Syllabus

January 10, 2019 8:37 PM

Digital Logic Design: Basic Concepts

January 10, 2019 10:23 PM

Binary Logic

- Binary logic consists of:
  - Binary variables
    - Denoted with letters such as x, y, etc.
    - Have 2 distinct values: 0 or 1
  - Logical operations
    - AND
    - OR NOT

  - AND
    - Notation: x · y (x and y)
    - Alternative notation: xy
    - Truth table

x	y	xy
0	0	0
0	1	0
1	0	0
1	1	1

  - OR
    - Notation: x + y (x OR y)
    - Truth table

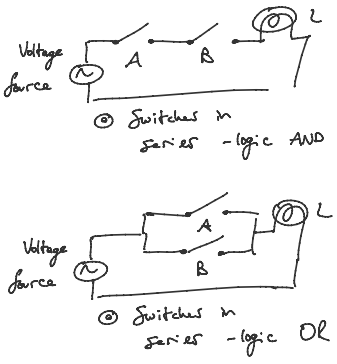
x	y	xy
0	0	0
0	1	1
1	0	1
1	1	1

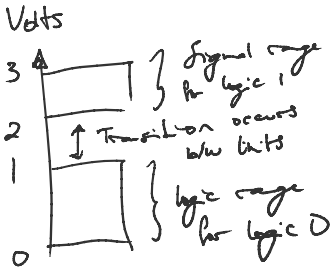
  - NOT
    - Notation: x'
    - Alternative notation: $\bar{x}$

x	$\bar{x}$
0	1
1	0

Switching Circuits and Binary Signals

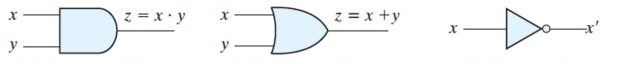
- Logic Diagrams can be realized with electrical switching circuits:
  ------(DIAGRAM)------>>>>>>>>>>
- In solid-state circuits, transistors are used to switch signals
  - Logic 1 is represented with a high voltage
    - Typically 3.3V or 5V
  - Logic 0 is represented with a low voltage
    - Usually 0V
- Mano, p.30 or 36 (Example of Binary Signals)
  ------(DIAGRAM)------>>>>>>>>>>





Logic Gates

- Are hardware circuits that produce a logic-1 or logic-0 output signal
-



(a) Two-input AND gate (b) Two-input OR gate (c) NOT gate or inverter

FIGURE 1.4
Symbols for digital logic circuits

Readings and Exercises

Skim Mano Section 1.1 - 1.7
Read 1.8 - 1.9

////////////////////////////////////

Boolean Algebra

- George Boole developed Boolean algebra in 1854
  - Is an algebraic system for describing logic
- Claude Shannon introduced two-valued Boolean algebra in 1938
  - He called it switching algebra
  - Suitable for describing electrical switching circuits

Two-valued Boolean Algebra

- Has a set of 2 elements, B = {0.1}
- Has the 2 binary operators · and + (AND and OR)
  - Their results are strictly defined:

x	y	x · y
0	0	0
0	1	0
1	0	0
1	1	1

x	y	x + y
0	0	0
0	1	1
1	0	1
1	1	1

- Define a complement (NOT operator):
- Has a number of basic postulates:
  - Are assumptions from which theorems and properties are deduced

Basic Postulates

1. Closed with respect to + and ·
   a. The result of these operators is always 0 or 1 (see truth tables)
2. A) The identity element for + is 0
   B) The identity element for · is 1
3. Commutative with respect to + and ·
   a. $x + y = y + x$
   b. $x \cdot y = y \cdot x$
4. A) · is distributive over +
   a. $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$
   B) + is distributive over ·
   a. $x + (y \cdot z) = (x + y) \cdot (x + z)$
5. For every x, there exists a complement x' that satisfies the rules $x + x' = 1$ and $x \cdot x' = 0$
6. There are at least 2 elements x and y in the set of elements B such that $x \neq y$.

Duality Principle

- A valid algebraic expression can be derived from another by interchanging the operators and identity elements.
- To find the *dual*:
  - Interchange · and + operators
  - Replace the 1's with 0's, and vice versa
- Eg: the dual of $x + 1 = 1$ is $x \cdot 0 = 0$
- Most of the postulates and derived theorems have duals

Basic theorems

- Theorem 1
  $x + x = x$
  $x \cdot x = x$
- Theorem 2
  $x + 1 = 1$
  $x \cdot 0 = 0$
- Theorem 3, involution
  $(x')' = x$
- Theorem 4, associative law
  $x + (y + z) = (x + y) + z$
  $x(yz) = (xy)z$
- Theorem 5, DeMorgan's theorem
  $(x + y)' = x'y'$
  $(xy)' = x' + y'$
- Theorem 6, absorption (can be used for a larger problem)
  $x + xy = x$
  $x(x + y) = x$
- Algebraic proofs exists for all these problems
  - Can be checked w/ truth problems

Operator Precedence

(From high to low)

Operator	Precedence
( )	1
'	2
·	3
+	4

- Eg: (x + y)'
  - OR inside ( ) done first
  - NOT done second

Boolean Functions

- Are algebraic expressions formed with:
  - Binary variables
  - Binary operators OR (+) and AND (·)
  - Unary operator NOT (')
  - Parentheses
  - Equal sign

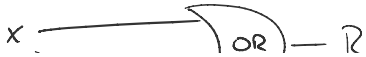
- Eg: F1 = xyz'

- Boolean Functions can be represented with truth tables
  - List 2^n combinations of 0's and 1's for n number of binary variables
    - Each is a row in the table
    - Count from 0 to (2^n) -1
  - Create a column that shows when the function equals 0 or 1

- Eg: F1 = x + y'z F2 = xy' + x'z

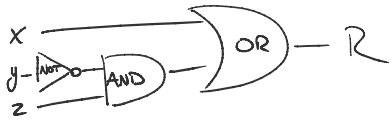
X	Y	Z	F1	F2
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	0	1
1	0	0	1	1
1	0	1	1	1
1	1	0	1	0
1	1	1	1	0

- Two functions are equal if they have the same value for all possible combinations of variables
  - i.e. The columns match



1	1	0	1	0
1	1	1	1	0

- Two functions are equal if they have the same value for all possible combinations of variables
  - i.e. The columns match
- Boolean functions can be represented with a logic diagram composed of AND, OR, and NOT gates
- -----DIAGRAM ----->>>>>



Algebraic Minimization

- A *literal* is a primed or unprimed variable
  - Each literal in a function corresponds to an input to a gate
- A *term* consists of one or more literals grouped together with an AND or OR operator
  - Eg: f = x'y'z' + x'yz + xy' has 3 terms and 8 literals
  - Each term is implemented with a gate
- In designing logic, we often begin with a truth table
  - From this a Boolean function is obtained
  - From this a hardware design is made
- The hardware design should be as simple as possible (i.e. minimized)
  - Reduces cost
  - Reduces power consumption
  - Reduces package count
  - Increases speed
  - Simplifies testing and debugging
- Algebraic minimization of a function can be attempted using the postulates and theorems
  - Minimize the number of literals
  - Minimize the number of terms

- Eg: minimize f = x'y'z + x'yz + xy'
  - Note: has 3 terms and 8 literals

f = x'y'z + x'yz + xy'
= x'zy' + x'zy + xy' (Commutative property)
= x'z(y' + y) + xy' (Distributive property)
= x'z(1) + xy' (Postulate 5a)
= x'z + xy' (Postulate 2b)

- Note: now has 2 terms and 4 literals

- Eg: minimize f = yz + z'
  - Basic idea: factor out terms of type (A + A'), which equal 1

F = yz + z'
= z' + y (Commutative property)
= (z' + y)(z' + z) (Distributive property)
= (z' + y) (1) (Postulate 5a)
= z' + y (Postulate 2b)

- Eg: minimize f = y (y' + z) (x + z')
  - Basic idea: factor out terms of type AA', which = 0

f = y (y' + z) (x + z')
= (yy' + yz) (x + z') (Distributive property)
= (0 + yz) (x + z') (Postulate 5b)
= yz (x + z') (Postulate 2a)
= xyz + y (zz') (Distributive property + Associative property)
= xyz + y (0) (Postulate 5b)
= xyz + 0 (Theorem 2)
= xyz (Postulate 2a)

Complement of a function

- The complement of a function F is F
  - In the truth table, invert all the 0's and 1's in the column that represents the function
- DeMorgan's theorem is used to find the complement algebraically
  - For 2 variables:
    - (x + y)' = x'y'
    - (xy)' = x' + y'
- DeMorgan's theorem can be extended to any number of variables
  - (A + B + C + ... + F)' = A'B'C'...F'
  - (ABC...F)' = A' + B' + C' + ... + F'
- To find the complement of a function :
  - Complement the entire function
  - Apply DeMorgan's theorem repeatedly

- Eg: f = xyz' + x'yz

F' = [xyz' + x'yz]'
= (xyz')' (x'yz)'
= (x' + y' + z) (x + y' + z')

- Short cut method:
  - Form the dual of the function
  - Complement each literal

- Eg: xyz' + x'yz

(x + y + z') (x' + y + z) dual
f' = (x' + y' + z) (x + y' + z')

Readings and Exercises

Read Mano 2.1 -2.5

//

Canonical Forms

- A *minterm* is one combination of binary variables ANDed together
  - If there are *n* number of variables, there are 2^n minterms, each having *n* literals
- A *maxterm* is one combination of binary variables ORed together
  There are 2^n maxterms, each having n literals
- Mano, p.49 (P.59)

Table 2.3
Minterms and Maxterms for Three Binary Variables

			Minterms		Maxterms	
x	y	z	Term	Designation	Term	Designation
0	0	0	$x'y'z'$	m_0	$x + y + z$	M_0
0	0	1	$x'y'z$	m_1	$x + y + z'$	M_1
0	1	0	$x'yz'$	m_2	$x + y' + z$	M_2
0	1	1	$x'yz$	m_3	$x + y' + z'$	M_3
1	0	0	$xy'z'$	m_4	$x' + y + z$	M_4
1	0	1	$xy'z$	m_5	$x' + y + z'$	M_5
1	1	0	xyz'	m_6	$x' + y' + z$	M_6
1	1	1	xyz	m_7	$x' + y' + z'$	M_7

- Any Boolean function can be represented as a *sum of minterms*
  - Can be determined from the truth table
- Eg:

	X	Y	Z	F	Minterms	Complements
0	0	0	0	0		x'y'z'
1	0	0	1	1	x'y'z' m_1	
2	0	1	0	1	x'yz' m_2	
3	0	1	1	0		x'yz
4	1	0	0	0		xy'z
5	1	0	1	1	xy'z m_5	
6	1	1	0	1	xyz' m_6	
7	1	1	1	1	xyz m_7	

- From the above:

F = x'y'z + x'yz' + xy'z + xyz' + xyz
 = $m_1 + m_2 + m_5 + m_6 + m_7$
= Σ (1,2,5,6,7)

- This is one of the canonical forms
  - Is def, **not** in min form.
- Consider the complement of f:

F' = x'y'z' + x'yz + xy'z'
F= (F')' = [x'y'z' + x'yz + xy'z']'
= (x + y + z) (x + y' + z') (x' + y + z) (DeMorgan)
= $M_0M_3M_4$
= π (0,3,4)

- This is another canonical form called the *product of maxterms*
- To convert from one canonical form to another:
  - Interchange Σ and π
  - List those numbers missing in the original form
  - Eg: F(x,y,z) = Σ (1,3,6,7) = π (0,2,4,5)
- If a function is not in canonical form, it can be converted by:
  - Using the truth table technique above, or
  - Algebraic manipulation

Eg: express f = A + B'C as a sum of minterms
F = A + B'C
= A (B + B') + B'C
= AB + AB' + B'C
= AB (C + C') + AB' (C + C') + B'C (A + A')
= ABC + ABC' + AB'C + AB'C' + **AB'C** + A'B'C'
= ABC + ABC' + (AB'C + **AB'C**) + AB'C'+ A'B'C'
= ABC + ABC' + AB'C + AB'C'+ A'B'C'
= A'B'C + AB'C' + ABC + ABC'+ ABC
= $m_1 + m_4 + m_5 + m_6 + m_7$
= Σ (1,4,5,6,7)

Standard Forms

- Sum of products (SOP)
  - Has one or more product terms
    - Consists of one or more literals ANDed ("multiplied") together
  - The product terms are ORed ("summed")
  - Eg: f = y' + xy + x'yz'

- Product of Sums (POS)
  - Has one or more *sum terms*
    - Consists of one or more literals ORed ("summed") together
  - The sum terms are ANDed ("multiplied")
  - Eg: f = x (y' + z) (x' + y + z' + w)

Other Logic Operations

- There are 2^(2ⁿ) functions for *n* binary variables
  - For 2 variables, there are 16 possible functions
  - (Mano, p.56): (65)
    - **Table 2.7** *Truth tables for the 16 functions of 2 Binary Variables*
    - ---------->>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
- These 16 functions can be expressed algebraically
  - Many can be expressed with special operator symbols
  - (Mano, p.56): (66)
    - **Table 2.8** *Boolean Expressions for the 16 Functions of 2 Variables*
- Of the new symbols, only exclusive-OR ⊕ and equivalence ⊙ are commonly used
- F_0 and F_{15} produce constant 0 or 1
- F_3 and F_5 are unary transfers
- F_{10} and F_{12} are unary complements
- All other functions are binary operators and 2 operands
- Exclusive-OR is often written as XOR
- Equivalence often written as XNOR
  - X ⊙ = (x ⊕ y)'

Digital Logic Gates

- Symbol for common logic gates are shown below (Mano, p.79) (**Figure 2.5**): ---->>>>
  - The small circle means to complement
- A buffer is used to amplify a signal
- NAND and NOR gate are commonly used
  - Are easily constructed with transistor circuits
  - Boolean functions are easily implemented with them

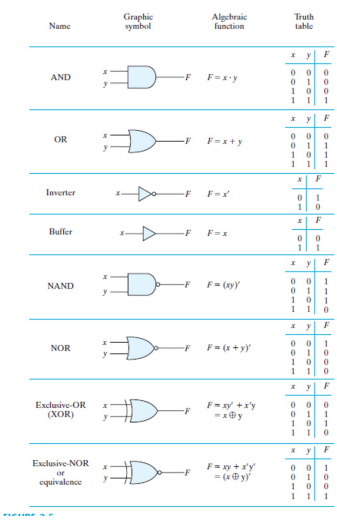
Readings and Exercises

Table 2.7
Truth Tables for the 16 Functions of Two Binary Variables

x	y	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Table 2.8
Boolean Expressions for the 16 Functions of Two Variables

Boolean Functions	Operator Symbol	Name	Comments
$F_0 = 0$		Null	Binary constant 0
$F_1 = xy$	$x \cdot y$	AND	x and y
$F_2 = xy'$	x/y	Inhibition	x, but not y
$F_3 = x$		Transfer	x
$F_4 = x'y$	y/x	Inhibition	y, but not x
$F_5 = y$		Transfer	y
$F_6 = xy' + x'y$	$x \oplus y$	Exclusive-OR	x or y, but not both
$F_7 = x + y$	$x + y$	OR	x or y
$F_8 = (x + y)'$	$x \downarrow y$	NOR	Not-OR
$F_9 = xy + x'y'$	$(x \oplus y)'$	Equivalence	x equals y
$F_{10} = y'$	y'	Complement	Not y
$F_{11} = x + y'$	$x \subset y$	Implication	If y, then x
$F_{12} = x'$	x'	Complement	Not x
$F_{13} = x' + y$	$x \supset y$	Implication	If x, then y
$F_{14} = (xy)'$	$x \uparrow y$	NAND	Not-AND
$F_{15} = 1$		Identity	Binary constant 1



- NAND and NOR gate are commonly used
  - Are easily constructed with transistor circuits
  - Boolean functions are easily implemented with them

Readings and Exercises

- Read Mano sections. 2.6 - 2.8

$F_{11} = x + y'$	$x \subset y$	Implication	If y, then x
$F_{12} = x'$	x'	Complement	Not x
$F_{13} = x' + y$	$x \supset y$	Implication	If x, then y
$F_{14} = (xy)'$	$x \uparrow y$	NAND	Not-AND
$F_{15} = 1$		Identity	Binary constant 1

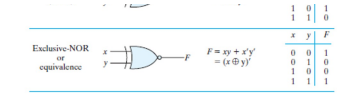


FIGURE 2.5
Digital logic gates

//

Simplification of Boolean Functions

The Map Method

- Developed in 1952 - 53 by E.W. Veitch and M. Karnaugh
  - Also known as Karnaugh maps (K-maps)
- A Boolean function can be represented with:
  - A *unique* truth table
  - *Many* different algebraic expressions
    - Most will be in minimized form
- The map method is a simple procedure for minimizing Boolean functions
  - Usually easier than algebraic minimization
- A map is a diagram made up of squares
  - Each square represents one minterms
  - The squares are ordered according to Gray code
    - Where one bit changes at a time
- A map represents all the possible ways a function can be expressed in standard form
  - By recognizing groupings of squares, one can derive alternative algebraic expressions
  - The simplest one is usually chosen
    - i.e. The one with the fewest literals

Two-Variable Maps

- Have 4 minterms, thus we have 4 squares

Figure 3.1 Two-variable K-Maps Mano (p.74) ---------->>>>>>>>>

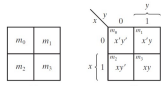


FIGURE 3.1
Two-variable K-map

- Note that there are 4 distinct "areas" equal to x, x', y and y'
- The function x'y' + x'y + xy' = $m_0 + m_1 + m_2$ is represented as:
  - Diagram ---------->>>>>>>>>>>>>>>>>>>
- To simplify, try to form groups of adjacent squares, all filled with 1:
  - Diagram ---------->>>>>>>>>>>>>>>>>>>
  - The top group is represented with the term x'
    - The value of y doesn't matter (can be 0 or 1)
  - The left group is represented with y'
    - The value of x doesn't matter
- Each group represents a term
- The complete function is formed by ORing the terms together
  - The function is simplified to: x' + y'
- This result can be confirmed algebraically:

F = x'y' + x'y + xy'

= (x'y' + x'y') + x'y + xy'	Theorem 1
= (x'y' + x'y) + (x'y' + xy')	Postulate 3
= x' (y' + y) + y' (x' + x)	Postulate 4a
= x' + y'	Postulate 5

Three-Variable Maps

- Have 8 minterms, and thus 8 squares
  - Are arranged in a Gray code sequence

Figure 3.3 Three-Variable K-map Mano (p75) ---------->>>>>>>>>>>>>>>>

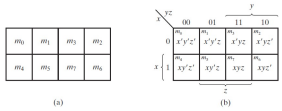
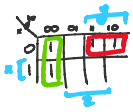


FIGURE 3.3
Three-variable K-map

- When 2 squares are grouped together, one of the literals can be dropped
  - Eg:

F = $m_1 + m_3$ = x'y'z + x'yz
= x'z (y' + y)
= x'z



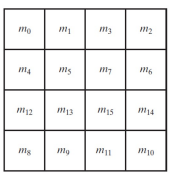
- Eg: Minimize Σ (0,2,3,4)
  - Diagram
  - 2 groups of 2 squares can be formed
    - The right group forms the term x'y
    - The left forms y'z'
  - The overall expression is formed by ORing these terms: x'y + y'z'
- If 4 squares are grouped together, 2 literals can be dropped
  - Eg: minimize f = Σ (0,1,2,3)
  - Diagram
    - f = x'
- Note that one sometimes "wraps around" to form a group
  - Eg: minimize f = Σ (0,2,4,6)
  - Diagram
    - f = z'
- If 8 squares are grouped together, the function is a constant 1
  - Eg: f = Σ (0,1,2,3,4,5,6,7)
  - Diagram
    - f = 1
- Sometimes squares are shared among 2 or more groups
  - Eg: f = Σ (1,2,3,6,7)
  - Diagram
    - f = x'z + y
- The number of squares in a group must be a power of 2

# of squares	# of literals in term
1	3
2	2
4	1
8	0 (f = 1)

Four-Variable Maps

- Have 16 minterms and squares
  - Figure 3.8 Four-Variable K-map Mano (p.80) ---------->>>>>>>>>>>>>>>
- The number of squares in the group determines the number of literals in the term

# of squares	# of literals in term
1	4
2	3
4	2
8	1



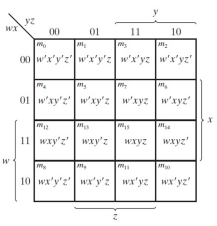
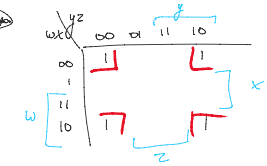


FIGURE 3.8
Four-variable map

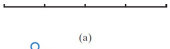
1	4
2	3
4	2
8	1
16	0 (f = 1)

- Eg: Simplify f = Σ (1, 5, 9, 12, 13, 14)
  - Diagram a
  - f = y'z + wxz'
- The four corners yield a group of 4
- Eg: Simplify f = Σ (0, 2, 8, 10)
  - Diagram
  - f = x'z'

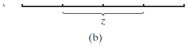


Readings and Exercises

- Read Mano sections. 3.1 - 3.3



(a)



(b)

FIGURE 3.8
Four-variable map

Digital Logic Design: Combinational Logic

January 10, 2019 10:28 PM

General

- A combinational circuit consists of:
  - Input variables
  - Logic gates
  - Output variables

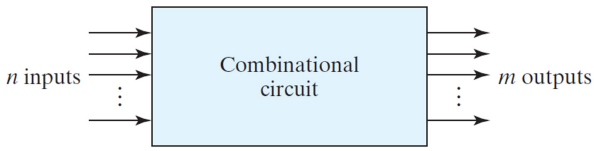


FIGURE 4.1
Block diagram of combinational circuit

Figure 4.1 Block Diagram of comb. (p. 126) ---------->>>>>>>>

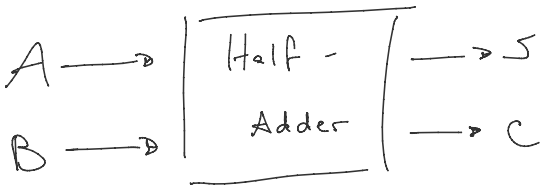
- For n input variables, there are 2^n possible input combinations
- There are m Boolean functions
  - One for each output variable
    - Each is expressed in terms of the input variables
    - Each is represented by one column in the truth table
- We will assume that the input variables are present in normal and primed form
  - i.e. Appear on 2 input wires

Design Procedure

1. State the problem in words
2. Determine the input and output variables
3. Assign letter symbols to the variables
4. Create the truth table that defines the relationships between inputs and outputs
5. Obtain the simplified function for each output
6. Implement the functions using appropriate gates

Half-Adder

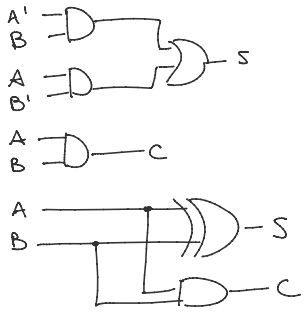
- Adds 2 bits together
  - 0 + 0 = 0
  - 0 + 1 = 1
  - 1 + 0 = 1
  - 1 + 1 = 10
  - Assumes there is no carry-in bit
- There are 2 input variables
  - A: augend
  - B: addend
- There are 2 output variables
  - S: sum
  - C: carry-out
  - Diagram



- Truth table:

A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

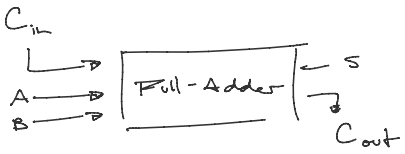
- The output function are:
  - S = A'B + AB' = A ⊕ B
  - C = AB
  - These cannot be simplified
- Two possible gate implementations:
  - Diagram



Full-Adder

- Like the half-adder, but also has a carry-in bit
- There are 3 inputs:

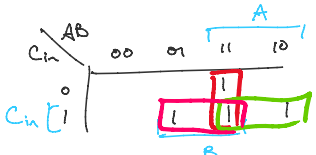
    - A: augend
    - B: addend
    - C_{in}: carry-in
- There are 2 outputs:
    - S: sum
    - C_{out}: carry-out
- Diagram



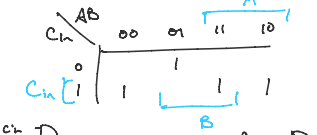
- Truth table 4.4 (p.135):

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

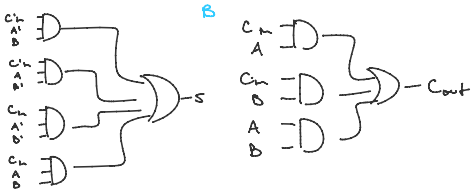
- Simplify each function (if possible):
- Diagram



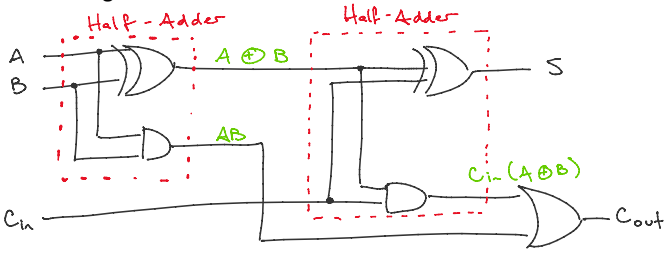
    - $C_{out} = C_{in}A + C_{in}B + AB$
- Diagram



    - $S = C'_{in}A'B + C'_{in}AB' + C_{in}A'B' + C_{in}AB$
- AND-OR implementation:
- Diagram



- Can be implemented with 2 half-adders
    - From the truth table, note that:
      $S = C_{in} \oplus (A \oplus B)$
    - $C_{out} = C_{in} (A \oplus B) + AB$
- Diagram



- Full adders are chained together to form a *binary adder*
    - Eg: 4-bit adder (Mano, p.138):
        - Figure 4.9 4-bit adder

Readings and Ex:

Read Mano sections 4.1 - 4.5

Digital Logic Design: Synchronous Sequential Logic

January 10, 2019 10:29 PM

Sequential Circuits

- Consists of:
  - Inputs
  - A combinational circuit
  - Memory elements
  - A feedback path from memory elements to the combinational circuit

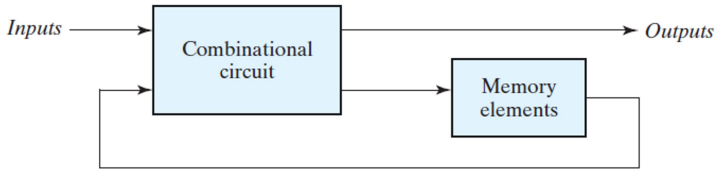


FIGURE 5.1
Block diagram of sequential circuit

- Memory elements are devices that store binary information
  - Defines the current *state* of the circuit
- A sequential circuit is defined by a time sequence of inputs, current state, and outputs
  - Outputs are a function of inputs and current state
  - The *next state* is also a function of inputs and current state

Synchronous Sequential Circuits

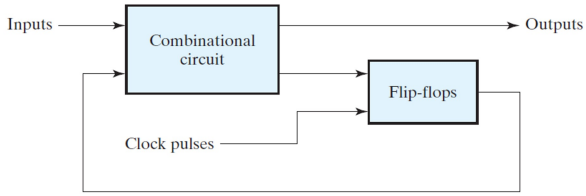
- Are circuits where changes of state occur at discrete instants of time
  - i.e. Are *synchronized*
- Most practical computer systems use clocked sequential circuits
  - A master clock generates a series of clock pulses
- 

(b) Timing diagram of clock pulses

  - The state changes with the arrival of each clock pulse

Set up Inputs ------->>>> clock pulse arrives ------->>>> outputs assume final state
>>>>>>>>>>>>>>>>----------TIME---------->>>>>>>>>>>>>>>>>

- The clock pulses are fed into memory units called *flip-flops*:



(a) Block diagram

(b) Timing diagram of clock pulses

FIGURE 5.2
Synchronous clocked sequential circuit

- Clocked circuits are stable and their behavior is easy to predict

Asynchronous Sequential Circuits

- Changes of state are not synchronized, and thus may occur at any time
- May have unpredictable behavior
    - Inputs arriving in a different order may result in different output states
    - The feedback loop may cause oscillations or brief glitches

Latches and Flip-Flops

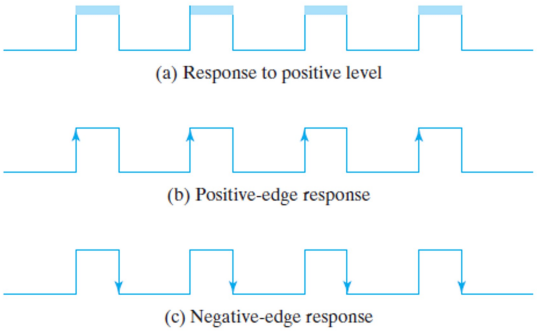
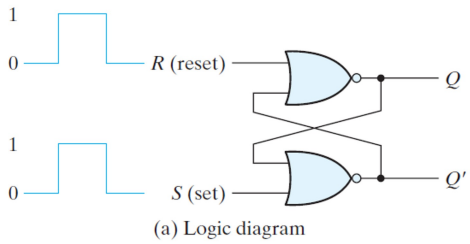
- Are circuits that store one bit of information
    - Also called *binary cells*
- Maintain their binary state as long as they have power
- Have two outputs: normal value and complemented value
- Are used as memory elements in clocked sequential circuits
- *Latches* are controlled by *signal levels*
- *Flip-flops* are controlled by *signal transitions*
    - i.e. The rising or falling edge of a clock pulse (CP)
    - Eg (Mano, p.197):
    - 

FIGURE 5.8
Clock response in latch and flip-flop

- Latches on their own are not useful in synchronous sequential circuits
    - However, they form an important component of flip-flops

SR Latch

- Can be built with 2 cross-coupled NOR gates
    - Has 2 inputs:
        - Set (S)
        - Reset (R)
    - Has 2 outputs:
        - Q - the normal output
        - Q' - the normal output primed
    - 

(a) Logic diagram

Default @ 0 V

FIGURE 5.3
***SR* latch with NOR gates**

- The latch is *set* by applying a *momentary* 1 to the S
    - Q goes back to 1, and stays at 1 when S goes back to 0
        - i.e. Stores a 1
    - Called the *set state*
- Is *reset* by applying a *momentary* 1 to R
    - Q goes back to 0, and stays at 0 when R goes back to 0
        - i.e. Stores at 0
    - Called the *clear state*

- Analysis
    - Note the truth table for the NOR gate:

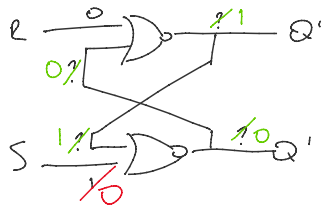
A	B	(A + B) '
0	0	1
0	1	0
1	0	0
1	1	0

- Apply set (S = 1, R = 0)

Diagram

- Remove set (S = 0, R = 0)

Diagram

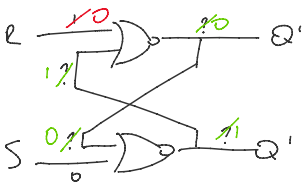


- Apply reset (S = 0, R = 1)

Diagram

- Remove reset (S = 0, R = 0)

Diagram



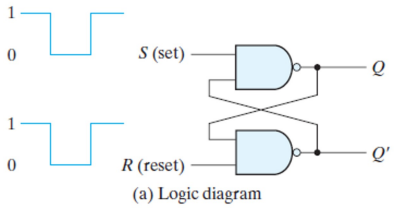
- The above is summarized in a *function table*:

S	R	Q	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	(forbidden)

(b) Function table

1) Apply set
2) Remove set
3) Apply reset
4) Remove reset
5) Undefined (avoid)

- Note that rows 2 and 4 have the same inputs, *but different outputs*
  - i.e. Depends on the previous state
- You should never apply 1 to R and S at the same time
  - Both Q and Q' will be 0, which is illogical
  - The state will be indeterminate when R and S go back to 0
    - Depends on which one is released first
    - Is said to be *metastable*
- Can also be made from cross-coupled NAND gates (called a S'R' latch)
  - R and S are normally at 1
  - Are set or reset by applying a momentary 0



(a) Logic diagram

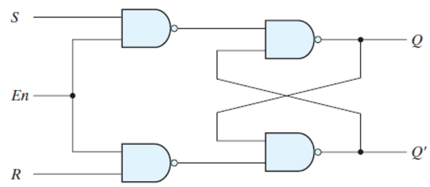
S	R	Q	Q'	
1	0	0	1	
1	1	0	1	(after $S = 1, R = 0$)
0	1	1	0	
1	1	1	0	(after $S = 0, R = 1$)
0	0	1	1	(forbidden)

(b) Function table

S is at the top,
R is at the bot

FIGURE 5.4
SR **latch with NAND gates**

- To control *when* the latch changes state, we add
  - A 3rd input called *enable*
  - 2 AND gates when using the SR latch
    - Or, 2 NAND gates with the S'R' latch

- 

  (a) Logic diagram

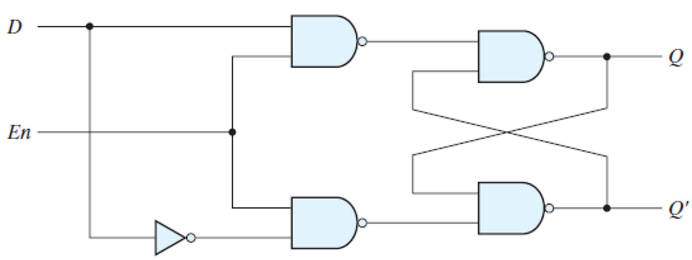
En	*S*	*R*	Next state of *Q*
0	X	X	No change
1	0	0	No change
1	0	1	$Q = 0$; reset state
1	1	0	$Q = 1$; set state
1	1	1	Indeterminate

  (b) Function table

  X is "don't care condition"

  FIGURE 5.5
  SR latch with control input

D Latch (Transparent Latch)

- Is a refinement of the SR latch
  - Has one input called D (Data)
    - Applied directly to S input
    - Its complement is applied to R input
  - An inverter ensures there are no indeterminate states
  - Has an *enable* state
- Is suitable for storing one bit of data
- Logic Diagram (Mano, p.195):
  - 

    (a) Logic diagram

En	*D*	Next state of *Q*
0	X	No change
1	0	$Q = 0$; reset state
1	1	$Q = 1$; set state

    (b) Function table

    FIGURE 5.6
    D latch

Latch Symbols

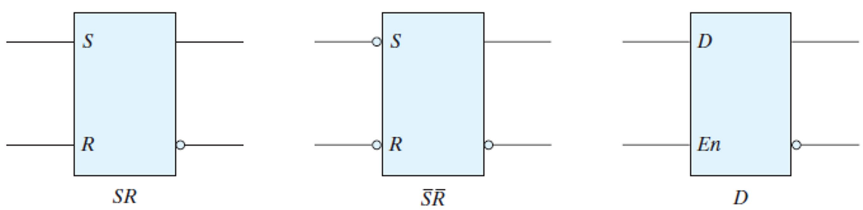
- Mano, p.196:
- 

  FIGURE 5.7
  Graphic symbols for latches

Readings

5.1 - 5.3

\\

Triggering of Flip-Flops

- The level-triggered latches seen so far have problems
  - May oscillate if the CP is too wide
  - Outputs will be incorrect if the inputs change before the CP returns to 0
- Solution: Trigger the device on the rising or falling *edge* of the CP.
  Master-slave and the *edge-triggered* flip-flops follow this approach

Master-Slave D Flip-Flops

- Is constructed with:
  - A *master* D latch
  - A *slave* D latch

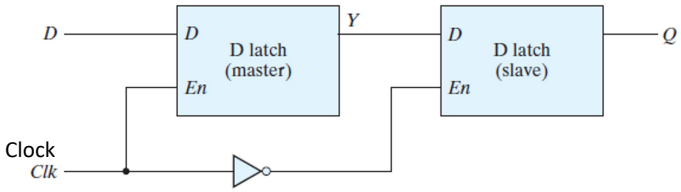
- An inverter
- 

FIGURE 5.9
Master–slave *D* flip-flop

- The inverter ensures only one latch is active at a time
  - When CP = 1, input D will produce an output on Y
    - However, Q (and Q') remain unaffected (CP = 0 for slave)
  - When CP = 0, new input to D is "locked out"
    - However, Y produces output on Q and Q' (CP = 1 for slave)
- Most master-slave flip-flops read input on the rising edge of CP, and produce new output on the falling edge
  - Can be reversed by adding another inverter
  - Allows flip-flops to be chained together
  - Even allows the output to be fed back into input

Edge-Triggered D Flip-Flop

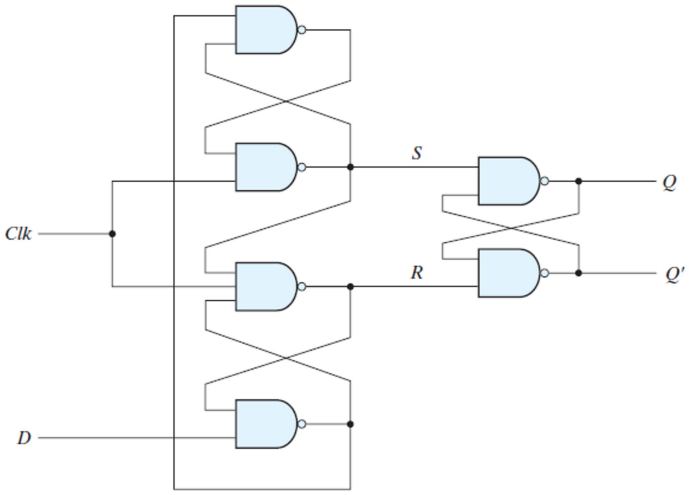
- Is triggered by the CP exceeding a threshold level
  - Inputs are then locked out until CP returns to original level
  - Can be positive- or negative-edge triggered
- 

FIGURE 5.10
D-type positive-edge-triggered flip-flop

- Consists of 3 S'R' latches
- Note that:
  - Q = 1 when S = 0 and R = 1
  - Q = 0 when S = 1 and R = 0
  - Q remains unchanged when S = R = 1
  - S = R = 0 is illegal
- Operation:
  - When CP = 0, and D = 0 or D = 1, then S = R = 1
    - *Missing diagram*
  - If D = 0 and CP goes to 1, then S = 1 and R = 0 (Q = 0)
    - A change of D to 1 has no effect while CP = 1
      - i.e. Output of gate 4 stays at 1
    - When CP goes to 0, S = R = 1 (Q remains at 0)
      - *Missing Diagram*
  - If D = 1 and CP goes to 1, then S = 0 and R = 1 (Q = 1)
    - A change of D to 0 has no effect while CP = 1
      - i.e. Output gate 1 stays at 1
    - When CP goes to 0, S = R = 1 (Q remains at 1)
      - *Missing Diagram*

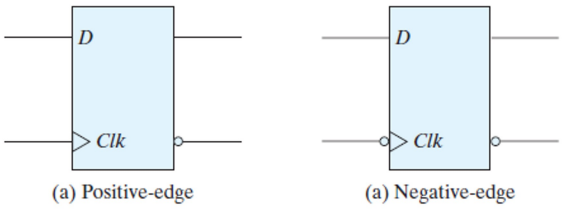
- D must be held constant for a minimum time before applying CP
  - Known as the *setup time*
  - Equal to propagation delay through gates 4 and 1
- D must be held constant for a minimum time after the CP transition
  - Called the *hold time*
  - Equal to propagation delay through gate 3
- Flip-Flop symbols
  - Arrow head indicates positive-edge triggering
  - Adding a circle indicates negative-edge triggering
  - 

FIGURE 5.11
Graphic symbol for edge-triggered *D* flip-flop

JK Flip-Flop

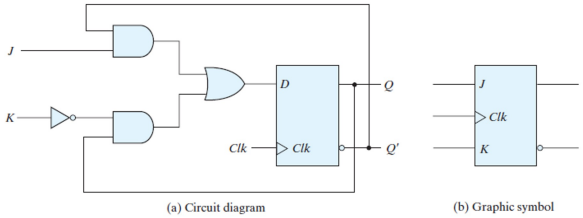
- Has 2 inputs (other than the clock input):
  - J: acts as a *set* input
  - K: acts as a *reset* input
  - When both J and K are asserted , the output is *toggled*
    - Avoids the indeterminate behavior seen in the SR latch
  - Can be created by adding 4 gates to the inputs of a D flip-flop
    - Eg (Mano, p. 201)
  - 

FIGURE 5.12
JK flip-flop

T (toggle) Flip-Flop

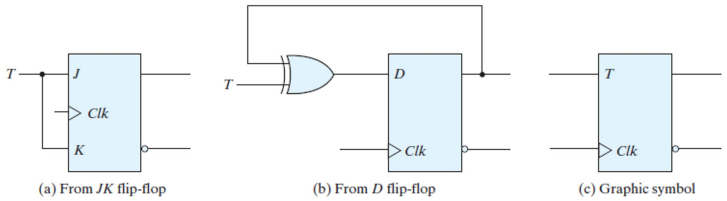
- Created by connecting the J and K inputs of a JK flip-flop together
  - This input is called T (Toggle)
- If T = 1 and a CP arrives, the outputs are complemented (toggled)
- Logic Diagram (Mano, p.201):
- 

FIGURE 5.13
T flip-flop

Characteristic Tables

- Show the operations of a flip-flops
  - Q(t) (often abbreviated Q) is the *present state*
    - i.e. The state before the application of a CP
  - Q(t+1) is the next state
    - i.e. The state after the CP

Q(t) |``` (change state) ```| Q (t+1)
>>>>>>----------TIME---------->>>>>>>>

- The column(s) on the left show all possible inputs into the flip-flop
- CP is not shown explicitly, but it is assumed to cause a change to the next state

- D flip-flop characteristic table:
  - **D Flip-Flop**

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

- T flip-flop
  - **T Flip-Flop**

T	Q(t + 1)	
0	$Q(t)$	No change
1	$Q'(t)$	Complement

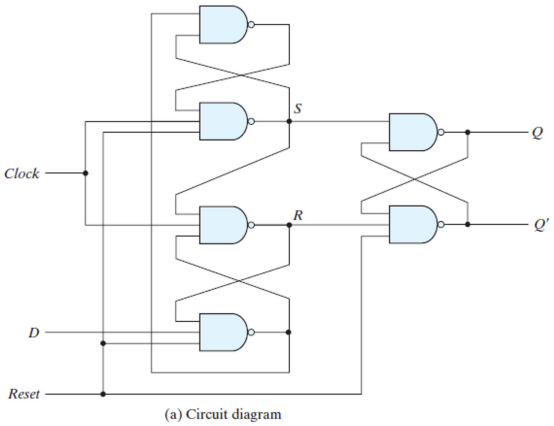
- JK flip-flop:
  - **JK Flip-Flop**

J	K	Q(t + 1)	
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	$Q'(t)$	Complement

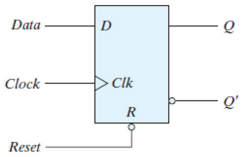
Characteristic Equations

- Express the information in characteristic tables algebraically
  - Can also be derived from the circuit diagram
- D flip-flop equation: Q(t+1) = D
- T flip-flop: Q(t+1) = T $\oplus$ Q = TQ' + T'Q ***MEMORIZE***
- JK flip-flop: Q(t+1) = JQ' + K'Q

Direct Inputs

- Allows one to set or clear a flip-flop *asynchronously*
  - i.e. Without a CP
  - Useful for initializing the state of flip-flops after powering up, before clocked operation
- *Preset*: sets the flip-flop
- *Direct reset*: clears the flip-flop
- Eg (Mano, p.204):
  - 

(a) Circuit diagram



(b) Graphic symbol

0
1
1

R	Clk	D	Q	Q'
0	X	X	0	1
0	↑	0	0	1
0	↑	1	1	0

(b) Function table

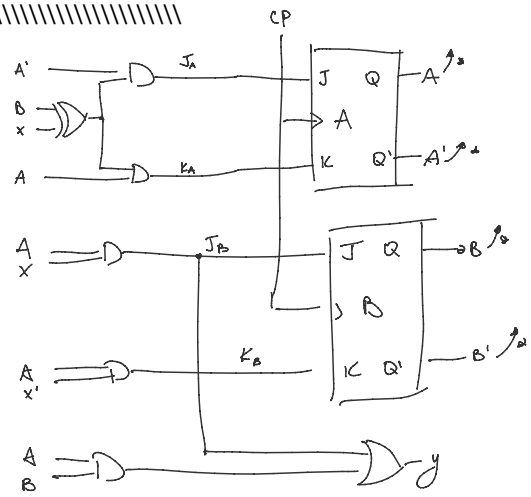
FIGURE 5.14
D flip-flop with asynchronous reset

Readings and Ex

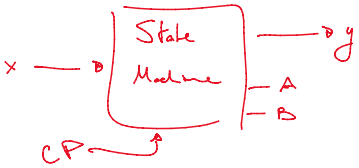
Mano sct. 5.4

Analysis of Clocked Sequential Circuits

- Determined from circuit inputs, circuit outputs, and the state of the flip-flops
  - Outputs and the next state depend on inputs and the present state
- Results in a *state table* and/or a *state diagram*
- Procedure:
  - Analyze the circuit to find the *circuit output functions* and *flip-flop input functions*
  - Create the state table
  - Derive the state diagram
  - Eg: Analyze the following:
    - Diagram (FEEDBACK PATHS ARE NOT SHOWN)



  - Note that the outputs of flip-flops serve as inputs to the combinational circuit
  - Input is x, and output is y
    - A and B (also A' and B') can be used as outputs, if desired
  - Block Diagram:
    - Diagram



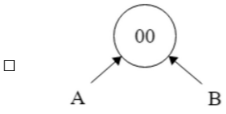
  - Find the circuit output function and flip-flop input functions
    - y = Ax + AB
    - $J_A = A'(B \oplus x)$
    - $K_A = A(B \oplus x)$
    - $J_B = Ax$
    - $K_B = Ax'$
  - Create the state table
    - Answers the question: What happens when the clock pulses are applied with different values of x?
    - Need to systematically analyze ABx from 000 to 111

Present State A	B	Input x	Next State A	B	Output y	Flip-flop Inputs J_A	K_A	J_B	K_B
0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	1	0	0	0
0	1	0	1	1	0	1	0	0	0
0	1	1	0	1	0	0	0	0	0
1	0	0	1	0	0	0	0	0	1
1	0	1	0	1	1	0	1	1	0
1	1	0	0	0	1	0	1	0	1
1	1	1	1	1	1	0	0	1	0

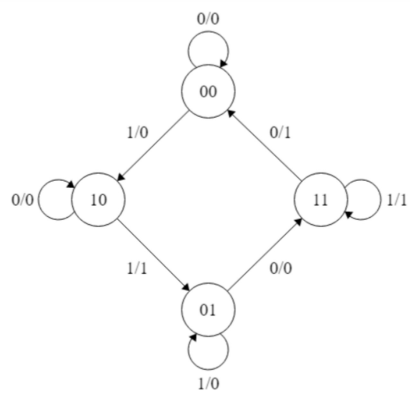
  - Find next state using present state, flip-flop inputs, and JK characteristic table
  - This table is often presented in second form to help derive the state diagram

Present State	Next State		Output	
AB	X=0	X=1	X=0	X=1
	AB	AB	Y	Y
00	00	10	0	0
01	11	01	0	0
10	10	01	0	1
11	00	11	1	1

  - Create the state diagram
    - Each state is represented with a circle, and is labeled with the value of the flip-flops (A and B in this example)



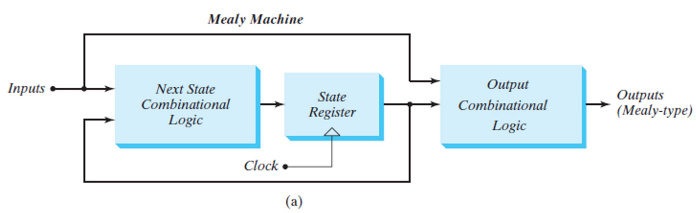
    - Transitions between states are represented with arrows labeled *input/output* (x/y in this example)



- A maximum of 4 states are possible with 2 flip-flops
- The *input* label on the arrow gives the value that causes a transition to another state
    - Some transitions loop back to the originating state
- The *output* label on each arrow indicates the value of the output when the input is applied
    - This is before a transition is made(i.e. before a CP)

Mealy and Moore Models

- Mealy Model: outputs are a function of present state and inputs
    - Eg: previous circuit
        - The output y depends on x and the present state of A and B
    - The outputs may change during a clock pulse period, since inputs are not synchronized with the CP
        - Outputs should only be sampled on a CP transition
- Moore Model: outputs are a function of the present state only
    - i.e. Outputs (possibly with some additional gates) are connected to flip-flops outputs *only.*
    - Outputs are synchronized to the CP, since the flip-flops are triggered by the CP
- Block Diagrams (Mano, p.216)



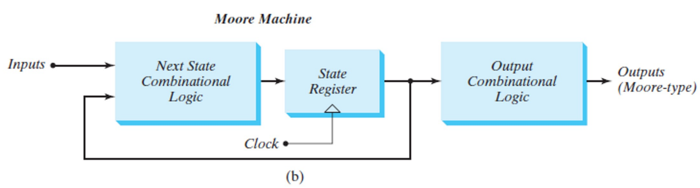


FIGURE 5.21
Block diagrams of Mealy and Moore state machines

Readings and Ex

- Mano sct. 5.5

\\

Flip-Flop Excitation Tables

- Are more convenient than characteristic tables for sequential circuit synthesis
- Show the input needed to go from state Q(t) to Q(t+1)
    - Note: X means "don't care" (0 or 1)
- D Flip-flop:

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

1	0	0
1	1	1

- T Flip-flop:

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

- JK flip-flop:

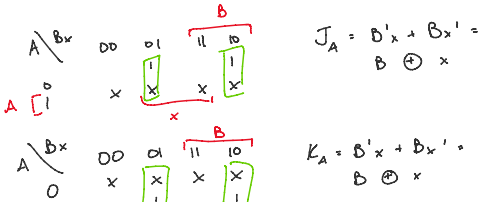
Q(t)	Q(t+1)	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

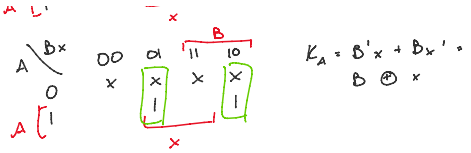
Synthesis of Clocked Sequential Circuits

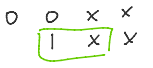
- Goal is a logic diagram, given a state diagram
- Procedure:
  1. Determine inputs, outputs, number of flip-flops needed, type of flip-flops
  2. Derive the excitation table for the state machine
  3. Derive the circuit output functions and flip-flop input functions, using the map method
  4. Draw the logic diagram
- Eg: synthesize a clocked sequential circuit for the given state machine using JK flip-flops
  - State Diagram 1.0
- We need:
  - One input x
  - One input y
  - 2 flip-flops, A and B, to represent 4 states
  - JK flip-flops are specified
- Derive the state machine excitation table

Combinational Circuit Input					Combinational Circuit Output				
Present State		Input	Next State		Flip-flop Input				
A	B	X	A	B	J_A	K_A	J_B	K_B	y
0	0	0	0	0	0	x	0	x	0
0	0	1	1	0	1	x	0	x	0
0	1	0	1	1	1	x	x	0	0
0	1	1	0	1	0	x	x	0	0
1	0	0	1	0	x	0	0	x	0
1	0	1	0	1	x	1	1	x	1
1	1	0	0	0	x	1	x	1	1
1	1	1	1	1	x	0	x	0	1

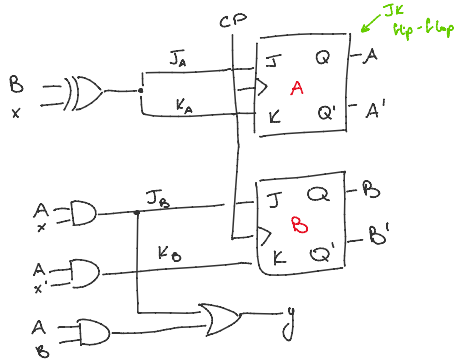
- Find next state and output y from state diagram
- Use flip-flop excitation table to find flip-flop input
- Use maps to find equations for flip-flop inputs and y







- Draw the logic diagram
  - Share terms where possible
  - Note this is simpler than originally seen in the analysis



Readings and Ex

Read Mano sct. 5.8

//

Counters

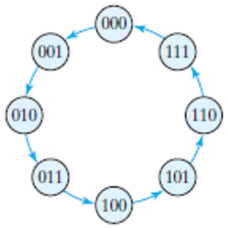
- Always step from one state to the next on application of a CP
- An n-bit *binary counter*:
  - Consists of n flip-flops
  - Counts in binary from 0 to 2^n-1, then "wraps around" to 0
- Eg: 3-bit counter
  - State diagram:
  - 

    FIGURE 5.32
    State diagram of three-bit binary counter
  - Note the only input is CP
    - The next state depends only on the present state
  - Outputs come from the flip-flop outputs
- Binary counters are easily made with T flip-flops
  - State table:

- Table 5.14
  State Table for Three-Bit Counter

Present State			Next State			Flip-Flop Inputs		
A_2	A_1	A_0	A_2	A_1	A_0	T_{A2}	T_{A1}	T_{A0}
0	0	0	0	0	1	0	0	1
0	0	1	0	1	0	0	1	1
0	1	0	0	1	1	0	0	1
0	1	1	1	0	0	1	1	1
1	0	0	1	0	1	0	0	1
1	0	1	1	1	0	0	1	1
1	1	0	1	1	1	0	1	1
1	1	1	0	0	0	1	1	1

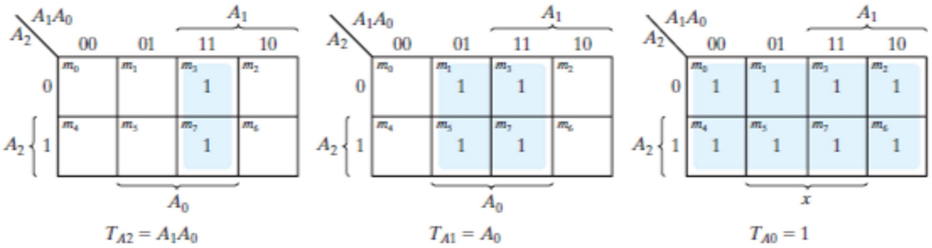
- 
  $T_{A2} = A_1A_0$ $T_{A1} = A_0$ $T_{A0} = 1$

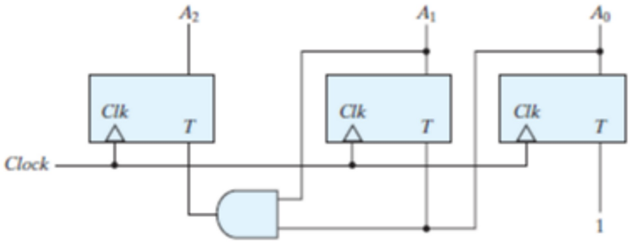
  FIGURE 5.33
  Maps for three-bit binary counter
- Logic Diagram (Mano, p. 245)
- 

  FIGURE 5.34
  Logic diagram of three-bit binary counter

Readings and Ex

Mano sct. 5.8 and 6.4

//

Register

- A *register* is a group of binary cells
  - Each cell stores one bit of information
- An *n*-bit register
  - Has *n*- flip-flops, each storing 1 bit
    - Usually D type
  - May also have additional gates to control input and output, or to clear
  - Eg (Mano, p.256):

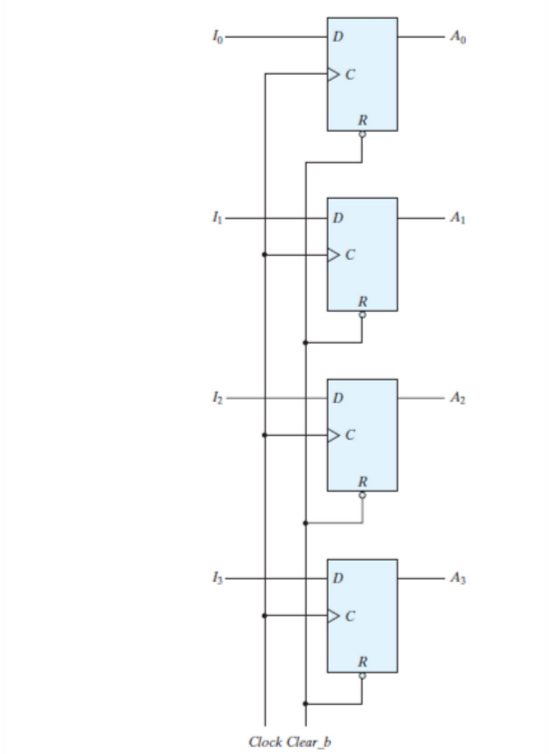
- 

FIGURE 6.1
Four-bit register

Register with Parallel Load

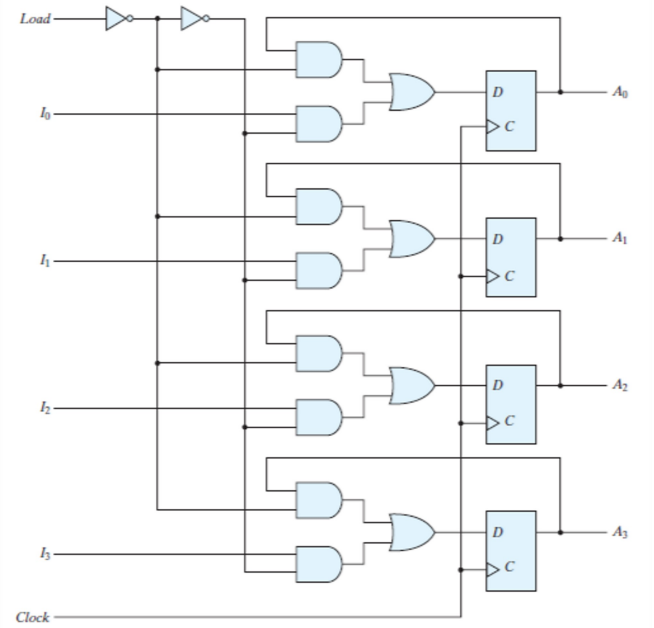
- All bits are loaded simultaneously when CP is applied
- Problem: since the master clock supplies *continuous* pulses, load will occur at every CP unless the circuit is altered
- Potential solution: put an AND gate on CP input
  - When load control is 1, CP will reach the flip-flops:
  - CP and Load Diagram 1.0 (check workbook)
    - Propagation delay: 5-10 nanoseconds
  - Adding gates to CP introduces propagation delays
    - May throw the system out of sync
- Better solution: add load control circuitry to the flip-flop data inputs
- Eg (Mano, p.258):
- 

FIGURE 6.2
Four-bit register with parallel load
- When Load = 0, A outputs are fed back into D inputs
  - i.e. Flip-flops maintain current content

- When Load = 1, I inputs are loaded into the flip-flops

Shift Registers

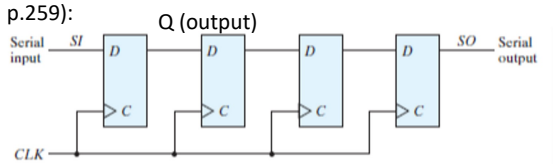
- Shift bits either left or right
- Consists of cascaded flip-flops
  - Output of one flip-flop is connected to the input of the next
- Eg(Mano, p.259):
  - 

FIGURE 6.3
Four-bit shift register
  - Shift occurs on every CP
  - Bits are shifted in on SI(serial input), and out on SO (serial output)

Serial Transfer

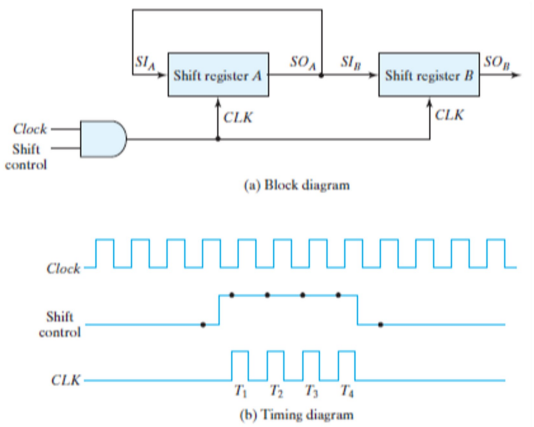
- Where information is moved or manipulated one bit at a time
- Eg(Mano, p.260):
- 

FIGURE 6.4
Serial transfer from register *A* to register *B*
  - When Shift control = 1, CP passes to registers A & B
    - Contents of A are transferred to B
    - Shift control must equal 4 clock pulses in length to transfer 4 bits
  - Initial contents of register B are shifted out
    - Lost, unless connected to another shift register
  - Feedback path ensures register A retains original contents
- Computers can operate in:
  - Serial mode
    - Slow, since 1 bit at a time
    - Cheap, since less hardware required
  - Or parallel mode
    - Fast, since 1 word at a time
    - Expensive, since more hardware required
  - Or both

Universal Shift Register

- Support serial I/O
  - Can shift left or right
- Supports parallel I/O
  - Can do parallel load
  - Has parallel outputs
- Has *mode control* inputs to select one of the above
  - Eg (Mano, p.266):

- Table 6.3
  Function Table for the Register of Fig. 6.7

Mode Control		
s_1	s_0	Register Operation
0	0	No change
0	1	Shift right
1	0	Shift left
1	1	Parallel load

- Logic diagram (Mano, p.265):

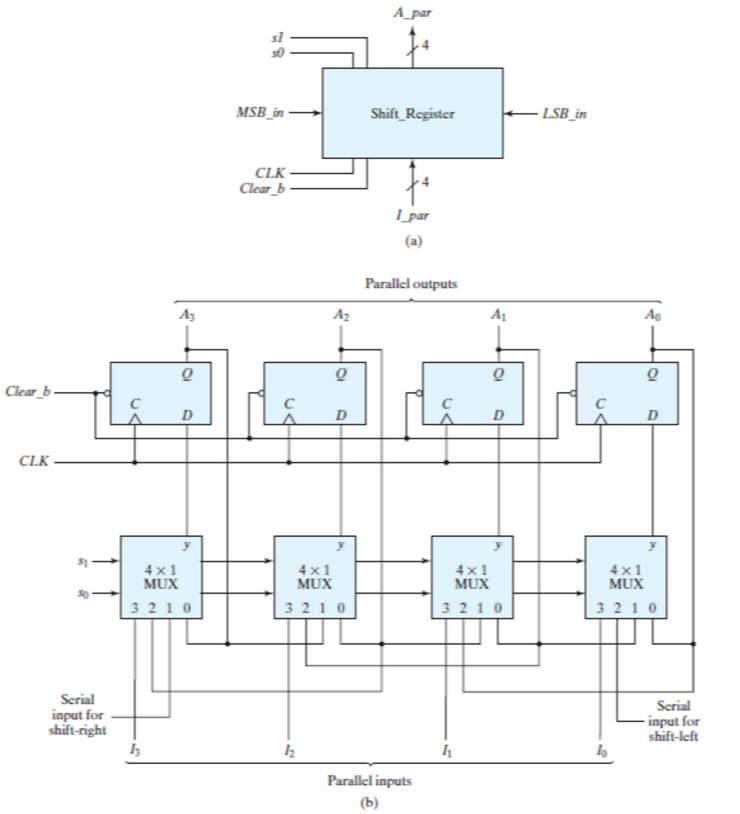


FIGURE 6.7
Four-bit universal shift register

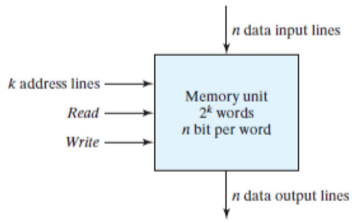
- MUX (Multiplexer)
- S_0 and S_1 are the mode control settings
- The mode control uses the bottom of the MUX

Readings and Ex

Read Mano sct. 6.1 and 6.2

//

Random Access Memory

- Is an IC that stores *m* words, each *n* bits in length
  - Thus *m* x *n* bits total
- Has *n* data input and output lines
  - i.e. One per bit in a word
- Requires $\log_2 m$ address selection lines
  - If *k* address lines, stores 2^k words
- Has *read* and *write* control inputs
- 

  2^k x n

FIGURE 7.2
Block diagram of a memory unit

- Binary data on the address lines selects a particular word
  - Addresses range from 0 to $2^k - 1$

- Eg: 256 x 8 memory

Address		
Decimal	Binary	Contents
0	0000 0000	1001 1100
1	0000 0001	0110 1101
2	0000 0010	1000 1011
...	...	...
255	1111 1111	0111 0101

- To write to memory:
  1. Place the binary address on the address lines
  2. Place data on the data input lines
  3. Activate the write control input

- To read from memory:
  1. Place the binary address on the address lines
  2. Activate the read control input

- Many RAM chips have:
  - A single read/write (apply a bar later) control line
  - A memory enable line
    - Sometimes called *chip select* (CS)
  - Operation:

Read/Write	Memory Enable	Memory Enable
X	0	None
0	1	Write
1	1	Read

- Many RAM chips have common terminals for data I/O
  - i.e. Data lines are *bidirectional*
  - Reduces the number of external pins

Types of Memory

- RAM can be *static* or *dynamic*
  - Static (SRAM)
    - Bits are stored in flip-flops
    - Easy to use
    - Short read/write cycles
    - High power consumption
    - Small storage capacity
  - Dynamic (DRAM)
    - Bits are stored in capacitors
      - Tend to discharge over time, so must be continually *refreshed*
    - More complicated to use
    - Longer read/write cycles
    - Lower power consumption
    - Larger storage capacity
- *Volatile* memory loses stored information when the power is turned off
  - Eg: SRAM, DRAM
- *Nonvolatile* memory retains data when powered off
  - Eg: ROM, disk, tape, flash

Binary Cell

- Logic Diagram (Mano, p.307):

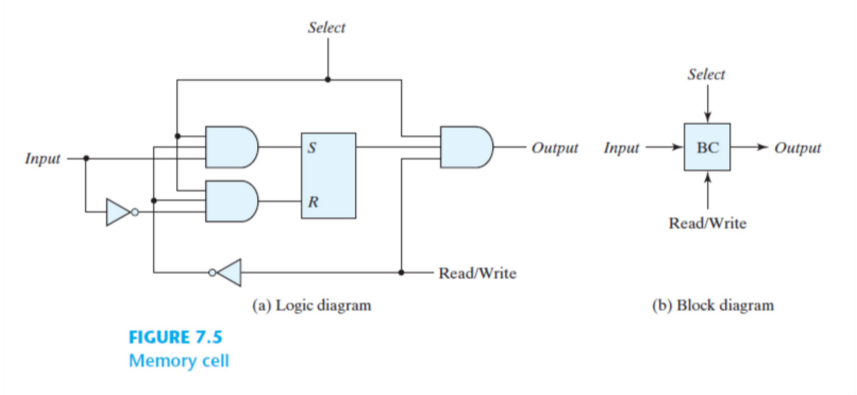
- 

FIGURE 7.5
Memory cell
    - Select: enables cell for reading or writing
    - Read/write: determines operation when cell selected
        - 0 = write, 1 = read
- The flip-flop operates without a CP input
    - Similar to an SR latch
    - The above is a logical representation
        - Actually constructed more simply with 2 multiple-input transistors

RAM Construction

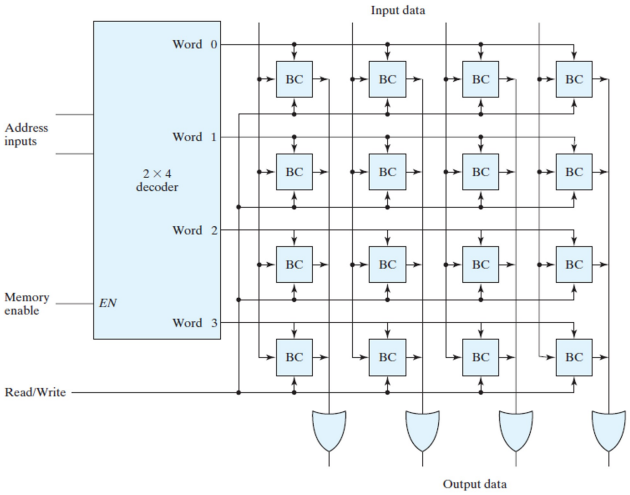
- A 2^k x n RAM has:
    - k address lines
    - k x 2^k decoder
    - 2^k x n binary cells
    - n OR gates (with 2^k inputs)
    - n output lines
- Eg (Mano, p.308)
- 

FIGURE 7.6
Diagram of a 4×4 RAM
    - If *memory enable* = 0, all outputs of the decoder are 0
        - i.e. No memory word is selected
    - If 1, one word is selected for read or write

Readings and Ex

Mano Sections 7.1 - 7.3

Raspberry Pi 3

January 10, 2019 10:29 PM

- The Raspberry Pi is an inexpensive computer housed on a credit-card-sized board
  - Pi 1 Model B first sold in 2012
    - Uses the Broadcom BCM2835 System on Chip (SoC)
      - Single ARM11 CPU running at 700 MHz
        - ARMv6 instruction set
      - Broadcom VideoCore IV Graphics Processing Unit (GPU)
      - 512 MB RAM (shared between CPU and GPU)
    - 10/100 Mbit/s Ethernet port
    - 2 (later 4) USB ports
  - Pi 2 introduced in 2015
    - Uses the BCM2835 SoC
      - Four Cortex-A7 CPU cores running at 900 MHz
        - ARMv7 instruction set
      - 1 GB RAM
  - Pi 3 introduced in 2016
    - Uses the BCM2837 SoC
      - Four Cortex-A53 CPU cores running at 1.2 GHz
        - ARMv8 instruction set (in AArch64)
    - Added wireless networking and Bluetooth
  - Pi 3+ introduced in 2018
    - Uses the BCM2837B0
      - Now 1.4 GHz
    - 10/100/1000 Mbit/s Ethernet
    - Improved wireless performance

Introduction

- The Pi can run many different operating systems:
  - Normally *Raspian* (a version of Debian Linux)
  - Many variants of Linux
  - Windows 10 IoT Core
  - OS downloads available at:
  - https://www.raspberrypi.org/downloads/
- In this course, we will work *without* an OS!
  - i.e. We will do "bare metal" programming
  - Allows us to explore concepts such as:
    - Memory-mapped I/O
    - General purpose I/O (GPIO)
    - Interrupts and interrupt-driven I/O
    - Cross compilation software development
  - These are critical to understanding how to program for *embedded computer systems*

Raspberry Pi Kit

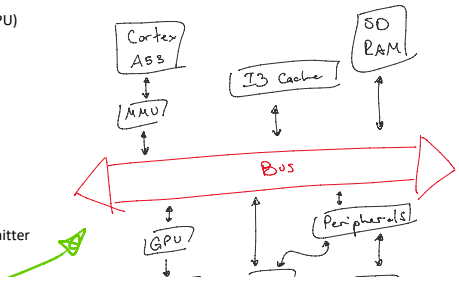
- Ict 602 mon1-4:30 tue/thursd 1-3:30 natasha
- Contains:
  - Raspberry Pi 3+ Model B
  - Plastic Case
  - AC power supply
    - Powers the Pi using a USB micro connector
  - Ribbon cable (40-pin to 26-pin) for GPIO
  - Micro SD card
    - Preloaded with NOOBS
      - Allows you to install Raspbian or LibreELEC
  - USB SD Card Reader
    - Used on a host computer to load an image onto the SD card

Pi 3+ I/O Ports

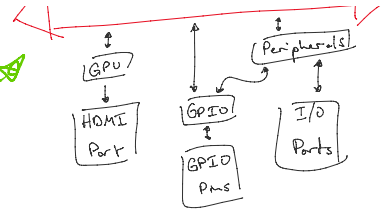
- Includes:
  - 4 USB ports
    - Can attach a keyboard and a mouse
  - Micro SD card slot
    - The SD hold the file system (secondary memory)
      - i.e. No hard drive
  - Ethernet port
  - Audio jack
    - For headphones or speakers
  - HDMI port
    - Connects to a monitor
      - Also supports audio input
  - Micro USB power connector
    - Plugged in last to provide power
      - No on/off switch
  - 40 GPIO pins
    - Can connect LEDs and switches and other electronic components

Broadcom BCM2837 SoC

- Integrates the following components:
  - ARM Cortex-A53 Processor
    - Contains 4 CPU cores
  - Broadcom VideoCore IV Graphics Processing Unit (GPU)
    - Clocked at 400 MHz
  - 1 GB of Synchronous Dynamic RAM (SDRAM)
  - Cache memory
  - Memory Management Unit (MMU)
  - Peripherals
    - Timers
    - Interrupt Controller
    - GPIO
    - USB
    - 2 UARTs
      - Universal Asynchronous Receiver Transmitter
    - 3 SPI masters, 1 SPI slave
      - Serial Peripheral Interface



        - USB
        - 2 UARTs
            - Universal Asynchronous Receiver Transmitter
        - 3 SPI masters, 1 SPI slave
            - Serial Peripheral Interface
        - Others
- Diagram

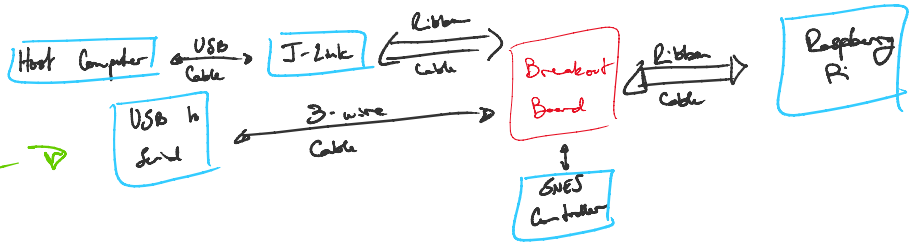


ARM Cortex-A53 Processor

- Contains:
    - 4 ARMv8-A cores. Each has:
        - ARMv8 CPU
        - Level 1 instruction and data caches
        - Floating Point Unit (FPU)
        - Memory Management Unit (MMU)
    - Level 2 cache (shared with all cores)
    - Bus interface
- Check D2L for document
    - Figure 2.2

Lab Environment

- Each workstation consists of:
    - Host computer
        - Runs Linux (Fedora 28)
        - Installed with cross-compiler gcc toolchain (Linaro 7.3.1)
        - Installed with the Qemu emulator 3.0
            - Can emulate the Pi 3
    - Extra monitor (HDMI), keyboard and mouse (USB) for the Pi
    - J-Link JTAG debugging interface
        - Does not work with the Pi 3, so not used in this course
    - USB-to-serial interface
        - Allows the host to act as a terminal for the Pi
    - Super Nintendo Entertainment System (SNES) game controller
    - Breakout Board
        - Facilitates connections between the Pi and all the above components
- Diagram



Raspberry Pi Boot Sequence

- Initial state: power disconnected and micro SD card in place
    - Must have the following boot files installed:
        - bootcode.bin
        - start.elf
        - fixup.dat
    - These are available on D2L
        - Or at: https://github.com/raspberrypi/firmware/tree/master/boot
    - Must also have a kernel image file installed
        - Normally called *kernel8.img*
        - This can be a pre-existing OS image file, or a file that you create (a bare metal program)
- Boot sequence:
    1. Apply power
    2. The GPU turns on
        i. The CPU cores are off
        ii. SDRAM is disabled
    3. The GPU executes the *first stage bootloader*
        i. This is a program installed in read-only memory (ROM)
    4. The first stage bootloader loads the *second stage bootloader* program *bootcode.bin* into L2 cache from the SD card, and starts executing it on the GPU
    5. The second stage bootloader enables SDRAM, and then loads the *start.elf* file from the SD card into SDRAM, and starts executing it on the GPU
        i. This is the GPU's *firmware*
    6. The GPU firmware powers up the cores, and loads the kernel image into SDRAM from the *kernel8.img* file on the SD card
        i. The *fixup.dat file* is read to configure the partitioning of memory between the CPUs and GPU
        ii. If there is *config.txt* file on the SD card, it will be read to supply system configuration information (optional)
    7. The reset signal on the ARM cores is released, and the kernel code starts executing on the CPUs (all 4)
        i. Since the firmware read from a *kernel8.img* file, the CPUs start in the AArch64 execution state

Preparing the Micro SD Card

- The supplied SD card is preloaded with the Raspian OS
- Recommended to buy an additional card for your work in this class
    - 16 GB micro SD card: $13.99 at London Drugs (lol)
- The card must be formatted with the FAT32 (file allocation table) file system
    - Most new cards are already formatted this way
- If necessary, you can format the card using any computer with an SD card reader
    - Note: this erases the contents of the card!
- To format on the lab host machines:
    1. Start the *Disks* application (under Accessories)
    2. Plug the SD card reader into a USB port
    3. Slide the SD card into the card reader
    4. In the *Disks* application, find the 16 GB Drive (Generic Storage Device)
    5. Click on the ≡ button, then "Format Disk..."
        i. Choose Partitioning: "Compatible with all systems and devices (MRB/DOS)"
        ii. Click "Format..."
            1) Results in a volume with no partitions
    6. Click the + button (create partition)
        i. Choose the following options:
            1) Erase: Don't overwrite
            2) Type: Compatible with all systems and devices (FAT)
            3) Name: <choose a name for your disk>
    7. Click the "gears" button
        i. Click "Edit Partition"
        ii. Change the type to "W95 FAT32"
        iii. Make sure "Bootable" is checked
- When done, the partition type should be: W95 FAT32 (Bootable)
- Make sure you always "eject" the card (by clicking on the eject button in the GUI) before removing the card from the reader

- Once formatted, copy the following files onto the card (can be done by dragging and dropping in the Linux GUI):
  - bootcode.bin
  - start.elf
  - fixup.dat

Cross Compilation Process

- A *cross compiler* creates an executable the runs on a *target* architecture different from the development computer
- The Linaro 7.3.1 cross compiler toolchain is installed on the lab machines
  - Are Intel CPUs (x86 64-bit architecture)
  - The compiler produces machine code that runs on the ARMv8 architecture (AArch64)
- Installation directory:
  - /usr/local/linaro/gcc-linaro-7.3.1-2018.05-x86_64_aarch64-elf/bin/
- Contains the following tools:
  - Compiler: aarch64-elf-gcc
  - Assembler: aarch64-elf-as
  - Linker: aarch64-elf-ld
  - Object copy: aarch64-elf-objcopy
    - Translates an .elf executable into an .img raw binary image
  - Object dump: aarch64-elf-objdump
    - Creates a text file that displays information about an object file (usually a .elf file)
    - Useful when debugging
- General procedure:
  - Use gcc to compile all .c files (C source code) into corresponding .o files (object code)
  - Use *as* to assemble all .s files (A64 assembly source code) into .o files (object code)
  - Use *ld* to link all .o files into a kernel8.elf binary
    - ELF: Executable and Linkable Format
  - Use *objcopy* to translate kernel8.elf into kernel8.img
    - The Pi boots using this raw binary image
- A *Makefile* is provided on D2L to automate the build process
  - Type 'make' or 'make all' at the command line to create the kernel8.img file from source code files
  - Uses a link script (*link.ld*) to guide the linking process
  - Type 'make run' to execute the kernel8.img in the Qemu emulator
    - Flags are set to emulate the Pi 3
    - Not all features are emulated
  - Type 'make clean' to remove al intermediate files
- Once created, you copy the kernel8.img file to your SD card
  - Can be done by dragging and dropping in the Linux GUI
- "Eject" the SD card from the host machine, and put it off into the Pi (with the power off)
- Boot the Pi by plugging in the USB power cable

Bare Metal Startup Code

- The file *start.s* is provided to initialize the Pi and prepare an environment in which a main() routine can execute
- Is written in A64 assembly
  - New instructions:
    - mrs: move contents of a system register into an x register
    - wfe: wait for event
    - cbz: conditional branch if register is zero
    - cbnz: conditional branch if register not zero
- Does the following:
  1. Reads the *multiprocessor affinity register* (mpidr_el1) and tests on which core the code is running
     i. If core 0, skip forward to the next step
        1) Else, put cores 1-3 into an infinite loop
           a) Are not used to run our code
  2. Initialize the stack pointer (SP) register
     i. Set to point to free memory, so that main() and other functions can push and pop back frames
  3. All bytes in the .bss (block starting symbol) section are set to 0
     i. The address and size of this section are provided by the linker
     ii. Is done by writing 8 bytes of zeros to RAM in a loop
  4. Branch to the main() routine
     i. Can be written in C or assembly
     ii. main() should not return (ends with an infinite loop)
        1) However, if it does we put the core into an infinite loop
- Code for start.s (code in D2L w/ *example code*)

UART Terminal

- The files *uart.c* and *uart.h* are provided so that the Pi can communicate with the host computer
  - The host acts as terminal to the Pi
    - The Pi can output text to the screen
    - The Pi can receive text from the keyboard
  - Is useful when debugging
- Your code should first call uart_init() to setup the UART
  - 8-bit mode, 115200 Baud rate
- Your code can then output text using:
  - uart_putc(): for a single character
  - uart_puts(): for a string
  - uart_puthex(): for a 32-bit integer
- And receive text using:
  - uart_getc(): gets a single character
- Example (*main.c* on D2L):
- The host is physically connected to the Pi using the USB-to-serial cable
  - The cable is plugged into a USB port on the host
  - The 3 wires on the other end are connected to the UART pins on the breakout board:
    - GND (ground): black wire
    - RXD (received data): red wire
    - TXD (transmit data): white wire
  - The ribbon cable connects to the corresponding pins on the Pi:
    - GND to GND
    - TXD to pin 14
    - RXD to pin 15
  - Note: NEVER connect the 5V pin UART pin!
    - Power is already supplied to the Pi
- A terminal emulation program must be run on the host computer
  - *Minicom* is already installed
- To run Minicom:
  1. Open a terminal window
  2. Start Minicom by typing:

MIDTERM INFO

30 mc (20-30 min) 50%
Recall of facts
Boolean alg
Manipulation

1 long answer design (30 min) 50%
Design combinational logic
Not synch. sequential

Up to and including SYNCHRONOUS SEQ LOGIC

//////////////////

Design study assignment 1
For the long answer
Everything from the notes

i. Minicom -b 115200 -D /dev/ttyUSB0
    1) Baud rate is 115200
    2) The connection to the Pi device is represented with ttyUSB0
ii. Make sure that Minicom is set to 8N1 (8-bit data, odd parity, 1 stop bit) and that hardware flow control is set to 'N'
    1) Use control-a z to for a help menu
        a) Configure Minicom

3. Communicate with the Pi by typing characters
    i. Any text sent by the Pi appears on the console
4. To quit Minicom, type: control-a x

Readings and Ex

- *Programmer's Guide for ARMv8-A*
    - Sections 4.3, 14.1.1
- *ARMv8 Instruction Set Overview*
    - Sections 5.9.2, 5.2.1
- Browse *BCM2837 ARM Peripherals*
- Browse *Minicom* man page
    - On host machine terminal, type: man minicom

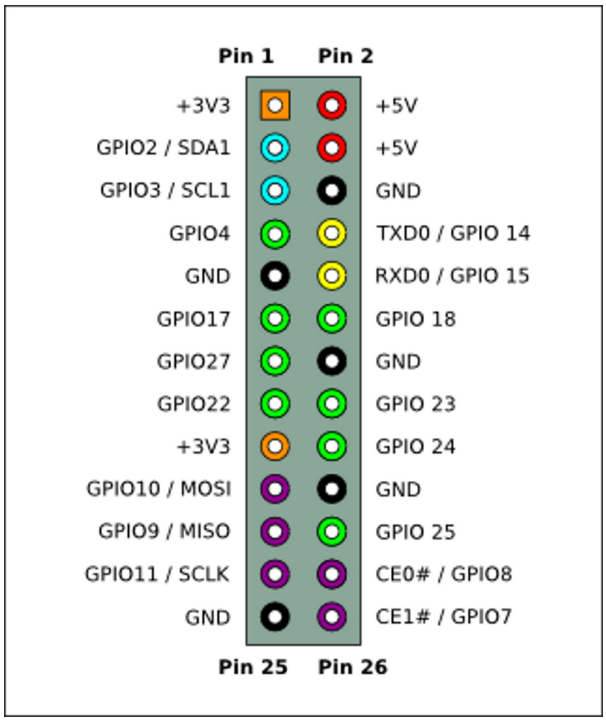
General Purpose Input and Output (GPIO)

January 10, 2019 10:29 PM

GPIO on the Pi 3

- Provides a flexible means to connect to a variety of I/O devices. For example:
  - Sensors
    - Eg: light, temperature, pressure, etc.
  - Controls
    - Eg: switches, rotary encoders, joysticks, etc.
  - Actuators
    - Eg: stepper motors, rotary valves, etc.
  - Displays
    - Eg: light emitting diodes (LEDs), liquid crystal displays (LCDs), etc.
  - Custom circuits
    - Often prototyped on a "breadboard"
- GPIO is not "plug and play"
  - Requires low-level programming

GPIO Pins

- The BCM2837 SoC provides 54 GPIO lines
- The Pi 3 routes 26 of these to the 40-pin header
- Of these, only 17 are available on the breakout board
  - Other pins provide power
    - GND
    - +3.3V
    - +5V
- 

- Note: GPIO pins 2 and 3 are labelled 0 and 1 on the breakout board
  - For an earlier version of the Pi

- Many of the pins can be used by SoC peripherals
  - i.e. have a dual use
  - Eg: GPIO pins 14 can be used for the UART TXD line

Memory Mapped Registers

- The GPIO on the SoC is programmed by reading or writing *GPIO registers*
  - These registers appear to the programmer as 32-bit-wide memory locations
  - One writes to memory (the GPIO register) using an *str* instruction
    - And reads using an *ldr* instruction
  - In C, one can use pointers to read/write registers
- The addresses for the GPIO registers are documented in the BMC2837 manual
  - These are *bus addresses*
  - Since the MMU maps *physical addresses* to the bus addresses, one must replace the 7E in the manual with 3F in your program to specify the correct physical address
    - Eg: The GPFSEL0 register has a bus address of 0x7E200000
      - Use 0x3F200000 in your program
- Assembly example

```
ldr x19, =0x3F200000
ldr w20, [x19] //read GPIO reg

mov w20, 0x7
str w20, [x19] //write GPIO reg
```

- In C, one can use the pointers defined in the gpio.h header file to read or write
  - Available on D2L

```
#include "gpio.h"
unsigned int value;

//Read GPFSEL0 register
value = *GPFSEL0;

//Write GPFSEL register
value = 0x7;
*GPFSEL = value;
```

GPIO Registers

- There are 41 GPIO registers
  - GPFSEL0 - GPFSEL5 are used to configure a particular GPIO pin to one of the following:
    - Input
      - Can detect a high or a low voltage level (or a rising or falling edge)
    - Output
      - Can produce a high or low output voltage level
    - One of 6 "alternative functions"
      - Allows a SoC peripheral to use the pin for I/O
  - GPFSET0/GPSET1 sets a particular GPIO output pin to a high value (3.3V)
  - GPCLR0/GPCLR1 clears a particular GPIO output pin to a low value (0V)
  - GPLEV0/GPLEV1 reads the level of a particular GPIO input pin
    - Low (0) is 0V
    - High (1) is 3.3V
  - GPPUD AND GPPUDCLK0/1 are used to enable/disable pullup/pulldown resistors for a particular pin
  - There are several registers to configure and detect interrupts

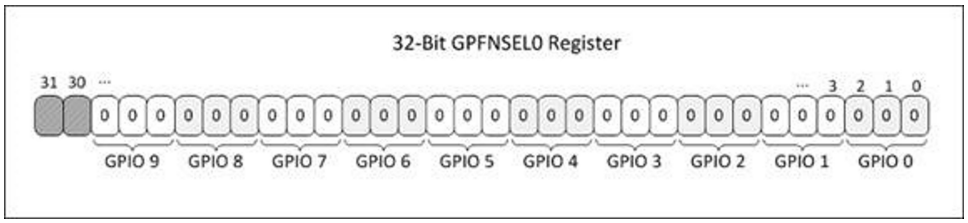
- Covered later

Function Select Registers

- Are the GPFSEL0-GPFSEL5 (function select) registers
- Each can configure a subset of the 54 GPIO lines:

Register	Pins
GPFSEL0	0-9
GPFSEL1	10-19
GPFSEL2	20-29
GPFSEL3	30-39
GPFSEL4	40-49
GPFSEL5	50-53

- Each pin is configured with a 3-bit field within these registers
  - Eg: GPFSEL0



  - Use one these bit patterns to select the function for a pin:

Bit pattern	• Function
000	Input
001	Output
100	Alt 0
101	Alt 1
110	Alt 2
111	Alt 3
011	Alt 4
010	Alt 5

- Eg: set pin 7 to Alt 2

```
*GPFSEL0 &= ~(0x7 << 21); //Clear bits 21-23 to 000
*GPFSEL0 |= (0x6 << 21); //Set bits 21-23 to 110
```

- Eg: set pin 3 to output

```
*GPFSEL0 &= ~(0x7 << 9); //Clear bits 9-11 to 000
*GPFSEL0 |= (0x1 << 9); //Set bits 9-11 to 001
```

GPIO Output

- Must first configure the desired pin to be an output pin
- To set the pin to a high value (3.3V), set the corresponding bit in the GPSET0/GPSET1 register to 1

- Eg: Set pin 3 high

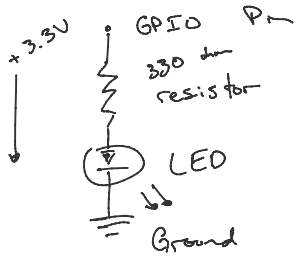
```
*GPSET0 = (0x1 << 3);
```

- Note: the 0's in the bit mask do not clear do not clear the other pins (they retain their current values)

- To set the output pin to a low value (0V), set the corresponding bit in the GPCLR0/GPCLR1 register to 1
  - Eg: Set pin 3 low

```
*GPCLR0 = (0x1 << 3);
```

  - Note: the 0's in the bit mask do not set the other pins (they retain their current values)
- We can use the GPIO pin to turn an LED on and off
  - Use the following circuit:



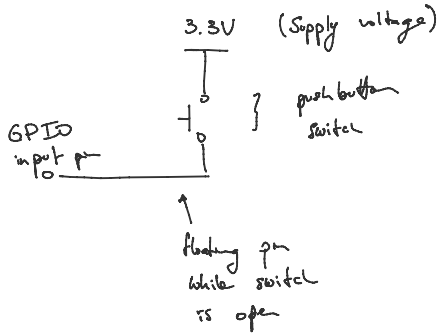
  - Note: be sure to use the 330 ohm current-limiting resistor
    - Otherwise the LED will burn out and your Pi may be damaged

GPIO Input

- Must first configure the desired pin to be an input pin
- Since we will use external pull-up or pull-down resistors, we must disable the internal pull up/down circuitry
  - Described later
- To read the pin's input value, read the corresponding bit in the GPLEV0/1 register
  - Eg: read the level on pin 3

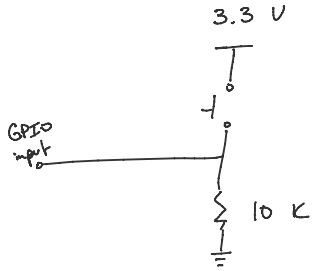
```
unsigned int r, level;

//Read the GPIO register
r = *GPLEV0;

//Isolate bit 3
//0 means 0V, 1 means 3.3V
level = (r >> 3) & 0x1;
```

- We can use a pushbutton switch to control the voltage level on an input GPIO pin
  - The following circuit is possible, but does not work well:

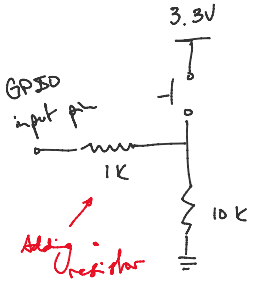


  - When the switch us closed, the pin receives a high voltage

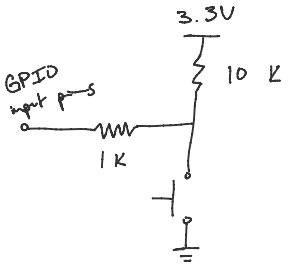
- However, when the switch is open the input pin does not have a definite voltage
  - It "floats"
  - The wire connected to the pin acts as an antenna, and the voltage level fluctuates
- To remedy this, the pin is "pulled down" (when the switch is open) by connecting the pin to ground
  - A 10K ohm resistor is used to limit the current flow from the +3.3V supply to ground when the switch is closed
    - Without this, an electrical short would occur



  - If the pin is accidentally configured to be an output pin and its output is low, then a short will be created when the switch closes
    - Is fixed by adding a 1K ohm current limiting resistor:



- A "pull up" circuit is similar, except the pin is at 3.3.V (high) until the switch is closed
  - It then goes low (0V)



- IMPORTANT: always use a 3.3V supply
  - Using a higher voltage (such as 5V) will damage your Pi

Disabling Pullups/Pulldowns

- Since we will use external circuitry to pull an input pin high or low, we must disable the internal pull up/down control for the pin
- Procedure:
  1. Set bits 1:0 in GPPUD to 00
  2. Wait 150 cycles
  3. Write a 1 to the corresponding bit in GPPUDCLK0
  4. Wait 150 cycles
  5. Clear all bits in GPPUDCLK0
- Example code (available on D2L):

Readings and Ex:

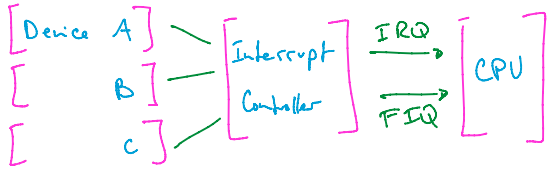
- BCM2837 ARM Peripherals Chapter 6

Exceptions and Interrupts

January 10, 2019 10:30 PM

Basic Concepts

- An *exception* is a system event or condition that requires action by privileged software to deal with the event
- This software is called an *exception handler*
  - Or *interrupt handler*
  - Or *interrupt service routine* (ISR)
- Some exceptions are *synchronous*
  - They result from the execution (or attempted execution) of an instruction
  - Once handled, control returns to the next instruction
  - Abort exceptions:
    - *Instruction abort*: generated by a failed instruction fetch
    - *Data abort*: generated by a failed data access
  - Debug exceptions
  - Exception generating instructions:
    - Supervisor call (SVC)
      - Used to invoke system functions from EL0 (eg. File I/O)
    - Hypervisor call (HVC)
    - Secure Monitor call (SMC)
    - Handlers for these run at a higher privilege level
      - Eg: SVC handlers run at EL1
    - On exception return, we change back to the original exception level
- Some exceptions are *asynchronous*
  - Called *interrupts*
  - May occur at any time
    - i.e. during the execution of *any* instruction in a program
  - Before handling, we save the program state (PSTATE) and current program counter (PC) value
  - Once handled, we restore program state and return control to the next instruction (i.e. To PC + 4)
  - Interrupts are usually triggered by external hardware
    - i.e. by an electrical signal applied to an input pin on the CPU
  - ARMv8 has 2 types of interrupts:
    - IRQ: for normal interrupt requests
    - FIQ: for fast interrupt requests
      - Or secure interrupt requests
  - Since there may be many sources of hardware interrupts, these are routed through an *interrupt controller*:



  - The controller serializes and prioritizes contending interrupts
  - Can also assign a particular device to the FIQ line
- There normally is one exception handler for each type of exception
  - The CPU selects the exception handler to use by means of an *exception vector table*
    - Or *interrupt vector table*

Exception Handling Registers

- Processor State (PSTATE) fields record the current state of the CPU
  - Condition flags: N, Z, C, V
  - nRW: current execution state
    - 0: AArch64 1: AArch32
  - EL: current exception level
    - 2 bits:
      - 00: EL0 01: EL1
      - 10: EL2 11: EL3
  - Stack pointer selection bit
    - 0: use SL_EL0 for SP
    - 1: use SP_ELx for SP, where x is the current EL
  - Exception mask bits:
      - D: Debug
      - A: System Error (SError)
      - I: IRQ
      - F: FIQ
    - If set to 1, the exception is disabled (is *masked*)
      - If 0, the exception is enabled
  - Others
  - Are read using these special purpose registers: NZCV, CurrentEl, SPSel, DAIF
    - Eg:

```
mrs x0, SPSel
```

  - Individual DAIF flags can be set using the DAIFSet register
    - Eg:

```
msr DAIFSet, 0b0010
```

  - Or cleared using DAIFClr
    - Eg:

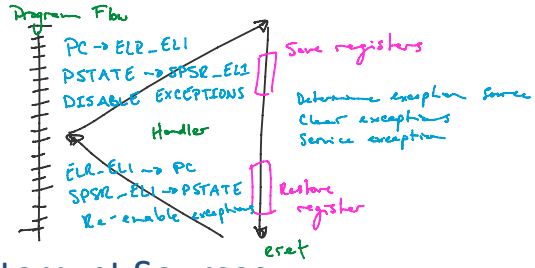
```
msr DAIFClr, 0b0010
```

- The *Saved Program Status Register* records PSTATE information when an exception is taken
  - Actually 3 registers, for EL 1,2, or 3:
    - SPSR_EL1, SPSR_EL2, SPSR_EL3
  - *Programmer's Guide* p.10-4
    - Figure 10-2 When exceptions are taken from AArch64
- The *Exception Link Register* records the return address for an exception
  - Actually 3 registers: ELR_EL1, ELR_EL2, ELR_EL3
  - Is loaded with the current PC value when the exception is taken

Exception Handling Process

- When an exception is triggered, the hardware automatically:
  - Records PSTATE information into the appropriate SPSR register
    - Depends on which EL is used to service the exception
      - May be the current EL, or a higher EL
  - Records the current PC value into the appropriate ELR register
  - Disables exceptions by masking the DAIF flags
  - Uses the vector table to branch to the handler

- The exception handler should:
  - Save any general purpose registers it will use to the stack, allocating memory as needed
  - Determine the exception source
  - Clear the *pending* bit for this source
    - i.e. *clear* the interrupt or exception
  - Do work to service the exception
    - Will be specific to the kind of exception
  - Restore the contents of any registers used from the stack, deallocating memory as needed
  - Return from the exception using the *eret* instruction
- On returning from an exception, the hardware automatically:
  - Restores the pre-exception PSTATE from the SPSR register
    - The original DAIF flags are restored, so interrupts will be re-enabled
    - The original EL will be restored (if it changed)
  - Loads the PC register from ELR
    - Causes execution to resume at the point where the exception was triggered



Interrupt Sources

- Most SoC peripherals can be configured to trigger interrupts
  - Eg: Trigger an interrupt when a rising edge is detected on GPIO input pin 17

```
*GPREN0 = (0x1 << 17); //GPIO rising edge detect enable
```

- Also have registers to record if a particular interrupt event occurred
  - Eg: check if pin 17 generated an interrupt

```
if (*GPEDS0 == (0x1 << 17 )) {...} //Event detect status reg
```

- Write a 1 to the same bit to clear the interrupt
  - This also clears the *pending* flag in the interrupt controller
  - Eg: `*GPEDS0 = (0x1 << 17);`
- SoC interrupts are routed through the interrupt controller
- The ARM CPU can also be a source of interrupts

Interrupt Controller

- Has 3 registers to enable specific IRQs (interrupt requests)
  - Eg: enable interrupts for all GPIO pins
    - This is IRQ 52, so set bit 20 in Interrupt Enable Register 2

```
*IRQ_ENABLE_IRQS_2 = (0x1 << 20);
```

- Has 3 registers to disable IRQs
- Has 3 registers that record which specific IRQ is *pending* (i.e. need to be serviced)
  - Eg: Check if IRQ 52 is pending

```
if (*IRQ_PENDING_2 == (0x1 << 20)) {...}
```

- *Broadcom Manual* p.112 - 113:

Registers overview:

Address offset[7]	Name	Notes
0x200	IRQ basic pending	
0x204	IRQ pending 1	
0x208	IRQ pending 2	
0x20C	FIQ control	
0x210	Enable IRQs 1	
0x214	Enable IRQs 2	
0x218	Enable Basic IRQs	
0x21C	Disable IRQs 1	
0x220	Disable IRQs 2	
0x224	Disable Basic IRQs	

ARM peripherals interrupts table.

#	IRQ 0-15	#	IRQ 16-31	#	IRQ 32-47	#	IRQ 48-63
0		16		32		48	smi
1	system timer match 1	17		33		49	gpio_int[0]
2		18		34		50	gpio_int[1]
3	system timer match 3	19		35		51	gpio_int[2]
4		20		36		52	gpio_int[3]
5		21		37		53	i2c_int
6		22		38		54	spi_int
7		23		39		55	pcm_int
8		24		40		56	
9	USB controller	25		41		57	uart_int
10		26		42		58	
11		27		43	i2c_spi_slv_int	59	
12		28		44		60	
13		29	Aux int	45	pwa0	61	
14		30		46	pwa1	62	
15		31		47		63	

ARM peripherals interrupts table.

0	ARM Timer
1	ARM Mailbox
2	ARM Doorbell 0
3	ARM Doorbell 1
4	GPU0 halted (Or GPU1 halted if bit 10 of control register 1 is set)
5	GPU1 halted
6	Illegal access type 1
7	Illegal access type 0

- Note: GPIO IRQs are grouped into 3 banks
  - IRQ 49: pins 0 - 27
  - IRQ 50: pins 28 - 45
  - IRQ 51: pins 46 - 53
  - IRQ 52: pins All pins

- ARM-side interrupts (*Manual* p.113):

Exception Vector Table

- Contains an entry for each type of exception
  - *Programmer's Guide* p. 10 - 12:
- The table must start at an address evenly divisible by 2048
  - Use .align 11 to ensure this (2^{11} = 2048)
- Each entry must start at an address evenly divisible by 128
  - Use .align 7 to ensure this (2^{7} = 128)
- One of the *Vector Base Address Registers* (VBAR_Elx) must be set to point to the table
  - Implies that EL 1, 2, and 3 can each have their own vector tables
- In this course, we assume:
  - Our code runs in EL1 and in Aarch64
  - Exceptions are taken in EL1
    - i.e. we don't change to a higher EL
  - We always use SP_EL0 for SP
- Thus, we only use the first 4 table entries
  - And use VBAR_EL1 to hold the table address
- Each entry is large enough to hold 32 individual instructions
  - If the handler is short, it can be coded right into the table
  - Often done for FIQ handlers
- Usually each entry branches to a handler "stub" written in assembly
  - This in turn may call a C function that does most of the work
  - The stub saves/restores all general purpose registers
    - Allows C code to function as usual
  - The stub ends with *eret*
- Example: startV2.s
  - On D2L

Programming Exception Handlers

- A handler may have to service exceptions from different sources
- Thus, you need to identify the source of the exception, and handle it with purpose-built code
  - You query:
    - The interrupt controller
    - The SoC peripheral
- Must also clear the exception
- The handler can share data with the rest of a program by writing to a global variable
  - The program can read this later (often in a loop)
- Eg: handlers.c (on D2L)

Program Setup

- In general:
  - Declare and initialize a global shared variable
  - Configure the SoC peripheral to trigger an interrupt under particular conditions
  - Enable the appropriate IRQ in the interrupt controller
  - Enable IRQ exceptions by clearing the appropriate DAIF flag (I flag for a3)
    - These are usually masked on power up
  - Enter an event loop that polls the shared variable, and reacts as necessary
- Eg: main.c (on D2L)

Reading and Ex

- *BCM2837 ARM Peripherals* Chapter 7
- *Programmer's Guide for ARMv8-A* sections 10.1 - 10.5

Super Nintendo Entertainment System (SNES) Controller

January 10, 2019 10:30 PM

The SNES Controller

- Is a game controller for a video game console developed in the early 1990s:
- Uses a simple serial transfer protocol to send button data to the console
  - Easy to program on the Pi using only 3 GPIO pins
- Has 12 buttons

Button Number/Clock Cycle	Button
0	B
1	Y
2	Select
3	Start
4	Joy-pad UP
5	Joy-pad DOWN
6	Joy-pad LEFT
7	Joy-pad RIGHT
8	A
9	X
10	Left
11	Right
12 - 15	Unused

GPIO SNES Pins

- The controller is connected to the Pi using an SNES socket on the breakout board
- The pins in this socket are hardwired to particular GPIO pins:

GPIO pin	GPIO Function	SNES pin	SNES Function
3V3	Power	VRef	Voltage reference
10	Input	DAT	Serial Data
9	Output	LAT	Latch
11	Output	CLK	Serial Clock
GND	Power	GND	Ground

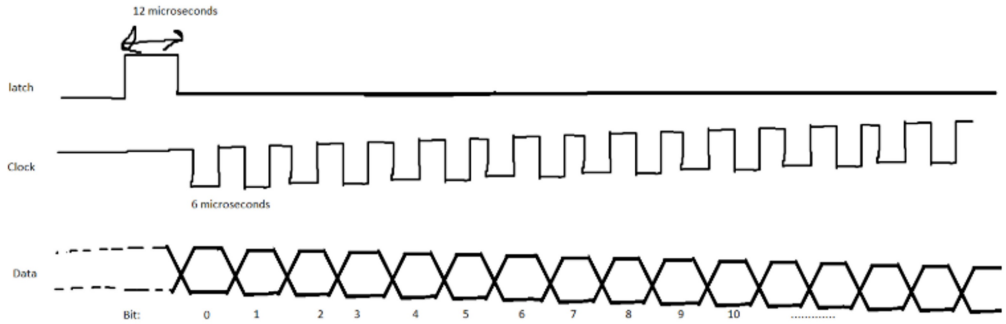
- The Pi provides 3.3V power to the circuitry in the controller using the 3V3 and GND pins
- GPIO pins 10, 9, and 11 use 0V for low (binary 0), and 3.3V for high (binary 1)
  - None of these lines float
    - Thus, internal GPIO pullup/pulldown circuitry should be disabled

GPIO Pin Initialization

- Pins 9 and 11 must be configured as output pins
  - And pin 10 as input
- Must also disable pullup/pulldown circuitry

- Eg: main.c (on D2L in 07_SNESController)

Communication Protocol

- Communication with the controller is initiated by the CPU
- The *Latch* line is held for 12 µs
- The controller responds by *latching* the current state of the buttons into its internal 16-bit shift register
  - 0 means pressed, and 1 means un-pressed
  - Bits 12-15 are always 1
- The CPU is *sampling* the state of the buttons at a discrete moment in time
- The CPU then "clock in" the data by sending 16 clock pulses over the *Clock* line to the controller
  - The controller send 16 bits in series back to the CPU over the *Data* line by shifting bits out of the register
    - This is an example of *serial transfer*
  - The CPU reads the *Data* bit value on the falling edge of each cycle
    - The rising edge shifts the next bit onto the Data line
  - The clock period is 12 µs
- This protocol can be shown using a *timing diagram*:
  - 

- The original SNES sampled the buttons 60 times per second
  - i.e. With a period of 16667 µs

Reading SNES Data

- Can be done in a program by synthesizing the Latch and Clock signals on GPIO pins 9 and 11
  - Set these pins to 0 or 1 with the proper timing
    - Need to insert delays with microsecond accuracy
- The data is read one bit at a time from GPIO pin 10 on the falling edge of the Clock signal
  - The bit pattern is stored into a 16-bit variable (or register)
- Eg: main.c (on D2L)
- This function is called at the desired sample rate inside a loop
  - Eg: main.c (on D2L)

System Timer

- Is a SoC peripheral that provides 64-bit counter that is incremented every microsecond
- The high order 32 bits of the counter are read using the register: SYSTEM_TIMER_CHI
  - And the low order bits with: SYSTEM_TIMER_CLO
  - These are memory mapped registers at addresses 0x3F003008 and 0xF003004
- To delay *n* microseconds:
  1. Read the counter to get the current time
  2. Add *n* to this number to get the future time
  3. Poll the counter in a loop until it is >= the future time
- Eg: systimer.c (on D2L)

Readings and Ex:

- *BCM2837 ARM Peripherals Chpt. 12*

Video Programming

January 10, 2019 10:30 PM

Color Theory

- Visible lights is that part of the electro-magnetic spectrum detectable by the human eye
    - Ranges from red (\lambda \about 700 nm) to violet (\lambda \about 400 nm)
- [Electromagnetic Spectrum Diagram]
- [Color Wavelength Diagram]
- The spectrum describes the *physical colors*
- The eye *perceives* color using 3 types of photoreceptors
    - Each responds to a limited part of the spectrum
        - Roughly red, green, and blue
- [ROYGBV Diagram]
- [Sensitivity of Photoreceptors Diagram]
- Depending on the wavelength of the stimulus, the RGB receptors will be stimulated in different proportions
    - Eg: Yellow stimulates R and G receptors
        - Brain interprets combinations as yellow
        - Increasing R stimulation gives orange
    - Magenta is a non-physical color
        - Not on the spectrum
        - Arises when R and B stimulated together
        - Approximates physical violet
- Additive color model
    - Red, green, and blue light of varying intensity combine to produce a range of perceived colors
    - R + G = yellow
        - R, G are primary colors
        - Y is secondary
    - R + B = magenta
    - B + G = cyan
    - R + B + G = white
    - [Spotlight sample]
- [Blue Green Red Spotlight Model]
- Computer Monitors and projectors commonly use the RGB additive model
    - The intensity of each component can be represented numerically
        - E.g. 24-bit representation uses 1 byte for each
            - 0 (fully dark) to full 255 (full intensity)
            - Gives 2^{24} (\about 16 million) possible colors
        - Each pixel on the screen has its own RGB value
    - [RGB 0 - 255 Variations Diagram]
- Subtractive color model
    - Inks, paint and dyes *absorb* some *wavelengths* of light, and reflects others
    - Starts with white light
        - Inks or dyes come between light source and viewer
            - Or reflected light in case of paper
        - Inks are combined to *subtract* selected wavelengths to create desired color
    - [Subtractive Model]
- Color printers typically uses the CYMK subtractive color model
    - K = key (black)

        - Needed since C + Y + M yields a dark brown, not a true black
- [Butterfly Color Diagram]

Display Technologies

- Liquid Crystal Displays (LCDs)
    - Consists of a 2D array of pixels
    - Each pixel (or cell) has 3 light valves (subpixels) with colored filters covering them
    - A backlight shines white light through the cells
    - Each valve is opened in proportion to an input voltage
        - Gives a particular RGB value
        - Light from each subpixel is blended together by a diffuser to give the colored pixel you perceive
    - [Subpixel of a Color LCD Diagram]

Frame Buffer Architectures

- Frame buffer
    - An array in memory (VRAM), where each element represents a pixel on the display
    - The entire array represents one complete frame
    - The 2D frame is mapped to the ID buffer
        - E.g. VGA is 640 x 480
            - Maps to a 1D array with 307 200 elements
- Common resolutions:
    -

Standard	Size	Aspect Ratio
VGA	640 x 480	4:3
SVGA	800 x 600	4:3
XGA	1024 x 768	4:3
SXGA	1280 x 1024	5:4
UXGA	1600 x 1200	4:3

    - [Wiki: Vector Video Standard]
- Each element in the frame buffer represents the pixel's color
    - Size (*depth*) is specified in *bits per pixel* (bpp)
        - Gives 2^{bbp} colors
    - Common formats (color depths)
        - 1-bit
            - Monochrome
        - 4-bit
            - 14 fixed colors
        - 8-bit indexed
            - Choice of 256 colors from a palette
        - 16-bit *highcolor*
            - 65 536 colors
            - 5 bits for R, B; 6 bits for G
        - 24-bit *truecolor*
            - 16 777 216 colors
            - 8 bits each for R, G, and B
        - 32-bit RGBA
            - Like truecolor, plus 8-bit alpha channel (for transparency effects)
    - Units smaller than 8 bits are packed into a byte to minimize VRAM size
- Higher resolution and/or color depth requires a larger frame buffer

    - Practical now since RAM is cheap
- To draw a single pixel, set its value in the frame buffer
    - Must map from 2D logical space to 1D physical RAM
        - Must know width and height for a particular resolution
    - Use formula: element offset = (y * width) + x
        - X , y are pixel's coordinates
        - Origin is upper left-hand corner
        - x range: 0 to width-1
        - y range: 0 to height-1
    - physical offset = element offset * element size in bytes
    - E.g. 1024 x 768 RGBA (32-bit)
        - Pixel (0, 0)
            - [(0 * 1024) + 0] * 4 = 0
        - Pixel (1023, 767) (lower right-hand corner)
            - [(767 * 1024) + 1023] * 4 = 3 145 724

Mailboxes

- Are used on the Pi to communicate between the CPU and Video Core (GPU)
    - Allows us to allocate and initialize a frame buffer in RAM
- The CPU makes a *request* to the GPU using mailbox 1
- The GPU replies with a *response* using mailbox 0
- Data is passed back and forth using a shared data structure
    - Consists of a variable number of *tags*
        - Each has a specified number of fields
- Each mailbox defines 10 channels
    - We will channel 8: CHANNEL_PROPERTY_TAGS_ARMTOVC
- The *address* of the shared data structure is combined with the *channel number* using |
    - Implies the address must be quadword aligned (i.e. The low-order 4 bits are 0)
    - This combined address is written to register MAILBOX1_WRITE to initiate a *request*
        - And read from MAILBOX0_READ when a *response* arrives
- Must make sure we don't:
        - Write to mailbox 1 if it's full
        - Read from mailbox 0 if it's empty
    - Done by polling the MAILBOX1_STATUS and MAILBOX0_STATUS registers
- Example query (mailbox.c on D2L):
- The shared data structure consists of:
    - Header:
        - Size in bytes of the whole data structure
        - Request/response code
            - 0 to make a request
            - Set to 0x80000000 by GPU if successful
    - Variable number in tags. Each has:
        - Tag identifier
        - Size in bytes of the following buffer
        - Request/response code
            - Request: 0
            - Response: size in bytes of the response | 0x80000000
                - Should be <= buffer size
            - Buffer
                - Size varies according to the tag type
    - End tag
- Example shared data structure declaration:

```
volatile unsigned int __attribute__((aligned(16)))
mailbox_buffer[36];
```

- Tags to allocate a 1024 x 768 x 32 frame buffer (framebuffer.c on D2L):
  - CODE
- This structure is used to make a mailbox query
  - Once the GPU responds, we find the returned settings by reading the buffers
  - Eg: framebuffer.c

Video Programming

- To draw a single pixel:
  - Must know the x (column) and y (row) position of the pixel
  - Must know the RGB value
    - Can use the color codes defined in framebuffer.c (on D2L)
  - Eg: Draw a blue pixel at (5, 36)

```
unsigned int *pixel = framebuffer;
int columnm = 5, row = 36;

pixel[(row * frameBufferWidth) + column] = 0x000000FF
```

- To draw a filled square:
  - Must specify:
    - Upper left-hand corner coordinates
    - Size of the square in pixels
    - The fill color
  - Algorithm:
    - Draw the square row by row, from top down
    - Eg: (framebuffer.c)
      - `drawSquare` method

Readings and Ex

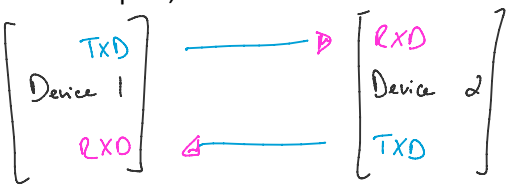
- htmlcolorcodes.com
- github.com/raspberrypi/firmware/wiki/Mailboxes

UART Protocol

January 10, 2019 10:30 PM

Basic Concepts

- Universal Asynchronous Receiver Transmitter (UART) is a low-speed serial I/O protocol
  - Commonly used to connect a computer to a terminal or other computer
  - Is inexpensive
  - May be implemented on a discrete IC
    - Or integrated onto a SoC
- Communication may be:
  - *Simplex*: in one direction only, over one wire
  - *Full duplex*: in both directions at the same time using two wires
  - *Half duplex*: devices take turns transmitting and receiving over one wire
- If simplex or half duplex, then there is only one *data* line
- If full duplex, there are two *data* lines: TXD (transmit data) and *RXD* (receive data)



- Since the protocol is asynchronous, there is *no* clock line
  - Thus, both devices must be set to the same *baud rate* (frequency in bits per second)
- Each device has shift registers connected to its RXD and TXD lines
  - When transmitting, the device:
    - Fills the register with a *word* of data using its parallel load lines
    - Shifts out the bits serially over the TXD line
      - Starts with the LSB
  - When receiving, the device:
    - Shifts in bits from the RXD line
    - Signals to the device when the complete word has been shifted in
      - The data word is then available using the parallel output line
  - Some devices incorporate FIFO buffers, so that several words can be stored when doing I/O

UART Protocol

- The data lien is 1 (high) when not transmitting
- A *start bit* of 0 (low) signals that a word is being transmitted
  - i.e. signals the start of the *frame*
- The data bits are then serially transmitted
  - Word size can be 5, 6, 7, or 8 bits
    - Both the transmitter and receiver must set the same
- Optionally, a *parity bit* (*check bit*) is transmitted
  - Used for error detection on a noisy line
  - A 0 or 1 is transmitted so that the total number of 1's is *even* or *odd*
  - Both TX and RX must be set the same
    - i.e. must agree if a parity bit is used, and if it is even or odd
- A *stop bit* of 1 (high) is transmitted to mark the end of the *frame*
  - Optionally, 2 stop bits can be specified
    - Must be set the same in both the TX and RX
- Timing diagram: 8-bit data, no parity, 1 stop bit

        - Must be set the same in both the TX and RX
- Timing diagram: 8-bit data, no parity, 1 stop bit

Raspberry Pi Mini UART

- Is a SoC peripheral connected to particular GPIO pins
- Handles in hardware the serialization of data over TXD0 and RSD0 lines
    - Also automatically frames the data with a start bit and 1 stop bit
    - No parity bit allowed
    - Settable word size: 7 or 8 bits
    - Has both transmit and receive FIFO buffers
        - Each holds up to 8 words
        - Must be enabled
    - Optionally, transmit and receive interrupts can be enabled

UART Registers

- Are memory mapped registers with the base address: 0x3F215040
- AUX_MU_IO (Auxiliary, mini UART, input/output)
    - When written to, data is put into the transmit FIFO
    - When read from, data comes from the receive FIFO
- AUX_MU_LSR (Auxiliary, mini UART, L. status register)
    - Bit 0: if set, the receive FIFO holds at least 1 word
    - Bit 5: if set, the transmit FIFO can accept at least 1 word
- Other registers are used to configure the UART
    - See below

UART Initialization

- Must configure the device so that:
    - GPIO pins 14 and 15 are set to TXD and RXD
        - Done by setting FSEL14 and FSEL15 in GPFSEL1 to alternate function 5
    - The Mini UART is enabled
        - Set bit 0 in AUX_ENABLE to 1
    - Disable UART interrupts and turn off flow control
        - Set both AUX_MU_IER and AUX_MU_CNTL to 0
    - Set the word size to 8 bits
        - Set bits 1:0 in AUX_MU_LCR to 11
    - Enable both the receive and transmit FIFOs, and clear their content
        - Set bits 7:6 and 2:1 in AUX_MU_IIR to 11
    - Set the Baud rate to 115200
        - Write 270 to AUX_MU_BAUD
        - Formula: rint((systemClockRate/(8 * 115200)) - 1)
            - Clock rate is 250 MHz
    - Enable the UART's transmitter and receiver
        - Set bits 1:0 in AUX_MU_CNTL to 11
- Example: (uart.c on D2L)

Reading and Writing Character

- To write:
    - Poll AUX_MU_LSR, looping until the *transmitter empty* bit (bit 5) is set
        - Must make sure the transmit FIFO is not full

    - Write the character (1 byte) to the AUX_MU_IO
        - Data is put into the FIFO
        - Will eventually be transmitted serially over TXD
- Eg: (uart.c)
- To read:
    - Poll AUX_MU_LSR, looping until the *data ready* bit (bit 0) is set
        - Indicates that the receive FIFO contains at least one word
    - Read the character (1 byte) from AUX_MU_IO
- Eg: (uart.c)

Readings and Ex

BCM Peripherals manual section 2.2

Final Exam

Thursday, April 11, 2019 4:15 PM

April 27, 7 - 9 pm
Science B 103

120 min

4 Sections:

- 35 MC
- Design question
    - Sequential Logic Design
        - Assignment 2
    - 30 min
- 2 coding Questions
        - Code on the Pi
    1. GPIO / Interrupts
        i. Assignment 3
    2. Frame buffer programming
        a) Assign 4
    a) Example
        1) Single pixel
        2) Rectangle
        3) Square
        4) Line
    - SNES tested elsewhere
- Timing diagram
    - For protocol
        - UART
        - SNES
    - How info is transmitted
- Long answers

- Emphasis on 2 half of the course

PREP with assign 2, 3, 4

Syllabus:

- Digital Logic Design
    - MC
- Synthesis of Clocked Sequential Circuits
    - Assignment 2
- Raspberry Pi
    - MC
- GPIO
    - Coding questions
- Exceptions and Interrupts
    - Coding questions
- SNES controller
    - MC and long answer
- Frame buffer
    - Coding questions

- UART
    - MC
    - Long answer