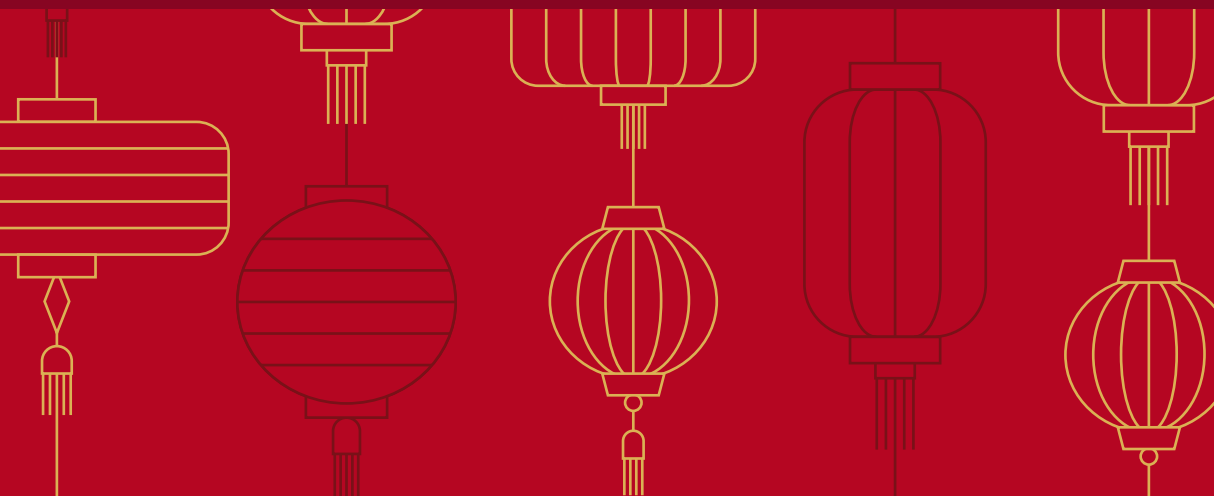


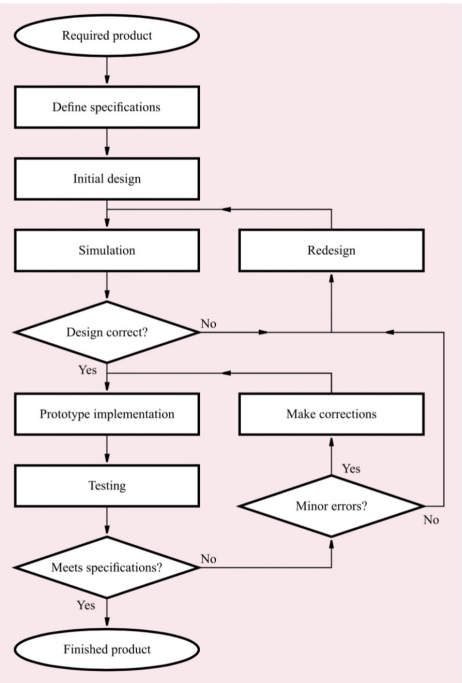
week 1

INTRODUCTION

- analog computers use continuously changeable aspects of physical phenomena (e.g. electrical quantities) to model problem
- digital computers rep varying quantities symbolically & by discrete values of both time & amplitude
  - ↳ signal encodes value instead of directly rep
- digital circuits are electronics that operate on digital signals
  - ↳ usually have 2 states: low (rep 0) & high (rep 1)
  - ↳ 3-valued logic used in high-density memory
- circuits are constructed from logic gates (small electronic circuits) used to create combinational logic
  - ↳ each logic gate performs function of boolean logic
  - ↳ gates are created from transistors (electrically controlled switches)

- digital computer is collection of many digital circuits
  - ↳ gates are organized into a set of packages (i.e. chips) that interconnect on motherboard
  - ↳ e.g. CPU, circuits for data storage, etc.
- transistors are electronic components that behave in non-linear/discrete manner
- combine transistors to form circuits w/ specific logic rules & these are logic gates
- combine logic gates to form more complex circuits
- combine complex circuits to form computer circuit

COMPUTING SOLUTIONS



- digital circuit design flow:
  - ↳ describe target behaviour
  - ↳ behaviour mapped into set of logical expressions
  - ↳ validate expressions
  - ↳ expressions mapped into circuit design based on network of elemental logic operators (e.g. AND, OR, NOT)
  - ↳ network of logic circuits is simulated & tested
  - ↳ prototype of circuit constructed from physical hardware
  - ↳ test prototype against desired behaviour
  - ↳ circuit is produced as final product

- to map real-world values to digital domain, some quantities may require sophisticated conversion

NUMBER REPRESENTATION

- common # radices (i.e. bases) are 2, 4, 8, 10, 16
- to write # in certain base, define coeffs a_i in radix r: $a_n r^n + a_{n-1} r^{n-1} + \dots + a_0 r^0 + a_{-1} r^{-1} + \dots + a_{-m} r^{-m}$
  - ↳ $0 \leq a_i < r$

General Radix Representation

r = 10 (Dec.)	r = 2 (Binary)	r = 8 (Octal)	r = 16 (Hex)
00	0000	00	0
01	0001	01	1
02	0010	02	2
03	0011	03	3
04	0100	04	4
05	0101	05	5
06	0110	06	6
07	0111	07	7
08	1000	10	8
09	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

- binary to octal requires grouping bits into groups of 3-bits
  - ↳ e.g. $(100\ 111\ 100\ 101)_2 = (4745)_8$
- binary to hex requires grouping bits into groups of 4-bits
  - ↳ e.g. $(1001\ 1110\ 0101)_2 = (9E5)_{16}$

in computers, rep of -ve #s often uses complements

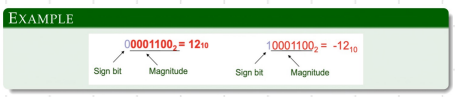
- 2 forms of complements for radix r:
  - ↳ r's complement (radix complement)
  - ↳ (r-1)'s complement (diminished radix complement)

NOTE
- if N = 0, then complement is just 0

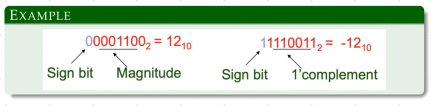
- for r's complement, given +ve # N w/ n digits so $N = (a_{n-1}\, a_{n-2} \ldots a_0)$, complement is $r^n - N$
  - ↳ another way is:
    - $\overline{a_i} = (r-1) - a_i$
    - $\overline{N} = \overline{a}_{n-1} \ldots \overline{a}_0 + 1$
- for (r-1)'s complement, given +ve # N w/ n digit integer part & m digit fractional part so $N = (a_{n-1}\, a_{n-2} \ldots a_0 . a_{-1}\, a_{-2} \ldots a_{-m})$, complement is $r^n - r^{-m} - N$
  - ↳ another way is:
    - $\overline{a_i} = (r-1) - a_i$
    - $\overline{N} = \overline{a}_{n-1} \ldots \overline{a}_0$
- complement of complement returns original #
- in binary # system:
  - ↳ 1's complement is obtained by flipping bits
  - ↳ 2's complement is obtained by flipping bits & adding 1

SIGNED NUMBERS

- both +ve & -ve #s are rep in same n-bit format & leftmost bit rep sign
  - ↳ 0 = +ve
  - ↳ 1 = -ve
- 3 common reps are:
  - ↳ sign & magnitude
  - ↳ signed 1's complement
  - ↳ signed 2's complement
- for +ve #s, all 3 have same rep
- in signed magnitude.



- in 1's complement:



↳ to get -ve of signed #, take 1's complement of whole # including sign bit

· in 2's complement:

EXAMPLE

$00001100_2 = 12_{10}$ $11110100_2 = -12_{10}$

Sign bit Magnitude Sign bit 2's complement

↳ to get -ve of signed #, take 2's complement of whole # including sign bit

rule for addition in 2's complement is to add 2 #s including their sign bits & discard any carry out of sign bit position

↳ subtraction is treated as addition of -ve #s rep in 2's complement

EXAMPLE

+6	00000110	- 6	11111010
+13	00001101	+13	00001101
+19	00010011	+7	00000111

+6	00000110	-6	11111010
-13	11110011	-13	11110011
-7	11111001	-19	11101101

↳ from above, to convert 1111 1001 to -7:

- flip all bits excluding sign: 0000110
- add 1: 0000111
- $(111)_2 = 7_{10}$

· subtraction by addition w/o sign bit:

↳ take 2's complement of 2nd #

↳ add 2nd # to 1st #

↳ if final carry of sum is 1, drop it & result is +ve

↳ if final carry of sum is 0, 2's complement of sum is result & it's -ve

week 2

NUMBER CONVERSION

- to convert decimal value into binary #:
  - ↳ integer divide decimal # by 2 & write down remainder
  - ↳ when quotient is 0, stop
  - ↳ write remainders from right to left

EXAMPLE

Convert 53 to binary

53	÷	2	=	26	→	1 Least significant bit
26	÷	2	=	13	→	0
13	÷	2	=	6	→	1
6	÷	2	=	3	→	0
3	÷	2	=	1	→	1
1	÷	2	=	0	→	1 Most significant bit

The result is that $53 = (110101)_2$

BINARY VARIABLES

- binary variables take on 2 discrete values: 0 & 1
  - ↳ 0 means open, no, false, etc.
  - ↳ 1 means closed, yes, true, etc.
- binary logic functions are expressions of binary variables or other functions that produce output of 0 or 1 as function of inputs
  - ↳ defined using truth tables that tells us value of function for each possible set of input values
- 3 basic logic operations: AND, OR, & NOT

THE LOGICAL OPERATORS AND OR NOT HAVE SYMBOLS

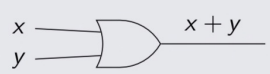
Logic Operator	Symbol	Example
AND	$\bullet$, $\wedge$, nothing	$f = x \bullet y$, $f = xy$
OR	$+$,$\vee$	$f = x + y$
NOT	!,',$\neg$,overbar	$!x$, x',$\neg x$,$\bar{x}$

  - ↳ orders have precedence in this order: ( ), then NOT, then AND, then OR
  - ↳ AND gate:

AND GATE

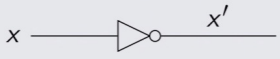
OPERATOR	Graphical SYMBOL	TRUTH TABLE
AND (2-inputs)		x y output 0 0 0 0 1 0 1 0 0 1 1 1

  - ↳ OR gate:

OPERATOR	Graphical SYMBOL	TRUTH TABLE
OR (2-inputs)		x y Output 0 0 0 0 1 1 1 0 1 1 1 1

↳ NOT gate

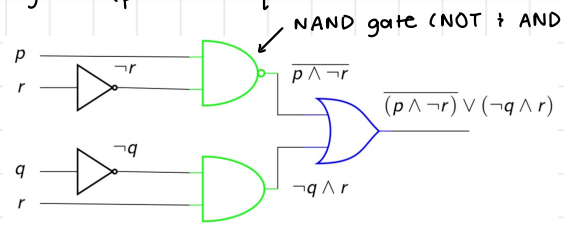
NOT GATE

OPERATOR	Graphical SYMBOL	TRUTH TABLE
NOT		x | f 0 | 1 1 | 0

- AND gate can have any # of inputs >= 2
  - ↳ only outputs 1 when all inputs are 1
- OR gate can have any # of inputs >= 2
  - ↳ outputs 1 when any of the inputs are 1

LOGIC FUNCTIONS

- can write logic function down algebraically
  - ↳ can also rep w/ truth table & schematic diagram
- e.g. $f = \overline{(p \wedge \neg r)} \vee (\neg q \wedge r)$



FROM LOGIC FUNCTION TO TRUTH TABLES

- The logic function and the truth table provide exactly the same information. While the truth table shows the value of a logic function for a given setting of the inputs, the logic function evaluates to the same value.

x	y	$f = !x \bullet !y + x \bullet y$		x	y	f
0	0	$f = !0 \bullet !0 + 0 \bullet 0 = 1 \bullet 1 + 0 = 1$		0	0	1
0	1	$f = !0 \bullet !1 + 0 \bullet 1 = 1 \bullet 0 + 0 = 0$	$\Longrightarrow$	0	1	0
1	0	$f = !1 \bullet !0 + 1 \bullet 0 = 0 \bullet 1 + 0 = 0$		1	0	0
1	1	$f = !1 \bullet !1 + 1 \bullet 1 = 0 + 1 = 1$		1	1	1

- interesting function: linguistically, the function is on, or true, only if both variables are equal. Comparator- that is.

BOOLEAN ALGEBRA

- axioms / postulates of Boolean algebra, which is defined by a set of elements B & 2 binary operators (+ & */·):
  1) a) if $x, y \in B$, then $x + y \in B$
     b) if $x, y \in B$, then $x \cdot y \in B$
  2) a) $x + 0 = x$
     b) $x \cdot 1 = x$
  3) a) $x + y = y + x$
     b) $x \cdot y = y \cdot x$
     ↳ commutative
  4) a) $x \cdot (y + z) = x \cdot y + x \cdot z$
     b) $x + y \cdot z = (x + y) \cdot (x + z)$
     ↳ distributive
  5) a) $x + x' = 1$
     b) $x \cdot x' = 0$
  6) there exists at least 2 elements $x, y \in B$ st $x \neq y$
- theorems in Boolean algebra:
  1) a) $x + x = x$
     b) $x \cdot x = x$

2) a) $x + 1 = 1$

b) $x \cdot 0 = 0$

3) $(x')' = x$

↳ involution

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

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

↳ associative

5) a) $(x + y)' = x' \cdot y'$

b) $(x \cdot y)' = x' + y'$

↳ DeMorgan's Laws

6) a) $x + x \cdot y = x$

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

↳ absorption

TIP
· absorption law:
$xy + x' = y + x'$

· DeMorgan's Laws:

↳ $(x + y) = (x'\ y')'$

↳ $(x \cdot y) = (x' + y')'$

↳ $(x' \cdot y') = (x + y)'$

↳ $(x' + y) = (x \cdot y')'$

· Consensus Theorem: consensus term of disjunction is defined when terms in function are reciprocals to each other (third term is taken out)

↳ normal form (sum of products around A): $AB + \bar{A}C + BC = AB + \bar{A}C$

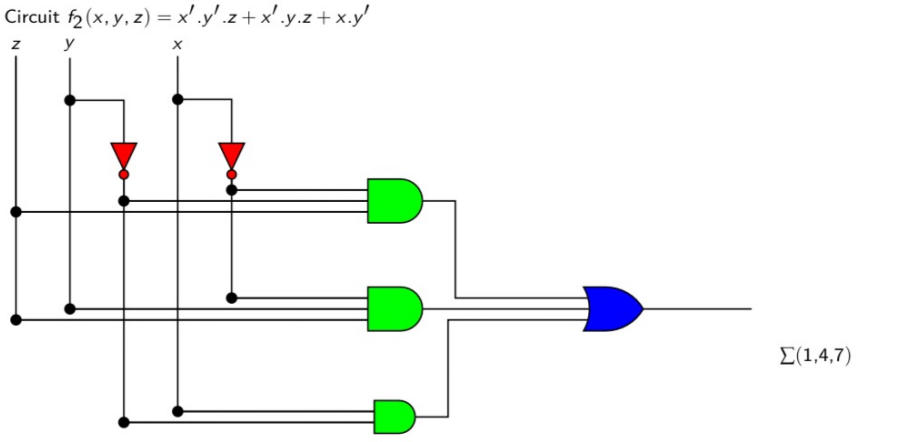
↳ dual form (product of sums around A): $(A+B)(\bar{A}+C)(B+C) = (A+B)(\bar{A}+C)$

° POS around B: $(A+B)(\bar{B}+C)(A+C) = (A+B)(\bar{B}+C)$

BASIC TERMINOLOGY AND CANONICAL FORMS

· synthesis is implementation of circuit

↳ draw circuits implementing function using logic gate symbols



● Cost: 3 AND + 1 OR + 11 GATE INPUTS = 15.

↳ inverters at beginning of circuit can be excluded from cost b/c ground can be used to switch input of 1 to 0

↳ to reduce cost, use Boolean algebra: $f_2 = x' \cdot y' \cdot z + x' \cdot y \cdot z + x \cdot y'$

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

$= x' \cdot z(1) + x \cdot y'$

$= x' \cdot z + x \cdot y'$

° cost is 2 AND + 1 OR + 6 GATE INPUTS = 9

· x is binary variable when it has values only in {0, 1}

↳ i.e. $x \in \{0, 1\}$

- uncomplemented version of x is +ve literal (i.e. x)
- complemented version of x is -ve literal (i.e. !x)
- truth table for an n-input function will have 2n rows
  - ↳ **minterm**: for each row, if variable is 1, include +ve literal; if variable is 0, include -ve literal
    - AND of each literal is minterm, denoted by m
  - ↳ **maxterm**: for each row, if variable is 1, include -ve literal; if variable is 0, include +ve literal
    - OR of each literal is maxterm, denoted by M
- e.g.

			Minterms	
x	y	z	Term	Designation
0	0	0	$x'y'z'$	m_0
0	0	1	$x'y'z$	m_1
0	1	0	$x'yz'$	m_2
0	1	1	$x'yz$	m_3
1	0	0	$xy'z'$	m_4
1	0	1	$xy'z$	m_5
1	1	0	xyz'	m_6
1	1	1	xyz	m_7

			Maxterms	
x	y	z	Term	Designation
0	0	0	$x+y+z$	M_0
0	0	1	$x+y+z'$	M_1
0	1	0	$x+y'+z$	M_2
0	1	1	$x+y'+z'$	M_3
1	0	0	$x'+y+z$	M_4
1	0	1	$x'+y+z'$	M_5
1	1	0	$x'+y'+z$	M_6
1	1	1	$x'+y'+z'$	M_7

  - ↳ when particular input pattern appears, associated minterm will be 1 while all other minterms will evaluate to 0
    - e.g. x=0, y=0, z=1 will make $m_1 = x'y'z = 1$ & all other minterms 0
    - same for maxterms except associated maxterm evaluates to 0 & others to 1
- minterms & maxterms are **duals** of each other
  - ↳ $!m_i = M_i$
  - ↳ $!M_i = m_i$

- **canonical sum-of-products (SOP)** rep of function: take OR/sum of minterms for which a function is 1

EXAMPLE

x	y	z	f_1	f_2
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$\begin{aligned} f_1 &= m_1 + m_4 + m_7 \\ &= x'y'z + xy'z' + xyz \\ f_2 &= m_3 + m_5 + m_6 + m_7 \\ &= x'yz + xy'z + xyz' + xyz \end{aligned}$$

Shortcut notation:

$$\begin{aligned} f_1 &= \sum(1,4,7) \\ f_2 &= \sum(3,5,6,7) \end{aligned}$$

  - ↳ when implemented w/ gates, SOP will have form:
    1) plane of NOT gates (to generate literals)
    2) plane of AND gates (to implement minterms)
    3) single OR gate (to take sum)

canonical product-of-sums (POS) rep of function: take AND/product of maxterms for which a function is 0

EXAMPLE

Canonical POS

x	y	z	f_1	f_2
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$\begin{aligned} f_1 &= M_0M_2M_3M_5M_6 \\ &= (x+y+z)(x+y'+z)(x+y'+z')(x'+y+z')(x'+y'+z) \end{aligned}$$

f_2 = ???

Shortcut notation:

$$f_1 = \Pi(0,2,3,5,6)$$

↳ e.g. $f_2 = M_0 M_1 M_2 M_4$

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

$= \Pi(0,1,2,4)$

↳ when implemented w/ gates, POS will have form:

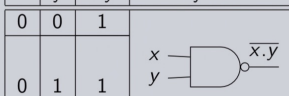
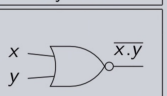
1) plane of NOT gates (to generate literals)
2) plane of AND gates (to implement maxterms)
3) single AND gate (to take product)

- SOP & POS often referred to as **2-level logic implementations**
  - ↳ NOT gates at input are free
- to **convert** from SOP to POS, change Σ to Π & list indices missing from original
  - ↳ e.g. $f_1 = \Sigma(1,4,7) = \Pi(0,2,3,5,6)$
  - ↳ vice versa for conversion from POS to SOP
- to reduce cost, start w/ canonical SOP/POS & use Boolean algebra to simplify it to a **standard SOP/POS**

OTHER LOGIC GATES

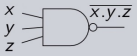
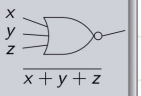
- **NAND** is combo of NOT-AND & **NOR** is combo of NOT-OR

OTHER LOGIC GATES: NAND AND NOR

x	y	$\overline{x.y}$	Symbol	x	y	$\overline{x+y}$	Symbol
0	0	1		0	0	1	
0	1	1		0	1	0	
1	0	1		1	0	0	
1	1	0		1	1	0	

↳ can have multiple inputs:

THREE INPUTS: NAND AND NOR

x	y	z	$\overline{x.y.z}$	Symbol	x	y	z	$\overline{x+y+z}$	Symbol
0	0	0	1		0	0	0	1	
0	0	1	1		0	0	1	0	
1	1	0	1		1	1	0	0	
1	1	1	0		1	1	1	0	

- **XOR** & **XNOR** gates are useful for arithmetic operations

XOR AND NXOR

- XOR gate (with 2-inputs performs a "difference operation"):

x	y	$f = x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

- NXOR gate (with 2-inputs performs a "equivalence operation"):

x	y	$f = \overline{x \oplus y}$
0	0	1
0	1	0
1	0	0
1	1	1

↳ XOR gate w/ 2+ inputs performs "odd operation" (i.e. output is 1 whenever odd # of inputs are 1)

EXAMPLE

Multiple Inputs XOR

x_1	x_2	x_3	$f_{XOR} = x_1 \oplus x_2 \oplus x_3$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

↳ XNOR gate w/ 2+ inputs performs "even operation" (i.e output is 1 whenever even # of inputs are 1)

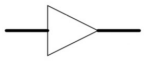
NXOR GATES WITH MULTIPLE INPUTS

Example: 3-input versions:

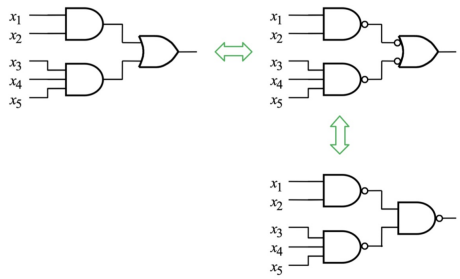
x_1	x_2	x_3	$f_{XNOR} = \overline{x_1 \oplus x_2 \oplus x_3}$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

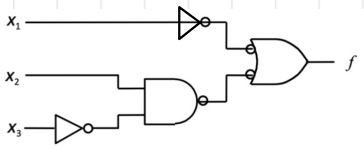
- buffers don't do anything logically & used to boost signal's strength

x	$f = x$
0	0
1	1

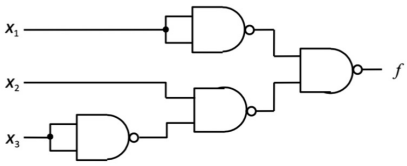


- associative gate means we can collapse many smaller gates into a single gate w/ multiple inputs
  - ↳ AND, OR, & XOR gates are associative
- NAND, NOR, & NXOR gates are not associative
- equivalence means circuits produce same logical output
  - ↳ circuits may not be equal
  - ↳ e.g. SOP implementation using NAND



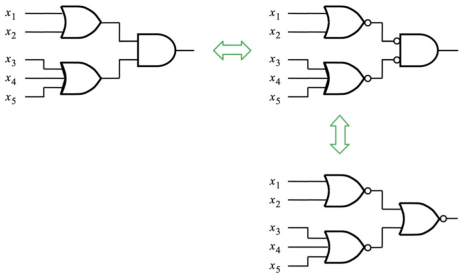


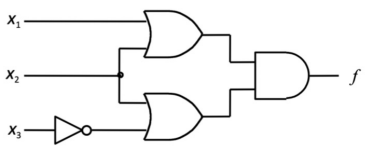
SOP implementation



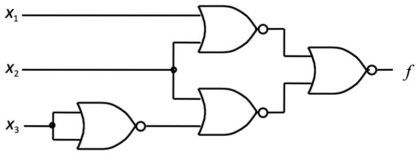
NAND implementation

  - ↳ e.g. POS implementation using NOR





POS implementation



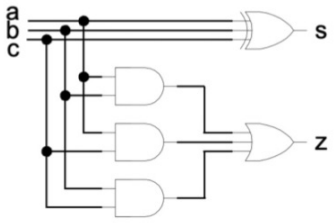
NOR implementation

CAD TOOLS AND VHDL

- computer-aided design (CAD) is use of computers to aid in creation, modification, analysis, or optimization of design

- VHDL stands for VHSIC Hardware Description Language
  - ↳ VHSIC stands for Very High Speed Integrated Circuit

```
-- VHDL code for First Look

library ieee;
use ieee.std_logic_1164.all;

entity FullAdder is
    port( a,b,c : in  std_logic;
          s,z   : out std_logic);
end FullAdder;

architecture prototype of FullAdder is
begin
    s   <= a xor b xor c;                          -- equation for s
    z   <= (a and b) or (a and c) or (b and c);    -- equation for z
end prototype;
```



  - ↳ comments are preceeded w/ " -- "
- VHDL uses data objects to capture info
  - ↳ signals (i.e. wires in circuit)
  - ↳ constants
  - ↳ variables
- signals can be declared in 3 places:
  - ↳ port of an entity declaration
  - ↳ declarations section of architecture description
  - ↳ declarations section of package
- naming convention for signals:
  - ↳ begin w/ letter
  - ↳ can't have 2 successive underscores
  - ↳ can't end w/ underscore
  - ↳ can't be VHDL reserved word
  - ↳ case insensitive
- entity declaration describes interface to rest of world

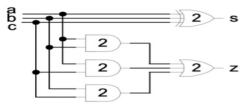
```
ENTITY entity_name IS
    PORT(
        SIGNAL signal_name : mode type ;
        SIGNAL signal_name : mode type ;
        ...
        SIGNAL signal_name : mode type ) ;
END entity_name;
```

  - ↳ port is where all inputs & outputs of circuit are listed
- architecture definition describes particular implementation of circuit

```
ARCHITECTURE architecture_name OF entity_name IS
    -- declarative section
    [SIGNAL declarations]
    [CONSTANT declarations]
    [TYPE declarations]
    [COMPONENT declarations]
    [ATTRIBUTE declarations]
BEGIN
    -- implementation
    [COMPONENT instantiation statements]
    [CONCURRENT ASSIGNMENT statements]
    [PROCESS statements]
    [GENERATE statements]
END architecture_name ;
```

- since real circuits w/ gates have delays, use VHDL to specify them:

```
-- VHDL code for First Look

library ieee;
use ieee.std_logic_1164.all;

entity FullAdder is
    port( a,b,c : in  std_logic;
          s,z   : out std_logic);
end FullAdder;

architecture prototype of FullAdder is
begin
    s   <= a xor b xor c                          after 2 ns; -- equation for s
    z   <= (a and b) or (a and c) or (b and c)    after 4 ns; -- equation for z
end prototype;
```

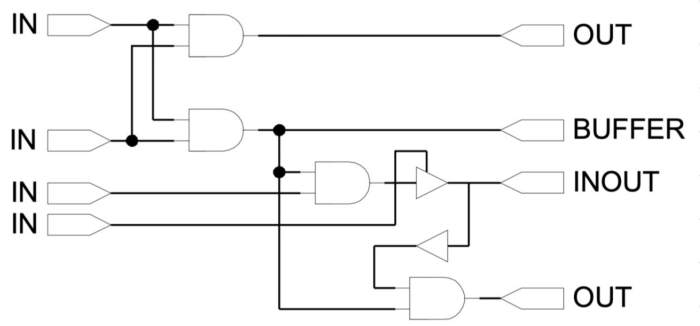


- VHDL descriptions may contain various types of operators
  - ↳ Boolean: AND, OR, NOT, XOR, NAND, NOR, etc.
  - ↳ relational: =, /=, <, >, etc.
  - ↳ arithmetic: +, -, &, *, /, **, etc.
- to assign values to signals, use concurrent signal assignment "<="
  - ↳ since all circuit components operate in parallel, order of assignments isn't important
  - ↳ i.e. freeze circuit at time t & do all calculations w/ values of signals at t; then new values will be concurrently assigned to signals at time t + Δt
  - ↳ e.g following order of signals is equivalent

```
library ieee;
use ieee.std_logic_1164.all;

entity Concurrent is
    port( a,b,c : in    std_logic;
          d,e   : inout std_logic);
end Concurrent;

architecture prototype of Concurrent is
begin
   e   <= c  or d  after 2 ns;
   d   <= a and b  after 2 ns;
end prototype;
```

```
library ieee;
use ieee.std_logic_1164.all;

entity Concurrent is
    port( a,b,c : in    std_logic;
          d,e   : inout std_logic);
end Concurrent;

architecture prototype of Concurrent is
Begin
   d   <= a and b  after 2 ns;
   e   <= c  or d  after 2 ns;
end prototype;
```

- when declared inside port of entity declaration, signal must be given one of 4 modes:
  - ↳ IN: data flows into circuit
  - ↳ OUT: data flows out of circuit
  - ↳ BUFFER: data flows out of circuit but is used internally in circuit
  - ↳ INOUT: data flows both into & out of circuit



- signals must also have type (i.e. state of wire)
  - ↳ signal type std_ulogic follows standard 9-value system below:

```
type std_ulogic is (
                'U',        -- Uninitialized
                'X',        -- Forcing Unknown
                '1',        -- Forcing 1
                '0',        -- Forcing 0
                'Z',        -- High impedance
                'W',        -- Weak Unknown
                'L',        -- Weak 0
                'H',        -- Weak 1
                '-'         -- Don't care
        );
```

↳ can also have multiple sources driving wire & if they have diff values, need to decide on resolved type

- signal type `std_logic` for multiple drivers on signal:

	U	X	0	1	Z	W	L	H	-
U	U	U	U	U	U	U	U	U	U
X	U	X	X	X	X	X	X	X	X
0	U	X	0	X	0	0	0	0	X
1	U	X	X	1	1	1	1	1	X
Z	U	X	0	1	Z	W	L	H	X
W	U	X	0	1	W	W	W	W	X
L	U	X	0	1	L	W	L	W	X
H	U	X	0	1	H	W	W	H	X
-	U	X	X	X	X	X	X	X	X

· always identify library & package before every VHDL entity declaration:

```
library ieee;
use ieee.std_logic_1164.all;
```

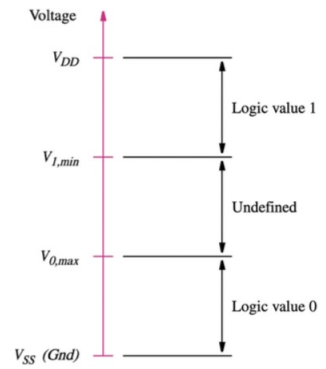
· e.g. implement SOP expression $f = \Sigma(0,1,3,4,5)$ via VHDL description

$$f = \sum(0,1,3,4,5)$$

```
-- VHDL code for Second Simple Example

library ieee;
use ieee.std_logic_1164.all;

entity SomeFunction is
    port( a,b,c : in  std_logic;
          f     : out std_logic);
end SomeFunction;

architecture prototype of SomeFunction is
begin
    f <=     ((not a) and (not b) and (not c))   -- m0
          or ((not a) and (not b) and c)         -- m1
          or ((not a) and b       and c)         -- m3
          or (a       and (not b) and (not c))   -- m4
          or (a       and (not b) and c) ;       -- m5
end prototype;
```

IMPLEMENTATION TECHNOLOGY

· to rep 2 logic values as voltage levels:



↳ V_{SS} is 0V, corresponding to ground

↳ V_{DD} rep power supply, usually btwn 1V - 5V

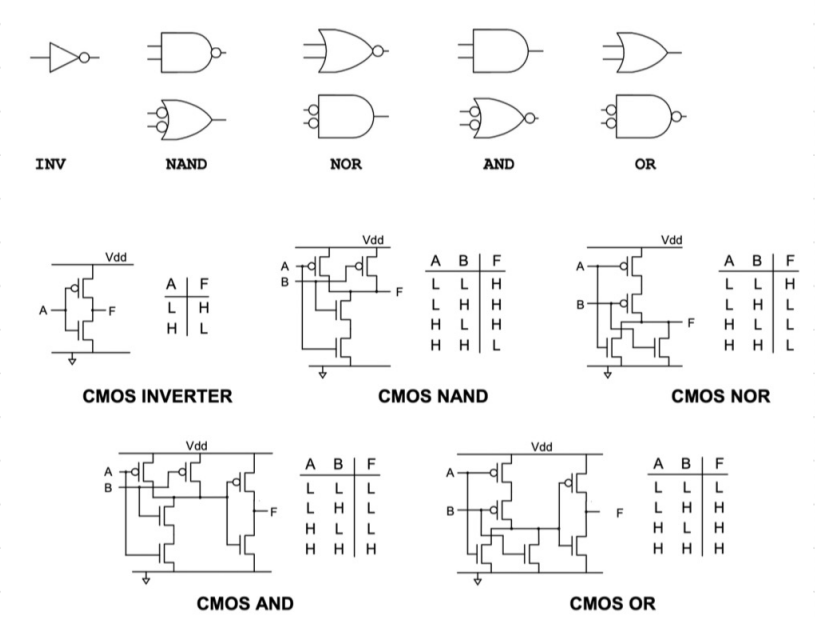
· logic circuits are built w/ transistors, which act like simple switches

↳ most popular type of transistor is metal oxide semiconductor field-effect transistor (MOSFET)

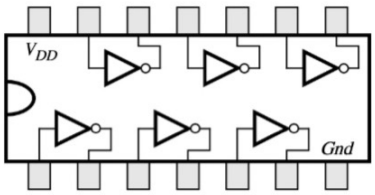
- type 1 is n-channel (aka NMOS)
- type 2 is p-channel (aka PMOS)

· can build FET circuits to implement logic relationships

↳ below are basic CMOS gates w/ truth tables



- 7400-series parts are chips that each contain a few logic gates
  - ↳ 2 pins are used to connect to V_{DD} & Gnd while others provide connections to specified gate
  - ↳ e.g. 7404 chip has 6 NOT gates



(b) Structure of 7404 chip

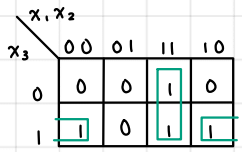
week 3

KARNAUGH MAPS

- all Boolean expressions can be converted into 2 standard forms: SOP or POS
  - ↳ canonical form specifies unique rep of Boolean function
  - ↳ normal form simplifies expression
- Karnaugh Maps (K-maps) allow us to minimize functions graphically for small logic functions (≤ 5 inputs)
- when constructing a K-map:
  - ↳ coordinates of square depend on values of inputs
  - ↳ content of square is function's output
  - ↳ grey code counting: when labelling rows & columns, only 1 variable can change btwn adjacent rows/columns
- steps to using a K-map for minimized SOP:
  - ↳ circle groups of 1s (i.e. minterms) w/rectangles so all 1s are covered
    - try to form largest possible groups so there's fewer literals & product terms
    - groups must be powers of 2
    - groups can wrap around rows/columns
    - it's okay to circle a 1 multiple times
  - ↳ if value of input changes within grouping, it's independent & not included in product term
  - ↳ look for inputs where their values stay the same
    - if input is opp Boolean value as output, invert input variable
  - ↳ one group forms a product & OR all groups tgt to get simplified SOP
- e.g. use K-map to create SOP function for below truth table:

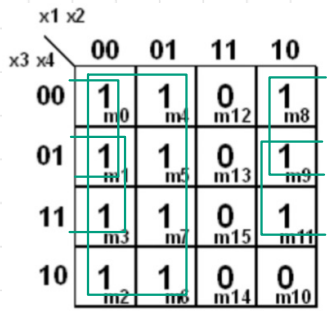
x_1	x_2	x_3	f	
0	0	0	0	m_0
0	0	1	1	m_1
0	1	0	0	m_2
0	1	1	0	m_3
1	0	0	0	m_4
1	0	1	1	m_5
1	1	0	1	m_6
1	1	1	1	m_7

SOLUTION



Thus, SOP of function is $f = x_1x_2 + x_2'x_3$.

- e.g working w/4 inputs:



$f = x_1' + x_2'x_3' + x_2'x_4$

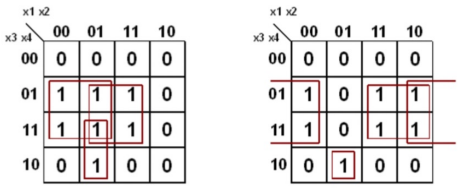
- when using K-maps to find minimized POS, use same process as SOP except:
  - ↳ circle 0s for maxterms
  - ↳ AND all the resulting sum terms
- when we don't care what output is, mark w/ X instead of 0/1
  - ↳ in K-maps, force Xs into 0/1, depending on how it helps us reach min rep w/fewest

possible literals & gates

- can get multiple expressions using K-map that are logically equivalent but not algebraically equal

MULTIPLE OUTPUT/ FUNCTION MINIMIZATION

- in absence of parentheses, operations have order of NOT, AND, then OR
- sometimes, when minimizing multiple functions, overall circuit implementation will be better if we consider both functions simultaneously rather than minimizing individually
  - ↳ i.e. find common product terms for both functions to share
  - ↳ individual expressions may not be the best but when considering shared product terms, total cost of circuit is minimized
- e.g. $f_1 = x_1'x_4 + x_2x_4 + x_1'x_2x_3$

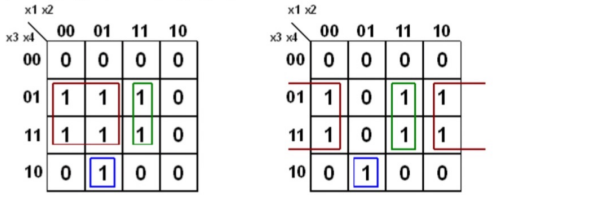


$f_2 = x_1x_4 + x_2'x_4 + x_1'x_2x_3x_4'$

NOTE
- gates can have multiple inputs

  - ↳ cost of f_1 = 3 AND + 1 OR + 10 gate inputs
    = 14
  - ↳ cost of f_2 = 3 AND + 1 OR + 11 gate inputs
    = 15
  - ↳ to implement entire circuit, cost is 29

$f_1 = x_1'x_4 + x_1x_2x_4 + x_1'x_2x_3x_4'$



$f_2 = x_2'x_4 + x_1x_2x_4 + x_1'x_2x_3x_4'$

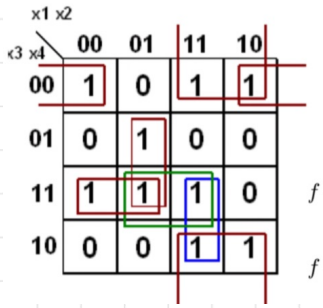
  - ↳ there's 2 shared product terms so entire circuit cost = 4 AND + 2 OR + 17 gate inputs
    = 23

IMPLICANTS

- function is completely specified when we're told output is 0/1 for every possible input
  - ↳ when there's "don't care" outputs, function is incompletely specified
- minterms can be separated into 3 sets:
  - ↳ on-set: require output to be 1
  - ↳ off-set: require output to be 0
  - ↳ dc-set: output of X (i.e. don't care)
- implicant: product term where logic function outputs 1 for all minterms in term
  - ↳ all rectangles in K-maps that contain only 1s are implicants
- prime implicant: removing any literal from implicant results in new product term that isn't implicant
  - ↳ removing literal is same as doubling area of rectangle on K-map
  - ↳ if we can't inc rectangle w/o including 0, implicant is prime
- cover: collection of implicants that account for all cases in which function is 1
  - ↳ can always form cover using only prime implicants

essential prime implicant: includes minterm that isn't found in any other prime implicant

- to minimize logic functions:
  - ↳ generate prime implicants
  - ↳ identify essential prime implicants
  - ↳ include all essential prime implicants & if function is covered, stop
  - ↳ include as few non-essential prime implicants to cover function if necessary
- e.g.



$$f = x'_2x'_3x'_4 + x'_1x_2x_4 + x'_1x_3x_4 + x_1x'_4 + x_1x_2x_3$$

$$f = x'_2x'_3x'_4 + x'_1x_2x_4 + x'_1x_3x_4 + x_1x'_4 + x_2x_3x_4$$

  - ↳ above K-map has 4 essential prime implicants & 2 non-essential ones

XOR GATE

if we can extract XOR gate, can get large savings in circuit size

- can implement XORs & XNORs using AND, OR, & NOT gates:
  - ↳ $f_{XOR} = x_1 \oplus x_2$
    $= \overline{x}_1 x_2 + x_1 \overline{x}_2$
  - ↳ $f_{XNOR} = \overline{x_1 \oplus x_2}$
    $= (x_1 + \overline{x}_2) \cdot (\overline{x}_1 + x_2)$
    $= x_1\overline{x}_1 + x_1x_2 + \overline{x}_1\overline{x}_2 + \overline{x}_2x_2$
    $= x_1x_2 + \overline{x}_1\overline{x}_2$
- XOR & XNOR gates have checker-board patterns in K-maps

x1x2 / x3x4	00	01	11	10
00	0	1	0	1
01	1	0	1	0
11	0	1	0	1
10	1	0	1	0

$$f_{XOR} = x_1 \oplus x_2 \oplus x_3 \oplus x_4$$

x1x2 / x3x4	00	01	11	10
00	1	0	1	0
01	0	1	0	1
11	1	0	1	0
10	0	1	0	1

$$f_{XNOR} = \overline{x_1 \oplus x_2 \oplus x_3 \oplus x_4}$$

- when writing function as XOR of several product terms, called Exclusive Sum-of-Products (ESOP)
  - ↳ e.g. $f = \overline{x}_1x_2\overline{x}_3\overline{x}_4 + \overline{x}_1x_2x_3x_4 + x_1\overline{x}_3x_4 + x_1x_3\overline{x}_4 + \overline{x}_2\overline{x}_3x_4 + \overline{x}_2x_3\overline{x}_4$
    $= (\overline{x}_1x_2)(\overline{x}_3\overline{x}_4 + x_3x_4) + x_1(\overline{x}_3x_4 + x_3\overline{x}_4) + \overline{x}_2(\overline{x}_3x_4 + x_3\overline{x}_4)$
    $= (\overline{x}_1x_2)(\overline{x}_3\overline{x}_4 + x_3x_4) + (x_1 + \overline{x}_2)(\overline{x}_3x_4 + x_3\overline{x}_4)$
    $= (\overline{x}_1x_2)(\overline{x_3 \oplus x_4}) + (x_1 + \overline{x}_2)(x_3 \oplus x_4)$
    $= (\overline{x}_1x_2)(\overline{x_3 \oplus x_4}) + (\overline{\overline{x}_1x_2})(x_3 \oplus x_4)$
    $= (\overline{x}_1x_2) \oplus (x_3 \oplus x_4)$
    $= \overline{x}_1x_2 \oplus x_3 \oplus x_4$

QUINE-MCCLUSKEY (TABULAR) METHOD

- if given function $f = x_1\overline{x}_2x_3\overline{x}_4x_5x_6 + x_1x_2\overline{x}_3\overline{x}_4\overline{x}_5x_6$ (straight-fwd implementation requires AND gates w/ fan-in of 6), & given gates w/ max fan-in of 4, factor f as $f = x_1\overline{x}_4x_6(x_2\overline{x}_3\overline{x}_5)$
- basis of tabular & K-map methods is combining property of Boolean algebra: $xy + xy' = x$
  - ↳ 2 terms combine if they differ in exactly 1 variable

- when # of variables is large, use **Quine-McCluskey tabular method** to get min SOP form
- process of tabular method:
  1) arrange given minterms in ascending order & group them based on # of 1s in binary rep
     - there will be at most n+1 groups if there's n Boolean variables
     - if there's any don't cares, treat as minterms
  2) compare minterms in successive groups; if they differ in only one-bit pos, merge pair & write 'x' in differing bit pos
     - put check beside minterms that have been merged
  3) repeat step 2 until we get all prime implicants (i.e. no more merging can occur)
     - prime implicants (PI) are any terms that weren't checked off
  4) create prime implicant table where each prime implicant is a row & each minterm is column; place check where PI includes minterm
     - exclude don't cares from 1st list
     - if there's single check in column, PI that covers the minterm is essential (EPI)
  5) remove rows w/ EPIs & columns covered by them
  6) find necessary non-EPIs & remove their rows & columns covered by them; keep going until cover table is empty or no further
     - when column dominates (i.e. covers same minterms but w/ more checks), remove it
     - when row is being dominated (i.e. covers same minterms but w/ less checks), remove it
- e.g. find min-cost implementation of $f(x_1, \ldots, x_4) = \Sigma m(0, 4, 8, 10, 11, 12, 13, 15)$

SOLUTION

$m_i - x_1x_2x_3x_4$	# of 1s	minterm m_i	4-L implicants (4-literal)	
0 - 0 0 0 0	0	0	0000	✓
4 - 0 1 0 0	1	4	0100	✓
8 - 1 0 0 0		8	1000	✓
10 - 1 0 1 0	2	10	1010	✓
11 - 1 0 1 1		12	1100	✓
12 - 1 1 0 0	3	11	1011	✓
13 - 1 1 0 1		13	1101	✓
15 - 1 1 1 1	4	15	1111	✓

combined minterms	3-L implicants	
0, 4	0x00	✓
0, 8	x000	✓
4, 12	x100	✓
8, 10	10x0	
8, 12	1x00	✓
10, 11	101x	
12, 13	110x	
11, 15	1x11	
13, 15	11x1	

combined minterms	2-L implicants
0, 4, 8, 12	xx00

prime implicants	minterms							
	m_0	m_4	m_8	m_{10}	m_{11}	m_{12}	m_{13}	m_{15}
p_1 = 10x0			✓	✓				
p_2 = 101x				✓	✓			
p_3 = 110x						✓	✓	
p_4 = 1x11					✓			✓
p_5 = 11x1							✓	✓
p_6 = xx00	✓	✓	✓			✓		

→ p_6 is essential

PIs	minterms			
	m_{10}	m_{11}	m_{13}	m_{15}
p_1 = 10x0	✓			
p_2 = 101x	✓	✓		
p_3 = 110x			✓	
p_4 = 1x11		✓		✓
p_5 = 11x1			✓	✓

→ remove p_1 & p_3 due to row dominance

PIs	minterms			
	m_{10}	m_{11}	m_{13}	m_{15}
$p_2 = 101X$	✓	✓		
$p_4 = 1X11$		✓		✓
$p_5 = 11X1$			✓	✓

↳ only p_2 covers m_{10} & only p_5 covers m_{13} ; these 2 PIs cover m_{11} & m_{13} so remove p_4

↳ final cover is $\{p_2, p_5, p_6\}$

The min-cost implementation is $f = p_2 + p_5 + p_6 = x_1\overline{x_2}x_3 + x_1x_2x_4 + \overline{x_3}\,\overline{x_4}$

week 4

PETRICK'S METHOD

- known as branch-&-bound method

  after applying row & column dominance methods for QM method, use Petrick's method to find necessary non-EPIs if PI cover table isn't empty

- process of Petrick's method:

  1) form logical that's true when all columns are covered
     - for each column, form sum of PIs that cover minterm
     - do logical AND on all sums to get a POS
  2) reduce POS to min SOP
     - $(x+y)(x+z) = x+yz$
     - $x + xy = x$
     - each product term in SOP rep solution (i.e. set of PIs that covers all minterms)
  3) determine min solutions
     - find product terms that contain min # of PIs
     - for each of above terms, count # of literals in each PI
  4) choose term composed of min total # of literals & write out corresponding sum of PIs

MULTIPLEXERS

- if there's 2 signals x_1 & x_2, & we want to choose only one signal in some case & other one in another case, use 2-input multiplexer
  - ↳ also need selector signal S

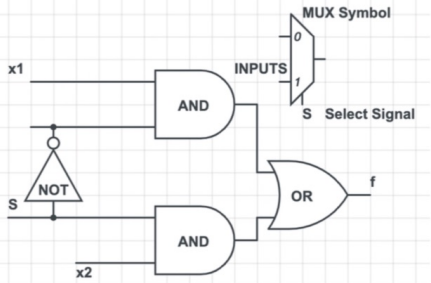
x_2	x_1	S	f	f
0	0	0	x_1	0
0	0	1	x_2	0
0	1	0	x_1	1
0	1	1	x_2	0
1	0	0	x_1	0
1	0	1	x_2	1
1	1	0	x_1	1
1	1	1	x_2	1

S	f
0	x_1
1	x_2

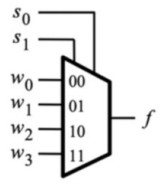
,

	x_2x_1	00	01	11	10
S					
0		0	1	1	0
1		0	0	1	1

$f = S'.x_1 + S.x_2$



- for a 4-to-1 mux, function is $f = s_1's_0'w_0 + s_1's_0w_1 + s_1s_0'w_2 + s_1s_0w_2$

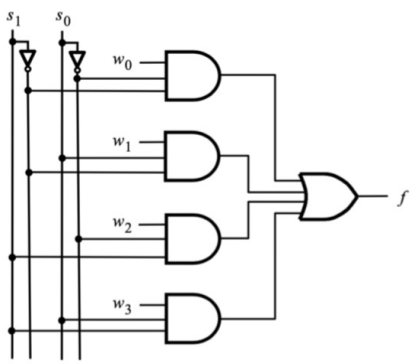


Symbol

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

Truth table

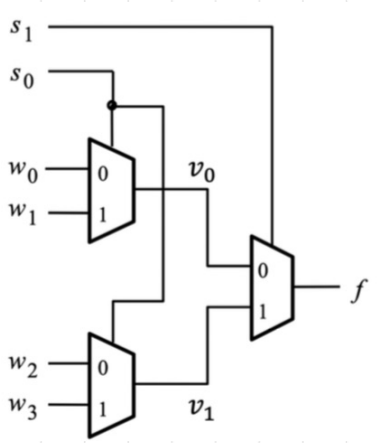
S_1 \ S_0	0	1
0	w_0	w_1
1	w_2	w_3



- to construct 4-to-1 mux from 2-to-1 muxes, we'll need 2 muxes for each half of the data input (to select within each group) & a 3rd one to select from the groups
  - ↳ algebraically, $f = s_1' v_0 + s_1 v_1$

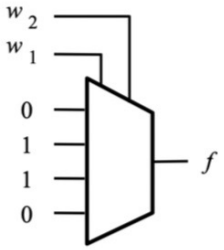
    $= s_1'(s_0' w_0 + s_0 w_1) + s_1(s_0' w_2 + s_0 w_3)$

    $= s_1' s_0' w_0 + s_1' s_0 w_1 + s_1 s_0' w_2 + s_1 s_0 w_3$
  - ↳



- any n-input logic function can be synthesized using a 2^n-to-1 mux by feeding values of function f into inputs of mux & using n-inputs as selection variables
  - ↳ e.g.

w_1	w_2	f
0	0	0
0	1	1
1	0	1
1	1	0

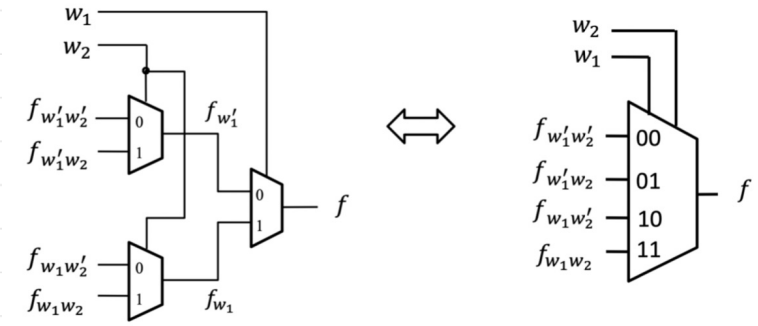


Straight-forward implementation using a 4-to-1 multiplexer

SHANNON'S EXPANSION

Shannon's expansion theorem: any Boolean function $f(w_1, \ldots, w_n)$ can be written in form $f(w_1, w_2, \ldots, w_n) = w_1' \cdot f(0, w_2, \ldots, w_n) + w_1 \cdot f(1, w_2, \ldots, w_n)$

- ↳ i.e. $f = w_1' \cdot f_{w_1'} + w_1 \cdot f_{w_1}$
- ↳ implemented using 2-to-1 mux
- ↳ can repeat application of Shannon's expansion (will need 4-to-1 mux)
  - $f_{w_1'} = w_2' \cdot f_{w_1' w_2'} + w_2 \cdot f_{w_1' w_2}$
  - $f_{w_1} = w_2' \cdot f_{w_1 w_2'} + w_2 \cdot f_{w_1 w_2}$



- can factor Boolean function wrt to any variable: $f = f(x_0, x_1, \ldots, x_n)$

  $f = x_0' f(0, x_1, \ldots, x_n) + x_0 f(1, x_1, \ldots, x_n)$
  - ↳ 2 resulting functions that don't include x_0 are cofactors
    - cofactors are like dividing truth table of f into halves

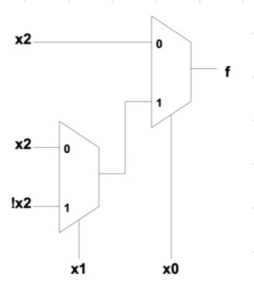
- e.g. implement $f = x_0'x_2 + x_0x_1'x_2 + x_0x_1x_2'$ using only 2-to-1 muxes

SOLUTION

$f = x_0'x_2 + x_0x_1'x_2 + x_0x_1x_2'$

$= x_0'(x_2) + x_0(x_1'x_2 + x_1x_2')$

$= x_0'(x_2) + x_0(x_1'(x_2) + x_1(x_2'))$



- e.g. implement $f(w_1, w_2, w_3) = m_3 + m_5 + m_6 + m_7$ w/ 4-to-1 mux

SOLUTION

$f = w_1'w_2w_3 + w_1w_2'w_3 + w_1w_2w_3' + w_1w_2w_3$

$f_{w_1'} = w_2w_3$

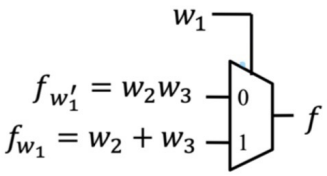
$f_{w_1} = w_2'w_3 + w_2w_3' + w_2w_3$

$= w_2'w_3 + w_2(w_3' + w_3)$

$= w_2'w_3 + w_2$

$= w_2 + w_3$ ← absorption law

2-to-1 mux:

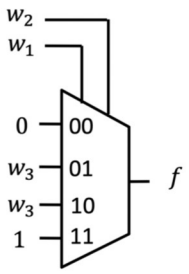


$f_{w_1'w_2'} = 0$ $\quad f_{w_1w_2'} = w_3$

$f_{w_1'w_2} = w_3$ $\quad f_{w_1w_2} = 1 + w_3$

$= 1$

4-to-1 mux:

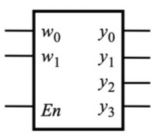


DECODERS

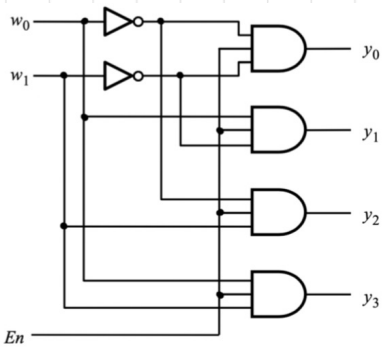
- binary decoder has n inputs & 2^n outputs
  - ↳ only one output is asserted at a time & each output corresponds to one valuation of inputs
  - ↳ has enable input En
    - if En = 0, none of outputs are asserted
- e.g. 2-to-4 decoder:

En	w_1	w_0	y_0	y_1	y_2	y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	x	x	0	0	0	0

Truth table



Symbol



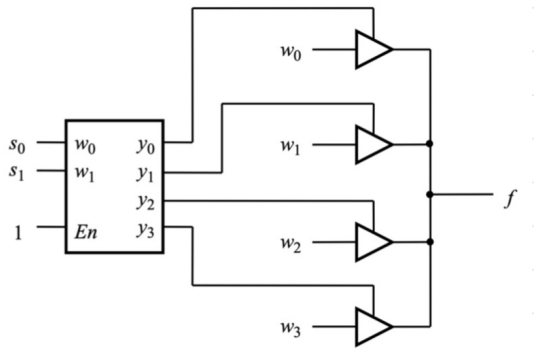
Logic circuit

Note that

$y_0 = En\, w_1'w_0'$,

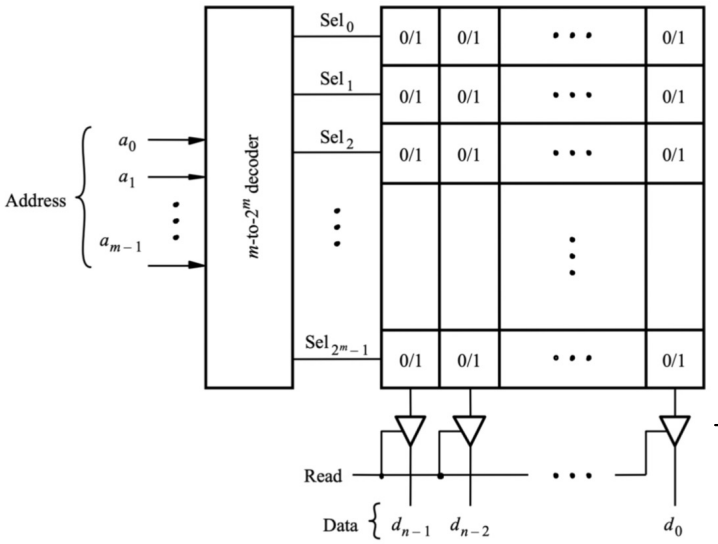
$y_1 = En\, w_1'w_0$, ...

- decoder w/ tri-state buffers can build 4-to-1 mux



↳ buffers act as a wire or wall for w_0, w_1, w_2, & w_3 based on output from decoder

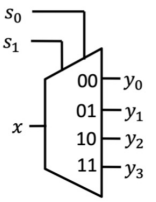
- e.g. use of buffer in $2^m \times n$ ROM block



→ decoder selects correct register from block

→ only bits w/ 1 are read into bus

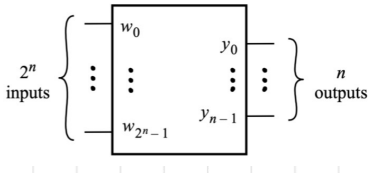
- in ROM, stored info can be read out but not changed
- when cell contents can be modified/programmed, memory is programmable ROM (PROM)
  - ↳ diff types of memories based on programming method
    - EPROM (erasable PROM)
    - EEPROM (electrically EPROM)
    - RW (read-write) memory
- de-multiplexer has only one data input, n select inputs, & 2^n outputs
  - ↳ opp operation of mux (i.e. places input value at one of outputs)
  - ↳ e.g. 1-to-4 de-mux:



s_1	s_0	y_0	y_1	y_2	y_3
0	0	x	0	0	0
0	1	0	x	0	0
1	0	0	0	x	0
1	1	0	0	0	x

Similar to decoder, note that
$y_0 = x\, s_1' s_0'$,
$y_1 = x\, s_1' s_0$, ...

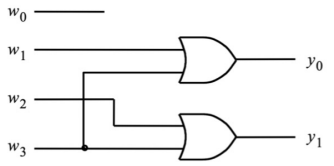
- binary encoder encodes info from 2^n inputs into n-bit code
  - ↳ one of its inputs should be 1 & outputs present binary # that identifies which input is 1
  - ↳ symbol of 2^n-to-n encoder:



↳ e.g. 4-to-2 encoder

w_3	w_2	w_1	w_0	y_1	y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

Truth table



Circuit

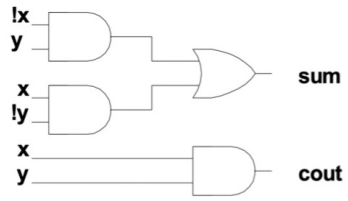
- $y_0 = w_1 + w_3$
- $y_1 = w_2 + w_3$

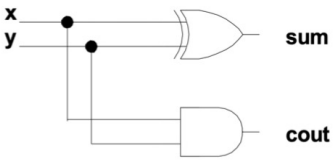
week 5

ADDERS

- basic definition of addition is take 2 bits & add them, producing sum & carry-out
  - ↳ circuit is called **binary half-adder**
  - ↳ can implement it in diff ways depending on availability of XOR gates

x	y	sum	cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



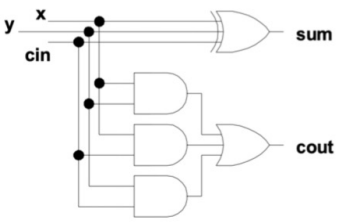


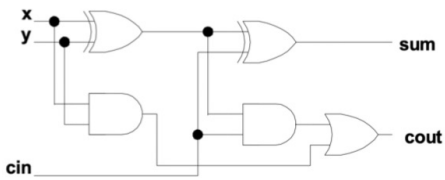
$$sum = x'y + y'x$$

$$cout = xy$$

- more commonly interested in adding n-bit #s so need to handle **carry in** signal
  - ↳ below circuit is **binary full-adder (FA)**
  - ↳ second implementation uses 2 half-adders

x	y	c_{in}	sum	c_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

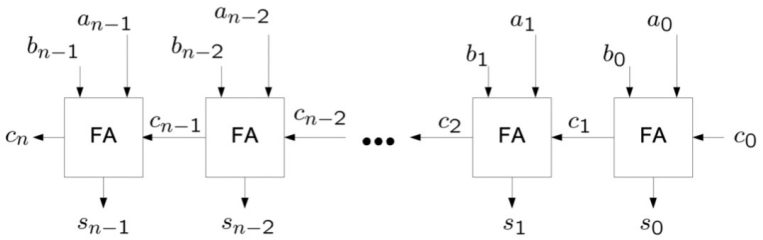




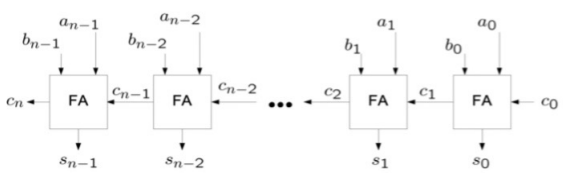
$$sum = x \oplus y \oplus c_{in}$$

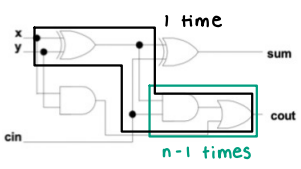
$$\begin{aligned} cout &= yc_{in} + xc_{in} + xy \\ &= xy + c_{in}(x + y) \\ &= xy + c_{in}(x \oplus y) \end{aligned}$$

- can build n-bit adders to add $A = (a_{n-1}\, a_{n-2} \ldots a_1 a_0)$ & $B = (b_{n-1}\, b_{n-2} \ldots b_1 b_0)$ by linking 1-bit full adders tgt
  - ↳ called **ripple adders**



- ripple adders can be slow for large # of bits
- assume logic gates have delay of 1 unit & full-adders are built from half-adders
  - ↳ delay of longest path tells us min amount of time we need to wait for output to be correct
  - ↳

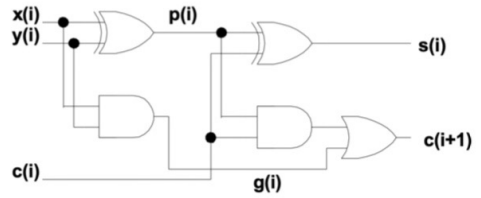




- change in LSB of A/B will cause change in MSB of c_{out}
- a_i / b_i values are ready at each FA but they have to wait for every c_{in} except 1st one
- delay = (n-1) x (1 AND + 1 OR) + (1 XOR + 1 AND + 1 OR)
  = 2n+1 gates of delay

· can calculate carry ins in separate circuit to inc speed

↳ identify 2 signals in i^{th} bit of adder: propagate p_i & generate g_i



↳ $c_{out} = C(i+1) = x_i y_i + c_i(x_i \oplus y_i) = g_i + c_i p_i$

- each i^{th} bit no longer depends on previous result
- $c_1 = g_0 + c_0 p_0$

  $c_2 = g_1 + c_1 p_1$

  $= g_1 + p_1 g_0 + p_1 p_0 c_0$

  $c_3 = g_2 + c_2 p_2$

  $= g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$

↳ $C_{(i+1)} = g_i + c_i p_i$ (1, 2, 3)

- adder will have 3 gates of delay to get c_n when using carry look-ahead (CLA)

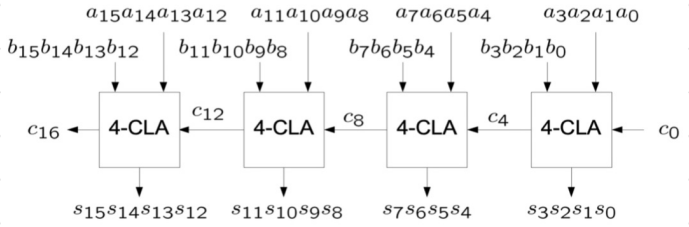
· CLAs are more expensive & higher # carries require more AND gates & AND/OR gates w/ larger # of inputs

↳ if we don't have large enough gate, might have to decompose into smaller gates

- e.g. 5-input AND gate decomposed into 4 2-input gates so gate delay is 3 instead of 1

· use combo of CLAs & ripple adders

↳ e.g. 16-bit adder w/ 4 4-bit CLAs

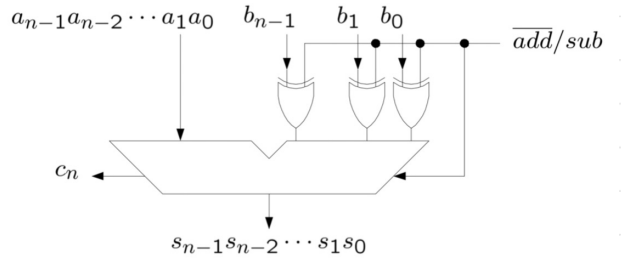


↳ performance = 3 + 2 + 2 + 2 = 9 gate delays

- 1st CLA has 3 gate delays but for others, $g_i = x_i y_i$ doesn't depend on c_i so can be performed immediately

· subtraction is done by taking 2s complement of subtrahend & adding

↳



↳ take any adder circuit & make it into combo adder/subtraction circuit w/ XOR gates & add/sub control signal
- when signal = 1, XOR gates perform 2s complement & subtraction happens (i.e $b_i = b_i'$)
- when signal = 0, it's normal adder (i.e. $b_i = b_i$)

· when 2 n-bit #s are added tgt & sum requires (n+1) bits, overflow has occurred
↳ true for signed/unsigned arithmetic
↳ for unsigned #s, overflow is detected if there's carry-out from MSB

COMPARATORS

· digital/magnitude comparator takes 2 #s as inputs & determine which one is larger/smaller or if they're equal

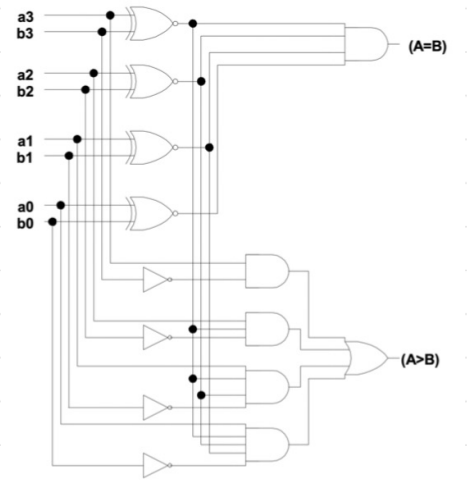
· to compare #s of n-bits for equality, compare bits in pairs from MSB to LSB
↳ if $A_i = B_i$, intro new signals e_i to detect this equality (i.e. XNOR)
- $e_i = a_i b_i + a_i' b_i'$

↳ #s are equal when all bits are equal (AND all e_i signals)
- $(A=B) = e_n e_{n-1} \cdots e_1 e_0$

· to compare #s for inequalities, let $i=n$ for MSB:
↳ if $a_i > b_i$, then $A > B$ & stop
↳ if $a_i < b_i$, then $B > A$ & stop
↳ if $a_i = b_i$, then let $i = i-1$ & repeat
- stop once $i=0$ & all bits are considered

· inequality comparator formulas:
↳ $(A>B) = a_n b_n' + (e_n)(a_{n-1} b_{n-1}') + \ldots + (e_n e_{n-1} \cdots e_1)(a_0 b_0')$
↳ $(A<B) = a_n' b_n + (e_n)(a_{n-1}' b_{n-1}) + \ldots + (e_n e_{n-1} \cdots e_1)(a_0' b_0)$

· magnitude comparator implementation:



↳ $(A<B)$ could be added using alr generated signals: $(A<B) = \overline{(A=B) + (A>B)}$

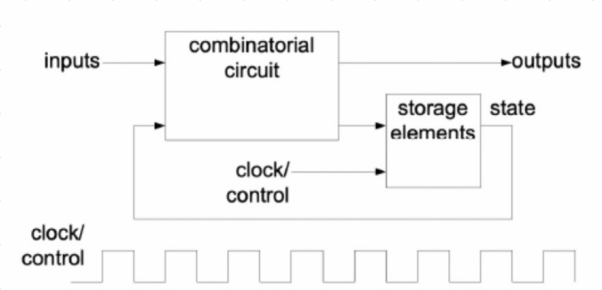
BUSSES

· bus: bundle of multiple wires running tgt to carry data from one place to another
↳ e.g. 8-bit bus symbol is —8/—

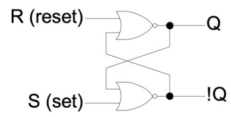
· if each wire in bus has multiple drivers (i.e. diff sources of info), can't have them driving wire at same time or electric short may occur
↳ to solve problem, make each wire driven by mux

SEQUENTIAL CIRCUITS

- circuits w/ simple logic gates are combinational circuits
- to create **sequential circuit**, include storage elements so that outputs are function of both current circuit inputs & **system state** (i.e. what happened in circuit before)
- 2 main types of sequential circuits:
  - **synchronous**: circuit behaviour is determined from knowledge of signal values at discrete instances in time
  - **asynchronous**: circuit behaviour is determined by signals at any instant in time & order in which input signals change
- **clock signals** are used to control behaviour of circuit at discrete instances in time by determining how & when memory elements can change their outputs



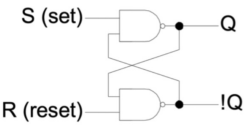
- 2 types of storage elements: latches & flip-flops
- **latches** are **level sensitive**, meaning they operate when control signal is either logic 0 or 1 & not at logic transitions
  - not necessarily useful for clocked sequential circuits but useful for async sequential circuits
- **NOR SR latch**: $Q(t+1) = Q(t)\bar{R} + S\bar{R}$



S	R	Q	$\bar{Q}$	
1	0	1	0	
0	0	1	0	(after $S = 1, R = 0$)
0	1	0	1	
0	0	0	1	(after $S = 0, R = 1$)
1	1	0	0	

Figure: NOR SR Latch

  - outputs are generally complements of each other
  - **set state**: $S=1$ & $R=0$ so output is $Q=1$ & $!Q=0$
  - **reset state**: $S=0$ & $R=1$ so output is $Q=0$ & $!Q=1$
  - **storage**: $S=0$ & $R=0$ so output holds it prev value
  - an exception is when $S=1$ & $R=1$, output is $Q='Q=0$ so circuit has undesirable/unknown behaviour
- **NAND SR latch**: $Q(t+1) = Q(t)R + \bar{S}R$

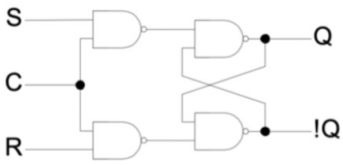


S	R	Q	$\bar{Q}$	
1	0	0	1	
1	1	0	1	(after $S = 1, R = 0$)
0	1	1	0	
1	1	1	0	(after $S = 0, R = 1$)
0	0	1	1	

↳ works similar to NOR implementation but input values are reversed for each of diff cases

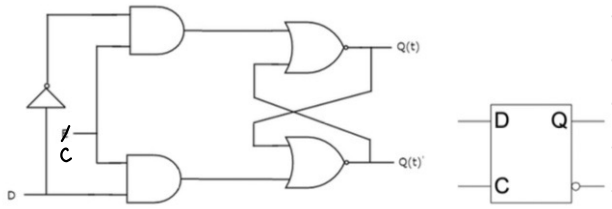
GATED LATCHES

- additional control input acts as enable signal
  - ↳ purpose is to control whether latch functions or not
  - ↳ e.g. gated S'R' latch:



C	S	R	n ext value of Q
0	X	X	(no change; hold)
1	0	0	(no change; hold)
1	0	1	0 (reset state)
1	1	0	1 (set state)
1	1	1	(undefined)

- latches can either have active high inputs (i.e. set when S=1 & reset when R=1) or active low inputs (set when S=0 & reset when R=0)
  - ↳ active low inputs have bubbles in front to rep NOT
- D-latch can be used to store one bit of info & we don't have to worry abt restricted cases like w/SR & S'R' latch



↳

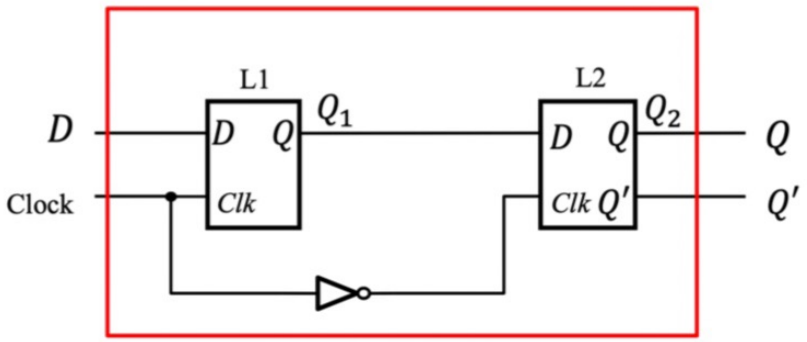
D	G	Q	$\bar{Q}$	
1	1	1	0	← set
0	1	0	1	← reset
0/1	0	latch		← hold

- latches don't allow for precise control b/c they're level sensitive
  - ↳ create interval in time over which state/output of memory element can change rather than instant in time

week 6

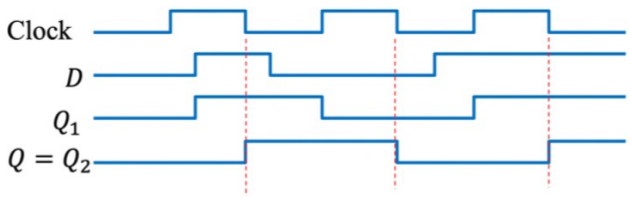
FLIP FLOPS

- flip flops are **edge triggered** devices, which means their output changes depending on values of FF inputs & when a **clock input** changes from 0→1 or 1→0
  - ↳ **+ve edge triggered**: outputs change when clock transitions from 0→1 (i.e. **rising edge**)
  - ↳ **-ve edge triggered**: outputs change when clock transitions from 1→0 (i.e. **falling edge**)
- master-slave FF built from 2 latches:



  - ↳ 1st latch is master & 2nd is slave
  - ↳ above is example of -ve edge triggered DFF:
    - when clock = 1, L1 will follow input D so $Q_1 = D$ but L2 is on hold so Q_2 doesn't change
    - when clock 1→0, L1 will hold & Q_1 will be D-value just before clock changed; L2 will set Q_2 to Q_1
  - ↳ net result is D-value just prior to clock 1→0 gets transferred to final output
  - ↳ Q_2 is what circuit stores & Q_1 is internal value
- characteristic equation is $Q(t+1) = D$
- 2 types of DFF:

- DFF timing diagram:



- **toggle flip flop (TFF)** flips/toggles output if input T = 1 & otherwise, output doesn't change
  - ↳

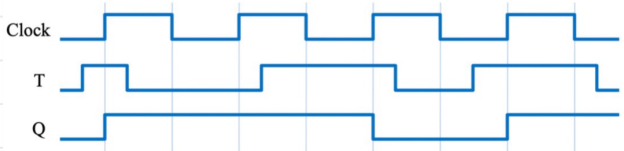
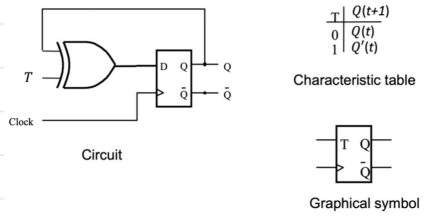


Figure: T-FF Timing

↳



↳ characteristic equation is $Q(t+1) = Q(t)\overline{T} + \overline{Q(t)}T = Q(t) \oplus T$

· JK flip flops are generalizations of DFFs & TFFs

↳ becomes TFF if $J = K$

↳ becomes DFF if $J = K'$

↳

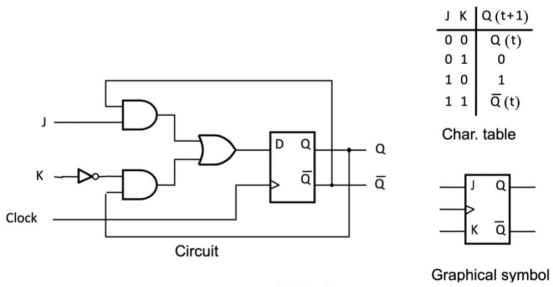


Figure: JK FF

↳ characteristic equation is $Q(t+1) = J\overline{Q(t)} + \overline{K}Q(t)$

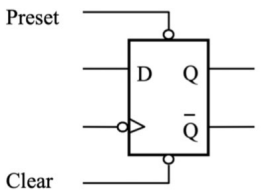
· additional input pins on FFs can include clear & preset

↳ clear means $Q = 0$ & preset means $Q = 1$, regardless of FF inputs & behaviour

- async operations

↳ to make sync clear, put AND gate w/inputs of data & clear' before D

↳



- both are active low

· enable signal prevent clock from causing change in Q

· takes time for gates to change their output values (i.e. propagation delays)

· 3 important timing parameters:

↳ setup time (t_{su}): amount of time data inputs need to be held stable prior to arrival of active clock edge

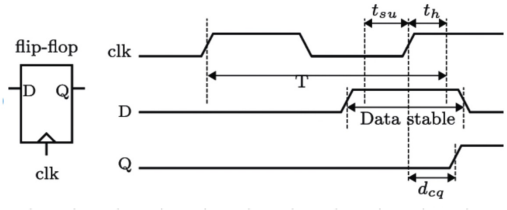
↳ hold time (t_h): amount of time data inputs need to be held stable after arrival of active clock edge

↳ clock-to-output time (t_{cQ}): amount of time it takes for output to become stable at its new value after arrival of active clock edge

- output of FF is unreliable before then

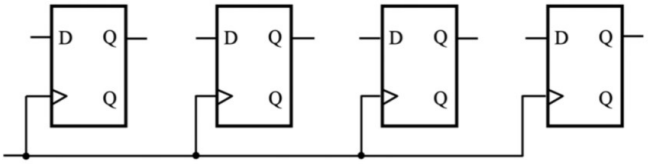
· if we fail to meet any of timing parameters, there's timing violation

·

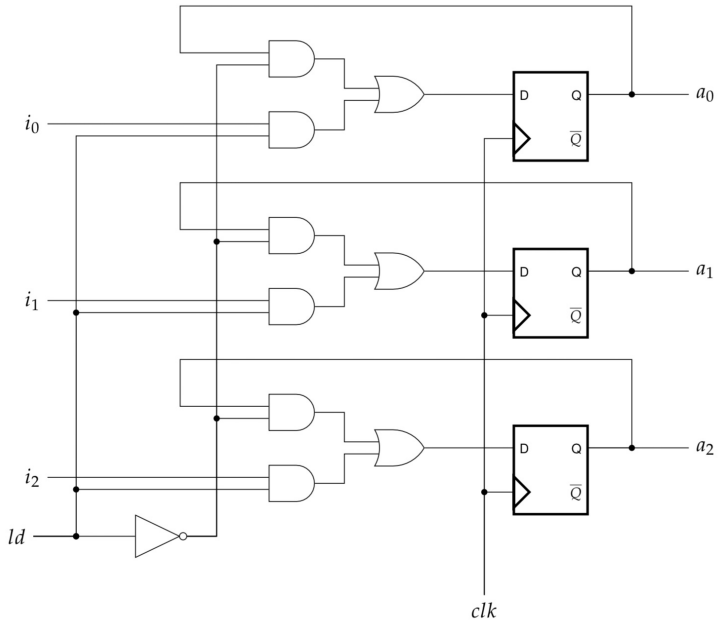


REGISTERS

- registers are **digital storage**
  - ↳ n-bit registers are made of n FFs grouped tgt to perform some task
  - ↳ FFs all use same clock
  - ↳ e.g. simple 4-bit register that loads new data on every active clock edge



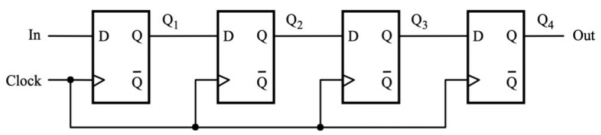
- e.g. register w/ ability to load new data & also hold current data at FF outputs (i.e. **parallel load & hold**)



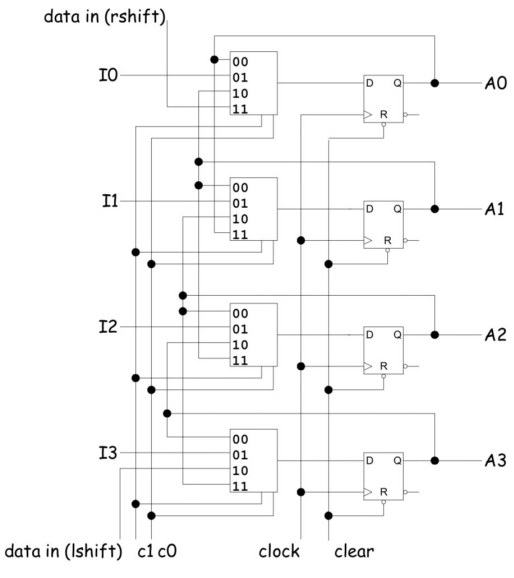
  - ↳ when ld = 1, register loads new data
  - ↳ when ld = 0, register holds current value

shift registers accept input & shift it one bit over on every active clock edge

  - ↳ e.g.



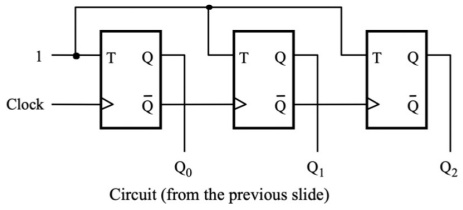
- **universal shift register** can allow us to choose if we want to clear, load, shift left, or shift right the register
  - ↳ use muxes in front of FF inputs to direct correct info in



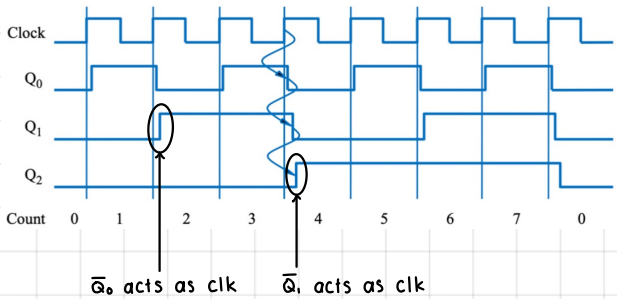
COUNTERS

- counters increment/decrement # in its storage
  - ↳ increment is **up-counter**
  - ↳ decrement is **down-counter**
  - ↳ 2 types: **sychronous** (bits of #s are updated at same time) & **asynchronous**
- async counters have FFs that don't change at same time that's synced w/ one master clock
  - ↳ a_0 (LSB) always toggles
  - ↳ a_i toggles when a_{i-1} makes transition from 1→0

↳ e.g 3-bit up-counter using TFFs



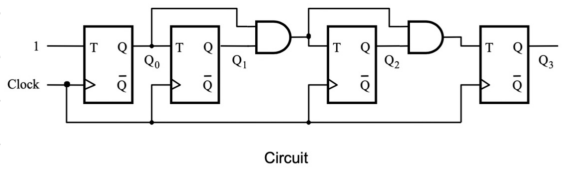
Circuit (from the previous slide)



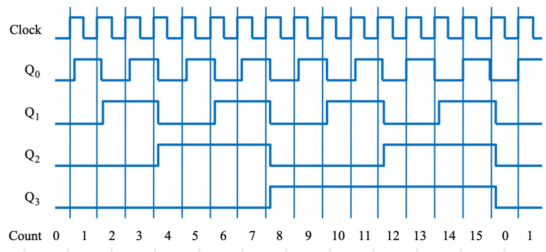
↳ TFFs are all +ve-edge-triggered

- async counters are slow w/ large # of bits since clocks form a chain (serial)

- **sync counters** are characterized by fact that single clock signal controls every FF
  - ↳ a_0 (LSB) always toggles
  - ↳ a_i should toggle when bits a_{i-1} down to a_0 are all 1
  - ↳ e.g. 4-bit sync counter



Circuit

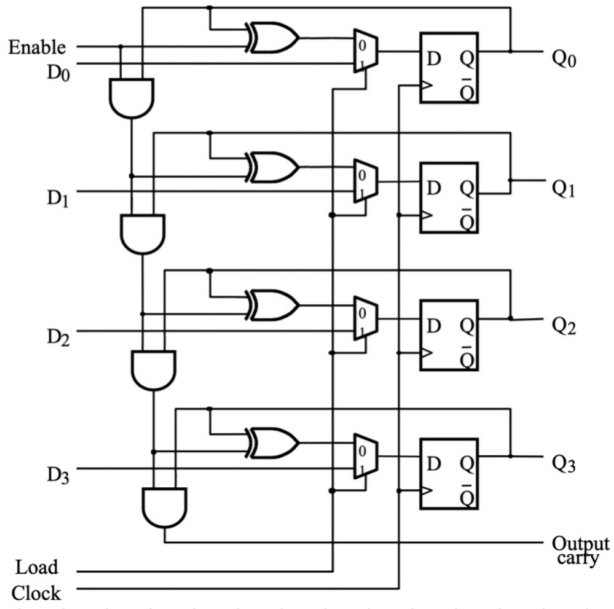


- to make into down counter, make one of inputs into AND gate from $\bar{Q}$ instead of Q b/c a_i should toggle when bits a_{i-1} down to a_0 are 0

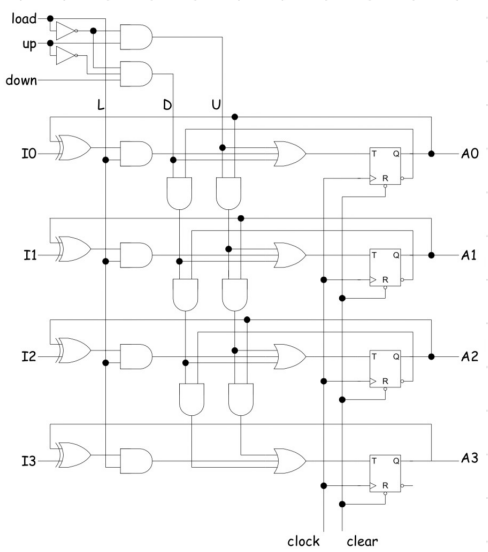
- can also build DFF-based counters by putting enable & Q signals through XOR
  - ↳

en	Q	D	
0	0	0	hold
0	1	1	
1	0	1	toggle
1	1	0	

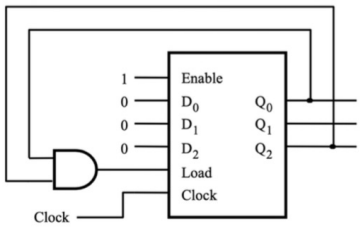
- e.g. counter w/ parallel-load capability



- e.g. binary up/down-counter w/ parallel load
  - ↳ if certain control signals & operations have priority, it should disable other operations
    - e.g if load is enabled, up/down are turned off

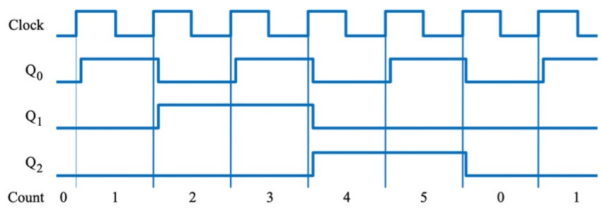


- a **modulo-n counter** starts from a specific value & only counts n bits before starting over
  - ↳ can construct from binary up-counter, add logic to detect when max count is reached & parallel load to restart count sequence
  - ↳ e.g. mod-6 counter w/sync reset



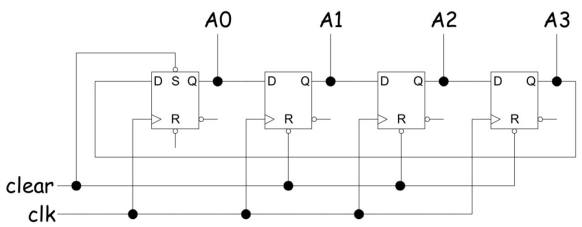
Block diagram

→ loads register when 101 (which is 5) is reached



ring counter: only 1 output bit is high/logic 1 at any time

↳ e.g. 4-bit ring counter:



- sequence produced: 1000, 0100, 0010, 0001, 1000, ...
- designed w/ DFF based shift register but to start count operation, 1st DFF is preset to 1 while others are cleared to 0 & output of last DFF is connected to input of 1st

· Johnson counter is modified ring counter where output of last DFF is inverted before being fed back as input to 1st

↳ output becomes:

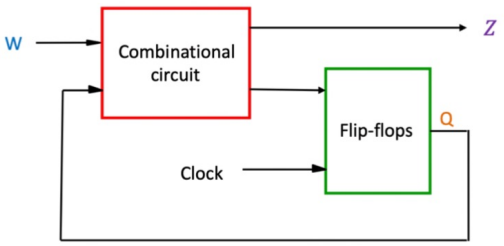
Clock Pulse	Flip-Flop Outputs			
	A_3	A_2	A_1	A_0
1	0	0	0	0
2	0	0	0	1
3	0	0	1	1
4	0	1	1	1
5	1	1	1	1
6	1	1	1	0
7	1	1	0	0
8	1	0	0	0
	repeat			

SEQUENTIAL CIRCUITS

· synchronous sequential circuits are characterized by combinational logic circuits, FFs, & clocks

↳ aka finite state machines (FSM)

↳ general form:

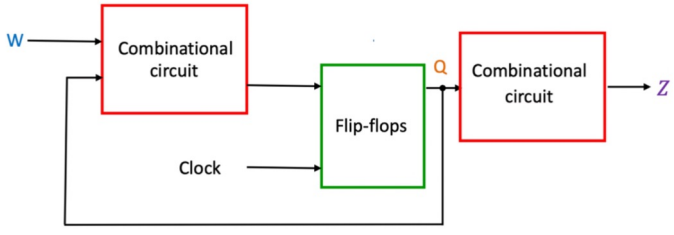


- W: circuit input
- Q: FF output / current state
- Z: circuit output

· a Moore machine is special case where Z depends only on current state

↳ output changes in sync w/clock

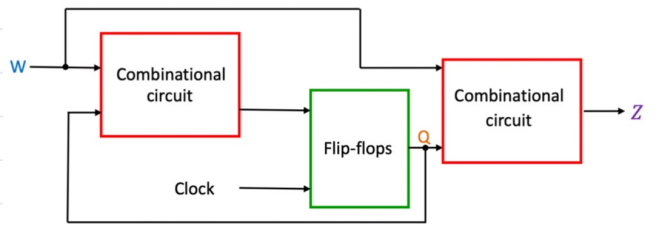
↳



· a Mealy machine is when Z depends on both current state & input

↳ output changes asynchronously w/clock

ب

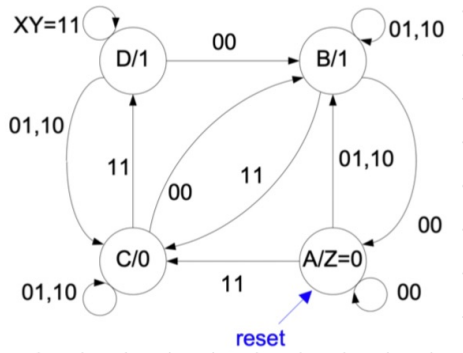

W
Combinational circuit
Flip-flops
Q
Clock
Combinational circuit
Z

week 7

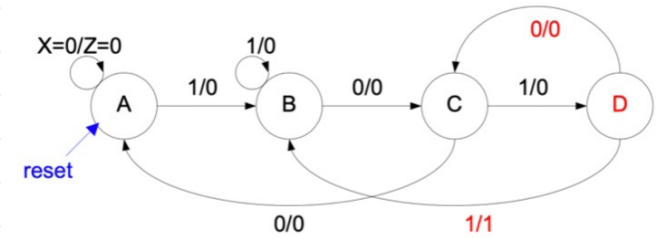
STATE DIAGRAMS

- state diagram is pictorial rep of how sync sequential circuit should operate to perform task
  - ↳ illustrates several things abt circuit:
    - possible states of circuit
    - possible transitions btwn states (which depend on inputs)
    - circuit output values (which depend on state, inputs, or both)
- state bubbles rep possible states of sequential circuit
- directed edges rep possible transitions btwn states
- circuit output values are labelled inside of bubbles or on arcs depending on type of machine
  - ↳ for Moore machine, outputs are labelled inside of bubble b/c outputs are a function of only current state (i.e. memory values)
  - ↳ for Mealy machine, outputs are labelled along edges b/c outputs are a function of both current state & circuit inputs
- state diagram might also show initial/reset state, which is desired start state for circuit

- e.g. state diagram for Moore machine

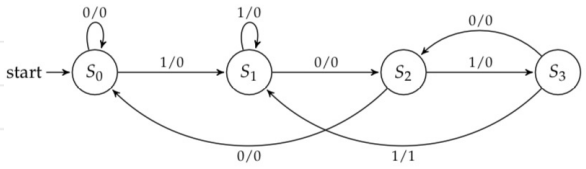


  - ↳ circuit has 2 inputs (X & Y) & one output (Z)
  - ↳ 4 states: A, B, C, D
  - ↳ outputs are labelled inside state bubbles
- e.g. state diagram for Mealy machine



  - ↳ circuit has one input (X) & one output (Z)
  - ↳ 4 states: A, B, C, D
  - ↳ outputs are labelled on edges
- any sequential circuit can be described by Mealy/Moore state diagram
  - ↳ only diff btwn 2 machines is where output values are labelled
- purpose of reset is to show where system starts when we power on/restart circuit
  - ↳ since in acc circuit, memory is implemented via FFs, force outputs to certain values in order to reset it

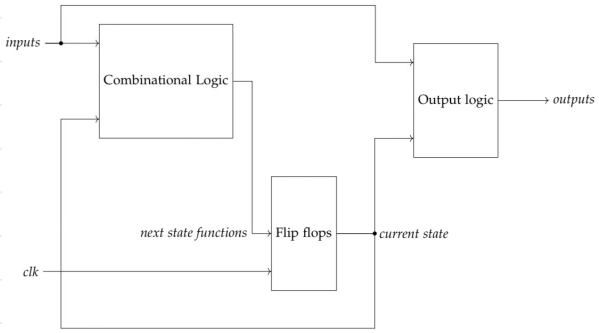
- state tables describe behaviour of sequential circuit in tabular form
  - ↳ aka transition tables
  - ↳ shows same info as state diagram
  - ↳ e.g.



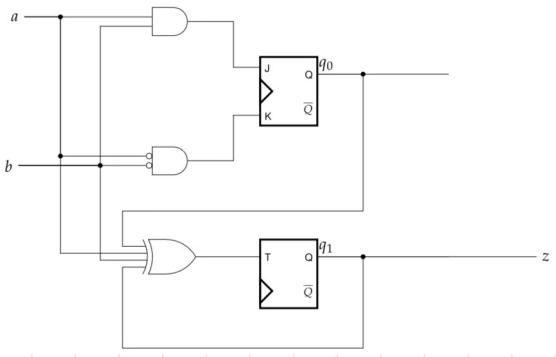
Current State	Next State		Output Z	
	$X=0$	$X=1$	$X=0$	$X=1$
S_0	S_0	S_1	0	0
S_1	S_2	S_1	1	1
S_2	S_0	S_3	0	0
S_3	S_2	S_1	0	1

SEQUENTIAL CIRCUIT ANALYSIS

- structure of synchronous sequential circuit:



  - ↳ current state is rep as binary pattern at FF outputs
  - ↳ next state functions are combinational logic gates that determine next state upon arrival of active clock edge based on circuit inputs & current state
- given circuit, synchronous circuit analysis involves figuring out what it's doing
  - ↳ figure out associated state table / diagram
- basic procedure:
  - ↳ identify FFs used to hold current state info
  - ↳ identify outputs of circuit
  - ↳ write down logic equations for circuit outputs & FF inputs (i.e. next state equations)
    - FF input equations can also be called excitation equations
  - ↳ use logic equations to derive state table which describes next state & circuit outputs
  - ↳ obtain state diagram from state table
  - ↳ abstract away anything particular to implementation
    - e.g. how states have been encoded
- e.g. obtain state table & diagram for circuit below



SOLUTION

↳ 2 inputs (a & b) & one output (z)

↳ 2 FFs so circuit is implementing state diagram w/ max 4 states

- $q_1 q_0$ = 00, 01, 10, 11

Output equation: $z = q_1$

Next state functions: $j_0 = ab$

$k_0 = \bar{a}\bar{b}$

$t_1 = a \oplus b \oplus q_0 \oplus q_1$

Table for FF input values for each possible state & setting of circuit inputs:

Current State	$j_0 k_0$				t_1			
$q_1 q_0$	ab=00	01	10	11	ab=00	01	10	11
00	01	00	00	10	0	1	1	0
01	01	00	00	10	1	0	0	1
10	01	00	00	10	1	0	0	1
11	01	00	00	10	0	1	1	0

Table of next FF output values upon arrival of active clock edge:

Current State	Next State q_1				Next State q_0			
$q_1 q_0$	ab=00	01	10	11	ab=00	01	10	11
00	0	1	1	0	0	0	0	1
01	1	0	0	1	0	1	1	1
10	0	1	1	0	0	0	0	1
11	1	0	0	1	0	1	1	1

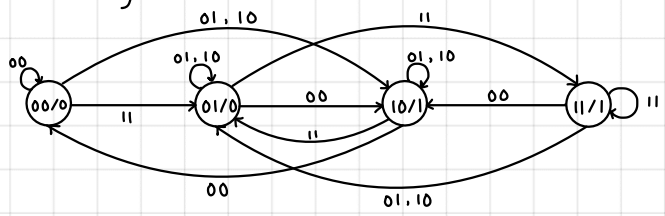
Final state table:

Current State	Next State $q_1 q_0$				Output z			
$q_1 q_0$	ab=00	01	10	11	ab=00	01	10	11
00	00	10	10	01	0	0	0	0
01	10	01	01	11	0	0	0	0
10	00	10	10	01	1	1	1	1
11	10	01	01	11	1	1	1	1

↳ output z is only function of current state

↳ initial state is not identified b/c circuit didn't have reset signal connected to FFs

State diagram:



SEQUENTIAL CIRCUIT DESIGN

circuit design is figuring out from verbal problem description how to implement circuit that accomplishes what's described

basic procedure:

↳ understand verbal description

↳ create state diagram/table

- states might only have symbolic names

↳ perform state reduction so circuit might have less FFs

- find equivalences btwn states & get smaller diagram/table

↳ perform state assignment

- i.e. give each state a binary # to rep it

↳ select FF types to store state info (current state)

- # of FFs depends on how many bits required to rep states

↳ derive output logic & next state (i.e. FF input) equations

↳ draw resulting circuit

e.g. A vending machine dispenses a package of gum which costs 15 cents. The machine has a single slot that accepts nickels and dimes only and a sensor indicates the type of coin deposited. Make a controller that sends a signal to a chute release once enough change has been deposited. Note that the machine does not give change or credits.

SOLUTION

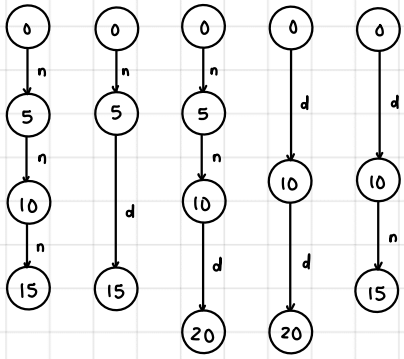
↳ 2 inputs, n & d, to rep coin deposited

- n & d can nvr be 1 at the same time

↳ clock & reset input

↳ one output, release, when enough change is deposited

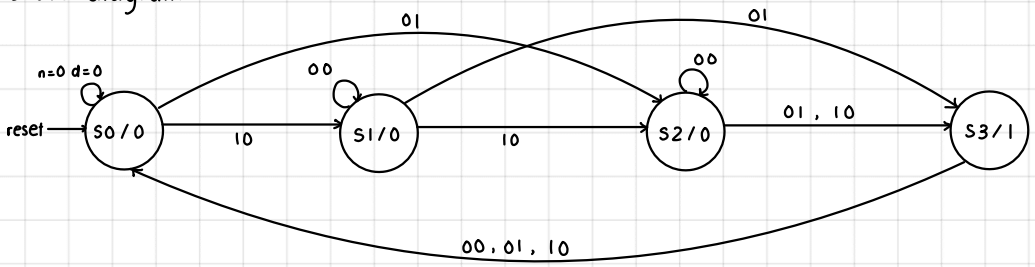
All sequences in which ≥ 15¢ is deposited:



↳ define state to rep amount of money deposited so far

- S0 = 0¢
- S1 = 5¢
- S2 = 10¢
- S3 ≥ 15¢

State diagram:



Encode state names as bit strings:

↳ S0 = 00

↳ S1 = 01

↳ S2 = 10

↳ S3 = 11

} since there's 2 bits, need 2 FFs

State table:

Current State (Q_1Q_0)	Input (nd)	Next State (Q_1Q_0)	Output (release)
00	00	00	0
00	01	10	0
00	10	01	0

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

Select DFF so we know output Q(t+1) equals input D(t) when active clock edge arrives.

↳ derive next state equations:

$D_0 = Q_0(next)$:

Q_1Q_0 / nd	00	01	11	10
00	0	1	0	0
01	0	1	0	1
11	X	X	X	X
10	1	0	0	1

$D_0 = \bar{Q}_1 Q_0 \bar{n} + Q_1 \bar{Q}_0 d + \bar{Q}_0 n$

$D_1 = Q_1(next)$:

Q_1Q_0 / nd	00	01	11	10
00	0	0	0	1
01	1	1	0	1
11	X	X	X	X
10	0	1	0	1

$D_1 = \bar{Q}_1 d + \bar{Q}_1 Q_0 n + Q_1 \bar{Q}_0$

NOTE

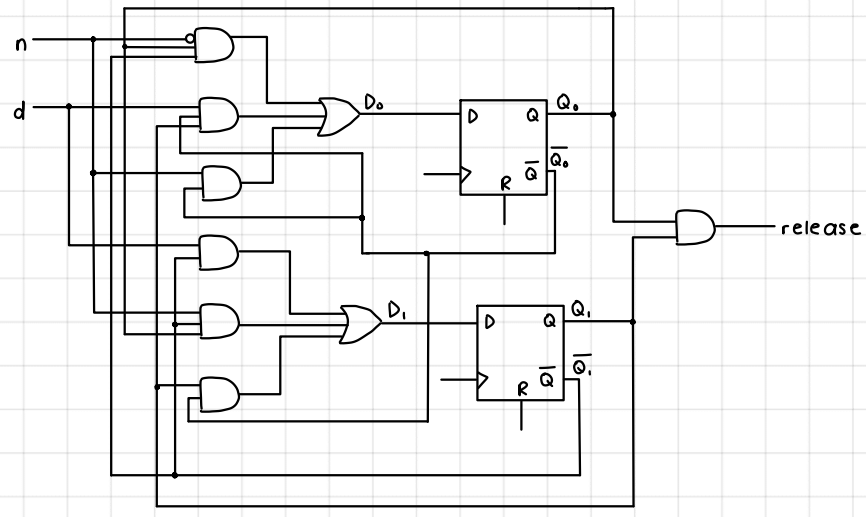
- DFF input equations should be equal to next state info in state table

↳ derive output equation:

Q_1Q_0 / nd	00	01	11	10
00	0	0	1	0
01	0	0	1	0
11	X	X	X	X
10	0	0	1	0

release = $Q_1 Q_0$

Resulting circuit / schematic:



- when designing sync circuits, using DFF is easiest b/c input equations are next state equations
  ↳ however, other FFs may result in much simpler logic equations

- when not using DFF, logic applied to FF inputs are not next state equations but will generate correct next state in conjunction w/ FF behaviour
- excitation tables are used to rep how FF outputs change depending on FF inputs

↳ DFF:

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

DFF Excitation Table

↳ TFF:

T	$Q(t)$	$Q(t+1)$
0	0	0
1	0	1
0	1	1
1	1	0

TFF Excitation Table

↳ JKFF:

J	K	$Q(t)$	$Q(t+1)$
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	1
0	0	1	1
1	0	1	1
0	1	1	0
1	1	1	0

JKFF Excitation Table

J	K	$Q(t)$	$Q(t+1)$
0	X	0	0
1	X	0	1
X	0	1	1
X	1	1	0

JKFF Excitation Table

- usually, JKFFs result in simpler implementation b/c they have a lot of Xs in their K-maps

week 9

STATE ASSIGNMENT

- **state assignment**: assign binary values to rep symbolic states in state diagram / table
  - ↳ sometimes beneficial to use more than min # of FFs b/c it results in simpler output & next state equations
- one way for state assignment is try for **min # of FFs**
  - ↳ require at least $\lceil \log n \rceil$ bits / FFs for n states
  - ↳ diff assignment of binary patterns to states will change FF input eqs & output eqs but difficult to predict which assignment will reduce final complexity
  - ↳ only benefit is that this will always require min # of FFs

output encoding is when we encode states st outputs are equal to current state

- ↳ benefit is that there's no output logic required after getting FF outputs
- ↳ e.g. counters
- ↳ may have to use more than min # of FFs
- ↳ ensure that every state has diff binary pattern

- e.g. use output encoding to implement circuit for state table

Current state	Output			
	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

→

Current state	Output				Extra bits
	A	B	C	D	E
S_0	0	0	0	0	0
S_1	0	0	0	0	1
S_2	0	1	0	0	X
S_3	1	0	0	0	X
S_4	1	0	0	1	X
S_5	1	0	1	0	X
S_6	1	0	1	1	X

  - ↳ 2 cases in assigning bits & deciding on # of FFs:
    - outputs for each state are distinct so use binary value of each output for FF outputs
    - outputs for some states are identical so add additional bits to distinguish states
  - ↳ in above example, need 5 FFs (2 more than min)
    - outputs for circuit are taken directly from first 4 FFs

can't predict complexity of next state (FF input) functions but there's no complexity of output logic

- need to be careful abt potential unused states

- **one-hot encoding** uses one DFF per state & only one output of any FF is high at any given time
  - ↳ when FF output is 1, indicates what state we're in
  - ↳ generally reduces logic for both next state & output equations but requires more FFs
- e.g. use one-hot encoding & derive next state & output equations

Current State	Next State		Output	
	$a=0$	$a=1$	$a=0$	$a=1$
$q_2q_1q_0$	$q_2q_1q_0$	$q_2q_1q_0$	z	z
001	100	001	1	1
010	001	010	0	0
100	010	100	1	0

TIP

- make sure no info is missing from trying to reduce clutter in diagram
  - ↳ e.g. self-loops on states might be left out but it's implied

SOLUTION

↳ instead of K-maps, can derive equations by observation

$d_0 = \bar{a}q_1 + aq_0$

$d_1 = \bar{a}q_2 + aq_1$

$d_2 = \bar{a}q_0 + aq_2$

$z = q_0 + \bar{a}q_2$

STATE REDUCTION

- when generating states from verbal description, might get redundant states

pair of states are **equivalent** if for every possible circuit combo:

↳ states give exactly same circuit output

↳ states transition to same next state / equivalent state

- accomplished using implication chart & merger diagram or partitioning

to get **implication chart**:

1) decide which states are definitely not equivalent due to differing output values
2) decide which states are definitely equivalent due to having same outputs & always same next states
3) decide which states are equivalent only when other states are equivalent
   - conditions for equivalence
4) check conditions & if can't be met, mark states subject to conditions as not equivalent

- after getting implication chart, draw **merger diagram**

↳ to decide what states to merge tgt, look for **cliques** of states & check conditions to ensure they're valid

- clique means every state in it has edge connecting it to every other state in clique

- e.g. reduce state table:

Current State	Next State		Output (z)	
	$a=0$	$a=1$	$a=0$	$a=1$
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

SOLUTION

Implication chart:

s_1	X			
s_2	✓	X		
s_3	X	(s_0, s_1) X (s_3, s_4)	X	
s_4	X	(s_0, s_2) ✓	X	(s_1, s_2) X (s_1, s_3)
	s_0	s_1	s_2	s_3

↳ boxes w/ only 'X' in them means they have diff outputs

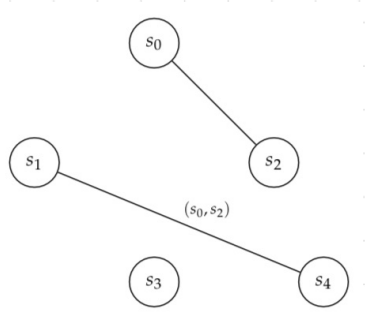
↳ boxes w/ only '✓' in them means they have same outputs & next states

↳ boxes w/ conditions means they're only equivalent if listed pairs of states are equivalent

- for s_1 & s_3, (s_0, s_1) are not equivalent so (s_1, s_3) are not equivalent
- for s_1 & s_4, (s_0, s_2) are equivalent so (s_1, s_4) are also equivalent

• for s_3 & s_4, (s_1, s_2) are not equivalent so (s_3, s_4) are not equivalent

Merger diagram:



↳ make sure every state is included

↳ merge s_0 & s_2 into single state denoted by s_0

↳ merge s_1 & s_4 into single state denoted by s_1 (since equivalence of s_0 & s_2 is satisfied)

Reduced state table:

Current State	Next State		Output (z)	
	$a=0$	$a=1$	$a=0$	$a=1$
s_0	s_3	s_0	1	1
s_1	s_0	s_1	0	0
s_3	s_1	s_3	0	0

· procedure for state reduction by partitioning:

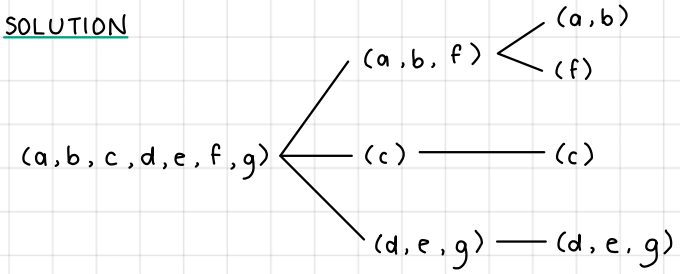
1) group states according to circuit outputs produced
2) for each group, if for any input pattern, diff states in group result in transition to diff other groups, those states aren't equivalent
   ↳ separate group into 2 smaller groups
3) continue partitioning groups until all states in any group transition to same other group for any input pattern

· e g. reduce state table below

State Table to be Reduced.

resent State	Next State		Output	
	x=0	x=1	x=0	x=1
a	d	a	0	0
b	e	a	0	0
c	g	f	0	1
d	a	d	1	0
e	a	d	1	0
f	c	b	0	0
g	a	e	1	0

SOLUTION

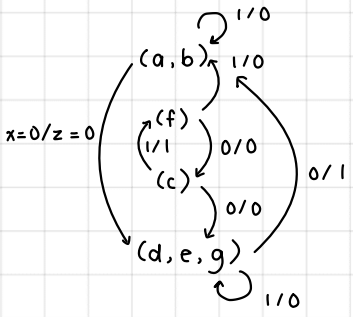


↳ x = 0 a → d, b → e, but f → c

↳ x = 0: d → a, e → a, g → a

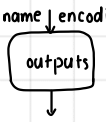
↳ x = 1: d → d, e → d, g → e

Reduced state diagram:



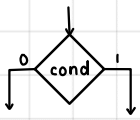
ALGORITHMIC STATE MACHINES

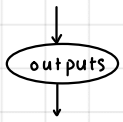
· algorithmic state machine (ASM) is alternative to state diagram

- 3 types of boxes in ASM:
  - ↳ state boxes
  - ↳ decision boxes
  - ↳ conditional output boxes
- state box has 1 entry & 1 exit point
  - ↳ 

  - ↳ state name/binary encoding listed above
  - ↳ any outputs that are 1 & depend only on state labelled inside

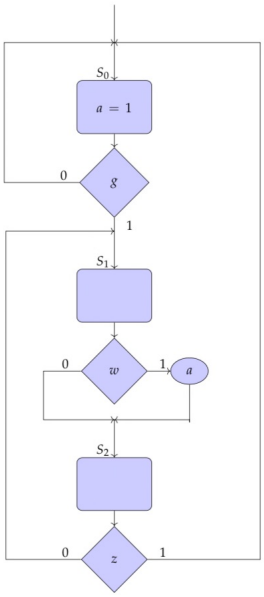
NOTE
- outputs are only shown in an ASM when they're 1

- decision box has 1 entry & 2 exits
  - ↳ 

  - ↳ inside labelled w/ logic expression
- conditional output box has 1 entry & 1 exit
  - ↳ 

  - ↳ specifies output that occurs when transition takes place
  - ↳ required for Mealy circuits
- in ASM: # of states = # of state boxes, next state transitions based on state boxes & decision boxes, & circuit outputs based on state boxes & conditional output boxes

e.g.

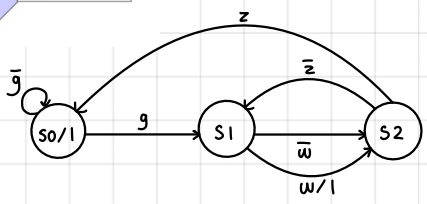


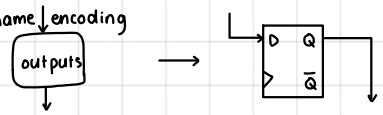
State table:

Current state	Input g	Input w	Input z	Next state	Output
S_0	0	X	X	S_0	1
S_0	1	X	X	S_1	1
S_1	X	0	X	S_2	0
S_1	X	1	X	S_2	1
S_2	X	X	0	S_1	0
S_2	X	X	1	S_0	0

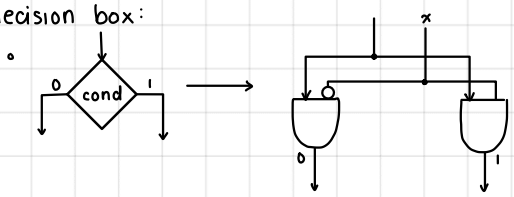
↳ written in diff form b/c there's large # of inputs & don't care situations in decision boxes

State diagram:

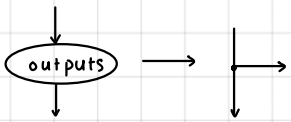


- every part of an ASM has corresponding circuit element so it's possible to draw circuit directly from ASM
- hardware equivalents:
  - ↳ state box is DFF.
    - 

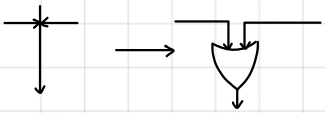
↳ decision box:



↳ conditional output box:

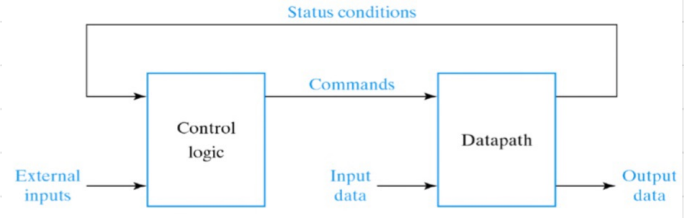


↳ diff paths connect tgt:



REGISTER TRANSFER LEVEL

- digital system at register transfer level specified by 3 components
  - ↳ set of registers in system
  - ↳ operations to be performed on registers
  - ↳ control applied to registers & sequence of operations
- rep of **register transfer flow**:
  - ↳ transfer from 1 register to another is $R2 \leftarrow R1$
  - ↳ conditional statement is if $(T1 = 1)$ then $R2 \leftarrow R1$
  - ↳ clock signal isn't generally included b/c sequential behaviour is implied
  - ↳ multiple choices are possible: if $(T1 = 1)$ then $(R2 \leftarrow R1,\ R2 \leftarrow R2)$
  - ↳ addition is $R1 \leftarrow R1 + R2$
  - ↳ increment is $R3 \leftarrow R3 + 1$
  - ↳ shift right is $R4 \leftarrow R4$
  - ↳ clear is $R5 \leftarrow 0$
- digital system split into 2 components:
  - ↳ **datapath**: manipulates data according to system reqs
  - ↳ **control unit / logic**: generates signals for sequencing operations in data processor
  - ↳



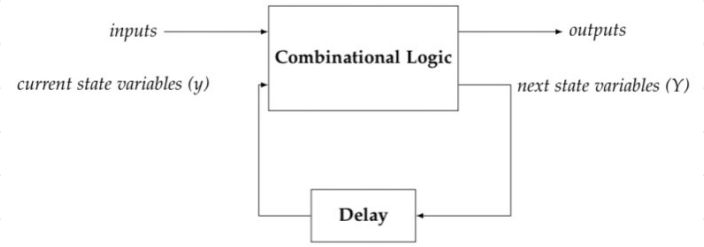
week 10

ASYNCHRONOUS CIRCUIT ANALYSIS

async circuits exhibits concept of state/memory like a clocked sequential circuit but uses no clocks & FFs

- ↳ concept of memory obtained thru latches, combinational feedback paths (loops), & circuit delays

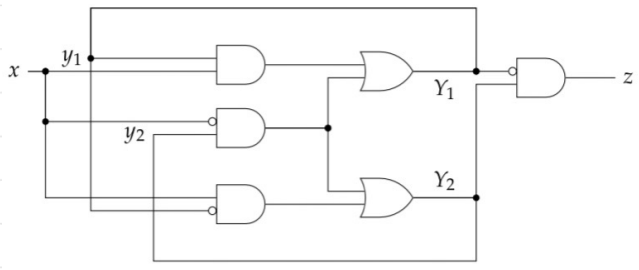
conceptual block diagram for async circuit:



- ↳ next state variables Y are aka excitation variables
- ↳ current state variables y are aka secondary variables
- ↳ Y & y are values at opp ends of same wire which loops back in circuit
  - separated by delay so they're thought of as separate variables if conceptually break feedback loops
  - equal to each other after delay
- ↳ delay elements are hypothetical & typically result of gate delays

- circuit is stable if it's reached steady state in which excitation & secondary variables are equal & unchanging
- circuit is operating in fundamental mode if we force restrictions on inputs:
  - ↳ only 1 input can change at a time
  - ↳ input changes only after circuit is stable

- to analyze, derive transition tables (show binary state assignments) & flow tables (purely symbolic)
  - ↳ can also use state diagrams
- procedure for async analysis:
  1) identify feedback paths
     - hypothetically break them & turn circuit into combinational one
  2) label excitation & secondary variables
  3) derive equations for excitation & output variables in terms of circuit inputs & secondary variables
  4) derive transition table & box stable states
  5) derive flow table by undoing state assignment from transition table
- e.g



SOLUTION

↳ no latches but there's feedback paths so it's async

↳ 1 input x, 1 output z, 2 excitation variables Y_2Y_1 & 2 secondary variables y_2y_1

$Y_1 = xy_1 + \bar{x}y_2$

$Y_2 = \bar{x}y_2 + x\bar{y}_1$

$z = \bar{y}_1 y_2$

Transition table:

Current state y_2y_1	Next state Y_2Y_1		Output z	
	$x=0$	$x=1$	$x=0$	$x=1$
00	**[00]**	10	0	0
01	00	**[01]**	0	0
10	11	**[10]**	1	1
11	**[11]**	01	0	0

TIP
- box all stable states (i.e. current state = next state)

↳ if $y_2y_1 = 00$ & $x = 0$, $Y_2Y_1 = 00$ as well but if $x = 0 \to 1$, then Y_2Y_1 will change to 10 & y_2y_1 will change to 10 after some delay

↳ according to table, will end up in another stable state where $y_2y_1 = 10$ & $x = 1$

↳ if circuit always reaches steady state when input variable is changed, output only needs to be specified when it's in steady state

Alternative transition table:

Current state y_2y_1	Next state Y_2Y_1		Output z	
	$x=0$	$x=1$	$x=0$	$x=1$
00	**[00]**	10	0	–
01	00	**[01]**	–	0
10	11	**[10]**	–	1
11	**[11]**	01	0	–

↳ when circuit's unstable, z is irrelevant & can potentially change

↳ since there's 2 feedback paths, potential for 4 states

↳ undo binary assignment to get flow table

Flow table:

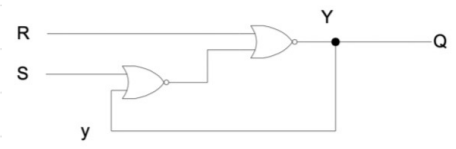
Current state y_2y_1	Next state Y_2Y_1		Output z	
	$x=0$	$x=1$	$x=0$	$x=1$
a	**[a]**	c	0	–
b	a	**[b]**	–	0
c	d	**[c]**	–	1
d	**[d]**	b	0	–

↳ a = 00, b = 01, c = 10, d = 11

or

	$x=0$	$x=1$
a	**[a]**, 0	c, 0
b	a, 0	**[b]**, 0
c	d, 1	**[c]**, 1
d	**[d]**, 0	b, 0

- flow table w/only 1 stable state per row is primitive flow table
- e.g. analyze SR latch



S	R	Q	$\bar{Q}$	
1	0	1	0	
0	0	1	0	(after $S=1, R=0$)
0	1	0	1	
0	0	0	1	(after $S=0, R=1$)
1	1	0	0	

SOLUTION

↳ equation for excitation variable & output is same:

$$Y = Q = \overline{R + (\overline{S + y})}$$
$$= \bar{R}(\overline{\overline{S + y}})$$
$$= \bar{R}(S + y)$$
$$= \bar{R}S + \bar{R}y$$

↳ to avoid SR = 11 situation, write $Y = S + \bar{R}y$ if SR = 0

Transition table:

Current state	Next state Y				Output
y	SR = 00	01	11	10	Q
0	[0]	[0]	[0]	1	0
1	[1]	0	0	[1]	1

Flow table:

Current state	Next state Y				Output
y	SR = 00	01	11	10	Q
a	[a]	[a]	[a]	b	0
b	[b]	a	a	[b]	1

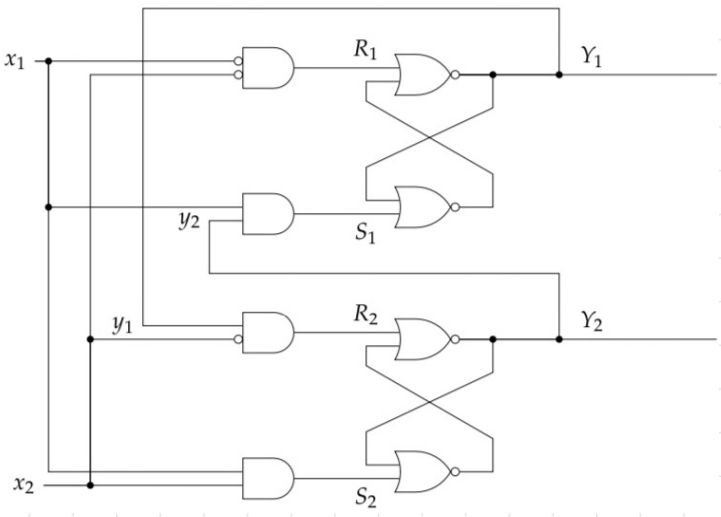
TIP
- SR ≠ 11 for NOR latches
- SR ≠ 00 for NAND latches

↳ undesirable case when SR = 11 & inputs change
- if SR = 11 → SR = 10 → SR = 00, then stable state has output 1
- if SR = 11 → SR = 01 → SR = 00, then stable state has output 0

↳ stable state is unpredictable so be careful if we transition from states & need to pass through SR = 11

· analysis when latches are present:
1) label latch outputs as Y_i & feedback path as y_i
2) derive logic equations for latch inputs S_i & R_i in terms of circuit inputs & secondary variables
3) check that SR = 0 for NOR latches & S'R' = 0 for NAND latches
4) create logic equations for latch outputs Y_i using known behaviour of latch
   - $Y = S + R'y$ for NOR latches
   - $Y = S' + Ry$ for NAND latches
5) construct transition table using logic equations for latch outputs & box stable states

e.g



SOLUTION

↳ 2 inputs x_2x_1, 2 excitation variables Y_2Y_1 (latch outputs), & 2 secondary variables y_2y_1

↳ logic equations for latch inputs:

$S_1 = x_1y_2$, $R_1 = \bar{x}_1\bar{x}_2$

$S_2 = x_1 x_2$, $R_2 = \overline{x}_2 y_1$

$S_1 R_1 = x_1 y_2 \overline{x}_1 \overline{x}_2 = 0$

$S_2 R_2 = x_1 x_2 \overline{x}_2 y_1 = 0$

↳ since there's NOR latches, we see $SR = 0$ ∴ latches don't ever enter undesirable state

↳ excitation equations:

$Y_1 = S_1 + R_1' y_1$

$= x_1 y_2 + (\overline{\overline{x}_1 \overline{x}_2}) y_1$

$= x_1 y_2 + (x_1 + x_2) y_1$

$= x_1 y_2 + x_1 y_1 + x_2 y_1$

$Y_2 = S_2 + R_2' y_2$

$= x_1 x_2 + (\overline{\overline{x}_2 y_1}) y_2$

$= x_1 x_2 + (x_2 + \overline{y}_1) y_2$

$= x_1 x_2 + x_2 y_2 + \overline{y}_1 y_2$

Transition table:

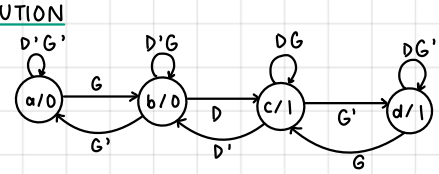
Current state	Next state $Y_2 Y_1$			
$y_2 y_1$	$x_2 x_1 = 00$	01	11	10
00	**00**	**00**	10	**00**
01	00	**01**	11	**01**
11	00	**11**	**11**	01
10	**10**	**10**	11	11

ASYNCHRONOUS CIRCUIT DESIGN

- when designing, make **fundamental mode assumption**
  - ↳ only 1 input can change at any time
  - ↳ circuit has to stable state prior to changing a circuit input
- derive **primitive flow table**, meaning there's exactly 1 stable state per row
- **design procedure** for async circuit:
  1) derive state diagram ∴ primitive flow table using fundamental mode assumption
     - begin from any initial stable state
     - let input change ∴ transition to another stable state
     - continue until we've returned to prev created state
     - fill in any additional transitions
  2) perform state reduction to obtain smaller flow table w/ less states
  3) perform state assignment to obtain transition table
     - must avoid critical races
  4) derive equations for excitation variables (next state) in terms of secondary variables (current state) ∧ circuit inputs
     - avoid combinational hazards
  5) derive output equations in terms of secondary variables ∴ circuit inputs
     - avoid output glitches
  6) draw circuit
- e.g.

> We will consider an example to illustrate the procedure. Consider a circuit with two inputs D and G. The circuit has one output Q. When $G = 0$, the output Q maintains its current value, but when $G = 1$, the output Q is equal to the input D. Design an asynchronous circuit that exhibits this behaviour.

SOLUTION



↳ by changing inputs one at time, can derive incomplete

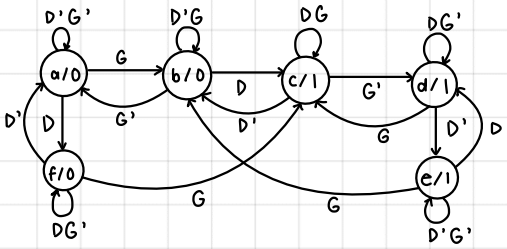
diagram of 4 stable states

↳ in state d, circuit is holding output value of 1

- if D = 1 → 0, then we would go to state a DG = 00 but output would be 1
- create another stable state w/ DG = 00 & Q = 1

↳ same logic applies above for when D = 0 → 1 so we need another state w/ DG = 10 & Q = 0

Final state diagram:



Primitive flow table:

Current state	Next state				Output Q			
	DG = 00	01	11	10	DG = 00	01	11	10
a	a	b	–	f	0	–	–	–
b	a	b	c	–	–	0	–	–
c	–	b	c	d	–	–	1	–
d	e	–	c	d	–	–	–	1
e	e	d	–	b	1	–	–	–
f	a	–	c	f	–	–	–	0

Reduced flow table:

a, b, c, d, e, f
- a, b, f → a
- c, d, e → c

Current state	Next state				Output Q			
	DG = 00	01	11	10	DG = 00	01	11	10
a	a	a	c	a	0	0	–	0
c	c	a	c	c	1	–	1	1

Let a = 0 & c = 1.

Transition table:

Current state y	Next state Y				Output Q			
	DG = 00	01	11	10	DG = 00	01	11	10
0	0	0	1	0	0	0	–	0
1	1	0	1	1	1	–	1	1

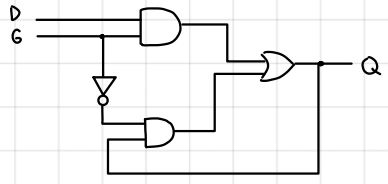
Y

y \ DG	00	01	11	10
0	0	0	1	0
1	1	0	1	1

$Y = G'y + DG$

$Q = y$

Final circuit:



- to implement circuits w/ latches, consider their excitation tables (assume SR = 11 nvr occurs for NOR latch & SR = 00 nvr occurs for NAND latch):

↳ SR(NOR) latch:

S	R	y	Y	
0	X	0	0	hold / reset
1	0	0	1	set
0	1	1	0	reset
X	0	1	1	hold / set

↳ S'R' (NAND) latch:

S	R	y	Y	
1	X	0	0	hold / reset
0	1	0	1	set
1	0	1	0	reset
X	1	1	1	hold / set

- e.g. reconsider above example & try to make circuit w/ S'R' latches
  - ↳ use K-maps to get necessary latch inputs
    - S

y \ DG	00	01	11	10
0	1	1	0	1
1	X	1	X	X

S = D' + G'

S = (DG)'

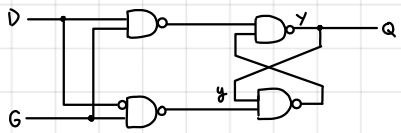
R

y \ DG	00	01	11	10
0	X	X	1	X
1	1	0	1	1

R = G' + D

R = (D'G)'

  - ↳ check to see SR = 00 nvr happens
    - S + R = (DG)' + (D'G)'
      = D' + G' + G' + D
      = D' + D + G'
      = 1 + G'
    - thus, S + R ≠ 0 so S & R will nvr be 0 at same time
  - ↳ final circuit:



ASYNCHRONOUS STATE REDUCTION

- since there's a lot of unspecified entries in flow table, can group them w/ anything
- instead of being equivalent, states are compatible
  - ↳ states A & B produce same outputs & have compatible next states where specified
- e.g. reduce flow table:

curr state	inputs (DG)				output			
	00	01	11	10	00	01	11	10
a	c	(a)	b	-	-	0	-	-
b	-	a	(b)	e	-	-	1	-
c	(c)	a	-	d	0	-	-	-
d	c	-	b	(d)	-	-	-	0
e	f	-	b	(e)	-	-	-	1
f	(f)	-	-	e	1	-	-	-

SOLUTION

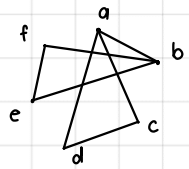
Build implication chart:

b	✓				
c	✓ (d,e)×				
d	✓ (d,e)×	✓			
e	(c,f)×	✓	(c,f)× (d,e)×	X	
f	(c,f)×	✓	X	(c,f)× (d,e)×	✓
	a	b	c	d	e

NOTE

- remember to check each state is included at least once ? implied compatibilities are true

Merger diagram:



↳ possible cliques are (a, c, d) ? (b, e, f)

Reduced flow table:

Current state	Next state				Output			
	DG=00	01	11	10	DG=00	01	11	10
a	a	a	b	a	0	0	–	0
b	b	a	b	b	1	-	1	1

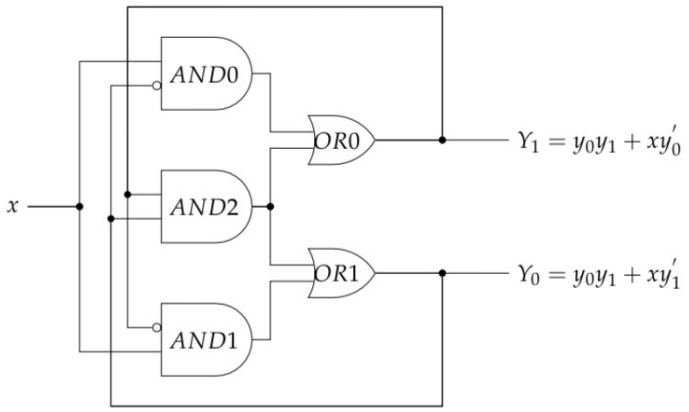
week 11

RACES

- **race condition** occurs in async circuit when 2+ state variables are required to change at same time in response to input change
  - ↳ unequal circuit delays means that state variables might not change at same time
- 2 types of races:
  - ↳ **non-critical**: circuit reaches final state regardless of order that state variables change
  - ↳ **critical**: circuit can reach diff final states depending on order that state variables change

always avoid designs w/ critical races

- e.g. circuit below has **critical race**



TIP
- when finding next states due to input changes, keep following transition table until a stable state is reached

↳ transition table:

Current state y_1y_0	Next state Y_1Y_0 $x = 0$	$x = 1$
00	**00**	11
01	00	**01**
11	**11**	**11**
10	00	**10**

↳ assume current state is 00 ∴ input changes $x = 0 \rightarrow 1$
- correct transition is to stable state 11

↳ if all circuit delays are equal, Y_1Y_0 would change $00 \rightarrow 11$ as expected

↳ if delay thru AND0 ∴ OR0 is very fast, then Y_1 will change $0 \rightarrow 1$ almost instantaneously ∴ it'll propagate back as $y_1 = 1$ to inputs of AND1 ∴ AND2
- $00 \rightarrow 10$ ∴ final stable state state is 10, which is wrong

↳ if delay thru AND1 ∴ OR1 is very fast, then Y_0 will change $0 \rightarrow 1$ almost instantaneously ∴ it'll propagate back as $y_0 = 1$ to inputs of AND0 ∴ AND2
- $00 \rightarrow 01$ ∴ final stable state is 01, which is wrong

- e.g. below is **non-critical race** (don't need to be concerned abt)

Current state y_1y_0	Next state Y_1Y_0 $x = 0$	$x = 1$
00	**00**	11
01	11	**01**
11	10	01
10	**10**	11

↳ 3 possible scenarios:

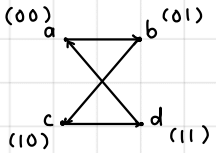
- [00] → 11 → [01]
- [00] → [01]
- [00] → 10 → 11 → [01]

↳ regardless of state variables' change order, always end up in 01 stable state

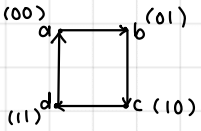
- races are a result of state assignment
  - ↳ don't exist in flow tables b/c there's no binary state assignment
  - ↳ avoid races (preemptive strategy by being careful during state assignment

- transition diagrams can help w/ visualizing when races occur
  - ↳ usually drawn as multi-dimensional cube
  - ↳ e g transition diagram for flow table:

Current state	Next state $x=0$	Next state $x=1$
a	[a]	b
b	c	[b]
c	[c]	d
d	a	[d]



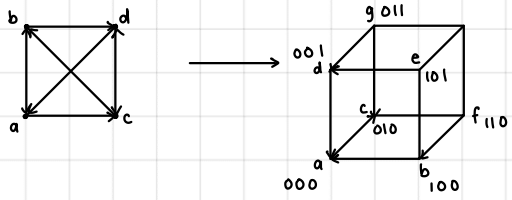
  - ↳ diagonal edges require 2+ state variables to be changed ∴ indicates race
    - redraw diagram to avoid race:



- races can be avoided using transition diagram
  - ↳ assign states to corners of cube ∴ adjust flow table so that transitions are made along edges of cube rather than diagonals
  - ↳ might need addition of temp unstable states
- e.g. remove races from below flow table:

Current state	Next state $x_1x_0=00$	$x_1x_0=01$	$x_1x_0=11$	$x_1x_0=10$
a	[a]	[a]	c	b
b	a	[b]	d	[b]
c	[c]	b	[c]	d
d	c	a	[d]	[d]

SOLUTION



TIP
- to do state assignment, use coordinates of each cube vertex

↳ state e allows b → d

↳ state f allows c → b

↳ state g allows c → d ∴ d → c

Transition table:

	Current state	Next state $x_1x_0=00$	01	11	10
000	a	[a]	[a]	c	b

100	b	a	**b**	e	**b**
010	c	**c**	f	**c**	g
001	d	g	a	**d**	**d**
101	e	-	-	d	-
110	f	-	b	-	-
011	g	c	-	-	d

- can use **one hot encoding** to avoid races
  - ↳ since each state has only 1 unique active bit, we require 2 bits to change when changing states
    - use **intermediate unstable states** ∴ force bits to change 1 at a time
  - ↳ i.e. to go from state i to j: $0\ldots\underset{i}{0}1\,0\ldots\underset{j}{000}\ldots0 \longrightarrow 0\ldots\underset{i}{00}\,0\ldots\underset{j}{01}0\ldots0$, we'll use intermediate state:

$$0\ldots\underset{i}{0}1\,0\ldots\underset{j}{000}\ldots0 \longrightarrow 0\ldots\underset{i}{0}1\,0\ldots\underset{j}{01}0\ldots0 \longrightarrow 0\ldots\underset{i}{00}\,0\ldots\underset{j}{01}0\ldots0$$

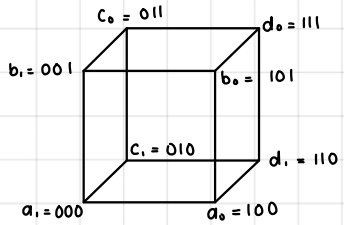
- e.g. remove races from below flow table:

	Current state	Next state $x_1x_0=00$	$x_1x_0=01$	$x_1x_0=11$	$x_1x_0=10$
0001	*a*	**a**	**a**	c	b
0010	*b*	a	**b**	d	**b**
0100	*c*	**c**	b	**c**	d
1000	*d*	c	a	**d**	**d**

SOLUTION

State assignment	Current state	Next state $x_2x_1=00$	01	11	10
0001	a	**a**	**a**	e	f
0010	b	f	**b**	g	**b**
0100	c	**c**	h	**c**	i
1000	d	i	j	**d**	**d**
0101	e	-	-	c	-
0011	f	a	-	-	b
1010	g	-	-	d	-
0110	h	-	b	-	-
1100	i	c	-	-	d
1001	j	-	a	-	-

- **state duplication** can be used to avoid races when # of states ≤ 4
  - ↳ duplicate each state into pair of equivalent states
- drawn as cube:



  - ↳ can move btwn equivalent states by changing only 1 bit b/c they're adjacent to each other

↳ each original state is adjacent to each other when equivalent states are taken into account

· e.g.

Current state	Next state $x_1x_0 = 00$	$x_1x_0 = 01$	$x_1x_0 = 11$	$x_1x_0 = 10$
a	$\boxed{a}$	$\boxed{a}$	c	b
b	a	$\boxed{b}$	d	$\boxed{b}$
c	$\boxed{c}$	b	$\boxed{c}$	d
d	c	a	$\boxed{d}$	$\boxed{d}$

↓

State assignment	Current state	Next state $x_2x_1 = 00$	$x_2x_1 = 01$	$x_2x_1 = 11$	$x_2x_1 = 10$
000	a_1	$\boxed{a_1}$	$\boxed{a_1}$	c_1	b_1
100	a_0	$\boxed{a_0}$	$\boxed{a_0}$	a_1	b_0
001	b_1	a_1	$\boxed{b_1}$	b_0	$\boxed{b_1}$
101	b_0	a_0	$\boxed{b_0}$	d_0	$\boxed{b_0}$
010	c_1	$\boxed{c_1}$	c_0	$\boxed{c_1}$	d_1
011	c_0	$\boxed{c_0}$	b_1	$\boxed{c_0}$	d_0
110	d_1	c_1	a_0	$\boxed{d_1}$	$\boxed{d_1}$
111	d_0	c_0	d_1	$\boxed{d_0}$	$\boxed{d_0}$

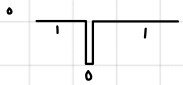
HAZARDS

· hazards aren't unique to async circuits ∴ can happen in any circuit

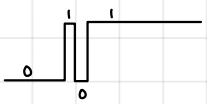
· **hazard**: momentary unwanted switching transient at output of logic function
  ↳ produces glitch at output of circuit
  ↳ due to propagation delay along diff paths in combinational circuit
  ↳ can be static / dynamic ∴ classified by type of glitch they produce

· **static hazards** are when output of circuit shouldn't change due to input change but there's a brief period of time where it switches
  ↳ **static-0 hazard**: function should've remained constant at 0

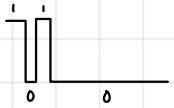
  ↳ **static-1 hazard**: function should've remained constant at 1



· **dynamic hazards** are when inputs change ∴ output is supposed to change from 0→1 or 1→0 but logic output flips 1+ times
  ↳ signal change 0→1:



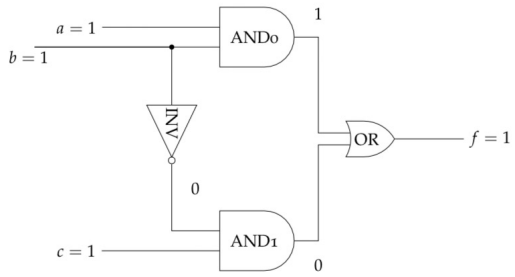
  ↳ signal change 1→0:



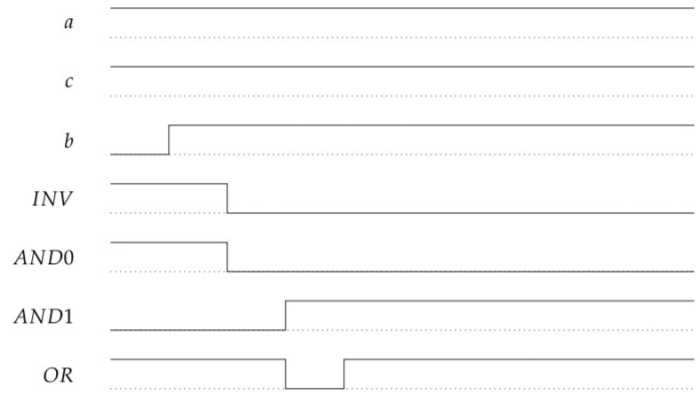
· in sequential circuits, hazards aren't concern b/c we have clock period for signals to stabilize prior to active clock edge

· to illustrate static hazard, consider SOP implementation of $f = ab + \bar{b}c$
  ↳ assume every logic gate incurs 1 unit of delay
  ↳ b changes 1→0 at time t=0

↳

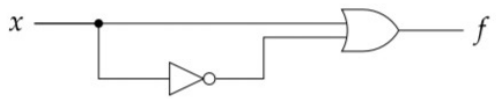


↳ timing diagram:



- OR gate output should've always remained at 1 but temporarily changed to 0
- circuit has static-1 hazard

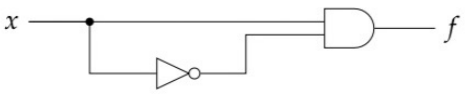
· basic static-1 hazards are characterized by hypothetical circuit:



↳ $f = x + \bar{x}$ would logically always result in $f = 1$ but can't rely on both x & $\bar{x}$ arriving at input of OR gate at same time due to circuit delay

- consider them as separate signals
- OR gate might see 0 at both inputs for a brief moment

· basic static-0 hazards are characterized by hypothetical circuit:



↳ $f = x\bar{x}$ will always logically be $f = 0$ but if considering x & $\bar{x}$ as separate signals, AND gate will see 1 at both inputs for brief moment

· SOP expressions may potentially have static-1 hazards

↳ won't have static-0 & dynamic hazards

↳ to mask (i.e. cover) hazards in SOP expressions, add redundant product terms

- if adjacent minterms aren't covered by same product term, hazard exists

↳ e.g. mask hazards in logic function $f = ab + \bar{b}c$

SOLUTION

a \ bc	00	01	11	10
0	0	1	0	0
1	0	1	1	1

When b changes 1 → 0, we jump from 1 product term to another so add redundant term that's independent of b

↳ $f = 1$ will hold while $b = 1 \rightarrow 0$

Final SOP implementation of f is $f = ab + \bar{b}c + ac$

- POS expressions may potentially have static-0 hazards
  - ↳ won't have static-1 & dynamic hazards
  - ↳ to mask hazards in POS expressions, add redundant sum terms
    - if adjacent maxterms aren't covered by same sum term, hazard exists
- to fix hazards w/ latches:
  - ↳ SR latch can tolerate momentary 0s at inputs (might briefly go to hold state from set/reset state)
  - ↳ S'R' latch can tolerate momentary 1s at inputs
  - ↳ e.g. mask hazards in logic function $f = ab + \bar{b}c$ w/ latches

SOLUTION

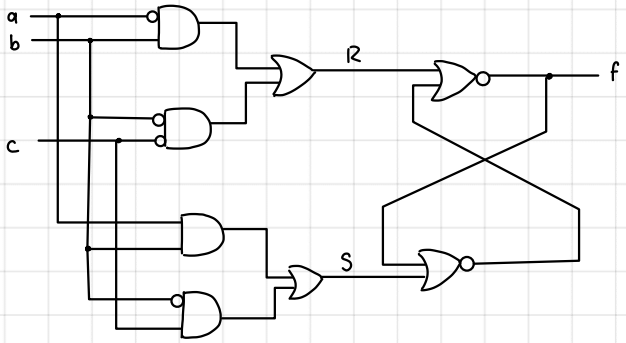
a \ bc	00	01	11	10
0	0	1	0	0
1	0	1	1	1

As seen previously, we need to fix static-1 hazards so use SR latches to tolerate momentary 0s at latch inputs

- ↳ latch is set when f = 1
- ↳ latch is reset when f = 0

$S = ab + b'c$

$R = b'c' + a'b$



- dynamic hazards occur in multilevel circuits in which there's 3+ paths from circuit inputs to outputs & they all have unequal delays
  - ↳ more difficult to mask
  - ↳ could instead implement f as 2-level SOP/POS & remove static hazards

flow/transition tables might have unspecified entries for circuit outputs but we might temporarily pass thru them while transitioning from diff stable states

- ↳ if some outputs are don't cares, can have glitches at circuit outputs

to avoid output glitches, when transitioning btwn 2 stable states b/c of input change:

- ↳ if both stable states produce same output, change don't care to that value
- ↳ if stable states produce diff outputs, leave don't care as is

- e.g.

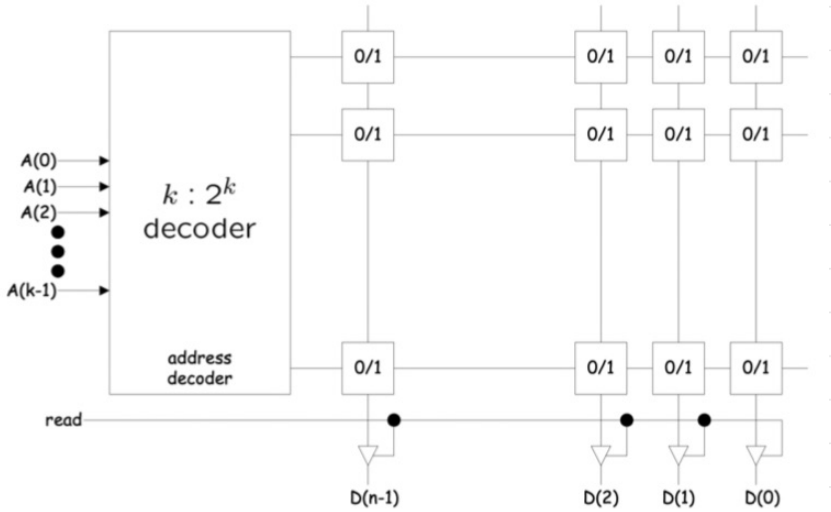
curr state	next state x=0	next state x=1	output x=0	output x=1
a	(a)	b	0	-
b	c	(b)	-	0
c	(c)	d	1	-
d	a	(d)	-	1

curr state	next state x=0	next state x=1	output x=0	output x=1
a	(a)	b	0	0
b	c	(b)	-	0
c	(c)	d	1	1
d	a	(d)	-	1

- ↳ above changes will avoid temp glitches at outputs

RAMS AND ROMS

- **Read Only Memory (ROM)** is device that allows permanent storage of info
  - ↳ has k input/address lines & n output/data lines
  - ↳ can store $2^k \cdot n$ bits of info inside device
  - ↳ **address lines** specify memory location
  - ↳ **data outputs** rep value stored at mem location specified on address line
  - ↳ block diagram of ROM:

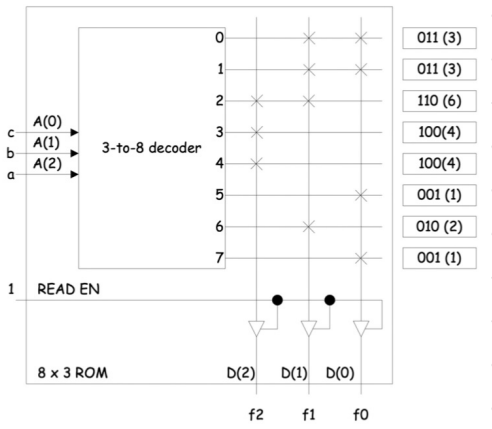


- can implement multi-input/output logic functions inside ROM
  - ↳ data outputs are logic functions & address lines are logic function inputs
  - ↳ create **ROM table** to store logic functions
    - e.g. implement 3-input logic functions using ROM: $f_0 = \Sigma(0,1,5,7)$, $f_1 = \Sigma(0,1,2,6)$, $f_2 = \Sigma(2,3,4)$

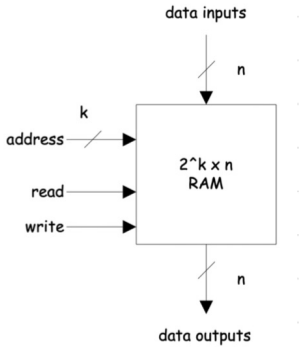
SOLUTION

a	b	c	f_2	f_1	f_0
0	0	0	0	1	1
0	0	1	0	1	1
0	1	0	1	1	0
0	1	1	1	0	0
1	0	0	1	0	0
1	0	1	0	0	1
1	1	0	0	1	0
1	1	1	0	0	1

→ Xs rep where f = 1

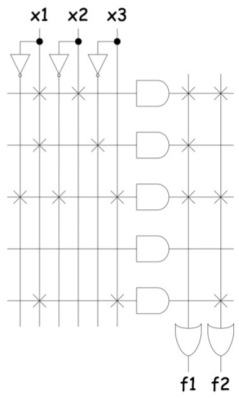


- **Random Access Memory (RAM)** is storage device that can read & write info
  - ↳



- programmable logic devices are "generic" b/c they can be programmed to implement wide variety of circuits

- **Programmable Logic Array (PLA)**: can implement functions expressed in SOP
  - ↳ consists of input buffers & inverters, programmable AND plane, then programmable OR plane
  - ↳ can implement m logic functions of n variables
    - limit is # of product terms that can be generated inside device
  - ↳ e.g. 2 logic functions w/ 3-5-2 PLA:

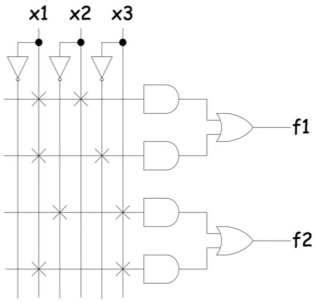


$$f_1 = x_1x_2 + x_1\bar{x}_3 + \bar{x}_1\bar{x}_2x_3$$

$$f_2 = x_1x_2 + \bar{x}_1\bar{x}_2x_3 + x_1x_3$$

NOTE
- Xs in AND & OR plane rep programmable bits that can be set to make connections

- **Programmable Array Logic (PAL)** is similar to PLA but only AND plane is programmable & OR plane is fixed
  - ↳ e.g.



$$f_1 = x_1x_2 + x_1\bar{x}_3$$

$$f_2 = \bar{x}_2x_3 + x_1x_3$$