12.4 Feynman

Combinational Systems Intro

If y is the same, the output will be the same

To show inputs and outputs, we have a truth table:

y	f(y)
0	0
1	0

y	f(y)
0	1
1	1

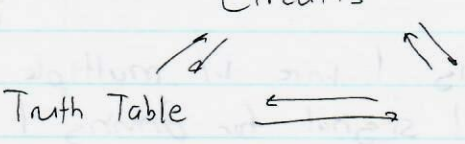
y	f(y)
0	0
1	1

← identity function.

All possible inputs: increasing period based on power of 2.

1 var alternates every 1 input, 2nd alternates every 2 inputs, 3rd var alternates every 4 inputs...

→ # of input combinations: 2^n (n = # of variables)



Logic Gates

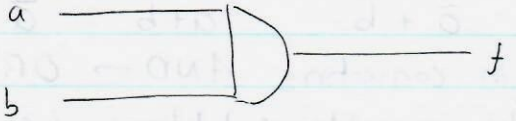
- Buffer:

Propagates

- Inverter / NOT:

Inverts input (0→1, 1→0)

- AND:



a	b	f
0	0	0
0	1	0
1	0	0
1	1	1

- OR:

a	b	f
0	0	0
0	1	1
1	0	1
1	1	1

→ Can take n inputs.

- NAND: not-and:

a	b	ab	$\overline{ab}$
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

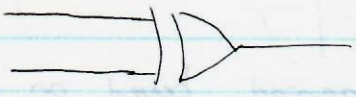
⇒ Inverting AND output

- NOR: not-or

a	b	a + b	$\overline{a+b}$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

⇒ Inverting OR output

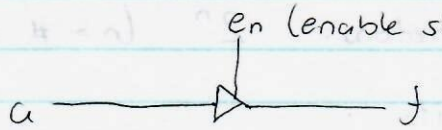
- XOR:



a	b	a ⊕ b
0	0	0
0	1	1
1	0	1
1	1	0

  - By extension, understand NXOR

- Tristate buffer:



en	a	f
0	0	disconnect
0	1	disconnect
1	0	0
1	1	1

Useful when driving 1 wire w/ multiple signals
↳ Only use 1 signal for driving 1 wire. This helps us control which signal drives the wire.

BOOLEAN ALGEBRA

- Know basic rules
- Tips:
  - Add in terms: Use 0+1 rule ($a\bar{a} = 0$, $(a+\bar{a}) = 1$)
  - Additive distribution: $a + bc = (a+b)(a+c)$
  - De Morgan's:
    $$\overline{ab} = \bar{a} + \bar{b}, \quad \overline{a+b} = \bar{a}\,\bar{b}$$
    - Useful in converting AND ↔ OR
  - Factoring. Use variable substitution: i.e. ⇒ $X = b \oplus d$
    ↳ Helps find patterns!
- Literals:

x ⇒ positive literal, $\bar{x}$ ⇒ negative literal.

Circuit Cost

- Rules of calculating cost
  - Inversion are free if on the input
    - Ex: $\bar{a}$ is a cost 0, $\overline{(a+bcd)} \Rightarrow$ cost $\neq 0$
  - Each logic gate is 1
  - Each input on the logic gate is 1.

→ Bigger + more gates is bad.

- Ex: //

$$f = a + cd + ab + !(!(cd) + a)$$

① ② ④ ③ ⑤

①: AND gate, 2 inputs: $1+2=3$

②: AND gate, 2 inputs: $1+2=3$

③: Not gate (1-input), AND gate (2 inputs) = (Not) $(1+1)+(1+2)=5$

④: Not gate (1-input): 2

⑤: OR gates: $3+3+3+3=12$

Easier way:

1. Go through each type of input gate.

   2 input AND = $1+2=3 \Rightarrow 3\times$

   ~~2~~ 1 input NOT = $1+1=2 \Rightarrow 2\times$

   2 input OR = $1+2=3 \Rightarrow 1\times$

   4 input OR = $1+4=5 \Rightarrow 1\times$

2. Compute sum

   Cost = $(3\times3)+(2\times2)+3+5 = 9+4+8 = 21$

Circuit Synthesis

Circuits

Sum of Products (SOP) ⟷ Product of sums (POS)

$f = ab + bcd + \bar{a}c$ $\qquad$ $f = (a+b)(c+\bar{d})$

Hilroy

Sum of Products

1. Truth table + identify 1 and 0 in answer column.

a	b	c	f	
0	0	0	0	m_0
0	0	1	1	m_1
→ 0	1	0	1	
0	1	1	0	
1	0	0	0	
→ 1	0	1	0	
1	1	0	0	
1	1	1	1	m_7

2. Minterms: algebraic representation of each row.
   For each row, complement variable if 0. Otherwise keep same.

   Ex: // Row 3: $\bar{a}b\bar{c}$

   Row 6: $a\bar{b}c$

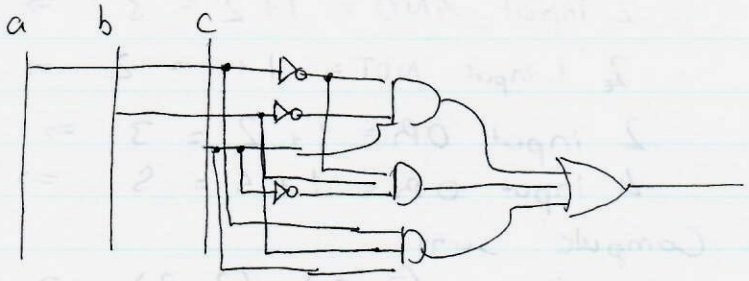
3. Add minterms that correspond to 1

$$f = \bar{a}\bar{b}c + \bar{a}b\bar{c} + abc$$

OR

$$f = \sum m(1, 2, 7)$$ ← which minterms you are adding.

4. Circuit:



Product of Sums

1. Truth table + identify 1 + 0 in the function column.
2. Maxterm: algebraic expression for the row.
   For each row, complement variable if its a 1.

   Ex: // Row 3: $a + \bar{b} + c$ ⇒ maxterm = $\overline{\text{minterm}}$

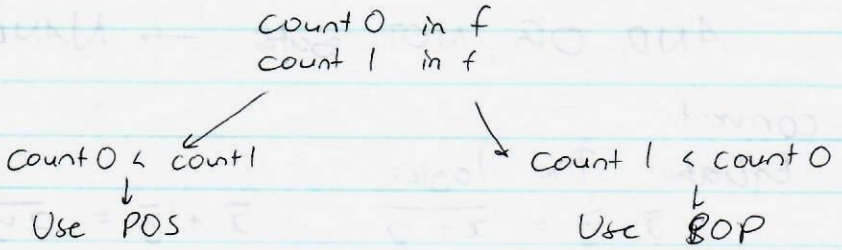
   Row 5: $\bar{a} + b + \bar{c}$

3. Multiply maxterms that correspond to 0.

$$f = \Pi M(0, 3, 4, 5, 6)$$

4. Circuit: OR inputs → 1 and.

To decide whether to use POS / SOP



SPOS vs. POS, SSOP vs. SOP

SPOS + SSOP: adding/multiplying minterms/maxterms.
- ↳ Every variable included
- ↳ Expensive to implement

Boolean Algebra

Simpler expression (POS, SOP): sum/product terms (no min/max)
- ↳ Simplify → terms that may not have min/maxterms.
- ↳ Cheaper to implement.

Conversion from POS ↔ SOP:

Double De Morgan.

Ex:// $f(x, y, z) = xy + xz + yz$. Convert to POS.

$$f(x,y,z) = \overline{\overline{xy + xz + yz}}$$

$$= (\overline{xy})(\overline{xz})(\overline{yz})$$

$$= (\overline{x} + \overline{y})(\overline{x} + \overline{z})(\overline{y} + \overline{z})$$

Combine brackets one by one

$$= (\overline{x} + \overline{x}\,\overline{z} + \overline{y}\,\overline{x} + \overline{y}\,\overline{z})(\overline{y} + \overline{z})$$

$$= (\overline{x}(1 + \overline{z}) + \overline{y}\,\overline{x} + \overline{y}\,\overline{z})(\overline{y} + \overline{z})$$

$$= (\overline{x} + \overline{y}\,\overline{z})(\overline{y} + \overline{z})$$

$$= (\overline{x}\,\overline{y} + \overline{x}\,\overline{z} + \overline{y}\,\overline{z})$$

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

Simplified to ~~POS~~ SOP before evaluating last negate

Trick: if function is in SPOS / SSOP, choose non-selected terms.

$$f = \sum m(1, 4, 7) = \prod M(0, 2, 3, 5, 6)$$

NAND AND NOR CIRCUITS

AND, OR, NOT gates $\longrightarrow$ NAND, NOR gates.

To convert:

1. Equation The logic:

$\bar{x} \cdot \bar{y} = \overline{x + y}$ $\qquad$ $\bar{x} + \bar{y} = \overline{xy}$ $\qquad$ (De Morgan)

2. Equation:

De Morgan + identity NAND + NOR

- Ex: //

$f = (a+b+c)(a+\bar{b}+\bar{c})$. Convert to ~~NAND~~/NOR.

1. Double D.M.

$$f = \overline{\overline{(a+b+c)(a+\bar{b}+\bar{c})}}$$

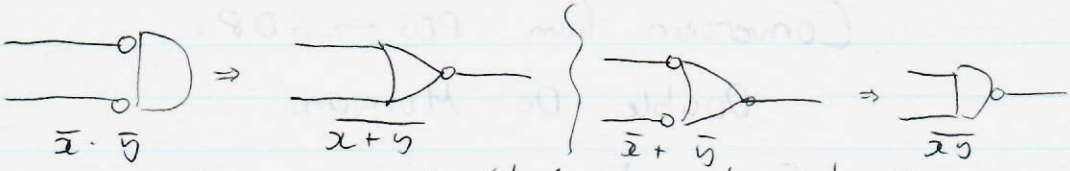
2. Evaluate inner D.M. ← NOR

$$f = \overline{\overline{(a+b+c)} + \overline{(a+\bar{b}+\bar{c})}}$$

(each $\overline{(\cdot)}$ is a NOR)

2. Circuit:

Basics: From D.M.



Simply insert inversions after/before each gate. If you add an inversion to 1 branch, do another one.

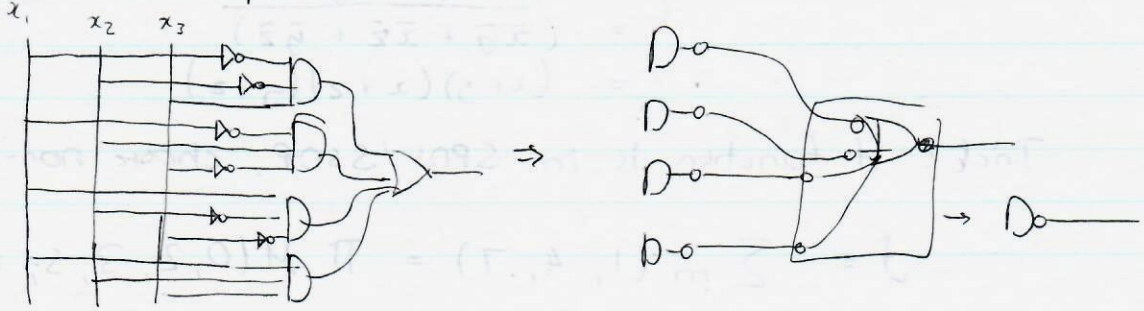
- Ex: // $f = \bar{x}_1 \bar{x}_2 x_3 + \bar{x}_1 x_2 \bar{x}_3 + x_1 \bar{x}_2 \bar{x}_3 + x_1 x_2 x_3$

1st option: Equation DDM:

$$\bar{\bar{f}} = \overline{\bar{x}_1 x_2 x_3 + \bar{x}_1 x_2 \bar{x}_3 + x_1 \bar{x}_2 \bar{x}_3 + x_1 x_2 x_3}$$

$$= \overline{(\bar{x}_1 x_2 x_3)(\bar{x}_1 x_2 \bar{x}_3)(x_1 \bar{x}_2 \bar{x}_3)(x_1 x_2 x_3)}$$

2nd option: circuit:



General guidelines:

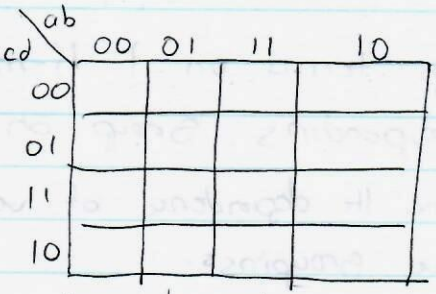
1. If SOP implementation → NAND/NOR
   Negate any AND/OR → NAND/NOR respectively. Other gates must use double inversion for conversion. Give 2 inversions always
2. If POS implementation → NAND/NOR
   Same advice.

K MAPS

Goal of K map: optimal function (minimal)

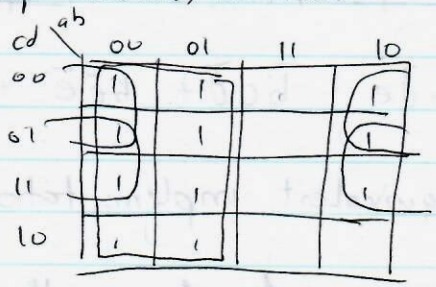
2, 3, 4 var K map:

1. Create truth table + identify minterms / maxterms.
2. Create a grid in the following manner.



← Grey coding: only changing by 1 bit.

3. Place minterms / maxterms in grid s.t. it corresponds to coding:
   ↳ Ex: // $\bar{a}b\bar{c}\bar{d}$ is minterm. Place in row 1, col 2
4. Group terms into maximal rectangle of size 2^n. Cover all ter.
   - Row, column groupings
   - Wrap around, corners!



5. Create the corresponding minterm / maxterm for each group.
   ↳ Look at which variables matter (var value is constant)
6. Combine

2 Tips:

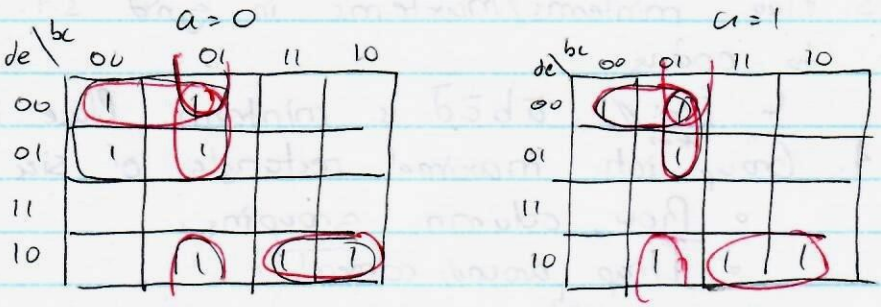
1. If POS (alternative funct) is needed after, simply group the non term squares.
2. If $f(x,y,z) = \ldots$ (variables):
   1. Create a code for every term (binary)
      - ↳ Ex: $a\bar{b}cd = 1011$
      - ↳ Ex: $acd = 1011$ or 1111 (b not in it)
   2. Place terms on K map.
      - ↳ If var missing, place possible terms

Higher Order K-maps: (5-6 vars).

1. Create a 2 k-maps (each of same size) with one K map as var1 = 0, and other is var2 = 0.

2. Groupings: group terms on 1 K-map + see if corresponding group on other K-map to (remove the dependence of var1). Then, create unique groupings.

- Ex: //



$$f(a,b,c,d,e) = \bar{b}\bar{d}\bar{e} + \bar{b}c\bar{d} + \bar{b}\bar{c}\bar{e} + \bar{a}\bar{b}\bar{d} + bd\bar{e}$$

* Functionally equivalent implementations, logically different

6 var K-maps: 4, 4-var K-maps.

Don't cares:

$$f = g_{POS/SOP} + D(\ldots)$$ ← Either 0/1

We can use don't cares to create minimal functions.

1. Try grouping terms by itself.
2. See if you can enlarge groupings by using few don't cares as possible.
3. Continue.

Ex: //

cd \ ab	00	01	11	10
00	X	1	1	X
01	0	X	1	0
11	0	0	1	0
10	0	0	1	0

$\Rightarrow f = b\bar{c}\bar{d} + ab$

↳ Cost:

2 input or = 3
2 input and = 3
3 input or = 4
} 10

Expanded version: $f = ab + b\bar{c}$

↳ Cost: 9

Key: use don't cares as sparingly as possible + do it AFTER grouping terms.

Don't care minimization results in logically/algebraically unequivalent statements but functionally equivalent

Output sharing:

If 2 functions available, check if you can do product sharing.

1. Take function that is uglier to minimize + compare w/ better function to make parallel groupings.
2. Use 1 term to do work for both functions.

While doing shared + unique terms DO NOT OVERLAP ZEROES

Terminology:

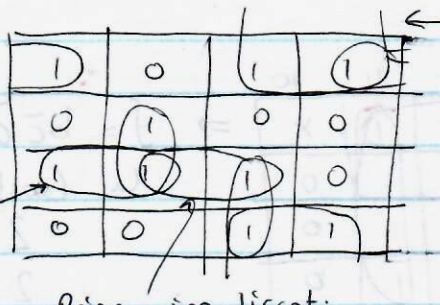
Completely specified

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

Incompletely specified.

0	X		
	0	1	
		X	

$\sum(\cdot,\cdot,\ldots)$ on set $\quad \Pi(\cdot,\cdot,\cdot)$ off set $\quad D(\cdot,\cdot,\cdot)$ don't care set



← Essential prime implicant: Prime implicant that contains unique minterm.

Implicant: only contains 1

Prime implicant: Largest possible rectangle

XOR K maps:
Stripes + checkerboards!

QUINE - McCLUSKEY OPTIMIZATION

Useful if looking at 4< inputs. SOP

Ex:// $f = \sum(0, 1, 4, 5, 16, 17, 21, 25, 29)$ ← 5 input function.

1. Create binary representation of each minterm + make table
2. Group the minterms s.t. ~~the~~ each group has the same # of 1s.

1 1s	0	00000 ✓
	1	00001 ✓
	4	00100 ✓
	16	10000 ✓
2 1s	5	00101 ✓
	17	10001 ✓
3 1s	21	10101 ✓
	25	11001 ✓
4 1s	29	11101 ✓

3. Take each minterm in each group + compare it to each minterm in adjacent groupings. If differ by 1 bit, then put into new table (grouped up similarly) w/ 1 bit ripped as don't care. Check mark each minterm covers.

0, 1	0000x ✓	5, 21	x0101 ✓
0, 4	00x00 ✓	17, 21	10x01 ✓
0, 16	x0000 ✓	17, 25	1x001 ✓
1, 5	00x01 ✓	21, 29	1x101 ✓
1, 17	x0001 ✓	25, 29	11x01 ✓
4, 5	0010x ✓		
16, 17	1000x ✓		

4. Repeat but only match adjacent minterms iff Xs match up. Do this until you cannot match.

~~0, 1, 4, 5~~	00x0x
0, 1, 16, 17	x000x
1, 5, 17, 21	x0x01
17, 21, 25, 29	1xx01

w/ unsroped P.I.

5. Create a P.I. table. Check mark ☒ minterms covered by each prime implicant.

	0	1	4	16	5	17	21	25	29
0, 1, 4, 5	✓	✓	✓		✓				
0, 1, 16, 17	✓	✓		✓		✓			
1, 5, 17, 21		✓			✓	✓	✓		
17, 21, 25, 29						✓	✓	✓	✓

6. Go through P.I. + remove if all minterms already covered
   ↳ Which minterms are uniquely covered by the P.I.
   1, 5, 17, 21 is useless

7. Function original P.I. + convert

$$f = \bar{x}_1 \bar{x}_2 \bar{x}_4 + \bar{x}_2 \bar{x}_3 \bar{x}_4 + x_1 \bar{x}_4 x_5$$

Hilroy

QM w/ don't cares: treat don't care like minterms but do not include on P.I. table

Note: If we have 2 P.I. w/ everything already covered except both cover a unique minterm ⇒ use either one.

MULTILEVEL CIRCUITS

Multilevel circuits (>2 gates to function output) allow us to optimize circuit:

- Product sharing: factor Boolean expression
- Create implementation ⇒ NAND/NOR: create sub POS/SOP exp
- Can use XOR factoring

NUMBER REPRESENTATIONS

Binary ⟷ Decimal (Dynamic range: $[0, 2^n - 1]$)

- Converting between different bases:
  - Base 2 → Base 16: group binary into 4 bits + convert each bit
    1100111100101_2 = ~~1100 1111 0010~~
    $= 0001\ 1001\ 1110\ 0101$ ← group from front!
    $= 19E5_{16}$
  - Base 2 → Base 8: group binary into 3 bits.
    $1100111100101_2 = 001\ 100\ 111\ 100\ 101$
    $= 14745_8$
  - How to find base of number given equation:
    Fundamental trick: $58 = 5b + 8$ (b is base) ⇒ 5 8 / b^1 b^0
    $135 = b^2 + 3b + 5$ → 1 3 5 / b^2 b^1 b^0
    Create equation + solve for b
- Unsigned addition:
  - Regular addition except might be limited by # of bits, cannot represent last carry if non-zero (overflow)

```
  1 0 1 0 1 0 1 0   170
+ 1 0 1 1 0 0 0 0   176
-----------------
1 0 1 0 1 0 1 0 1 0
```

Result if 8 bit limit

- Unsigned subtraction:
  - Like regular subtraction except if non-zero borrow at end, we have unsigned underflow

```
   0 1 10 1 0
U X 10 X 1 0 1 0 1 0     170
 -  0 0 1 1 0 0 0 0       48
 ---------------------
    0 1 1 1 1 0 1 0      122
```

- Signed numbers: +/- numbers

2's complement:

-48: $-(00110000)$

1. Flip all bits:

   11001111

2. Add 1:

```
      1 1 1 1
  1 1 0 0 1 1 1 1
+               1
-----------------
  1 1 0 1 0 0 0 0   => 2's complement of 48 (-48)
```

$r^n - V$ if $V \neq 0$

0 if $V = 0$

r (base), V is number.

Quicker way: find right-most 1 bit + flip all bits to left of it.

Dynamic range: $[-2^{n-1}, 2^{n-1} - 1]$

Operations:

- Addition: same but <u>ignore carry out</u>
- Subtraction: use 2's complement as adder
  - ↳ Ex: $6 - 13 = 6 + (-13)$
- Overflow + underflow:

  Only happens when carry in ≠ carry out for last digit.

- Fixed point:
  - Use same trick as 137:

    0.735 → base 4:

$0.735 \times 4 = 2.94 \Rightarrow 2$

$0.94 \times 4 = 3.6 \Rightarrow 3$

$0.6 \times 4 = 2.4 \Rightarrow 2$

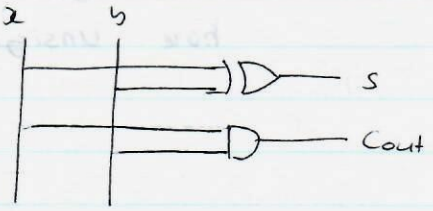
$0.4 \times 4 = 1.6 \Rightarrow 1$

$\Rightarrow \ldots .2321\ldots_4$ (2×4^{-1}, 3×4^{-2} ...)

Arithmetic Circuits

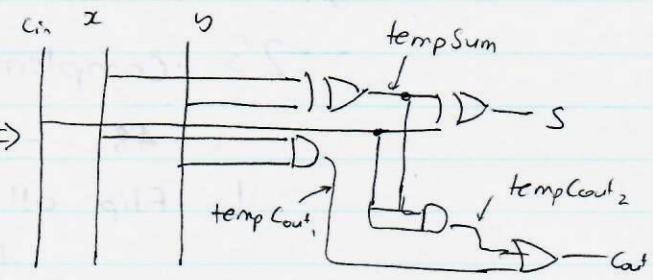
- Half adder circuit: adding 2 bits:

x	y	S	Cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



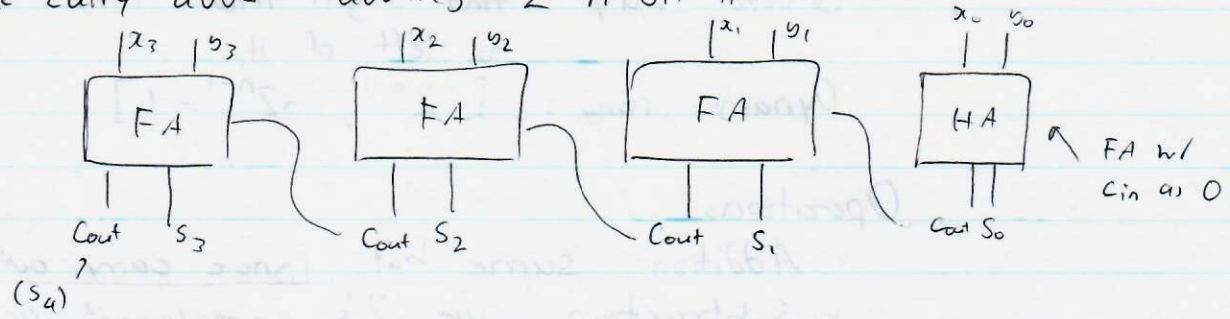
- Full adder circuit: adding multiple digits, accounting for C_{in}

x	y	C_{in}	S	Cout
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1



- Full adder made of half adders:
  1. HA for $x + y \Rightarrow$ tempSum + tempCout$_1$
  2. HA for tempSum + $C_{in} \Rightarrow$ sum + tempCout$_2$
  3. OR tempCout$_1$ and tempCout$_2 \Rightarrow$ Cout

- Ripple carry adder: adding 2 n-bit #:



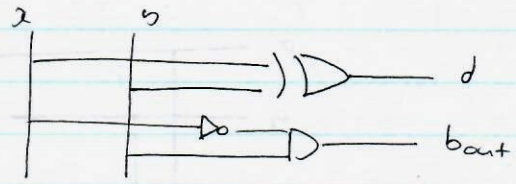
- Delay: 0^{th} bit $\Rightarrow$ 1 XOR, 1 AND, 1 OR
  i^{th} bit $\Rightarrow$ $C_{in} \Rightarrow$ 1 AND, 1 OR
  } Assuming RCA w/ only FA

Total delay: (1 XOR + 1 AND + 1 OR) + (n-1)(1 AND + 1 OR)

∴ If all gates have same delay:
Delay = $2n + 1$ $\Rightarrow$ More digits $\Rightarrow$ bigger delay

- Half subtractor: $x + y \Rightarrow$ difference + bout (1 if borrow required)

x	y	d	bout
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

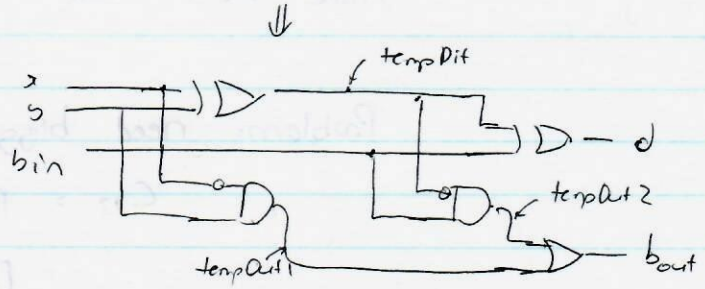


- Full subtractor: incorporate borrow in ($b_{in} = 1 \Rightarrow$ circuit is giving borrow)

x	y	bin	d	bout
0	0	0	0	0
0	0	1	1	1
0	1	0	1	0
0	1	1	0	0
1	0	0	1	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

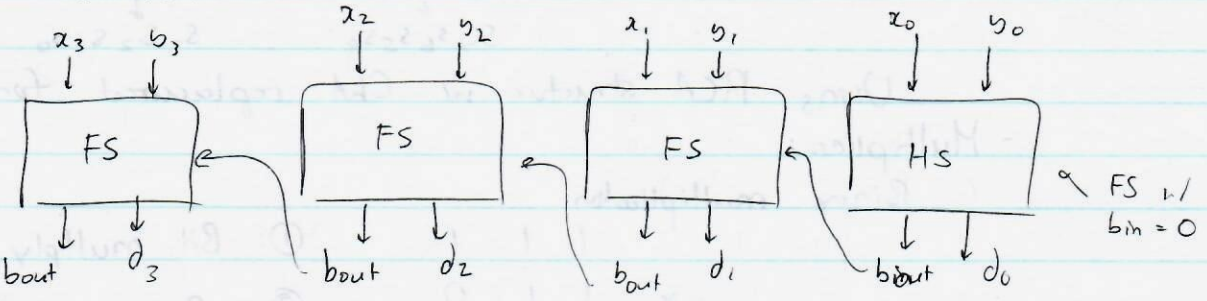
$\Rightarrow \quad d = x \oplus y \oplus b_{in}$

$b_{out} = \bar{x}y + b_{in}(\overline{x \oplus y})$



- Very similar to full adder:
  1: Half subtractor w/ $x + y \Rightarrow$ tempDif + tempOut1
  2: Half subtractor w/ tempDif + bin $\Rightarrow$ d + tempOut2
  3. OR of tempOut 1 and tempOut 2 $\Rightarrow$ bout

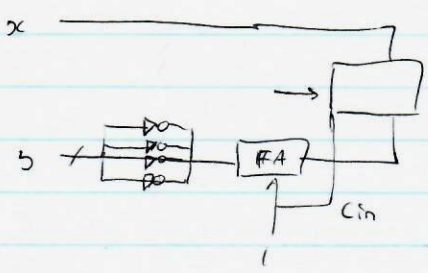
- Ripple subtractor:



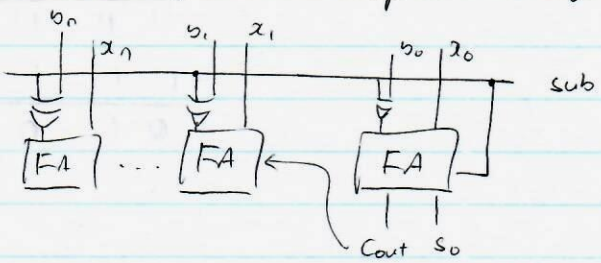
- Delay: similar to FA ripple

- Adding + subtracting in 1 circuit: adding two's complement of y if sub

#1:

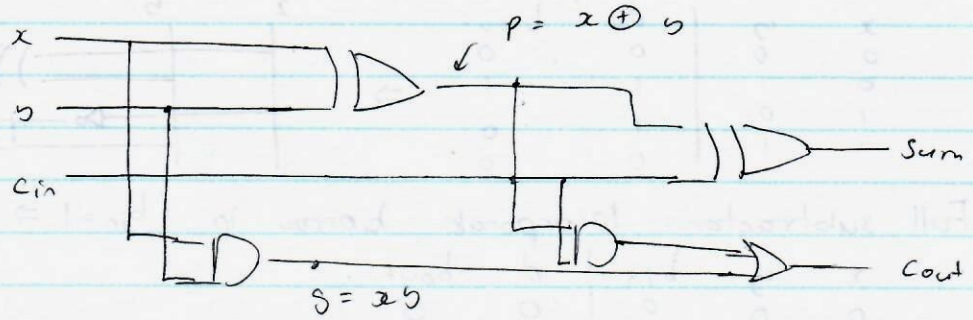


#2:



XOR: invert y, iff sub = 1 $\Rightarrow$ add in 1 in 1st carry in. $\Rightarrow$ TC

- Carry lookahead adder:



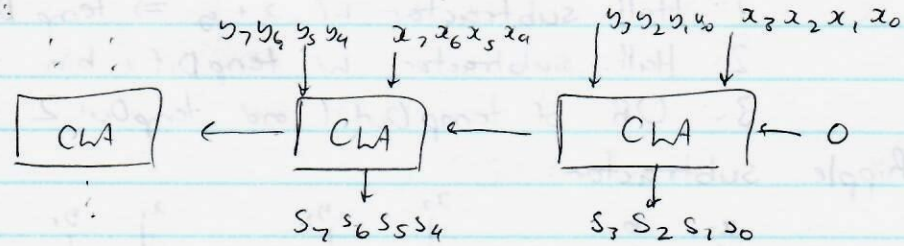
$$\therefore \quad C_{out} = c_{in} p + xy = c_{in} p + g$$

Make RCA faster: Compute p and g for all bit adders + easily compute carry

Problem: need bigger gates for higher input

$$C_{13} = P_{13} \cdot C_{12} + g_{13}$$
$$= P_{13} (P_{12} C_{11} + g_{12}) + g_{13}$$

- Compromise:



Using RCA structure w/ CLA replacement for FA

- Multipliers:

Binary multiplication:

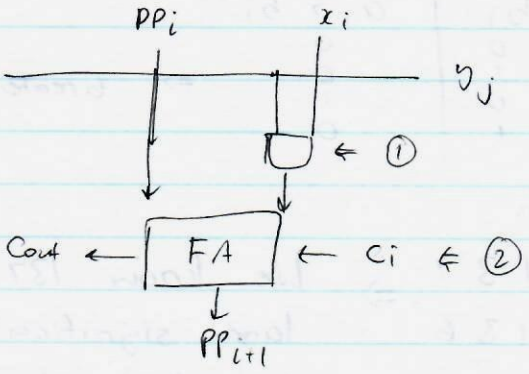
$$\begin{array}{rrrrrr} & & 1 & 1 & 1 & \\ & \times & 1 & 1 & 0 & \\ \hline \text{partial product} \rightarrow & 0 & 0 & 0 & & \\ + & 1 & 1 & 1 & & \\ \hline & 1 & 1 & 1 & 0 & \\ + & 1 & 1 & 1 & 0 & \\ \hline 1 & 0 & 1 & 0 & 0 & 0 \end{array}$$

① : Bit multiply via AND for y_0

② : Partial product add

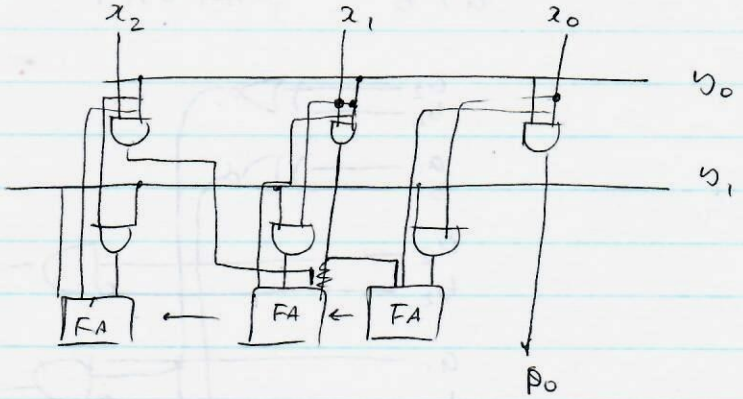
Combo of AND + FA circuits.

General circuit block ⇒ single bit multiplier.



3 bit multiplier:

Look @ textbook diagram



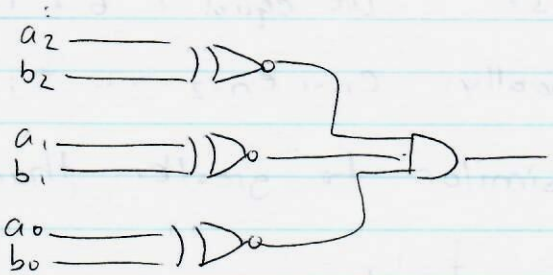
Common Circuit Blocks

- Comparator: equal, greater than, less than.
  - Equality:

x	y	=
0	0	1
0	1	0
1	0	0
1	1	1

⇒ Equality: $\overline{x \oplus y}$

Multi-bit: all bits should be equal:



- Greater than:

a_i	b_i	$a_i > b_i$
0	0	0
0	1	0
1	0	1
1	1	0

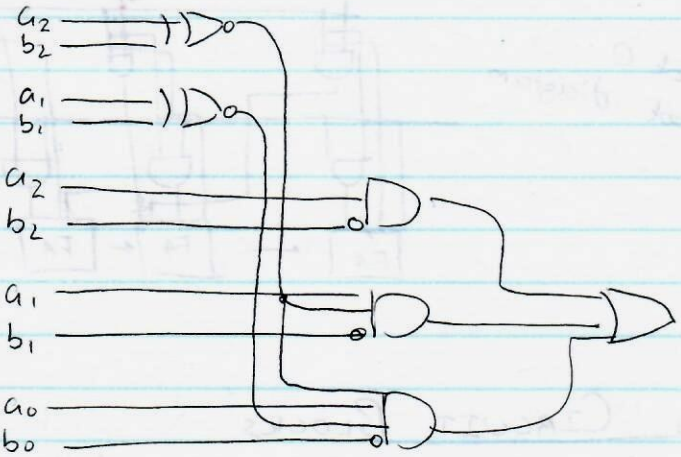
$\Rightarrow$ Greater than: $a_i \bar{b}_i$

Multi-bit:

137
136 $\Rightarrow$ We know 137 > 136 b/c all larger significant digits are equal and one place is different (7 > 6)

"1" vs. "1" ; "3" vs. "3" ; "7" vs. "6"

$$f_{a>b} = a_{n-1} b'_{n-1} + e_{n-1} a_{n-2} b_{n-2} + \cdots e_{n-1} e_{n-2} \cdots a_0 b_0$$



3 bit comparator "greater than"

- Less than:

a_i	b_i	$a_i < b_i$
0	0	0
0	1	1
1	0	0
1	1	0

$\Rightarrow$ Less than: $\bar{a}_i b_i$

Multi-bit:

136
137 $\Rightarrow$ 136 < 137 b/c all larger sig digs are equal + 6 < 7.

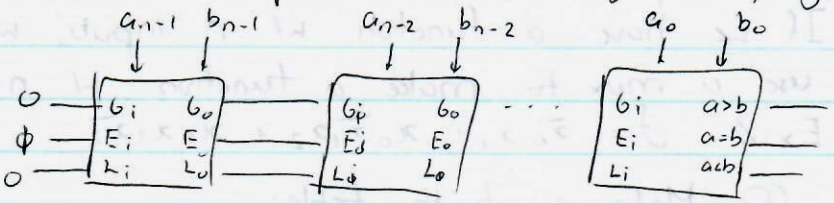
$\therefore$ Generally: $e_{n-1} e_{n-2} \cdots e_{i+1} \bar{a}_i b_i$

V. similar to greater than implementation

OR $f_{a=b}$, $f_{a>b}$ $\rightarrow$ $f_{a<b}$

$\hookrightarrow f_{a<b}$ if $a \neq b$ or a not $> b$

- Iterative comparator: w/ larger bit, gates become lager.

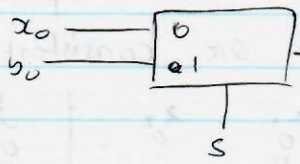


$$G_i = G_{i+1} + E_{i+1} a_i \bar{b}_i$$

$$E_i = E_{i+1} \cdot \overline{a_i \oplus b_i}$$

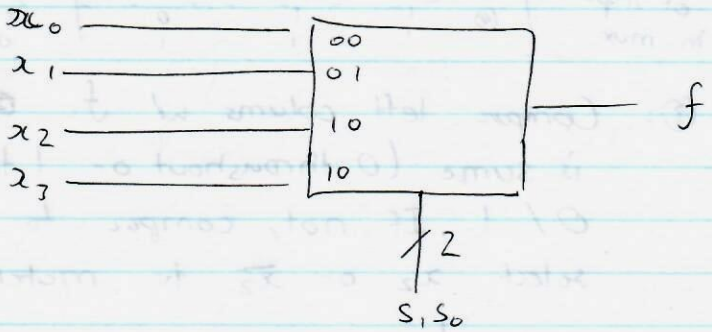
$$L_i = L_{i+1} + E_{i+1} \bar{a}_i b_i$$

- Multiplexers: if-else block:
  - 2-1 multiplexer:



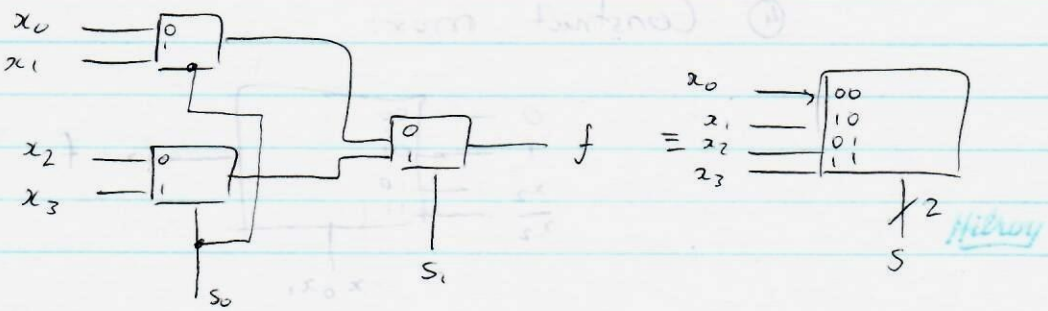
When s is 0, $x_0 = f$
" 1, $x_0 = f$

  - 4-1 multiplexer:



  - Multiplexer trees:
    We can build bigger multiplexers from 2-1 multiplexers

    4-1 mux out of 2-1 mux:



- Function implementation via mux

  If we have a function w/ n inputs, we can use a mux to make a function w/ n-1 inputs.

- Ex:// $f = \bar{x}_0 x_1 + x_0 \bar{x}_1 x_2 + x_0 x_1 \bar{x}_2$

① Make a truth table:

x_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	1
1	1	1	0

②: Divide truth tables into pieces where n-1 left columns are consistent (going to be selectors!)

Number of division = # of inputs in mux

	x_0	x_1	x_2	f	
①	0	0	0	0	] → 1 division
②	0	0	1	0	
③	0	1	0	1	
④	0	1	1	1	
	1	0	0	0	
	1	0	1	1	
	1	1	0	1	
	1	1	1	0	

③: Compare left colums w/ f. ~~Eh~~ If f is same (0 throughout or 1 throughout), choose 0 / 1. If not, compare to right column + select x_2 or $\bar{x}_2$ to match

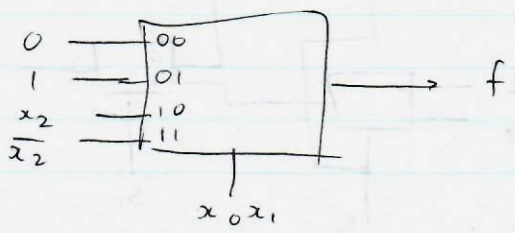
①: $f = 0$ ⇒ input is 0 when $x_0 x_1 = 00$

②: $f = 1$ ⇒ input is 1 when $x_0 x_1 = 01$

③: $f = x_2$ ⇒ input is x_2 when $x_0 x_1 = 10$

④: $f = \bar{x}_2$ ⇒ input is $\bar{x}_2$ when $x_0 x_1 = 11$

④ Construct mux:



- If we don't have big muxes, there is an alternative:
  Cofactoring: group any expression in terms of x_i and $\bar{x}_i$
  - Ex: // $f = \bar{x}_0 x_1 + x_0 \bar{x}_1 x_2 + x_0 x_1 \bar{x}_2$

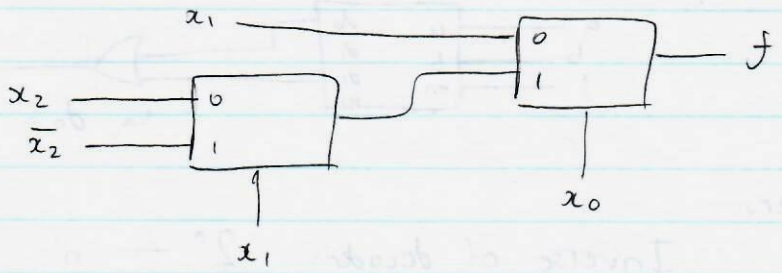
  ①: Group x_0 and $\bar{x}_0$

  $$f = \bar{x}_0 (x_1) + x_0 (\bar{x}_1 x_2 + x_1 \bar{x}_2)$$

  ②: Recurse + group x_1

  $$f = \bar{x}_0 (x_1) + x_0 (\bar{x}_1 (x_2) + x_1 (\bar{x}_2))$$

  ③: Create multiplexer by working in → out.
  ↳ Note the term needed when $x_1 = 0$ ($\bar{x}_1$) or $x_1 = 1$ (x_1). Use this as input lines.



Logic: When x_1 is 0 ($\bar{x}_1$), x_2 is selected. When x_1 is 1 (x_1), $\bar{x}_2$ is selected. When $x_0 = 0$, x_1 is selected. Otherwise $\bar{x}_1 x_2 + x_1 \bar{x}_2$ is selected.

Order of variable groupings ⇒ final circuit

DECODERS + ENCODERS

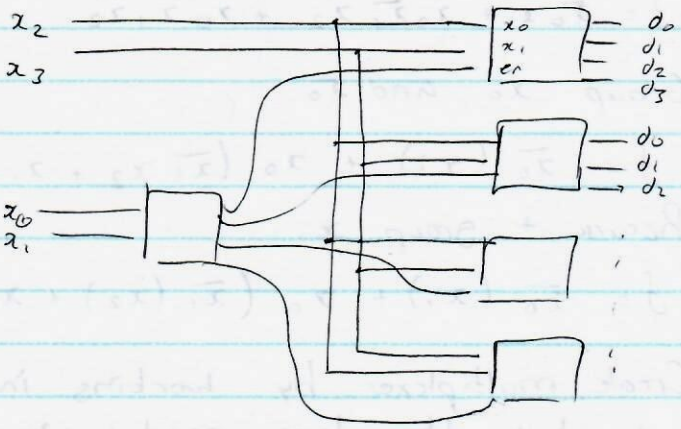
n inputs → 2^n outputs

x_0	x_1	en	d_0	d_1	d_2	d_3
x	x	0	0	0	0	0
0	0	1	1	0	0	0
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	1

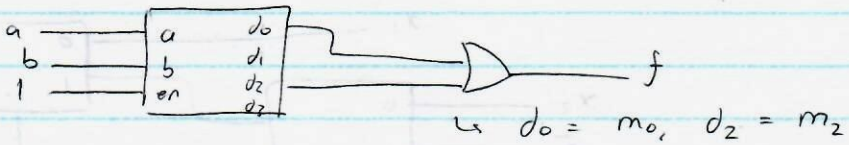
⇒ Distinct codings for each minterm.
↳ $d_0 = m_0$
$d_1 = m_1$
⋮

Hilroy

Create bigger encoders out of smaller encoders.
↳ 2 first digit decoder ⇒ enable's second stage.



Function implementation: connect right type of minterms!
Ex:// Implement $f(a, b) = \Sigma(0, 2)$



↳ $d_0 = m_0$, $d_2 = m_2$

- Encoders:

Inverse of decoder: $2^n \rightarrow n$

Problems:

1. What if 2^n input = 0 . . . 0
   ↳ Valid signal: 0 in that case, 1 otherwise
2. What if input has multiple 1s?
   ↳ Only decode based on highest index element:

Note how coded based on highest index ⇒ priority encoder

d_0	d_1	d_2	d_3	x_0	x_1	valid
0	0	0	0	0	0	0
1	0	0	0	0	0	1
x	1	0	0	0	1	1
x	x	1	0	1	0	1
x	x	x	1	1	1	1

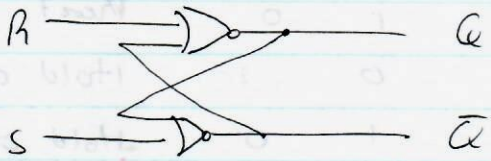
- Demux: 1 input → several outputs (decoders)
  ↳ Decoder enable = demux input, decoder input = demux selector

LATCHES

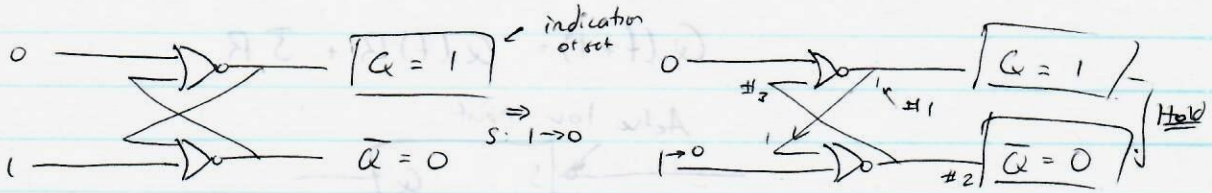
Storage element ← memory.

① : SR latch
② : $\overline{S}\,\overline{R}$ latch
③ : D Latch

① : SR Latch:

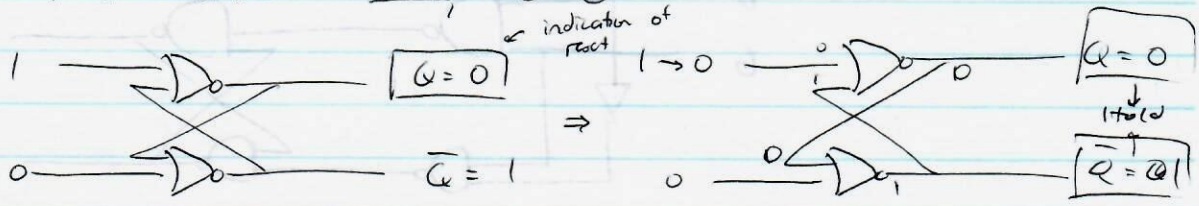


#1: Set state: R = 0, S = 1



Trick: looked @ R and S right before + during switch. ESP. LOOK AT WHAT CHANGED (ex., S) FIRST

#2: Reset state: R = 1, S = 0



#3: Restricted / undesirable: R = 1, S = 1

Different inputs based on assumptions.

To account for this:

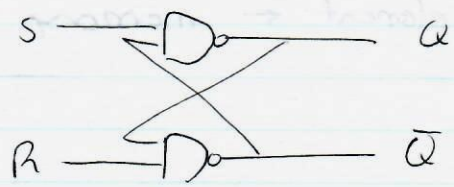
#1: Set dominated latch: cause output to be set
#2: Reset dominated latch: " reset
#3: Toggle / flip (JK latch)

$$Q(t+1) = Q(t)\,\overline{R} + S\overline{R}$$

new → $Q(t+1)$; $Q(t)$ ← current

History ($\overline{R}$ + S are t+1 values)

② $\bar{S}\bar{R}$ latch

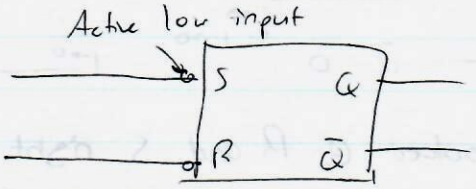


Analysis:

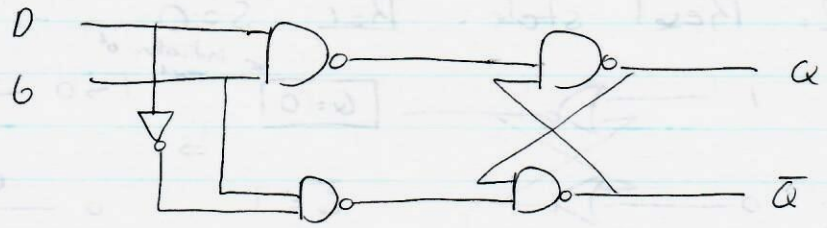
S	R	Q	$\bar{Q}$	Action
1	0	0	1	Set
0	1	1	0	Reset
1	1	0	1	Hold after set
1	1	1	0	Hold after reset
0	0	?	?	Restricted

Opposite of SR: 0 → 1 instead of 1 → 0

$$Q(t+1) = Q(t)R + \bar{S}R$$



③ Gated D Latch



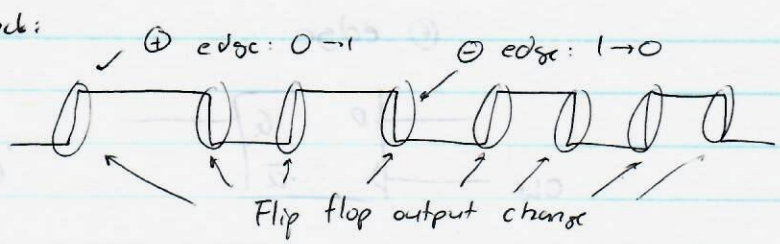
Analysis:

D	G	Q	$\bar{Q}$	Action
1	1	1	0	Set
0	1	0	1	Reset
0/1	0	Prev. Val.		Hold

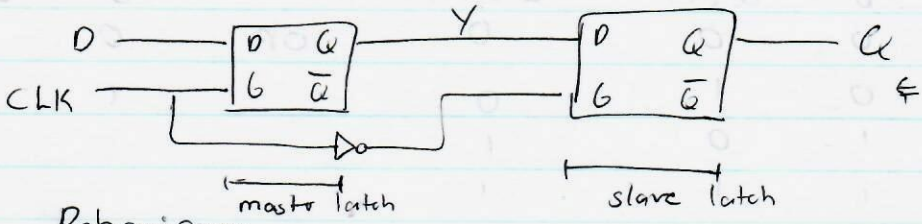
$$Q(t+1) = DG + Q(t)\bar{G}$$

FLIP FLOP

Edge triggered ⇒



① Master-slave flip flop:



⇐ Negative edge.

Behaviour:

Assume CLK = 1 ⇒ Y = D

Slave latch: D = Y, G = 0 ⇒ hold

↳ Hold: Q = whatever D right before CLK changing.

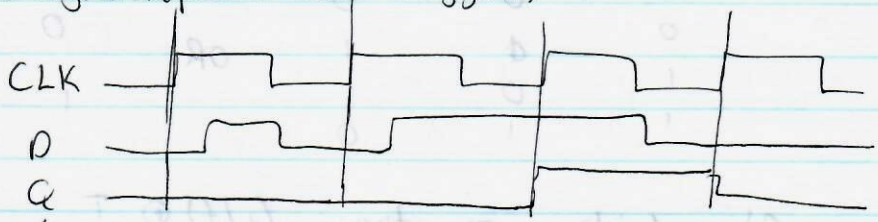
Assume CLK: 1 → 0 ⇒ Y is held at D (Y has no change)

Slave: Q = Y b/c G of slave = 1

↳ Following D prior!

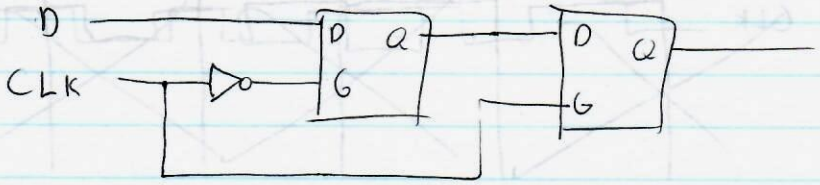
Why "master-slave": slave latch follows master output.

* Time diagram (positive edge triggered) *



↳ Essentially: Whenever CLK is ⊕ trigger, takes in D value prior to input.

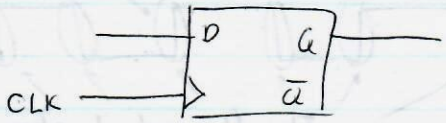
⊕ MSFF:



Hilroy

Symbol for D-type flip flop:

⊕ edge:



⊖ edge:

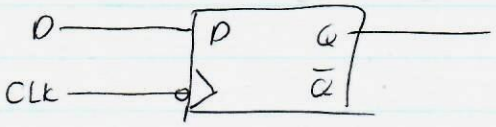


Table:

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

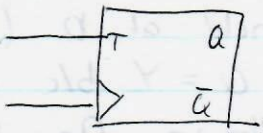
OR

D	Q(t+1)
0	0
1	1

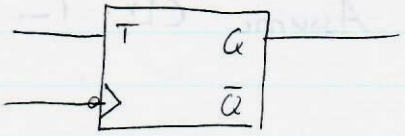
Characteristic equation: $Q(t+1) = D$

② : Toggle Flip Flop (TFF)

⊕ edge



⊖ edge:



Characteristic table:

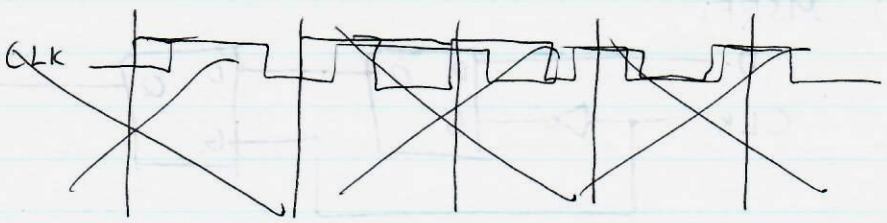
T	Q	$\tilde{Q}(t+1)$
0	0	0
0	ϕ	1
1	0	1
1	1	0

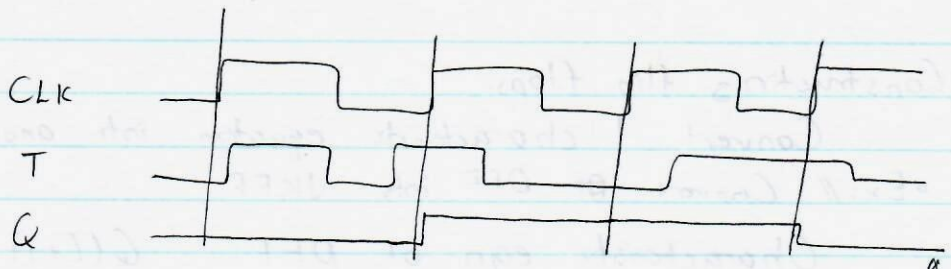
OR

T	Q(t+1)
0	Q(t)
1	$\overline{Q(t)}$

Characteristic equation: $Q(t) \oplus T$

Time diagram



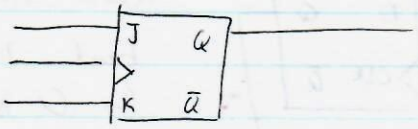


Toggles Q if T is 1 at positive edge.

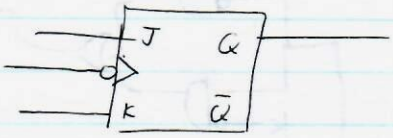
Assumed it was at reset

③ JK Flip Flop

⊕ edge:

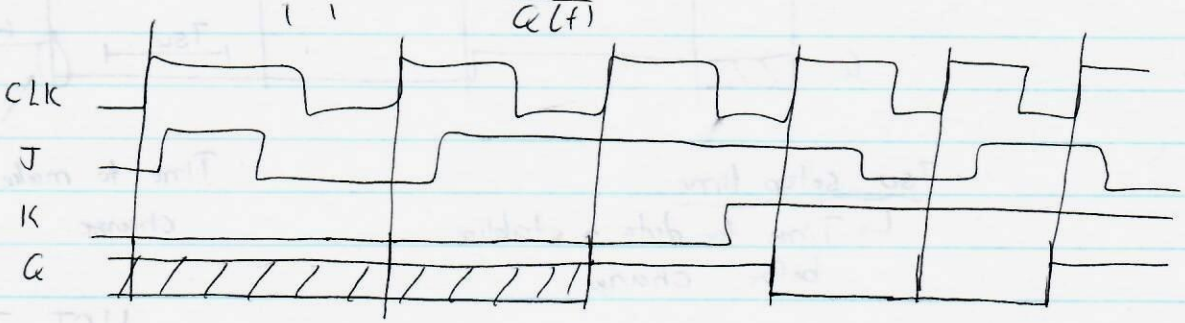


⊖ edge:



Characteristic table:

J	K	Q(t+1)	
0	0	Q(t)	(hold)
0	1	0	(reset)
1	0	1	(set)
1	1	$\overline{Q(t)}$	



Characteristic equation: $Q(t+1) = J\,\overline{(Q(t))} + \bar{K}\,Q(t)$

- Other pins:
  - Reset pin: forces $Q = 0$
  - Set pin: forces $Q = 1$
  - Enable pin: blocking changes based on CLK

} Synchronous: follows clk edge
Asynchronous: happens immediately.

- Constructing flip flops:

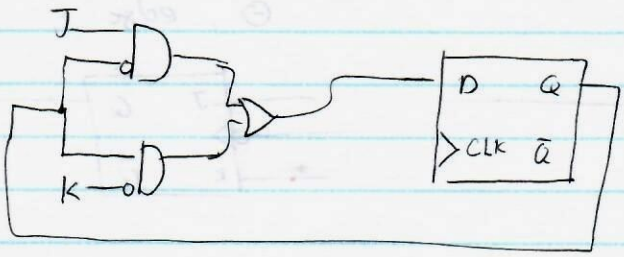
  Convert 1 characteristic equation into another.

  - Ex: // Convert ~~D~~ DFF into JKFF.

    Characteristic eqn of DFF: $Q(t+1) = D$

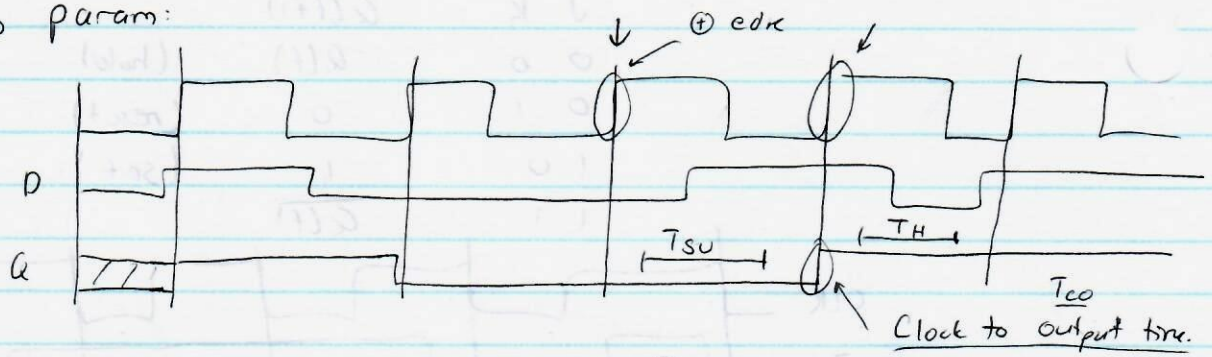
    " of JKFF: $Q(t+1) = J\overline{Q(t)} + \overline{K}Q(t)$

    ∴ Idea: Use $J(\overline{Q(t)}) + \overline{K}Q(t)$ as D



Note how Q is fed back into circuit!
↳ Q is input AND output.
OR use $\bar{Q}$ as input param

- Timing param:



T_{SU} setup time
↳ Time for data to stablize before change

T_H: hold time
↳ Time for data to stable hold after change.

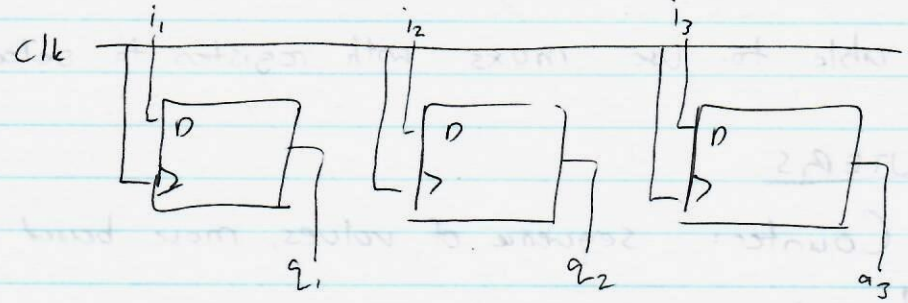
Time to make output stable after change.

NOT INSTANTANEOUS CHANGES

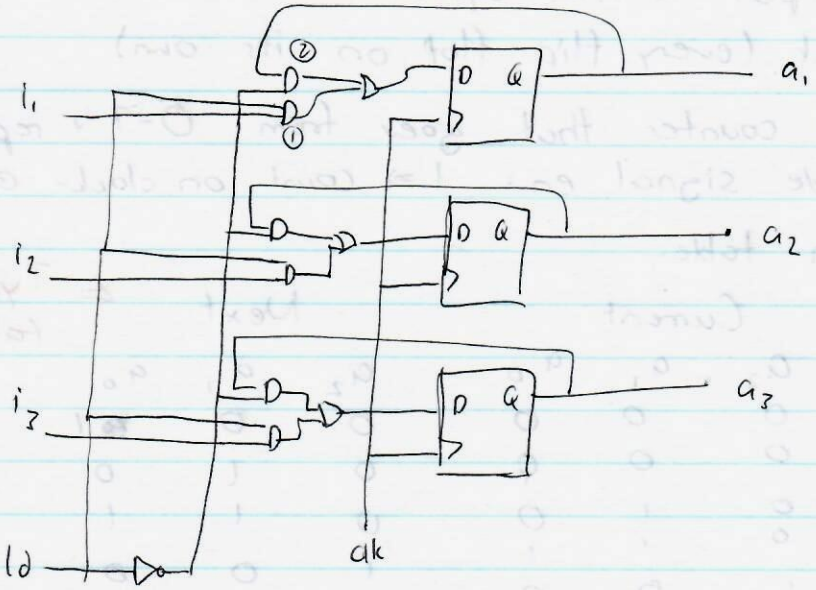
REGISTERS

Register ← group of flip flops chained together w/ same clock.

Register that loads data on all active clock edge.



Register to hold + load data:

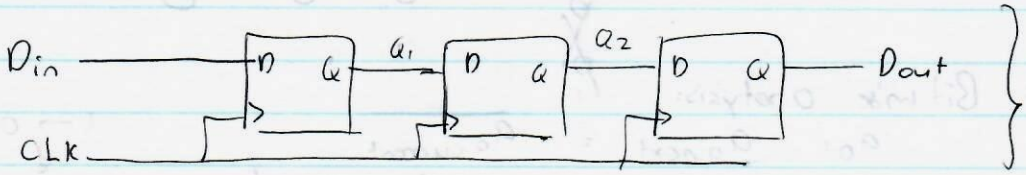


Register holds values if ld = 0, loads data if ld = 1.

When ld = 1 ⇒ ② is off, ① is on, so new data loaded

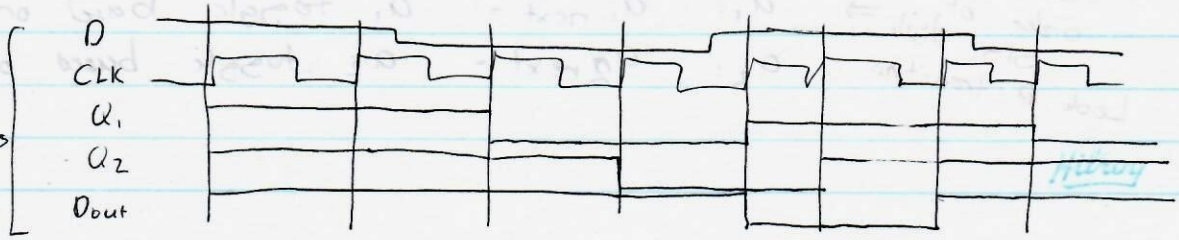
When ld = 0 ⇒ ② is on, ① is off, so data held

Shift register:



n-bit register:
$q_0 \rightarrow q_{n-1}$

Notice shift!



Universal register: shift register augmented

Shift up: $q_0 \rightarrow q_{n-1}$

Shift dom: $q_{n-1} \rightarrow q_0$

Load: ——————

load	up	dn	Action
0	0	0	load
0	0	1	shift dom
0	1	x	shift up
1	x	x	load

Be able to use muxs with register to select operation.

COUNTERS

Counter: sequence of values, move based on clock.

① Asynchronous counter

Output is not synchronized to master clock (every flip flop on its own)

Ex:// 3-bit counter that goes from 0-7 + repeats.
Include signal en: 1 ⇒ count on clock-edge, 0 ⇒ hold

① Truth table.

Current			Next		
a_2	a_1	a_0	a_2	a_1	a_0
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0

← Typical truth table for sequential ⇓ Current → next

② Bit wise analysis:

Look in order of stg. diff ⇒ Look @ transitions:

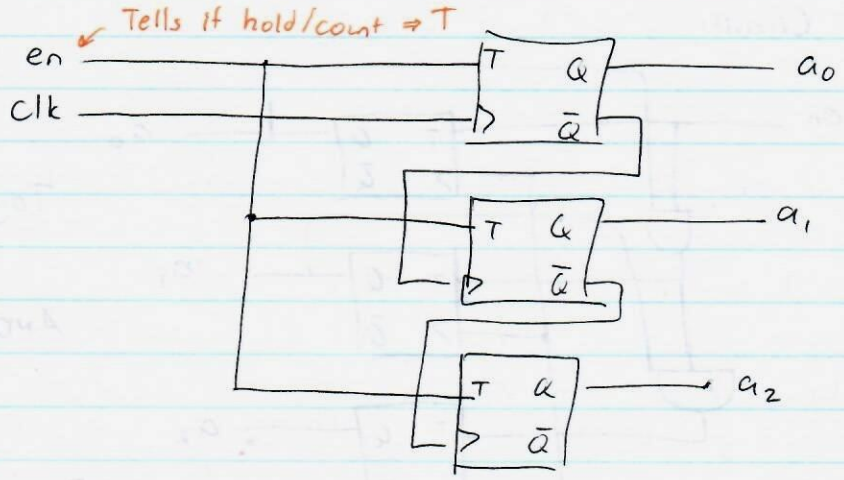
a_0: $a_{0\,next} = \overline{a_{0\,current}}$ (1 → 0)

a_1: $a_{1\,next}$ = a_1 toggle based on a_0 1 → 0

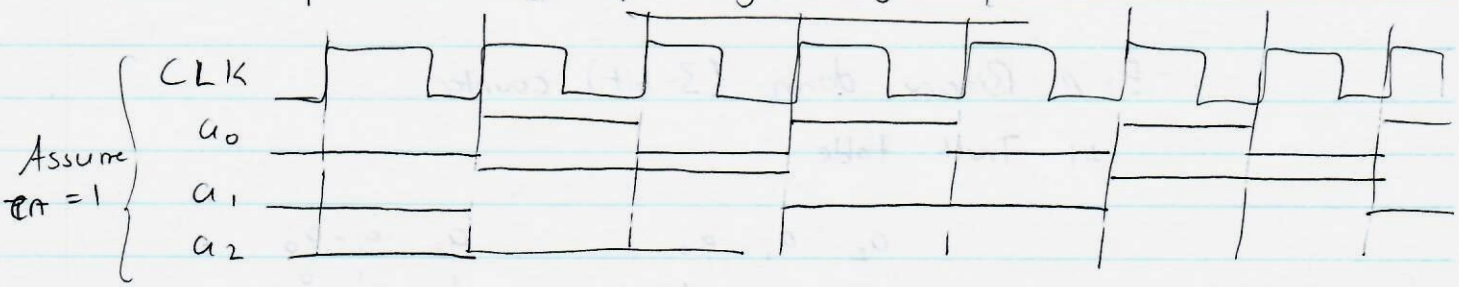
a_2: $a_{2\,next}$ = a_2 toggle based on a_1

③ Construct circuit:

B/c toggles ⇒ TFF. Since a_n depends on a_{n-1} output, we will link them via clocks!



To help w/ design: timing diagram.



② Synchronous Counters:

All counters have same clock ⇒ no counter is dependent on another counter

∴ Inputs should be combinational logic of past inputs b/c past inputs not CLK.

- Ex:// 3 bit binary counter:

#	Current a_2	Current a_1	Current a_0	Next a_2	Next a_1	Next a_0
1:	0	0	0	0	0	1
	0	0	1	0	1	0
	0	1	0	0	1	1
	0	1	1	1	0	0
	1	0	0	1	0	1
	1	0	1	1	1	0
	1	1	0	1	1	1
	1	1	1	0	0	0

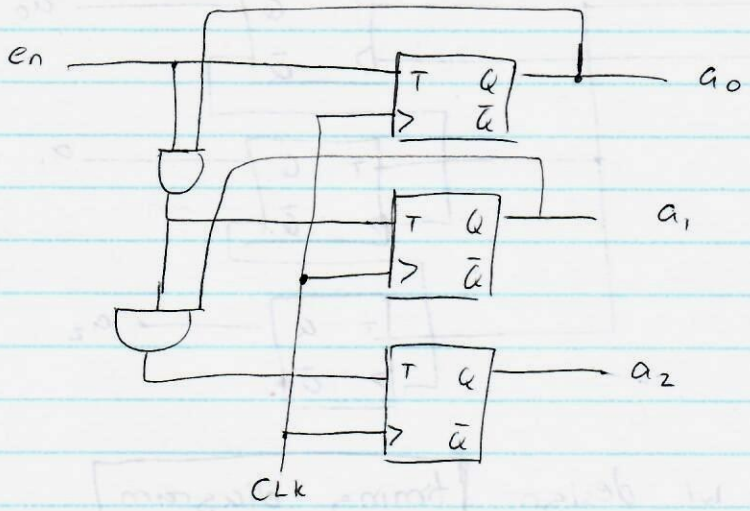
Hilroy

#2 Analysis:

a_0: toggles, a_1: toggles if $a_{0_{current}} = 1$.

a_2: toggles if both $a_{0_c} + a_{1_c} = 1$.

#3 Circuit:



Toggle rules:
bits are toggling

AND: we want BOTH bit AND en to be 1.

EN as toggle: If en is 1, then a_0 always toggles!

- Ex:// Binary down (3-bit) counter

#1 Truth table

a_2	a_1	a_0
1	1	1
1	1	0
1	0	1
1	0	0
0	1	1
0	1	0
0	0	1
0	0	0

a_2	a_1	a_0
1	1	0
1	0	1
1	0	0
0	1	1
0	1	0
0	0	1
0	0	0
1	1	1

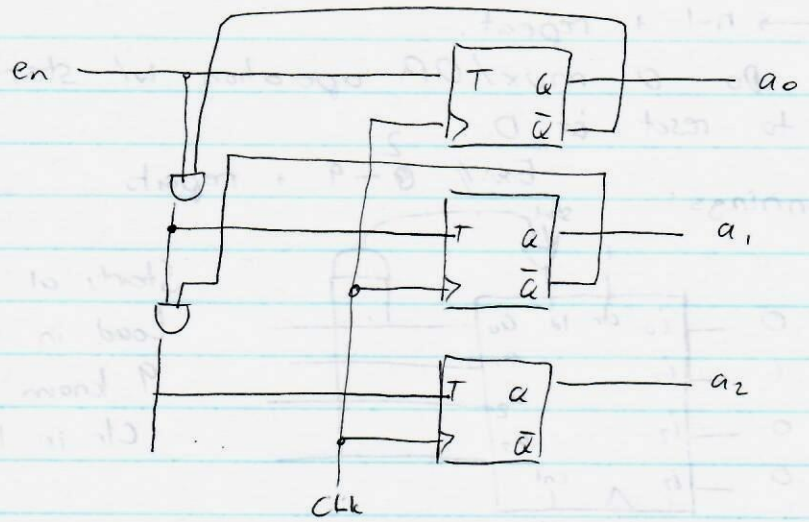
#2: Analysis:

a_0: Toggling always

a_1: Toggling if $a_0 = 0$

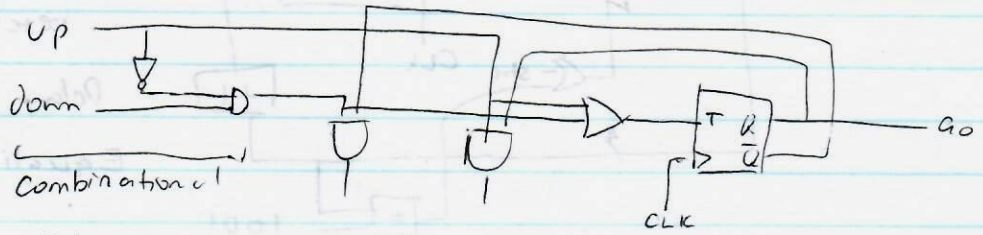
a_2: Toggling if $a_0 = a_1 = 0$.

#3: Circuit:



- Ex:// Implement D/U counter!
  - ↳ Create Down + up signals ⇒ combine + do same logic for toggling. Basically muxing.

Essentially:

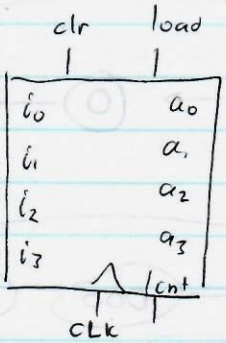


- Ex:// D/U counter w/ parallel load.

  Parallel load: $Q(a_i) \oplus i_i$ (new load)

  ↳ Implement for each flip flop, like above.

- Symbols:



+ Table explaining operations.

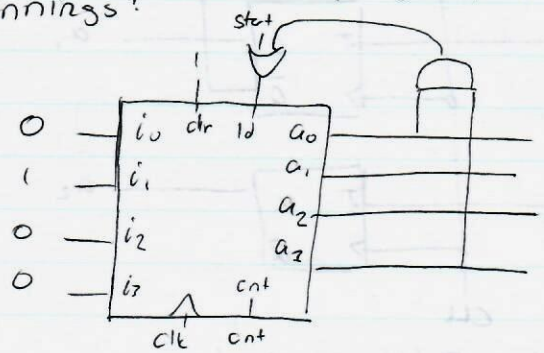
- Counter modifications

1. $0 \longrightarrow n-1$ + repeat.

   Do a mux/OR operation w/ start + output to reset at 0.

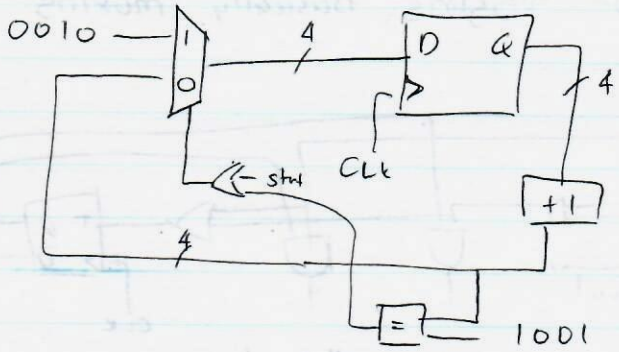
   Ex:// $2 \overset{2}{\circledcirc} 9$ + repeats.

   Kennings:



Starts at 2 ($i_{0\to 3}$).
Load in when at start/9
9 known ∈ comb. logic
Clr is 1 (never set to 0).

   Mathai:

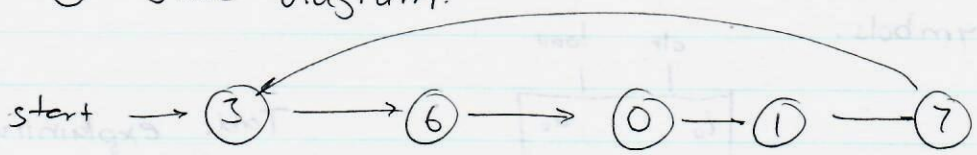


Mux operation to reset.
Delay counter to count up.
Equality to see if at 0.

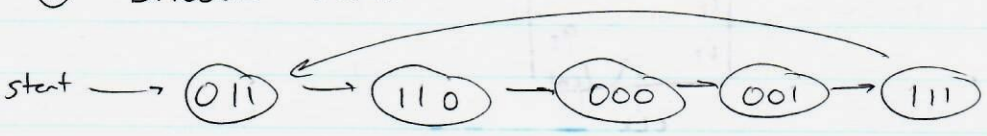
2. Complex:

   Ex:// Design counter that counts $3 \to 6 \to 0 \to 1, \to 7$

   ① State diagram:



   ② Encode values:



③ State table:

current state			next state		
a_2	a_1	a_0	a_2	a_1	a_0
0	1	1	1	1	0
1	1	0	0	0	0
0	0	0	0	0	1
0	0	1	1	1	1
1	1	1	0	1	1

④: Add extra columns that correspond to flip flop

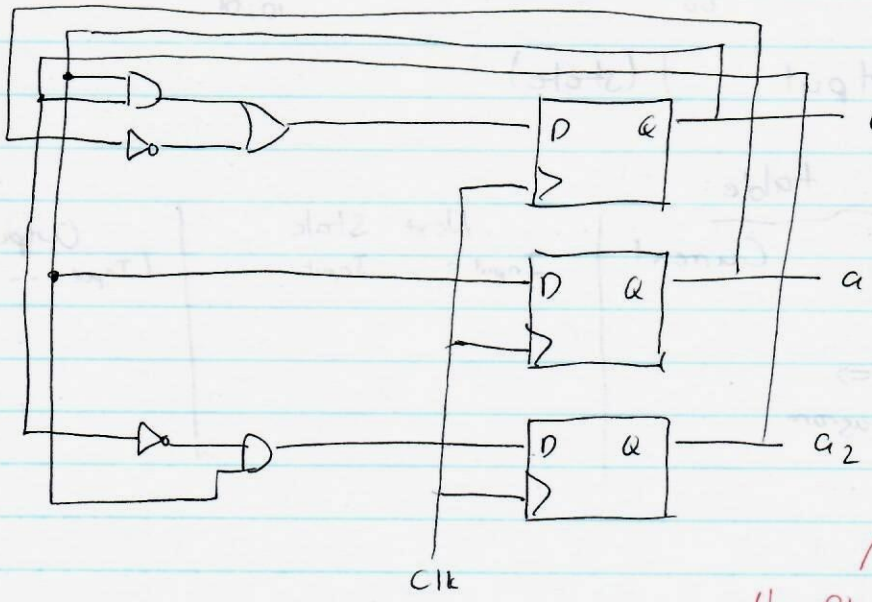
KEY

a_2	a_1	a_0	a_2	a_1	a_0	d_2	d_1	d_0	← Values generated s.t $a_{2_{old}} \rightarrow a_{2_{new}}$ is correct!
0	1	1	1	1	0	1	1	0	
1	1	0	0	0	0	0	0	0	
0	0	0	0	0	1	0	0	1	
0	0	1	1	1	1	1	1	1	
1	1	1	0	1	1	0	1	1	

⑤: Find equations for input (K Maps, Bool. Alg):

$$d_2 = a_2' a_0$$
$$d_1 = a_0$$
$$d_0 = a_1' + a_2 a_0$$

⑥ Draw circuit



Not shown:
set + reset.

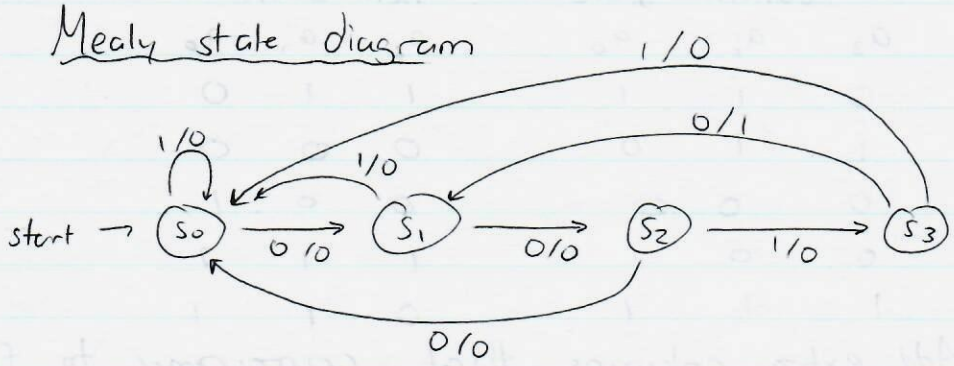
Analysis:
1. Set at 0

First value after reset
0 0 1
↓ 1 1 1
↓ 0 1 1
↓ 1 1 0
↓ 0 0 0
↓ 0 0 1
→ Find out sequence!

Use OL to find next # ⇒ continue until repeat.

SEQUENTIAL CIRCUITS

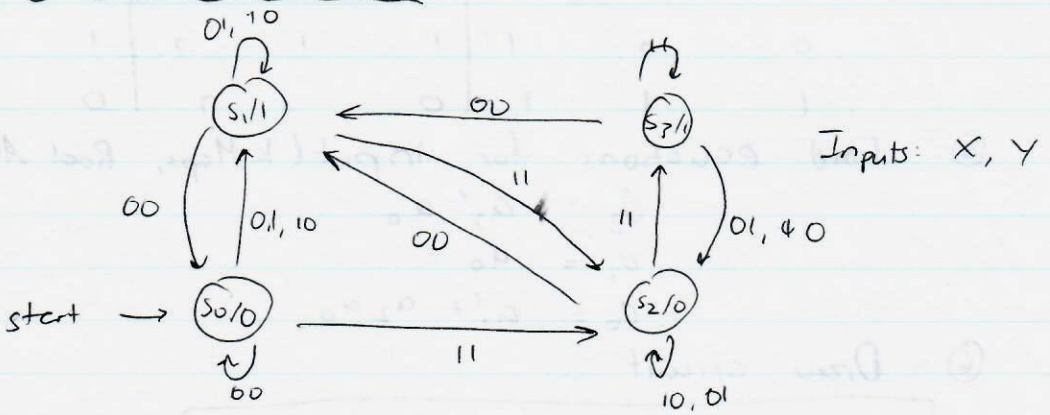
Mealy state diagram



Output shown as .../output_bit (in state/transition)

Output: f(state, input)

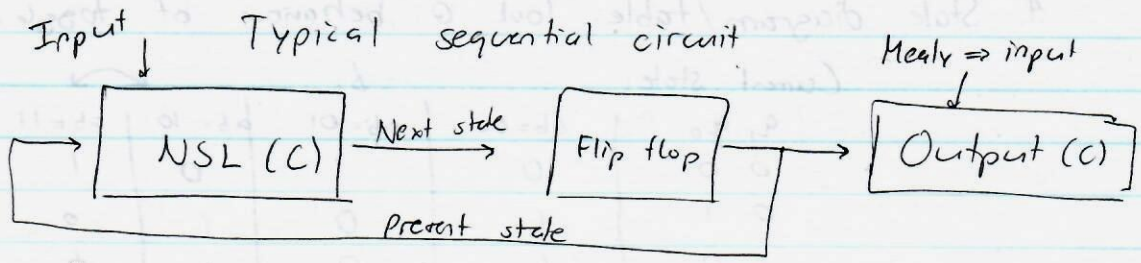
Moore state diagram



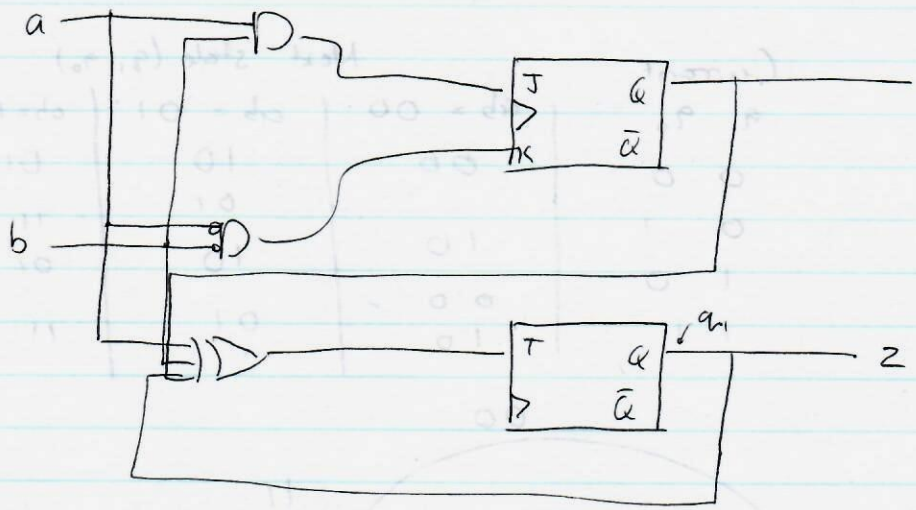
Output: f(state)

State table:

Current	Next State		Output	Mealy / Moore
	Input = ...	Input = ...	(Input = ... Input ...)	
Fill out based on ⇒ state diagram				



- Sequential circuit analysis:
  1. Identify flip flops → tells us number of states (# = 2^{n} flip flops)
  2. Understand circuit output + equations
  3. " Next state logic "
  4. State diagram / table
  5. Remove unecessary things.
  - Ex: //



1. 2 Flip flops ⇒ 4 states (00, 01, 10, 11)
   2 inputs (a, b), 1 output (z)
2. Output logic: $z = q_1$
3. NSL:

$$j_0 = a\,b$$
$$k_0 = \bar{a}\,\bar{b}$$
$$t_0 = a \oplus b \oplus q_1 \oplus q_0$$

Hilroy

4. State diagram / table: look @ behaviour of toggle / JK

Current state: b_1

q_1 q_0	ab = 00	ab = 01	ab = 10	ab = 11
0 0	0	1	0	1
0 1	1	0	1	0
1 0	1	0	1	0
1 1	0	1	1	1

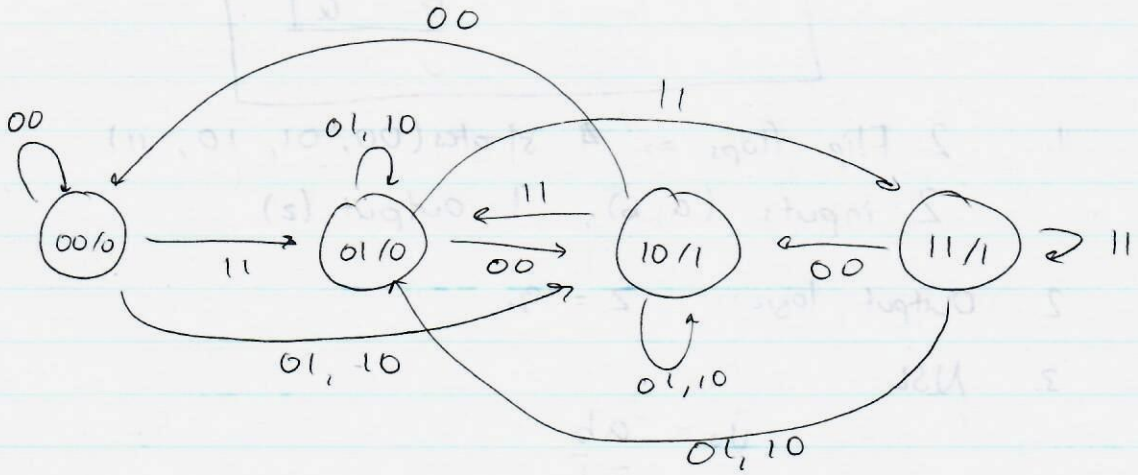
Current — j_0 k_0

q_1 q_0	ab = 00	ab = 01	ab = 10	ab = 11
0 0	01	00	00	10
0 1	01	00	00	10
1 0	01	00	00	10
1 1	01	00	00	10

Based on circuit + equation.

Current — Next state (q_1 q_0)

q_1 q_0	ab = 00	ab = 01	ab = 10	ab = 11
0 0	00	10	01	10
0 1	10	01	11	01
1 0	00	10	01	10
1 1	10	01	11	01



- Sequential circuit design:
  1. Understand verbal description of problem
  2. Create a state diagram
     - ↳ States
     - ↳ Transitions between states
     - ↳ Output logic
  3. State reduction: redundant states → simpler diagram
  4. Encode states
  5. Select flip flop to store information. # flip flops = # of encode bits
  6. Output + next state logic
  7. Draw the circuit.

- Ex: // Input + output z. x changes between clock edges
  If 101 detected, circuit outputs 1, else it outputs 0.

① States: start off w/ state w/ no input

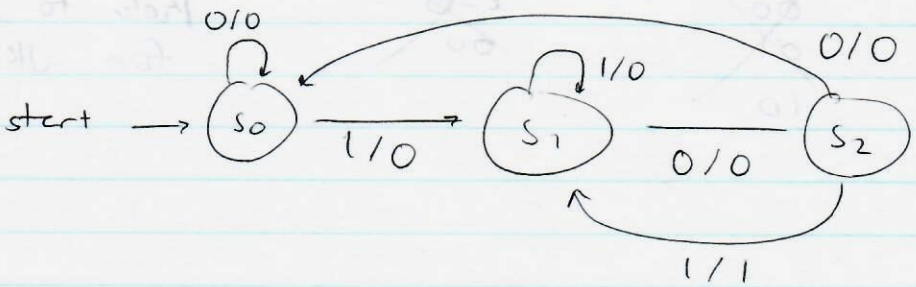
Transitions:

S_0 : 0 ⇒ remain in S_0
S_1 : 1 ⇒ stay in S_1 b/c seen 1
S_2 : 0 ⇒ breaks pattern, return to S_0
1 ⇒ go to S_1 b/c starting next pattn

Account for every input in each state.

Output:

0 ⇒ if not 1. 1 ⇒ S_2 gets a 1.



② Encode states:

$S_0 = 00$, $S_1 = 01$, $S_2 = 10$,

State table:

Current State x	Next State $x=0$	Next State $x=1$	Output $x=0$	Output $x=1$
00	00	01	0	0
01	10	01	0	0
10	00	01	0	1

* 11 is a don't care ! *

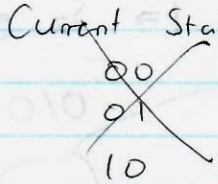
③ Flip flops:

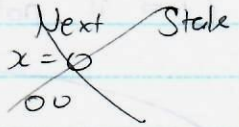
Current State	Next State $x=0$	Next State $x=1$	DFF $(d_1 d_0)$ $x=0$	DFF $(d_1 d_0)$ $x=1$
00	00	01	00	01
01	10	01	10	01
10	00	01	00	01

$\therefore \quad d_1 = \bar{x} q_0, \quad d_0 = x$

Current State	Next State $x=0$	Next State $x=1$	TFF $(t_1 t_0)$ $x=0$	TFF $(t_1 t_0)$ $x=1$
00	00	01	00	01
01	10	01	11	00
10	00	01	10	11

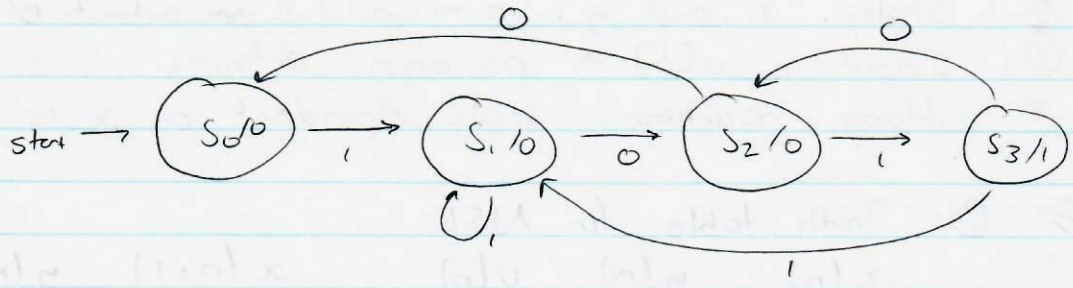
$t_1 = q_1 + \bar{x} q_0 \qquad t_0 = \bar{x} q_0 + x \bar{q}_0 = x \oplus q_0$



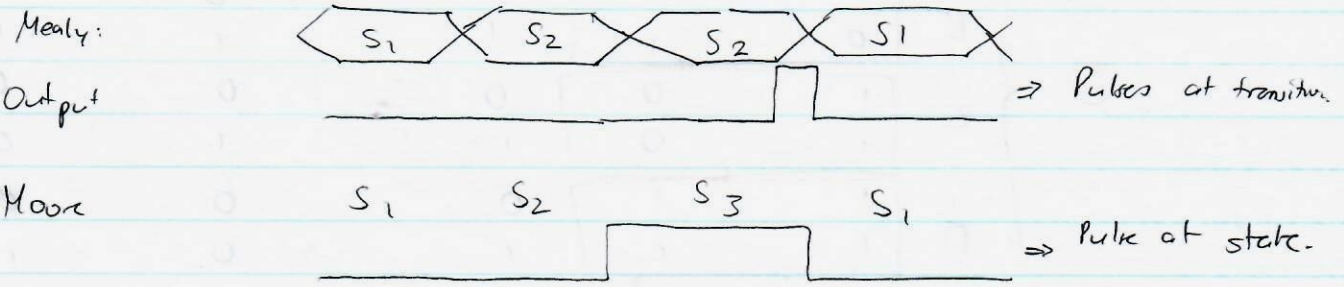


Refer to txtbk for JKFF implementation

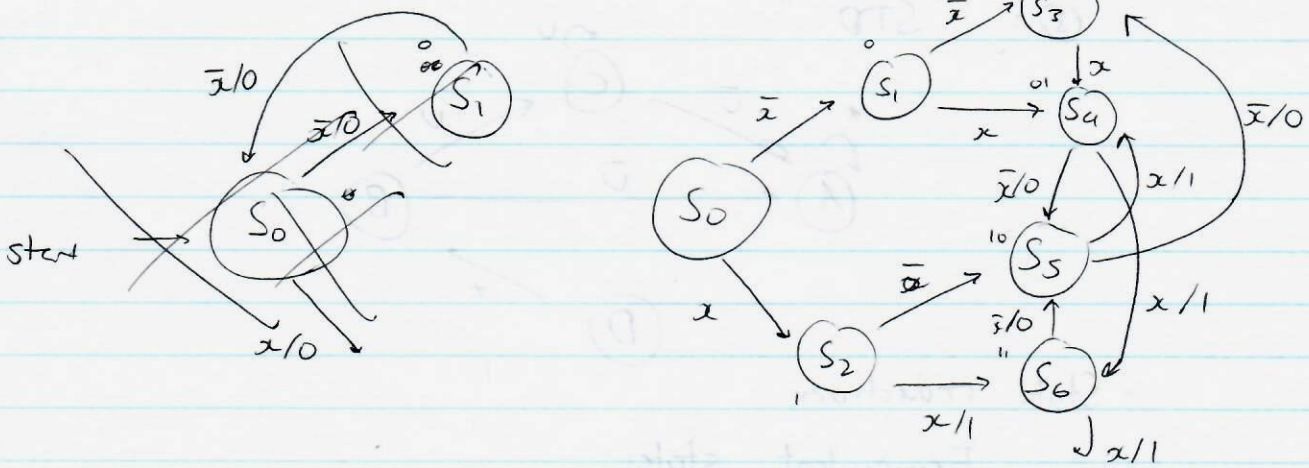
- Same example in Moore states:



Additional state needed vs. Mealy. Timing diagram makes more sense:



- Ex: // Mealy state diagram w/ 1 input x + 1 output z. x changes between clock edges. z = 1 ⇒ 2 of last 3 inputs are 1.



- Ex // Consider the equations below. Make STD

① : $x[n+1] = (x[n] \oplus y[n]) \cdot u[n]$

② : $y[n+1] = x[n] \cdot y[n]$

③ : $z[n] = x[n] + y[n]$

① : State machine: presence of $n+1$

② : States: x and y ($n \rightarrow n+1$) are outputs of NSL, inputs to FF

③ : Input is $v[n]$ ⇒ no eqn. defining

④ : Moore machine: only dependent on x, y

① : Truth table for NSL:

	$x(n)$	$y(n)$	$v(n)$	$x(n+1)$	$y(n+1)$
A	0	0	0	0	0
	0	0	1	0	0
B	0	1	0	0	0
	0	1	1	1	0
C	1	0	0	0	0
	1	0	1	1	0
D	1	1	0	0	1
	1	1	1	0	1

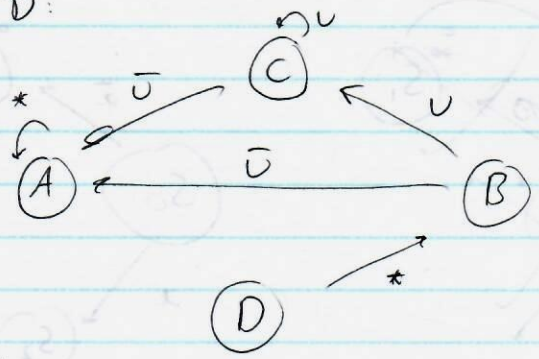
② : Fundamental realization:

x and y are <u>encodings of state</u>!

∴ 4 possible states: 00, 01, ~~10~~, 11

∴ Group into states

③ : STD:



- State reduction:

Equivalent states:

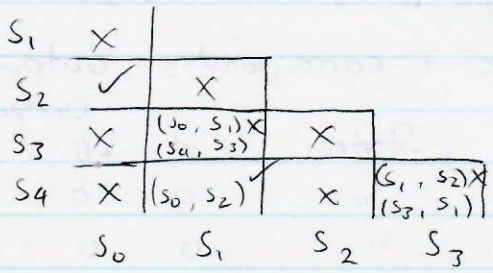
1. For all inputs, states give same output
2. For all inputs, states give same next states.

- Ex: //

Current State	Next State a=0	Next State a=1	Output a=0	Output a=1
S_0	S_3	S_2	1	1
S_1	S_0	S_4	0	0
⇒ S_2	S_3	S_0	1	1
S_3	S_1	S_3	0	0
⇒ S_4	S_2	S_1	0	0

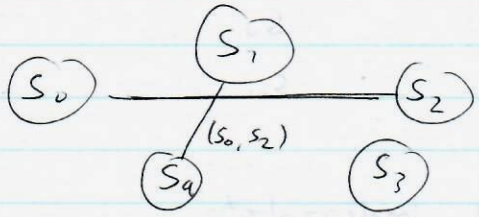
① Implication:

How do I create:
① Visualize full matrix
② Took out diagonal



1. Which states are def. not equivalent
   ↳ Compare output.
2. Which states are def. equivalent
   ↳ Look @ remaining states + look at NS2 ~~+ output~~
   ↳ If $S_i \to S_j$ + $S_j \to S_i$ for an input, that is OK.
3. Which states are condit. equivalent
   ↳ Look @ remains states + determine what states need to be equal for $S_i = S_j$
4. Look @ conditional states + determine equality by looking at state pairs in diagram

② Merger diagram:
① Draw all states:



② Draw edges based on implication (equal)
③ Merge cliques (state groups connected to each other)
↳ Can include states that we are not totally sure about (A9, Q4)!

- State encoding:

① : Minimum # of flip flops ⇐ minimum bits.

Take states + encode in binary.

∴ SM w/ 7 states ⇒ 7 states requires 3 bits

∴ Encode all states w/ 3 bits.

② : Output encoding:

Encoding of state = output (ex: // counters)

1) States + corresponding output

	Outputs			
States	A	B	C	D
S_0	0	0	0	0
S_1	0	0	0	0
S_2	0	1	0	0
S_3	1	0	0	0
S_4	1	0	0	1
S_5	1	0	1	0
S_6	1	0	1	1

We could use 4 flip flops (1 flip flop for each output)

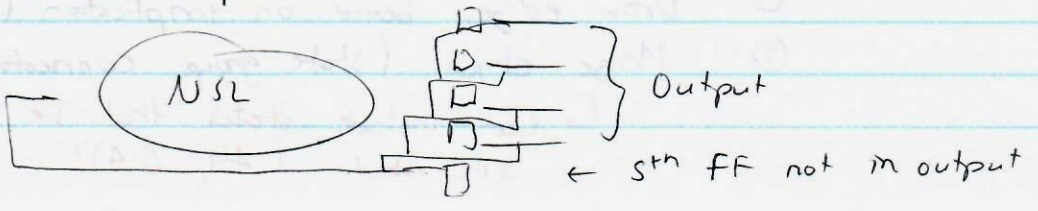
↳ Prob: S_0 and S_1 have same encoding → bad for NSL.

2) Increase complexity to prevent equivalent encodings.

↳ Introduce 5^{th} flip flop to distinguish S_0 + S_1

	E
S_0	0
S_1	1
	x
	x

Circuit implementation:



③. One-hot encodings (OFF)

S_0	000001	S_3	001000
S_1	000010	S_4	010000
S_2	000100	S_5	100000

} Notice that i^{th} bit of $S_i = 1$

Requires 1 flip flop for every state!

Ex:// N.S Output

C.S.	a=0	a=1	a=0	a=1
~~000~~	100	001	1	1
010	001	010	0	0
100	010	000	1	0

Trick to making NSL + OL from STD directly:
1. Associate each state w/ a state bit
2. Find out how to so into that state
3. Cook up via mapping states w/ state bits.

V. simple to find NSL:

$$q_0 = \bar{a}C_1 + aC_0$$
$$q_1 = \bar{a}C_2 + aC_1$$
$$q_3 = \bar{a}C_0 + aC_2$$

} Look @ 1's!

V. simple output logic:

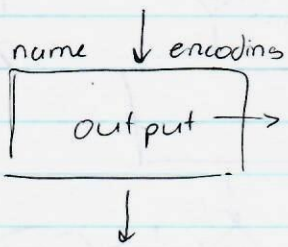
$Z = C_0 + \bar{a}C_2 \Rightarrow$ Considered 1s

ALGORITHMIC STATE MACHINES

ASM = alternatives to STD

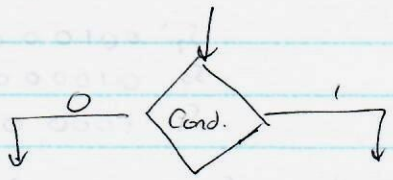
State boxes — Decision boxes — Conditional output boxes.

1. State box: state bubble in STD

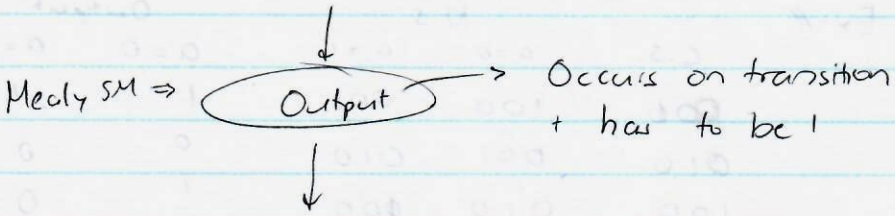


a) 1
b) Depends only on state

2. Decision box

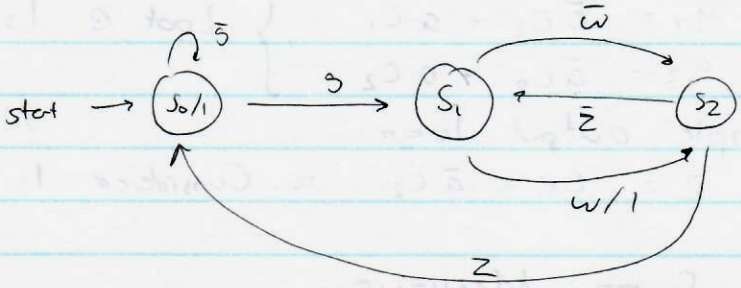


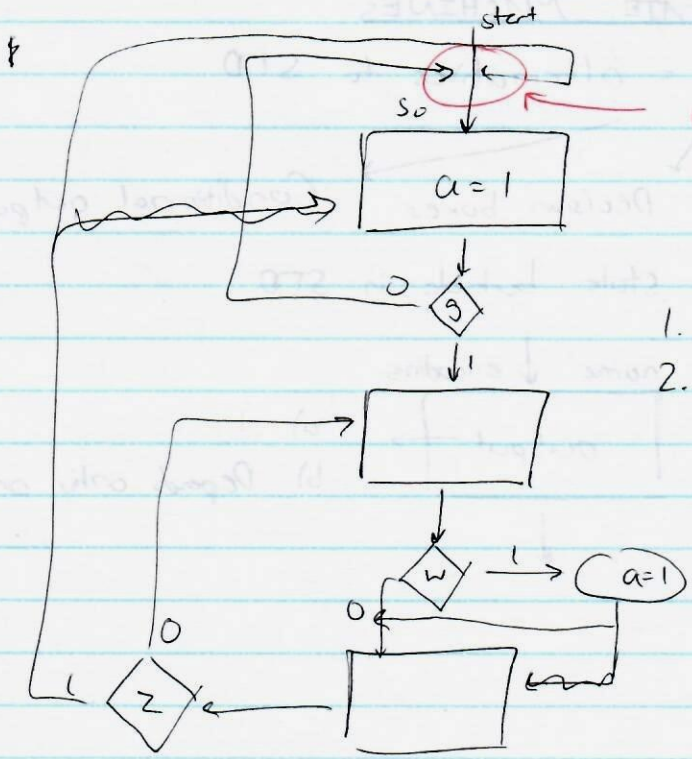
3. Conditional output box:



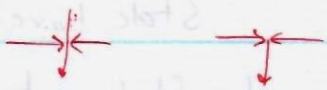
Applying same rules as sequential circuit design → ASM

- Ex:// Convert STD → ASM



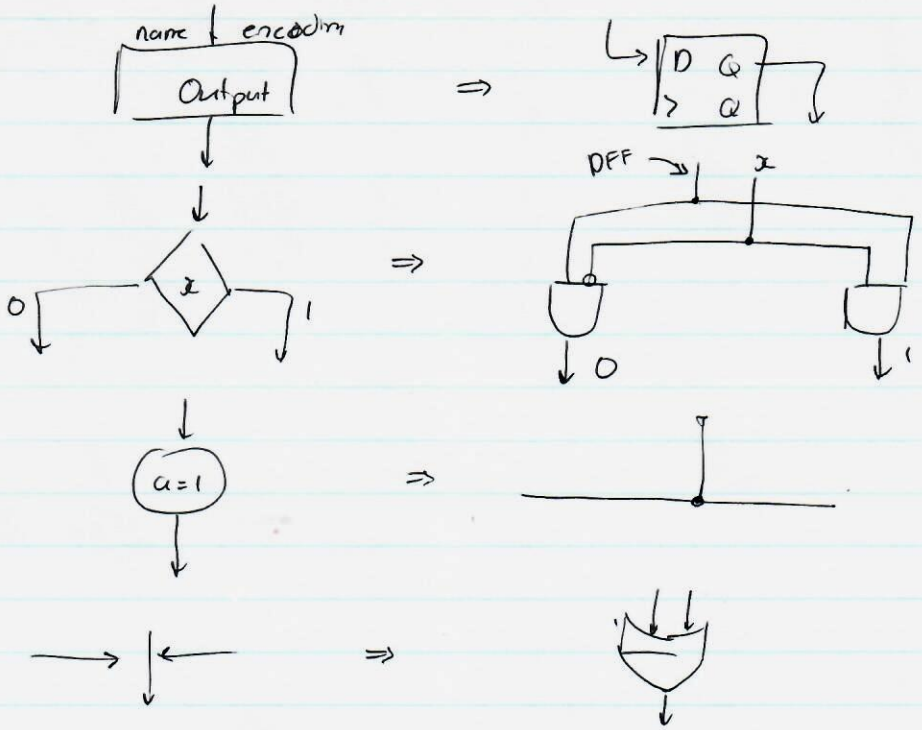


Can only have 1 input to state ⇒ combine signals.



1. Draw states
2. Look @ how to get out of state ← based on variables
   ↳ Decision boxes
3. If output ← transition, use conditional output box.

- Hardware implementation of ASM:



- Ex:// Convert prev. ASM into hardware