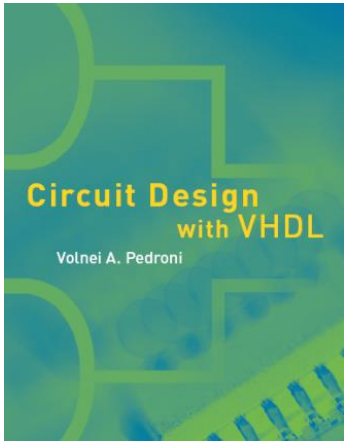
Circuit Design with VHDL

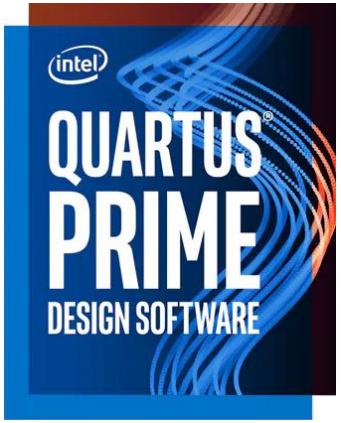
Chapters 1-3: Introduction

Instructor: Ali Jahanian

Text book & required tools

- Class
  - Textbook:
    - Volnei A. Pedroni, “Circuit Design with VHDL”, Third Edition, MIT Press, 2020.
  - Software:
    - ModelSim
    - ISE/Vivado
  - Slides’ reference:
    - Dr. Saeed Gorgin





Evaluation

- **Grades**
  - Homework & projects 30% **<No delivery= -1/2 grade>**
  - Quiz 10% **<No delivery= -1/2 grade>**
  - Final Project 10%
  - Final Exam 50%

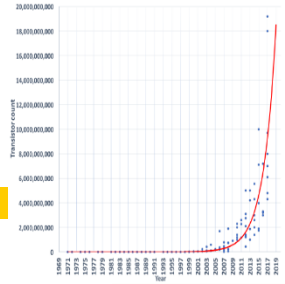
Course Strategy

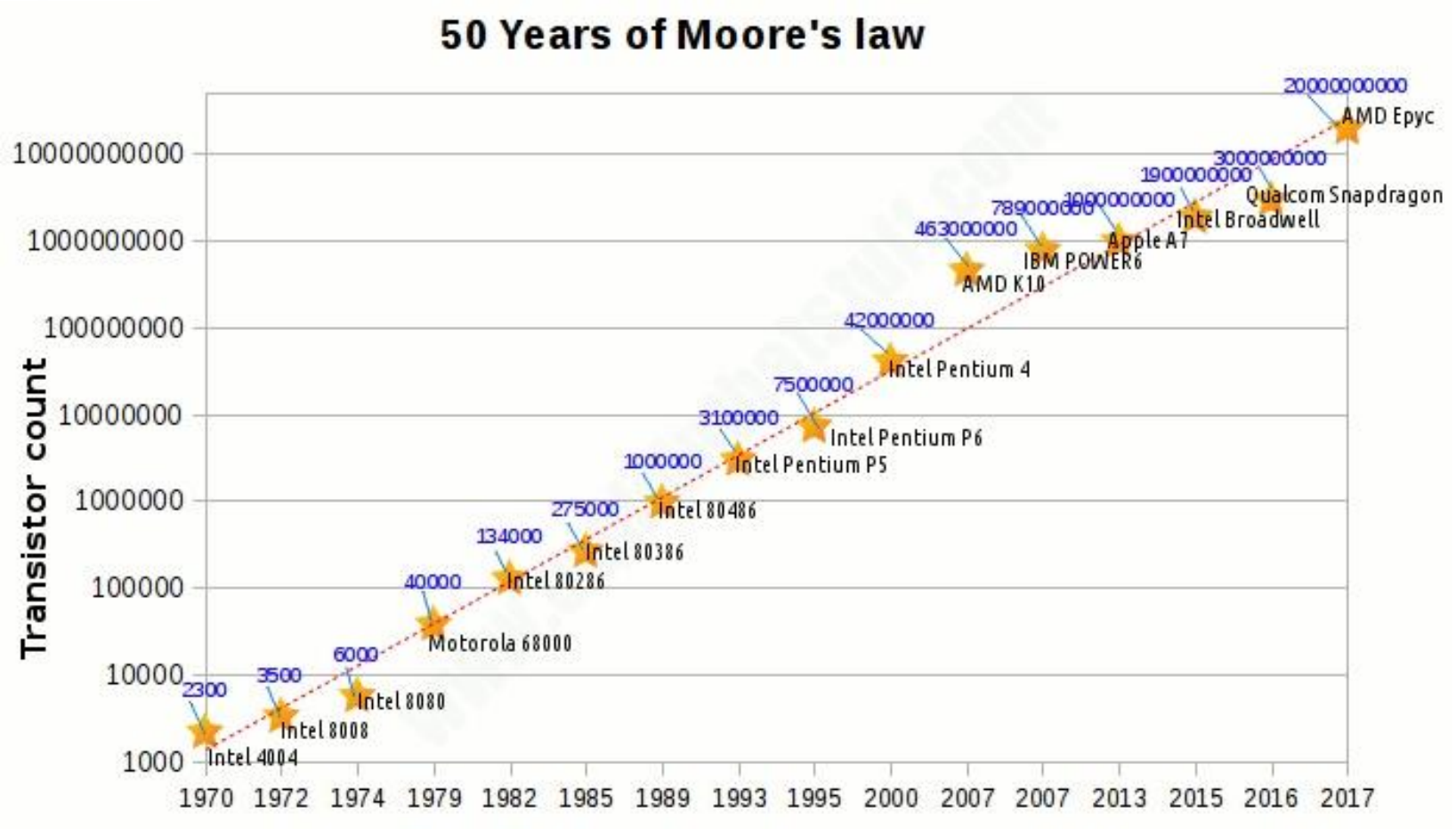
- At the end of this course you should be able to:
  - Concepts of Hardware Modeling
  - Design and modeling with VHDL
  - Synthesize and implement of digital systems on FPGAs
  - Master on High-Level Synthesis

Course Outline

- Introduction
- Circuit Design with VHDL
- FPGA structure
- Hardware Synthesis using ISE
- High-level Synthesis

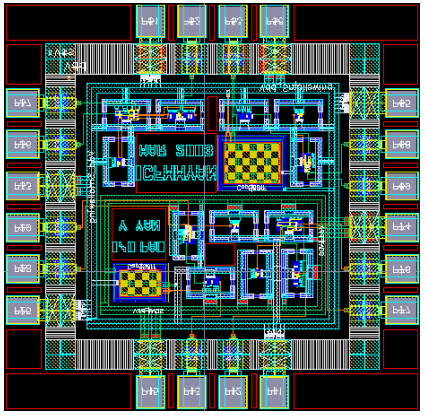
Moore's Law



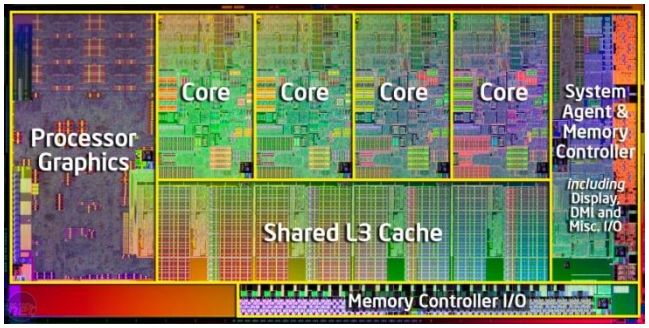

50 Years of Moore's law
Transistor count
10000000000
1000000000
100000000
10000000
1000000
100000
10000
1000
1970
1972
1974
1979
1982
1985
1989
1993
1995
2000
2007
2007
2013
2015
2016
2017
2300
Intel 4004
3500
Intel 8008
6000
Intel 8080
40000
Motorola 68000
134000
Intel 80286
275000
Intel 80386
1000000
Intel 80486
3100000
Intel Pentium P5
7500000
Intel Pentium P6
42000000
Intel Pentium 4
463000000
AMD K10
789000000
IBM POWER6
1900000000
Apple A7
3000000000
Intel Broadwell
20000000000
Qualcom Snapdragon
AMD Epyc

Design Styles

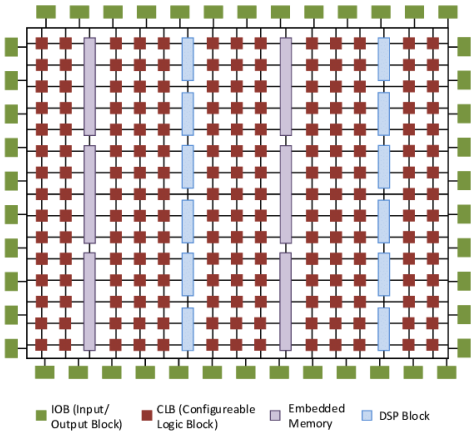
- Full-custom design



- Semi-custom design



- Gate array design



ASIC or FPGA?


FPGA
VS
ASIC
Comparison
Fast
Time to Market
Slow
Low
NRE
High
Simple
Design Flow
Complex
High
Unit Cost
Low
Medium
Performance
High
High
Power Consumption
Low
Medium
Unit Size
Low
HARDWAREBEE

Basic Logic Review

- Combinational Logic Review [©Hwan08]
  - Basic Logic Review
    - Basic Gates (AND,OR,XOR,NAND,NOR)
    - DeMorgan's Law
  - Combinational Logic Blocks
    - Multiplexers
    - Decoders, Demultiplexers
    - Encoders, Priority Encoders
    - Half Adders, Full Adders
  - Multi-Bit Combinational Logic Blocks
    - Multi-bit multiplexers
    - Multi-bit adders
    - Comparators

Basic Logic Review

- Sequential Logic Review [©Hwan08]
  - Sequential Logic Building Blocks
    - Latches, Flip-Flops
  - Sequential Logic Circuits
    - Registers, Shift Registers, Counters
    - Memory (RAM, ROM)
    - Simple Finite State Machines (Mealy, Moore)

Basic Logic Review

- Digital Logic
  - Digital logic gates implement Boolean functions
    - 1 represents true
    - 0 represent false
  - Two types of logic
    - Combinational logic: output depends on the input
    - Sequential logic: output depends on the input and previous outputs and/or inputs

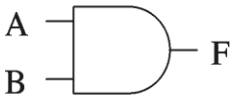
Digital Logic

- Digital logic gates implement Boolean functions
  - 1 represents true
  - 0 represent false
- Two types of logic
  - Combinational logic: output depends on the input
  - Sequential logic: output depends on the input and previous outputs and/or inputs

Basic Concepts

- Simple logic gates
  - AND → 0 if one or more inputs is 0
  - OR → 1 if one or more inputs is 1
  - NOT
  - NAND = AND + NOT
    - 1 if one or more inputs is 0
  - NOR = OR + NOT
    - 0 if one or more input is 1
  - XOR implements exclusive-OR function
- NAND and NOR gates require fewer transistors than AND and OR in standard CMOS
- Functionality can be expressed by a truth table
  - A truth table lists output for each possible input combination

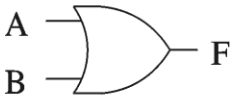
Basic Logic Gates



AND gate

A	B	F
0	0	0
0	1	0
1	0	0
1	1	1

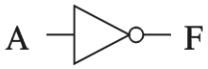
Logic symbol



OR gate

A	B	F
0	0	0
0	1	1
1	0	1
1	1	1

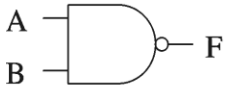
Truth table



NOT gate

A	F
0	1
1	0

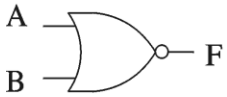
Logic symbol



NAND gate

A	B	F
0	0	1
0	1	1
1	0	1
1	1	0

Truth table



NOR gate

A	B	F
0	0	1
0	1	0
1	0	0
1	1	0

Truth table

XOR gate

A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

Logic symbol

Truth table

Number of Functions

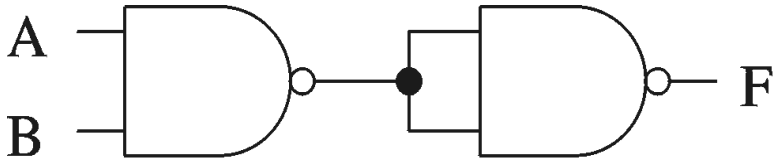
- Number of functions
  - With N logical variables, we can define
    2^{2^N} functions
  - Some of them are useful
    - AND, NAND, NOR, XOR, …
  - Some are not useful:
    - Output is always 1
    - Output is always 0

Complete Set of Gates

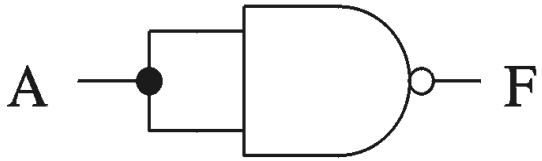
- Complete sets
  - A set of gates is complete
    - if we can implement any logical function using only the type of gates in the set
  - Some example complete sets
    - {AND, OR, NOT}
    - {AND, NOT}
    - {OR, NOT}
    - {NAND}
    - {NOR}
  - Minimal complete set
    - A complete set with no redundant elements.

NAND as a Complete Set

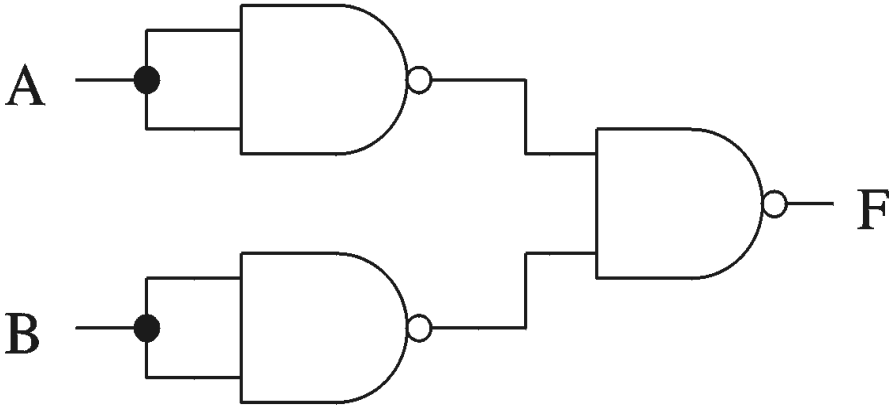
- Proving NAND gate is universal



AND gate



NOT gate



OR gate

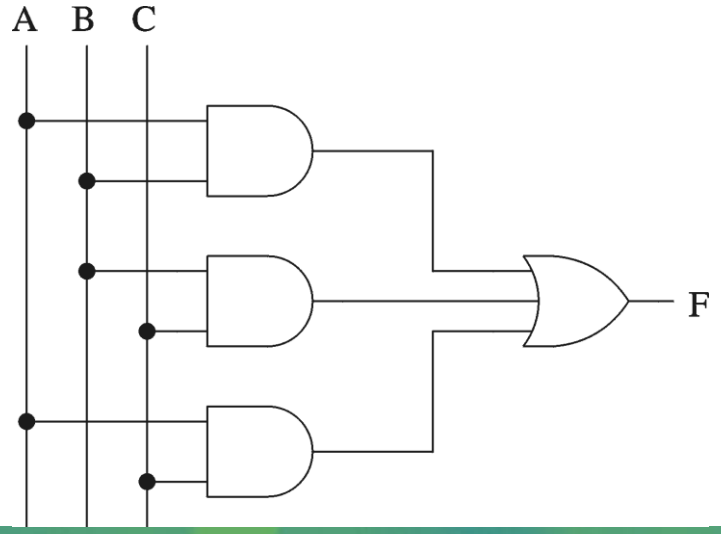
Logic Functions

- Logical functions can be expressed in several ways:
  - Truth table
  - Logical expressions
  - Graphical form
  - HDL code
- Example:
  - Majority function
    - Output is one whenever majority of inputs is 1
    - We use 3-input majority function

Logic Functions…

- Truth table
- Logical expression form

  $F = A\,B + B\,C + A\,C$
- Graphical schematic form



A	B	C	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Boolean Algebra

Boolean identities

Name	AND version	OR version
Identity	$x \cdot 1 = x$	$x + 0 = x$
Complement	$x \cdot x' = 0$	$x + x' = 1$
Commutative	$x \cdot y = y \cdot x$	$x + y = y + x$
Distribution	$x \cdot (y+z) = xy+xz$	$x + (y \cdot z) = (x+y)\,(x+z)$
Idempotent	$x \cdot x = x$	$x + x = x$
Null	$x \cdot 0 = 0$	$x + 1 = 1$

Boolean Algebra…

- Boolean identities (cont'd)

Name	AND version	OR version
Involution	$x = (x')'$	
Absorption	$x \cdot (x+y) = x$	$x + (x \cdot y) = x$
Associative	$x \cdot (y \cdot z) = (x \cdot y) \cdot z$	$x + (y + z) = (x + y) + z$
de Morgan	$(x \cdot y)' = x' + y'$	$(x + y)' = x' \cdot y'$

(de Morgan's law in particular is very useful)

Majority Function Using Other Gates

- Using NAND gates
  - Get an equivalent expression

$$A\,B + C\,D = (A\,B + C\,D)''$$

  - Using de Morgan's law

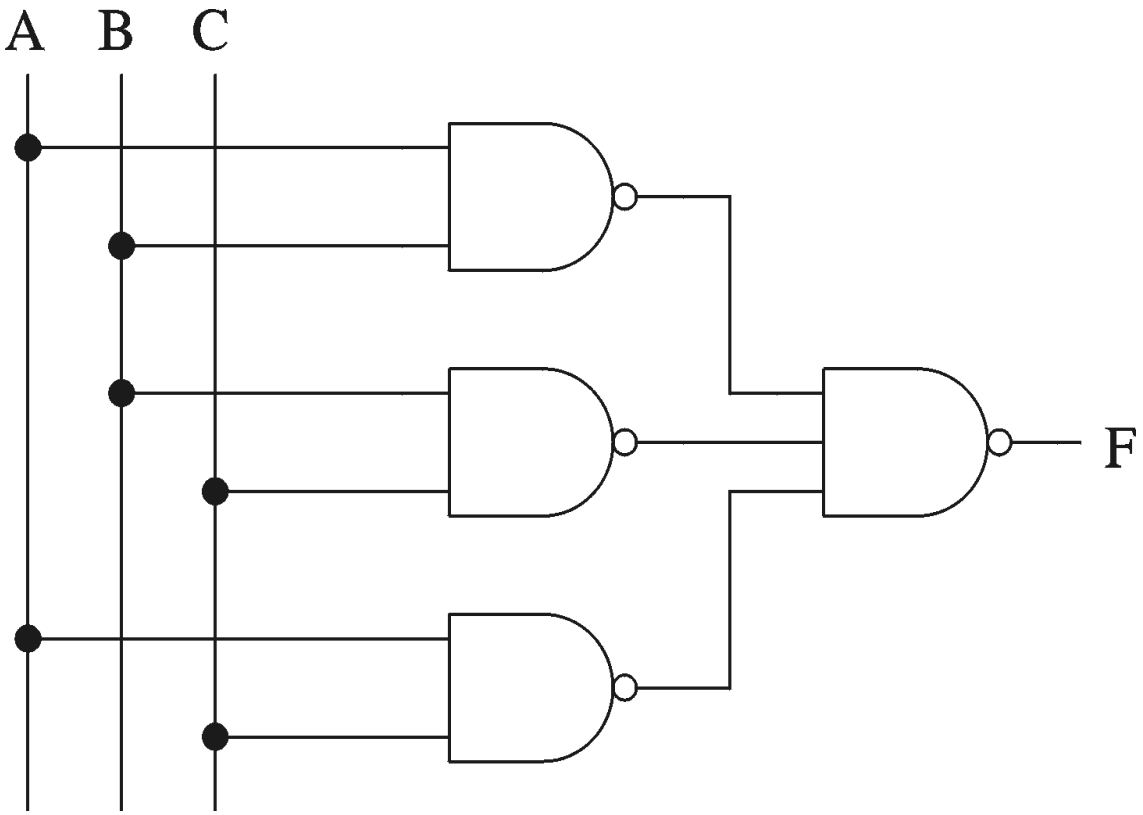
$$A\,B + C\,D = (\,(A\,B)' \cdot (C\,D)')'$$

  - Can be generalized
    - Example: Majority function

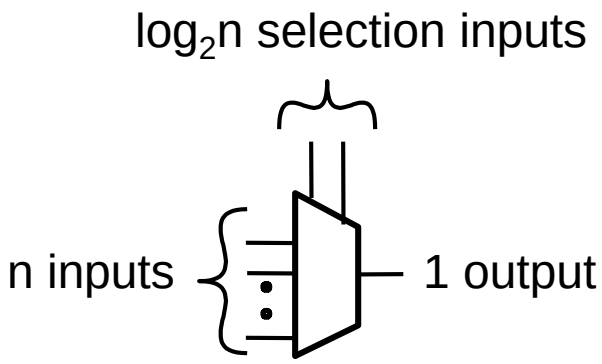
$$A\,B + B\,C + AC = ((A\,B)' \cdot (B\,C)' \cdot (AC)')'$$

Majority Function Using Other Gates…

- Majority function

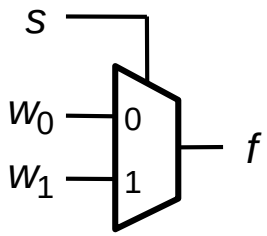


Combinational Logic Building Blocks



- Multiplexers
  - n binary inputs (binary input = 1-bit input)
  - $\log_2 n$ binary selection inputs
  - 1 binary output
  - Function: one of n inputs is placed onto output
  - Called **n-to-1** multiplexer

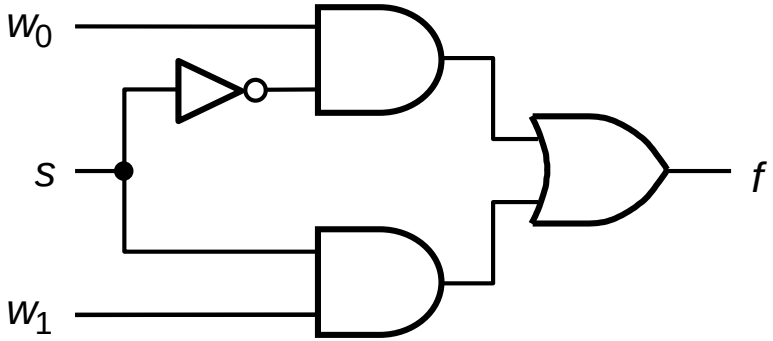
2-to-1 Multiplexer



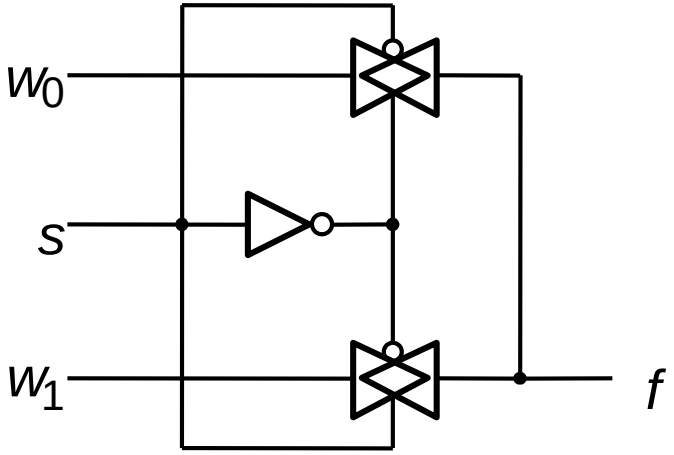
(a) Graphical symbol

s	f
0	w_0
1	w_1

(b) Truth table



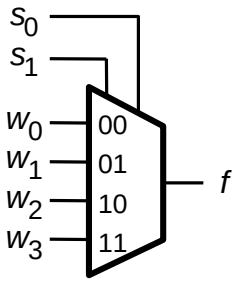
(c) Sum-of-products circuit



(d) Circuit with transmission gates

Source: Brown and Vranesic

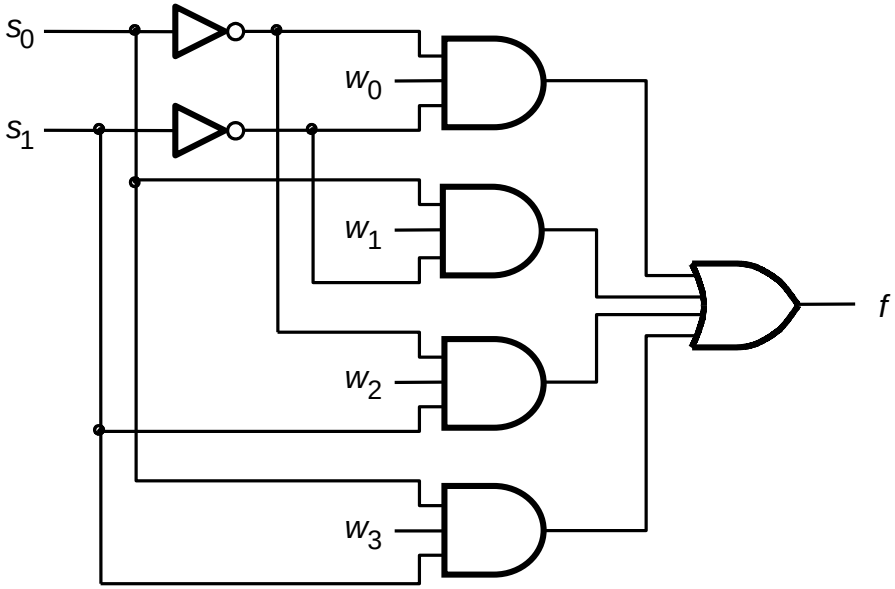
4-to-1 Multiplexer



(a) Graphic symbol

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

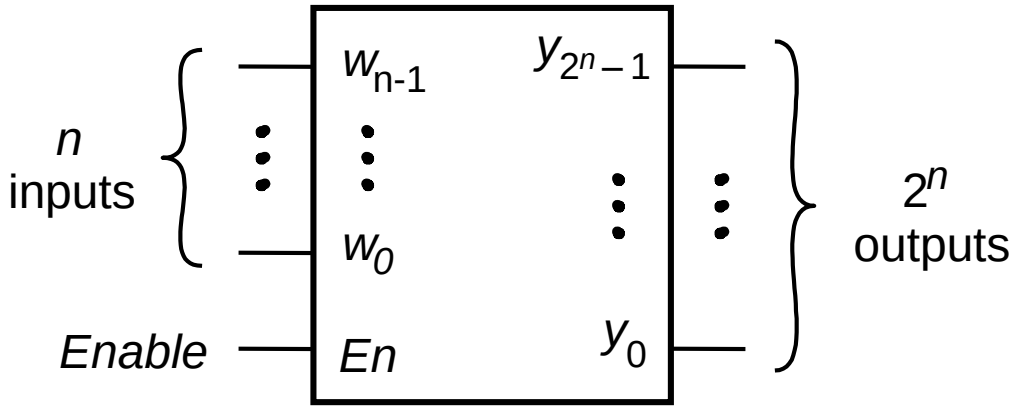
(b) Truth table



(c) Circuit

Source: Brown and Vranesic

Decoders

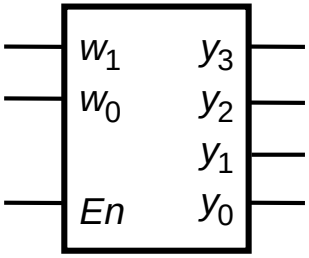


- Decoder
  - n binary inputs
  - 2^n binary outputs
  - Function: decode encoded information
    - If enable=1, one output is asserted high, the other outputs are asserted low
    - If enable=0, all outputs asserted low
  - Often, enable pin is not needed (i.e. the decoder is always enabled)
  - Called **n-to-2^n** decoder
    - Can consider n binary inputs as a single n-bit input
    - Can consider 2^n binary outputs as a single 2^n-bit output
  - Decoders are often used for RAM/ROM addressing

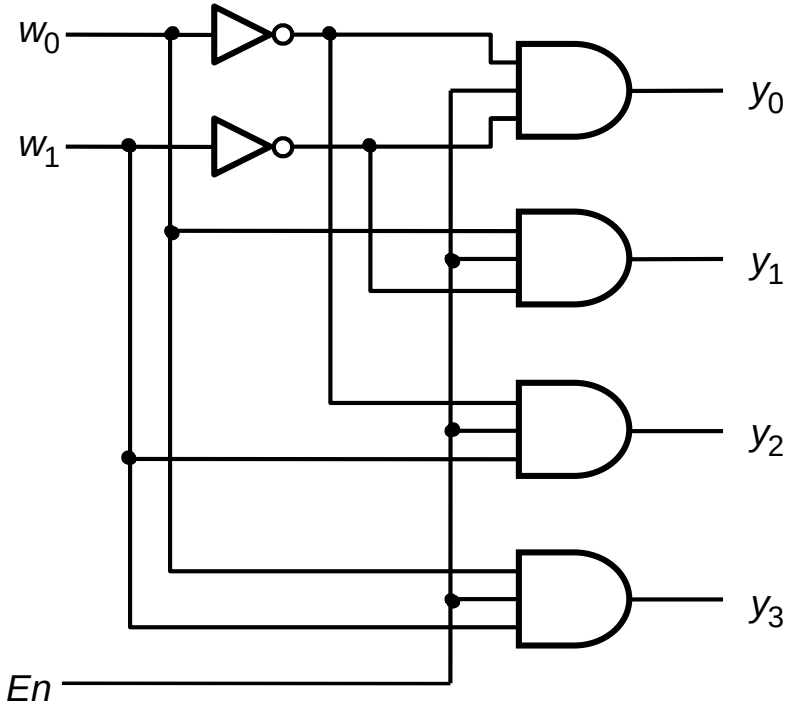
2-to-4 Decoder

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

(a) Truth table

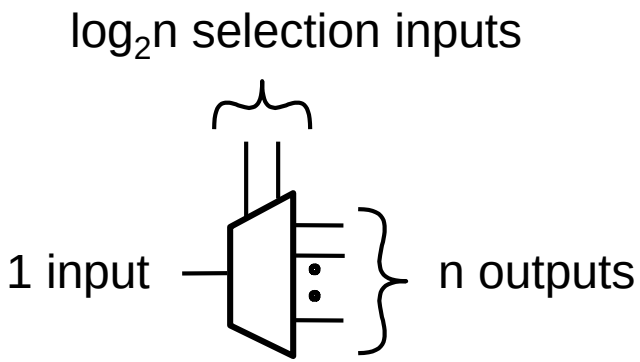


(b) Graphical symbol



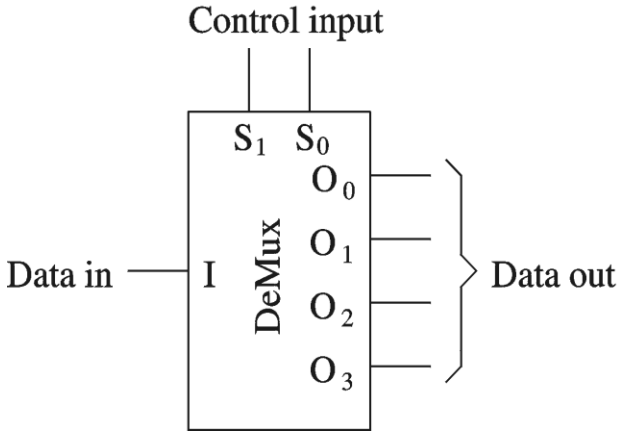
(c) Logic circuit

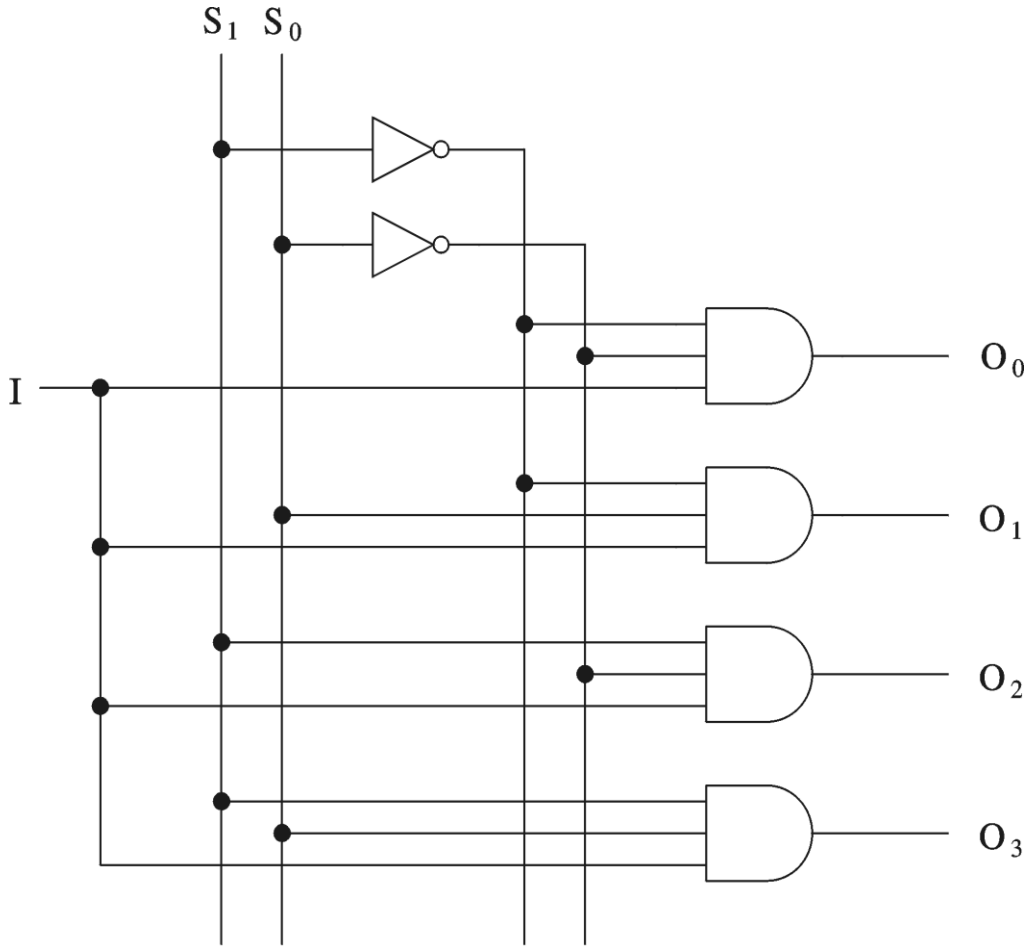
Demultiplexers



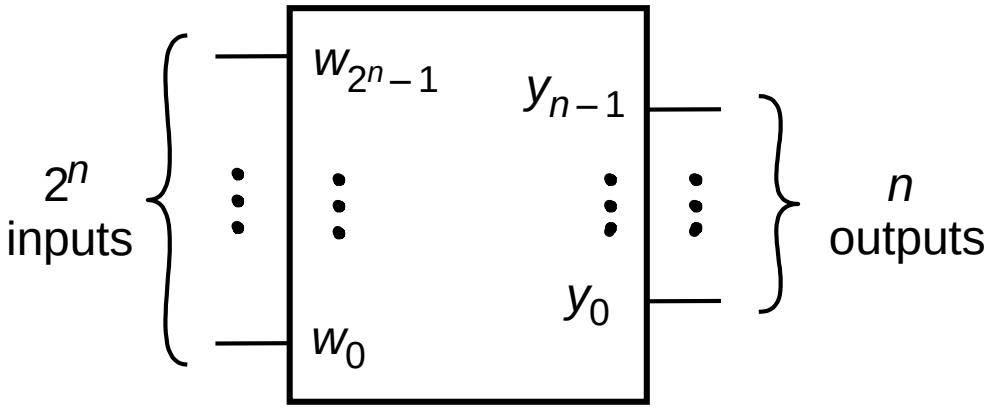
- Demultiplexer
  - 1 binary input
  - n binary outputs
  - $\log_2 n$ binary selection inputs
  - Function: places input onto one of n outputs, with the remaining outputs asserted low
  - Called **1-to-n** demultiplexer
- Closely related to decoder
  - Can build 1-to-n demultiplexer from $\log_2 n$-to-n decoder by using the decoder's enable signal as the demultiplexer's input signal, and using decoder's input signals as the demultiplexer's selection input signals.

1-to-4 Demultiplexer





Encoders

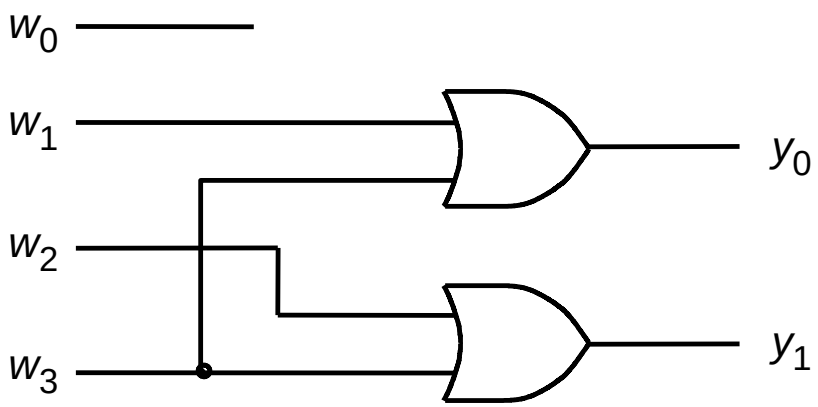


- Encoder
  - 2^n binary inputs
  - n binary outputs
  - Function: encodes information into an n-bit code
  - Called **2^n-to-n** encoder
    - Can consider 2^n binary inputs as a single 2^n-bit input
    - Can consider n binary output as a single n-bit output
- Encoders only work when **exactly one** binary input is equal to 1

4-to-2 Encoder

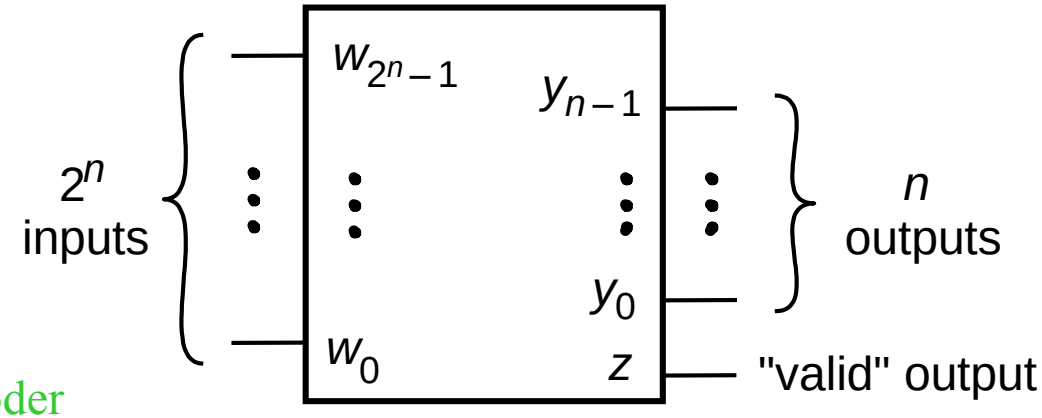
w_3	w_2	w_1	w_0	y_1	y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

(a) Truth table



(b) Circuit

Priority Encoders



- Priority Encoder
  - 2^n binary inputs
  - n binary outputs
  - 1 binary "valid" output
  - Function: encodes information into an n-bit code based on priority of inputs
  - Called **2^n-to-n** priority encoder
- Priority encoder allows for multiple inputs to have a value of '1', as it encodes the input with the highest priority (MSB = highest priority, LSB = lowest priority)
  - "valid" output indicates when priority encoder output is valid
  - Priority encoder is more common than an encoder

4-to-2 Priority Encoder

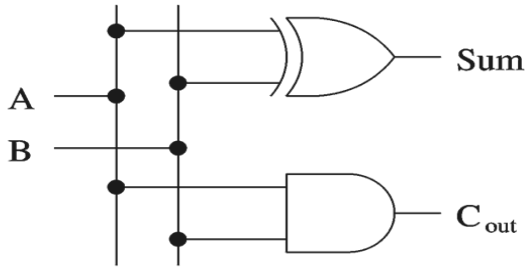
w_3	w_2	w_1	w_0	y_1	y_0	z
0	0	0	0	-	-	0
0	0	0	1	0	0	1
0	0	1	-	0	1	1
0	1	-	-	1	0	1
1	-	-	-	1	1	1

Single-Bit Adders

- Half-adder
  - Adds two binary (i.e. 1-bit) inputs *A* and *B*
    - Produces a *sum* and *carryout*
  - Problem: Cannot use it alone to build larger adders
- Full-adder
  - Adds three binary (i.e. 1-bit) inputs *A*, *B*, and *carryin*
    - Like half-adder, produces a *sum* and *carryout*
  - Allows building M-bit adders (M > 1)
    - Simple technique
      - Connect C_{out} of one adder to C_{in} of the next
    - These are called *ripple-carry adders*
    - Shown in next section

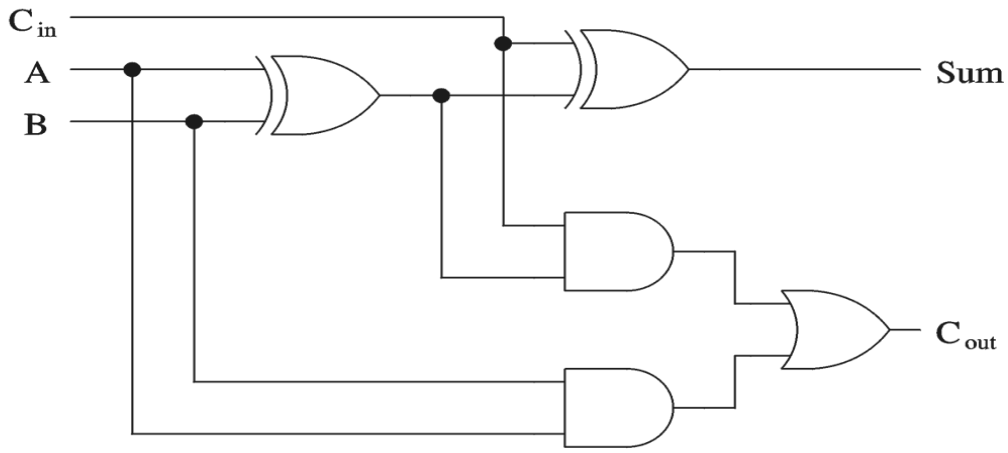
Single-Bit Adders …

A	B	Sum	C_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



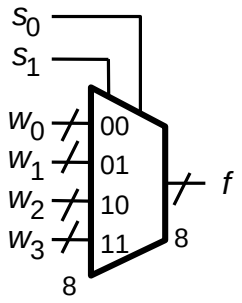
(a) Half-adder truth table and implementation

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



(b) Full-adder truth table and implementation

Multi-bit 4-to-1 Multiplexer



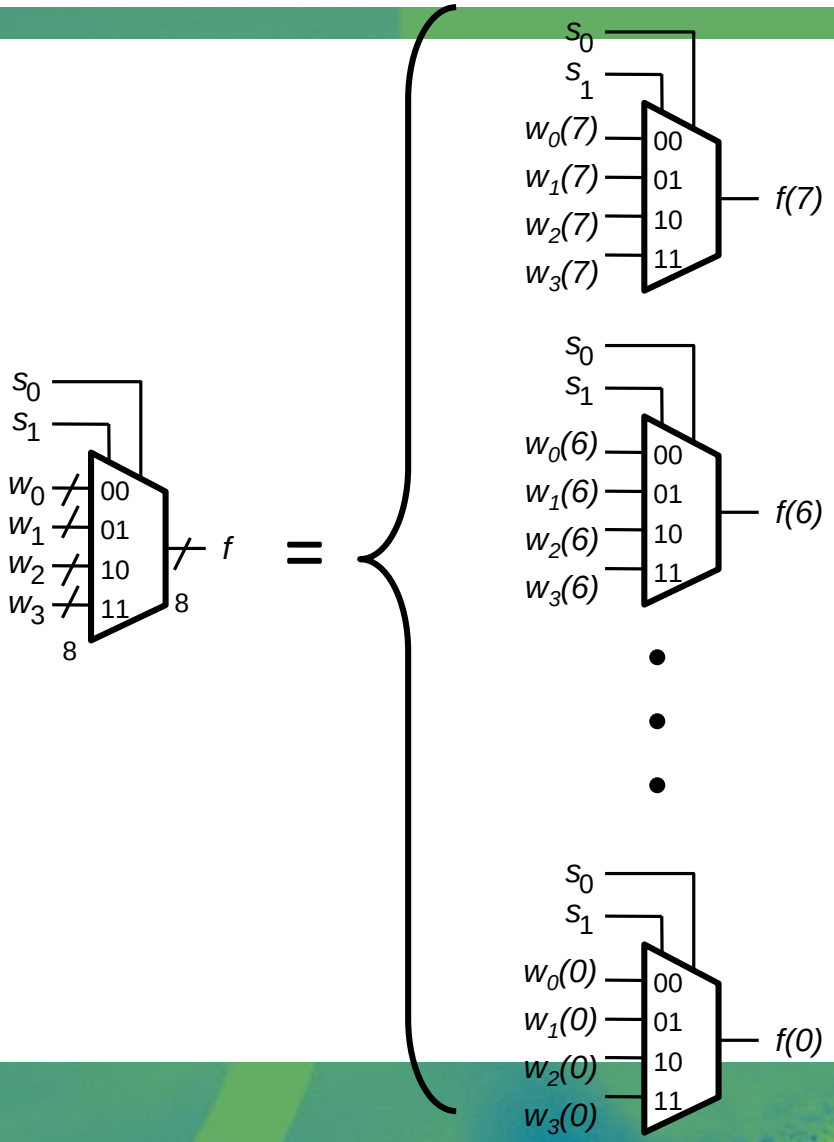
(a) Graphic symbol

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

(b) Truth table

- When drawing schematics, can draw **multi-bit** multiplexers
- Example: 4-to-1 (8 bit) multiplexer
  - 4 inputs (each 8 bits)
  - 1 output (8 bits)
  - 2 selection bits
- Can also have multi-bit 2-to-1 muxes, 16-to-1 muxes, etc.

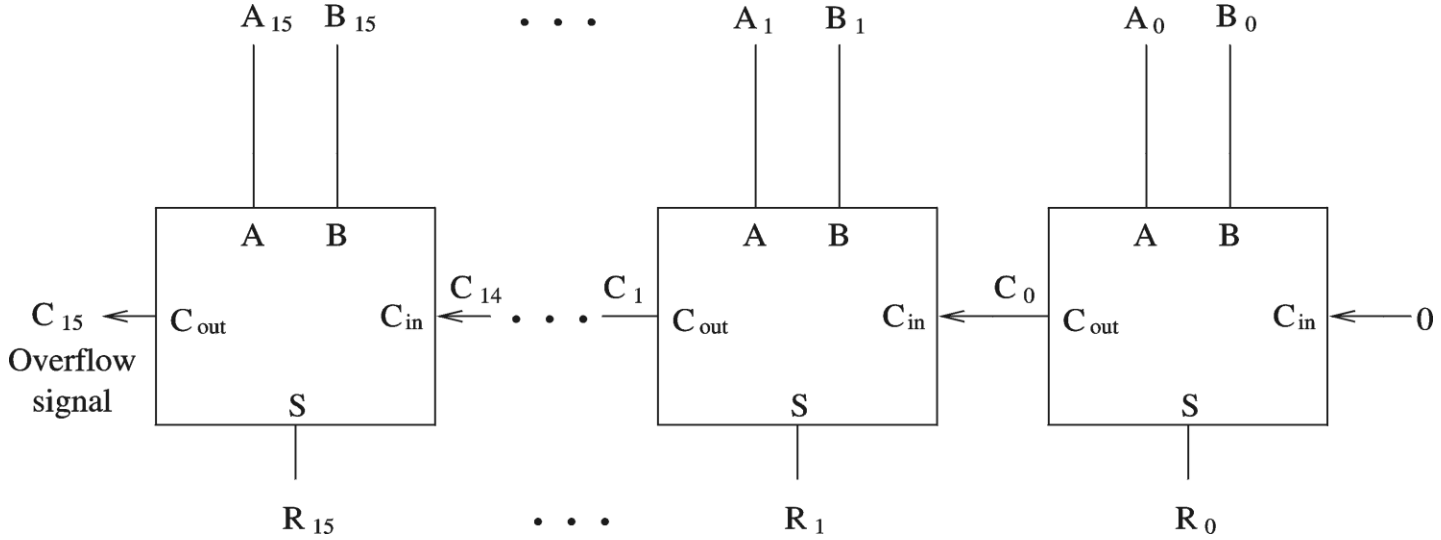
4-to-1 (8-bit) Multiplexer



A 4-to-1 (8-bit) multiplexer is composed of eight 4-to-1 (1-bit) multiplexers

Multi-Bit Ripple-Carry Adder

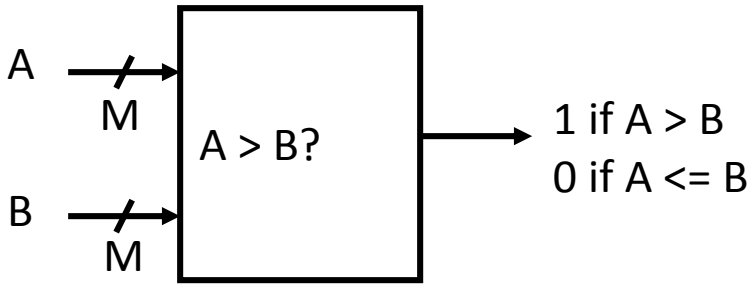
- A 16-bit ripple-carry adder is composed of 16 (1-bit) full adders
  - Inputs: 16-bit A, 16-bit B, 1-bit carryin (set to zero in the figure below)
  - Outputs: 16-bit sum R, 1-bit overflow



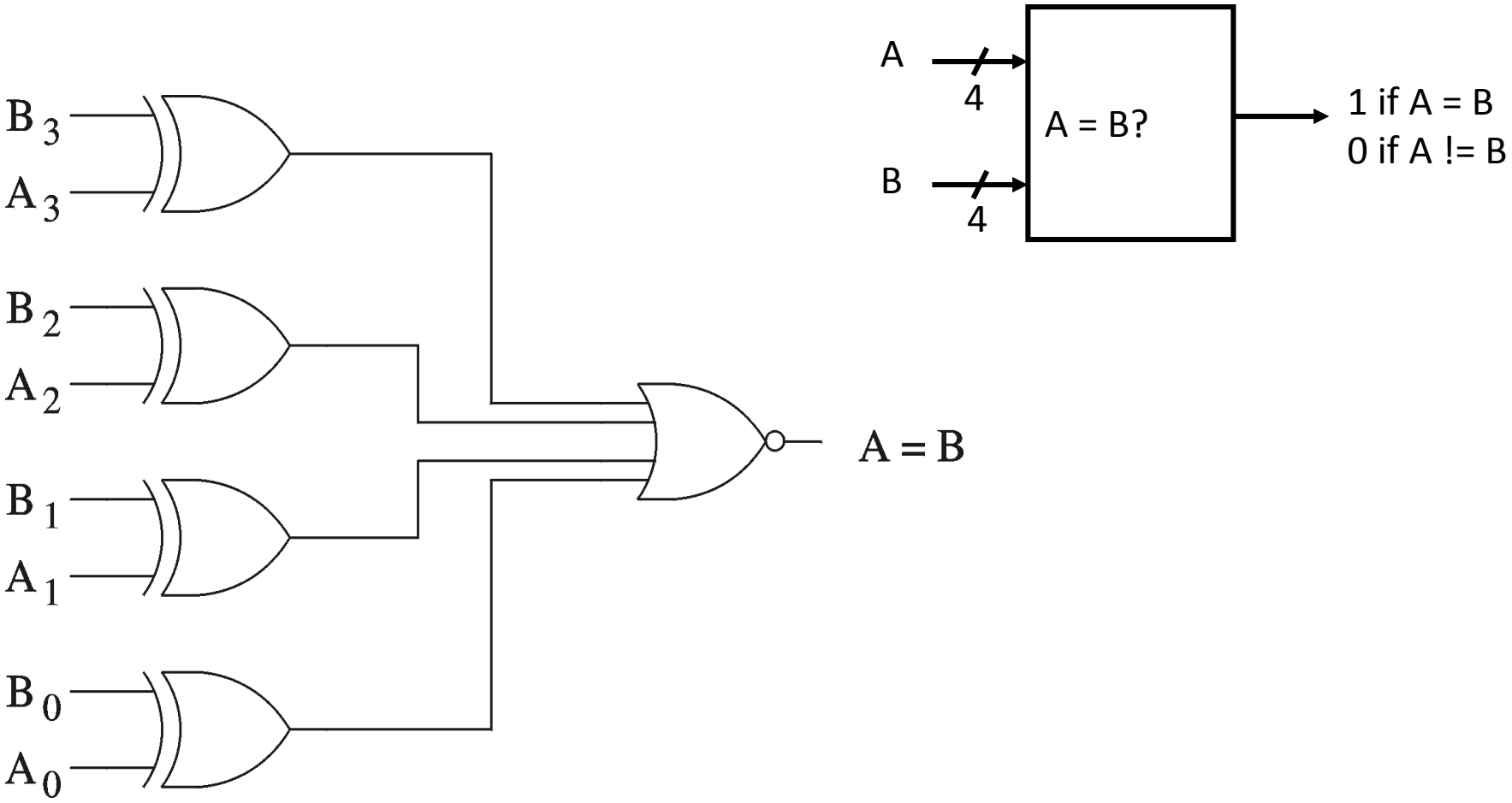
Called a ripple-carry adder because carry ripples from one full-adder to the next. Critical path is 16 full-adders.

Comparator

- Used two compare two M-bit numbers and produce a flag (M >1)
  - Inputs: M-bit input A, M-bit input B
  - Output: 1-bit output flag
    - 1 indicates condition is met
    - 0 indicates condition is not met
  - Can compare: >, >=, <, <=, =, etc.

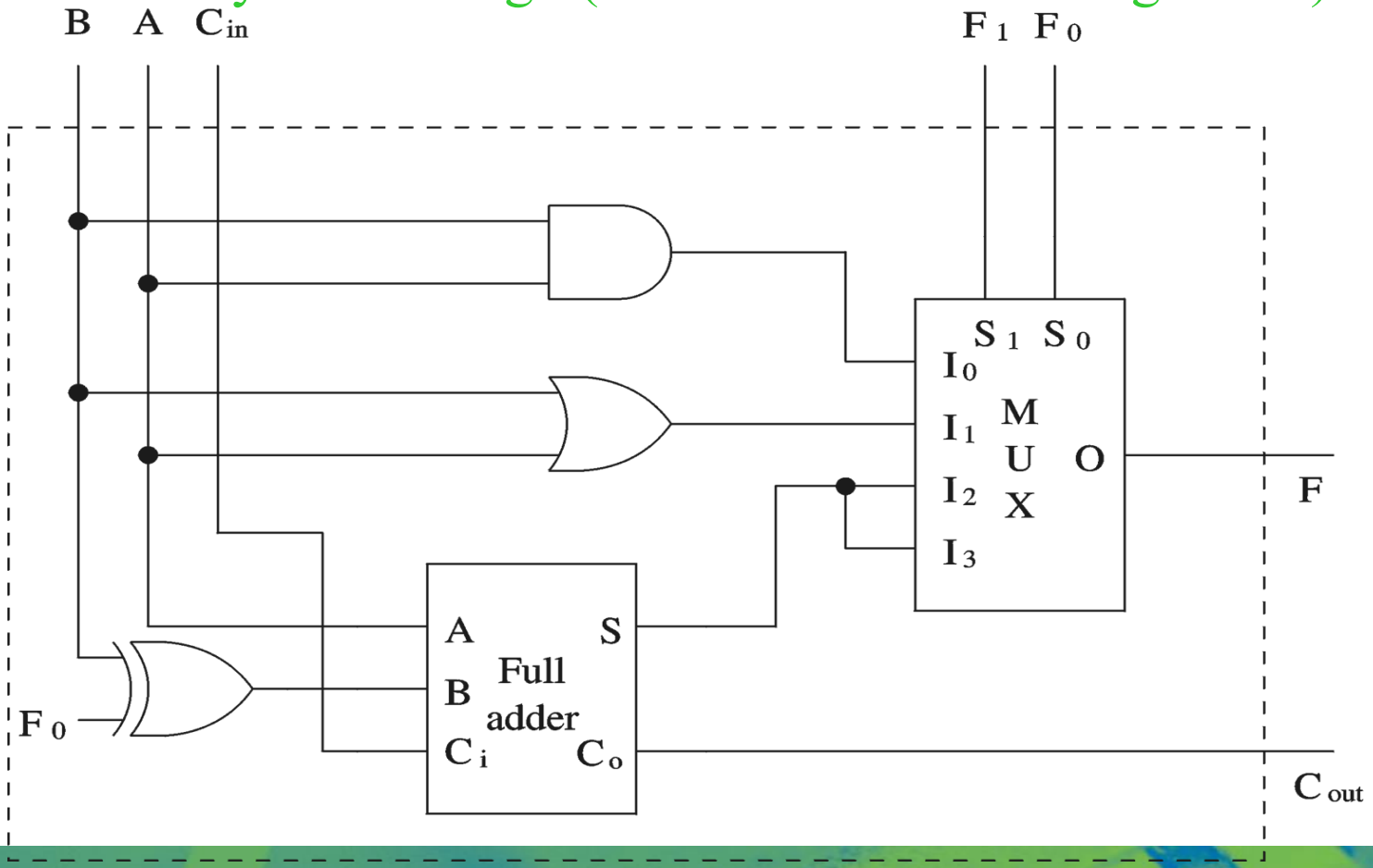


Example: 4-bit comparator (A = B)



Combinational Logic Example: ALU (1-bit)

- Preliminary ALU design (ALU = arithmetic + logic unit)



Sequential Logic Review

- Sequential Logic Building Blocks
  - Latches, Flip-Flops
- Sequential Logic Circuits
  - Registers, Shift Registers, Counters
  - Memory (RAM, ROM)
  - Simple Finite State Machines (Mealy, Moore)

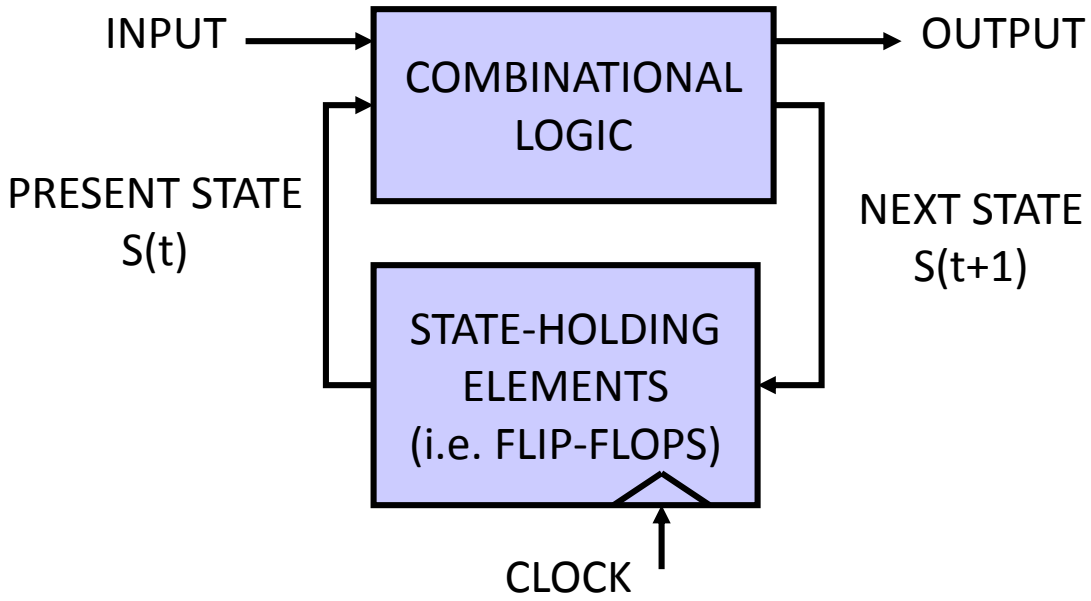
Introduction to Sequential Logic

- Output depends on current as well as past inputs
  - Depends on the history
  - Have "memory" property
- Sequential circuit consists of
    - Combinational circuit
    - Feedback circuit
  - Past input is encoded into a set of state variables
    - Uses feedback (to feed the state variables)
      - Simple feedback
      - Uses flip flops

Introduction …

- Main components of a typical synchronous sequential circuit

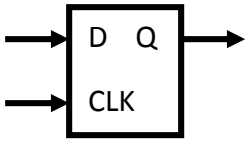
  (synchronous = uses a clock to keep circuits in lock step)

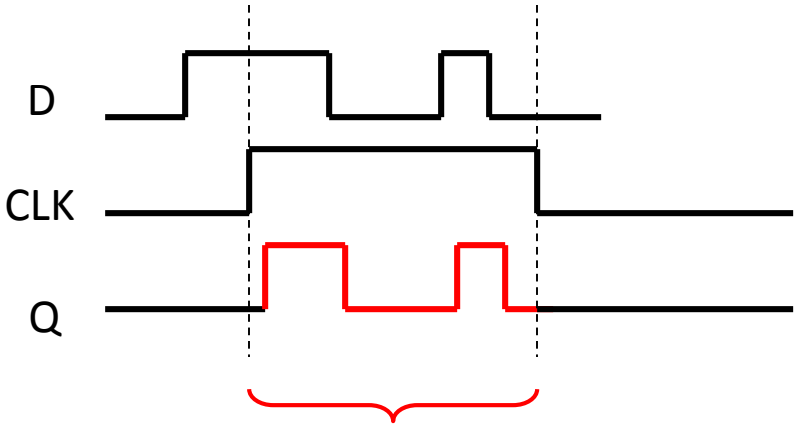


State-Holding Memory Elements

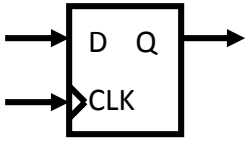
- Latch versus Flip Flop
  - Latches are level-sensitive: whenever clock is high, latch is transparent
  - Flip-flops are edge-sensitive: data passes through (i.e. data is sampled) only on a rising (or falling) edge of the clock
  - Latches cheaper to implement than flip-flops
  - Flip-flops are easier to design with than latches

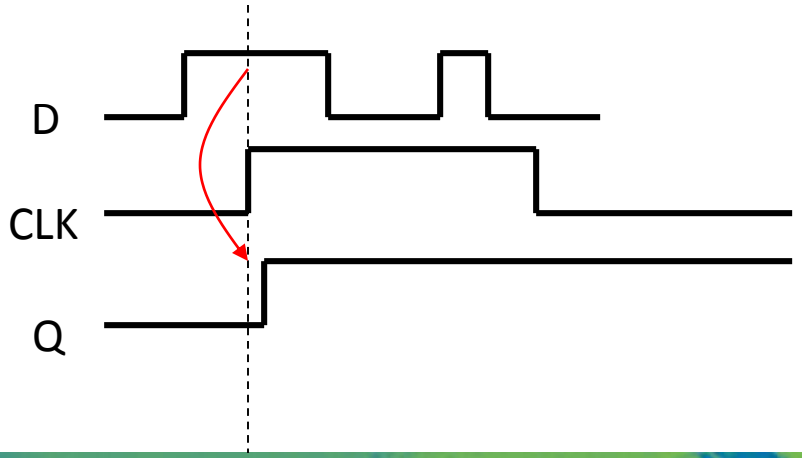
D Latch vs. D Flip-Flop





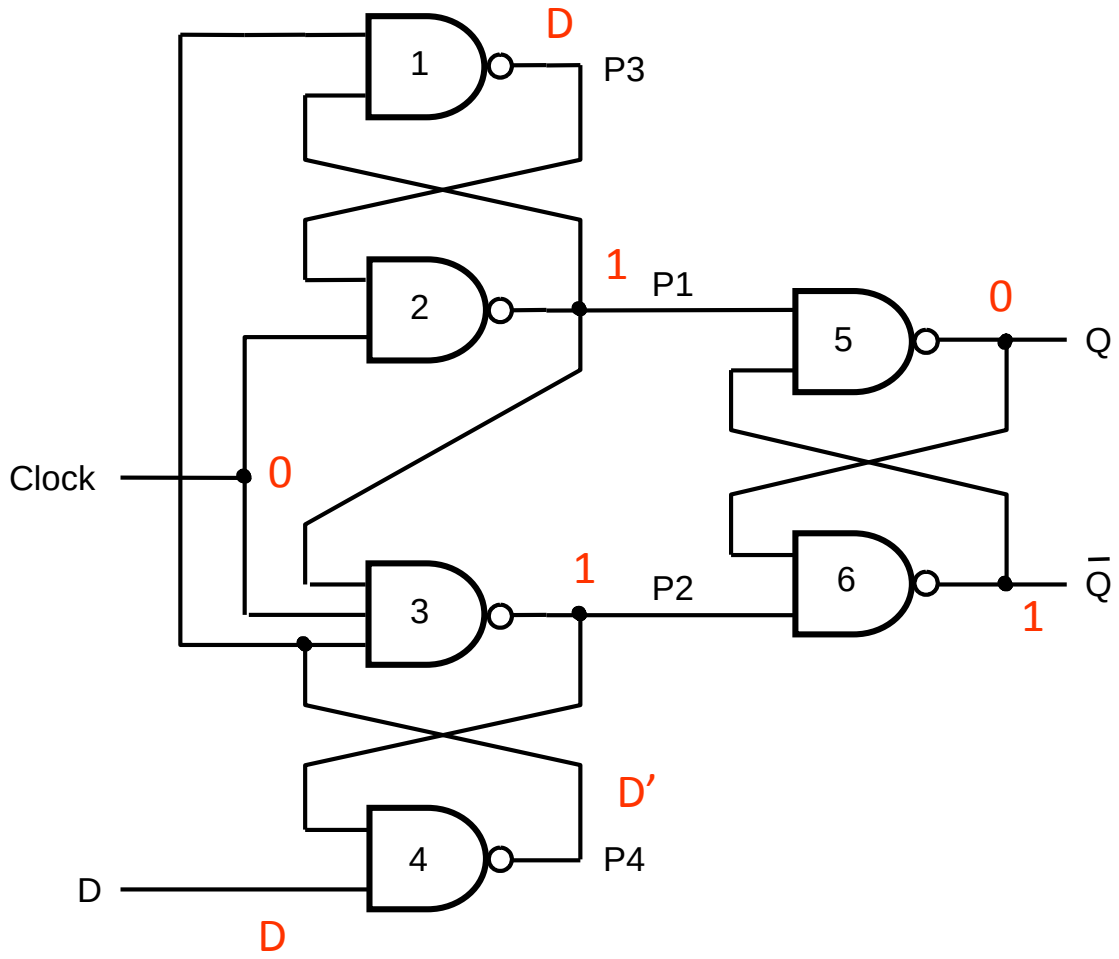
Latch transparent when clock is high



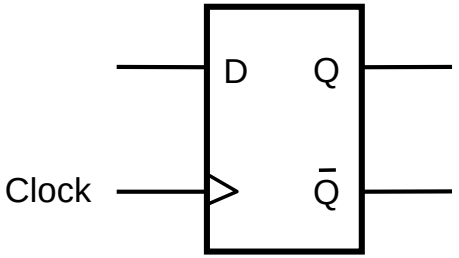


"Samples" D on rising edge of clock

D Flip-Flop (positive edge triggered)



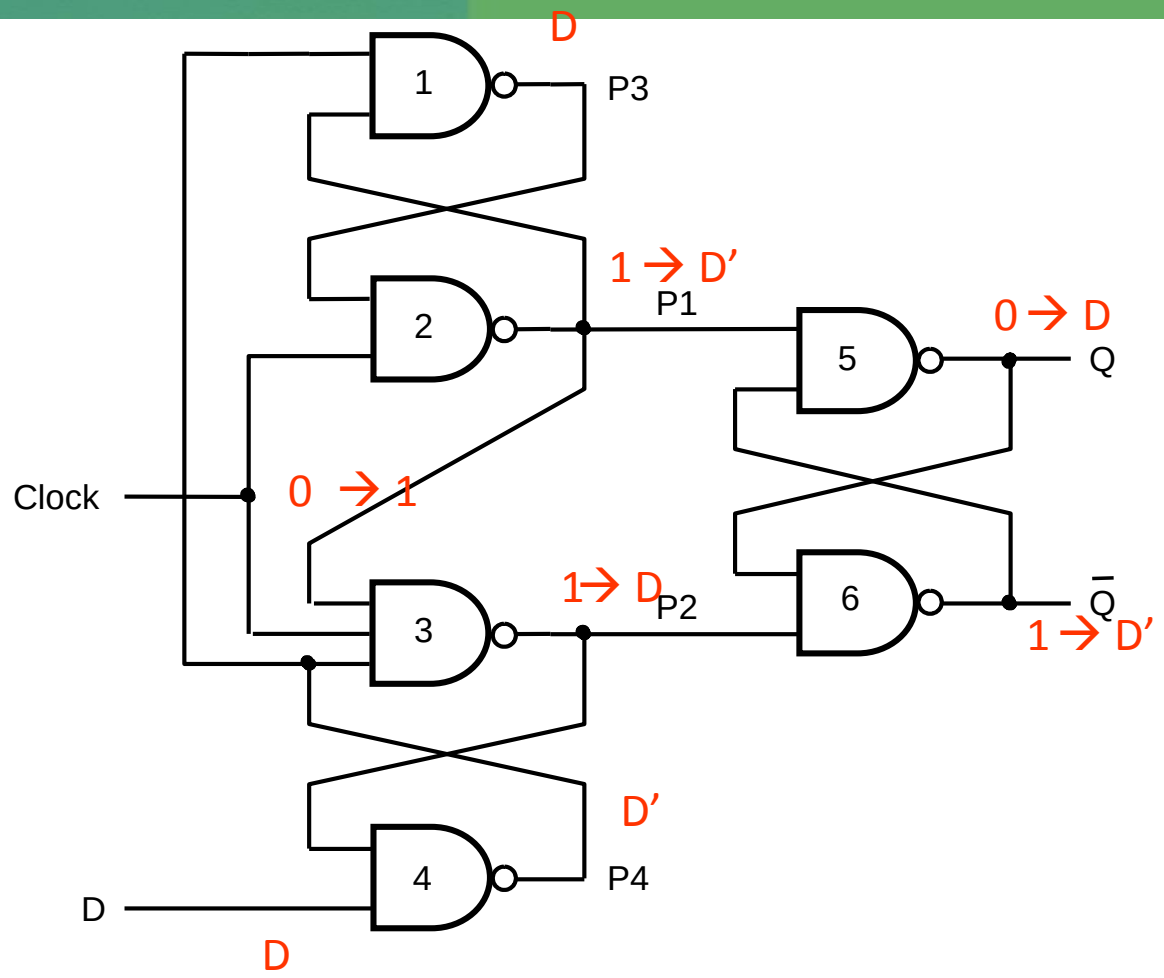
(a) Circuit



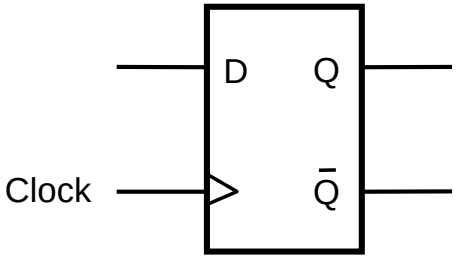
(b) Graphical symbol

NAND : one input 0, output always 1; all inputs 1 except A, output is A'

D Flip-Flop (positive edge triggered)



(a) Circuit

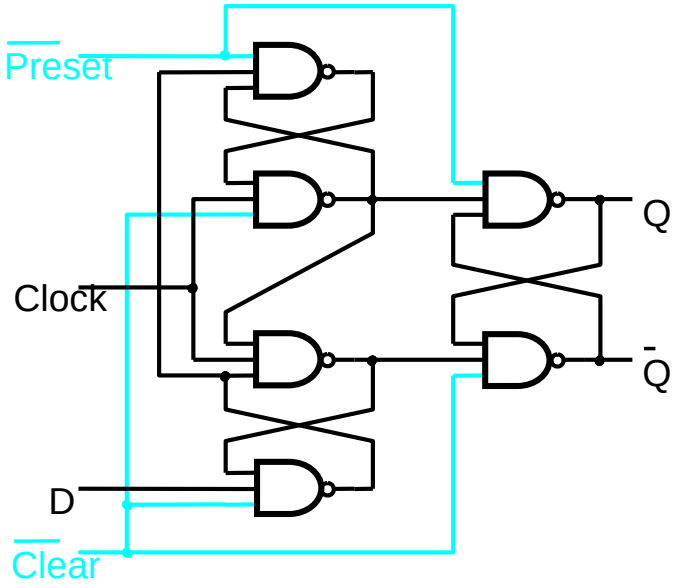


(b) Graphical symbol

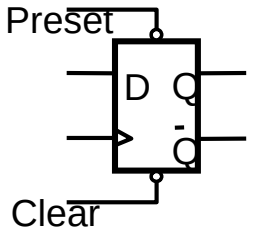
NAND : one input 0, output always 1; all inputs 1 except A, output is A'

D Flip-Flop with Asynchronous Preset and Clear

- Bubble on the symbol means "active-low"
  - When preset = 0, preset Q to 1
  - When preset = 1, do nothing
  - When clear = 0, clear Q to 0
  - When clear = 1, do nothing
- "Preset" and "Clear" also known as "Set" and "Reset" respectively
- In this circuit, preset and clear are asynchronous
  - Q changes immediately when preset or clear are active, regardless of clock

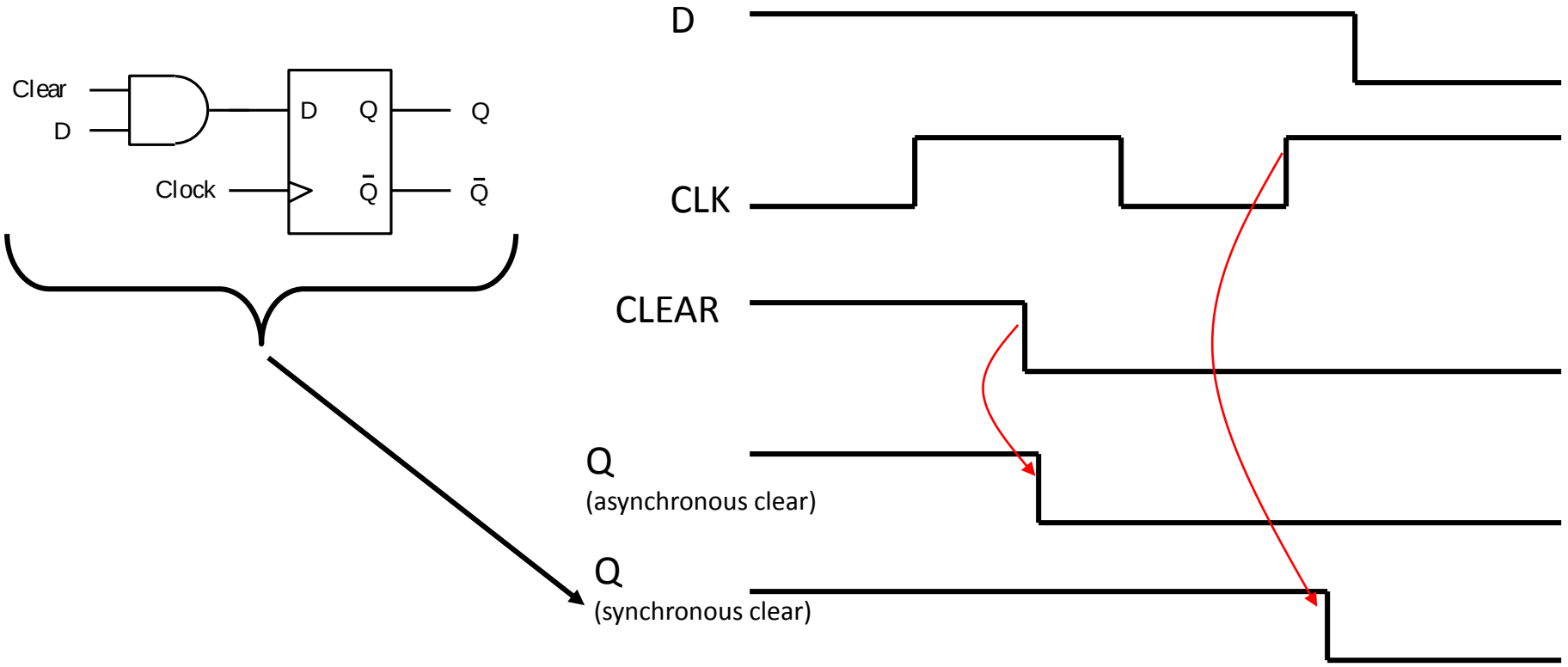


(a) Circuit



(b) Graphical symbol

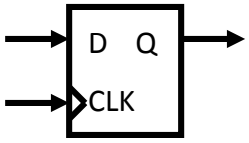
D Flip-Flop with Synchronous Clear



- Asynchronous active-low clear: Q immediately clears to 0
- Synchronous active-low clear: Q clears to 0 on rising-edge of clock

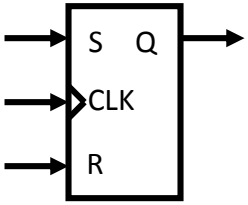
Other Types of Flip-Flops

D Flip-Flop



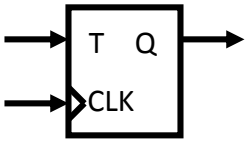
D	Q(t+1)
0	0
1	1

Set-Reset (SR) Flip-Flop



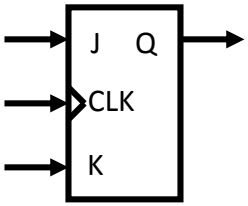
S	R	Q(t+1)
0	0	Q(t)
0	1	0
1	0	1
1	1	not allowed

Toggle (T) Flip-Flop



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

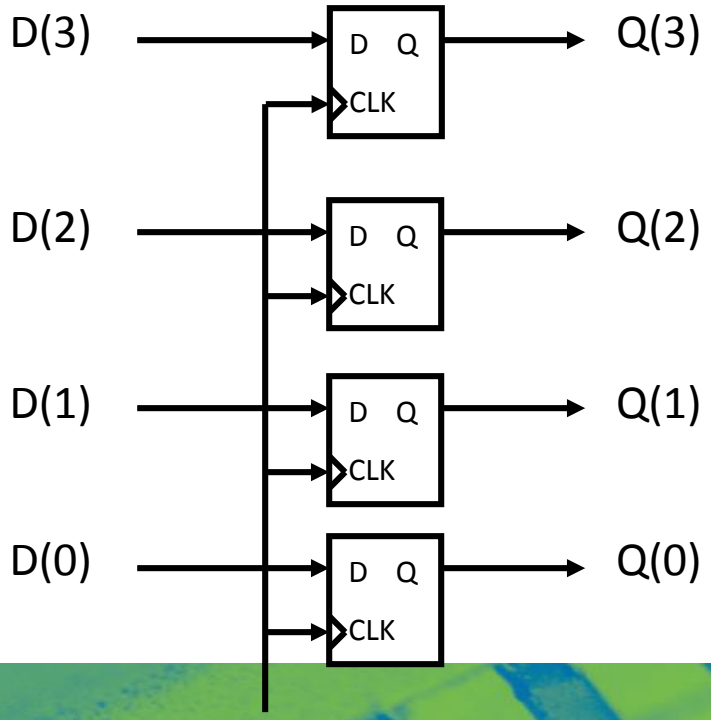
JK Flip-Flop



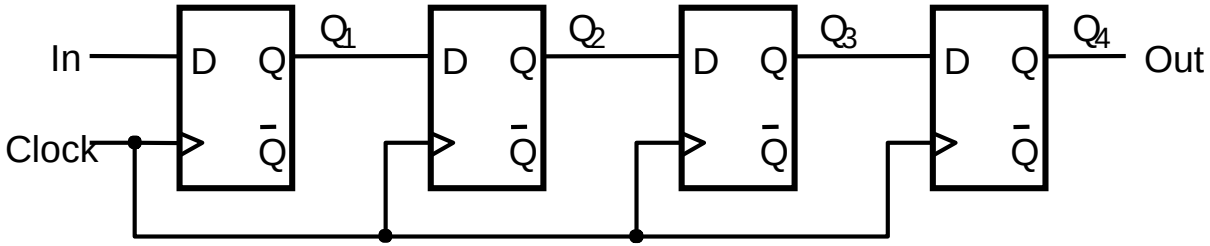
J	K	Q(t+1)
0	0	Q(t)
0	1	0
1	0	1
1	1	$\overline{Q(t)}$

Sequential Logic Circuits

- Register
  - In typical nomenclature, a register is a name for a collection of flip-flops used to hold a bus (i.e. std_logic_vector)



Shift Register

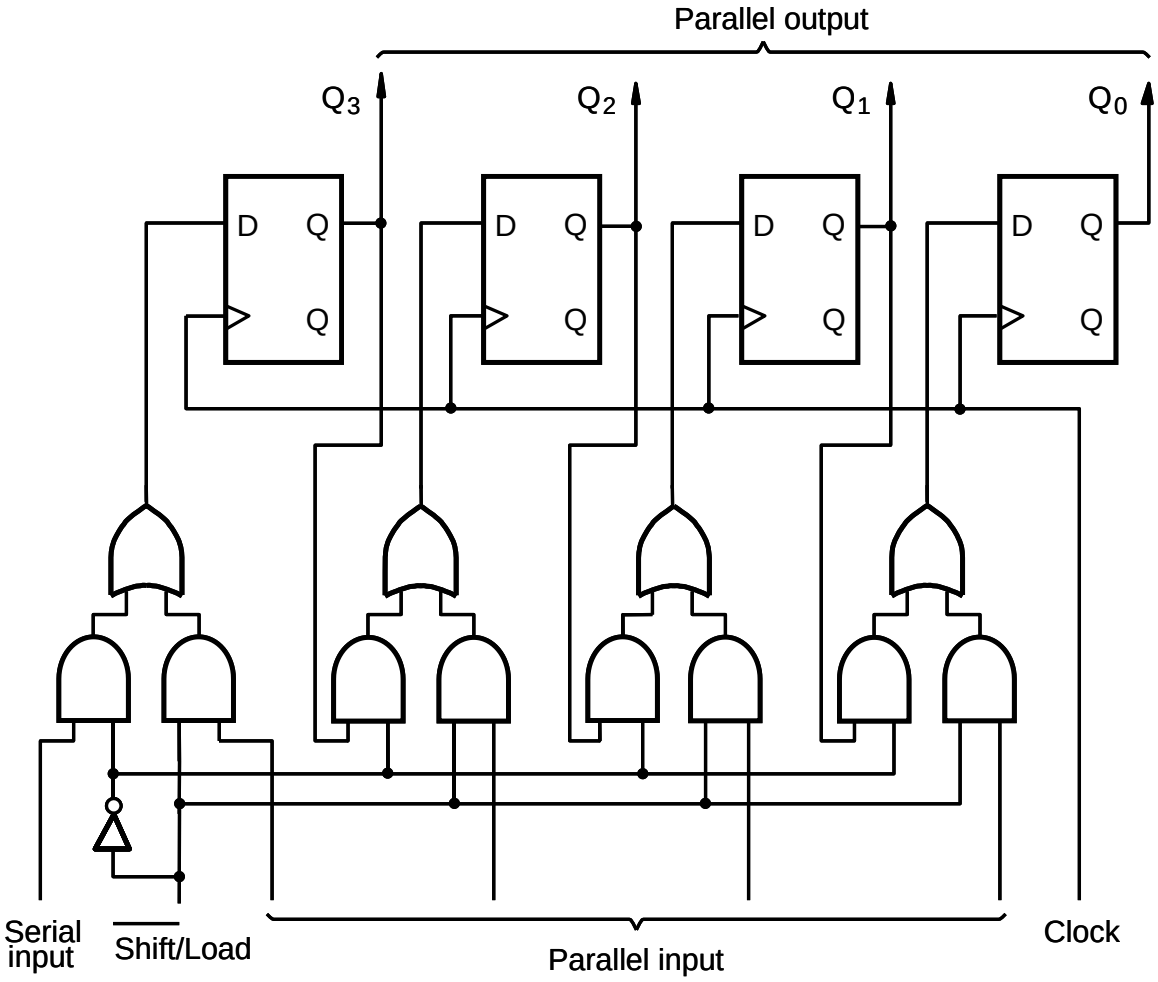


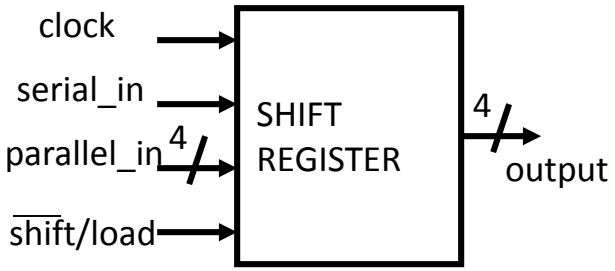
(a) Circuit

	In	Q_1	Q_2	Q_3	Q_4 = Out
t_0	1	0	0	0	0
t_1	0	1	0	0	0
t_2	1	0	1	0	0
t_3	1	1	0	1	0
t_4	1	1	1	0	1
t_5	0	1	1	1	0
t_6	0	0	1	1	1
t_7	0	0	0	1	1

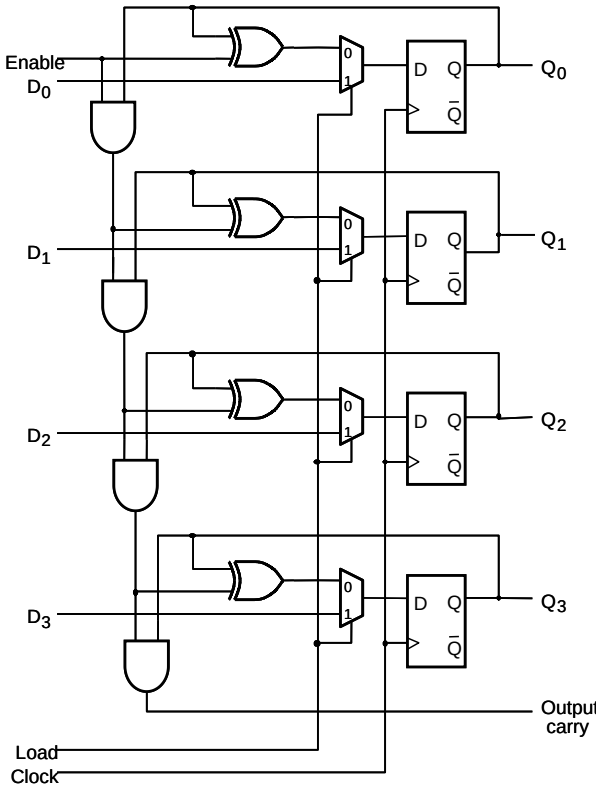
(b) A sample sequence

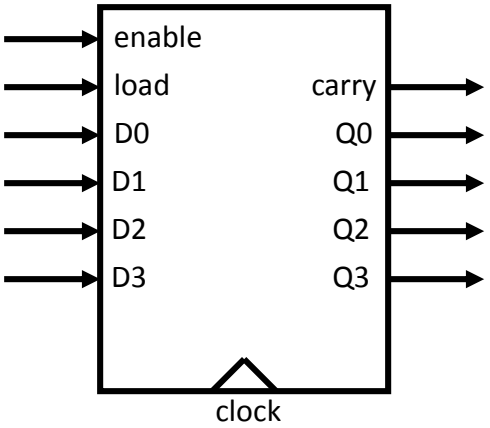
Parallel Access Shift Register





Synchronous Up Counter

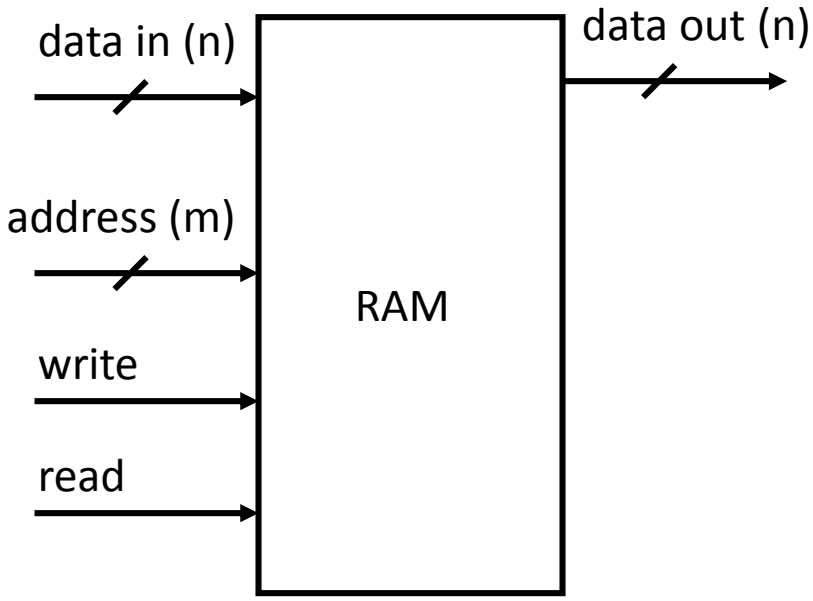




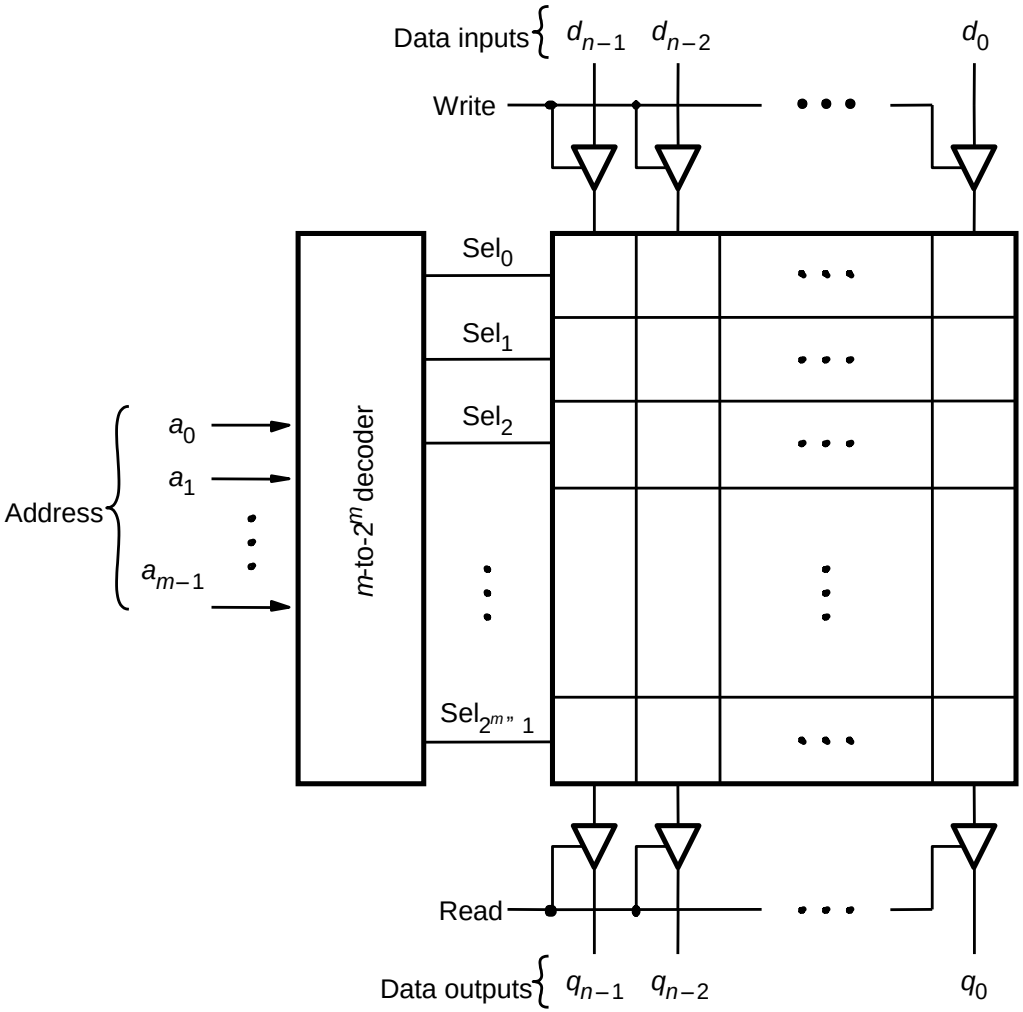
- Enable (synchronous): when high enables the counter, when low counter holds its value
- Load (synchronous) : when load = 1, load the desired value into the counter
- Output carry: indicates when the counter "rolls over"
- D3 downto D0, Q3 downto Q0 is how to interpret MSB to LSB

Memories (Random Access Memory)

- More efficient than registers for storing large amounts of data
- Can read and write to RAM
- Addressable memory
- Can be synchronous (with clock) or asynchronous (no clock)
- SRAM dimensions are:
  - (number of words) x (bits per word) SRAM
- Address is m bits, data is n bits
  - 2^m x n-bit RAM
- Example: address is 5 bits, data is 8 bits
  - 32 x 8-bit RAM
- Write
  - Data_in and address are stable
  - Assert write signal (then de-assert)
- Read
  - Address is stable
  - Assert read signal
  - Data_out is valid

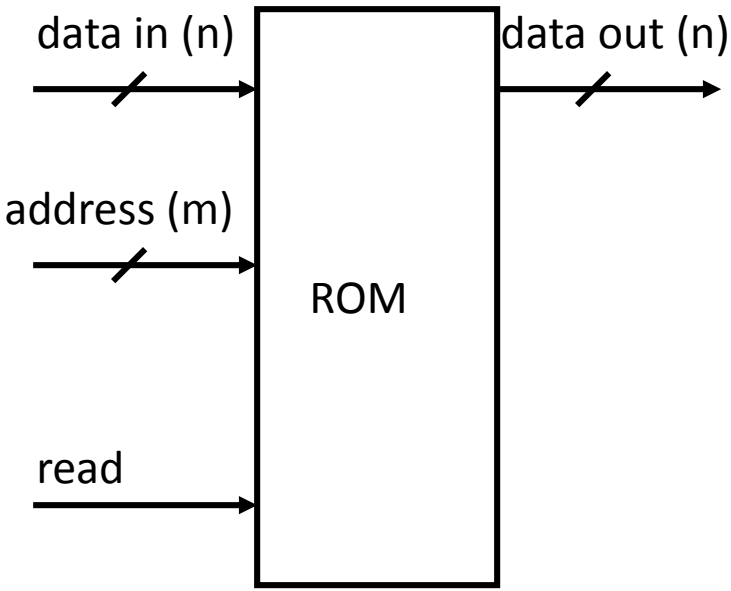


Random Access Memory (RAM) …

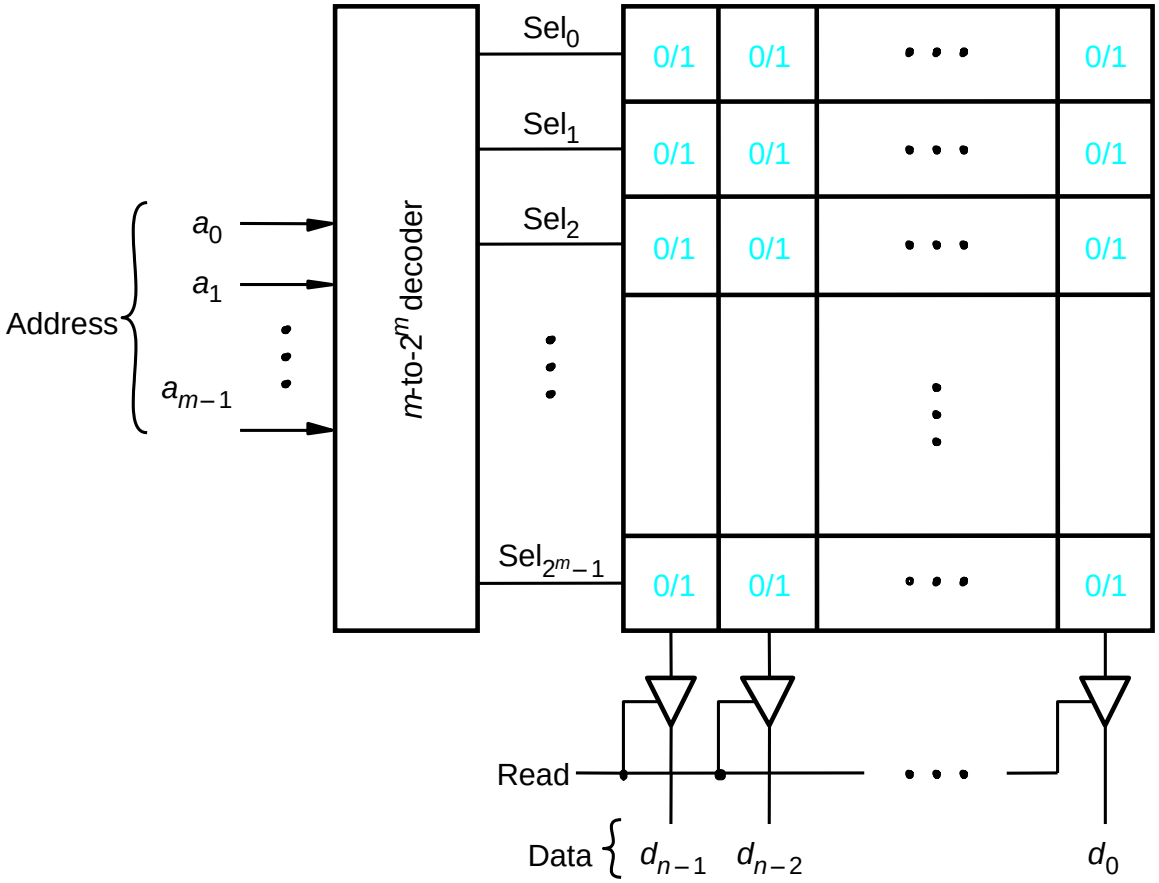


Read Only Memory (ROM)

- Similar to RAM except read only
- Addressable memory
- Can be synchronous (with clock) or asynchronous (no clock)



Read-Only Memory (ROM) …



Finite State Machines (FSMs)

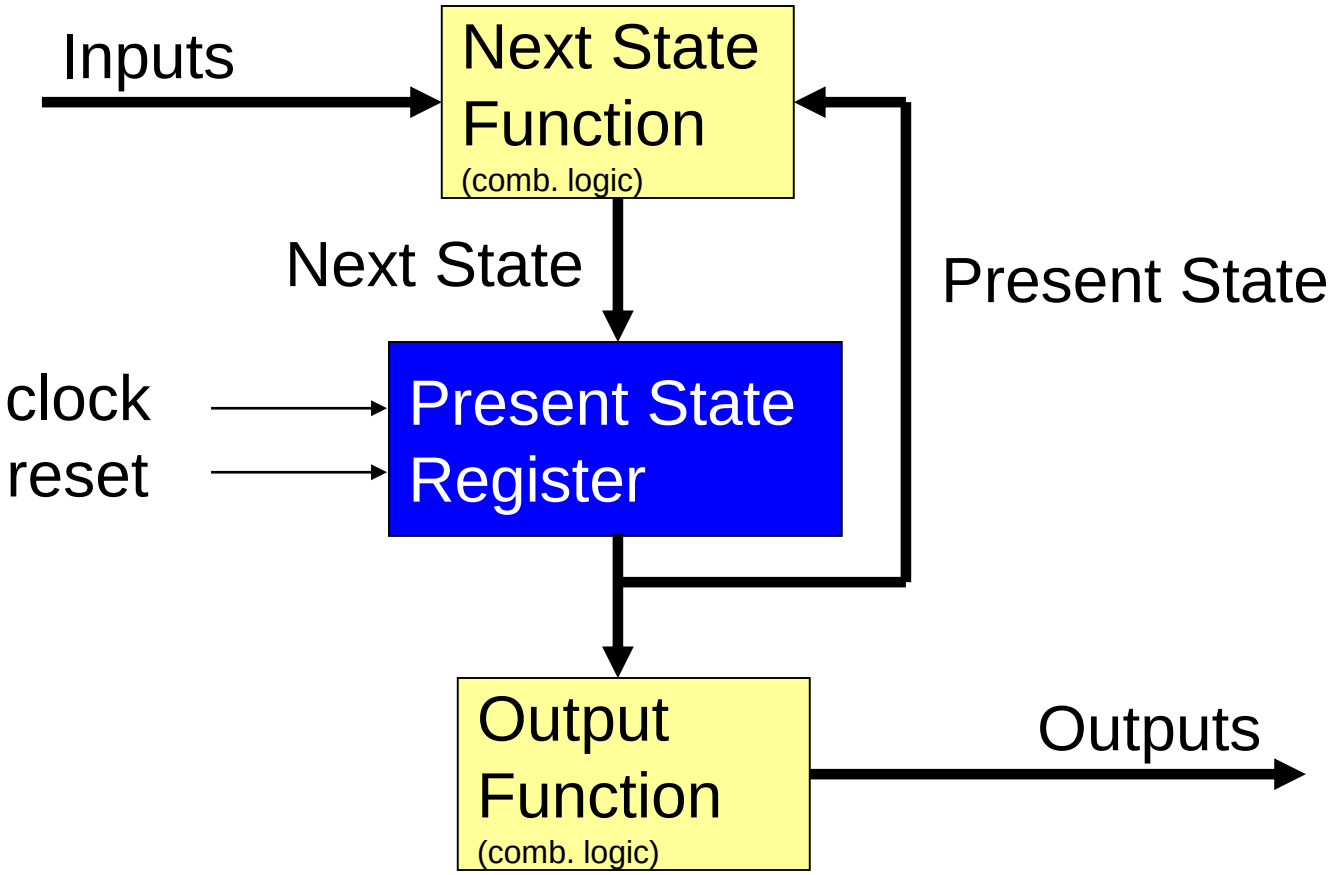
- Any Circuit with Memory Is a Finite State Machine
  - Even computers can be viewed as huge FSMs
- Design of FSMs Involves
  - Defining states
  - Defining transitions between states
  - Optimization / minimization
- Above Approach Is Practical for Simple FSMs Only

Mealy vs. Moore State Machines

- Finite State Machines (FSM) are of two types:
- Moore Machines
  - Next State = Function(Input, Present State)
  - Output = Function(Present State)
- Mealy Machines
  - Next State = Function(Input, Present State)
  - Output = Function(Input, Present State)

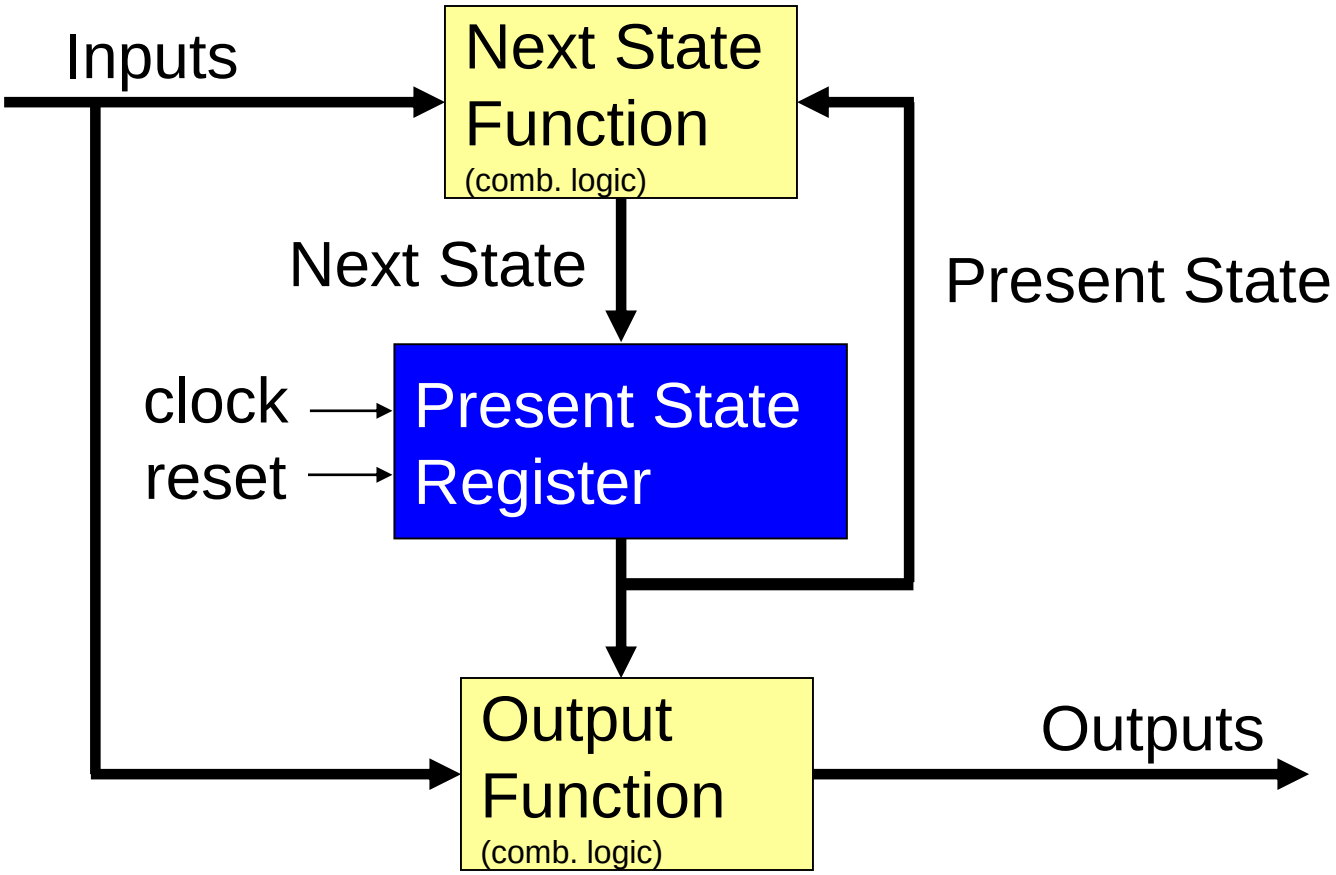
Moore FSM

- Output Is a Function of a Present State Only

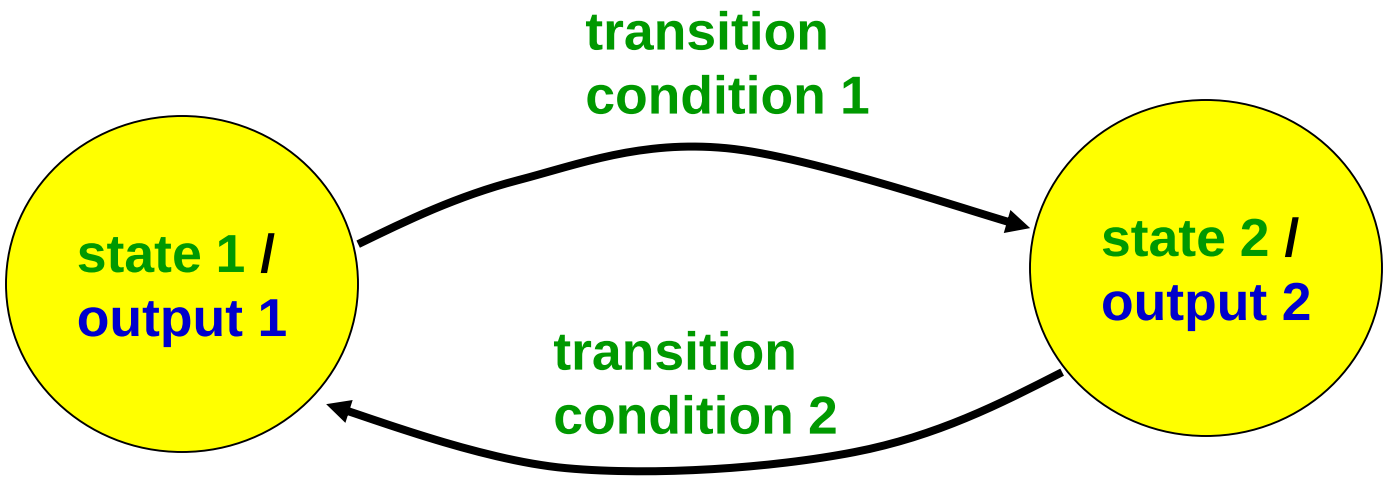


Mealy FSM

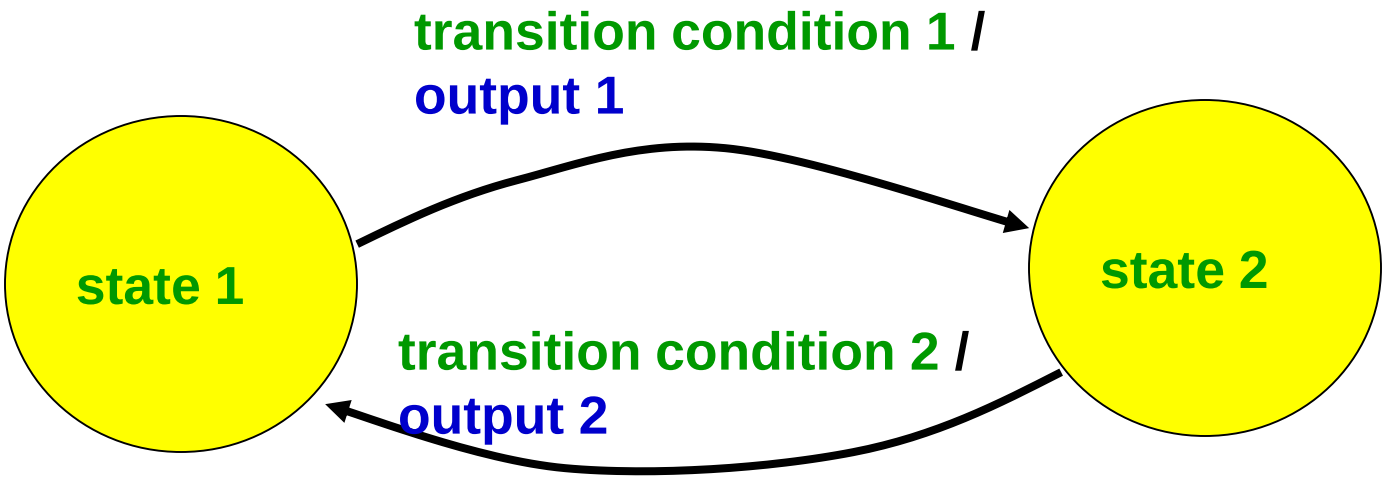
- Output Is a Function of a Present State and Inputs



Moore Machine



Mealy Machine



Moore vs. Mealy FSM (1)

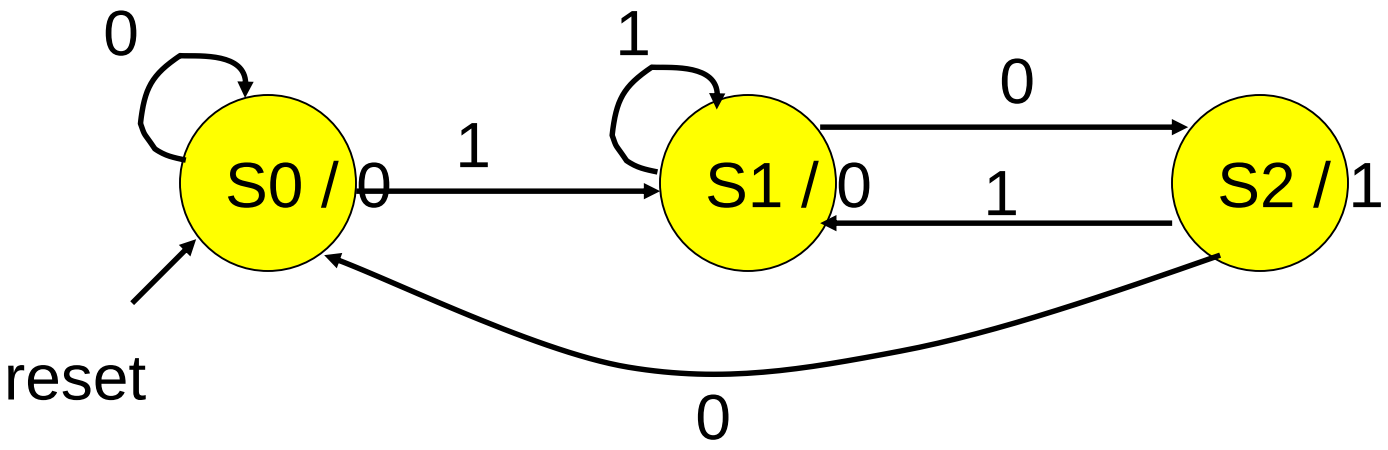
- Moore and Mealy FSMs can be functionally equivalent
  - Equivalent Mealy FSM can be derived from Moore FSM and vice versa
- Mealy FSM Has Richer Description and Usually Requires Smaller Number of States
  - Smaller circuit area

Moore vs. Mealy FSM (2)

- Mealy FSM computes outputs as soon as inputs change
  - Mealy FSM responds one clock cycle sooner than equivalent Moore FSM
- Moore FSM Has no combinational path between inputs and outputs
  - Moore FSM is more likely to have a shorter critical path
  - Moore outputs synchronized with clock; Mealy outputs may not be (may have race conditions, timing issues, etc.)

Moore FSM - Example 1

- Moore FSM that Recognizes Sequence “10”



Meaning of states:

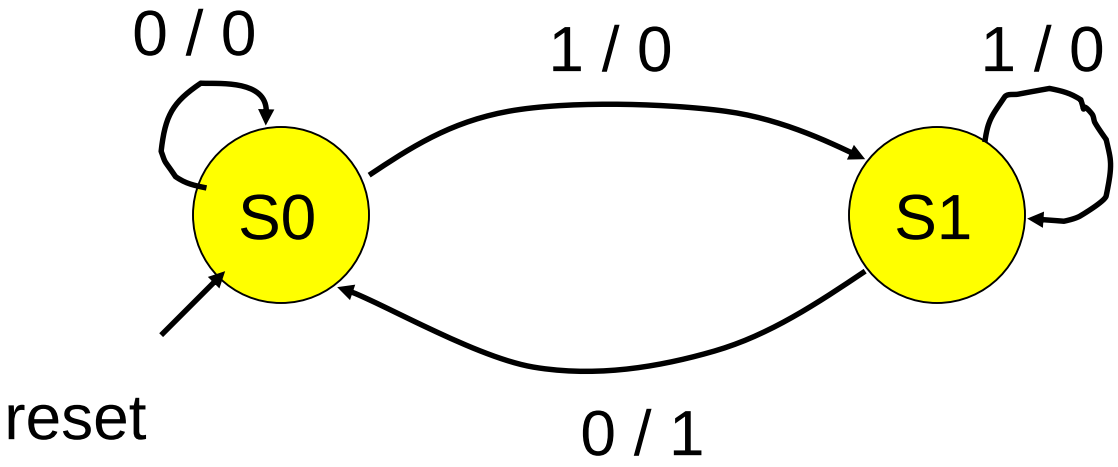
S0: No elements of the sequence observed

S1: “1” observed

S2: “10” observed

Mealy FSM - Example 1

- Mealy FSM that Recognizes Sequence "10"

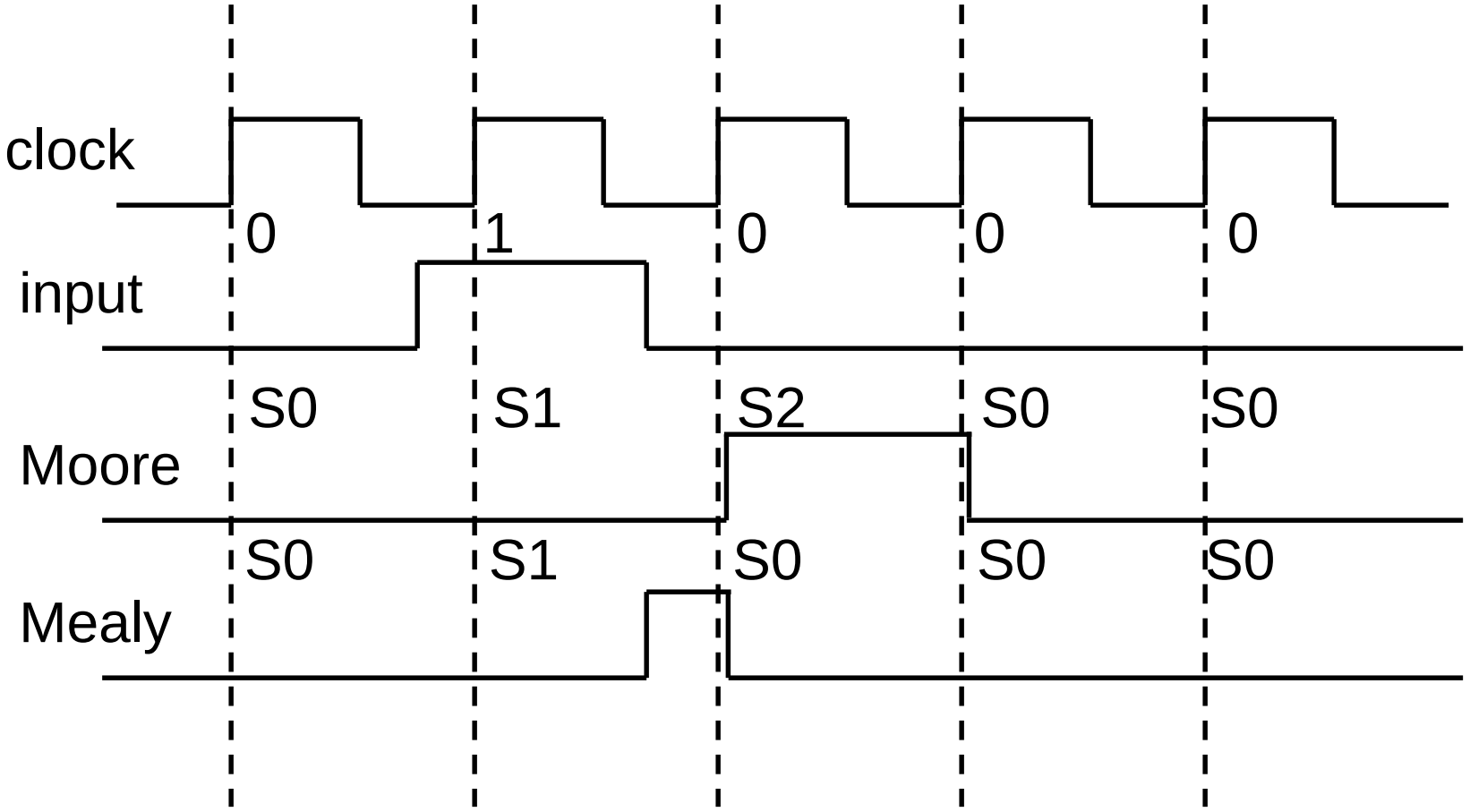


Meaning of states:

S0: No elements of the sequence observed

S1: "1" observed

Moore & Mealy FSMs – Example 1



FSM Limitations

- Simple finite state machines (those expressed using state diagrams and state tables) good only for simple designs
  - Many inputs and many outputs make it awkward to draw state machines
  - Often only one input affects the next change of state
  - Most outputs remain the same from state to state
- Instead use algorithmic state machines (ASM)

Next Session

- I-CIRCUIT DESIGN
  - 1 Introduction
    - 1.1 About VHDL
    - 1.2 Design Flow
    - 1.3 EDA Tools
    - 1.4 Translation of VHDL Code into a Circuit
    - 1.5 Design Examples