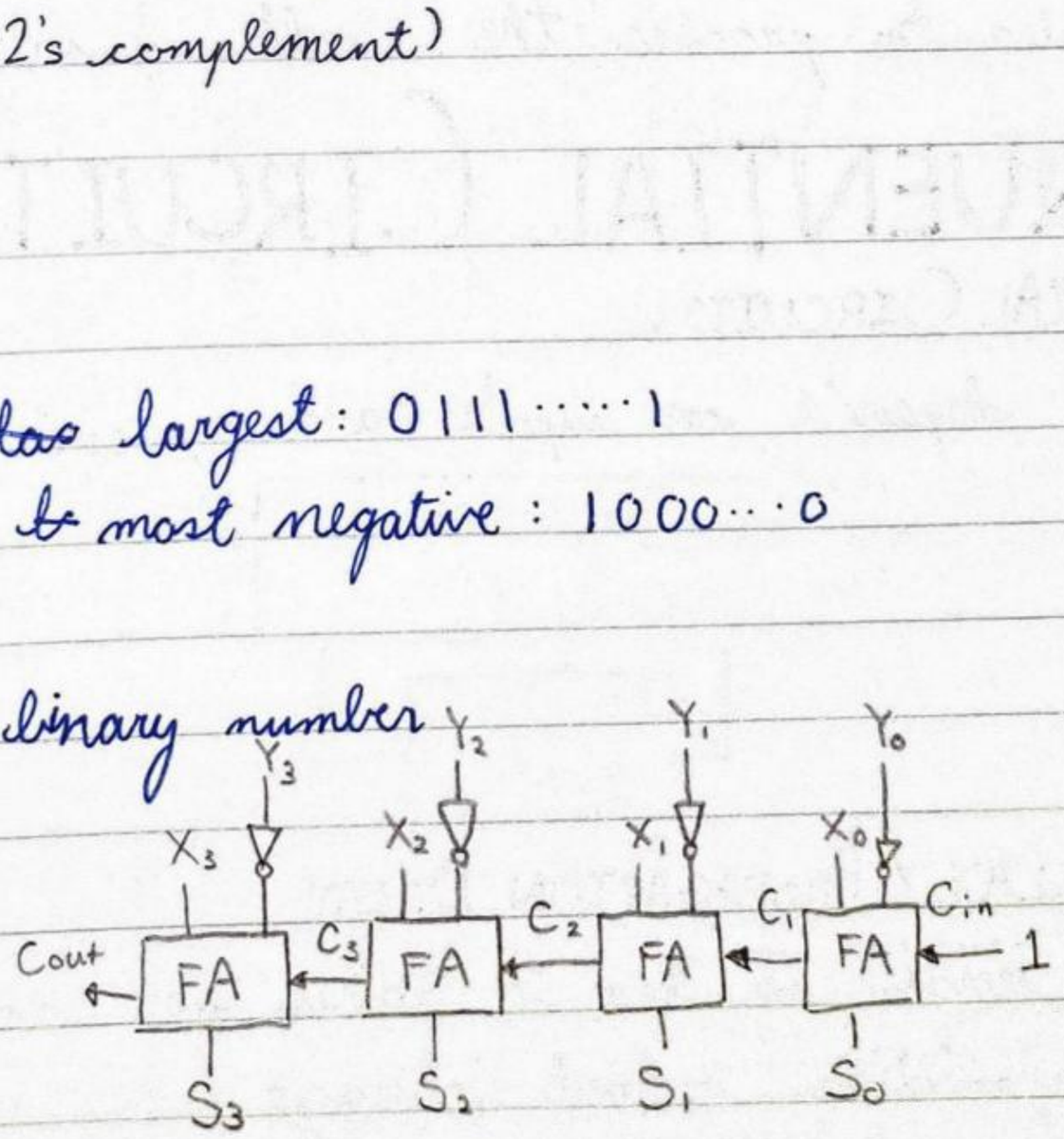
- 2^{n-1} -1 numbers are +ve, too largest : 0111……1
- 2^{n-1} numbers are -ve, & most negative : 1000…0
- 1 number is 0

2^n values for n-digit binary number

Subtraction Circuit

X - Y :

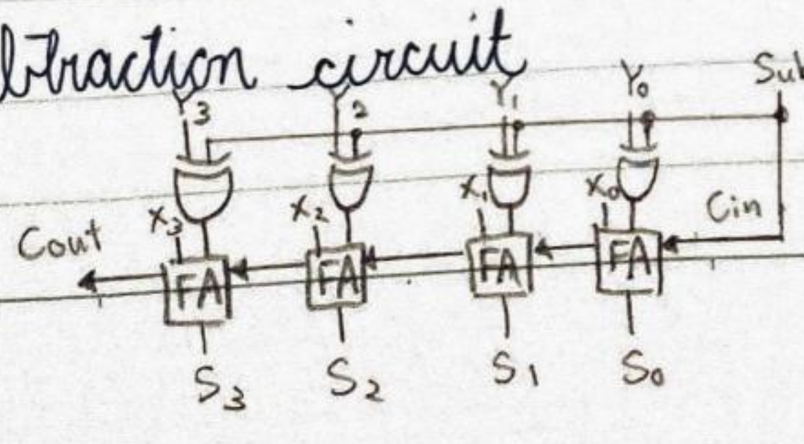
X plus 1's complement of Y + 1



Addition / Subtraction circuit

A ⊗ 0 = A

A ⊗ 1 = ¬A



Sub → Sub = 0 ⇒ addition

Sub = 1 ⇒ subtraction

No.

Date . .

When overflow happens...
would be indicated as an overflow flag in hardware
Subtracting Unsigned Numbers
X - Y

1. find 2's complement of Y (Y`)
2. X + Y`
3. if there is end carry (Cout is high), X-Y is positive => sign bit of output =0
4. if no end carry (Cout is low), X-Y is negative => 2's complement and sign bit of output =1

COMPARATORS

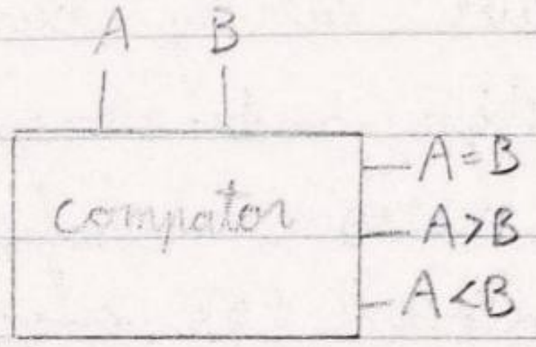
Basic (1-bit) comparator

A==B: $A \cdot B + \bar{A} \cdot \bar{B}$

A>B: $A \cdot \bar{B}$

A<B: $\bar{A} \cdot B$

A	B
0	0
0	1
1	0
1	1



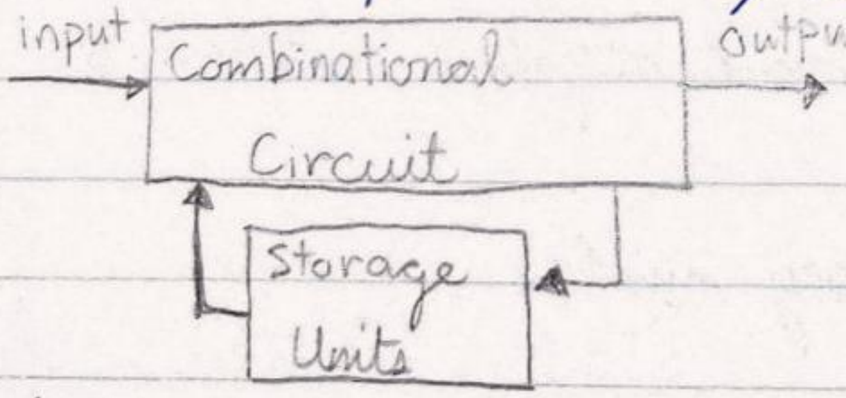
gets more complicated with more bits!
=> easier to process the result of subtraction

SEQUENTIAL CIRCUITS

28/01/2020

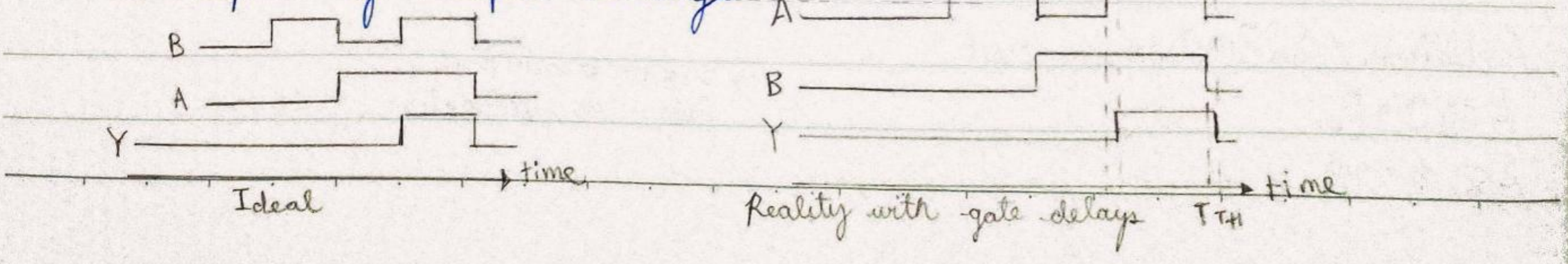
SEQUENTIAL CIRCUITS

- output depends on inputs and previous state of ~~circuit~~ circuit



GATE DELAY / PROPAGATION DELAY

A small length of time it takes for an input change to result in the corresponding output change

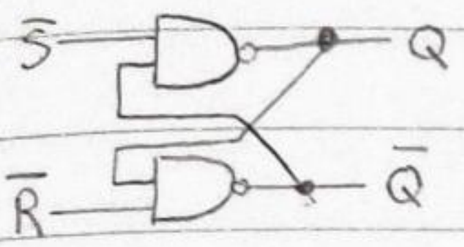


No.

Date

Problem : D-latch cannot switch outputs D-latch switches back and forth (unstable)

LATCHES



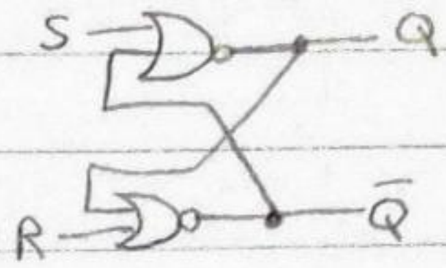
S̄R̄ latch

S̄	R̄	Q_T	$\bar{Q}_T$	Q_{T+1}	$\bar{Q}_{T+1}$
0	0	X	X	1	1
0	1	X	X	1	0
1	0	X	X	0	1
1	1	0	1	0	1
1	1	1	0	1	0

S̄ = "set" R̄ = "reset"

from 01 → 11 / 10 → 11, ~~value~~ output stored

* cannot 00 → 11 as dunno which input changes first (unstable behaviour)
  ↳ 00 forbidden state



SR latch

S	R	Q_T	$\bar{Q}_T$	Q_{T+1}	$\bar{Q}_{T+1}$
0	0	0	1	0	1
0	0	1	0	1	0
0	1	X	X	0	1
1	0	X	X	1	0
1	1	X	X	0	0

S = "set" R = "reset"

from 10 → 00 / 01 → 00, ~~output~~ output stored

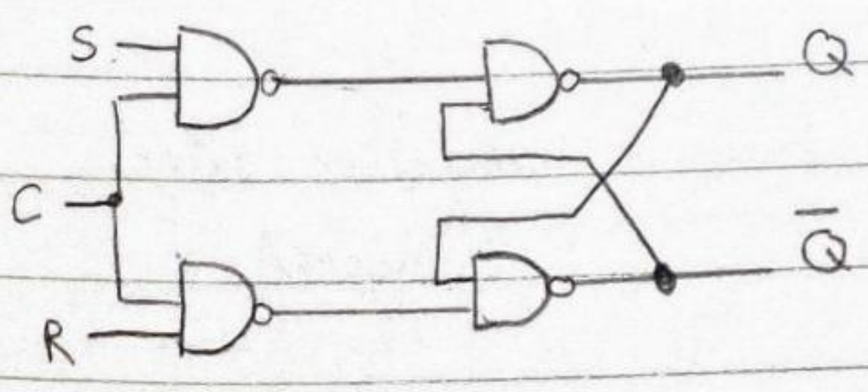
* cannot 11 → 00 because of unstable behaviour
  ↳ 11 forbidden state

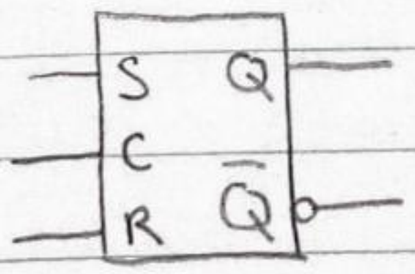
Problem : delay in feedback can cause unexpected behaviour

CLOCK SIGNALS

regular pulse signal, high values → output of latch can be sampled.

CLOCKED SR LATCH





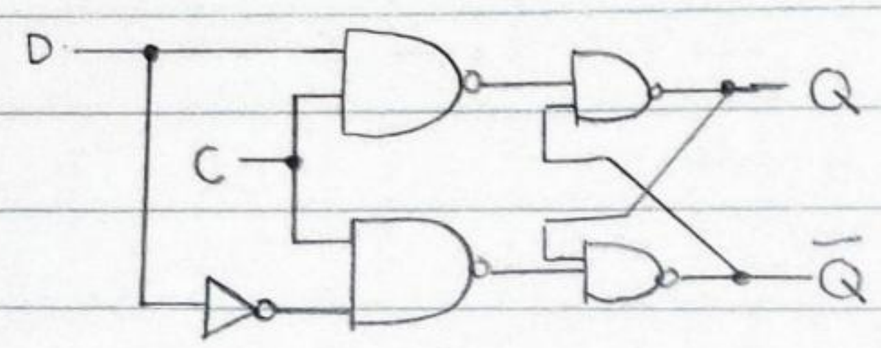
Q: Output only changes when clock input is high.
Not matter how S and R change when clock is low, output is not affected

Problem : Q still undetermined when S and R both 1

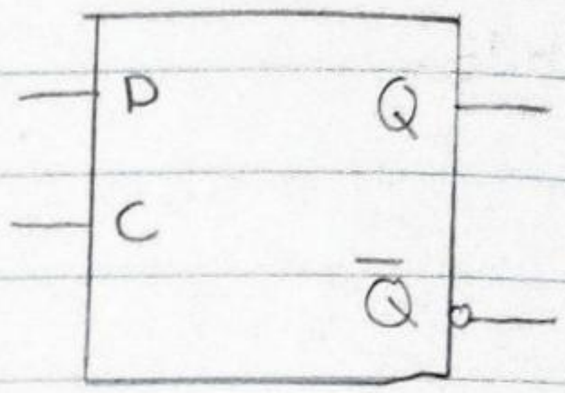
No.

Date

D LATCH



Q_T	D	Q_{T+1}
0	0	0
0	1	1
1	0	0
1	1	1

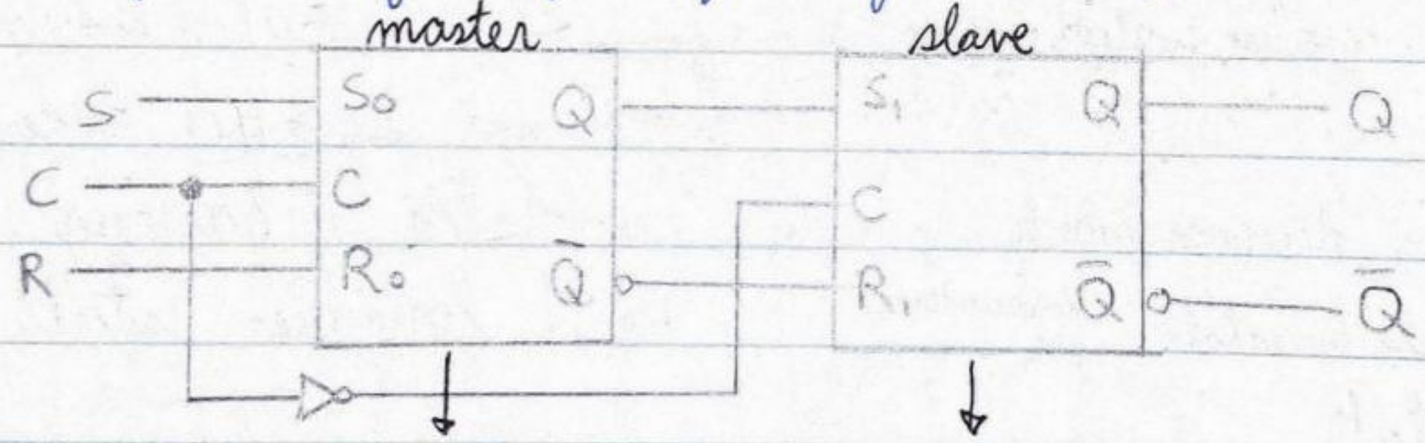


Make R and S dependent on only one signal D to avoid forbidden state.

⇒ D sets Q low / high whenever C is high

SR MASTER-SLAVE FLIP-FLOP

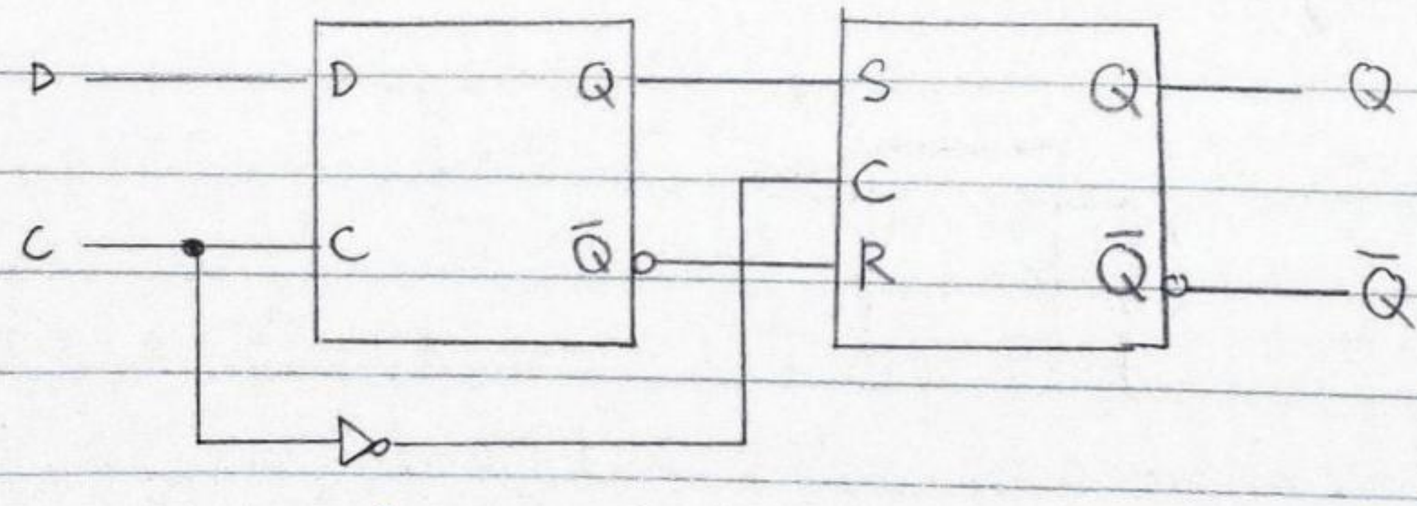
flip-flop - latched circuit whose output is triggered with the rising edge or falling edge of clock pulse



works like a regular latch

follows ~~output~~ output of master when clock is 0

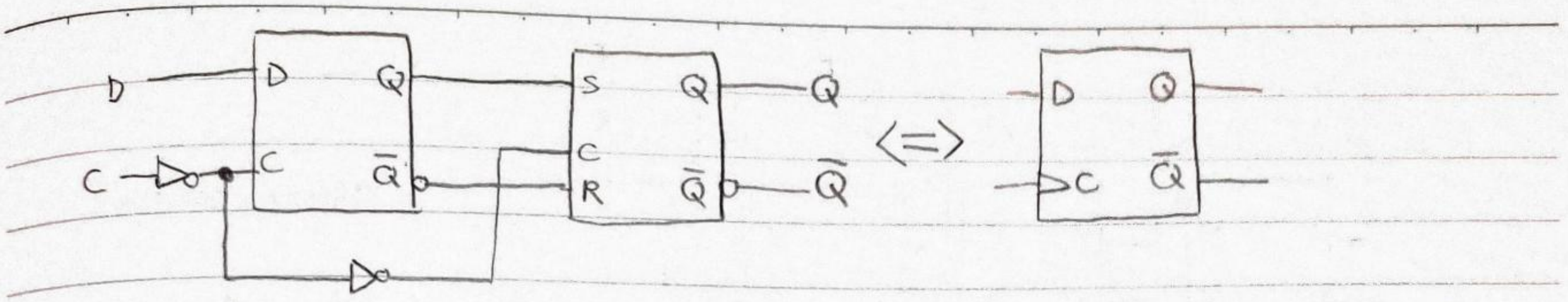
EDGE-TRIGGERED D FLIP-FLOP



negative edge triggered

when C=1, master reads input D, slave ignores any input

when C=0, master ignores input D, slave reads input from master



positive-edge triggered flip-flops (end output changes when clock is 1)

Other Flip-Flops

T-flip-flop

output changes whenever T is high (toggles)

JK flip-flop

J=0, K=0 ⇒ maintain output J=1, K=0, set output to 1

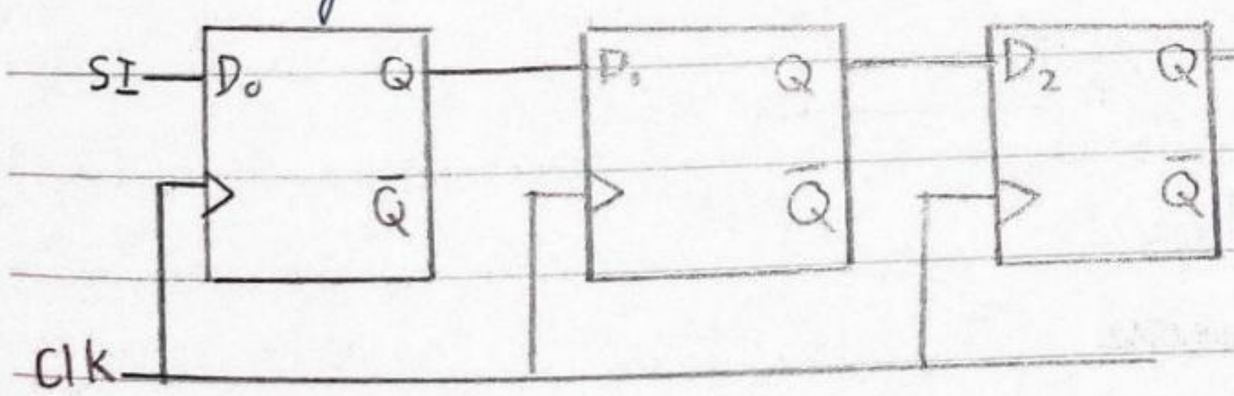
J=0, K=1 ⇒ set output to 0 J=1, K=1, toggle output

FSM

Finite State Machines

04/02/2020

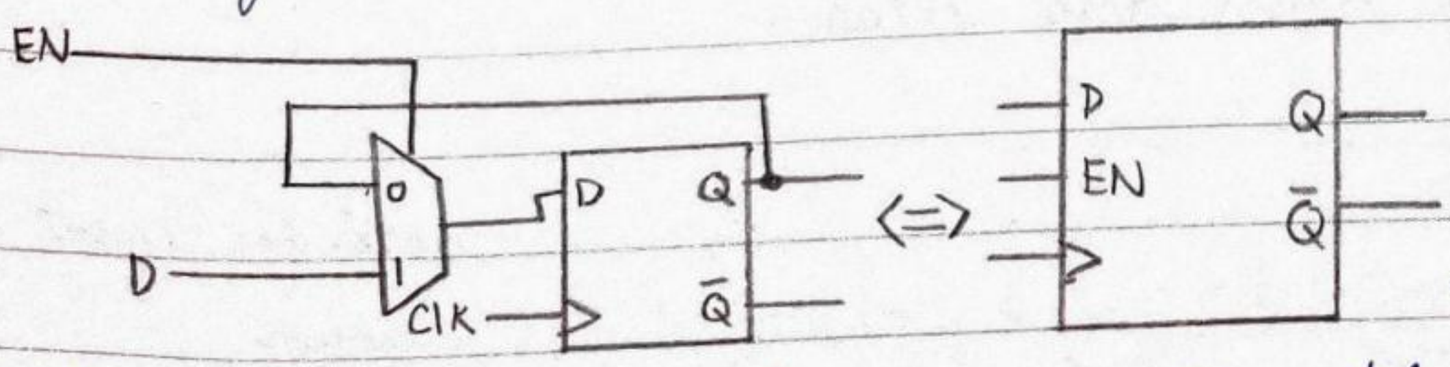
~~Load~~ Shift Register



1 bit can be shifted from D_n to D_{n+1} every clock cycle (D flip-flops in series)

↑ same clock

Load Registers



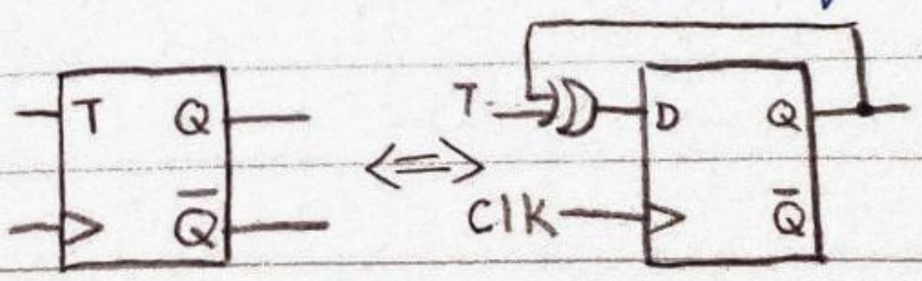
connect each register with a mux to load a bit (so doesn't have to start at 000...0)

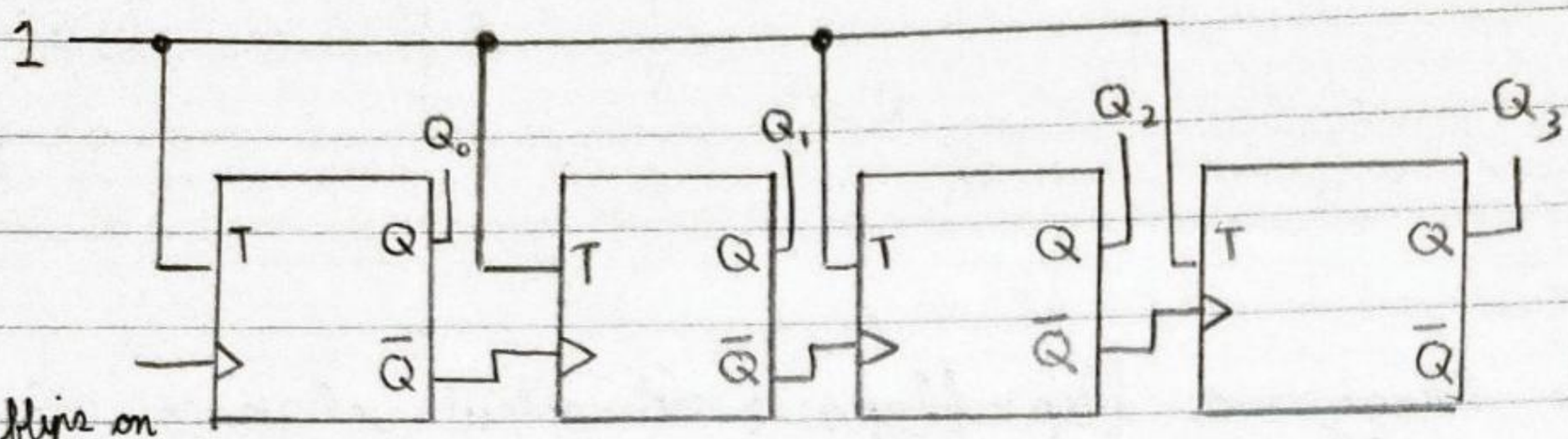
← D flip-flop with enable

Will maintain old value in register until EN is set to high (will be overwritten by load)

Counters

Connect T flip-flops together

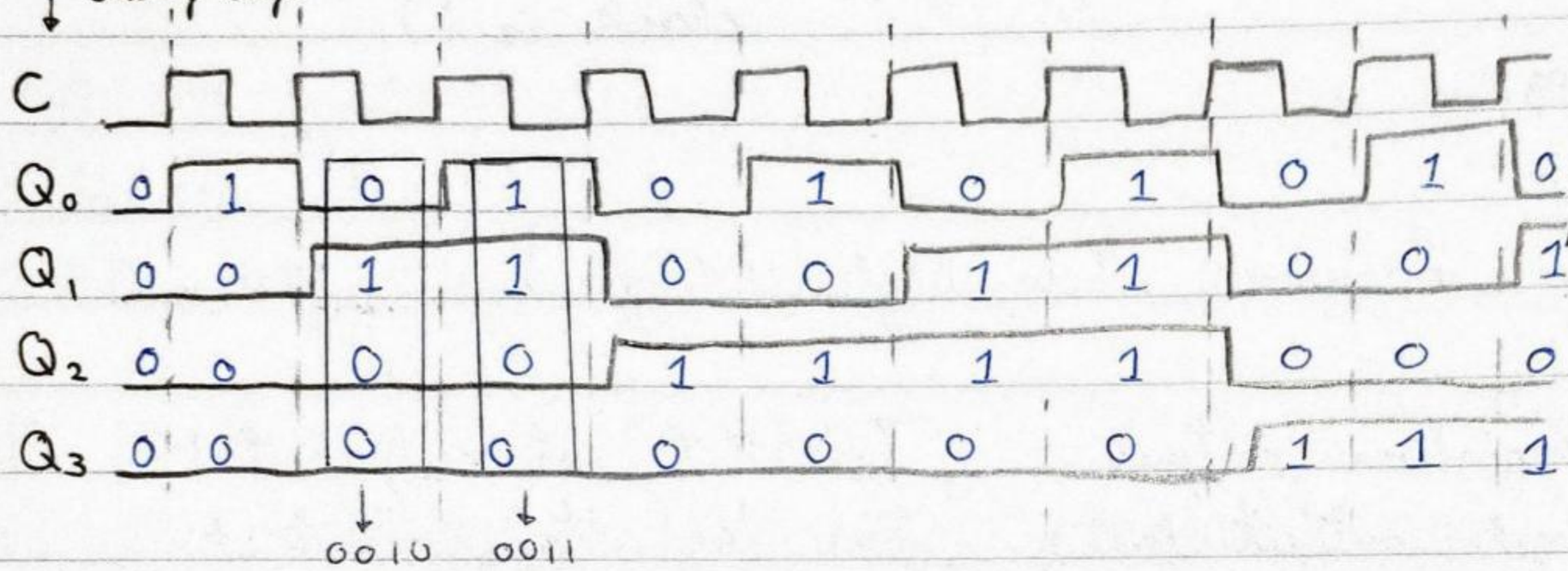




asynchronous

4-bit ripple counter

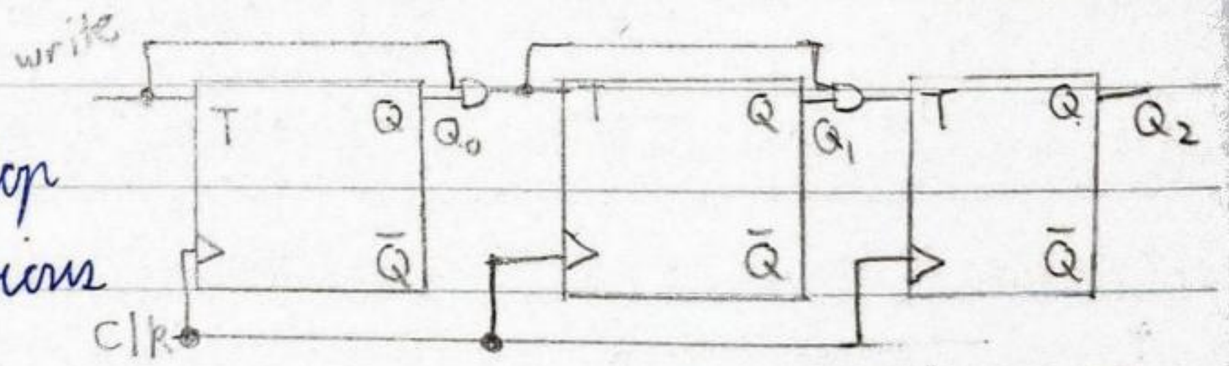
Q_0 flips on rising edge



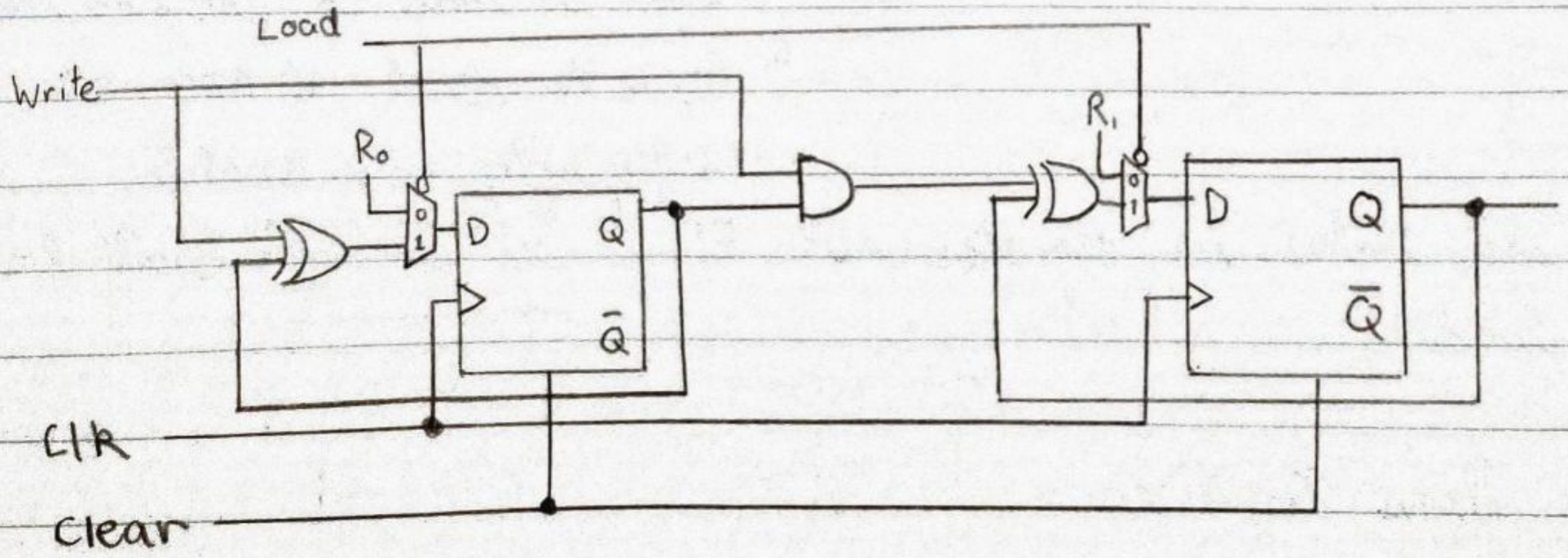
Q_0 has falling edge
$\Rightarrow \bar{Q}_0$ has rising edge

every clock cycle the counter increments ~~f~~ by one.
But because of gate delay, unreliable ~~to~~ for timing (different clock)

Synchronous Counter
slight delay
AND connect all outputs of flip-flop
changes value only when all ~~ft~~ previous
flip-flops are 1



implementing with parallel load and clear...



used for count down

No.

Date

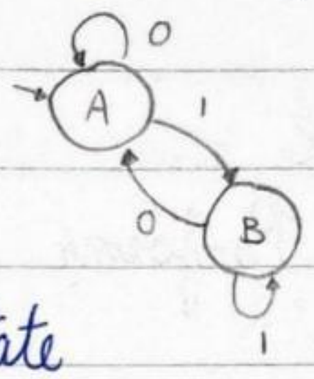
State Machines

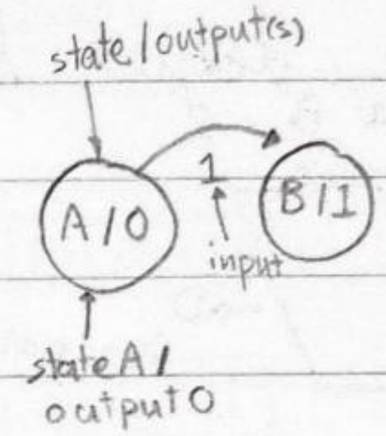
Sequential circuits uses combinational logic, ~~and~~ past state and current input to determine the output

state diagrams describe transition between states

Finite State Machine

abstract model that shows operations of sequential circuit



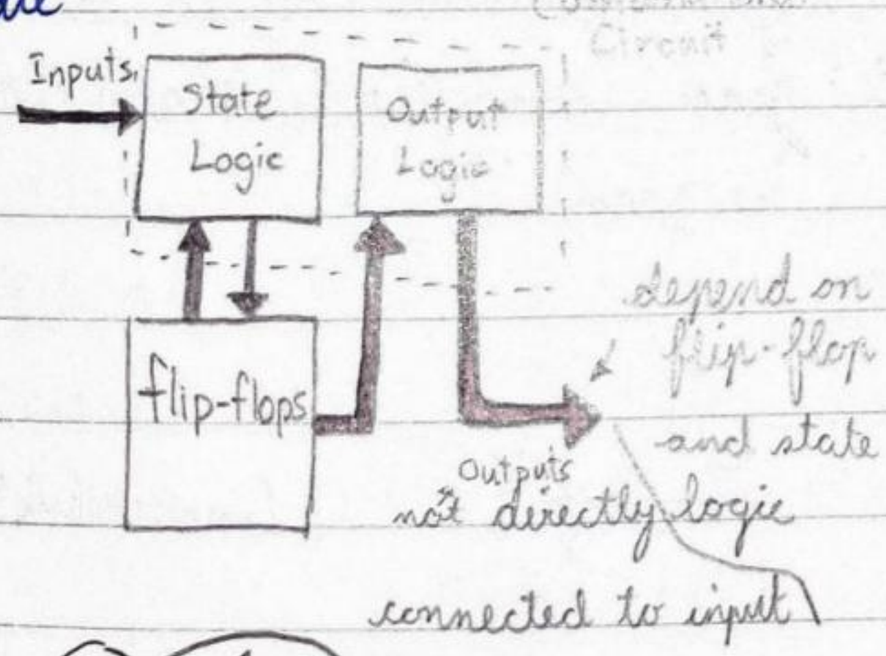


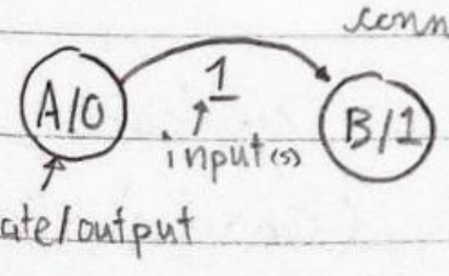
Design

1. Draw state diagram
2. Derive state table from state diagram
3. Assign flip-flop configuration to each state
   ↳ # flip-flops = $\lceil \log_2(\#\text{states}) \rceil$
4. Redraw state table with flip-flop values
5. Derive combinational circuit for output and for each flip-flop input

Moore machine

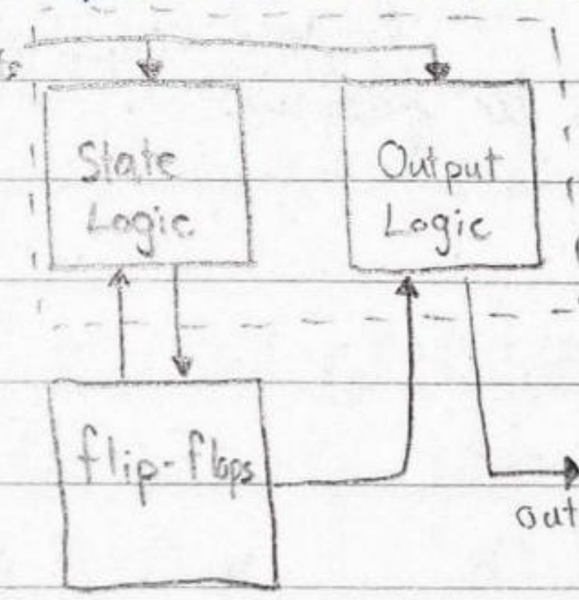
output only depend on current state





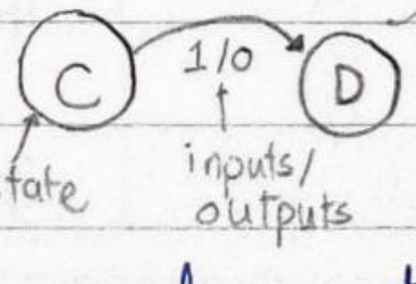
Mealy machine

output depend on state and input



input does not necessarily pass through flip-flops

- chance of transparent circuit



When assigning states, beware of situtations where two ~~input~~ more input values has to be change when transitioning to another state.

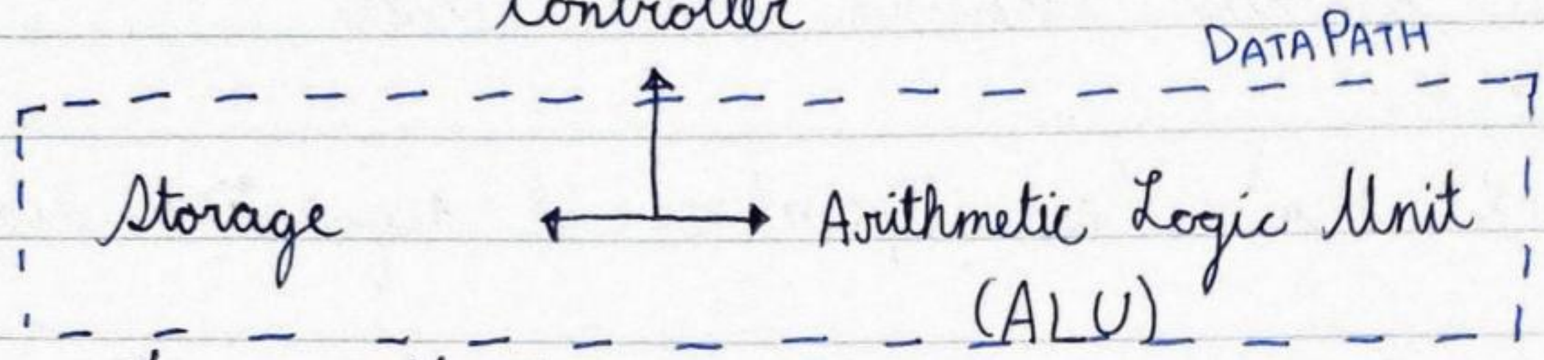
Chance of changing output when one of the ~~two~~ many inputs changes first

No.

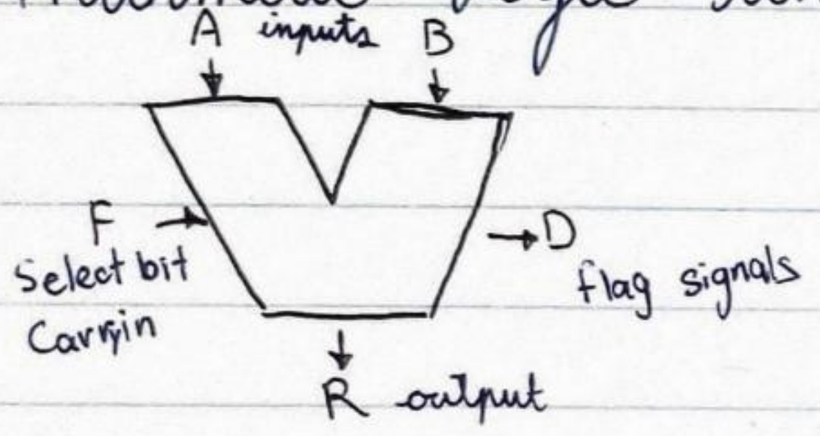
Date

PROCESSORS

23/02/2020



Arithmetic Logic Unit



Performs arithmetic operations (+, -, *, /) and logical operations (AND, OR, NOT)

inputs

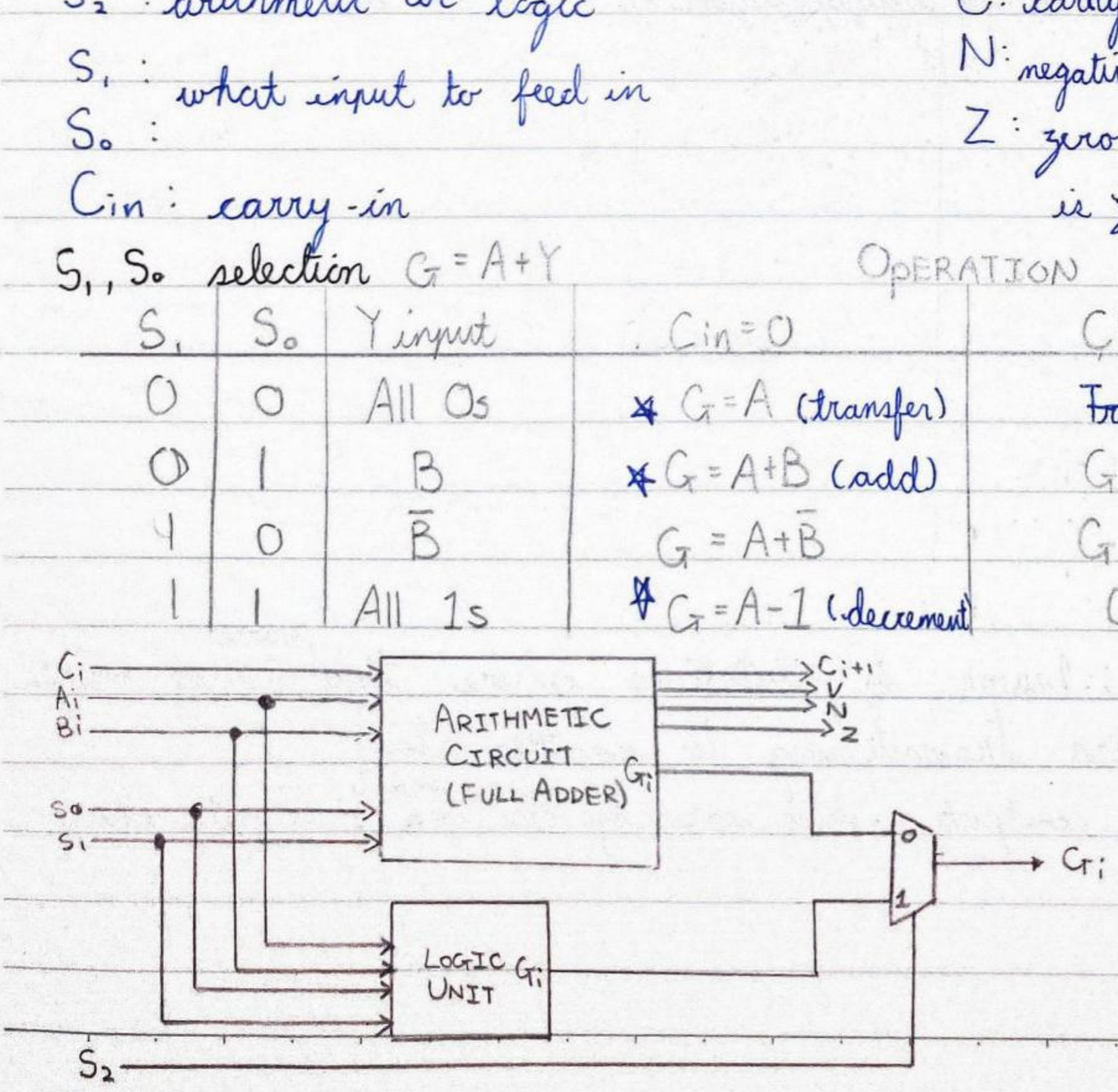
A, B: values to perform operations on
S_2: arithmetic or logic
S_1: what input to feed in
S_0:
C_{in}: carry-in

outputs

V: overflow condition
C: carry-out bit
N: negative indicator (signed bit)
Z: zero-condition indicator (result is zero)

S_1, S_0 selection $G = A + Y$

S_1	S_0	Y input	OPERATION $C_{in} = 0$	$C_{in} = 1$
0	0	All 0s	* $G = A$ (transfer)	~~Transfer~~ * $G = A + 1$ (increment)
0	1	B	* $G = A + B$ (add)	$G = A + B + 1$
1	0	$\bar{B}$	$G = A + \bar{B}$	$G = A + \bar{B} + 1$ * (subtract)
1	1	All 1s	* $G = A - 1$ (decrement)	$G = A$ (transfer)



S_2 selects output
$S_{0,1}$ select operation

Binary Multiplication

e.g.

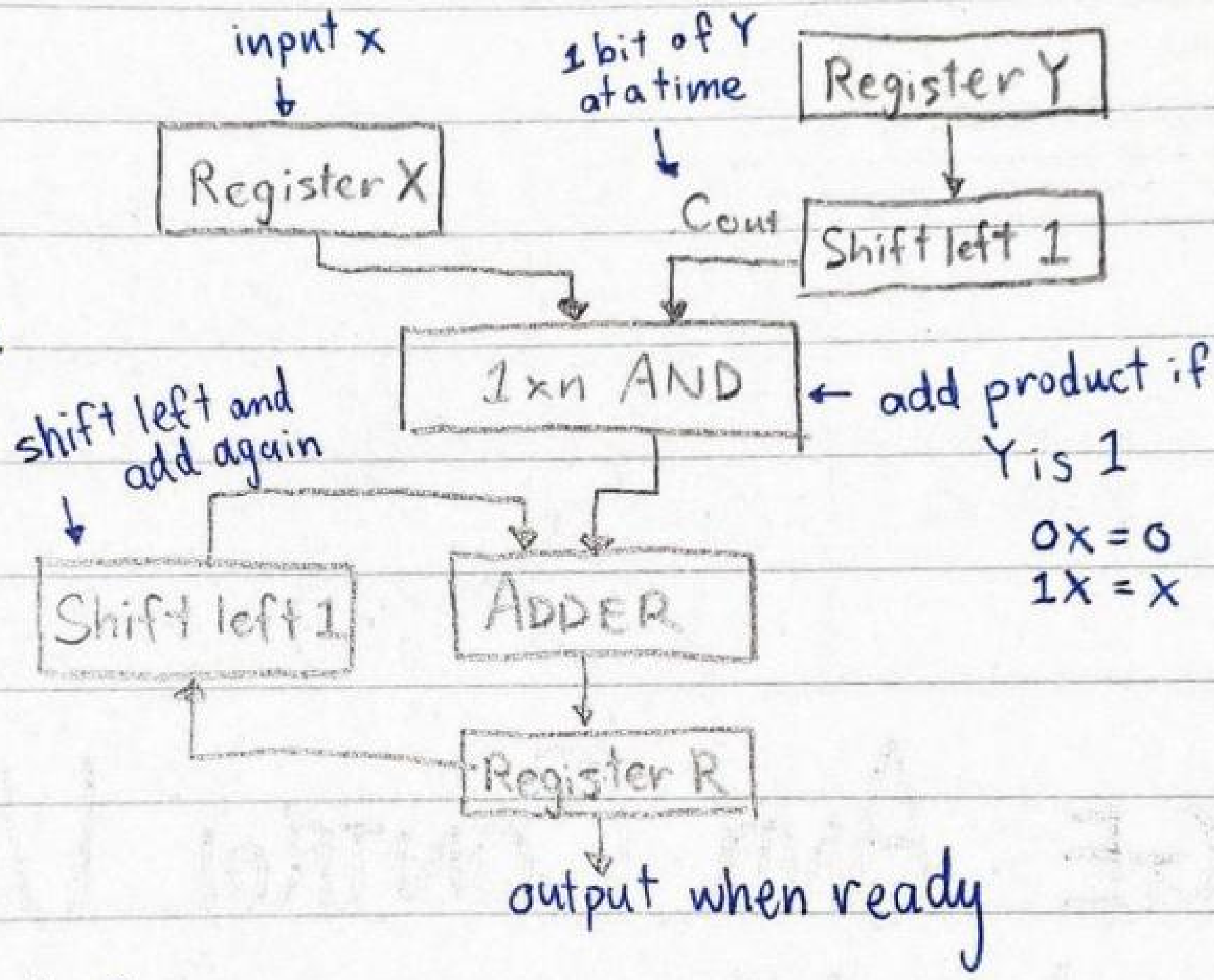
```
   101
 × 110
 -----
   000
  101
 101
 -----
 11110
```

Look at the digits on the lower one starting from the least-significant bit. If 0 then put zeros on that line; if 1 copy the top number with least-significant bit aligned with the bit you are working on. Move on to the next bit

if directly implement adders in circuit, need $O(N^2)$ half adders

Accumulator Circuits

have register to shift the bits



Sign Extension

adding most significant bit to the right extends the number while keeping its original value

0101 → 0000 0101
(5)

Booth's Algorithm

go through digits from n-1 to 0

if d[i] = 0 and d[i-1] = 1, the multiplicand is added to result at position i.

if d[i] = 1 and d[i-1] = 0, the multiplicand is subtracted from result at position i.

```
            01010010  → B
      × 0001 1 1 10   → A
    010100100          ← add B        (from boxed 01)
1111 10101110          ← 1's complement (subtract B, from boxed 10); 1111 = sign extension
-------------
0100110011100
```

1. Let the ~~md~~ multiplicands as A and B and result be P
2. Add an extra zero bit to the right-most side of A
3. Repeat for each bit in A (except the extra zero):
   a. if bit 1 == bit 0 do nothing
   b. if 01 then add B to highest bits of P, if 10 then subtract B from highest bits

No.

Date

c. perform one-digit arithmetic right-shift on P and A

4. P is the product of A and B

DATAPATH Vs CONTROL

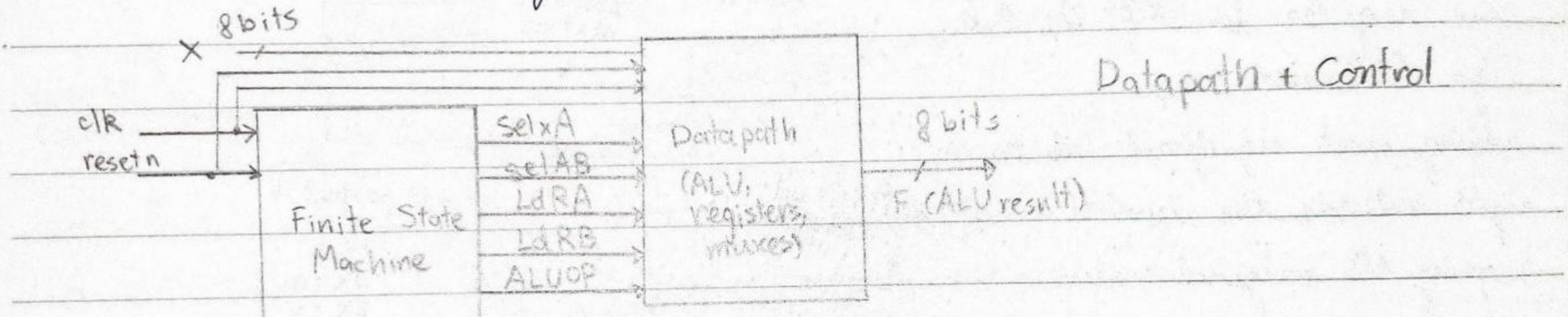
~~Data~~ Datapath: where all data computations take place

Control Unit: decides actions that take place in the datapath

- is a finite state machine

outputs datapath control signals

e.g. mux outputs / ALU inputs, ALU operation select, when to load registers



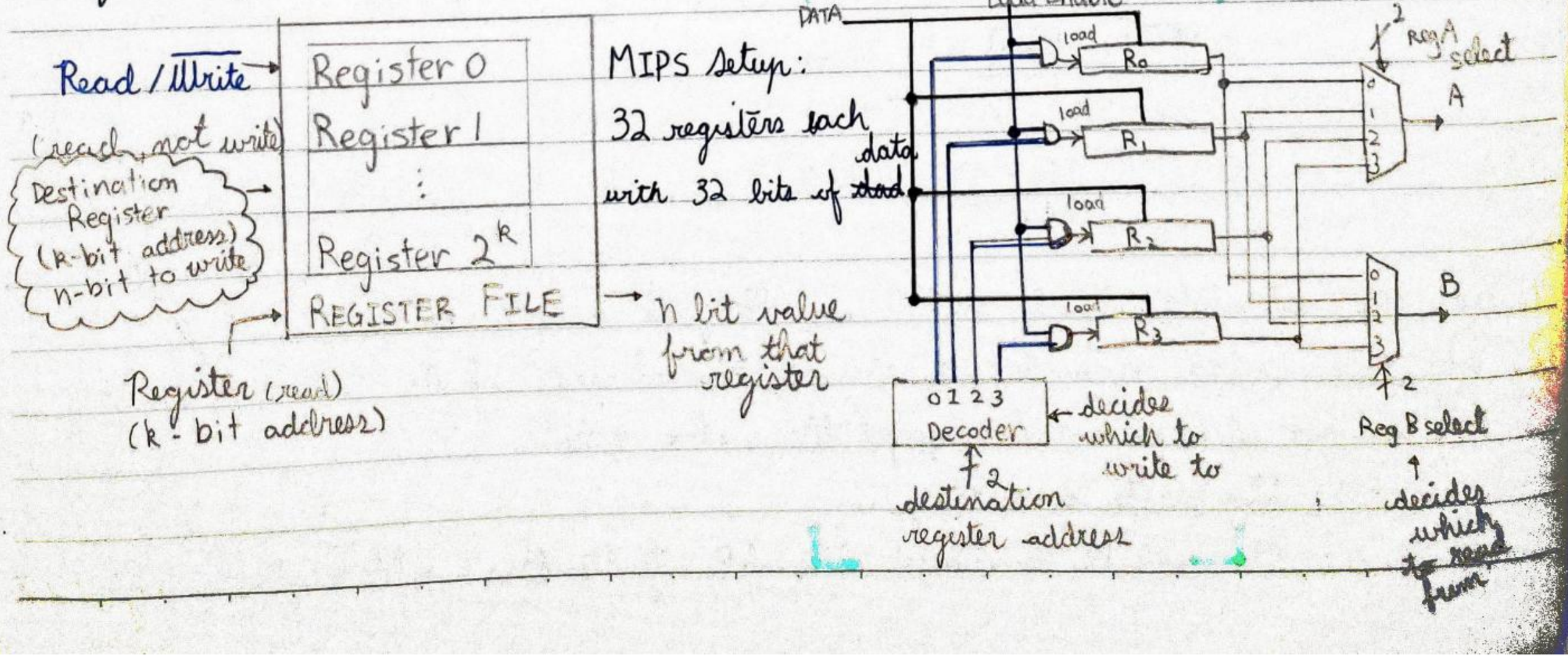
STORAGE AND CONTROL UNIT

25/02/2020

Register: few but fast memory units (read and write / for computation)

Main Memory: larger grid of slower memory cells (store main information)

Register file functionality



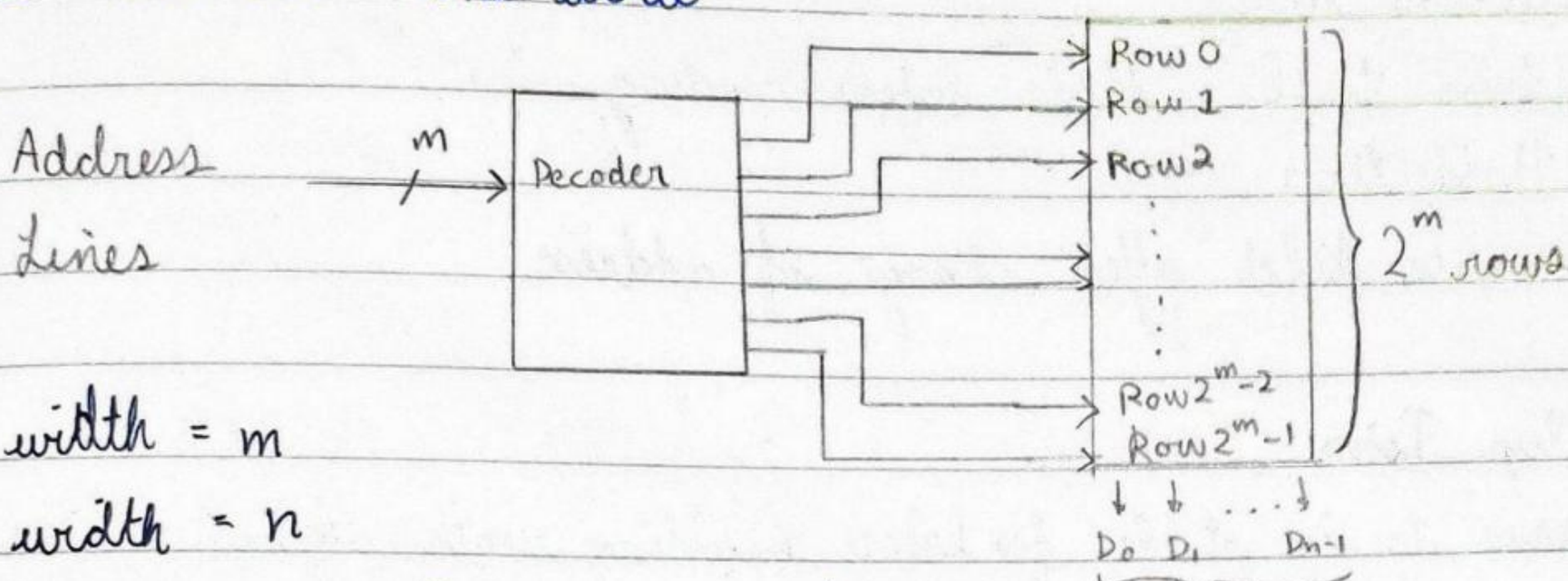
No.

Date

Main Memory and Addressing

in bytes (8-bits)

4 bytes = 32-bit = one word



address width = m

data width = n

size of memory = $2^m \cdot n$ bits $\Rightarrow 2^m \cdot \frac{n}{8}$ bytes

each row has n storage cells

~~other~~ e.g. RAM cell (SR latch and 4 AND gates)

DRAMIC cell (1 transistor and 1 capacitor)

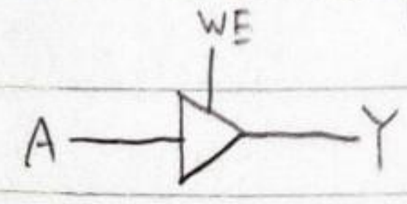
~~Wordline~~ Wordline which row (word) to read/write

Bitline read/write data

Data Bus

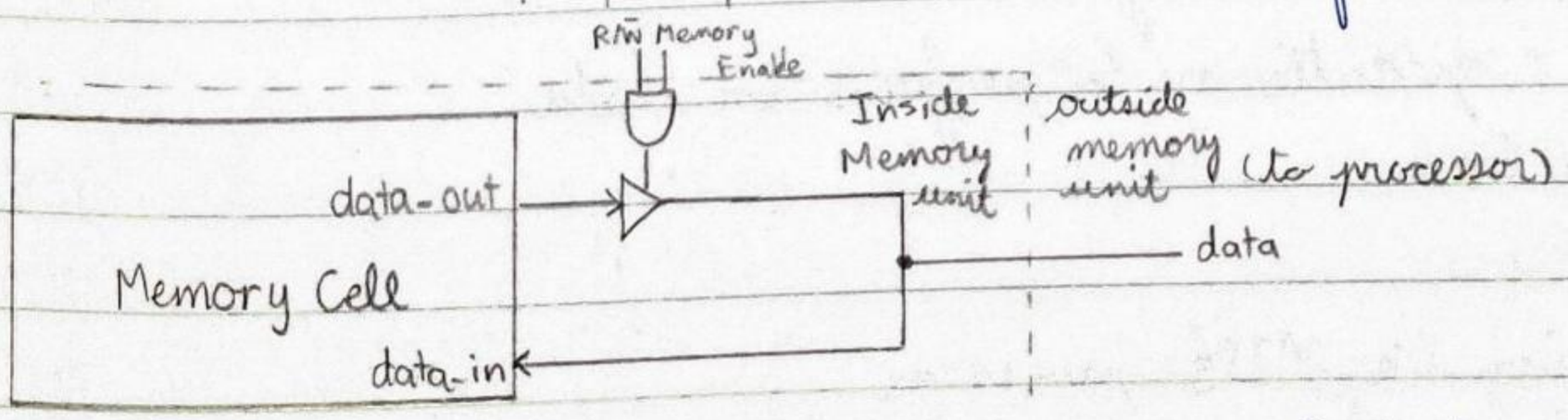
a wire that is connected to many components, but only one pushes output at one time (write)

Tri-state buffer



write enable WE	A	Y
0	X	Z ← high impedance (disconnected wire)
1	0	0
1	1	1

used to control data lines so that only one device can write onto the bus at any given time (read from multiple device is fine)



R/W̄ and Memory enable is 1 $\Rightarrow$ data is output to processor

R/W̄ is 0 $\Rightarrow$ data is ~~xx~~ input from processor to memory

No.

Date

Reading from RAM is slower; need signals to control read/write timing

Read:

t_{AA} = Address Access time

time for address to be stable before reading

t_{OHA} = Output Hold time

time output data held after change of address

Write:

t_{SA} = Address Setup Time

time for address to be stable ~~fo~~ before enabling write signal

t_{AW} = Address Setup Time to write End

t_{SD} = Data Setup to Write End

time for data-in value to be set-up at destination

t_{HD} = Data Hold from Write End

time data-in value should stay unchanged after write signal changes

Lec Load-Store Architecture

MIPS processor architecture

load data from main memory to registers

process them using ALU

store back in main memory

The control Unit

gets information from instruction

selects: source : where to get data

destination : where to go

operation : operation(s) to perform on data

loads next instruction,

Instruction

32 bit binary string in MIPS processor

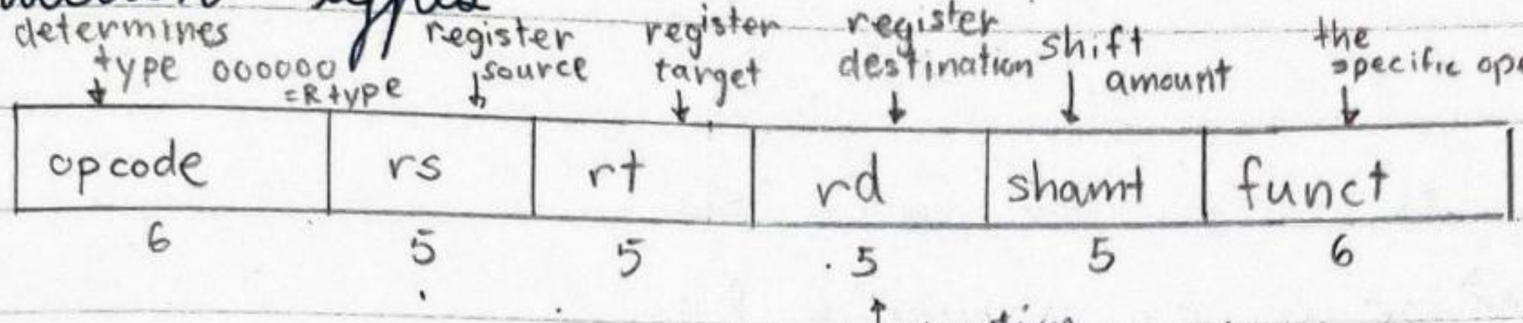
- AKA control ~~word~~ word

- ~~to each~~ instruction to be executed stored in Program Counter (PC)

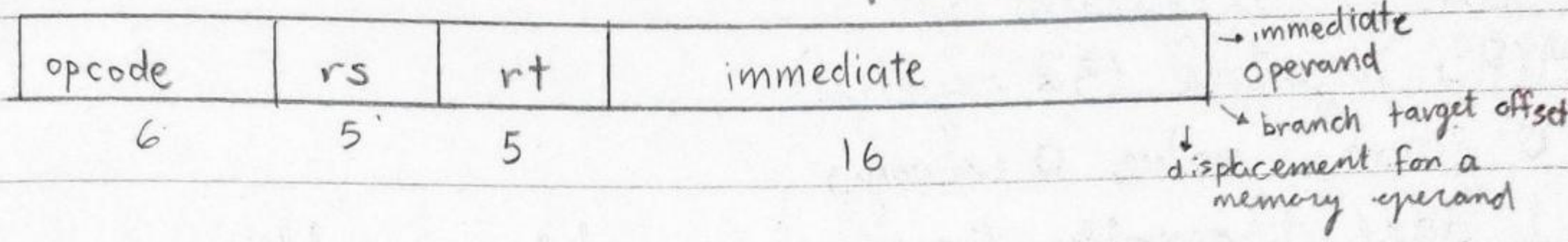
- increment by 4 (32 bits = 4 bytes)

MIPS instruction types

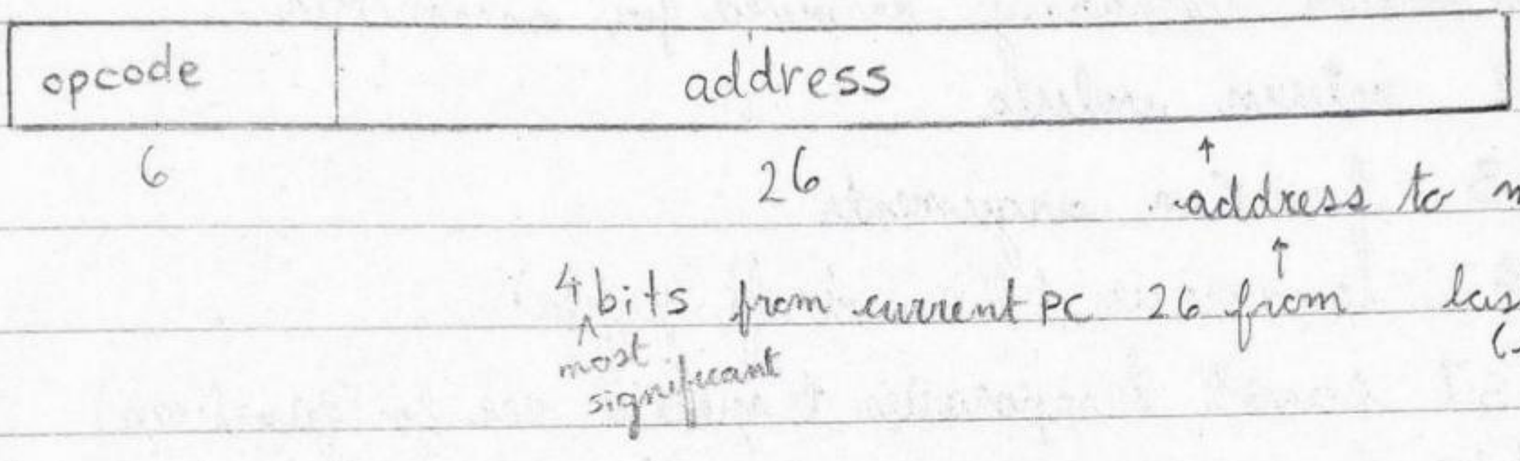
R-type register type



I-type immediate



J-type jump



Control Unit signals

PCWrite: write ALU output to PC
PCWriteCond: same as PCWrite, only if the zero condition has been met
IorD: ~~onto~~ instruction or data. tells whether the memory address is from PC (instructions) or an ALU operation (data)
MemRead: processor reading from memory
MemWrite: processor writing to memory
MemToReg: what goes into register (from memory, not from ALU output)
IRWrite: instruction register overwritten by new instruction from memory
PCSource: jump / ALU operation result (value of PC)
ALUOp (3+ wires): execution of an ALU operation
ALUSrcA: input A to ALU is from PC or register file
ALUSrcB (2 wires): input B to ALU is from register file, constant (4), instruction register or shifted instruction register
RegWrite: processor writing to register file
RegDst: decides destination address (rt / rd)

No.

Date . .

ASSEMBLY CODE

03/03/2020

Program

1. write code
2. compile code into machine code instructions
3. save instructions in an executable file
4. run the executable file

MIPS Registers (32 registers)

BLACK: special values
BLUE: function params
D.BLUE: store values

0 $zero : value 0 (always)
1 $at : assembler temporary, reserved for assembler
2-3 $v0, $v1 return values
4-7 $a0 - $a3 : function arguments
8-15 $t0 - $t7 temporaries (like draft paper)
16-23 $s0 - $s7 saved temporaries (special use for functions)
24-25 $t8, $t9 temporaries
26-27 $k0 - $k1 reserved for OS Kernel (not usable)
28-31 $gp, $sp, $fp, $ra : memory and function support (reserved for management)

$gp: global pointer to data segment
$sp: stack pointer to top of stack
$fp: frame pointer to function frame start
$ra: return address from func

PC, HI, LO accessed by special instructions

HI, LO: multiplication and division use

Assembly Language

Microarchitecture vs. ISA

ISA (instruction set architecture)
set of instructions supported by a processor (contract between processor and programmer); instructions you can give to processor

Microarchitecture
implementation of ISA on a processor
what processor actually does

Computer architecture / architecture : combination of both

No.

Date

INSTUCTION	OPCODE / FUNCTION	SYNTAX	OPERATION
Arithmetic			
add ← stop execution of current code if overflow	100000	$d, $s, $t	$d = $s + $t
addu ← unsigned	100001	$d, $s, $t	$d = $s + st
addi ← immediate	001000	$t, $s, i	$t = $s + SE(i)
addiu	001001	$t, $s, i	lowbits $t = $s + SE(i)
div ← different when doing sign extension	011010	$s, $t	integer div → lo = $s / $t ; hi = $s mod $t
divu ←	011011	$s, $t	lo = $s / $t ; hi = $s mod $t
mult	011000	$s, $t	high bits → hi:lo = $s * $t
multu	011001	$s, $t	hi:lo = $s * $t
sub	100010	$d, $s, $t	$d = $s - $t
subu ← unsigned	100011	$d, $s, $t	$d = $s - $t
Logical			
and	100100	$d, $s, $t	$d = $S & $t
andi	001100	$t, $s, i	$t = $s & ZE(i)
nor	100111	$d, $s, $t	$d = ~($s | $t)
or	100101	$d, $s, $t	$d = $s | $t
ori	001101	$t, $s, i	$t = $s | ZE(i)
xor	100110	$d, $s, $t	$d = $s ^ $t
xori	001110	$t, $s, i	$t = $s ^ ZE(i)
Shift			
sll ← shift logical left	000000	$d, $t, a	$d = $t << a
sllv ← v = variable	000100	$d, $t, $s	$d = $t << $s
sra ← shift right arithmetic	000011	$d, $t, a	$d = $t >> a
srav	000111	$d, $t, $s	$d = $t >> $s
srl ← shift right logical	000010	$d, $t, a	$d = $t >>> a
srlv	000110	$d, $t, $s	$d = $t >>> $s
Data Movement			
mfhi ← move from	010000	$d	$d = hi
mflo ←	010010	$d	$d = lo
mthi ← move to	010001	$s	$hi = $s
mtlo ←	010011	$s	lo = $s

sign extension

zero extend put 0 for upper bits

shift a bits

shift by value in $s

No.

Date

INSTRUCTION	OPCODE / FUNCTION	SYNTAX	OPERATION
lui Load Upper immediate			
lui - loads 16 bit immediate to upper half and lower set to 0	001111	$t, i	$t = i<<16
Branch			
beq	000100	$s, $t, label	if($s==st) pc←label
bgtz - greater than 0	000111	$s, label	if($s>0) pc←label
blez - lesser equal to 0	000110	$$s, label	if($s<=0) pc←label
bne	000101	s $t, label	if($s!=$t) pc←label
Jump			
j	000010	label	pc ← label
jal - jump and link	000011	label	$ra=pc; pc←label
jalr } not j-type	001001	$s	$ra=pc; pc=$s
jr } not j-type	001000	$s	pc=$s
Comparison			
slt - set less than	101010	$d, $s, $t	$d=($s<$t)
sltu - unsigned	101001	$d, $s, $t	$d=($s<$t)
slti	001010	$t, $s, i	$t=($s<SE(i))
sltiu	001001	$t, $s, i	$t=($s<SE(i))

($ra stores address that's used when returning from a subroutine)

(store 1 if true 0 else)

Formatting Assembly Code

```
.text - #start with this
main :    # label to tell assembler this is the first line
    <instruction>  <parameters>   # how instruction is written
```

IF/ELSE STATEMENT

```
#$t1 = i   $t2 = j
                                  # black means the way to "branch on else first
main : beq $t1, $t2, IF           # branch to IF if value of t1 = value of t2
       addi $t2, $t2, -1          # subtract 1 from t2 (else)
       j END                      # jump to END
IF :   addi $t1, $t1, 1           # add 1 to t1 (if)
END :  add $t2 $t2 $t1            # add t2 and t1 and store in t2
```

(bne) (ELSE) (ELSE) (both runs this)

No.

Date

Multiple conditions inside if

if (a || b)

```
main : (branch) <condition a> IF    # go to IF block
       (branch) <condition b> IF    # if either condition met
       ...                          # execute else block
       j END                        # if neither condition met
IF :   ...                          1 1 2 3 5 8 13 21 34 55
       j END
ELSE:  ...
END :  ...
```

if(a && b)

```
main : (branch) !<condition a> ELSE   # go to ELSE block
       (branch) !<condition b> ELSE   # if either condition not met
IF :   ...                            # execute if block if both condition met
       j END
ELSE:  ...
END :  ...
```

Loops

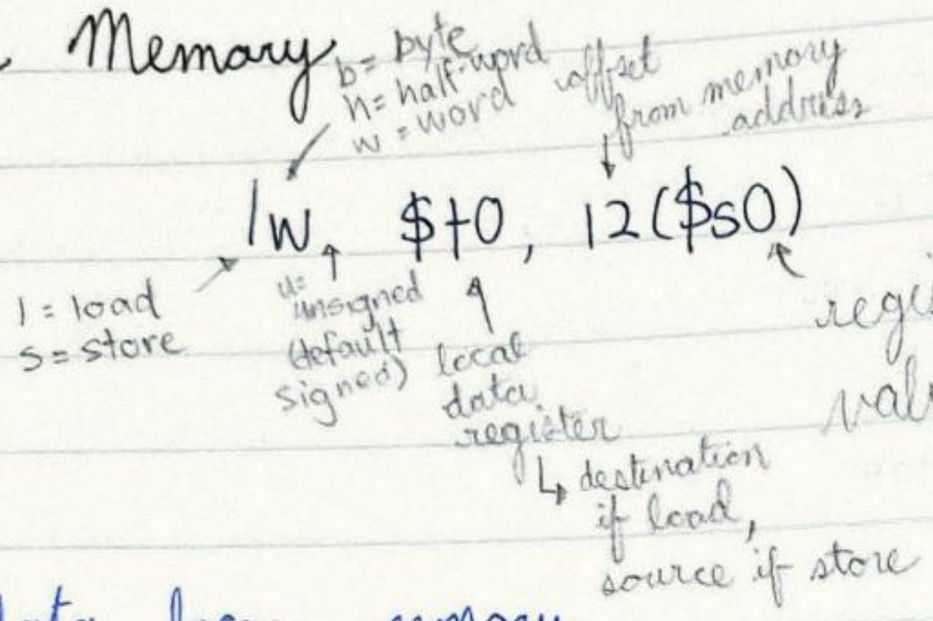
```
main : add  $t0 , $zero , $zero    # i = 0   (t0 = 0)
WHILE: beq
       beq  $t0 , $t1 , END        # branch to END if t1 = t0
       addi $t0 , $t1 , 100        # t1 = 100
       addi $t0 , $zero , 1        # t0 ++
       j WHILE                     # jump back to while
END :
```

No.

Date . .

24/03/2020

Load / Store Memory



load = get data from memory
store = save data to memory

Load & Store instructions

lb $t, i($s)	- $t = SE(MEM[$s+i]:1)	(sign extend) load byte from memory at address $s+i
lbu $t, i($s)	- $t = ZE(MEM[$s+i]:1)	(zero extend)
lh $t, i($s)	- $t = SE(MEM[$s+i]:2)	load half word " "
lhu "	- $t = ZE(MEM[$s+i]:2)	" " "
lw "	- $t = MEM[$s+i]:4	load word
sb "	- $MEM[$s+i]:1 = LB($t)	store byte
shsh "	- MEM[$s+i]:2 = LH($t)	sh store half-word
sw "	- MEM[$s+i]:4 = $t	store word

Alignment Requirements

word - divisible by 4 (address)
half-word - divisible by 2
byte - any

Pseudo-instructions

la $t, label	$t = address(MEM[label])	take address of label and store in t
li $t, i	$t = i	store immediate address

Labelling Data Storage

Labelled data storage / ~~vars~~ variables
create labels for memory locations that are used to store values

label: .type value(s)

No.

Date

e.g.
```
var1:    .word   3        # 4 byte integer called var1 with value 3
array1:  .byte   'a','b'  # two 1 byte characters ['a','b'] in an array with name array1
array2:  .space  40       # 40 consecutive bytes with uninitialized
                          # storage (type of element does not matter)
.data
# labels here are for variables

.text
main:
# labels here like main: are program labels and branch addresses
```

Arrays
sequence of data elements of SAME SIZE consecutive in memory
just a range of memory

e.g.
```
int A[100], B[100];
for (i=0; i<100; i++) {
    A[i] = B[i] + 1;
}
```

```
.data
A:    .space  400      # space for 400 bytes = 100 ints
B:    .word   21:100   # 100 words/integers with value 21
.text
main: la $t8, A        # address of
      la $t9, B        # first element
      add $t0, $zero, $zero   # holds 4*i
      addi $t1, $zero, 400    # holds 100*sizeof(int)
LOOP: bge $t0, $t1, END       # go to end if i >= 400
      add $t3, $t8, $t0       # address of A[i]
      add $t4, $t9, $t0       # address of B[i]
      lw $t5, 0($t4)          # get value in address stored in $t4
      addi $t5, $t5, 1        # +1 to $t5
      sw $t5, 0($t3)          # store value of $t5 into address in $t3
      addi $t0, $t0, 4        # add 4 to t1 (increment)
      j LOOP
END:
```

Structs
collection of fields one after another in memory

```
s:    .space  12       # space for intA, intB, intC
main: la      $t0, s
      addi    $t1, $zero, 5     # store 5 to A
      sw      $t1, 0($t0)
      add     $t1, $zero, 13    # store 13 to B (offset by 4)
      sw      $t1, 4($t0)
      addi    $t1, $zero, -7
      sw      $t1, 8($t0)       # offset by 8 store -7 to C
```

No.

Date

Writing a function func call(arg1, arg2, arg3)

caller calls callee

1. caller pushes arguments to stack
2. caller stores current PC into $ra, jumps to callee jal func
3. callee pops argument from stack
4. callee performs function
5. callee pushes return value onto stack
6. callee jumps to address stored in $ra
7. caller pops return value from stack
8. caller continues on its merry way

```
addi $t3, $zero, 5     # t3 has value 5
addi $sp, $sp, -4      # leave space for arg1 value
sw $t3, 0($sp)         # store t3 in location of sp
lw $t2, 0($sp)         # pop arg3 first
addi $sp, $sp, 4
addi $sp, $sp, -4
sw $t9, 0($sp)
jr $ra
```