Lecture 1

- midterm 2024-10-16 or October 16
- analog: signal changing in time over continuous values
- Digital: discrete signals (fixed values representing things)
  - Deg: binary (2 possible values) "0" or "1"
    - ground → "GND" or "Power Supply Voltage"
    - "Low" or "High"

Representing in Binary

- decimal system: base 10 → 0, 1, 2, ... 8, 9
- binary system: base 2 → 0, 1
- hexadecimal system: base 16 → 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Eg: $8547 = 8\times10^3 + 5\times10^2 + 4\times10^1 + 7\times10^0$

Binary: 0 or 1

$$V(B) = b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \cdots + b_0 2^0$$

Eg: $(1101)_2$ (MSB = 1, LSB = 1; binary, n=4)

$$\begin{aligned} &1\times2^3 && 8\\ &+1\times2^2 && 4\\ &+0\times2^1 && \\ &+1\times2^0 && +1\\ \hline &13 \end{aligned}$$

$(1101)_2 \rightarrow (13)_{10}$

bit	bit	#
0	0	0
0	1	1
1	0	2
1	1	3

MSB - most significant bit

LSB - least significant bit

- 8 bits = 1 byte

Lecture 2

Sep 5, 2024

- going from decimal ⟶ binary, other systems

Decimal to Binary

- divide by 2 and check if remainder is 1 or 0

÷2 ; remainders; LSB → MSB

Eg: $(9)_{10}$

	remainder	
$9 \div 2 = 4.5$	1	LSB
$4 \div 2 = 2.0$	0	
$2 \div 2 = 1.0$	0	
$1 \div 2 = 0.5$	1	MSB
$0 \div 2 = 0.0$		

$(9)_{10} \rightarrow (1001)_2$

Hexadecimal

- 4-bit binary segments
- hex reduces the size need to convey info

Binary	Hex	Decimal
0000	0	0
0001	1	
0010	2	⋮
0011	3	
⋮	⋮	
1001	9	9
1010	A	10
1011	B	11
⋮	⋮	⋮
1111	F	15

Eg: $(0001\ 0010\ 0011\ 0100)_2 \Longrightarrow (1234)_{16}$

1 2 3 4

MSB ⟶ LSB

- think about if binary represents
  - ↳ numbers
  - ↳ instructions
  - ↳ ASCII code
- currently missing: negatives, floats, 2.34×10^3

Hex to Decimal

$(241)_{16} = 1\times16^0 + 4\times16^1 + 2\times16^2$

MSB ↑ LSB ↑

$= (577)_{10}$

2 4 1
2 1 0
powers of 16

$\therefore (241)_{16} = (577)_{10}$

Hex to Binary

$(241)_{16} \rightarrow (0010\ \ 0100\ \ 0001)_2$

$(241)_{16} = (001001000001)_2$

Decimal to Hex [÷16, remainder]

$(577)_{16}$

$577 \div 16 = 36.0625$ (×16 = 1)

$36 \div 16 = 2.25$ (×16 = 4)

$2 \div 16 = 0.125$ (×16 = 2)

remainders: 1, 4, 2 (LSB → MSB)

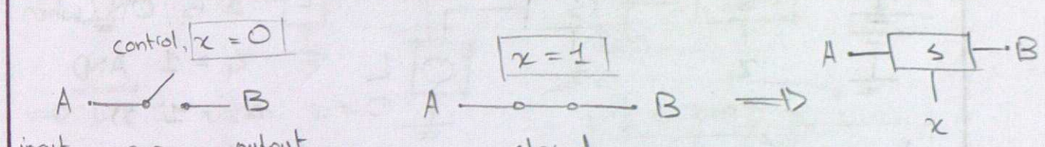
$\therefore (577)_{16} = (241)_{16}$

Look at
- Binary
- Hexadecimal

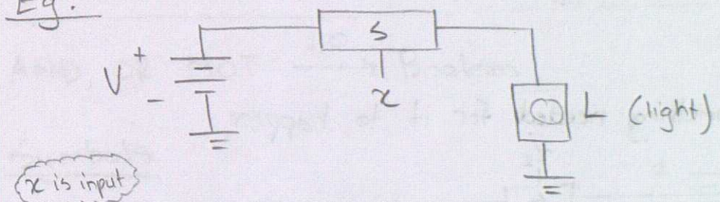
Logic Circuits

- design is for binary (1 and 0)
- on, off; high, low → think of "switches"
- Boolean logic

switch — simplifying switches to this →



Eg:



x is input

input	output	
x = 0	L	off = 0 } states
x = 1	L	on = 1 } states

- Logic Expression

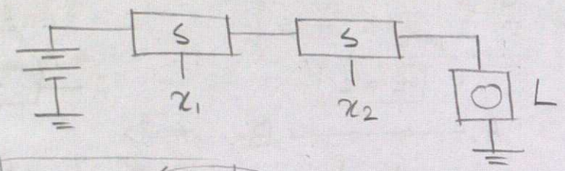
$L(x) = x$

Logic Circuits

- logic expressions:

→ **AND**

- 2 conditions need to be true for it to happen

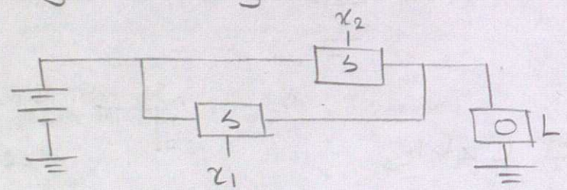


"L is ON when
$x_1 = 1$ AND
$x_2 = 1$ "
"L is OFF otherwise"

$L = x_1 \cdot x_2$ ← AND operator

→ **OR**

- something or something needed for it to happen

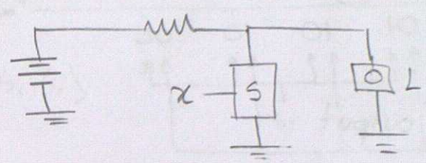


⇒ L is ON when $x_1 = 1$ OR $x_2 = 1$
⇒ L is OFF when $x_1 = 0$ and $x_2 = 0$

$L = x_1 + x_2$ ← OR operator

→ **NOT**

- complement of a signal or inverted signal
- one input and one output
- switch is path to ground ‖ to light

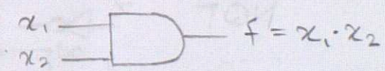


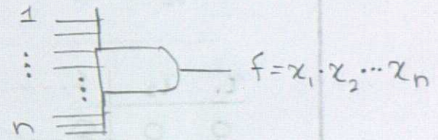
NOT operator
$L = \overline{x}$
typing code
$L = !x$
$L = \text{NOT } x$
$L = \sim x$

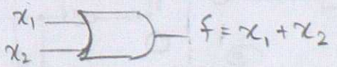
- ⇒ L = 1 when x = 0
  ⇒ L = 0 when x = 1

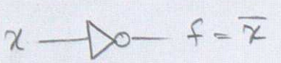
- AND, OR, NOT $\xrightarrow{1,0}$ Boolean

Symbols

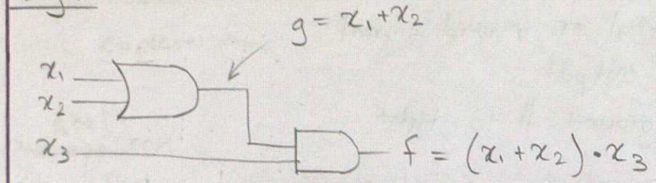
- AND 



- OR 

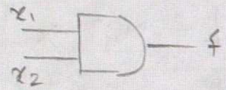
- NOT 

Eg:

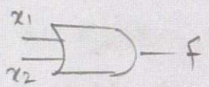


☆ → **Truth Table**

- consider all possible inputs → output

AND 

x_1	x_2	f
0	0	0
0	1	0
1	0	0
1	1	1

OR 

x_1	x_2	f
0	0	0
0	1	1
1	0	1
1	1	1

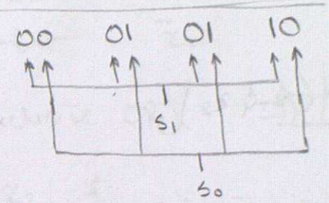
NOT 

x	f
0	1
1	0

Eg: Adder

- 1-bit adder

b_0	0	0	1	1
b_1	0	1	0	1
"sum"	00	01	01	10
$\{S_1, S_0\}$				

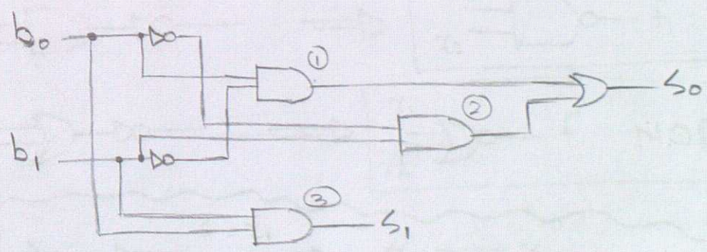


b_1	b_0	sum — $S_1 S_0$
0	0	00
0	1	01
1	0	01
1	1	10

- If b_1 AND b_0 are 1, then $S_1 = 1$
- If $b_1 b_0 = 01$ or $b_1 b_0 = 10$, then $S_0 = 1$
- Expression.: $S_0 = \overline{b_1} \cdot b_0$ ① $+ b_1 \overline{b_0}$ ②

$S_1 = b_1 b_0$ ③

- make ckt for HW

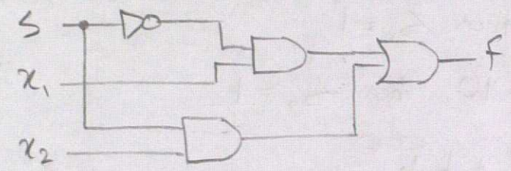


Sep 6, 2024

Verilog

$f = \bar{s}x_1 + sx_2$

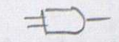
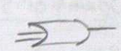
```
module example3 (x1, x2, s, f)
    input x1, x2, s;
    output f;
    assign f = (~s & x1) | (s & x2);
endmodule
```



Week 2: Lec 1 — Sep 10, 2024

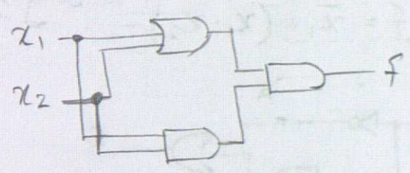
- logic gates, inverted logic, timing diagram, boolean algebra

Logic Gates

- AND  $x_1 \cdot x_2$
- OR  $x_1 + x_2$
- Not  $\bar{x}_1$
- XOR (exclusive OR)  $f = x_1 \oplus x_2$

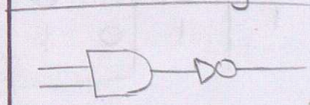
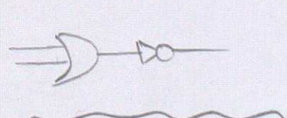
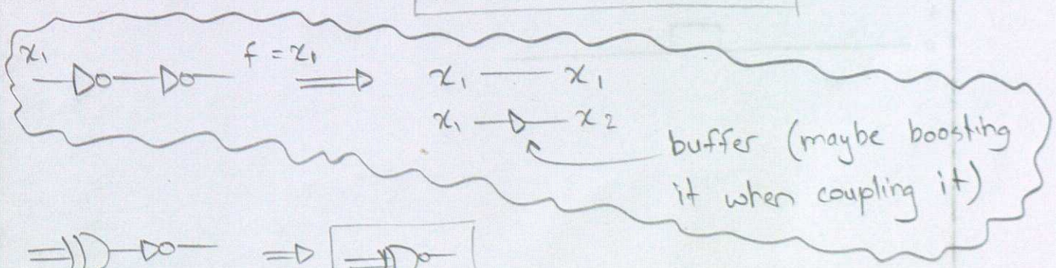
x_1	x_2	f
0	0	0
0	1	1
1	0	1
1	1	0

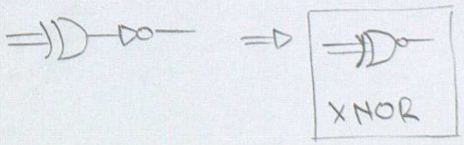
main difference from OR regular



implementing XOR using AND and OR gates

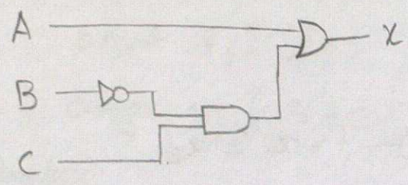
Inverted Logic

-  NAND
-  NOR
- 

- 

Exercise

$x = A + \overline{B} \cdot C$ (no inverted logic) (only A, B, and C)

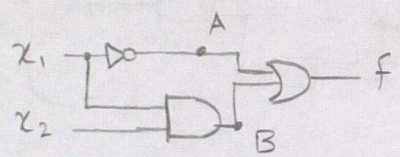


Order of Operations

$x = A + (\overline{B} \cdot C)$

Timing Diagrams

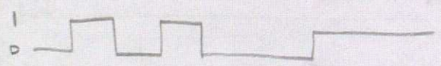
Eg: $f = \overline{x}_1 + (x_1 \cdot x_2)$ only have x_1 and x_2



helper box

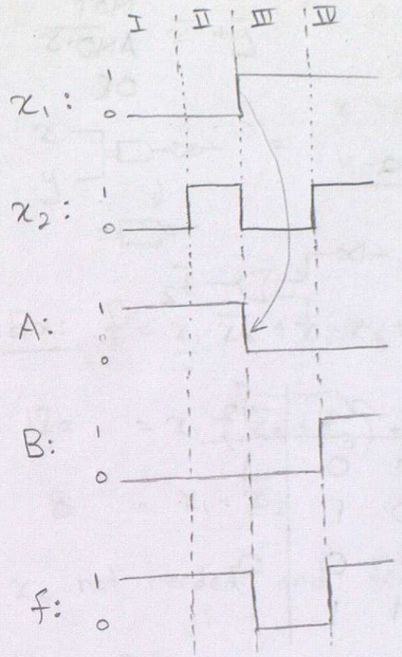
x_1	x_2	f	A	B
0	0	1	1	0
0	1	1	1	0
1	0	0	0	0
1	1	1	0	1

x_1



x_2 → time

- Timing diagram → no propagation delay for gates (input - output)

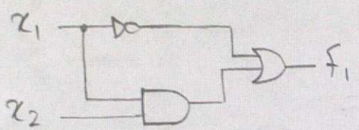


Boolean Algebra

$f_1 = \overline{x_1} + x_1 \cdot x_2$ $f_2 = \overline{x_1} + x_2$

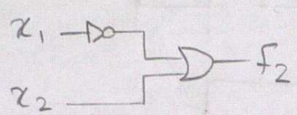
Order of Opp
NOT
AND
OR

f_1



x_1	x_2	f_1
0	0	1
0	1	1
1	0	0
1	1	1

f_2



x_1	x_2	f_2
0	0	1
0	1	1
1	0	0
1	1	1

- We can see f_1 and f_2 have same outputs for given inputs ... f_1 is equal to f_2

Axioms of Boolean Algebra

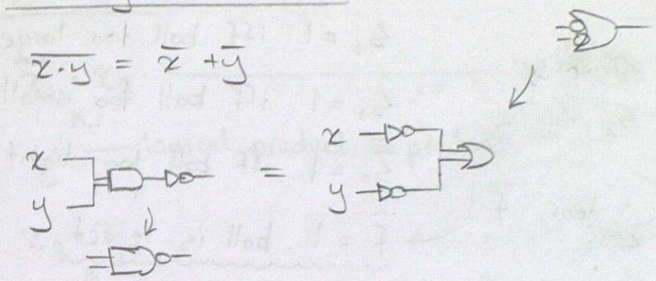
- Duality = flip sign and value { $0 \cdot 0 = 0$, $1+1=1$

12b. $x + y \cdot z = (x+y) \cdot (x+z)$

DeMorgan's Theorem: $\overline{x \cdot y} = \overline{x} + \overline{y}$ $\overline{x+y} = \overline{x} \cdot \overline{y}$

De Morgan's Theorem

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



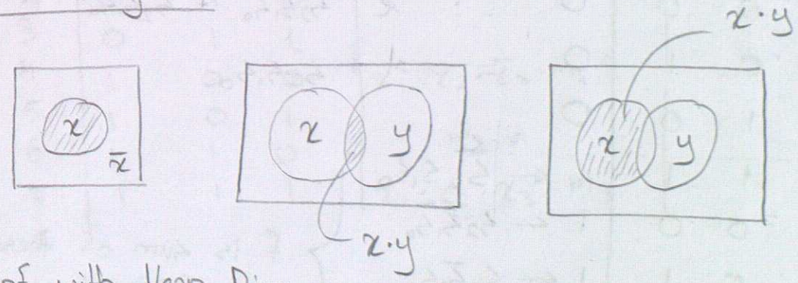
I Ex: $f = \underline{x_1} \cdot \overline{x_3} + x_1 \cdot x_3 + \overline{x_2} \cdot \overline{x_3} + \overline{x_2} \cdot x_3$

12a $= x_1 \cdot (\overline{x_3} + x_3) + \overline{x_2} \cdot (\overline{x_3} + x_3)$

8 $= x_1 + \overline{x_2}$

- x_3 not needed and fewer gates

Venn Diagrams

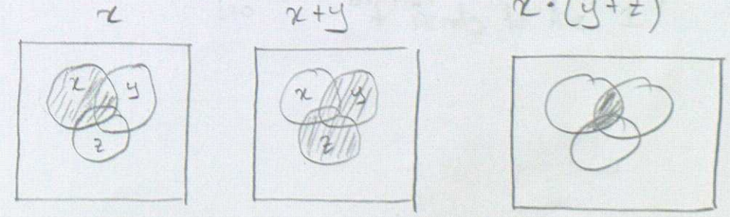


Proof with Venn Digg

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

Do RHS

L.H.S.



Designing

Sensors S_2 S_1 S_0

$S_2 = 1$ iff ball too large
$S_1 = 1$ iff ball too small
$S_0 = 1$ iff ball too light

. . . . → door [f] → $f = 1$ ball is rejcct

- $f=1$ if ball is too large or — cond 1
  if ball is too small and too light — cond 2

$$f = S_2 + S_1 S_0$$

Truth Table

Show the 2 f are the same

S_2	S_1	S_0	f	
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	1	← $\bar{S}_2 S_1 S_0$
1	0	0	1	← $S_2 \bar{S}_1 \bar{S}_0$
1	0	1	1	← $S_2 \bar{S}_1 S_0$
1	1	0	1	← $S_2 S_1 \bar{S}_0$
1	1	1	1	← $S_2 S_1 S_0$

f is sum of these

$$f = \bar{S}_2 S_1 S_0 + S_2 \bar{S}_1 \bar{S}_0 + S_2 \bar{S}_1 S_0 + S_2 S_1 \bar{S}_0 + S_2 S_1 S_0$$

Notation

$\bar{x}_1 \cdot x_2$ → $\bar{x} x_2$ product term

↑ logical product ≡ product

$x_3 + x_4$

↑ logical sum ≡ sum

ex: $xy + ab$ — sum of products

Precedence : NOT → AND → OR

Sum of Products (SOP)

what does it take to get 1?

row #	x_1	x_2	x_3	minterm	f
0	0	0	0	$\bar{x}_1 \cdot \bar{x}_2 \cdot \bar{x}_3 = m_0$	
1	0	0	1	$\bar{x}_1 \cdot \bar{x}_2 \cdot x_3 = m_2$	
2	0	1	0	⋮	
3	0	1	1		
4	1	0	0	$x_1 \cdot \bar{x}_2 \cdot \bar{x}_3 = m_4$	
5	1	0	1	⋮	
6	1	1	0		
7	1	1	1	$x_1 \cdot x_2 \cdot x_3 = m_7$	

canonical SOP:

$$f(x_1, x_2, x_3) = \sum_{i=0}^{n} m_i$$ where m_i is 1 when f needs to be 1

Sep 13, 2024

I

SOP Ex: function given as follows

row #	x_1	x_2	f	minterms
m_0	0	0	1	$\bar{x}_1 \cdot \bar{x}_2$
m_1	0	1	1	$\bar{x}_1 \cdot x_2$
m_2	1	0	0	$x_1 \cdot \bar{x}_2$
m_3	1	1	1	$x_1 \cdot x_2$

SOP: $f = m_0 \cdot f_0 + m_1 \cdot f_1 + m_2 f_2 + m_3 f_3$

$= m_0 + m_1 + m_2 \cdot 0 + m_3$

$= \bar{x}_1 \cdot \bar{x}_2 + \bar{x}_1 \cdot x_2 + x_1 \cdot x_2$ ← circuit

⋮ textbook simplify

$= x_2 + \bar{x}_1$ ← circuit (note: smaller)

HW simplify

$\bar{x}_1 \cdot \bar{x}_2 + \bar{x}_1 \cdot x_2 + x_1 \cdot x_2 = \bar{x}_1 \cdot (\bar{x}_2 + x_2) + x_1 \cdot x_2$

$= \bar{x}_1 \cdot \bar{x}_2 + x_2 \cdot (\bar{x}_1 + x_1)$

$= \bar{x}_1 \cdot \bar{x}_2 + x_2$

$= \bar{x}_1 + x_1 \cdot x_2$

$x_1 x_2$

00	01	11	10
1	1	0	1

$\bar{x}_1 +$ x_2

Product of Sums (PoS)

duality: $0 \leftrightarrow 1$, AND $\leftrightarrow$ OR

row #	x	y	f	maxterm
0	0	0	0	$x + y$
1	0	1	1	$x + \bar{y}$
2	1	0	1	$\bar{x} + y$
3	1	1	1	$\bar{x} + \bar{y}$

what does it take for the sum to be 0?

HW same truth table, but do SOP

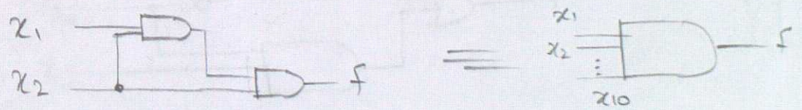
Canonical PoS: $f = x + y$

General outcome: $f = \prod M_i$ ← these are those that produce each needed 0

Cost of a Circuit

cost of a circuit = total number of gates + total number of inputs to all gates

II **Ex:**

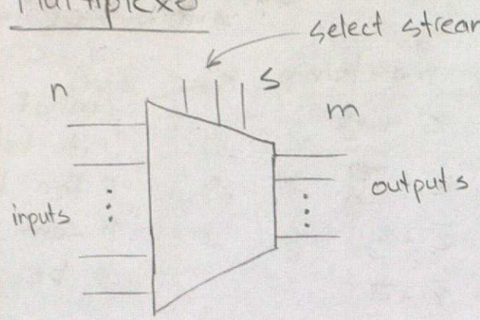


$C = 2 + 4 = 6$ $\qquad$ $C = 1 + 10 = 11$

Cost = (total # gates) + (total # inputs to all gates)

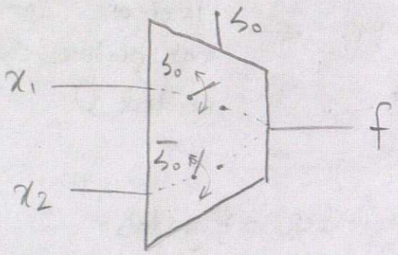
Sep 13, 2023

Multiplexer



s will select which input goes to output

Ex: 2-1 multiplexer



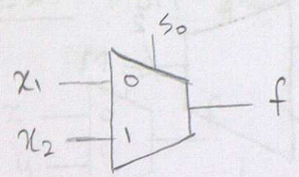
- if s_0 goes to x_1 then x_2 will get $\bar{s}_0$

Sep 17, 2024

Multiplexer (mux)

- if $s_0 = 0 \Rightarrow f = x_1$
- if $s_0 = 1 \Rightarrow f = x_2$

1-bit 2 to 1 multiplexer



s_0	x_1	x_2	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Sum of products

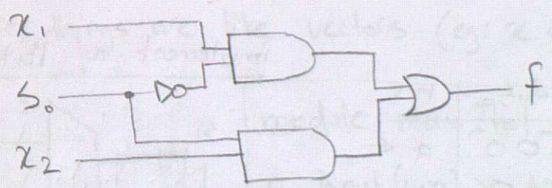
$= \bar{s}_0 x_1 \bar{x}_2 + \bar{s}_0 x_1 x_2 + s_0 \bar{x}_1 x_2 + s_0 x_1 x_2$

$= \bar{s}_0 x_1 (\bar{x}_2 + x_2) + s_0 x_2 (\bar{x}_1 + x_1)$

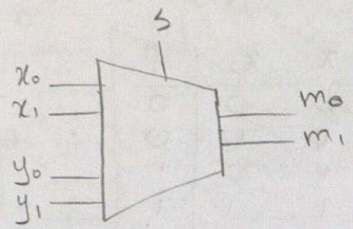
$= \bar{s}_0 x_1 + s_0 x_2$

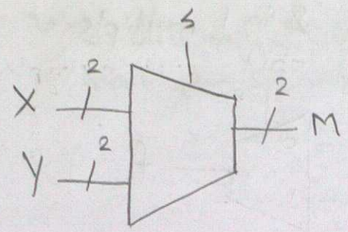
Circuit: $f = \bar{s}_0 x_1 + s_0 x_2$ ← Truth Table

s_0	f
0	x_1
1	x_2



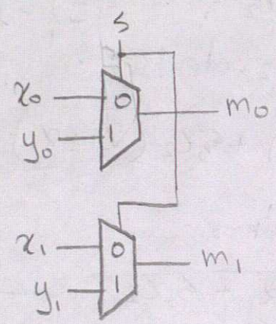
2 bit 2-1 Multiplexer



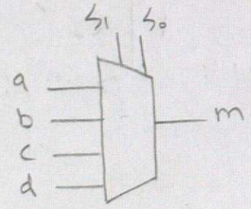


s	$m_0 m_1$
0	$x_0 x_1$
1	$y_0 y_1$

Implement using 1bit 2tol mux

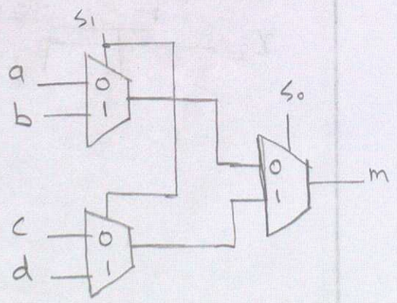


1 bit 4-1 Multiplexer



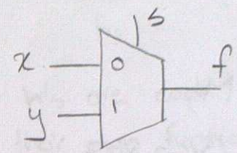
$s_1 s_0$	m
00	a ←
→10	b
01	c ←
→11	d

implement w/ 1bit 2tol mux



Hardware Description Language (HDL)

Ex: 1-bit 2tol multiplexer

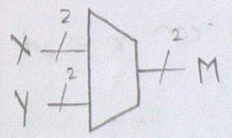


$f = \bar{s}x + sy$

```
module mux2to1(x, y, s, f);
    input x, y, s;
    output f;
    assign f = (~s & x) | (s & y);
endmodule
```

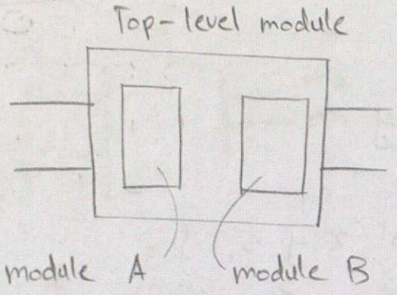
Ex: 2 bit 2tol multiplexer

here, things are like vectors (eg: x will have x_0 and x_1)




```
module mux2to1_2bit(X, Y, S, M);
    input [1:0] X, Y;
    input S;
    output [1:0] M;
    assign M[0] = (~S & X[0]) | (S & Y[0]);
    assign M[1] = (~S & X[1]) | (S & Y[1]);
endmodule
```

Verilog Heirarchy



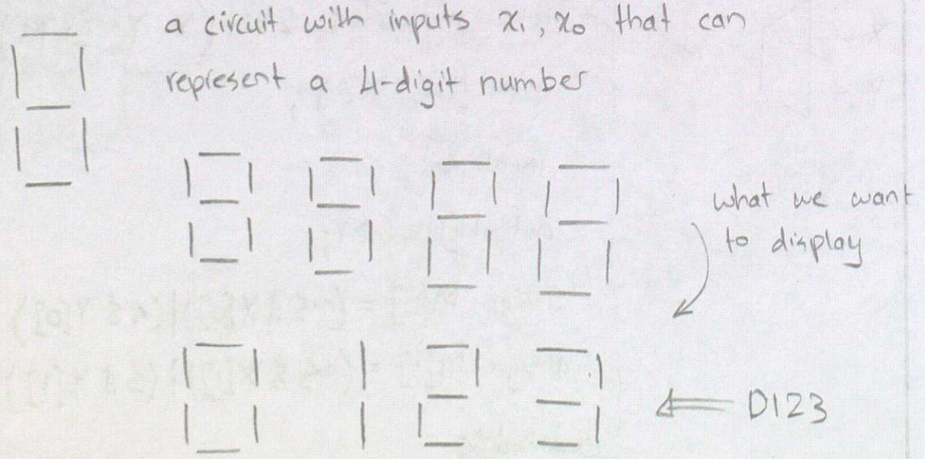

```
module mux2to1_2bit(X,Y,S,M);
    input [1:0] X,Y;
    input S;
    output [1:0] M;
    mux2to1 U1(X[0],Y[0],S,M[0]);
    mux2to1 U2(X[1],Y[1],S,M[1]);
endmodule
```

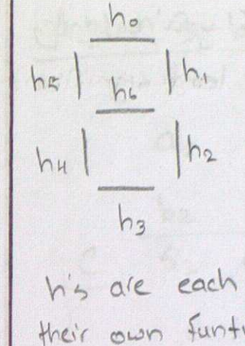
```
module mux2to1(x,y,s,m)
    input x,y,s;
    output m;
    assign M = (~s & x)|(s & y);
endmodule
```

↑ this goes after this main module

Ex: 7-segment display

a circuit with inputs x_1, x_0 that can represent a 4-digit number





h's are each their own funtion

display	x_1	x_0	h_0	h_1	h_2	h_3	h_4	h_5	h_6
0	0	0	1	1	1	1	1	1	0
1	0	1	0	1	1	0	0	0	0
2	1	0	1	1	0	1	1	0	1
3	1	1	1	1	1	1	0	0	1

we have to develop a funct. for each

$h_0 = \overline{x_1}\,\overline{x_0} + x_1\overline{x_0} + x_1 x_0 = x_1 + \overline{x_0}$ ——— SOP

$h_1 = 1$

$h_2 = \overline{x_1} + x_0$

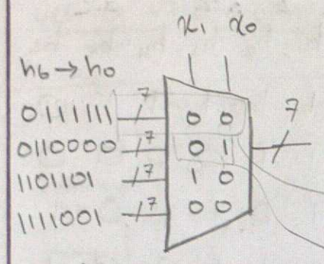
$h_3 = h_0 = x_1 + \overline{x_0}$

$h_4 = \overline{x_0}$

$h_5 = \overline{x_1}\,\overline{x_0}$

$h_6 = x_1$

Week 3: Lecture 3 — Sep 20, 2024



- you can designate if you're going most significant to least significant or vise versa

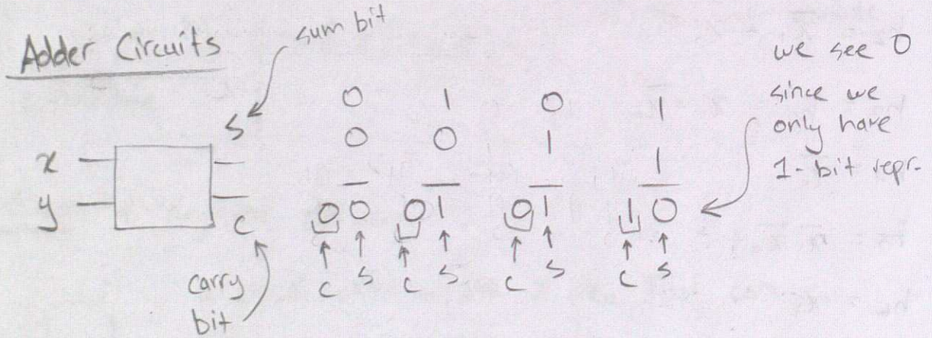
register input 00

register input 01

you preprogram what you put in "field programmable gate array"

Look-up Table Concept ~ FPGA

Adder Circuits

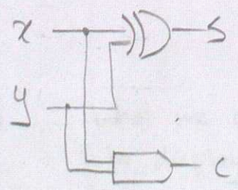


half adder (2-bit)

x	y	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$s = x \oplus y$

$c = x \cdot y$



Adder 3-bit

- assuming a_0 and b_0 are 1 and 1, the next bit over must add a_1 and b_1 but also take in a carry
- full adder since I can add numbers and also

$$\begin{array}{cccc} & a_2 & a_1 & a_0 \\ & b_2 & b_1 & b_0 \\ \hline c & s_2 & s_1 & s_0 \end{array}$$

Ex

$$\begin{array}{cccc} & c_2 & c_1 & c_0 \\ & 1 & 0 & \\ & 0 & 1 & 0 \\ \text{sum} & 1 & 1 & 1 \\ \hline 1, & 0 & 0 & 1 \end{array}$$

internal carry

carry outside

We give names to our carry's

3-input XOR

c_i	b_i	a_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$f = x \oplus y \oplus z$

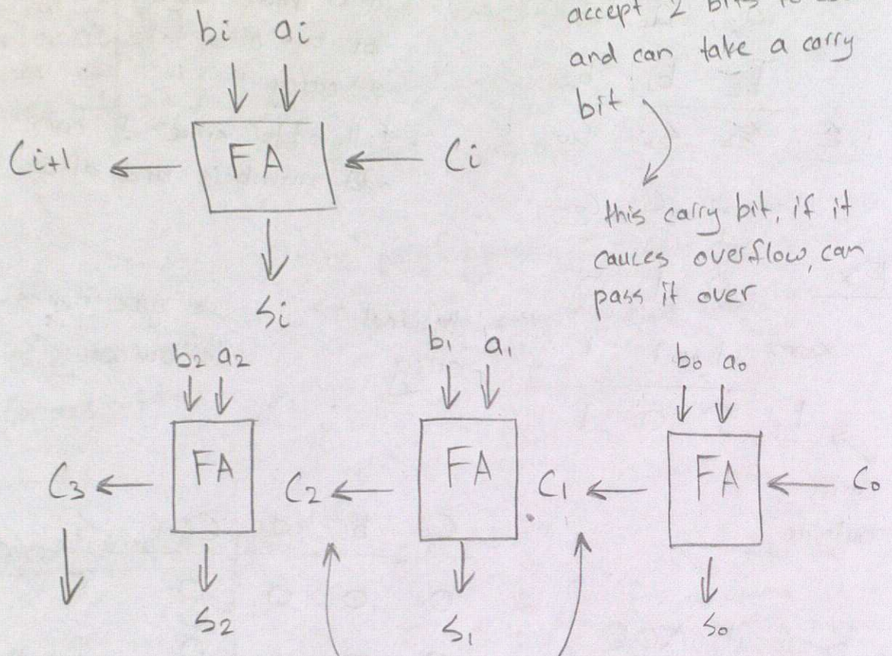
$= x \oplus (y \oplus z)$

$= (x \oplus y) \oplus z$

$s_i = a_i \oplus b_i \oplus c_i$

$c_{i+1} = a_i b_i +$

Define Full Adder



each full adder can accept 2 bits to add and can take a carry bit

this carry bit, if it causes overflow, can pass it over

define as wire

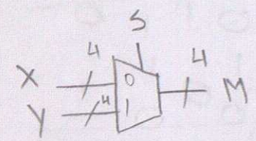
programming view: these are internal things ... a variable that changes over time, as the thing is working, these will change: a wire that can change as things are changing

NAND, NOR Logic Network

- can have (AND, OR) or (NAND, NOR) setups
- at transistor level, NAND has fewer transistors than AND (using CMOS technology)

Lab 1

part 2



4 bit 2-to-1 mux

assign m = (~s & x) | (s & y); (arrays)

SW_9 → "s" input

SW_{3-0} → "X" input

SW_{7-4} → "Y" input

X: [x0, x1, x2, x3]

Y: [y0, y1, y2, y3]

M: [m0, m1, m2, m3]

if s=0, M=X

s=1, M=Y

s	M
0	X
1	Y

display value "s" → $LEDR_9$

connect output "M" → $LEDR_{3-0}$

connect unused LEDR's to "0"

7.)

S_0	S_1	M
0	0	u
0	1	v
1	0	w
1	1	w

S_0	S_1	m_0	m_1
0	0	u	
0	1	v	
1	0	w	
1	1	w	

Part 3

S_1	S_0	U_1	U_0	V_1	V_0	W_1	W_0	m_1	m_0	
0	0	0	0	1	0	0	1	0	0	✓
0	1	0	1	0	1	0	0	0	1	✓
1	0	1	0	1	1	1	1	1	1	✓
1	1	0	0	0	0	1	1	1	1	✓
PINS	48		54		32		10		10	LED

Par 4

C_1 C_0	Char	h_0	h_1	h_2	h_3	h_4	h_5	h_6
00	d	0	1	1	1	1	0	1
01	E	1	0	0	1	1	1	1
10	1	0	1	1	0	0	0	0
11	" "	0	0	0	0	0	0	0

$h_0 = \bar{C}_1 C_0$

$h_1 = \bar{C}_1 \bar{C}_0 + C_1 \bar{C}_0$

$h_2 = \bar{C}_1 \bar{C}_0 + C_1 \bar{C}_0$

$h_3 = \bar{C}_1 C_0 + \bar{C}_1 C_0$

$h_4 = \bar{C}_1 \bar{C}_0 + \bar{C}_1 C_0$

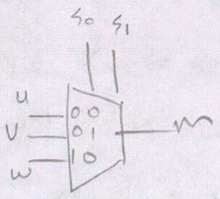
$h_5 = \bar{C}_1 C_0$

$h_6 = \bar{C}_1 C_0 + \bar{C}_1 C_0$

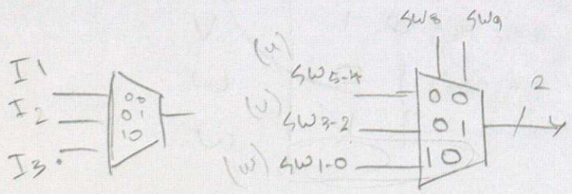
$C_1 C_2$

Part 5

S_1 S_0	m
00	u
01	v
10	w



Part 5



SW_9 SW_8	Characters
00	d E 1
01	E 1 d
10	1 d E

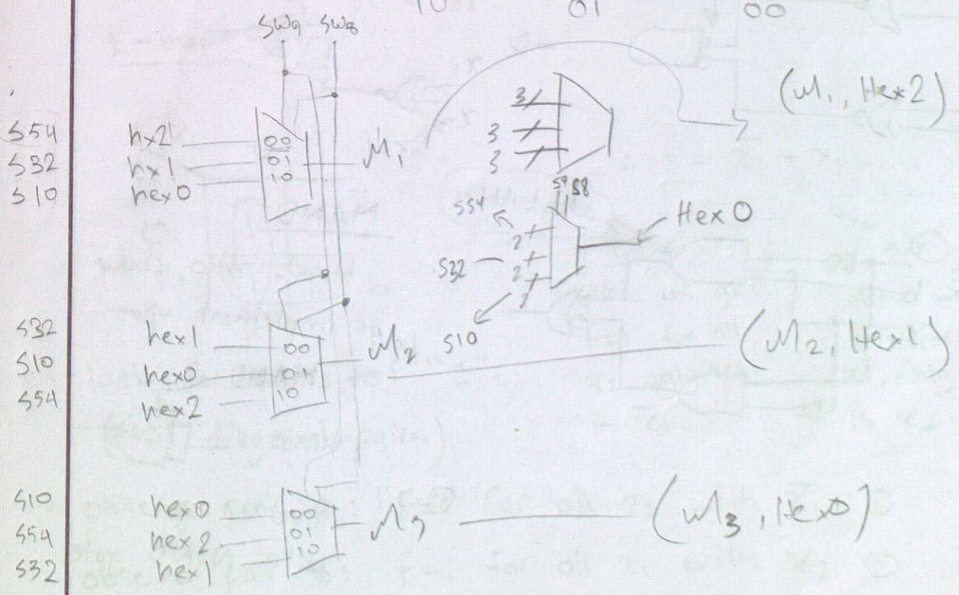
Hex2 Hex1 Hex0

SW_9	SW_8	Characters			S_{54} S_{32} S_{10}
0	0	00	01	10	
0	1	01	10	00	Hex2 Hex1 Hex0
1	0	10	00	01	↑ ↑ ↑
		HEX2	HEX1	HEX0	S_{54} S_{32} S_{10}

0	0	Hex2	Hex1	Hex0
0	1	Hex1	Hex0	Hex2
1	0	Hex0	Hex2	Hex1
		Disp2	Disp1	Disp0

HEX2 HEX1 HEX0

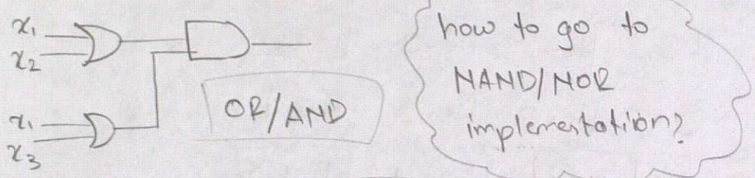
10 01 00



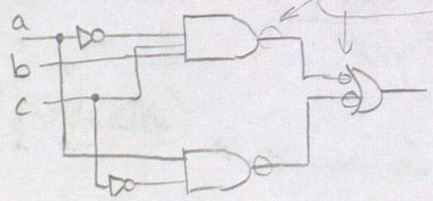
NAND, NOR

NOR $\overline{x+y} = \bar{x} \cdot \bar{y}$ **NAND** $\overline{x\,y} = \bar{x} + \bar{y}$

Ex $f = (x_1+x_2)(x_1+x_3)$ ← POS



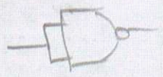
Ex



NAND

H.W. also, show or implement ⊳ as NAND

hint:



inverter with NAND gate

Karnaugh Maps (Minimize/Optimize Logic Ckt/func)

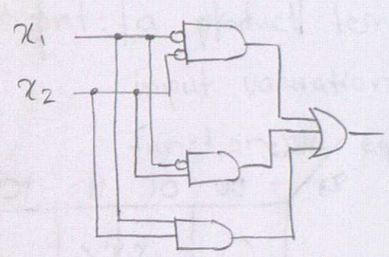
Ex:

x_1	x_2	f
0	0	1
0	1	1
1	0	0
1	1	1

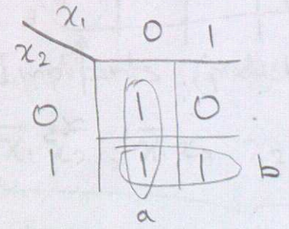
$f = \bar{x}_1\bar{x}_2 + \bar{x}_1 x_2 + x_1 x_2$ ①

$= \bar{x}_1(\bar{x}_2 + x_2) + x_1 x_2$

$= \bar{x}_1 + x_1 x_2$

① 

K-maps



$\therefore f = \bar{x}_1 + x_2$

① since we get $f=1$ for all x_2, only req: is $\bar{x}_1$

⑪ since we get $f=1$ for all x_1, only req. is x_2.

- look for pairs of "1"s (NOT diagonal pairs)
- observe pair a: $f=1$ for all x_2 with $\bar{x}_1$ ①
- observe pair b: $f=1$ for all x_1 with x_2 ⑪
- Karnaugh map uses combining (rule 14a): $x \cdot y + x \cdot \bar{y} = x$

K-maps 3 variables

x_1	x_2	x_3	f
0	0	0	m_0
⋮	⋮	⋮	⋮
1	1	1	m_7

- can think of K-map as a cylinder (wrapping)

you only want to change 1 bit at a time

x_3 \ x_1x_2	00	01	11	00
0	m_0	m_2	m_6	m_4
1	m_1	m_3	m_7	m_5

Ex:

x_1	x_2	x_3	f
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

x_3 \ x_1x_2	00	01	11	10
0	1	0	0	0
1	1	1	1	1

a: $\overline{x}_1\overline{x}_2$ (covers the two 1s in the $x_3 = 0$ row)

b: x_3 (covers the four 1s in the $x_3 = 1$ row)

these two together make f=1

$$f = \overline{x}_1\overline{x}_2 + x_3$$

Terminology

- Literal: a variable in a product term and the variable is (or can be) uncomplemented or complemented

e.g: $x_1\overline{x}_2x_3$ ← 3 literals

(x_1: a literal; $\overline{x}_2$: a literal; x_3: a literal)

- Implicant: a product term that indicates the input valuation for which a given function is equal to 1.

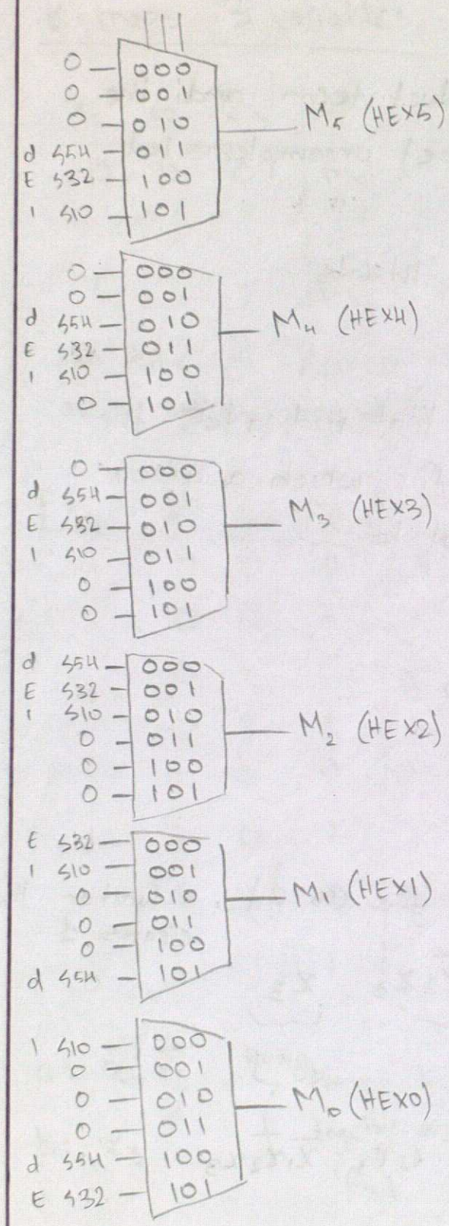
Ex:

x_3 \ x_1x_2	00	01	11	00
0	1	0	0	0
1	1	1	1	1

Implicants (product values that give us 1): a function that creates 1

$\overline{x}_1\overline{x}_2$, $\overline{x}_1x_3$, x_2x_3, x_1x_3, $\overline{x}_2x_3$ (pairs), x_3 (group)

$\overline{x}_1\overline{x}_2\overline{x}_3$, $\overline{x}_1\overline{x}_2x_3$, $\overline{x}_1x_2x_3$, $x_1x_2x_3$, $\overline{x}_1x_2\overline{x}_3$



base case

d	E	I
SW4	SW2	SW0

Prime Implicant (P.I.)

1. can't be combined into another implicant that has fewer literals.
2. not fully contained within another implicant.
3. tend to be the largest groups of 1's.

Cover

- a collection of implicants that account for all valuations for which a given function is equal to 1.
  - ↳ all the implicants that give you all the 1's.

Lowest Cost Implementation

- the cover of the prime implicants

Essential Prime Implicant

- those that include a 1 that is not in any other prime implicant.
- must be included in any cover of the function

Exam Tips

$f = \sum m(1,4,5,6)$ ← m_1, m_4, m_5, and m_6 have to each have /equal to 1

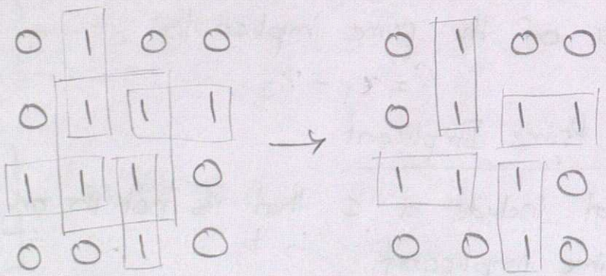
Summary: After making K-maps...

1) Find all the prime implicants
2) Identify essential prime implicants
   ↳ if all essential PI for the cover → done
3) Choose non-essentials to obtain a min-cost cover.

Mincost

- textbook: (inputs + all gates)
- general: (inputs + all gates ignoring inverters)

Ex:



Don't Cares

- 2 switches designed that they can't both be on at the same time

Switch (x_1, x_2)	f
1, 0	1
0, 1	1
0, 0	0

x_2 \ x_1	0	1
0	0	1
1	1	d

put "d" since can never happen

(minterms): $f = x_1\overline{x}_2 + \overline{x}_1 x_2$ ← assuming d = 0

- since the d situation will never happen, we can assume d is a zero or d is a 1
- so assume d = 1

x_2 \ x_1	0	1
0	0	1
1	1	d=1

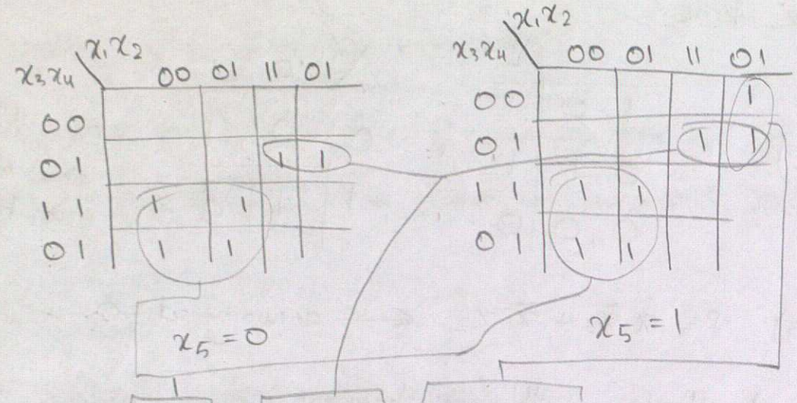
$f = x_1 + x_2$ ← assuming d = 1

- you can assume some d's to be 1's and some d's to be 0's to make ckt simpler
  ↳ pick d=1 to optimize
  ↳ pick d=0 to help ignore

Week 4: Lecture 3 — Sep 27, 2024

5-variable k-map

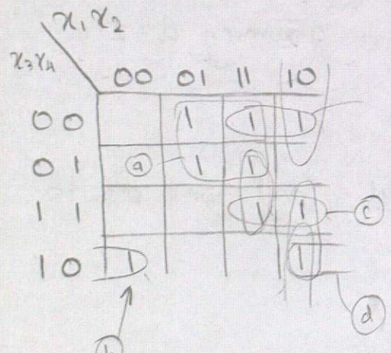
do one with $x_5 = 0$ and $x_5 = 1$



$f_1 = \bar{x}_1 x_3 + x_1 \bar{x}_3 x_4 + x_1 \bar{x}_2 \bar{x}_3 x_5$

4-Variable Example

Find all P.I. = 7



Essential P.I.:

ⓐ $x_2 \bar{x}_3$

ⓑ $\bar{x}_2 x_3 \bar{x}_4$

ⓒ $x_1 x_3 x_4$

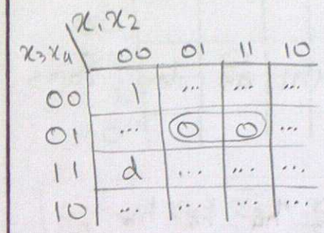
ⓓ $x_1 \bar{x}_2 \bar{x}_4$

$f = x_2 \bar{x}_3 + \bar{x}_2 x_3 \bar{x}_4 + x_1 x_3 x_4 + x_1 \bar{x}_2 \bar{x}_4$

ⓑ essential since no other grouping accounts for it

ⓓ can be replaced by $x_1 \bar{x}_3 \bar{x}_4$

K-Map POS



looking for 0's

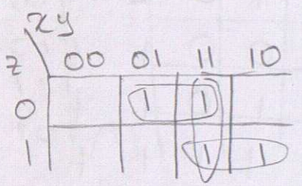
$f = (\ +\ )(\ +\ +\ ) \cdots$

$\bar{x}_2 + x_3 \bar{x}_4$

$\bar{x}_2$: we bar since we want 0

$x_3 \bar{x}_4$: trying to make 0

Ex: $f = xz + y\bar{z} + xy$ is this min cost sop expression?



current expression does 3 groups but lowest cost is

$f = y\bar{z} + xz$

boolean algebra → consensus: $x \cdot y + \bar{x} \cdot z + y \cdot z = xy + \bar{x} \cdot z$

$f = xz + xy + y\bar{z}$

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

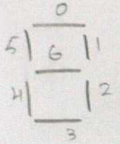
↳ introduce $(z + \bar{z}) = 1$ with a term to help

Lab 2 — Sep 28, 2024

Part 1

SW_{74} : HEX1 — display 0 to 9

SW_{30} : HEX0 — valuations 1010 to 1111 as don't cares



SW_{7654} or SW_{3210}	character	h_0	h_1	h_2	h_3	h_4	h_5	h_6
0 0 0 0	0	1	1	1	1	1	1	0
0 0 0 1	1	0	1	1	0	0	0	0
0 0 1 0	2	1	1	0	1	1	0	1
0 0 1 1	3	1	1	1	1	0	0	1
0 1 0 0	4	0	1	1	0	0	1	1
0 1 0 1	5	1	0	1	1	0	1	1
0 1 1 0	6	1	0	1	1	1	1	1
0 1 1 1	7	1	1	1	0	0	0	0
1 0 0 0	8	1	1	1	1	1	1	1
1 0 0 1	9	1	1	1	1	0	1	1
$\overline{x_4}$	$\overline{x_3}$	$\overline{x_2}$	$\overline{x_1}$					

h_0

x_4x_3 \ x_2x_1	00	01	11	10
00	1c	0	1d	ϕ
01	0	1	1	1e
11	0	0	0	0
10	1	1	0	0

a, b

SOP

$$f_0 = x_1x_3\overline{x_4} + \overline{x_2}\,\overline{x_3}x_4 + \overline{x_1}\,\overline{x_2}\,\overline{x_3}\,\overline{x_4} + x_1x_2\overline{x_3}\,\overline{x_4} + \overline{x_1}x_2x_3\overline{x_4} + \overline{x_1}x_2\overline{x_3}\,\overline{x_4}$$

h_1

x_4x_3 \ x_2x_1	00	01	11	10
00				
01		0		0
11				
10				

POS

$$f_1 = (\overline{x_1} + x_2 + \overline{x_3} + x_4)\cdot(x_1 + \overline{x_2} + \overline{x_3} + x_4)$$

h_2

$$f_2 = x_1 + \overline{x_2} + x_3 + x_4$$

h_3

$$f_3 = (\overline{x_1} + x_2 + x_3 + x_4)(x_1 + x_2 + \overline{x_3} + x_4)(\overline{x_1} + \overline{x_2} + \overline{x_3} + x_4)$$

h_4

x_4x_3 \ x_2x_1	00	01	11	10
00	1a			1b
01				1
11				
10	1			

SOP

$$f_4 = \overline{x_1}x_2\,\overline{x_4} + \overline{x_1}\,\overline{x_2}\,\overline{x_3}$$

h_5

x_4x_3 \ x_2x_1	00	01	11	10
00	1	0	0	0
01	1	1	0	1
11	1	1	1	1
10	1	1	1	1

POS

$$f_5 = (x_3 + x_4 + \overline{x_1})(x_1 + \overline{x_2} + x_3 + x_4)(\overline{x_1} + \overline{x_2} + \overline{x_3} + x_4)$$

h_6

$$f_6 = (x_1 + x_2 + x_3 + x_4)(\overline{x_1} + x_2 + x_3 + x_4)(\overline{x_1} + \overline{x_2} + \overline{x_3} + x_4)$$

POS

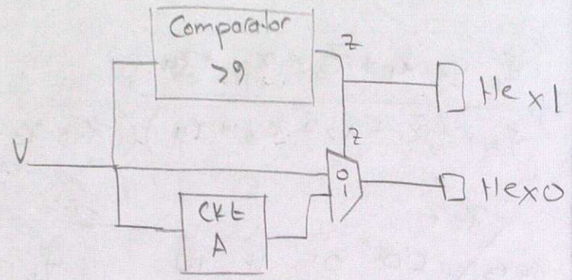
x_4x_3 \ x_2x_1	00	01	11	10
00	0	0		
01			0	
11				
10				

$$f_6 = (x_3 + x_4 + x_2)(\overline{x_1} + \overline{x_2} + \overline{x_3} + x_4)$$

Part 2

$V = V_3V_2V_1V_0 \longmapsto D = d_1d_0$

$V_3V_2V_1V_0$	d_1d_0
0000	00
0001	01
...	...
1001	09
1010	10
...	...
1111	15



z will be 0 for 0-9

z will be 1 for 10-15

$V_3V_2V_1V_0$	z
0000	0
0001	0
...	...
1001	0
1010	1
1011	1
1100	1
1101	1
1110	1
1111	1

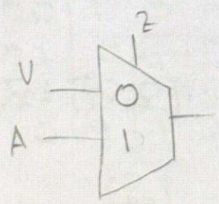
V_3V_2 \ V_1V_0	00	01	11	10
00				
01				
11	1	1	1	1
10			1	1

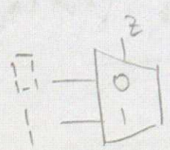
part 1

make sure to add $V_4 = 0$

SOP $z_1 = V_2V_3 + V_1\bar{V}_2V_3$

this is our select





designing ckt A:

$V_3V_2V_1V_0$	$A_3A_2A_1A_0$
1010	0000
1011	0001
1100	0010
1101	0011
1110	0100
1111	0101

$A_3 = 0$

$A_2 = V_1V_2V_3$

$A_1 = \bar{V}_1V_2V_3$

$A_0 = V_0V_2V_3 + V_0V_1V_3$

V_3V_2 \ V_1V_0	00	01	11	10
00	d	d	d	d
10				
11		1	1	
10			1	

HEX1 < z=0 : print ⌜ ⌟
z=1 : print ▯

HEX0 < z=0 : V
z=1 : A ckt

$SW_{30} \rightarrow V_{30}$

4bit 2to1 mux taken from lab1 part2

⌜ ⌟ : 0000

▯ : 0001

Part 3

XOR :

x_2 \ x_1	0	1
0		1
1	1	

$\bar{x}_2x_1 + \bar{x}_1x_2 \longrightarrow$ "^"

SW_{74} : A

SW_{30} : B

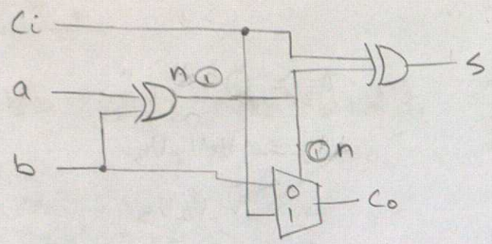
SW_0 : carry in Cin

$LEDR_{85} \leftarrow S$

$LEDR_9 \leftarrow Cout$

$LEDR_0 \leftarrow Cout \rightarrow LEDR_0$

$LEDR_1 \leftarrow S \rightarrow LEDR_{41}$



node ①: $a\bar{b} + \bar{a}b$

$n = a\bar{b} + \bar{a}b$

$n = a \wedge b$

$s = Ci\,\bar{n} + \bar{Ci}\,n$

$s = Ci \wedge n$

$B_3B_2B_1B_0$	$A_3A_2A_1A_0$	Cin	$S_3S_2S_1S_0$	Cout
0000	0000	0	0000	0
① 0001	0001	0	0010	0
② 0011	0001	1	0101	0
③ 1010	0101	0	1111	0
④ 1001	1111	1	1001	1
⑤ 1111	1111	1	1111	1

① 0001 + 0001 = 0010 (Cin = 1)

② 0011 + 0001 = 0101 (Cin = 1)

③ 1010 + 0101 = 1111 (Cin = 0)

④ 1001 + 1111 = 1001, carry 1 (Cin = 1)

⑤ 1111 + 1111 = 1111, carry 1 (Cin = 1)

Part 4

X : SW_{74}

Y : SW_{30}

Cin : SW_8

BCD(X) → HEX5

BCD(Y) → HEX3

BCD(S_1) → HEX1

BCD(S_0) → HEX0

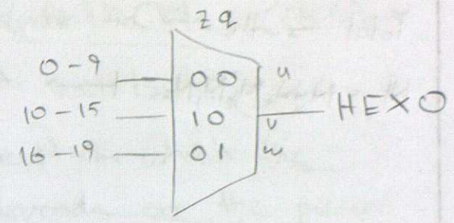
error : LEDR9

if X > 9

or Y > 9

$X_3X_2X_1X_0$

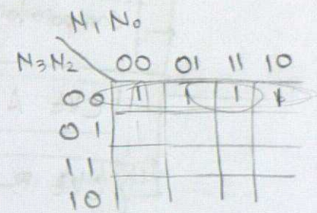
$X_7X_6X_5X_4$ $Y_3Y_2Y_1Y_0$ c_in



	$N_4N_3N_2N_1N_0$	z	q
(0)	00000	0	0
(1)	00001	0	0
	...	...	...
(9)	01001	0	0
(10)	01010	1	0
	01011	1	0
	...	...	...
(15)	01111	1	0
(16)	10000	0	1
(17)	10001	0	1
(18)	10010	0	1
(19)	10011	0	1

z = 1

q = 1



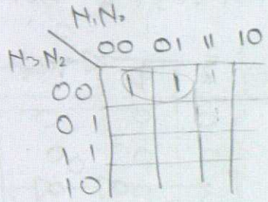
$N_4 = 1$

$q = N_4\bar{N}_3\bar{N}_2$

Designing CKT B:

$N_4N_3N_2N_1N_0$	$B_3B_2B_1B_0$
10000	0110
10001	0111
10010	1000
10011	1001

$B_2 = B_1$



$B_3 = N_4\bar{N}_3\bar{N}_2N_1N_0 + N_4\bar{N}_3N_2\bar{N}_1\bar{N}_0$

$B_2 = N_4\bar{N}_3\bar{N}_2\bar{N}_1$

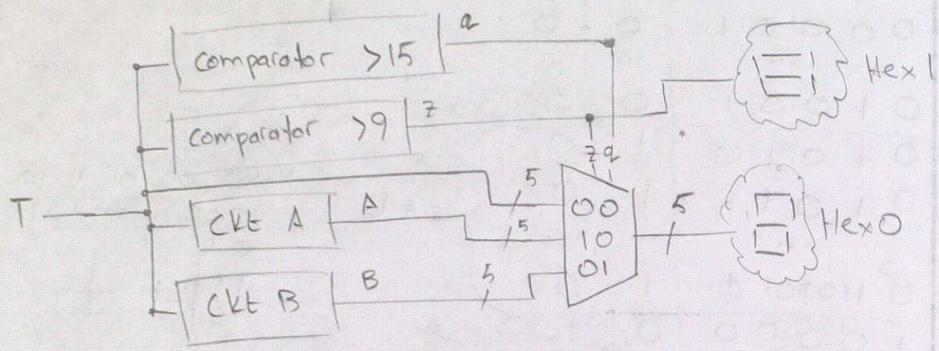
$B_1 = N_4\bar{N}_3\bar{N}_2\bar{N}_1$

$B_0 = N_4\bar{N}_3\bar{N}_2\bar{N}_1N_0 + N_4\bar{N}_3N_2\bar{N}_1\bar{N}_0$

$z = \bar{N}_4N_2N_3 + \bar{N}_4N_1\bar{N}_2N_3$

Total is the 5-bit number we get after summing X and Y

$N = N_4N_3N_2N_1N_0 \longmapsto S_1S_0$



Modified CKt A:

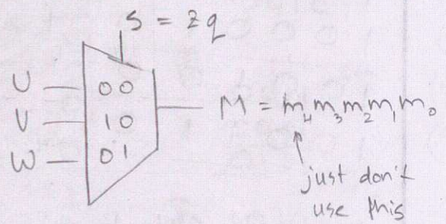
$A_3 = 0$

$A_2 = N_1N_2N_3\bar{N}_4$

$A_1 = \bar{N}_1N_2N_3\bar{N}_4$

$A_0 = N_0N_2N_3\bar{N}_4 + N_0N_1N_3\bar{N}_4$

number	z	q	hex1	hex0
0	0	0	0	0
1	0	0	0	1
...				
8	0	0	0	8
9	0	0	0	9
10	1	0	1	0
11	1	0	1	1
12	1	0	1	2
...				
14	1	0		
15	1	0		
16	0	1		
...				
19	0	1		



Modified CKt B

$B_3 = N_4\bar{N}_3\bar{N}_2N_1\bar{N}_0 + N_4\bar{N}_3\bar{N}_2N_1N_0$

$B_2 = N_4\bar{N}_3\bar{N}_2\bar{N}_1\bar{N}_0 + N_4\bar{N}_3\bar{N}_2\bar{N}_1N_0$

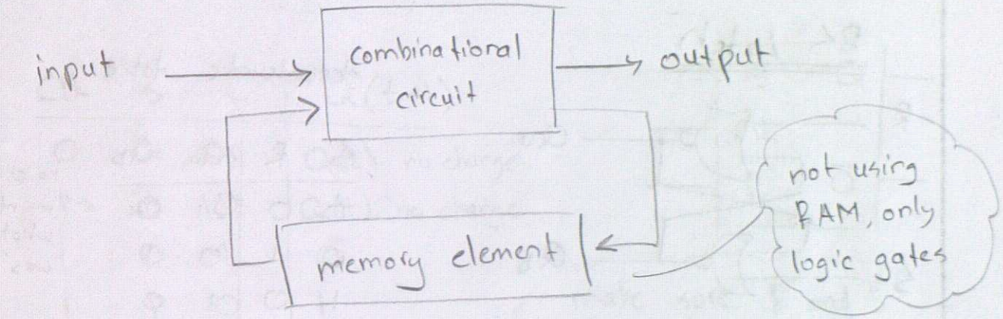
$B_1 = B_2$

$B_0 = N_4\bar{N}_3\bar{N}_2\bar{N}_1N_0 + N_4\bar{N}_3\bar{N}_2N_1N_0$

Storage Elements and Sequential Ckts (Ch.5)

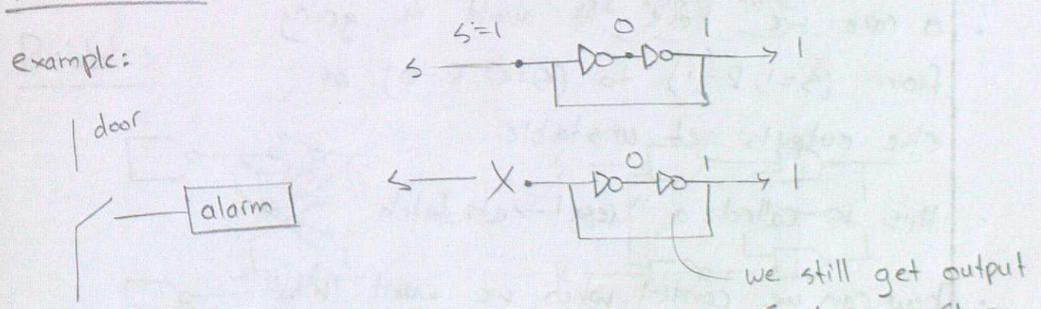
- So far, we've done combinational circuits
- combinational ckts: a logic circuit in which the output of the circuit only depends on the present value(s) of the input(s). (no way of remembering states)

Sequential Ckts



Basic Latch

example:

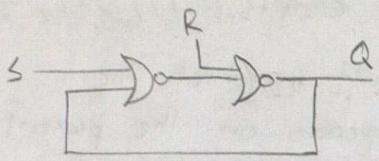


combinational logic

- door closed, no alarm
- door open, alarm
- door closed, no alarm

} we want alarm to stay on after activated

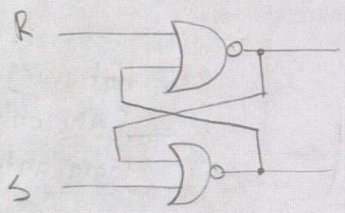
- how can we make it able to "reset?"



"reset" forces Q to 0

R=0	R=0	R=1
S=0 →	S=1 →	S=1
Q=0,0	Q=1	Q=0

RS Latch



characteristic table

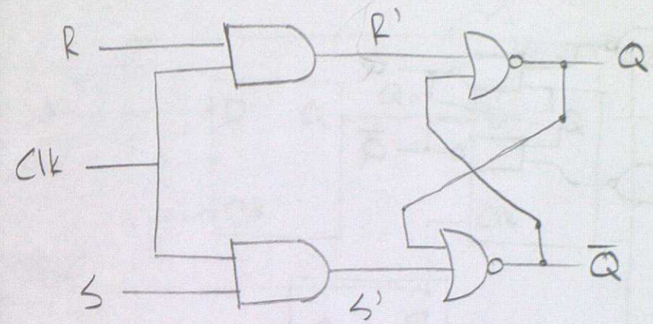
S	R	Qa	Qb	
0	0	0/1	0/1	← no change, maintain what it was
0	1	0	1	
1	0	1	0	
1	1	0	0	

- a case we have to avoid is going from (S=1, R=1) to (S=0, R=0) or else outputs get unstable.
- this is called a "reset-set latch"
- how can we control when we want this?

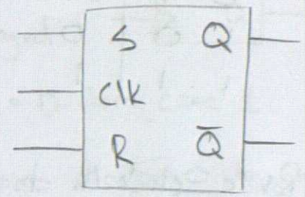
t1 t2 t3

R=
S=
Qa=

Gated SR Latch

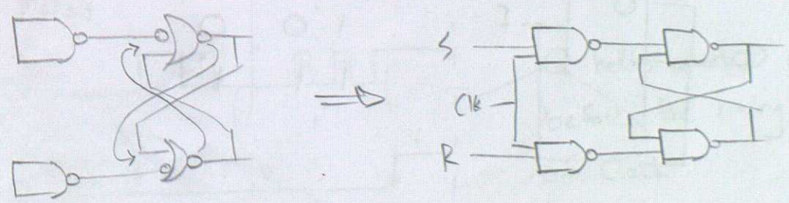


Clk	S	R	Q(t+1)
0	x	x	Q(t) no change
1	0	0	Q(t) no change
1	0	1	0
1	1	0	1
1	1	1	x

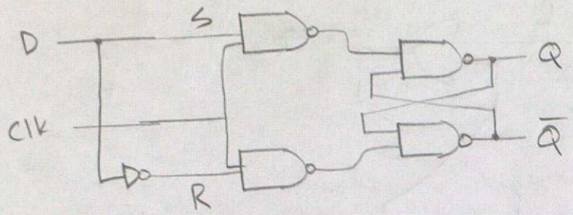


make sure R and S are not HIGH at the same time

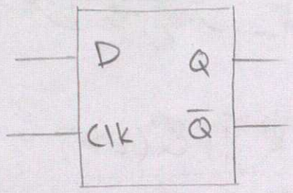
Bubbles



Gated D Latch

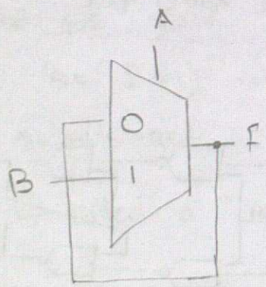


Clk	D	Q(t+1)
0	X	Q(t)
1	0	0
1	1	1



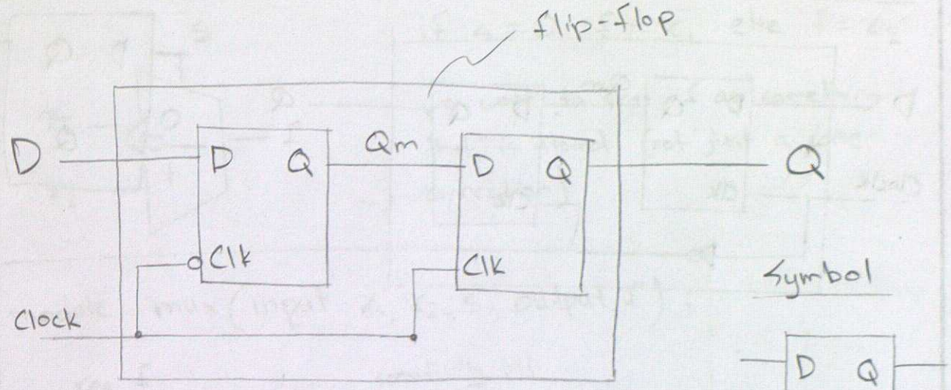
- while clock is high, Q follows D
- while clock is low, Q holds its value regardless of what D is

Multiplexer D-Latch

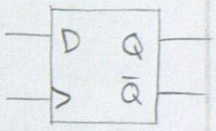


A	B	f
0	X	0/1 ← stored value from before
1	0	0
1	1	1

Edge-Triggered D Flip-Flops (positive-edge)



Symbol



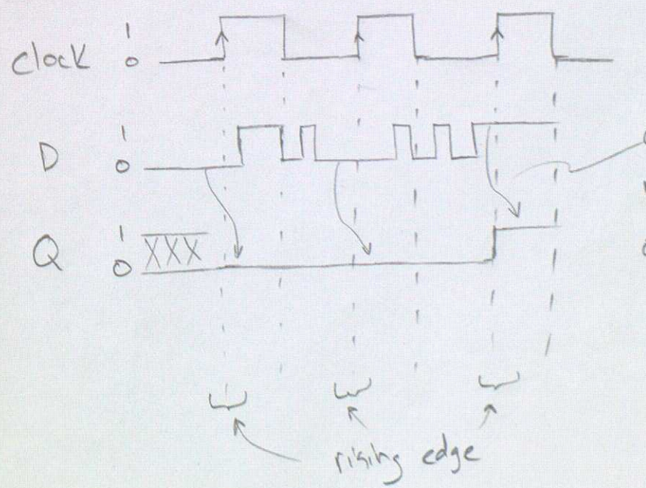
case 1: clock = 0, Qm = D, Q = unchanged

case 2: clock = 1, Qm = unchanged, Q = D from case 1

- Qm doesn't instantly change when we go Clock = 0 to Clock = 1
- recall: latch is level sensitive

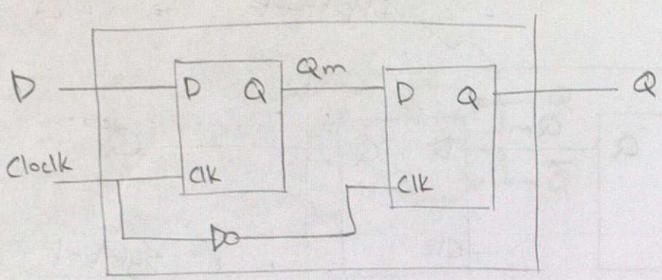
timing diagram

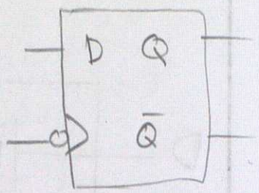
Q looks at what D was before rising edge and changes on rising edge of Clock



Q holds what D was before the rising edge of Clock

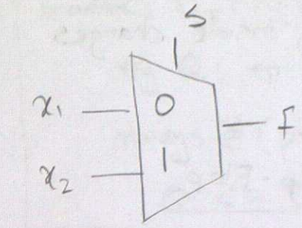
Negative Edge-Triggered Flip-Flops





Put in timing diagram and notes

Verilog for Sequential Circuits



if $s=0$ $f=x_1$ else $f=x_2$

we want to treat f as something that is stored (not just a wire connection)

```
module mux(input x1, x2, s, output f);
    reg f;
    always @(x1, x2, s);        // sensitivity list
        if (!s)
            f = x1;
        else
            f = x2;
endmodule
```

- always @(sensitivity list) ← commands within executed only when sensitivity list stuff changes
- always @* ← commands within executed when anything inside changes
- combinational v.s. sequential

Gated D-Latch

```
module latch(D, clk, Q);
    input D, clk;
    output reg Q;
    always @(D, clk)
        if (clk)
            Q = D
endmodule
```

else Q=Q implied
Q is unchanged

D-Flip-Flop

```
module flipflop (D, Clock, Q);
    input D, Clock;
    output reg Q;
    always @(posedge Clock)
        Q <= D;
endmodule
```

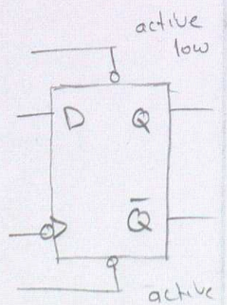
S = X + Y v.s. S <= X + Y
P = S[0] P <= S[0]

Preset and Clear

- preset_n = 0, things happen
- clear_n = 0, things happen

Asynchronous:

clear_n = 0	Q = 0
preset_n = 1	$\bar{Q}$ = 1
clear_n = 1	Q = 1
preset_n = 0	$\bar{Q}$ = 0

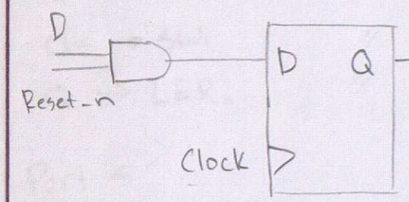


Flip Flop

```
module D_FF(input D, clock, output Q)
    reg Q; ← not "wire" anymore
    always @(posedge clock)
        Q <= D ← Q gets the value of what D was when we started
endmodule
```

looking for (+)ve edge of clock, because that's the only time Q can change (D can be whatever)

Reset (only writing code for reset)



↙ Synchronous reset

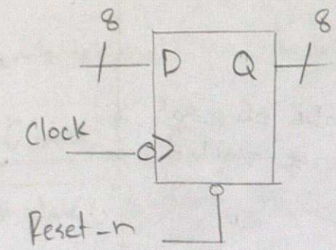
```
module DFF (D, Clock, Reset_n, Q);
    input D, Clock, Reset_n;
    output reg Q;
    always @(posedge Clock)
        if (Reset_n == 0)
            Q <= 1'b0;
        else
            Q <= D; ← non-blocking event... "when this event occurred, what was D? put that to Q, not something D is currently"
endmodule
```

Asynchronous Reset

```
always @(negedge Reset_n
         posedge Clock)
    if (Reset_n == 0)
        Q <= 1'b0;
    else
        Q <= D
```

8-bit Register

- flip-flop is a register (holds a value when we put clock)



We want this to be synchronous

```
module RS(D, Clock, Reset_n, Q);
    input [7:0] D;
    input Clock, Reset_n;
    output [7:0] reg Q;
always @(posedge Clock)
    if (Reset_n == 0)
        Q <= 8'b0;
    else
        Q <= D;
```


Part 1 Gated SR Latch

- When Clock is zero, Q has no change (no matter what S or R are)
- When Clock is 1, the following happens
  - R is 1 "resets" Q to zero
  - When S is 1, it sets our output Q to 1.
- When S and R are both 0, Q has no change (holds value)
- When S and R are both 1, we can't tell since Q is unstable

Part 2

$D \rightarrow SW_0$ //

$Clk \rightarrow SW_1$ //

$Q \rightarrow LER_0$ //

Part 3

//

Part 5

$(A)_{16} \rightarrow HEX_{32}$

$(B)_{16} \rightarrow HEX10$

$(S)_{16} \rightarrow HEX54$

$Cout \rightarrow LEDR0$

input flow

SW_{70} { A, B

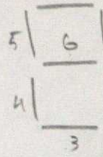
① set SW_{70} to A
② store A in 8 bit register
③ set SW_{70} to B
④ generate sum S = A + B
⑤ show sum S on HEX54
⑥ show carry out on LEDR0

things to do

- print out 0-16 in Hexadecimal

KEY0 → active-low asynchronous reset

KEY1 → manual clock input



$SW_{7:4}$ or $SW_{3:0}$	character	h_0	h_1	h_2	h_3	h_4	h_5	h_6
1010	A	1	1	1	0	1	1	1
1011	B	0	0	1	1	1	1	1
1100	C	1	0	0	1	1	1	0
1101	D	0	1	1	1	1	0	1
1110	E	1	0	0	1	1	1	1
1111	F	1	0	0	0	1	1	1

$x_4 x_3 x_2 x_1$

h_0

x_2x_1 / x_4x_3

	00	01	11	10
00	1	0	1	1
01	0	1	1	1
11	1	0	1	1
10	1	1	0	1

POS

$$f_0 = (\bar{x}_1 + x_2 + x_3 + x_4)\cdot(x_1 + x_2 + \bar{x}_3 + x_4)\cdot(\bar{x}_1 + x_2 + \bar{x}_3 + \bar{x}_4)\cdot(\bar{x}_1 + \bar{x}_2 + x_3 + \bar{x}_4)$$

h_1

x_2x_1 / x_4x_3

	00	01	11	10
00	1	1	1	1
01	1	0	1	0 b
11	0	1	0	0
10	1	1	0	1
			a	

POS

$$f_1 = (\bar{x}_1 + \bar{x}_2 + \bar{x}_4)\cdot(x_1 + \bar{x}_2 + \bar{x}_3)\cdot(\bar{x}_1 + x_2 + \bar{x}_3 + x_4)\cdot(x_1 + x_2 + \bar{x}_3 + \bar{x}_4)$$

h_2

x_2x_1 / x_4x_3

	00	01	11	10
00	1	1	1	0
01	1	1	1	1
11	0	1	0	0 a
10	1	1	1	1

POS

$$f_2 = (\bar{x}_4 + \bar{x}_3 + \bar{x}_2)\cdot(x_4 + x_3 + \bar{x}_2 + x_1)\cdot(\bar{x}_4 + \bar{x}_3 + x_2 + x_1)$$

h_3

x_2x_1 / x_4x_3

	00	01	11	10
00	1	0	1	1
01	0	1	0	1
11	1	1	0	1
10	1	1	1	0

POS

$$f_3 = (\bar{x}_2 + \bar{x}_1 + \bar{x}_3)\cdot(x_2 + \bar{x}_1 + x_4 + x_3)\cdot(x_1 + x_2 + \bar{x}_3 + x_4)\cdot(x_1 + \bar{x}_2 + x_3 + \bar{x}_4)$$

h_4

x_2x_1 / x_4x_3

	00	01	11	10
00	1	0	0	1
01	0	0	0	1
11	1	1	1	1
10	1	0	1	1

$$f_4 = (x_1 + x_2 + \bar{x}_3 + x_4)\cdot(\bar{x}_1 + x_2 + x_3 + \bar{x}_4)\cdot(\bar{x}_1 + x_4)$$

(indices: $0\ 1\ 2\ 3$ / $0\ 1\ 2\ 3$ / $0\ \ 3$)

h_5

x_2x_1 / x_4x_3

	00	01	11	10
00	1	0	0	0
01	1	1	0	1
11	1	0	1	1
10	1	1	1	1

POS

$$f_5 = (x_4 + x_3 + \bar{x}_1)\cdot(x_1 + \bar{x}_2 + x_3 + x_4)\cdot(\bar{x}_1 + \bar{x}_2 + \bar{x}_3 + x_4)\cdot(\bar{x}_1 + x_2 + \bar{x}_3 + \bar{x}_4)$$

h_6

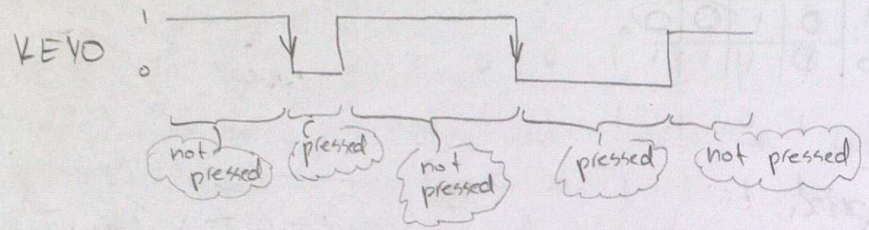
x_2x_1 / x_4x_3

	00	01	11	10
00	0	0	1	1
01	1	1	0	1
11	0	1	1	1
10	1	1	1	1

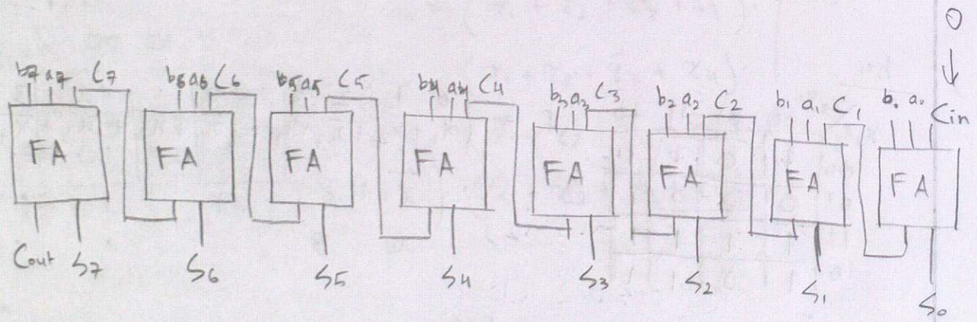
POS

$$f_6 = (x_4 + x_3 + x_2)\cdot(\bar{x}_1 + \bar{x}_2 + \bar{x}_3 + x_4)\cdot(\bar{x}_4 + \bar{x}_3 + x_2 + x_1)$$

- KEY0 being active-low means 0 V level means logical 1
  - ↳ not pressed : 0 V → KEY0 = 1
  - ↳ pressed and held : 1 V → KEY0 = 0



- We want to reset our register when we press KEY0
  - ↳ on negative edge, we do our resetting



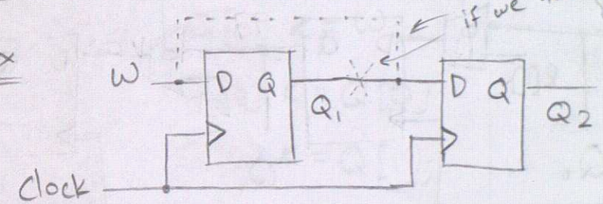
Blocking V.S. Non-blocking Statement

```
Q = D;   // blocking
Q <= D;  // non-block
```

make sure statements are non-blocking when you have chain flipflops

- within an always block, the blocking assignments are evaluated in the order in which they appear
- however, non-blocking statements are evaluated concurrently and their order is not relevant

Ex



- snapshots in time → rising edge
- on rising edge, Q_1 is what w was, Q_2 is what Q_1 was @ risedge
- Q_1 becomes what w was @ risedge
- Q_2 becomes what Q_1 was @ risedge

```
module example (w, Clock, Q1, Q2);
  input w, Clock;
  output reg Q1, Q2;
  always @(posedge Clock)
    begin
      Q1 = w;
      Q2 = Q1;
    end
endmodule
```

should be

```
Q1 <= w;
Q2 <= Q1
```

or

```
Q2 <= Q1;
Q1 <= w;
```

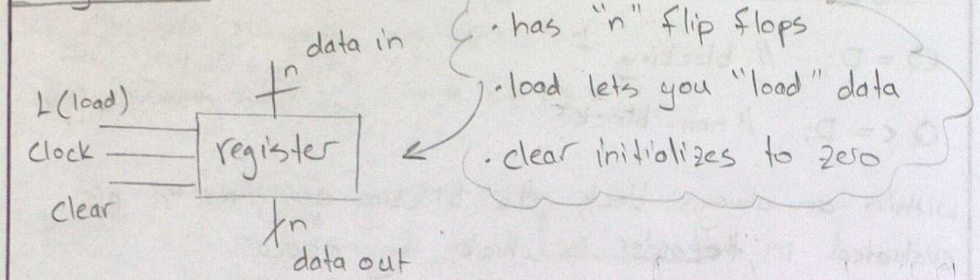
```
Q1 = w;
Q2 = Q1;
```

this is wrong

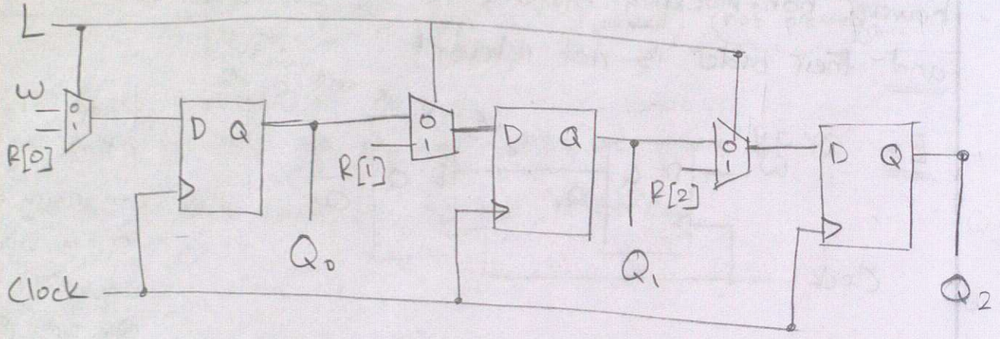
since these are blocking assignments, Q_1 becomes w, and then Q_2 becomes Q_1, which is w

$\therefore Q_2 = Q_1 = w$ X

Registers



- has "n" flip flops
- load lets you "load" data
- clear initializes to zero



parallel load

- If L = 1 and ↑ clock, each FF gets respective R

shift register

- If L = 0 and ↑ clock, Q_0 gets w; Q_1 gets Q_0; Q_2 gets Q_1 } we "shifted" or propagated down the line:

This ckt shows a shift register with the ability to parallel load

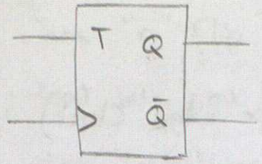
data simultaneously is loaded into register

```
module — (          )
   :
  always @ (posedge Clock)
      if (L == 1)
          Q <= R;
      else
        begin
            Q[0] <= w;
            Q[1] <= Q[0];
            Q[2] <= Q[1];

endmodule
```

Week 6: Lecture 2

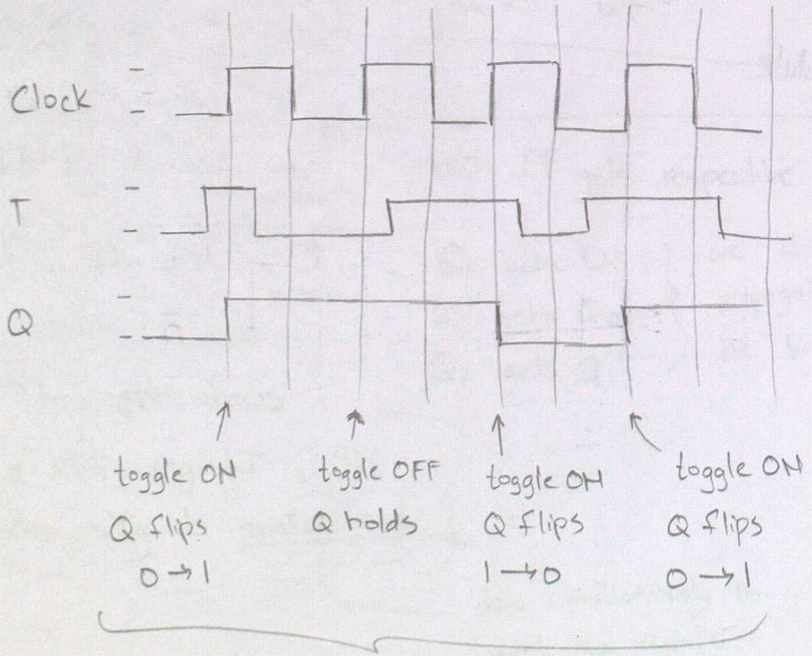
T-Flip Flop

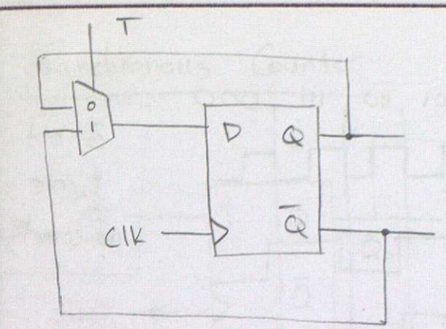
toggle flip flop



T	Q(t+1)
0	Q(t)
1	Q̄(t) ← output is inverse of what Q was

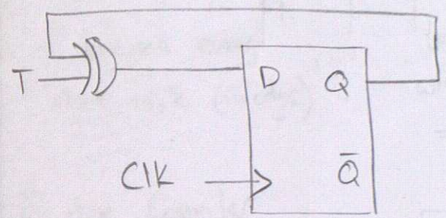
- on rising edge of clock
  - ↳ if toggle ON : Q flips its value
  - ↳ if toggle OFF : Q stays and holds its value





develop the table

T	Q(t+1)
0	
1	



exclusive OR

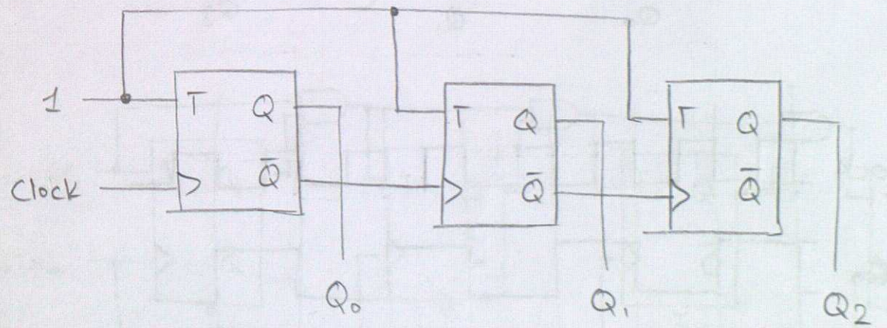
$D = T \oplus Q$

$T = 0,\ D = 0 \oplus Q = Q$

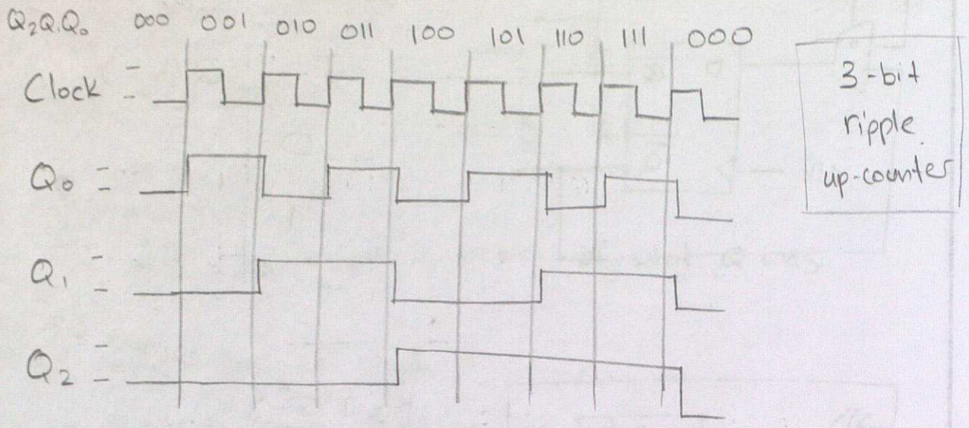
$T = 1,\ D = 1 \oplus Q = \overline{Q}$

Counter

3-bit up-counter

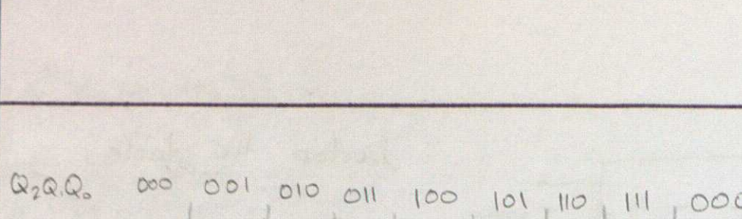


- not all flip flops are driven by our Clock
- we're giving same input to them
- asynchrous, triggered, but not on the same thing

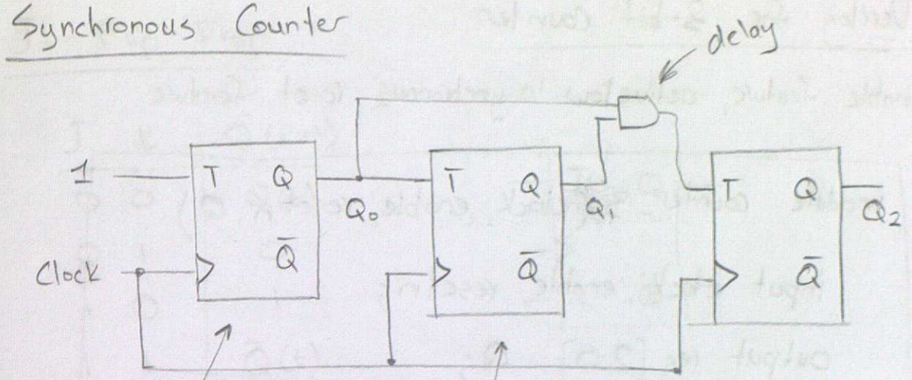


3-bit ripple up-counter

3-bit down-counter

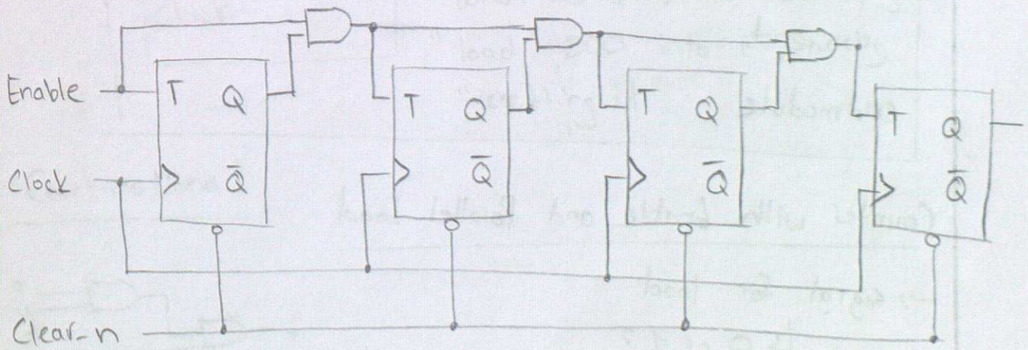


Synchronous Counter



Enable Counter

- replace "1" with Enable → when Enable = 1, counter works
  Enable = 0, counter stops (holds)
- add a clear_n capability



Verilog for 3-bit counter

- enable feature, activelow asynchronous reset feature

```
module counter_3 (clock, enable, resetn, Q)
    input clock, enable, resetn;
    output reg [2:0] Q;
    always @ (posedge clock, negedge resetn)
        begin
            if (resetn == 0)
                Q <= 3'b0;
            else if (enable == 1)
                Q <= Q + 1'b1 ;  <-- arithmetic "+"
        end
endmodule
```

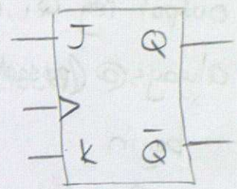
Counter with Enable and Parallel Load

→ signal for load
  ↳ 0 or 1?

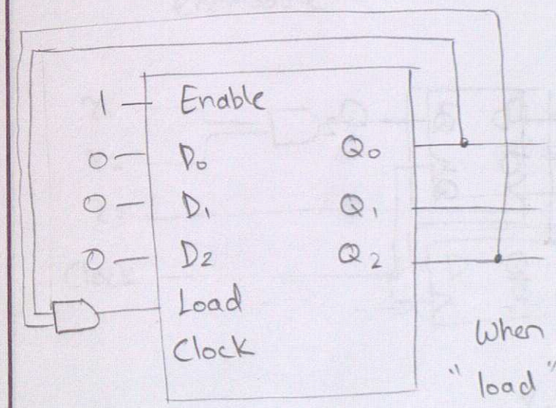
Q <= D;

JK-Flip Flop

J	K	Q(t+1)
0	0	Q(t)
0	1	0
1	0	1
1	1	$\bar{Q}(t)$

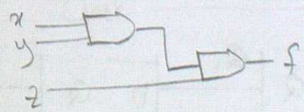


Reset Synchronization



When Q_0 & Q_2 are HIGH, you "load" 000 into FF, hereby "resetting it"

Combinational

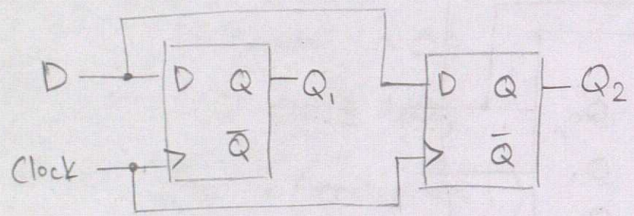


- f depends only on what x y and z are currently

Eg: module eg5(D, Clock, Q1, Q2)

```
input D, Clock;
output reg Q1, Q2;
always@(posedge Clock)
  begin
    Q1 = D;
    Q2 = Q1;
  end
endmodule
```

← combinational



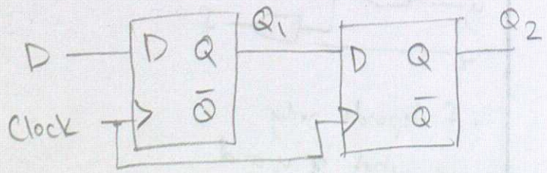
Sequential Version

- when we enter always block
  - ▷ evaluate assignments simultaneously

```
Q1 <= D;
Q2 <= Q1;
```

Q1 takes the D value as we entered always

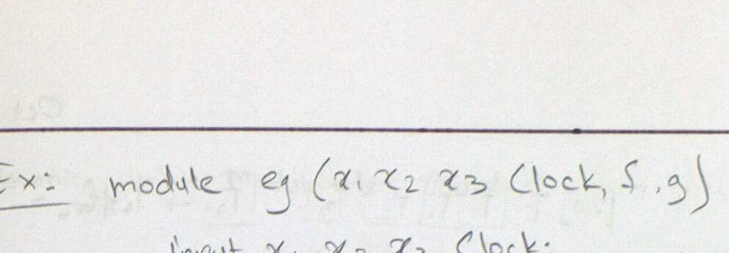
Q2 takes the Q1 value as we entered always



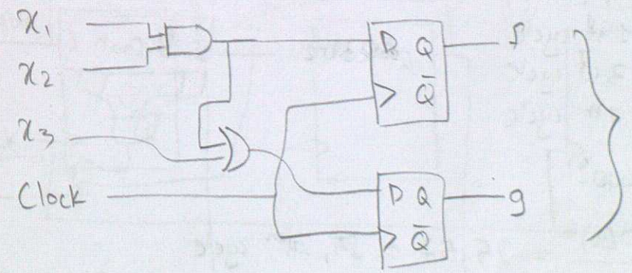
Ex: module eg (x1 x2 x3 Clock, f, g)

```
input x1 x2 x3 Clock;
output reg f, g;
always @(posedge Clock)
  begin
    f = x1 & x2;
    g = f | x3;
  end
endmodule
```

- representing numbers
- boolean expressions
- minimizing ckt
- POS, SOP
- k maps
- sequencing devices
- verilogs
- designs



f <= x1 & x2
g <= f | x3



f = x1 & x2;
g = f | x3;

Lab 4 Oct 12 2024

Part 1

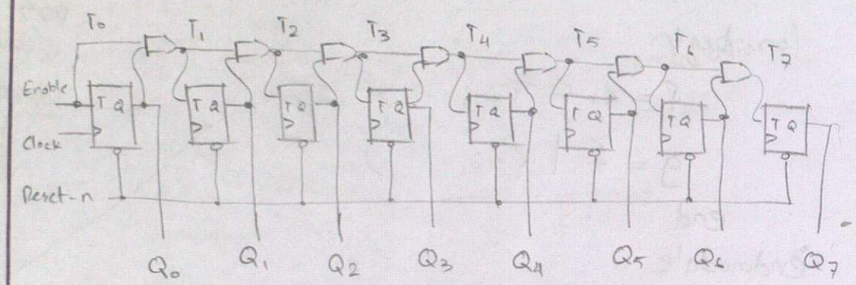
KEY0 → Clock

SW1 → Enable

SW0 → Clear

[7:0] T T_0 T_1 T_2 T_3 ⋯ T_7 → $T_6 \& Q_6$

SW1 → $T_0 \& Q_0$ → $T_1 \& Q_1$ → $T_2 \& Q_2$



Part 2

same inputs as part 1, except use HEX3-0

Part 3

50 MHz = 50×10^6 times per second

We want counter to update 1 time per second

1 bit → $2^1 = 2$, 1st cycle

2 bit → $2^2 = 4$, 3rd cycle

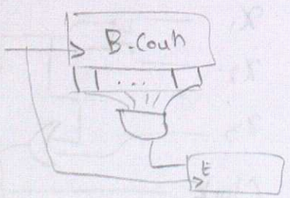
3 bit → $2^3 = 8$, 7th cycle

x bit → $2^x = 50 \times 10^6$,

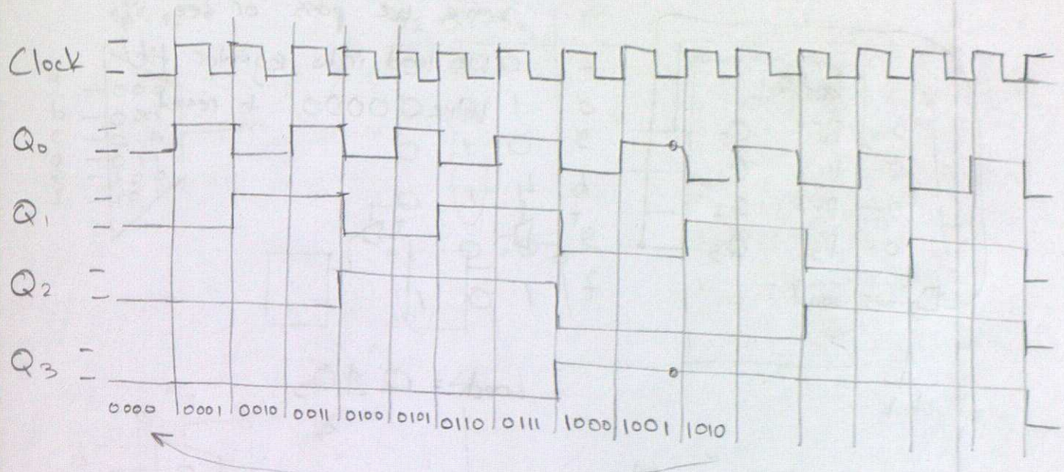
$$x = \frac{\ln 50(10)^6}{\ln 2} = 25.57 \approx 25, 24^{th} \text{ cycle}$$

↓

26 dig 2



Designing 4-bit Modulo-ten Counter

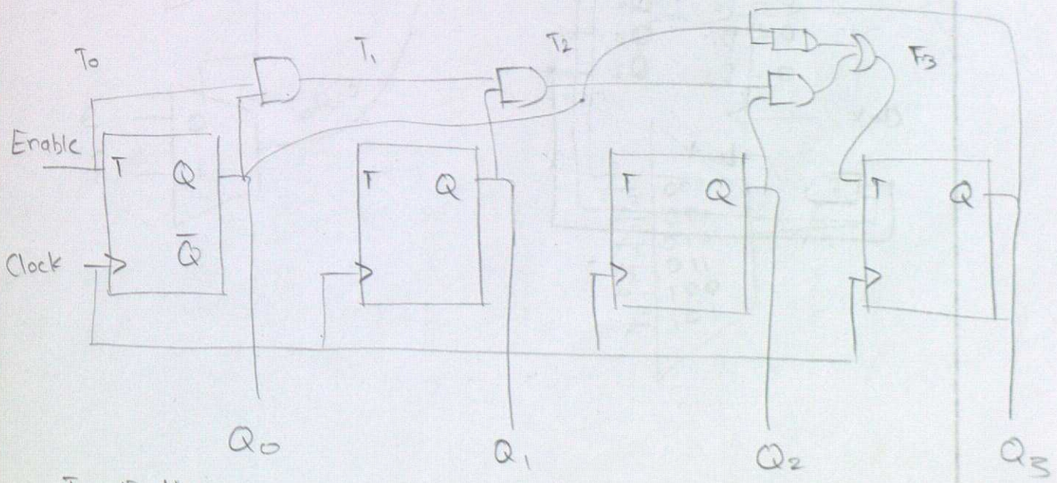


Q_0 toggles when Clock & Enable (not shown)

Q_1 toggles when Q_0

Q_2 toggles when $Q_0 \& Q_1$

Q_3 toggles when $Q_0 \& Q_1 \& Q_2 + \cancel{Q_0 \& Q_1 \& Q_2 \& Q_3}$ ($Q_0 \& Q_3$)



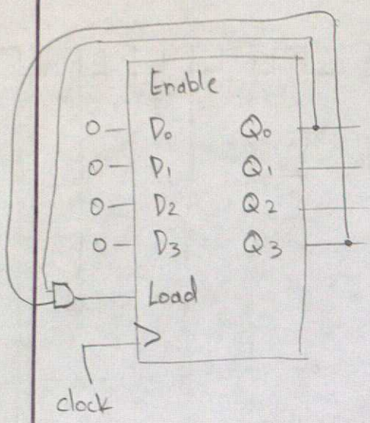
T_0 = Enable

$T_1 = T_0 \& Q_0$

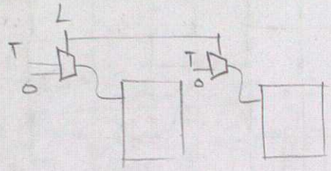
$T_2 = T_1 \& Q_1$

$T_3 = (T_2 \& Q_2) \,|\, (Q_1 \& Q_3)$

Reset Synchronization



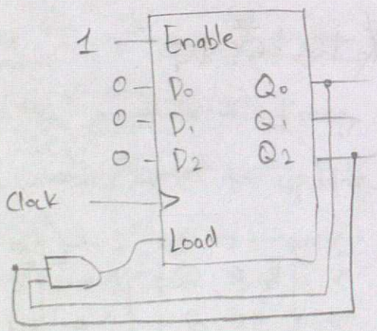
- once we pass "or see" 9, we load into register the value "0000" to reset.



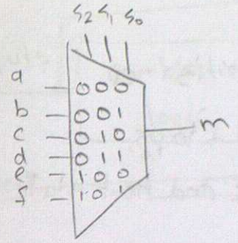
Load = $Q_0 \& Q_3$

Part 4

3-bit modulo 6 counter, Load = $Q_0 \& Q_2$

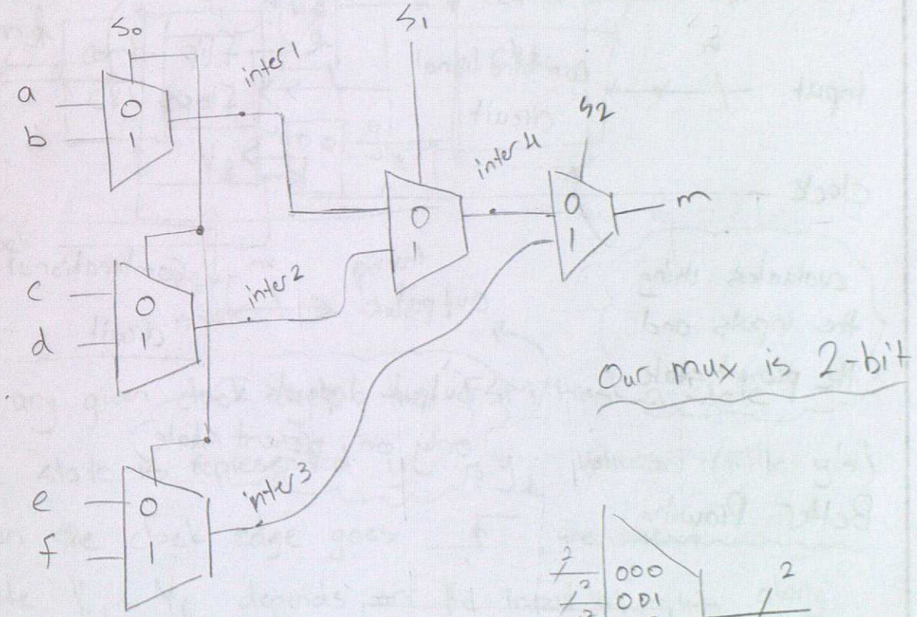


Designing 6 to 1 multiplexer

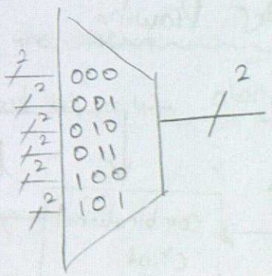


S_2	S_1	S_0	m
0	0	0	a
0	0	1	b
0	1	0	c
0	1	1	d
1	0	0	e
1	0	1	f

$= \bar{S_n} x + S_n y$



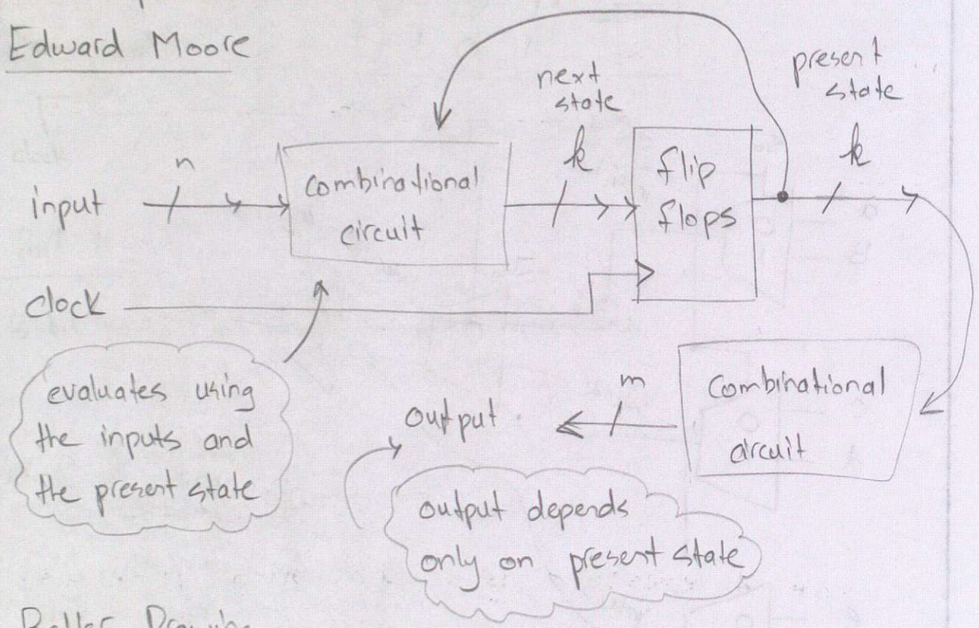
Our mux is 2-bit



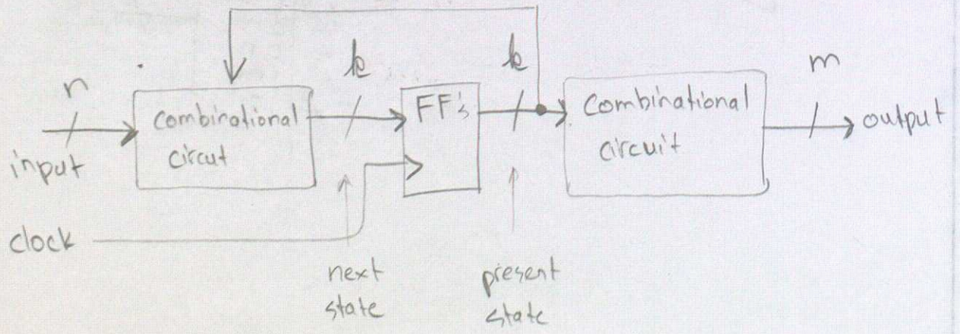
Finite State Machines (FSM) Ch.6

Definition: A finite state machine is a sequential circuit that has inputs, flip flops, and output. 1955, 1956: uses current state and next state in its process

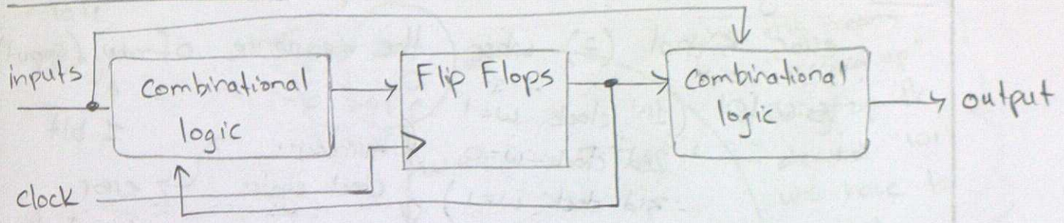
Edward Moore



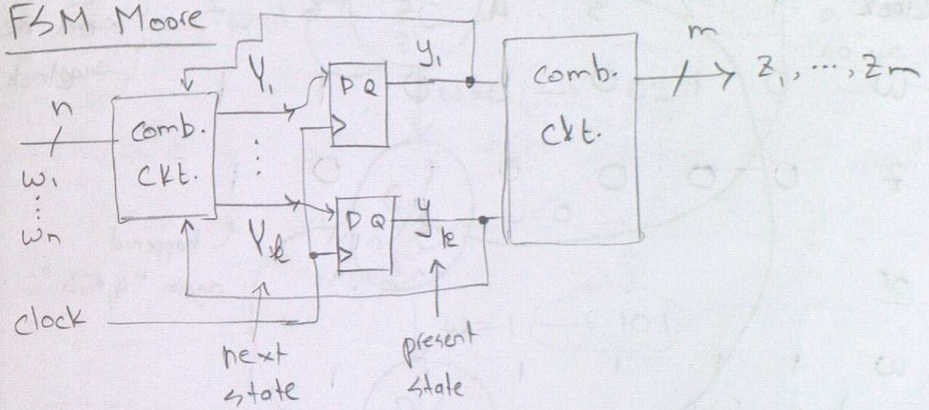
Better Drawing



George Mealy



FSM Moore



- At any given clock cycle, the FSM has a state
- The state is represented $y_1, \dots, y_k$ values (little y's)
- When the clock edge goes $\_\_\uparrow\_\_$, the next state $Y_1, \dots Y_k$ depends on the input $w_1, \dots, w_n$ along with the present state $y_1, \dots, y_k$

Ex: Design an FSM that will produce an error signal (z) when the sequence of w (input) is 101 (1st clock w=1, 2nd clock w=0, 3rd clock w=1) } in 3 successive clock cycles → error signal is "1" set z=1 on next clock edge

↑ 1 bit

Clock	1	2	3	4	5	6	7
w	0	1	0	1	0	1	1
z	0	0	0	0	1	0	1

↑ 101 from "2,3,4"

↑ happened again "4,5,6"

or

w	1	1	1	1	1	1	1
z	0	0	0	0	0	0	0

Step 1: create a state diagram (create a reset)

Legend: name / z — state / output

Step 2: create a state table

Step 3: number of flip flops

a flip flop can store a 0 or 1, so it can represent 2 states

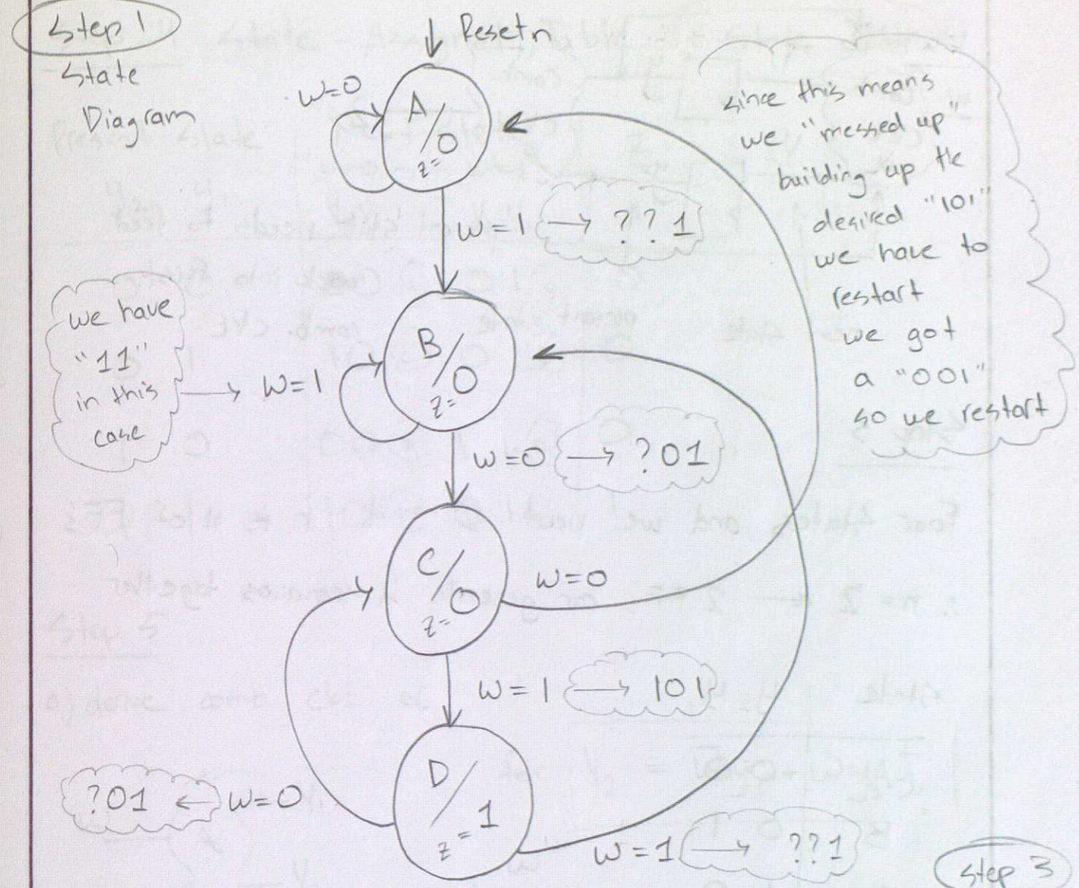
2^n ← number of FFs

$= 4$ ← number of states

$\Rightarrow n = 2$ in this case

Step 1

State Diagram



Step 2

present state	next state w=0	next state w=1	output z
A	A	B	0
B	C	B	0
C	A	D	0
D	C	B	1

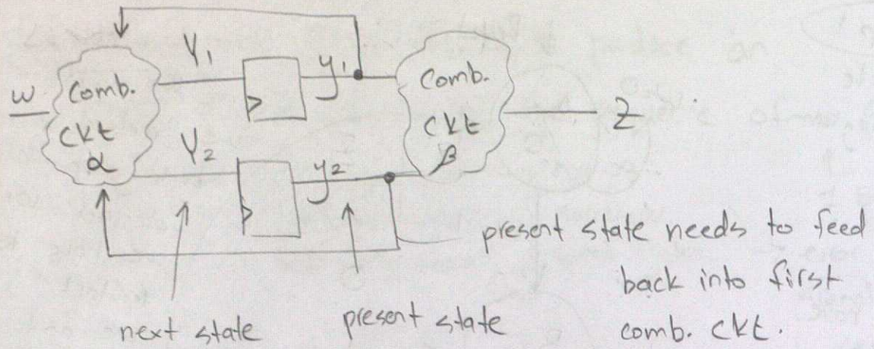
Step 3

2 FF's

State	y_2	y_1
A	0	0
B	0	1
C	1	0
D	1	1

y_2 ← output FF1

y_1 ← output FF2



Step 3

Four states and we need $2^n = 4$, n is # of FF's

$\therefore n = 2$ ← 2 FF's can generate 4 scenarios together

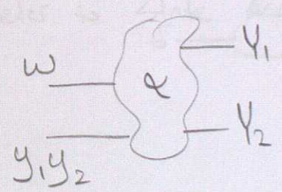
State	y_2 y_1
A	0 0
B	0 1
C	1 0
D	1 1

Step 4 State-Assigned Table + State Table

	Present State y_2 y_1	Next State $w=0$ Y_2Y_1	Next State $w=1$ Y_2Y_1	Z
A	0 0	00 (A)	01 (B)	0
B	0 1	10 (C)	01 (B)	0
C	1 0	00 (A)	11 (D)	0
D	1 1	10 (C)	01 (B)	1

Step 5

a) derive comb. ckt α



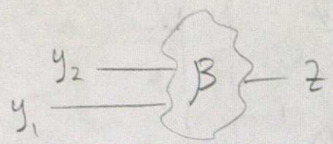
for $Y_2 = \bar{w}y_1 + wy_2\bar{y}_1$

y_2y_1 \ w	0	1
00	0	0
01	1	0
11	1	0
10	0	1

for $Y_1 = w$

y_2y_1 \ w	0	1
00	0	1
01	0	1
11	0	1
10	0	1

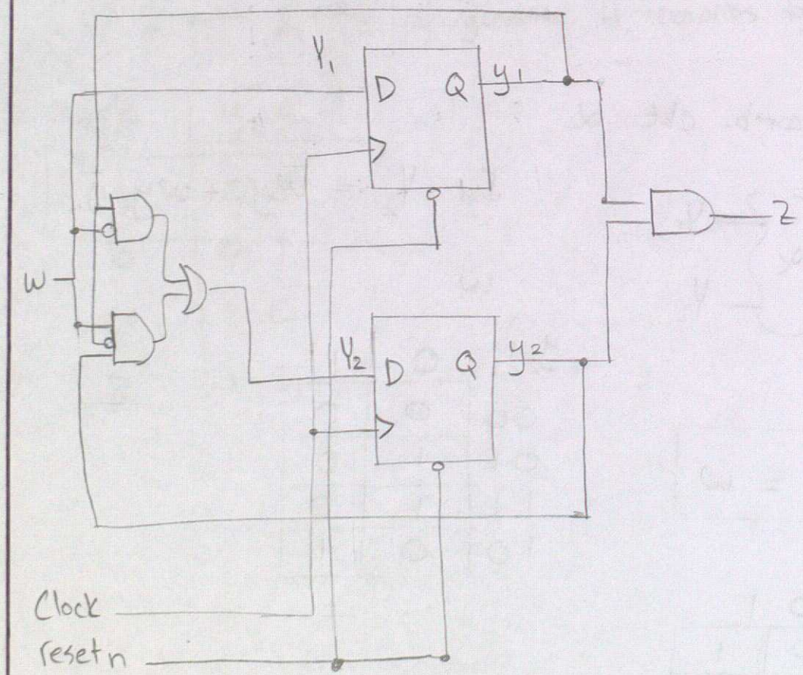
b) derive comb. ckt for β



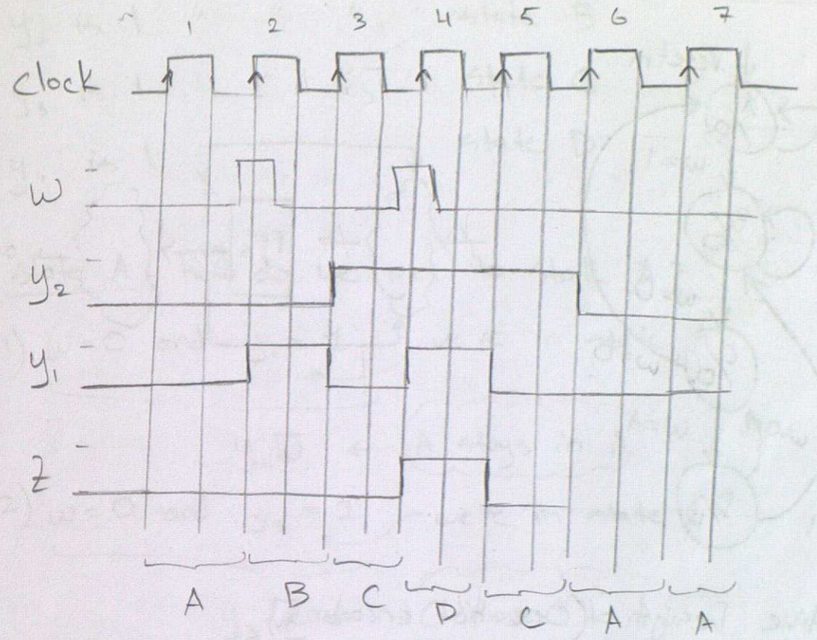
y_2 y_1	z
0 0	0
0 1	0
1 0	0
1 1	1

$\therefore$ $z = y_2 y_1$

Step 6 : implement



Timing Diagram

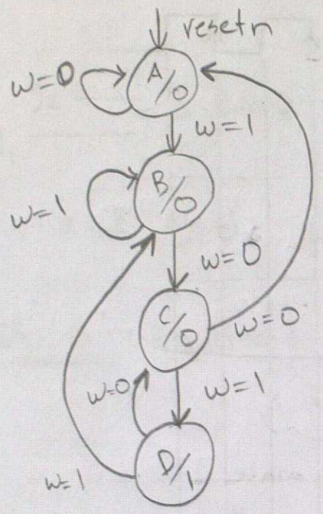


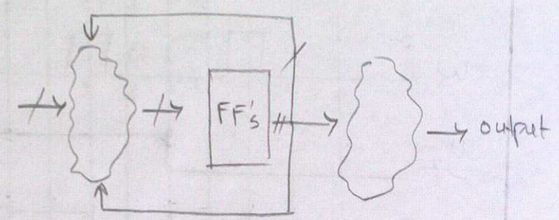
- refer to State Assign Table and ckt.

Week 7: Lecture 3

Oct 18, 2024

FSM Cont'd (fully encoded)

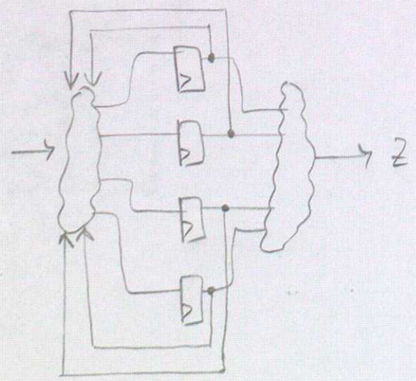




Alternative Design (One-hot encoding)

- use one FF for each state
- we had 4 states ⟶ four FF's

FF $y_4\ y_3\ y_2\ y_1$	State
0 0 0 1	A
0 0 1 0	B
0 1 0 0	C
1 0 0 0	D



- y_1 is 1 when we are in state A
- y_2 is 1 " " " " " state B
- y_3 is 1 " " " " " state C
- y_4 is 1 " " " " " state D

State A how do we get to state A?

1) $w=0$ and $y_1=1$ ← we're in state A

$y_1\bar{w}$ ← A stays in A

2) $w=0$ and $y_3=1$ ← we're in state C

$y_3\bar{w}$ ← C "moves" to A

$\therefore Y_1 = y_1\bar{w} + y_3\bar{w}$

State B

1) $w=1,\ y_2=1$
2) $w=1,\ y_4=1$
3) $w=1,\ y_1=1$

$Y_2 = y_1 w + y_2 w + y_4 w$

State C

1) $w=0,\ y_2=1$
2) $w=0,\ y_4=1$

$Y_3 = y_2\bar{w} + y_4\bar{w}$

State D

1) $w=1,\ y_3=1$

$Y_4 = y_3 w$

$Z = Y_4$

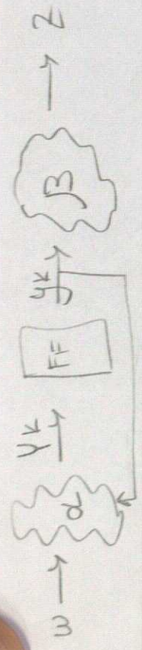
FSM 101 Sequence

```
module seq101 (input clock, w, resetn, output z);

  reg [2:1] y, Y;   // y1, y2 are present state FF's
                    // Y1, Y2 are next state FF's

  parameter [2:] A = 2'b00, B = 2'b01,
                C = 2'b10, D = 2'b11;

  // comb. ckt alpha
  always @(w, y)
  case (y)
    A: if (w) Y = B;
       else   Y = A;
    B: if (w) Y = B;
       else   Y = C;
    C: if (w) Y = D;
       else   Y = A;
    D: if (w) Y = B;
       else   Y = C;
  endcase
```

controlling expression, from which we have "situations"

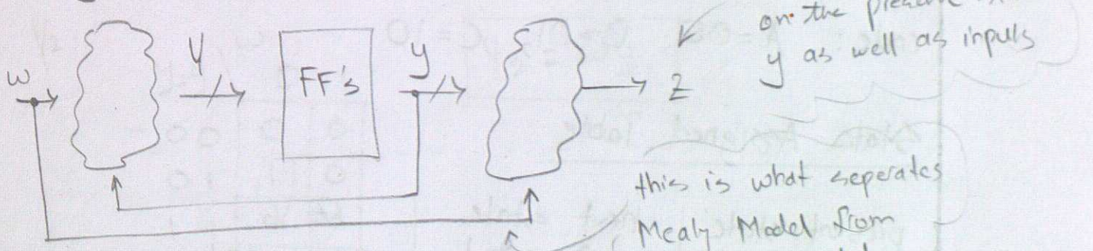
you get these from the state assigned table (see 5 pages before)

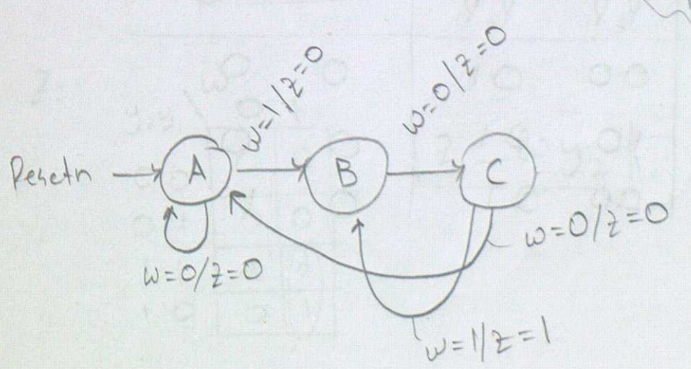
default: statement




```
  // Flip Flops
  always @(posedge clock)
    if (Resetn == 0)
      y <= A;
    else
      y <= Y;

  // comb. ckt beta
    assign z = (y == D);

endmodule
```

Mealy Model





State Table

Present State	Next State w=0	Next State w=1	z w=0	z w=1
A	A	B	0	0
B	C	B*	0	0
C	A	B	0	1

output uses input w

extra feature for Moore

Choose # FF's and Coding

- 3 states ⟹ requires 2 Flip-Flops

code: A=00, B=01, C=10

State Assigned Table

	present state y_2y_1	next state w=0 Y_2Y_1	next state w=1 Y_2Y_1	z w=0	z w=1
A	00	00	01	0	0
B	01	10	01	0	0
C	10	00	01	0	1

FSM : Mealy Model

- State Diagram
- State Table
- FF's and coding
- State Assigned Table
- Sythesize circuit

} General FSM process (either Moore, Mealy, one-hot)

Y_1 :

y_2y_1 \ w	0	1
00	0	1
01	0	1
11	d	d
10	0	1

$\therefore Y_1 = w$

Y_2 :

y_2y_1 \ w	0	1
00	0	0
01	1	0
11	d	d
10	0	0

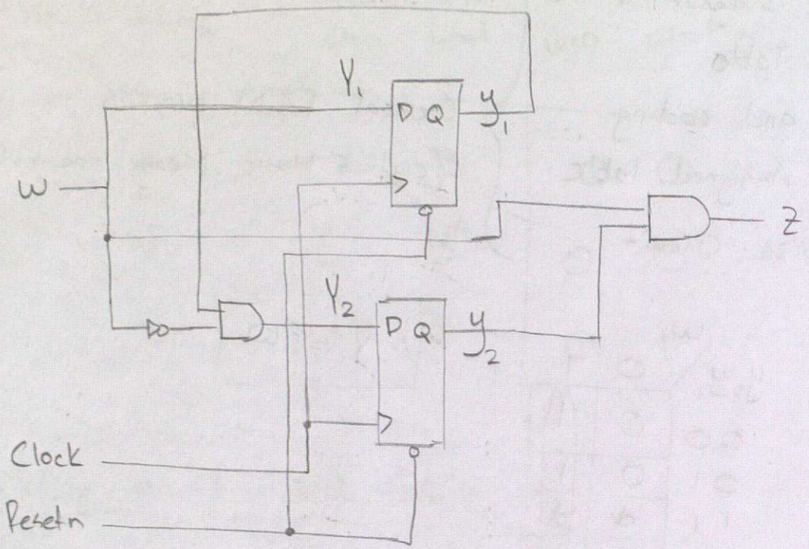
$Y_2 = \bar{w}y_1$

Z :

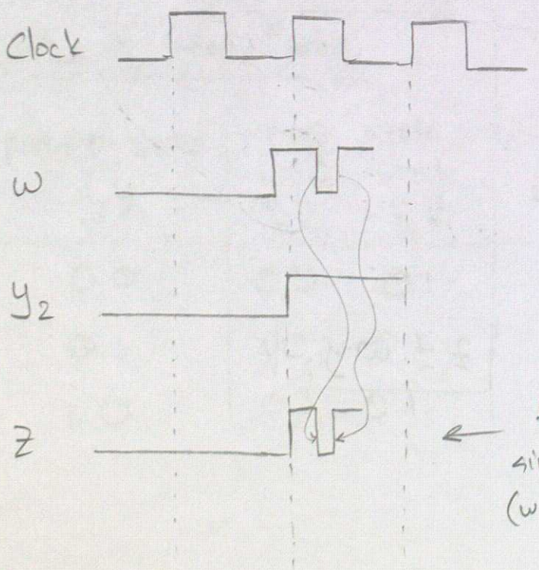
y_2y_1 \ w	0	1
00	0	0
01	0	0
11	d	d
10	0	1

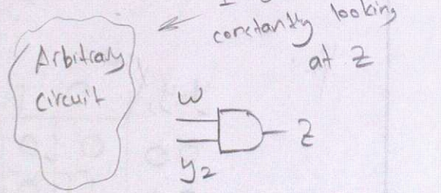
$Z = wy_2$

Circuit Mealy Model



Timing Diagram





← I won't be constantly looking at z

I'll sample z after x-time intervals.

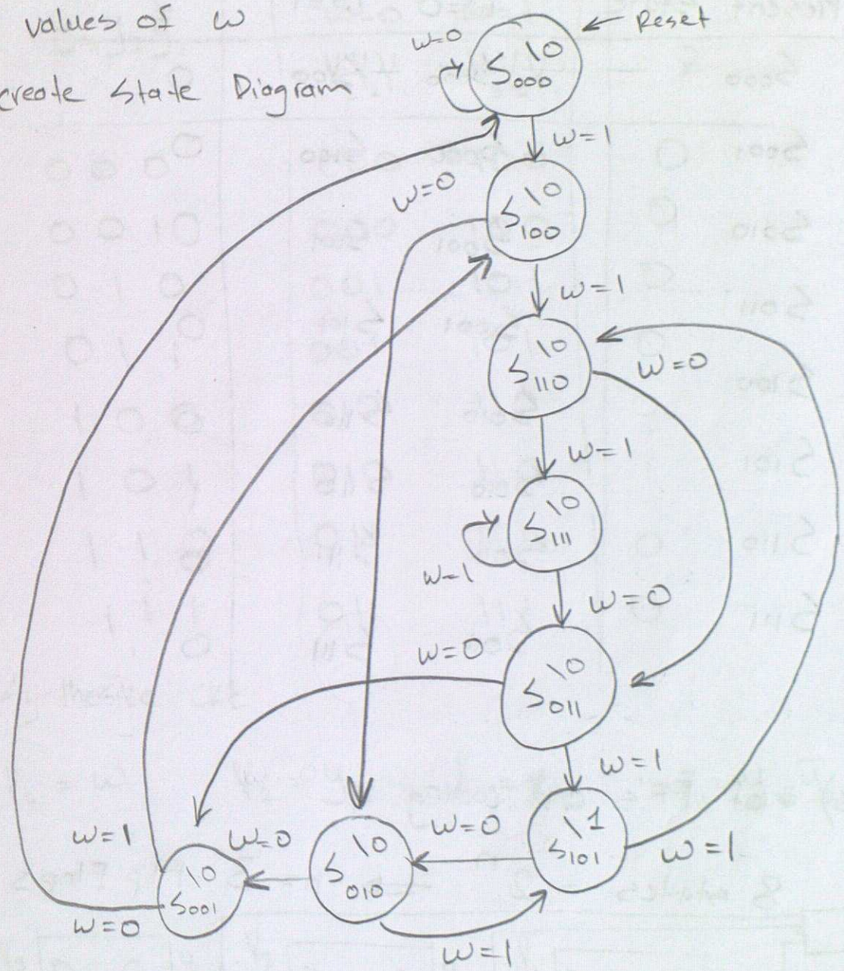
Glitch

← this is an issue since this is asyncronous (will change even not on clock edge)

Another Implementation

* Use a state diagram that represents the last 3 values of w

1) create State Diagram



2) State table

Present State	Next State w=0	Next State w=1	z
S_{000}	S_{000}	S_{100}	0
S_{001}	S_{000}	S_{100}	0
S_{010}	S_{001}	S_{101}	0
S_{011}	S_{001}	S_{101}	0
S_{100}	S_{010}	S_{110}	0
S_{101}	S_{010}	S_{110}	1
S_{110}	S_{011}	S_{111}	0
S_{111}	S_{011}	S_{111}	0

3) # of FF's and coding

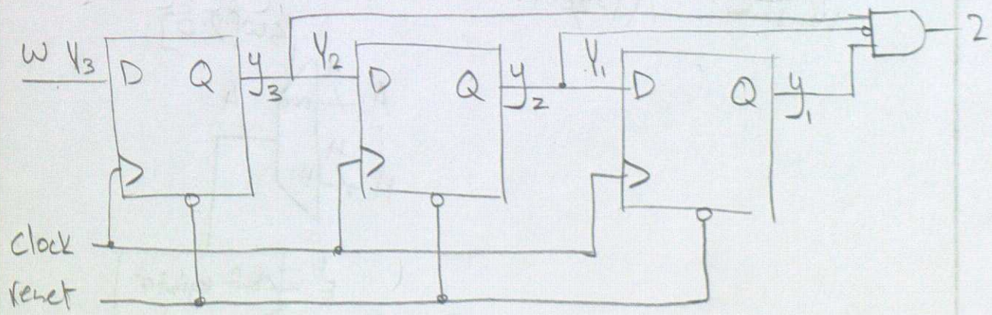
8 states $= 2^n \Longrightarrow n = 3$ Flip Flop's

4) State Assigned Table

present state $y_3y_2y_1$	next state w=0 $Y_3Y_2Y_1$	next state w=1 $Y_3Y_2Y_1$	z
000	000	100	0
001	000	100	0
010	001	101	0
011	001	101	0
100	010	110	0
101	010	110	1
110	011	111	0
111	011	111	0

5) Synthesize ckt

$Y_3 = w \qquad Y_2 = y_3 \qquad Y_1 = y_2 \qquad z = y_3 \bar{y}_2 y_1$



Lab 5 Spoilers

Morse Code

clock_50 

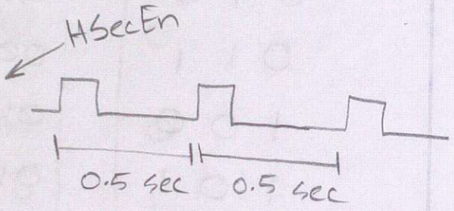
A · —
B — · · ·
⋮
V
H · · · ·

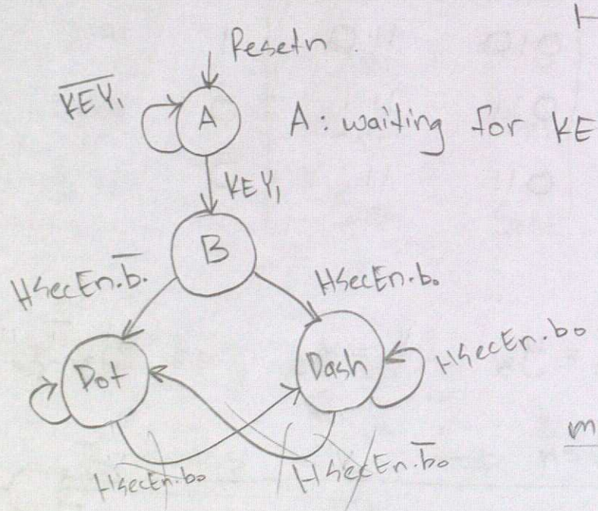
"·" is 0.5 sec pulse

"—" is 1.5 sec pulse

wait 0.5 sec b/n pulses

⇒ develop 0.5 sec enable

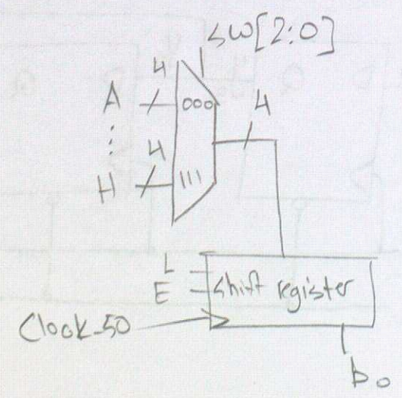




$\overline{b_0}$: dot

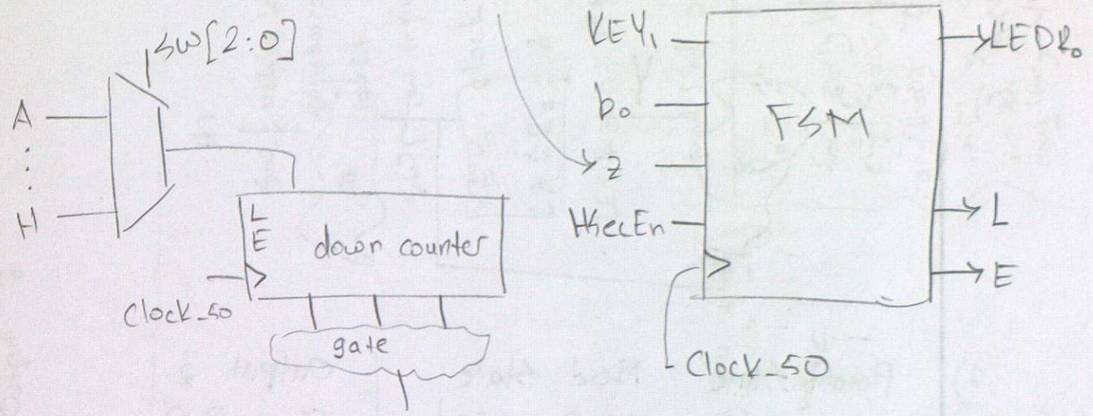
b_0 : dash

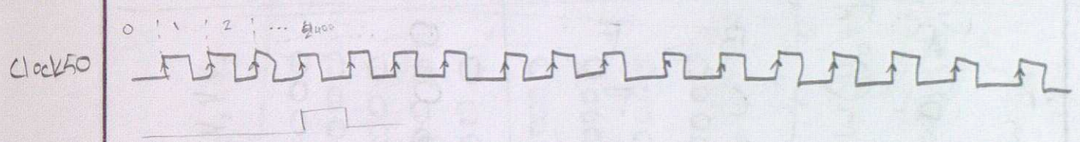
morse code



code length

so you know how to sequence light





Part 3

$SW_{2:0}$: inputs to set our letters

KEY_1: when pressed, transmit current letter

KEY_0: synchronous reset

SW_2	SW_1	SW_0	letter
0	0	0	A
0	0	1	B
0	1	0	C
0	1	1	D
1	0	0	E
1	0	1	F
1	1	0	G
1	1	1	H

Let b_0 represent dots & dashes

"·" is a "0"

"—" is a "1"

$\therefore \overline{b_0} = 0 = \cdot$

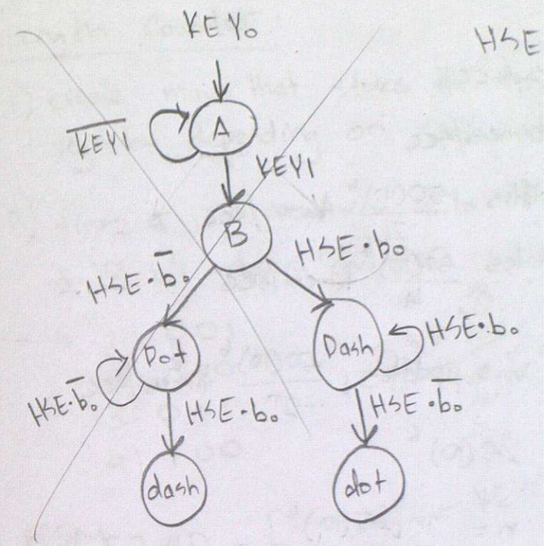
$b_0 = 1 = —$

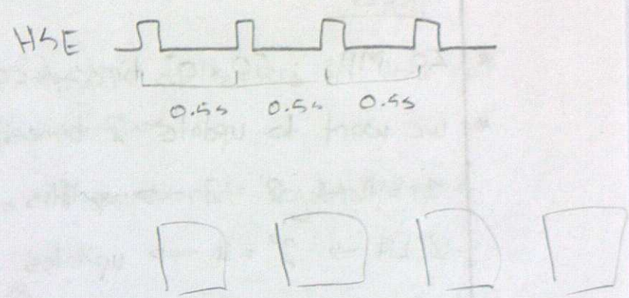
+ We have HSE

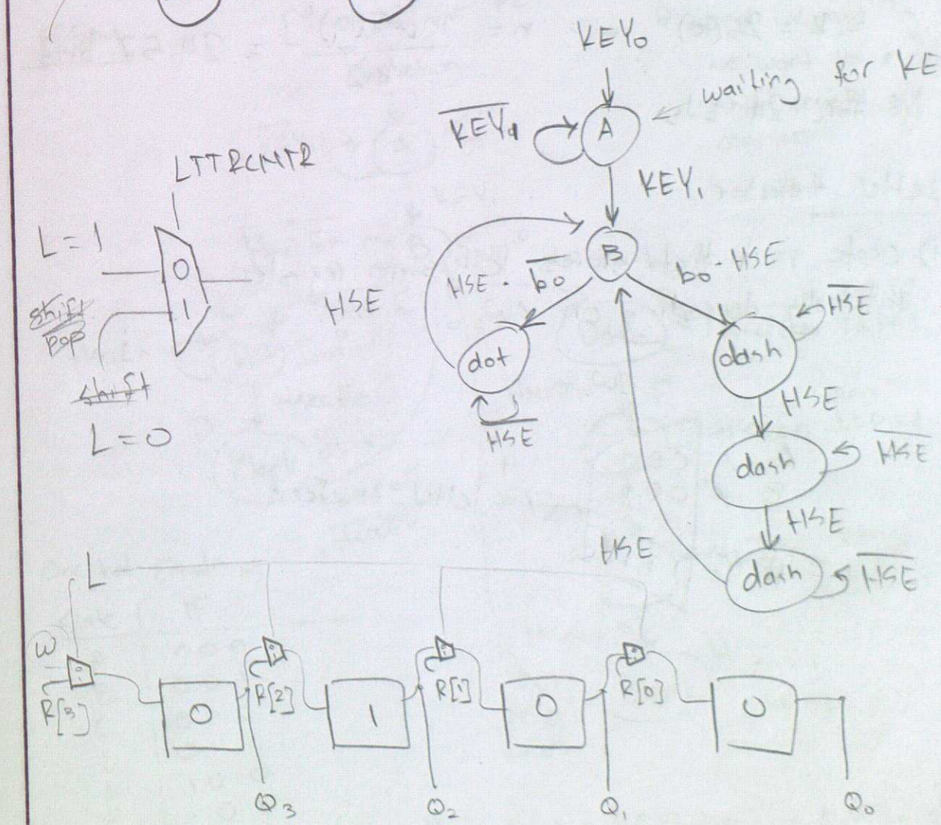
		LTTRCNTR		LTTRCNTR
A	·—	0 0 1 0	2	1 1 0 0 0
B	—···	0 0 0 1	4	1 1 1 1 0
C	—·—·	0 1 0 1	4	1 1 1 1 0
D	—··	0 0 0 1	3	1 1 1 0 0
E	·	0 0 0 0	1	1 0 0 0 0
F	··—·	0 1 0 0	4	1 1 1 1 0
G	——·	0 0 1 1	3	1 1 1 0 0
H	····	0 0 0 0	4	1 1 1 1 0

what to feed in →

←







HsecEn

- 50 MHz : 50×10^6 times/sec
- we want to update 2 times/sec

1 bit → $2^1 = 2$ ⇒ updates $\frac{50(10)^6}{2}$ times/sec

2 bit → $2^2 = 4$ ⇒ updates $\frac{50(10)^6}{4}$ times/sec

⋮

- in general : n bits → 2^n ⇒ updates $\frac{50(10)^6}{2^n}$ times/sec

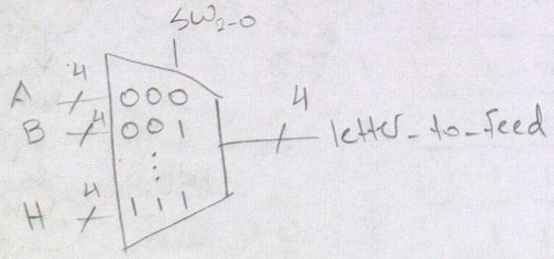
$$2 = \frac{50(10)^6}{k} \Rightarrow k = 25(10)^6$$

$$2^n = k = 25(10)^6 \Rightarrow n = \frac{\ln[25(10)^6]}{\ln 2} = 24.57 \text{ bits}$$

- go for 24 bits

Letter Selector

1) create mux that stores letters in register internally depending on $\langle sw_{2-0}$



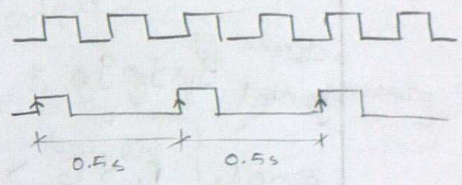
Length Counter

1) create mux that stores the length of letters in internal register depending on $\langle sw_{2-0}$ combos

2) from a wire reaching into ↑ mentioned register, load a 3-bit down counter

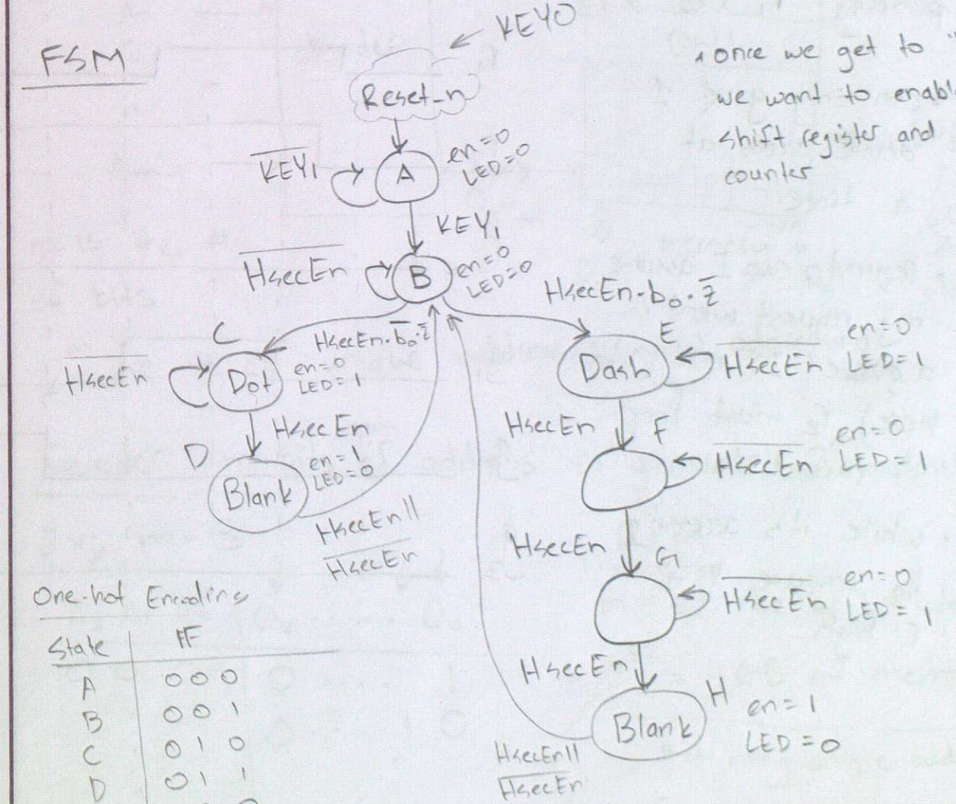
→ 1 : 001
2 : 010
3 : 011
4 : 100

$z = \overline{Q_0} \cdot \overline{Q_1} \cdot \overline{Q_2}$ (from our counter)



- once we get to "blank" we want to enable the shift register and down counter

FSM

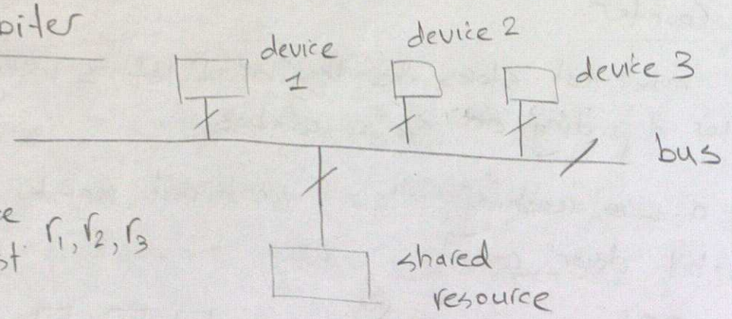


One-hot Encoding

State	FF
A	000
B	001
C	010
D	011
E	100
F	101
G	110
H	111

Clock, reset_n, KEY1, HsecEn, b_0, enable, LED

Arbiter

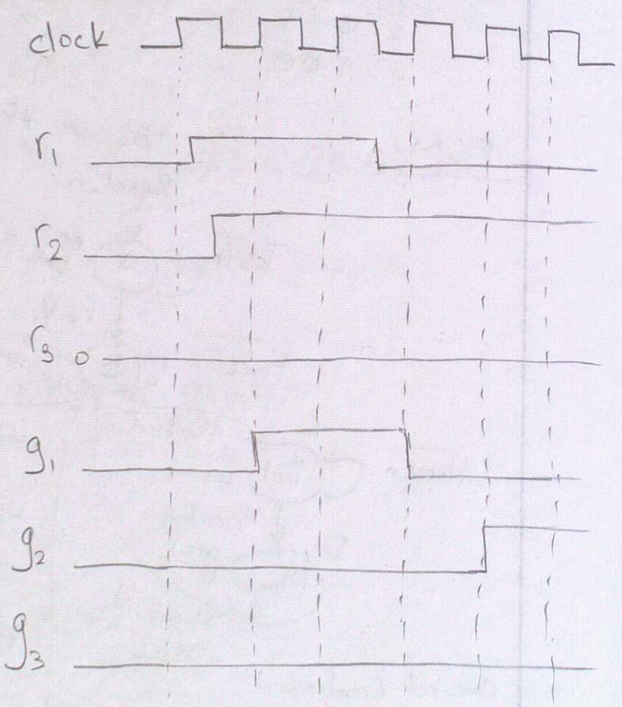


device request r_1, r_2, r_3

access granted g_1, g_2, g_3

priority $r_1 > r_2 > r_3$

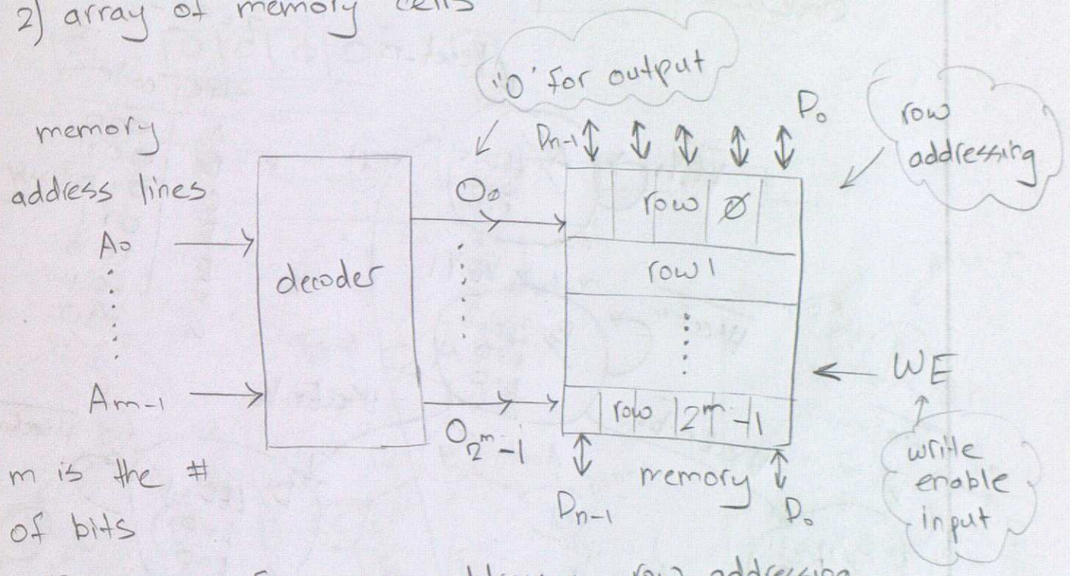
- can only give 1 device access at a time
- request: can I assume my request went in a queue? If not (like here) r_2 must keep it raised
- while it's accessing the resource, keep r high



Memory

storage of information

1) decoder
2) array of memory cells



m is the # of bits

2^m are # of unique addresses, row addressing

Decoder m-bits of address to generate 2^m outputs

Ex: m = 3

A_2 A_1 A_0	O_7 ..... O_0
0 0 0	0 1
	0 ... 1 0
	1 0 0 ... 0
1 1 1	

RAM

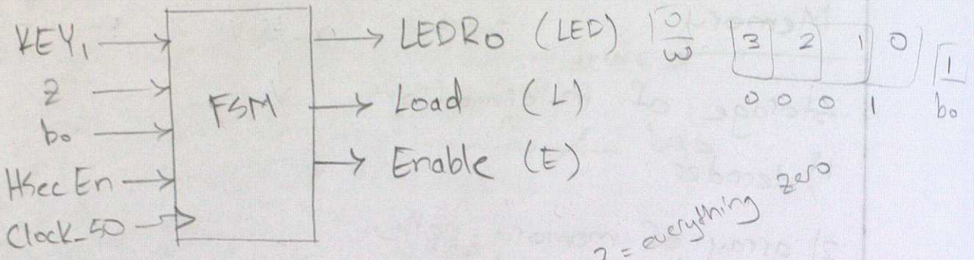
m = 32 (i.e 32-bit address)

2^{32} = 4 GB of memory

that many unique addresses

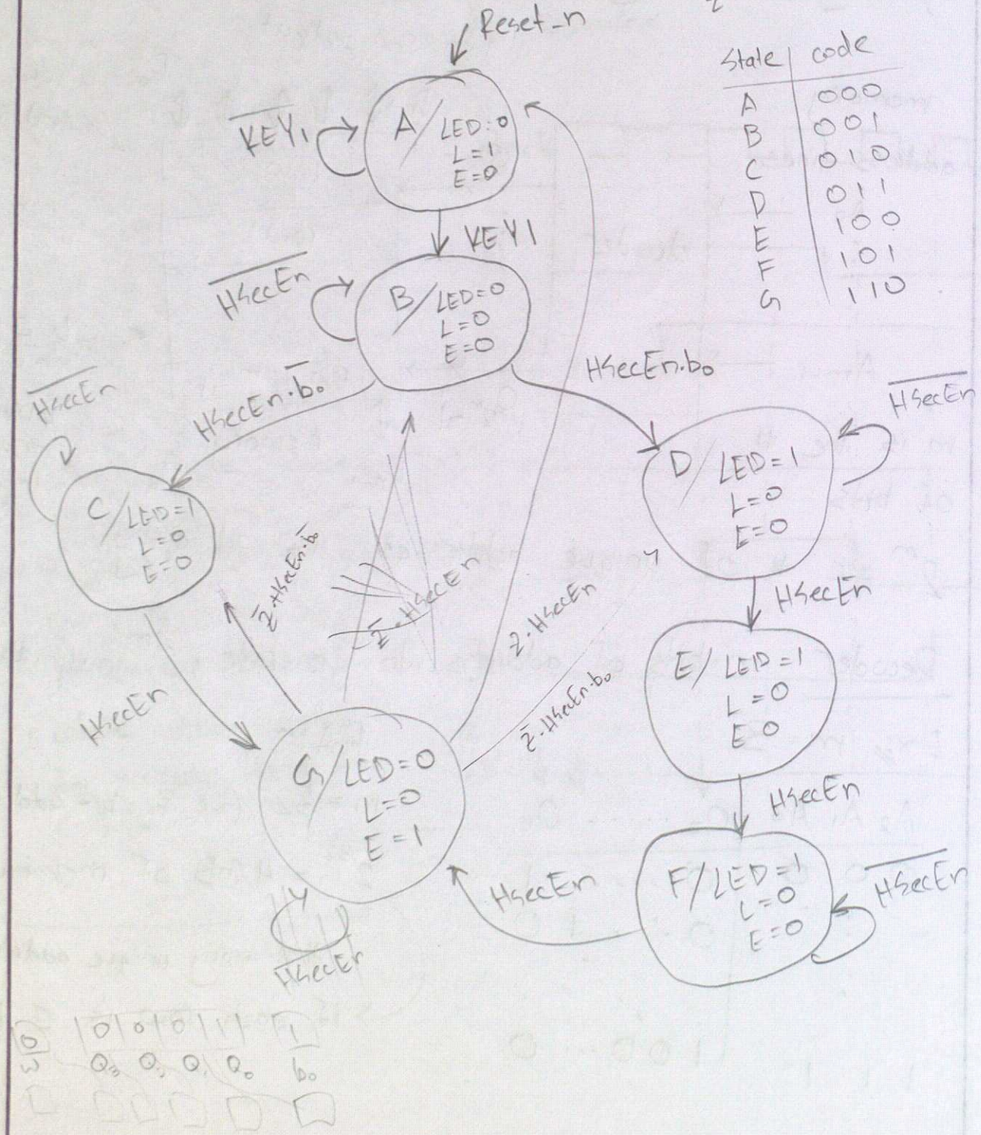
→ if each row is 1 byte

Lab 5 Part 3

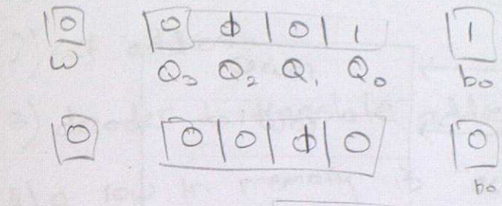


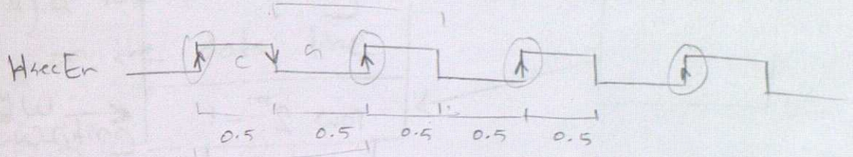
Z = everything zero

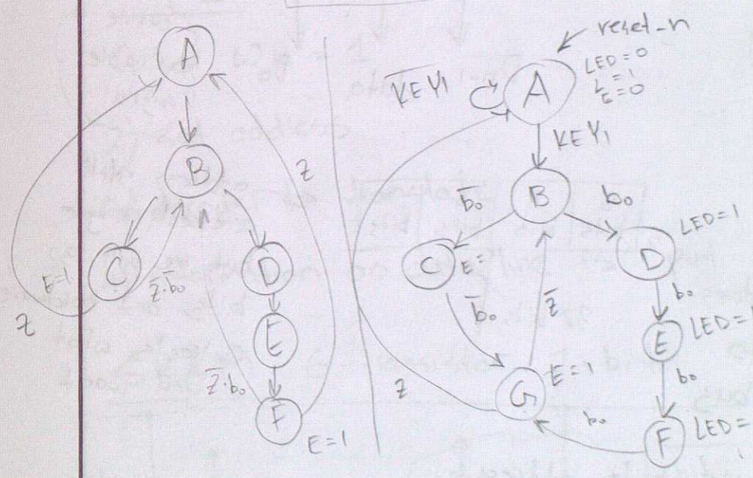
State	code
A	000
B	001
C	010
D	011
E	100
F	101
G	110

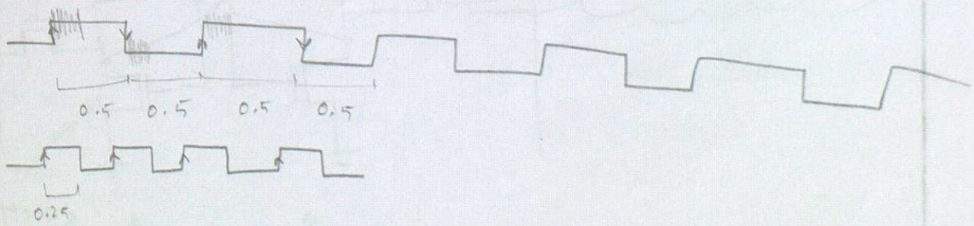


* when in G: you shift and downcount, then you go eighter



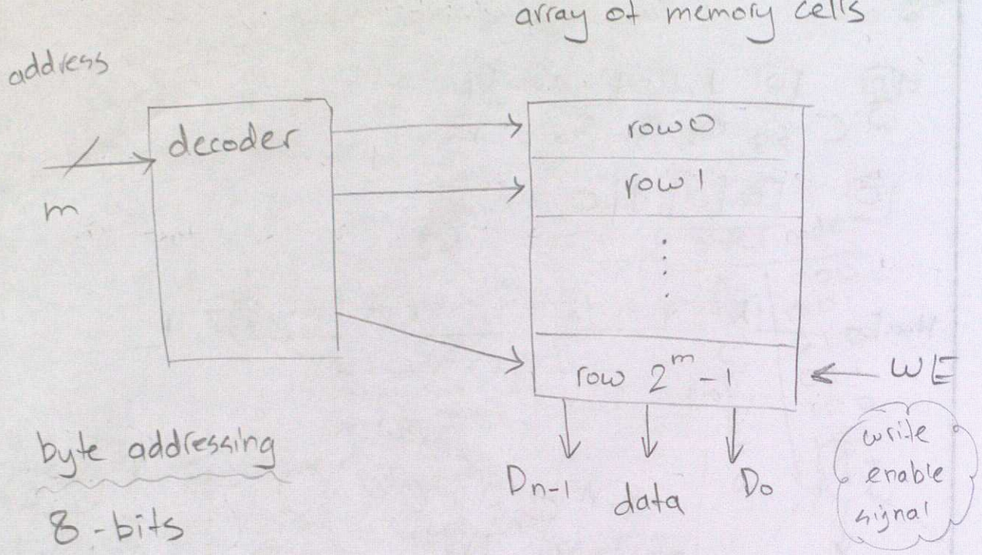






Week 9: Lecture 2

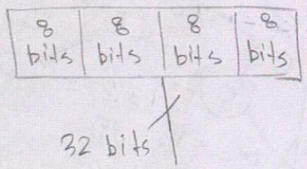
Memory Cont'd



byte addressing

8-bits

Another idea:



address still accessed a byte, but we pull 32 bits and hardware extracts what you want

byte addressing does not limit you to a certain data bus

row addressing vs. byte addressing

Reading

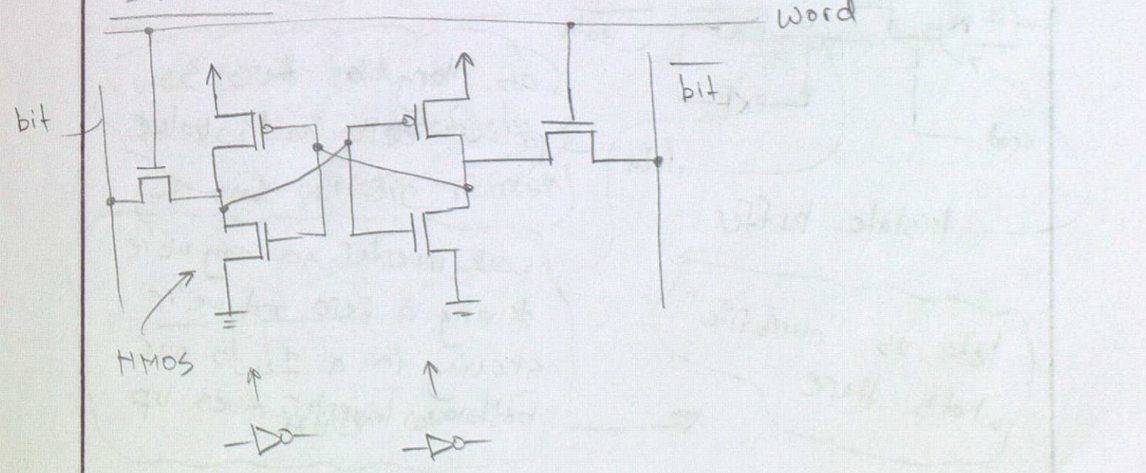
decoder is not clocked to anything

1) set WR = 0
2) set address
3) decoder to translate address to give you an O_x (output "x")
4) a row in memory is "activated" and data is on the data line

Writing

1) set WR = 1
2) set address
3) decoder to translate
4) information on data line is "put" into memory (ensure all these many bits have something (a 1 or a 0))

SRAM 6-transistor 1-bit

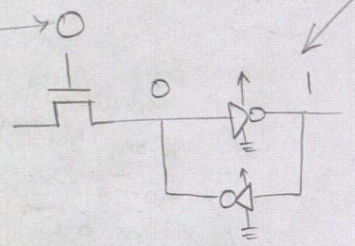


Memory Cell Array

1) storage cell ← individual bits
   (SRAM, DRAM)
   ↑ static ↑ dynamic
2) Tri-state buffer (0, 1, z) ← high impedance

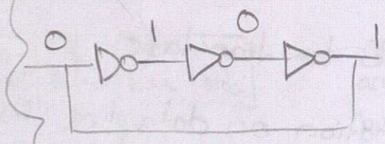
Storage Cell

output from decoder →

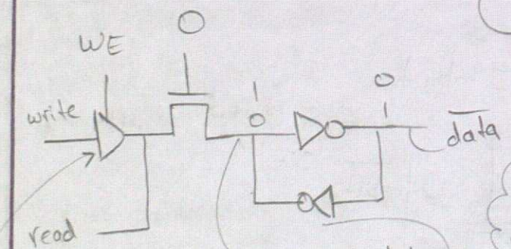


as long as inverters are powered, we hold values

Odd # of inverters



↑ ring oscillator



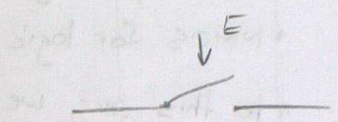
tristate buffer

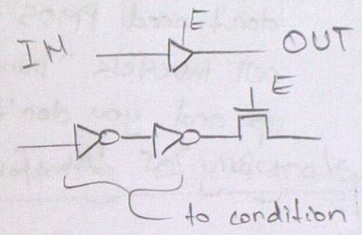
as long as there is power, cell holds value

weak inverter, so say we're driving a zero, and we're shoving in a 1, so our bottome inverter gives up

lets us "overwrite" what's there

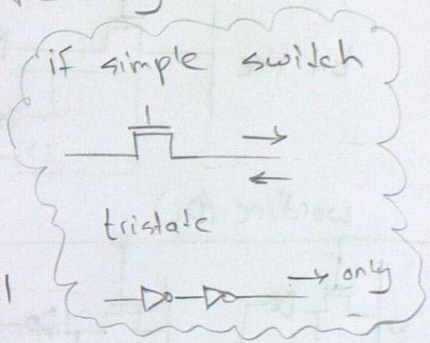
Tristate Buffer





to condition the signal

the two extra inverters make sure nothing goes the wrong direction

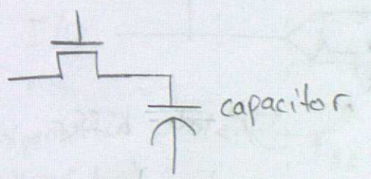


truth table

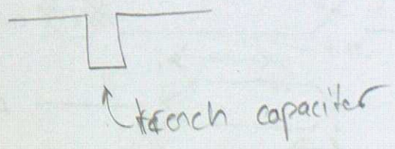
E	IN	OUT
0	x	high z
1	0	0
1	1	1

← impedence

DRAM

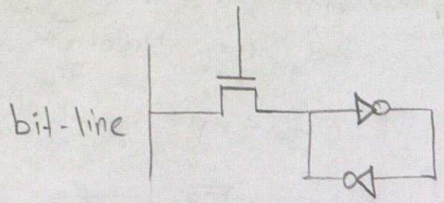


- we can't hold on to the value indefinetly (we need periodic refresh)

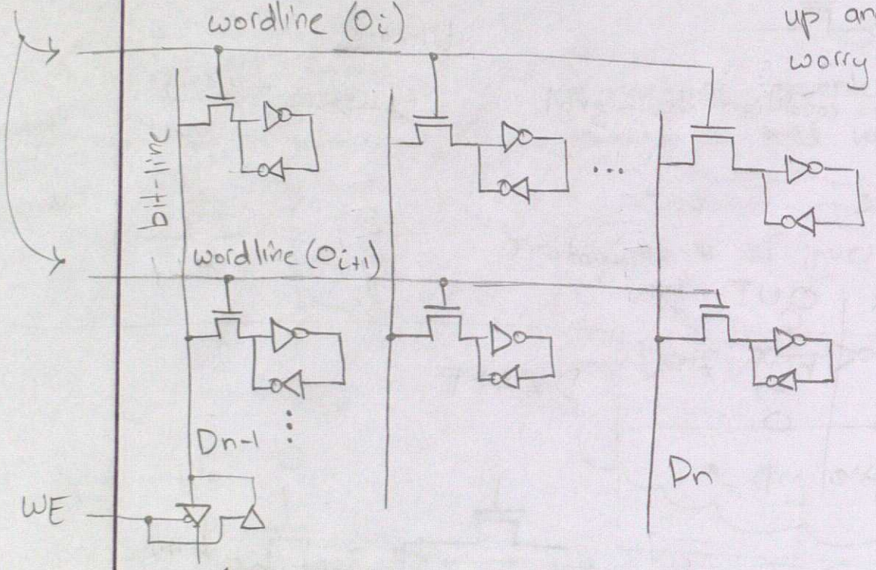


↑ trench capacitor

Array of Storage Cells

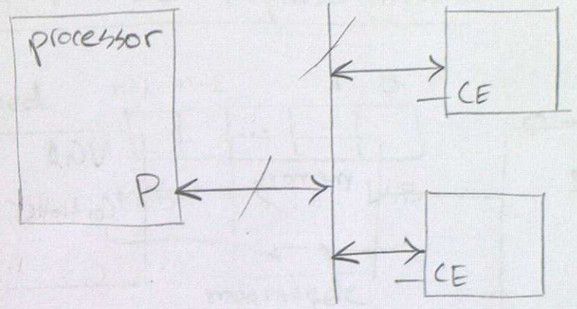


- PMOS for logic 1
- NMOS for logic 0
- in this case, we don't need PMOS since cell inverters "bump" up and you don't worry of $V_{threshold}$



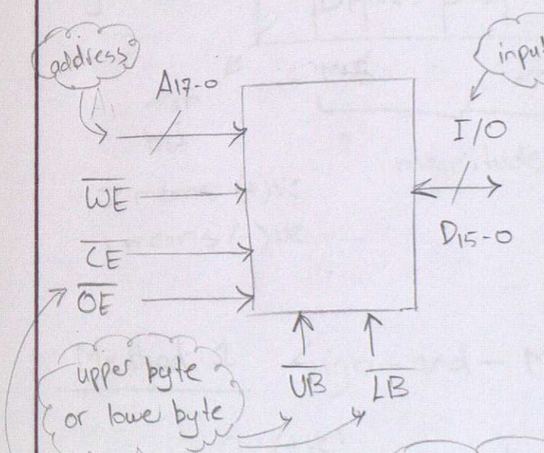
tristate buffer, we can make system less noisy as a bonus

* can replace this SRAM w/ inverters with DRAM (we'd have to add mechanism for periodic refresh)

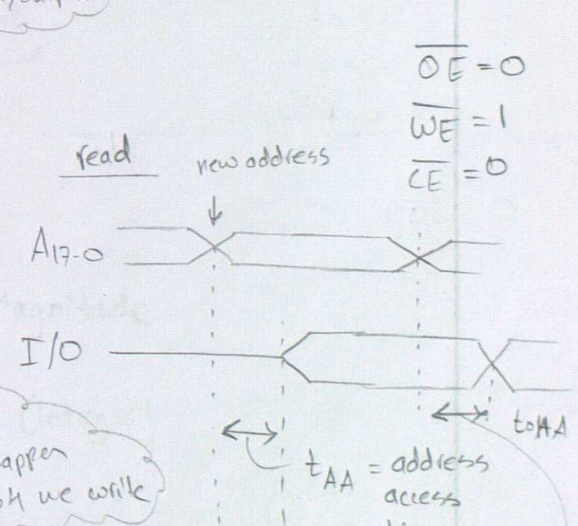


CE is a chip enable port

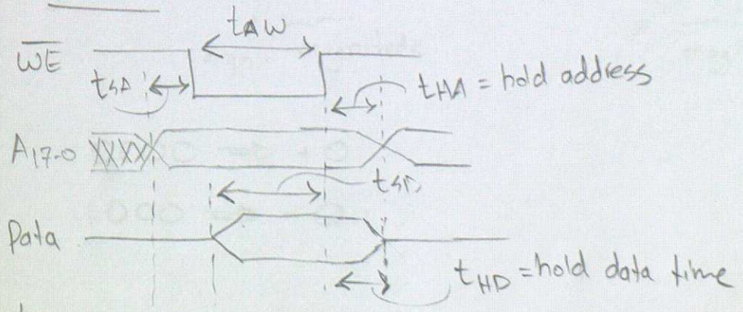
Timing of Signals



$\overline{OE} = 0$
$\overline{WE} = 1$
$\overline{CE} = 0$



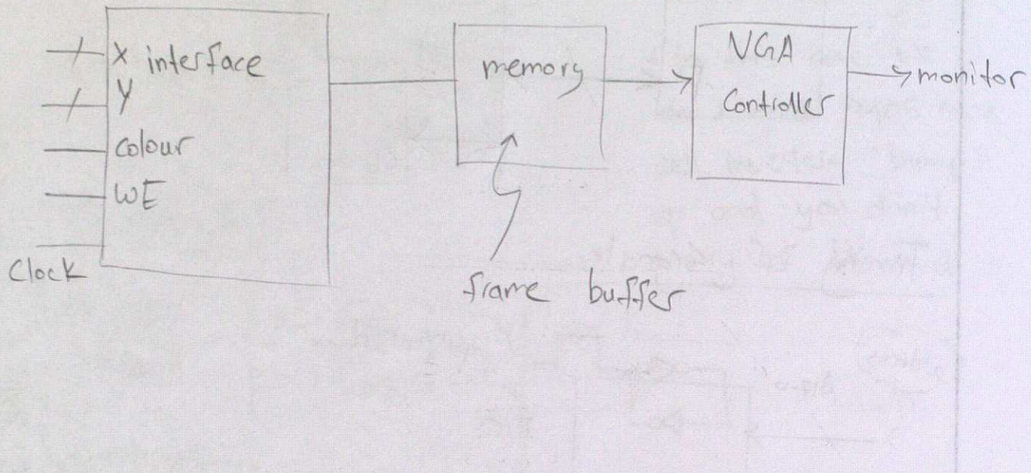
we want A17-0 to happen and settle b4 we write



t_{OHA} = output hold address

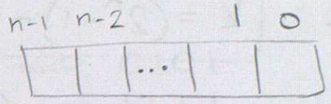
t_{SA} = setup address time, t_{AW} = address width time, t_{SD} = setup data time

VGA



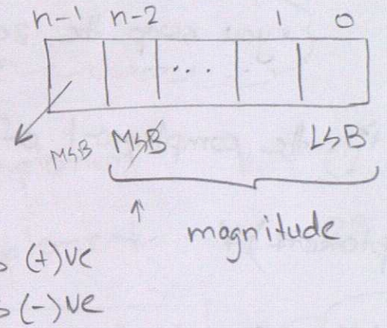
Signed Number Representation

unsigned



$0 \rightarrow 2^n - 1$

signed



0 means (+)ve
1 means (-)ve

Method 1 Sign-and-Magnitude

0 (+ve)
1 (-ve) magnitude (integer)

e.g. +5 ⇒ 0 101 (0 = sign, 101 = magnitude)

-5 ⇒ 1 101 (1 = sign, 101 = magnitude)

0000 ⇒ +0

1000 ⇒ -0

Method 2 (1's Complement)

negative number is K: $K = (2^n - 1) - P$ ← (+ve) number

ex// 4-bits

P = 0101 {+5}

K = $(2^4 - 1) - P$

base 10 = 15 - 5 = 10

binary

$(1111)_2 - 0101 = 1010$ {-5}

• still have 2 forms of zero

• you swap the zeroes and ones

* The 1's complement is the complement of each bit

Method 3 (2's Complement)

$K = 2^n - P$ ← the number you're trying to find the complement of

ex// P = +5

$K = 2^4 - P$

= 10000 - 0101

= 1011

0000	0
	1
	2
	3
	4
	5
	6
0111	7
1000	-8
1001	-7
1111	-1

math

5 + 1 = 6

5 + (-1) = 4 ⟹

```
  0101
+ 1111
-------
 10100
```

overflow doesn't really do anything ↑ this is 4

Observe

1's complement $K_1 = (2^n - 1) - P$

2's complement $K_2 = 2^n - P$

$\therefore \boxed{K_2 = K_1 + 1}$

take 1's complement and then add 1

Hand Calculation

* look at bits from right to left
* copy down all 0's and the first occurrence of 1
* then complement each remaining bit

bar everything, put a 1 at the right side

ex// -3 to 3

1101 → $\bar{1}\bar{1}\bar{0}1$ → 0011

↑ already in 2's complement

ex// 6 to -6

0110 → $\bar{0}\bar{1}\bar{1}0$

↓ 2's complement

1010

2's Complement to decimal

$(-2)^{n-1} + \cdots + 2^2 + 2^1 + 2^0$

ex// $(1011)_2 \longrightarrow -2^3 + 0 + 2^1 + 2^0 = -8 + 0 + 2 + 1 = -5$ ✓

Addition and Subtraction

Sign-and-magnitude

+ve number A
add +ve number B
+ve number A + number B

+ve number A
add -ve number B
→ subtract smaller magnitude from larger magnitude
→ correct the sign

Week 10: Lecture 2 Nov 14, 2024

1's Complement

1's complelement, so bar

+5	0101	-5	1010
+ +2	+ 0010	+ +2	+ 0010
+7	0111	-3	1100

0111: +ve, 7 — 1100: -ve, 3 → -3 1's complement

+5	0101	-5	1010
+ -2	+ 1101	+ -2	+ 1101
+3	1 0010	-7	1 0111

0010: +ve, 2; carry 1 → 0011: +ve, 3

0111 + carry 1 → 1000: -ve, 7 → -7 1's complement

2's Complement

+5	0101	-5	1011
+ +2	+ 0010	+ +2	+ 0010
+7	0111	-3	1101

0111 → +7; 1101 → -3

☆ →

+5	0101	-5	1011
+ -2	+ 1110	+ -2	1110
+3	10011	-7	11001 ✓

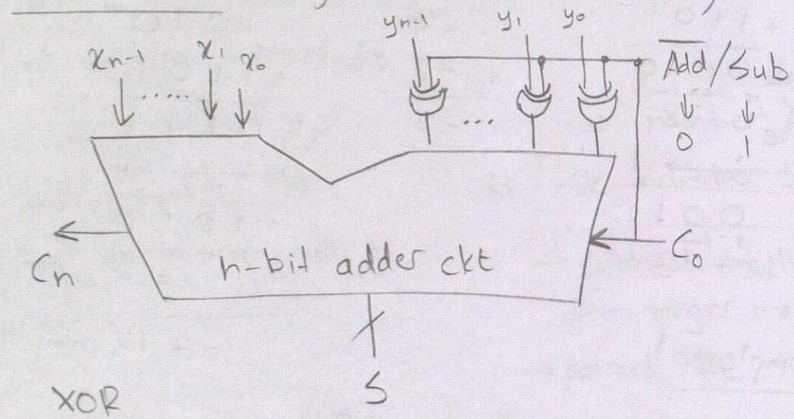
10011: +ve, 3 — 11001: (-)ve, -7

Subtract 2's Complement

"you can just add the negative number"

$$\begin{array}{r} +5 \\ -\ +2 \\ \hline +3 \end{array} \qquad \begin{array}{r} 0101 \\ -0010 \\ \hline \end{array} \longrightarrow \begin{array}{r} 0101 \\ +\ 1110 \\ \hline 10011 \\ \downarrow \\ +3 \end{array}$$

Adder CKT (right subtracted from left)



XOR

a	b	out
0	0	0
0	1	1
1	0	1
1	1	0

↑ think of b as selector bit

if b=0, it lets a thru (buffer)

if b=1, it inverts a (inverter)

Multiplication

unsigned integer

$$\begin{array}{r} 12 \\ \times 11 \\ \hline 132 \end{array} \qquad \begin{array}{r} 1100 \\ \times 1011 \\ \hline 1100 \\ 1100 \\ 0000 \\ 1100 \\ \hline 10000100 \end{array}$$

$2^7 + 2^2 = 132$

pp's for carry

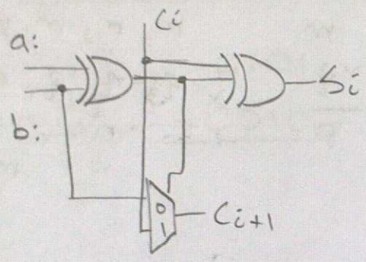
$$\begin{array}{r} m \\ \times q \\ \hline p \end{array} \qquad \begin{array}{r} m_3\ m_2\ m_1\ m_0 \\ \times\ q_3\ q_2\ q_1\ q_0 \\ \hline m_0 q_0 \end{array}$$

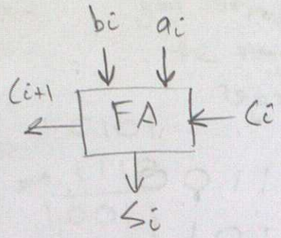
$$\begin{array}{rcccc} & & m_3 & m_2 & m_1 & m_0 \\ & \times & q_3 & q_2 & q_1 & q_0 \\ \hline \text{partial product 0} \rightarrow & & m_3q_0 & m_2q_0 & m_1q_0 & m_0q_0 \\ \text{for carry bit} \rightarrow & m_3q_1 & m_2q_1 & m_1q_1 & m_0q_1 & \\ \text{partial product 1} \rightarrow & PP1_5 & PP1_4 & PP1_3 & PP1_2 & PP1_1 \\ & m_3q_2 & m_2q_2 & m_1q_2 & m_0q_2 & \\ \text{partial product 2} \rightarrow & PP2_5 & PP2_4 & PP2_3 & PP2_2 & PP2_1 \\ & \vdots & & & & \end{array}$$

Week 10: Lecture 3

Nov 15, 2024

Adder Revisit





C_i	b_i	a_i	C_{i+1}	S_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

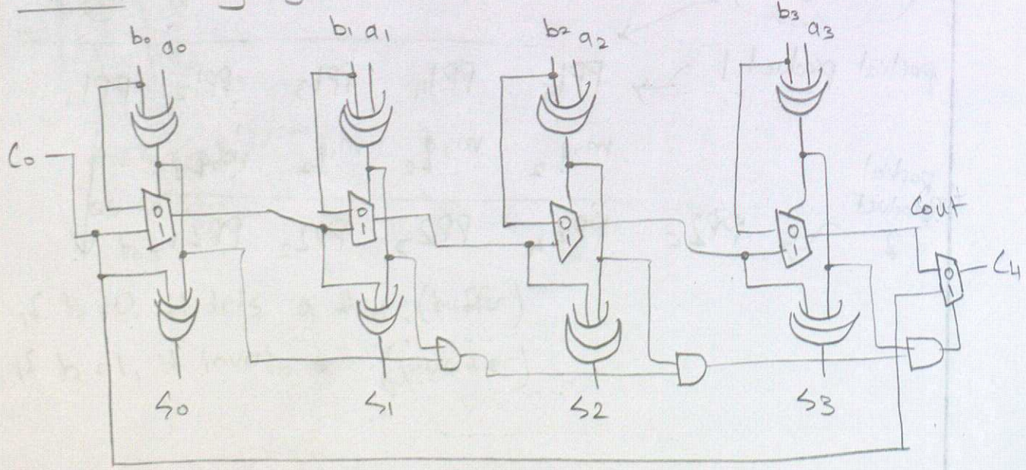
*highlight times when a and b equal

$a_i == b_i \rightarrow C_{i+1} = b_i$ or $= a_i$

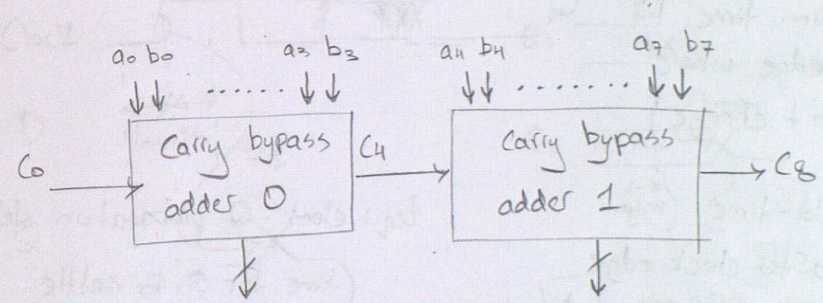
$a_i \neq b_i \rightarrow C_{i+1} = C_i$

- when a_i equals b_i, C_{i+1} is b_i (or C_{i+1} is a_i)
- when a_i is different from b_i, C_{i+1} is same as C_i

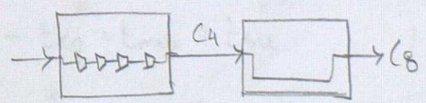
4-bit : Carry Bypass Adder



* iff all four pairs of $a_i b_i$ are different, then Cout is ∴ equal to C_0
* If we get a signal 1 at the last AND gate, then it means every $a_i b_i$ are different (due to XOR gate) so we can feed this the original C_0 for C_4 (this will make the thing a tad bit faster just for this one case of all $a_i b_i$'s being different) and this only speeds up "Cout" propagation (doesn't speed up the actual addition ... S_i's)

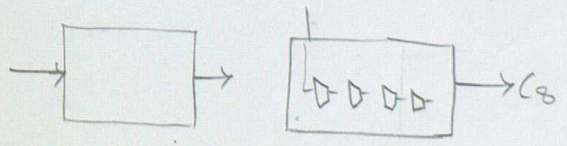


i) if all pairs of $a_4 b_4 \cdots a_7 b_7$ are different, $C_8 = C_4$,



any one of $a_0 b_0 \cdots a_3 b_3$ are equal (our trick from last page won't hold) so we must propagate

ii) If any pairs of $a_4 b_4 \cdots a_7 b_7$ are the same



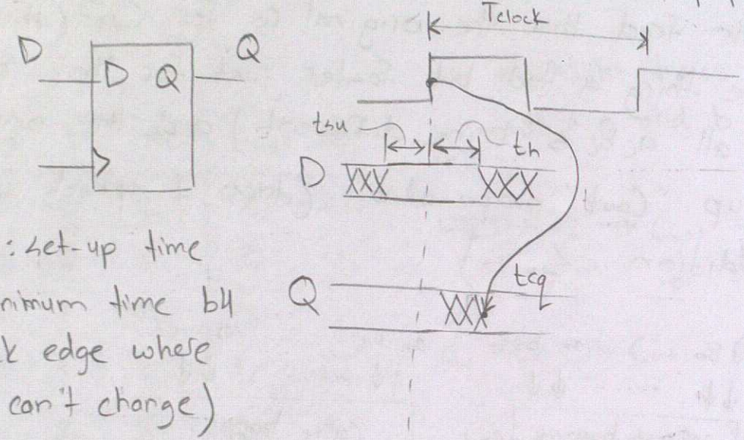
C_8 will make its own result, so C_8 independent of C_4

Timing Analysis

→ outcome is to determine the maximum clock frequency

Flip-Flop

+ rise time, hold time, propagation

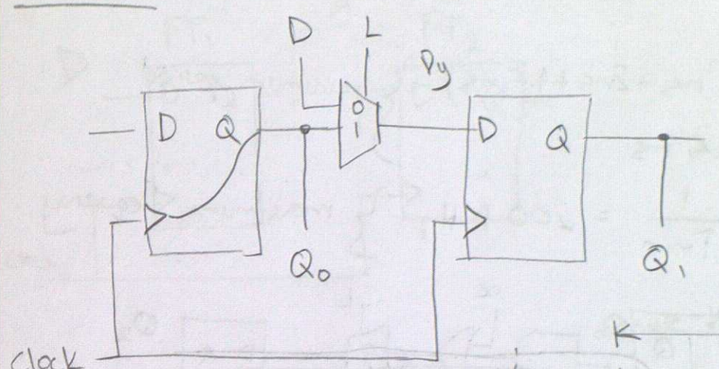


t_{su}: set-up time (minimum time b4 clock edge where D can't change)

t_h: hold-time (min time after clock edge where D can't change)

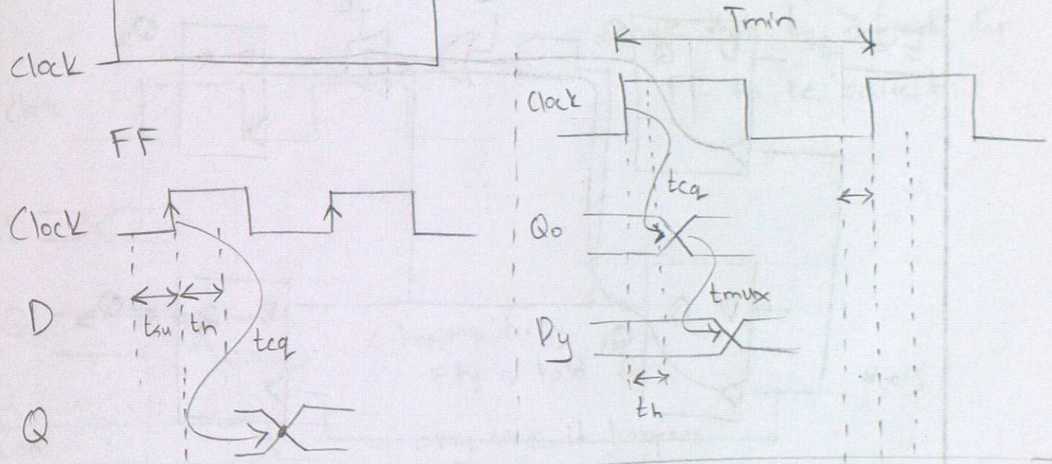
t_{cq}: clock-Q propagation delay (time for Q to settle and be valid)

Timing



if th is less than tcq, we aren't worried, but if th > tcq, we take th instead

FF



look for the critical path: the longest time

sometimes min/max

$T_{min} = t_{cq} + t_{mux} + t_{su}$

$t_{cq} > t_h$

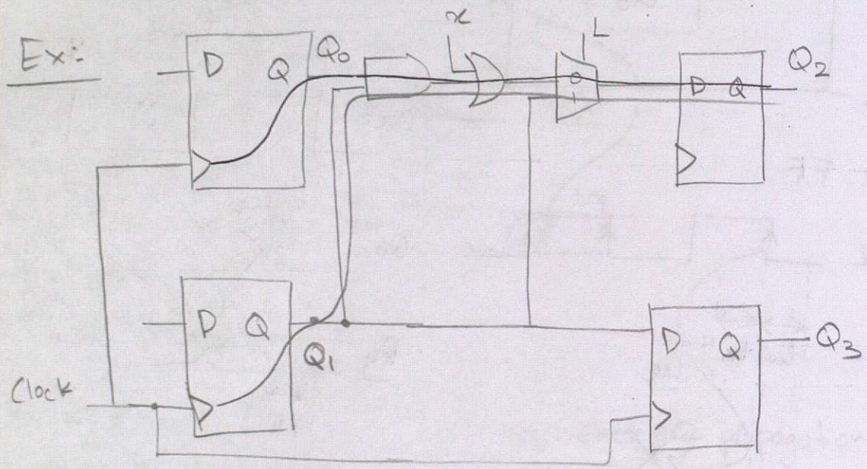
in general, $t_{cq} > t_h$

Ex: $t_{cq} = 1.5ns$, $t_{su} = 1ns$, $t_{mux} = 2ns$

$T_{min} = (1ns + 2ns + 1.5ns)$ } minimum period
$= 4.5ns$

$f_{max} = \frac{1}{T_{min}} = 200\,MH_z$ } maximum frequency

Ex:



Determine fastest frequency (look for T_{min})
look for the critical path

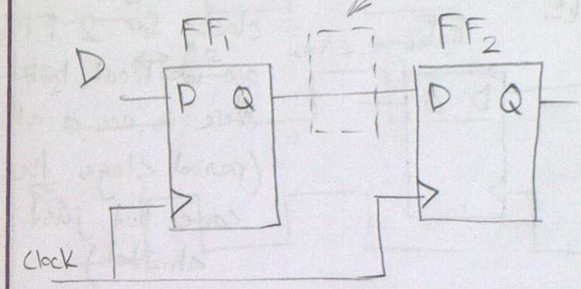
A: → we have 2 equally long paths

$Q_0 \rightarrow Q_2$
$Q_1 \rightarrow Q_2$ } $T_{min} = \underbrace{t_{cq} + t_{su}}_{\text{FF's}} + t_{AND} + t_{OR} + t_{mux}$

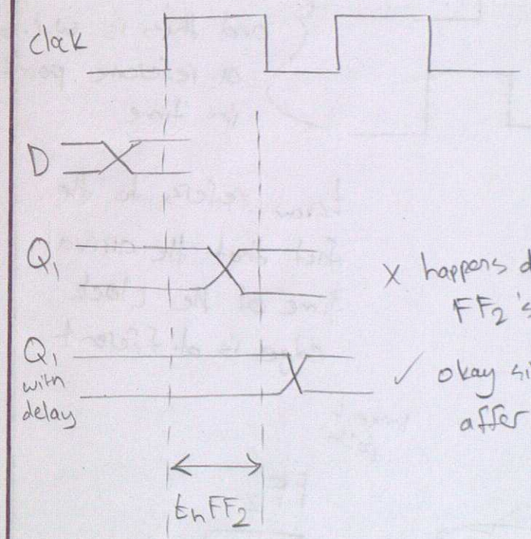
☆ t_{su} from Q_2?

Hold Time Violation

you add a comb. ckt to have a delay



ensure that $t_{cq} \geq t_{hold}$ for FF to be correct



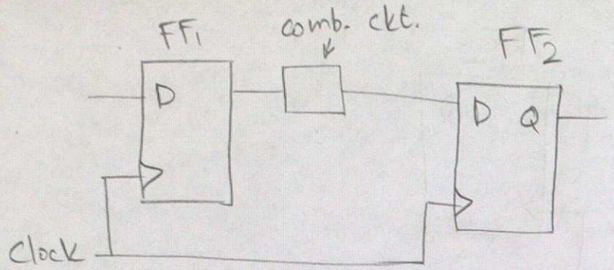
X happens during FF_2's hold time

✓ okay since it happens after FF_2's hold time

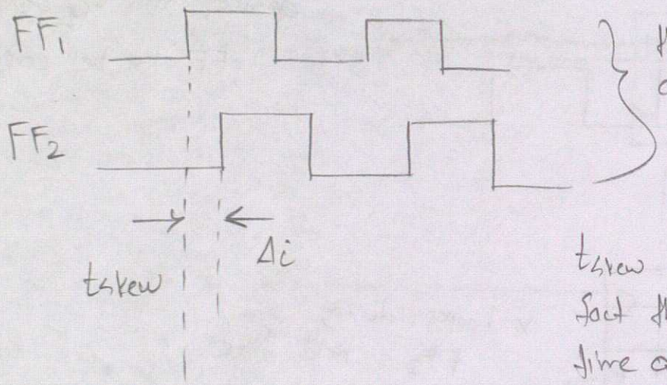
* Q_1 updated faster than it should have

* FF_2 picks up not the previous value of FF_1, but the current

Clock Skew



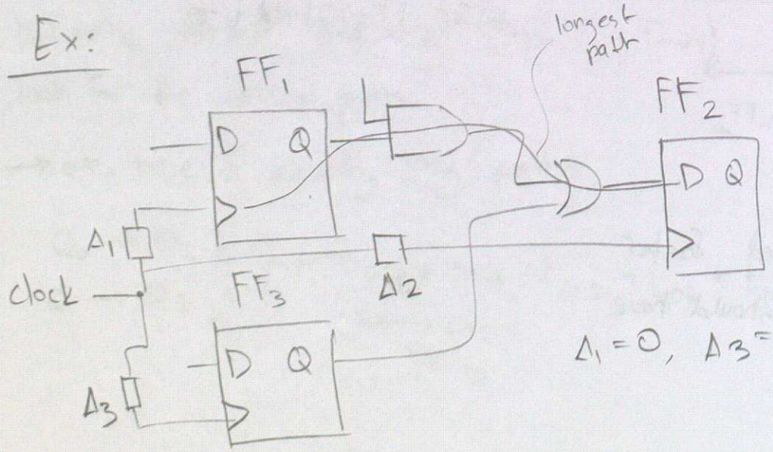
we usually assume clocks for 2 FF's are identical, but there is acc a shift (period stays the same, but just shifted)



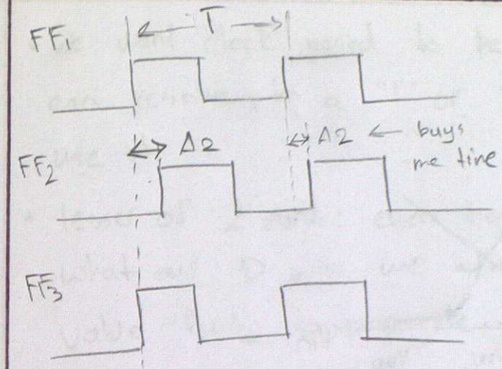
there is a shift, and this is w.r.t a reference point in time

t_{skew} refers to the fact that the arrival time of the clock edge is different

Ex:



$\Delta_1 = 0$, $\Delta_3 = 0$, $\Delta_2 > 0$

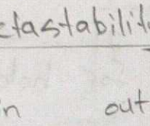


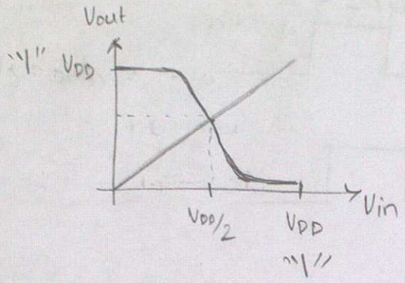
$T = T_{min} - t_{skew}$

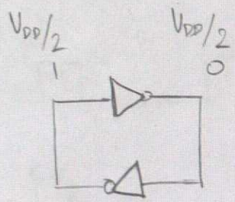
also check for hold-violations

Race Condition

Metastability



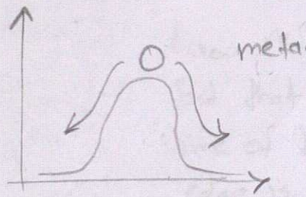


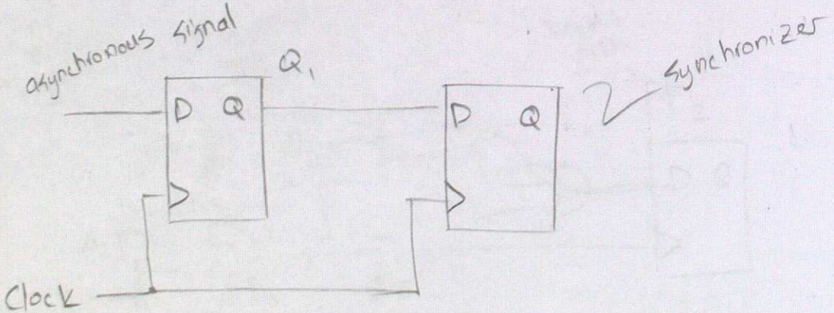


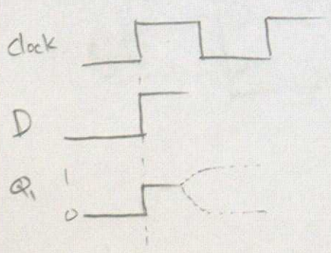
- putting 1 stores a zero
- putting $V_{DD}/2$ stores $V_{DD}/2$, so we don't know the logic level

the case our input is $V_{DD}/2$

- we might go either way, so hence "metastable"







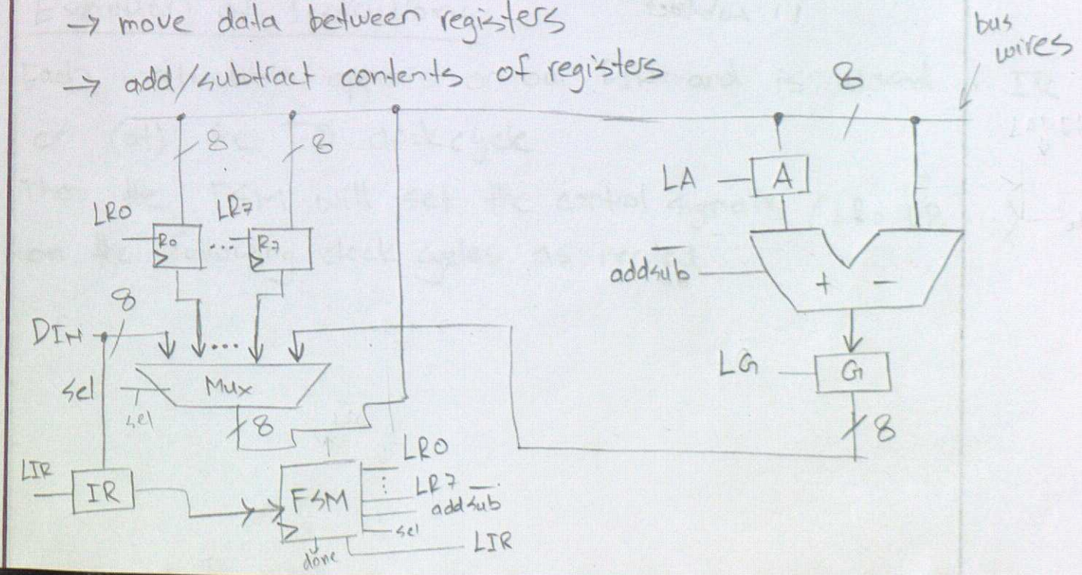
if Q_1 captures D at $V_{DD}/2$ for example, it may go up to 1 or go down to 0 (metastable)

- we want clock period to be long enough so that Q_1 can resolve to a "1" or "0" before second FF can use it.
- lesser of 2 evils: even though we didn't really know what our D was, we still don't want the nonsensical value $V_{DD}/2$ to propagate through the ckt, so we just wait till it stabilizes (use 2nd FF as "synchronizer")

Processor (platform dependent)

→ carry out operations
→ moving and processing data
→ uses step-by-step instructions (machine code)

Ex: → datapath is 8-bits
→ 8 registers: R0, R1, R2, ..., R7
→ inport (input port that can load data into registers)
→ move data between registers
→ add/subtract contents of registers



Instructions

Symbol $[R_y]$ ≡ contents of register R_y

0. mv R_x, R_y // move $R_x \leftarrow [R_y]$
1. mvi R_x, #D // $R_x \leftarrow$ #D ← D is a sequence of numbers from dataline (move immediate)
2. add R_x, R_y // $R_x \leftarrow [R_x] + [R_y]$
3. subtract R_x, R_y // $R_x \leftarrow [R_x] - [R_y]$

Binary Representation (defines Opcode, 8-bits,

II XXX YYY
instruction R_x R_y

↳ 00 mv
01 mvi
10 add
11 subtract

- we have 8 registers, so we need 3 bits to define registers

Ex: copy contents of R_4 into R_2

mv R2, R4 ↔ 00010100
"assembly" "machine code"

Ex: mvi R3, #D — assume that when the move immediate code is read, the processor can next read #D from DIN

Terminology

* mv, mvi, add, sub ← these are assembly language instructions. (note: these are specific to the processor platform
* The encoding of the instructions is the opcode
* Assembly code to get us to machine code

Execution of Instructions

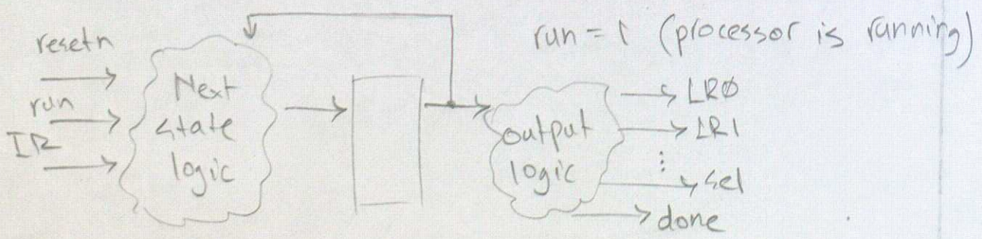
* Each instruction appears on our DIN and is stored in IR on (at) the T0 clock cycle
* Then the FSM will set the control signals ($LR_0, LR_1, \ldots$) on the following clock cycles as needed

Ex:

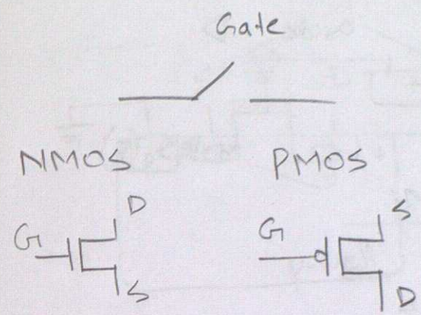
Instruction	T0	T1	T2	T3
mv	LIR	Sel=Ry, LRx=1, done		
mvi	LIR	Sel=DIN, LRx=1, done		
add	LIR	Sel=Rx, LA=1,	Sel=Ry, addsub=1, LG=1,	Sel=G, LRx=1, done
Sub	LIR	Sel=Rx, LA=1,	Sel=Ry, addsub=0, LG=1,	Sel=G, LRx=1, done

FSM

additional inputs: clock, resetn, run=0 (processor is stopped)
run=1 (processor is running)



Transistors and Logic Gates



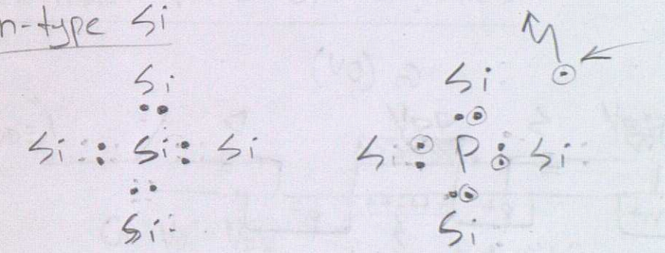
electron (n) channel

hole (P) channel

MOSFETs: Metal Oxide Semiconductor Field-Effect Transistor

- single power supply
  - VDD → "1"
  - Gnd → "0"

n-type Si

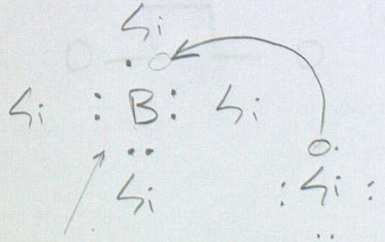


4 bonding electrons from each Si

P has 5 electrons

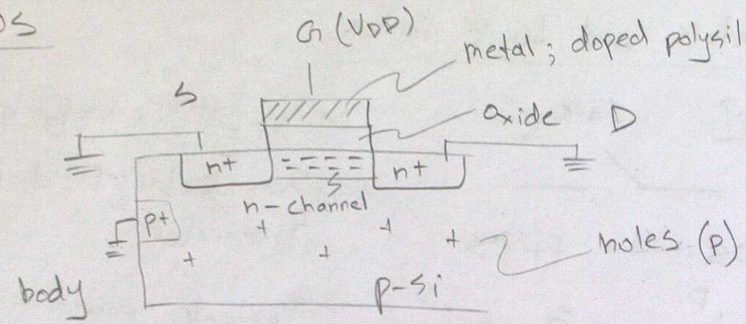
thermal energy 25meV releases this electron and this P is ionized

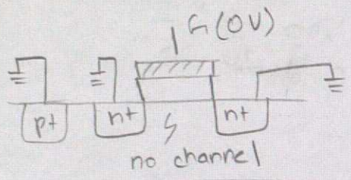
P-type Si



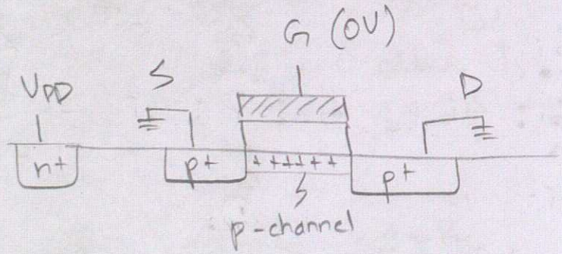
ionized -ve

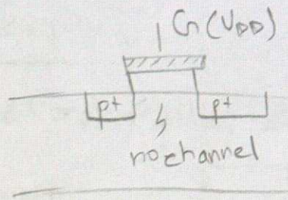
NMOS



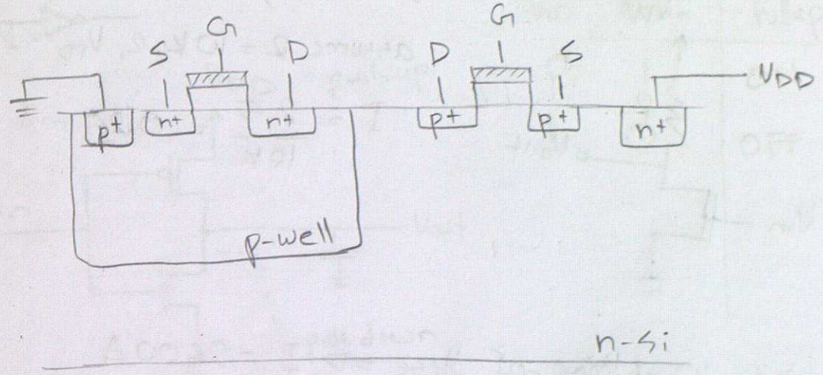


PMOS

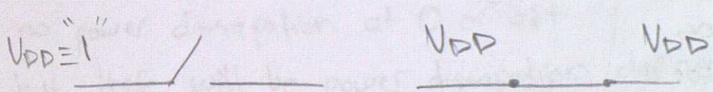


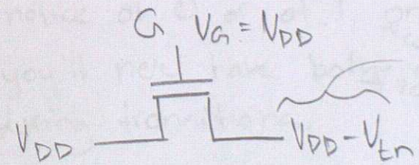


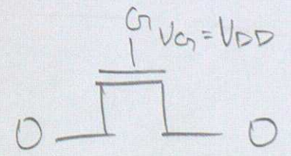
CMOS : Complementary MOS



Transistor Pass Characteristics



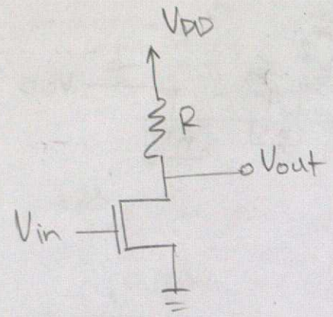




✓ this works

- NMOS best at passing 0
- PMOS best at passing 1

Aside



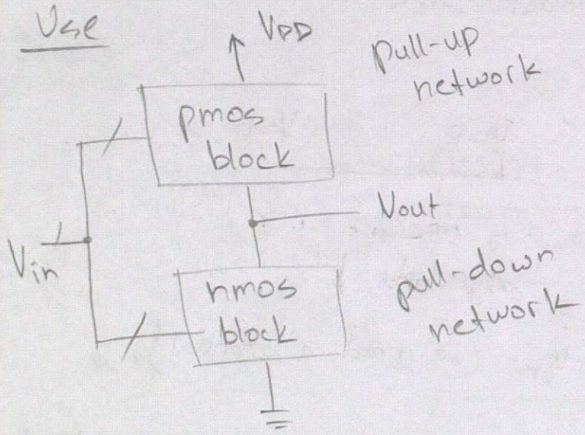
when ON;

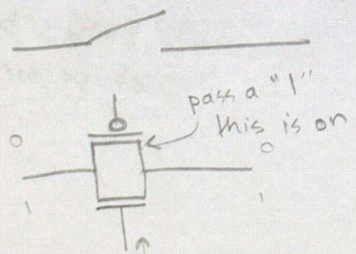
assume $R = 10K\Omega$, $V_{DD} = 2.5V$

$$I = \frac{2.5}{10K} = 0.25\,mA$$

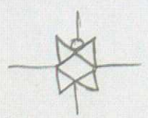
imagine 10 million of these $\Rightarrow I = 2500\,A$

Use

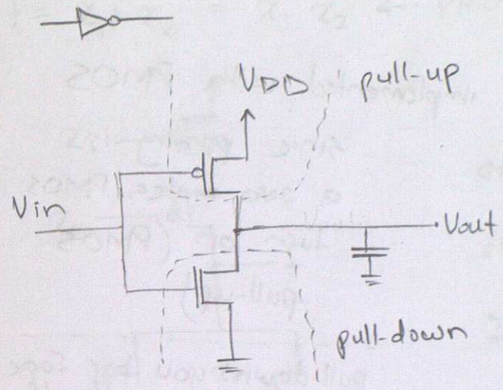




transmission gate



Inverter (NOT Gate)



Vin	Vout	Pullup	Pull-down
0	1	ON	OFF
1	0	OFF	ON

0 ≡ Gnd
1 ≡ VDD

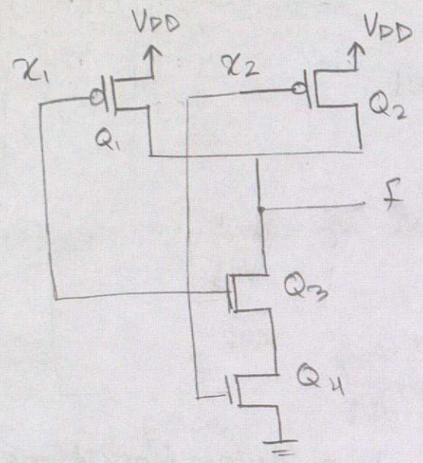
→ "0" turns on PMOS
→ PMOS passes V_{DD} (or 1)
→ no power dissapation at 0 or at 1
→ but there will be power dissapation during transitions
→ notice at 0 or at 1, only one network is ON.
→ you'll never have both off, and both are ON only during transitions.

NAND Gate

+ "OR" transistor in ∥ parallel
+ "AND" transistor in ↔ series

$f = \overline{x_1 x_2} = \overline{x_1} + \overline{x_2}$ ← parallel

can be implemented with PMOS



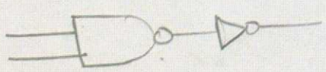
since passing in a zero makes PMOS turn on (PMOS pull-up)

pull-down: you bar func.

$\overline{f} = \overline{\overline{x_1 x_2}} = x_1 x_2$ ← series

x_1	x_2	which Q on/off? Q_1 Q_2 Q_3 Q_4	f
0	0	on on off off	1
0	1	on off off on	1
1	0	off on on off	1
1	1	off off on on	0

AND



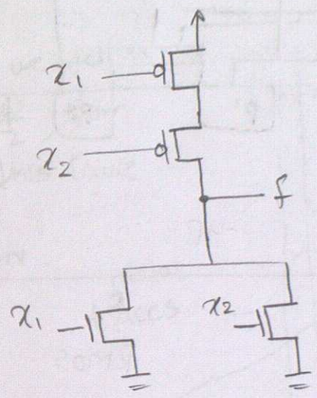
NOR Gate

$f = \overline{x_1 + x_2} = \overline{x_1}\,\overline{x_2}$ ← PMOS Pull-up (in series since anded)

now for pull-down:

$\overline{f} = \overline{\overline{x_1 + x_2}} = x_1 + x_2$

NMOS pull-down (in parallel since "or-ed")



Complex Gates : $f = \overline{x_1} + \overline{x_2}\,\overline{x_3}$ ← already bared, so pullup

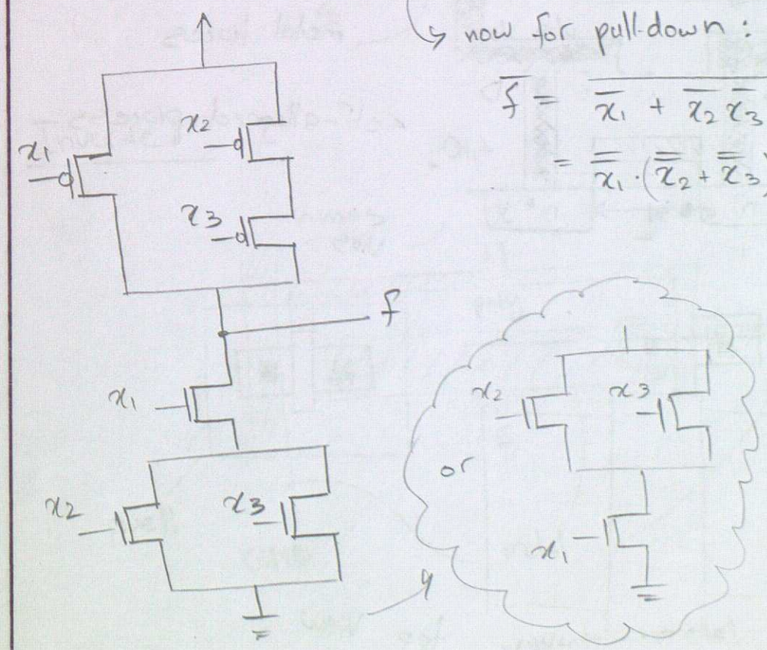
↳ now for pull-down : bar f

$\overline{f} = \overline{\overline{x_1} + \overline{x_2}\,\overline{x_3}}$

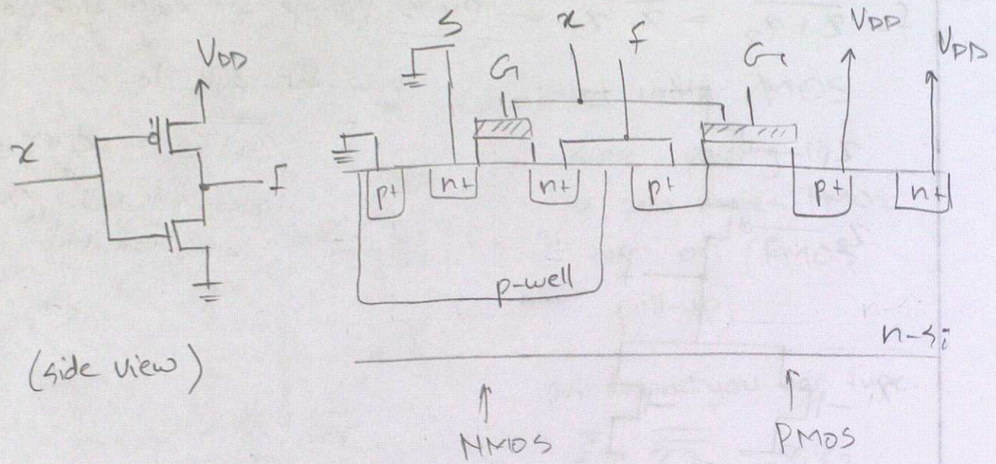
$= \overline{\overline{x_1}} \cdot \left(\overline{\overline{x_2} + \overline{x_3}}\right) = x_1 \cdot (x_2 + x_3)$

in parallel

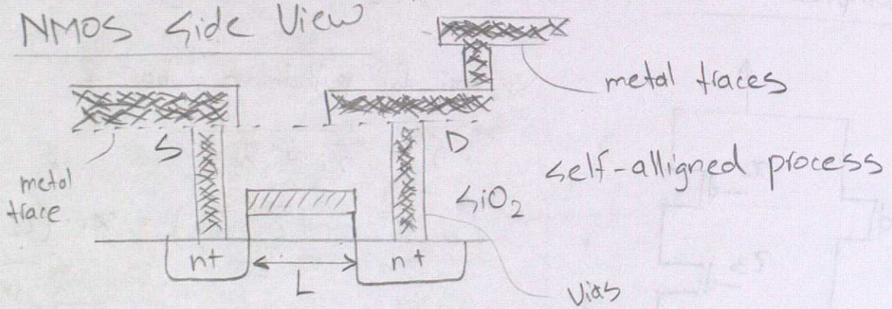
in series

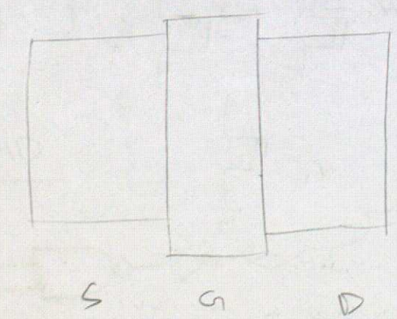


Timing Characteristics of transistors / gates



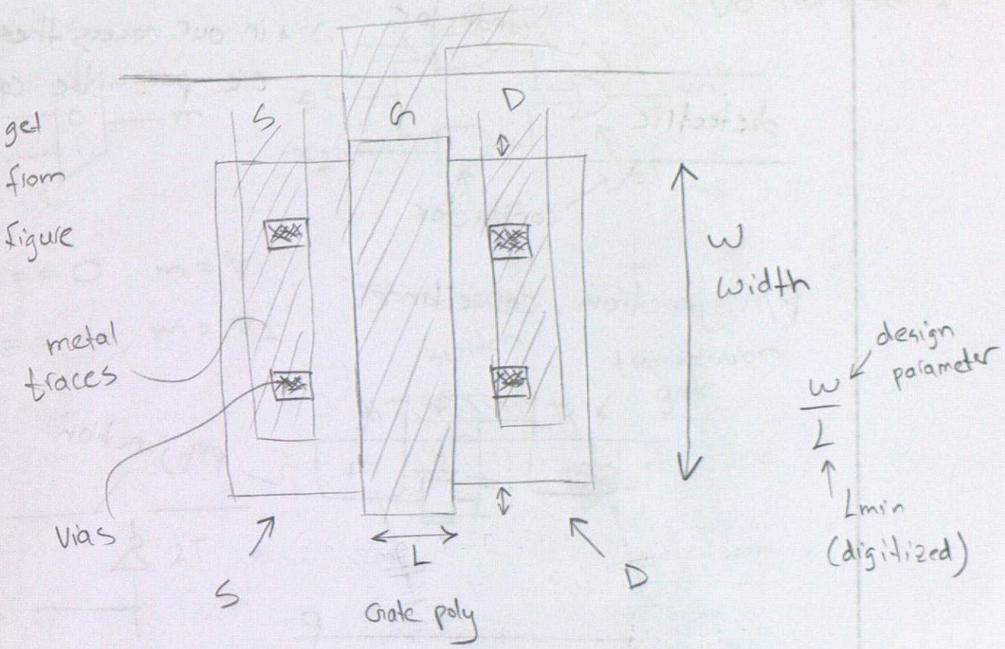
NMOS Side View



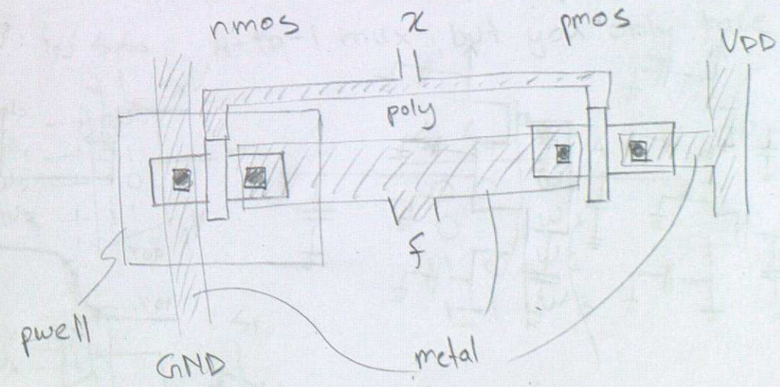


NMOS Top-View

* we get $(\frac{W}{L})$ from this figure



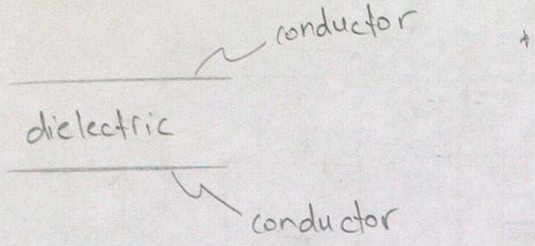
Inverter



* current → metal
* potential → polysilicon

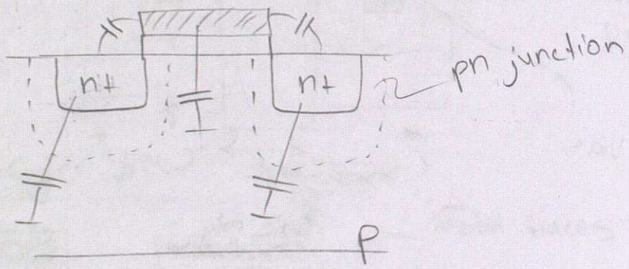
* missing are the body contacts

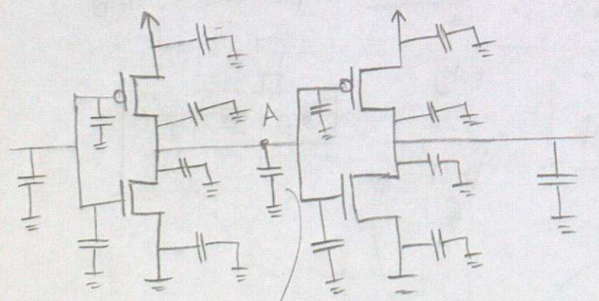
Capacitive Effect (ECE110)



+ in our cases, these are parasitic capacitors

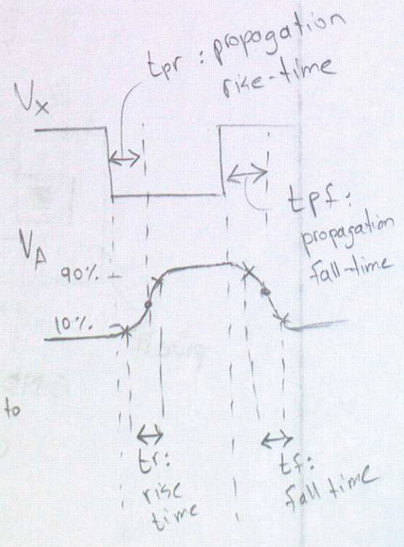
P/n junction capacitance



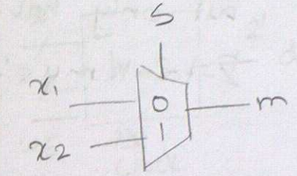


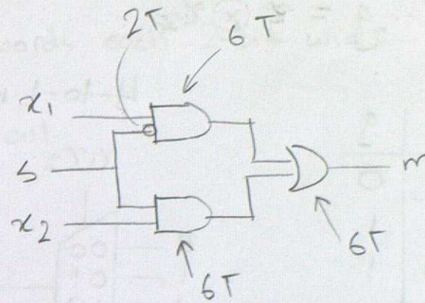
total capacitance of source/drain and source

delay to charge up for RC



Multiplexer

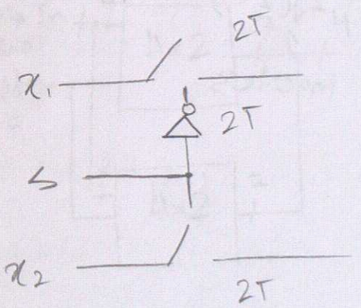


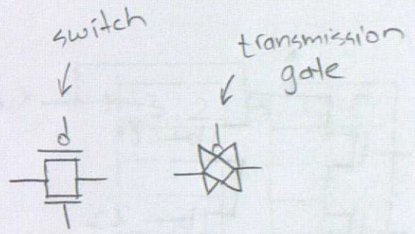


20 Transistors

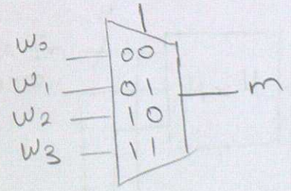
s == 0 , m = x1

s == 1 , m = x2

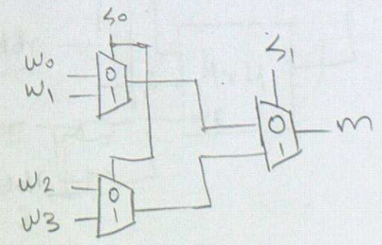




s1 s0 4-to-1 mux, but you only have 2-to-1 mux



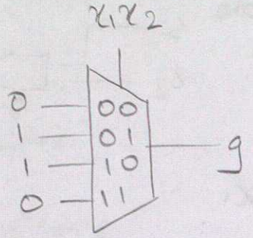
s_1 s_0	m
0 0	w_0
0 1	w_1
1 0	w_2
1 1	w_3

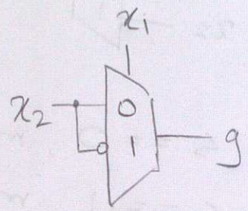


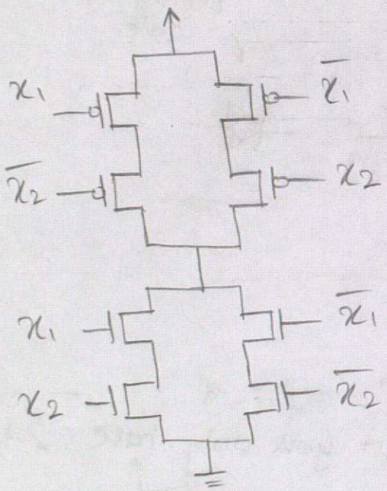
XOR $g = x_1 \otimes x_2$

x_1	x_2	g
0	0	0
0	1	1
1	0	1
1	1	0

4-to-1 mux, but only have 2-to-1 mux's?





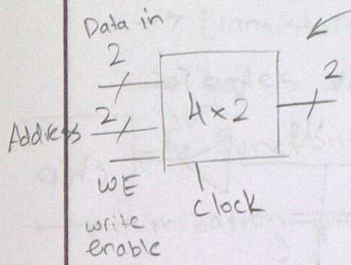


$8T + 2(2T) = 12$ Transistors

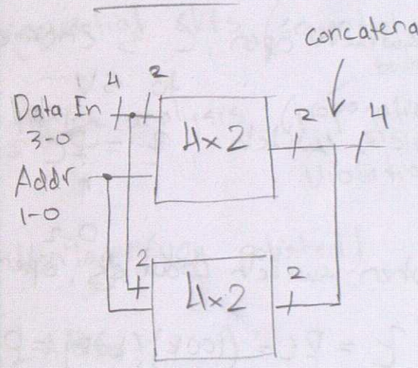
↑ inverters

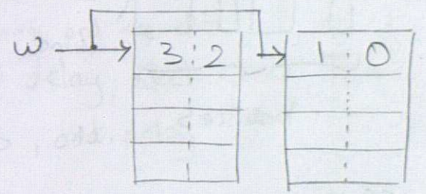
Memory

4 words, each 2-bit wide (8-bit memory)



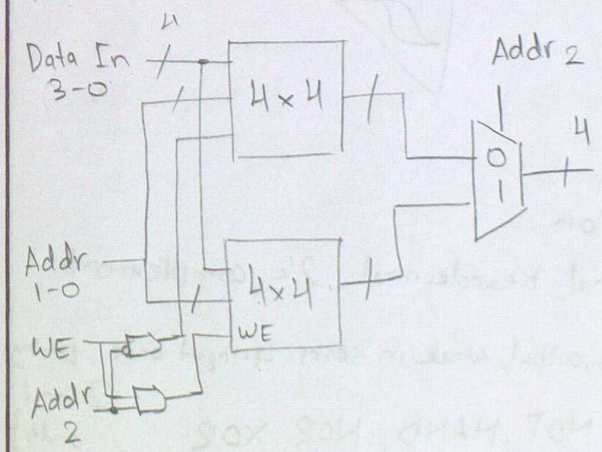
4×4: 4 words, each 4-bit wide (16-bit memory)





+addressing hasn't changed, only size of each

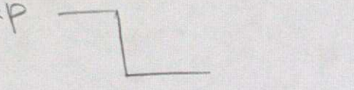
8×4; 8 words, each 4-bit wide



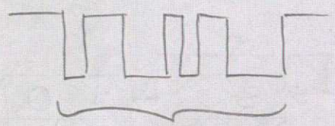
Debounce

mechanical switch

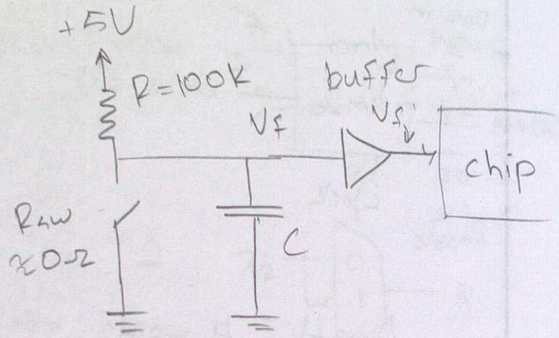
exp



observed



bounces



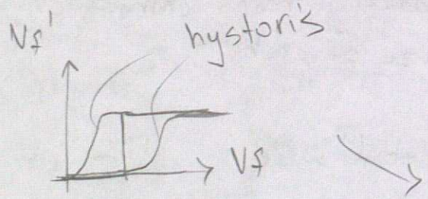
switch open, C is charged to 5V

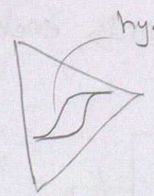
press switch, $\tau = RC \approx 0\,sec$ ($R \to 0\Omega$)

when switch bounces, open

$\tau = RC = (100k)(1\mu F) = 0.1\,sec$

buffer





Final Exam

- number representation
  - → binary, decimal, hexadecimal, 2's complement
- Logic Gates (input, output, stack in series, 4-input ands using 2 input ands)
  - → AND, OR, NOT, NAND, NOR, XOR
- CMOS (NANDS, NORs, NOT, ~XOR)
  - → transistors
  - → gates
  - → functions
- minimization procedures
- combinational ckts (no sequetional aspects, no clock edge to capture) → not edge-triggered
- sequential ckts (sequential elements)
- timing analysis (propagation delay (both comb. and seq. ckt.), clock skew, Tmin (min clock period, max freq.), violations, add delay
- Verilog (very polished)
- LABS
- FSM's