Inf2C - Computer Systems
Lecture 10-11
Logic Design

Vijay Nagarajan

School of Informatics

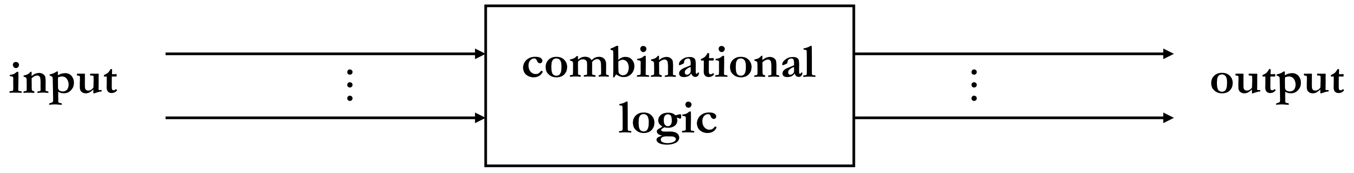
University of Edinburgh

Logic design overview

Binary digital logic circuits:

– Two voltage levels (ground and supply voltage) for 0 and 1
  - Built from transistors used as on/off switches
  - Digital logic with more than two states is not practical

Combinational logic: output depends only on the current inputs (no memory of past inputs)

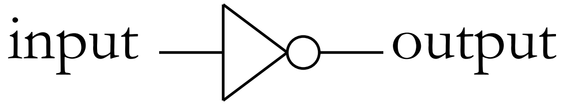


Sequential logic: output depends on the current inputs as well as (some) previous inputs → requires "memory"

Combinational logic circuits

- Inverter (or NOT gate): 1 input and 1 output

  "invert the input signal"



IN	OUT
0	1
1	0

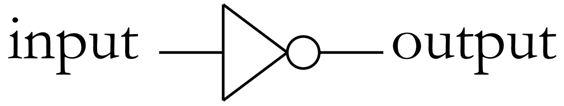
Truth table

$\text{OUT} = \overline{\text{IN}}$

Combinational logic circuits

- Inverter (or NOT gate): 1 input and 1 output

  "invert the input signal"

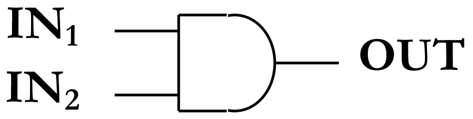


IN	OUT
0	1
1	0

$\mathbf{OUT} = \overline{\mathbf{IN}}$

- AND gate: 2 inputs and 1 output

  "output 1 only if both inputs are 1"

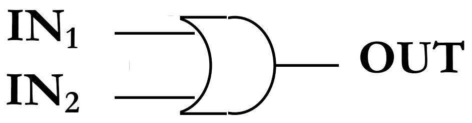


IN_1	IN_2	OUT
0	0	0
0	1	0
1	0	0
1	1	1

$\mathbf{OUT} = \mathbf{IN_1 \,.\, IN_2}$

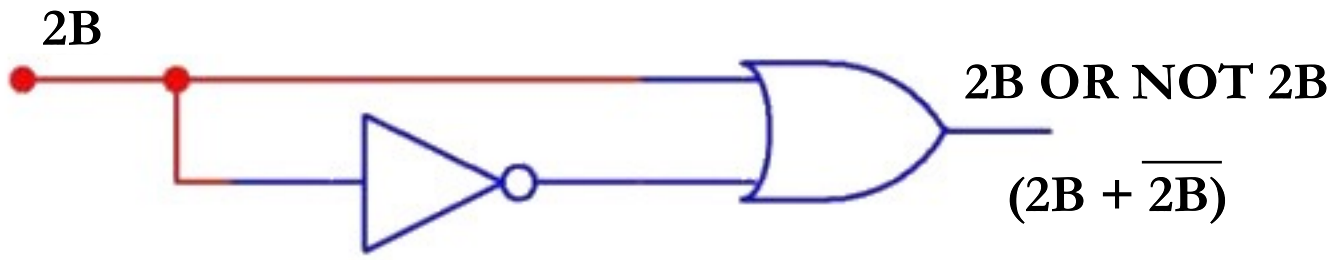
Combinational logic circuits

- OR gate: "output 1 if at least one input is 1"



IN_1	IN_2	OUT
0	0	0
0	1	1
1	0	1
1	1	1

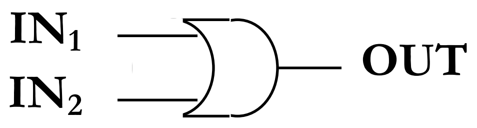
$OUT = IN_1 + IN_2$



$(2B + \overline{2B})$

Combinational logic circuits

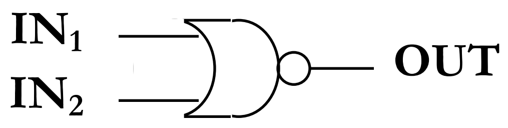
- OR gate: "output 1 if at least one input is 1"



IN$_1$	IN$_2$	OUT
0	0	0
0	1	1
1	0	1
1	1	1

$OUT = IN_1 + IN_2$

- NOR gate: "output 1 if no input is 1" (NOT OR)

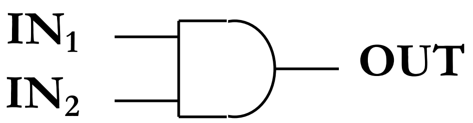


IN$_1$	IN$_2$	OUT
0	0	1
0	1	0
1	0	0
1	1	0

$OUT = \overline{IN_1 + IN_2}$

Combinational logic circuits

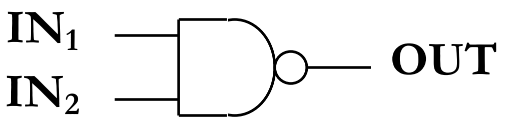
- AND gate: "output 1 if both inputs are 1"



IN_1	IN_2	OUT
0	0	0
0	1	0
1	0	0
1	1	1

$$OUT = IN_1 . IN_2$$

- NAND gate: "output 1 if both inputs are <u>not</u> 1" (NOT AND)

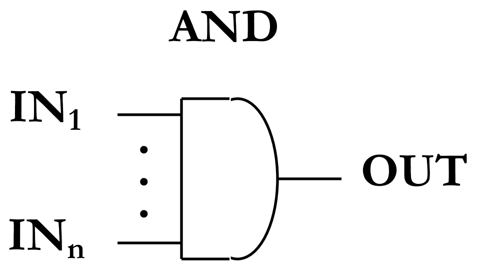


IN_1	IN_2	OUT
0	0	1
0	1	1
1	0	1
1	1	0

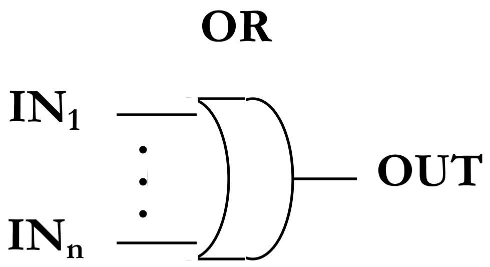
$$OUT = \overline{IN_1 . IN_2}$$

Combinational logic circuits

- Multiple-input gates:



OUT = 1 if <u>all</u> IN_i=1



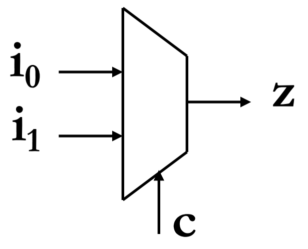
OUT = 1 if <u>any</u> IN_i=1

Combinational logic circuits

- Functional completeness:
  - Set of gates that is sufficient to express any boolean function
- Examples of functionally-complete sets of gates:
  - AND + OR + NOT
  - NAND
  - NOR

Multiplexer (mux)

- Multiplexer: a circuit for selecting one of multiple inputs



$$z = \begin{cases} i_0 & \text{iff } \mathbf{c=0} \\ i_1 & \text{iff } \mathbf{c=1} \end{cases}$$

c	i_0	i_1	z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

$$z = \bar{c}.i_0.\bar{i}_1 + \bar{c}.i_0.i_1 + c.\bar{i}_0.i_1 + c.i_0.i_1$$

$$= \bar{c}.i_0.(\bar{i}_1 + i_1) + c.(\bar{i}_0 + i_0).i_1$$

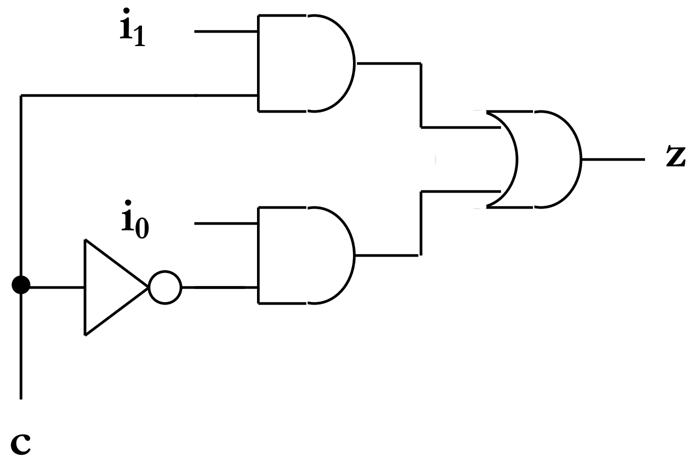
$$\boxed{= \bar{c}.i_0 + c.i_1}$$

minimized

"sum of products form"

A multiplexer implementation

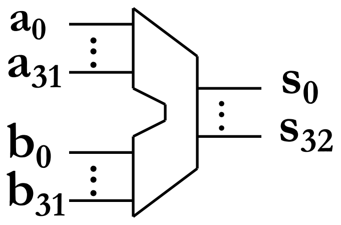
- Sum of products form: $\mathbf{i_1.c + i_0.\overline{c}}$
  - Can be implemented with 1 inverter, 2 AND gates & 1 OR gate:



- Sum of products is not practical for circuits with large number of inputs (n)
  - The number of possible products can be proportional to 2^n

Arithmetic circuits

- 32-bit adder

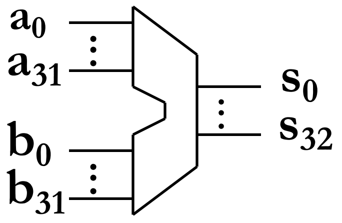


64 inputs → too complex for sum of products

- Idea: modularize!
  – Design a generic 1-bit adder block
  – Replicate it N number of times for an N-bit adder

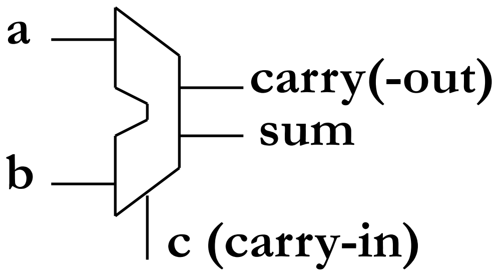
Arithmetic circuits

- 32-bit adder



64 inputs → too complex for sum of products

- Full adder:



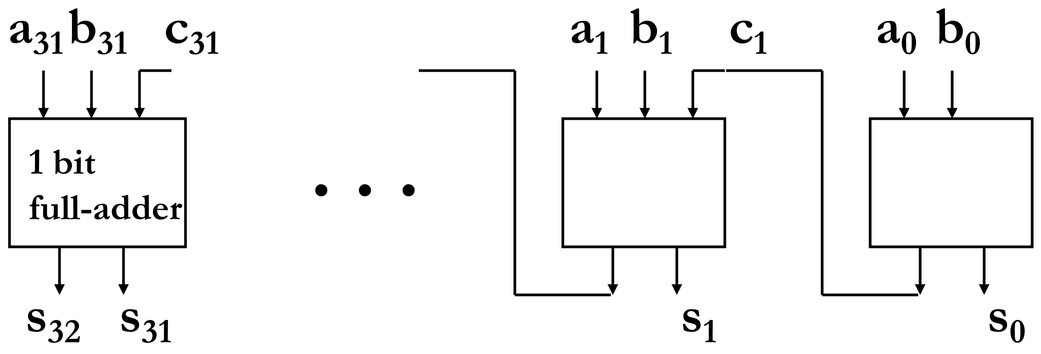
sum = $\overline{a}.\overline{b}.c + \overline{a}.b.\overline{c} + a.\overline{b}.\overline{c} + a.b.c$

carry = $b.c + a.c + a.b$

a	b	c	carry	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Ripple carry adder

- 32-bit adder: chain of 32 full adders



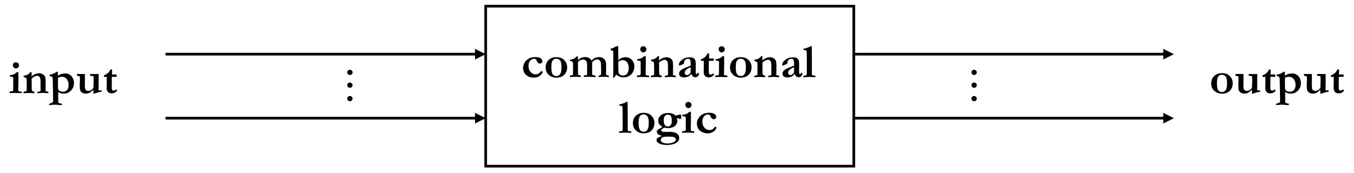
  – Carry bits c_i are computed in sequence $c_1, c_2, \ldots, c_{32}$ (where $c_{32} = s_{32}$), as c_i depends on c_{i-1}
  – Since sum bits s_i also depend on c_i, they too are computed in sequence

Propagation delays

- Propagation delay = time delay between input signal change and output signal change at the other end
- Delay depends on:
  1. technology (transistor parameters, wire capacitance, etc.)
  2. delay through each gate (function of gate type)
  3. number of gates driven by a gate's output (**fan out**)
- e.g.: 2-input mux: NOT → AND → OR ➔ 3 gate delays. Fast!
- What's the delay of a 32-bit ripple carry adder?
  - 65 gate delays → slow
  - AND2 + OR3 for each of 31 carries to propagate; followed by NOT + AND3 + OR4 for S_{31}

Combinational logic: summary

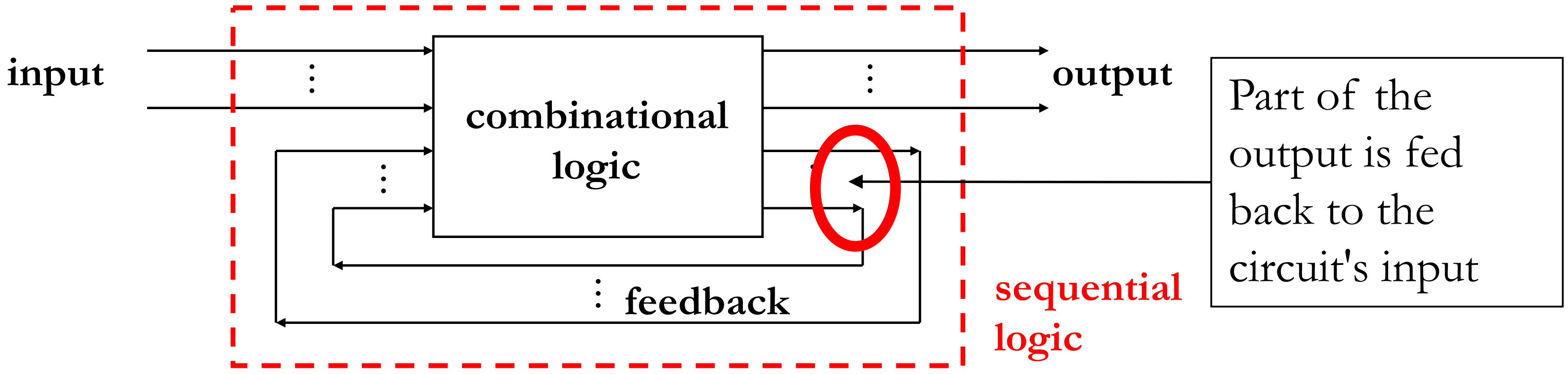
Combinational logic: output depends only on the current inputs



– Does not "remember" previous inputs or outputs

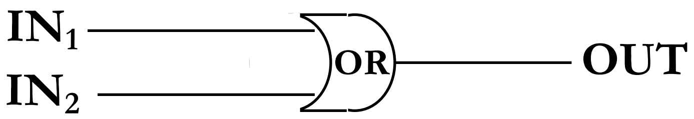
Memory is needed for more complex operations

Sequential logic circuits



- Output depends on current and (some of the) past inputs
  – The circuit has memory
- Sequences of inputs generate sequences of outputs ⇒ Sequential logic
  – With n feedback signals → up to 2^n states

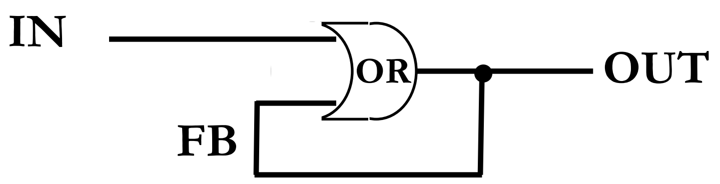
Sequential circuit: OR with feedback



IN_1	IN_2	OUT
0	0	0
0	1	1
1	0	1
1	1	1

$OUT = IN_1 + IN_2$

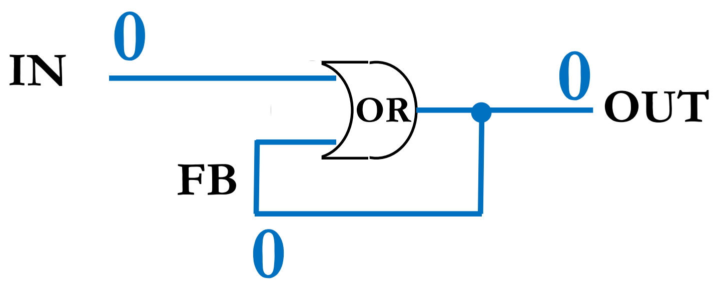
Sequential circuit: OR with feedback



IN	FB	OUT
0	0	0
0	1	1
1	0	1
1	1	1

OUT = IN + FB

Sequential circuit: OR with feedback

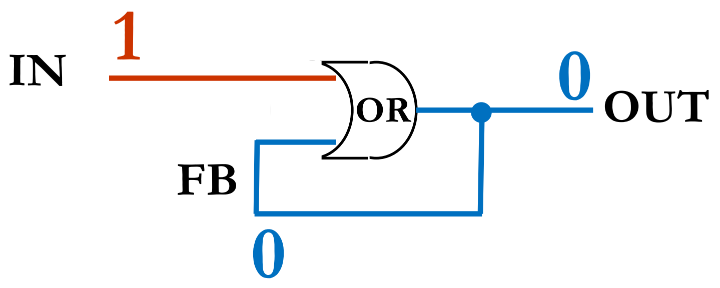


IN	FB	OUT
0	0	0
0	1	1
1	0	1
1	1	1

OUT = IN + FB

- Initial state: both IN and OUT are **0**

Sequential circuit: OR with feedback

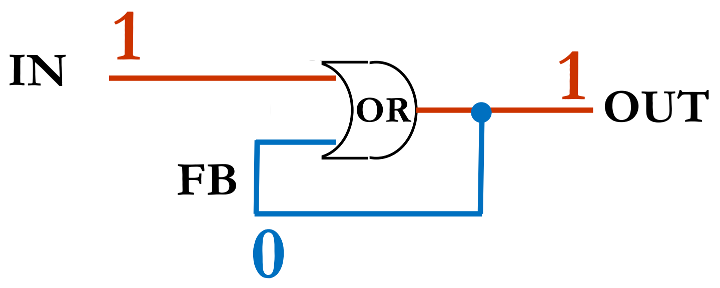


IN	FB	OUT
0	0	0
0	1	1
1	0	1
1	1	1

OUT = IN + FB

- Initial state: both IN and OUT are 0
- Let's set IN to 1

Sequential circuit: OR with feedback

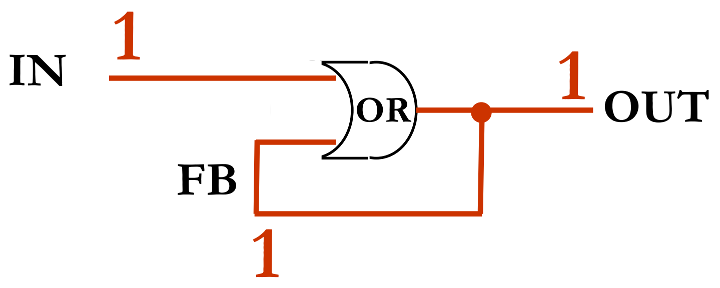


IN	FB	OUT
0	0	0
0	1	1
1	0	1
1	1	1

OUT = IN + FB

- Initial state: both IN and OUT are **0**
- Let's set IN to **1**
    - OUT becomes **1**

Sequential circuit: OR with feedback

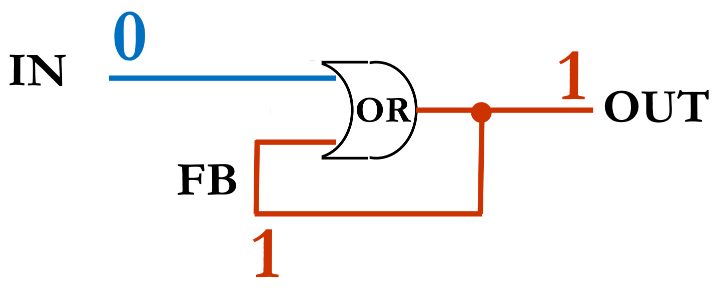


IN	FB	OUT
0	0	0
0	1	1
1	0	1
1	1	1

OUT = IN + FB

- Initial state: both IN and OUT are **0**
- Let's set IN to **1**
    - OUT and FB become **1**

Sequential circuit: OR with feedback

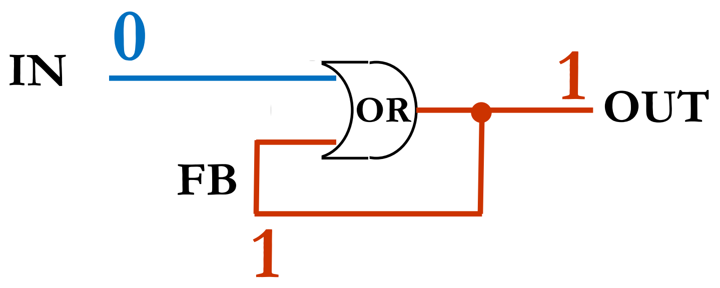


IN	FB	OUT
0	0	0
0	1	1
1	0	1
1	1	1

OUT = IN + FB

- Initial state: both IN and OUT are **0**
- Let's set IN to **1**
  - OUT and FB become **1**
- Let's set IN to **0**

Sequential circuit: OR with feedback

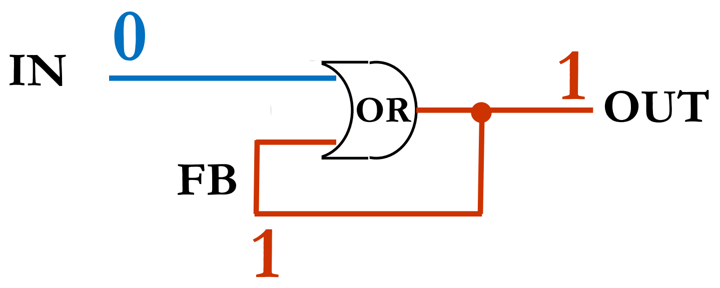


IN	FB	OUT
0	0	0
0	1	1
1	0	1
1	1	1

OUT = IN + FB

- Initial state: both IN and OUT are **0**
- Let's set IN to **1**
    - OUT and FB become **1**
- Let's set IN to **0**
    - OUT does not change.

Sequential circuit: OR with feedback

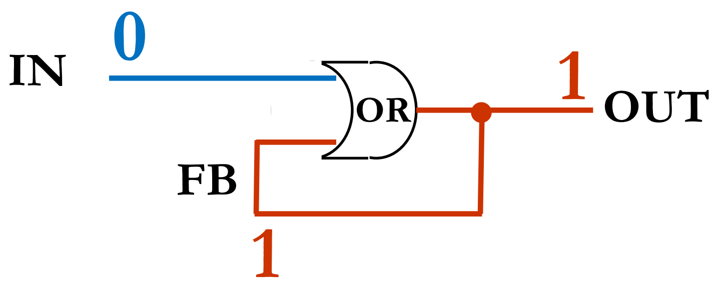


IN	FB	OUT
0	0	0
0	1	1
1	0	1
1	1	1

OUT = IN + FB

- Initial state: both IN and OUT are **0**
- Let's set IN to **1**
  - OUT and FB become **1**
- Let's set IN to **0**
  - OUT does not change. How to change OUT?

Sequential circuit: OR with feedback

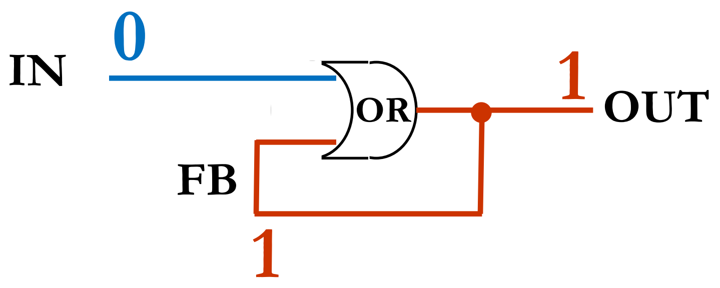


IN	FB	OUT
0	0	0
0	1	1
1	0	1
1	1	1

OUT = IN + FB

- Initial state: both IN and OUT are **0**
- Let's set IN to **1**
  - OUT and FB become **1**
- Let's set IN to **0**
  - OUT does not change. OUT will never change!

Sequential circuit: OR with feedback



IN	FB	OUT
0	0	0
0	1	1
1	0	1
1	1	1

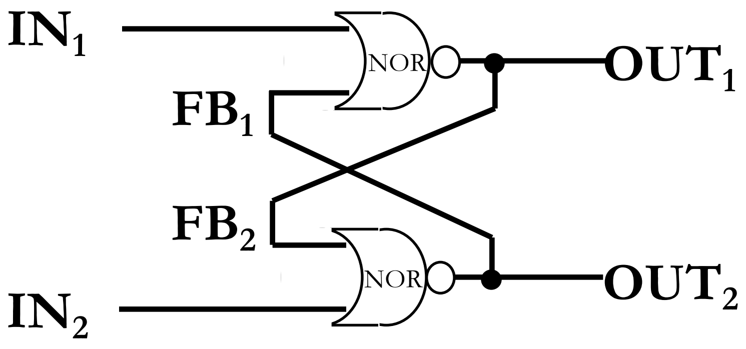
OUT = IN + FB

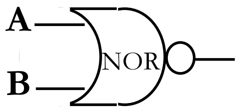
- Initial state: both IN and OUT are **0**
- Let's set IN to **1**
    - OUT and FB become **1**
- Let's set IN to **0**
    - OUT does not change. OUT will never change!

This circuit 'remembers' if there was 1 on IN

Works only once, no reset. Not very practical.

More practical sequential circuits

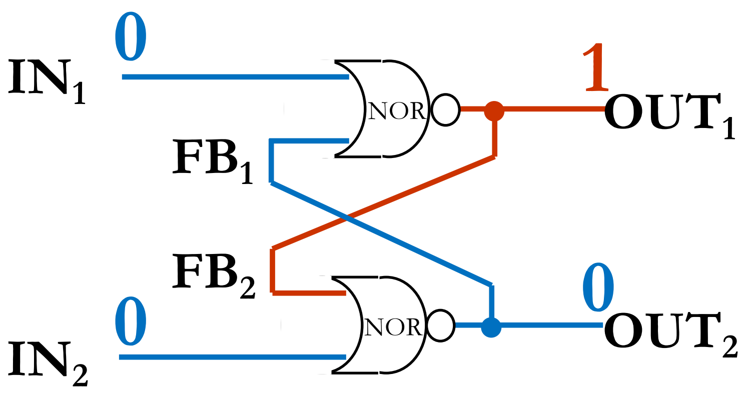


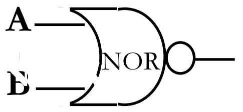


A	B	
0	0	1
0	1	0
1	0	0
1	1	0

- 2 NOR gates
- $OUT_1 = FB_2$; $OUT_2 = FB_1$

More practical sequential circuits

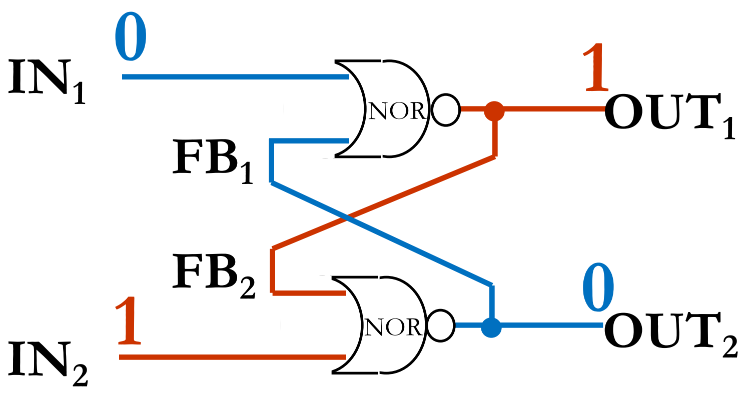


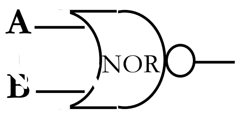


A	B	
0	0	1
0	1	0
1	0	0
1	1	0

- Initial state: IN_1, IN_2, OUT_2 are 0

More practical sequential circuits

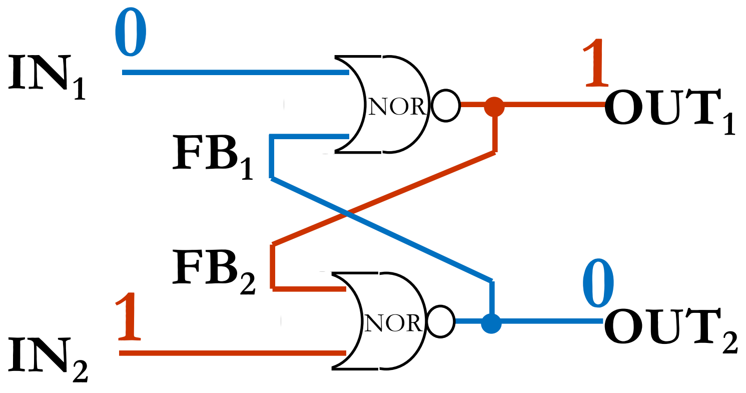


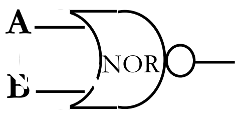


A	B	
0	0	1
0	1	0
1	0	0
1	1	0

- Initial state: IN$_{1,}$ IN$_2$, OUT$_2$ are 0
- Let’s set IN$_2$ to 1

More practical sequential circuits

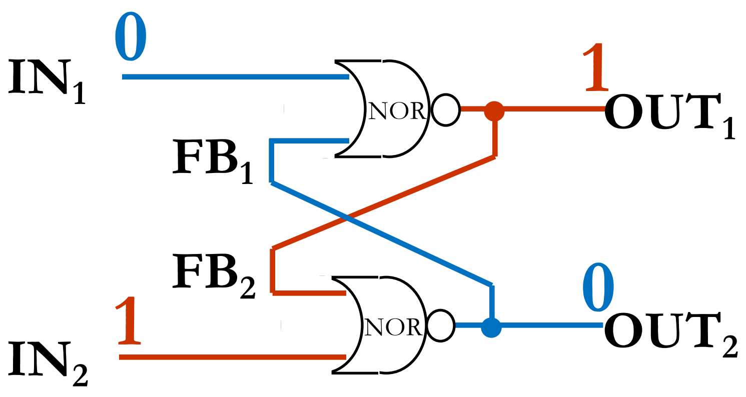


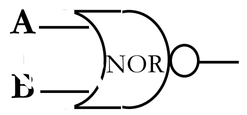


A	B	
0	0	1
0	1	0
1	0	0
1	1	0

- Initial state: IN_1, IN_2, OUT_2 are 0
- Let's set IN_2 to 1

More practical sequential circuits

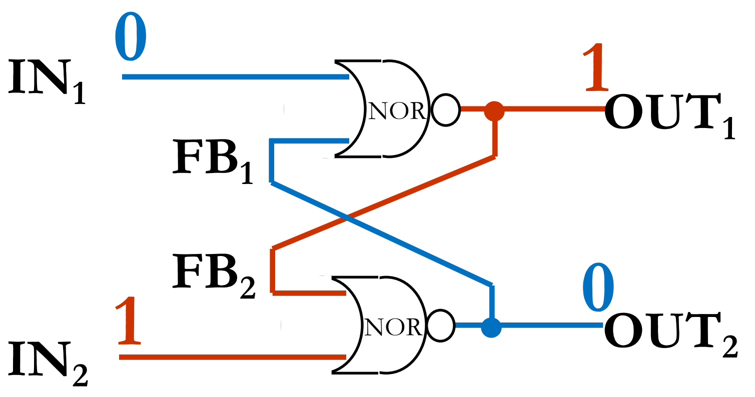


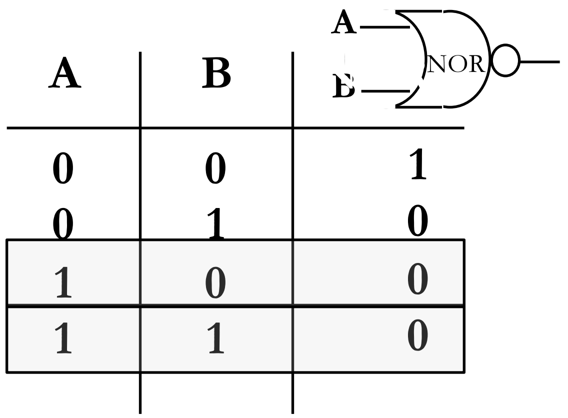


A	B	
0	0	1
0	1	0
1	0	0
1	1	0

- Initial state: IN$_{1,}$ IN$_2$, OUT$_2$ are 0
- Let's set IN$_2$ to 1
  - Nothing changes!

More practical sequential circuits

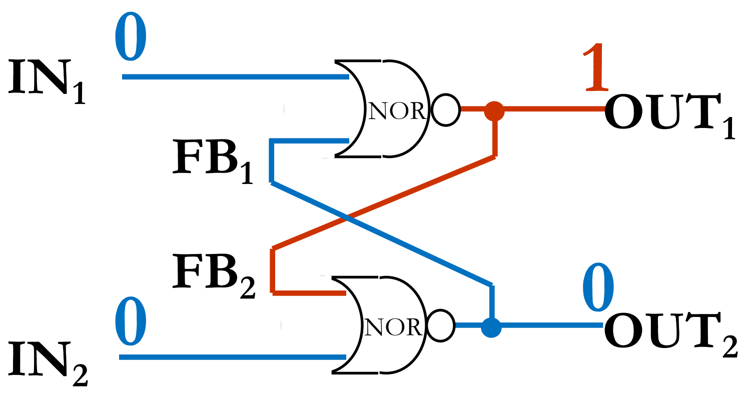


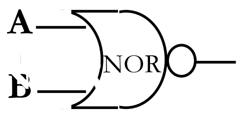


A	B	
0	0	1
0	1	0
1	0	0
1	1	0

- Initial state: IN_1, IN_2, OUT_2 are 0
- Let's set IN_2 to 1
  - Nothing changes!
  - In this state, IN_2 does not affect anything

More practical sequential circuits

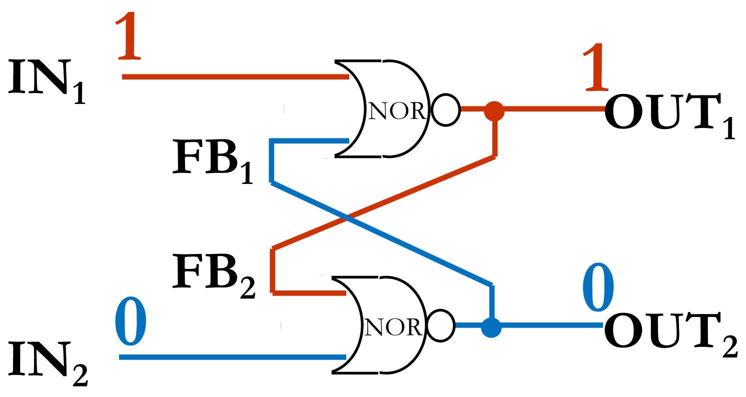


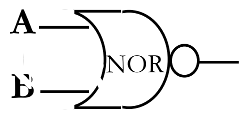


A	B	
0	0	1
0	1	0
1	0	0
1	1	0

- Initial state: IN_1, IN_2, OUT_2 are 0

More practical sequential circuits

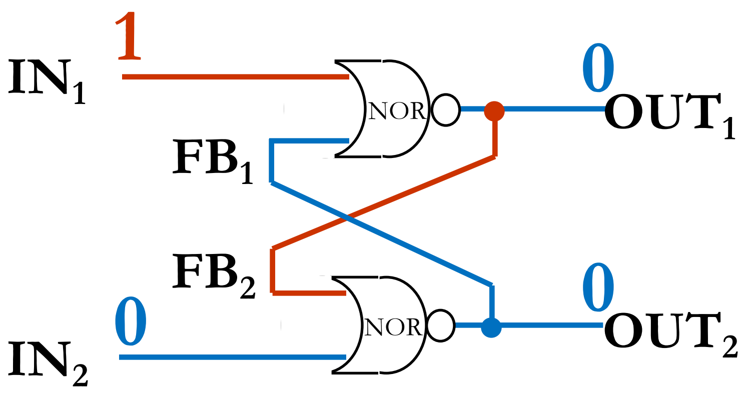


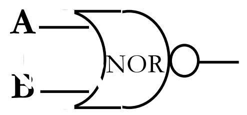


A	B	
0	0	1
0	1	0
1	0	0
1	1	0

- Initial state: IN$_{1,}$ IN$_2$, OUT$_2$ are 0
- Let's set IN$_1$ to 1

More practical sequential circuits

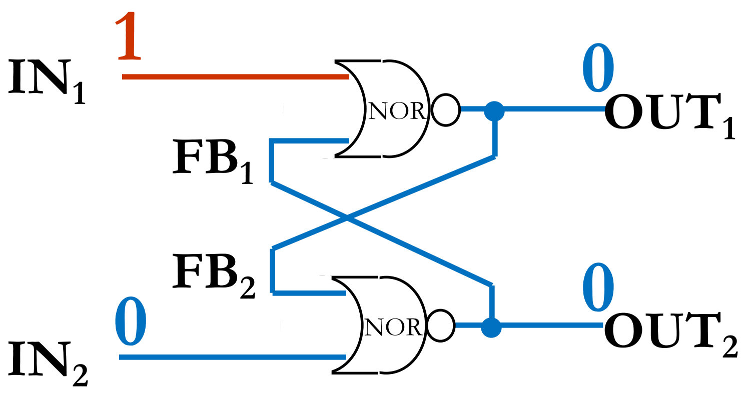


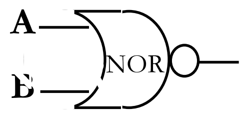


A	B	
0	0	1
0	1	0
1	0	0
1	1	0

- Initial state: IN$_1$, IN$_2$, OUT$_2$ are **0**
- Let's set IN$_1$ to **1**
  - OUT$_1$ => **0**

More practical sequential circuits

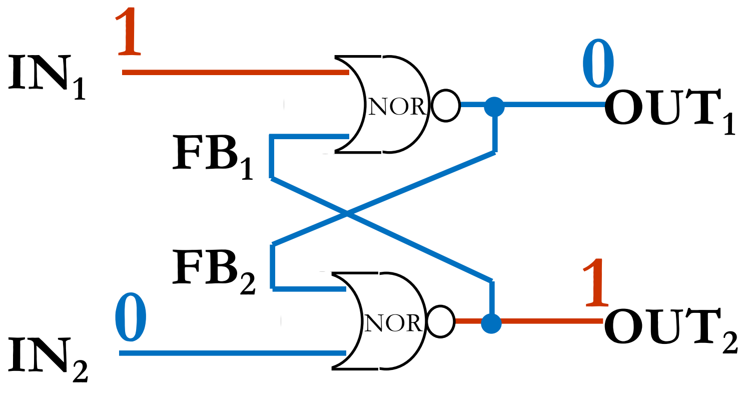


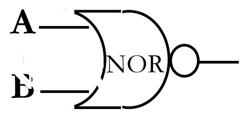


A	B	
0	0	1
0	1	0
1	0	0
1	1	0

- Initial state: IN_1, IN_2, OUT_2 are **0**
- Let's set IN_1 to **1**
  - OUT_1 => **0**
  - FB_2 => **0**

More practical sequential circuits

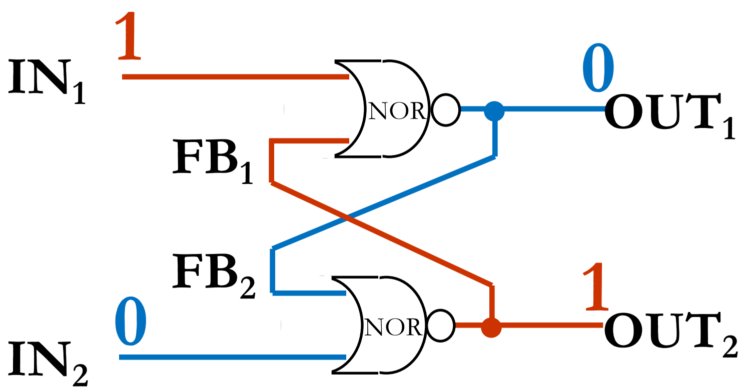


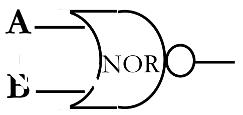


A	B	
0	0	1
0	1	0
1	0	0
1	1	0

- Initial state: IN$_1$, IN$_2$, OUT$_2$ are 0
- Let's set IN$_1$ to 1
    - OUT$_1$ => 0
    - FB$_2$ => 0
    - OUT$_2$ => 1

More practical sequential circuits

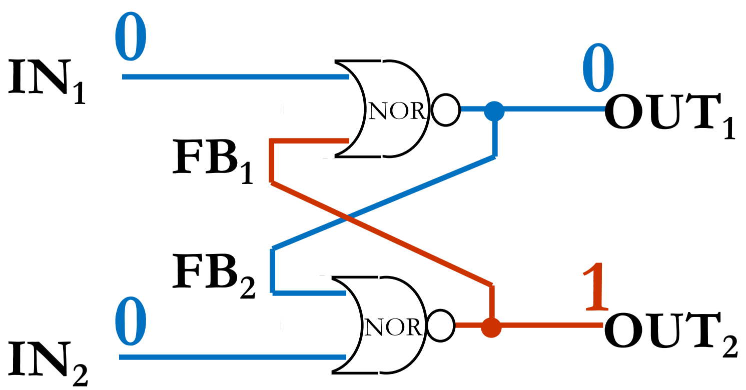


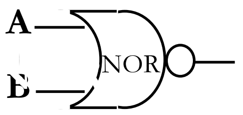


A	B	
0	0	1
0	1	0
1	0	0
1	1	0

- Initial state: IN_1, IN_2, OUT_2 are 0
- Let's set IN_1 to 1
  - OUT_1 => 0
  - FB_2 => 0
  - OUT_2 => 1
  - FB_1 => 1

More practical sequential circuits



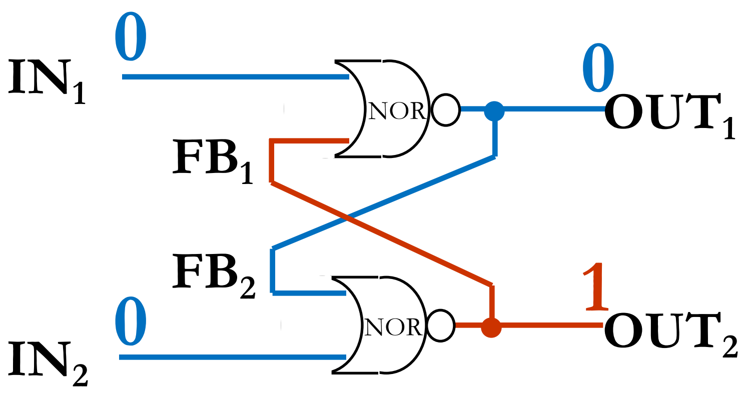


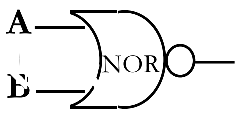
A	B	
0	0	1
0	1	0
1	0	0
1	1	0

- Initial state: IN_1, IN_2, OUT_2 are 0
- Let's set IN_1 to 1
  - OUT_1 => 0
  - FB_2 => 0
  - OUT_2 => 1
  - FB_1 => 1

- Set IN_1 back to 0

More practical sequential circuits

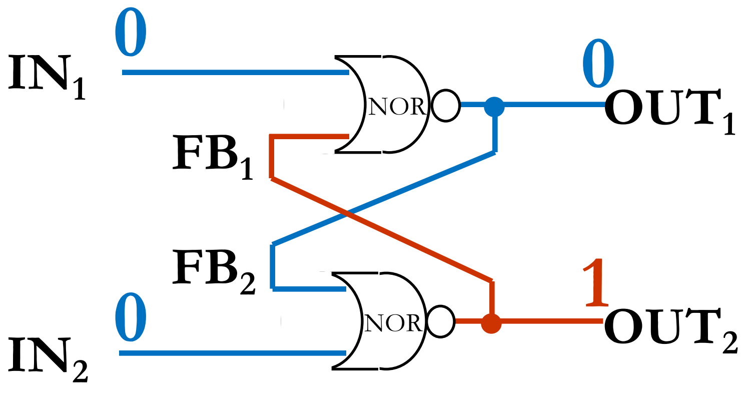


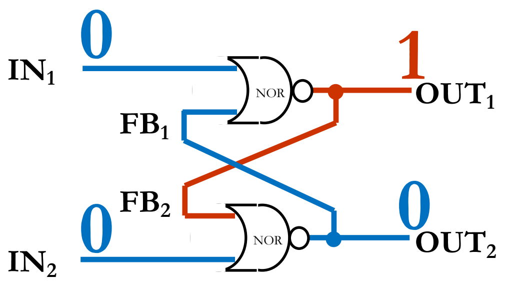


A	B	
0	0	1
0	1	0
1	0	0
1	1	0

- Initial state: IN$_1$, IN$_2$, OUT$_2$ are 0
- Let's set IN$_1$ to 1
  - OUT$_1$ => 0
  - FB$_2$ => 0
  - OUT$_2$ => 1
  - FB$_1$ => 1
- Set IN$_1$ back to 0
  - Nothing changes!

More practical sequential circuits

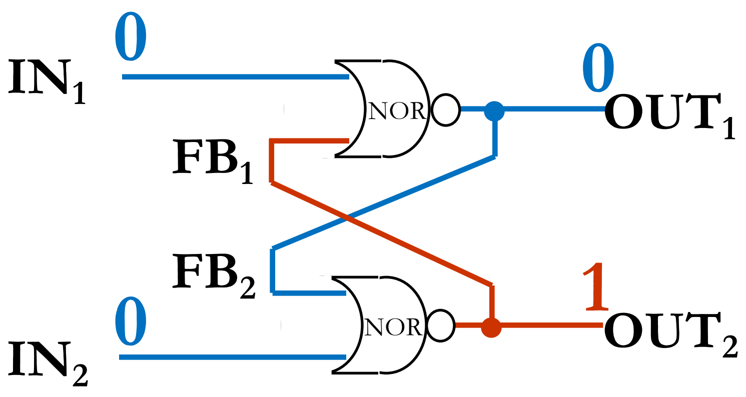


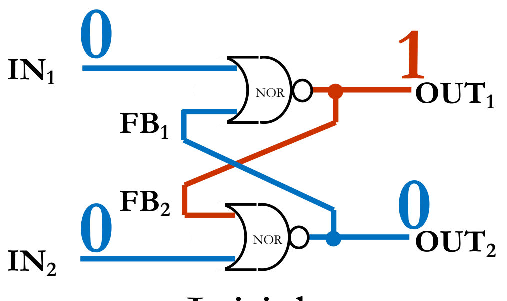


Initial state

- Initial state: IN_1, IN_2, OUT_2 are **0**
- Let's set IN_1 to **1**
  - OUT_1 => **0**
  - FB_2 => **0**
  - OUT_2 => **1**
  - FB_1 => **1**
- Set IN_1 back to **0**
  - Nothing changes!

More practical sequential circuits

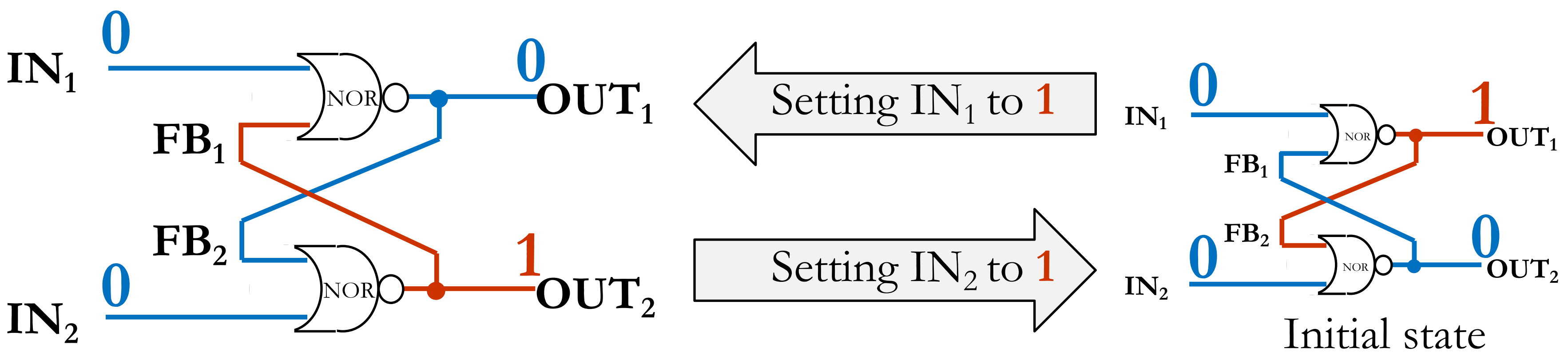




Initial state

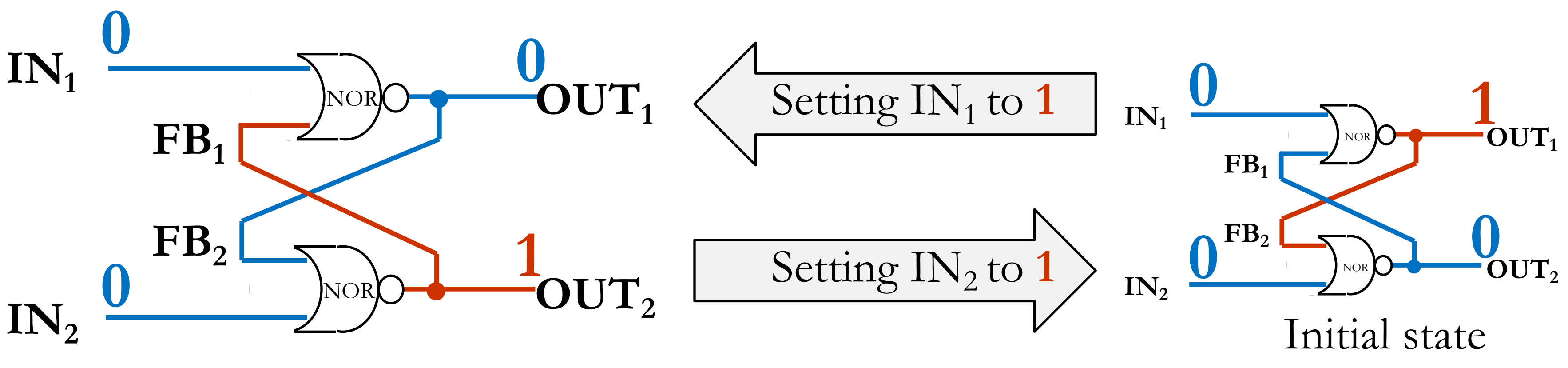
- Two "mirrored" stable states

More practical sequential circuits



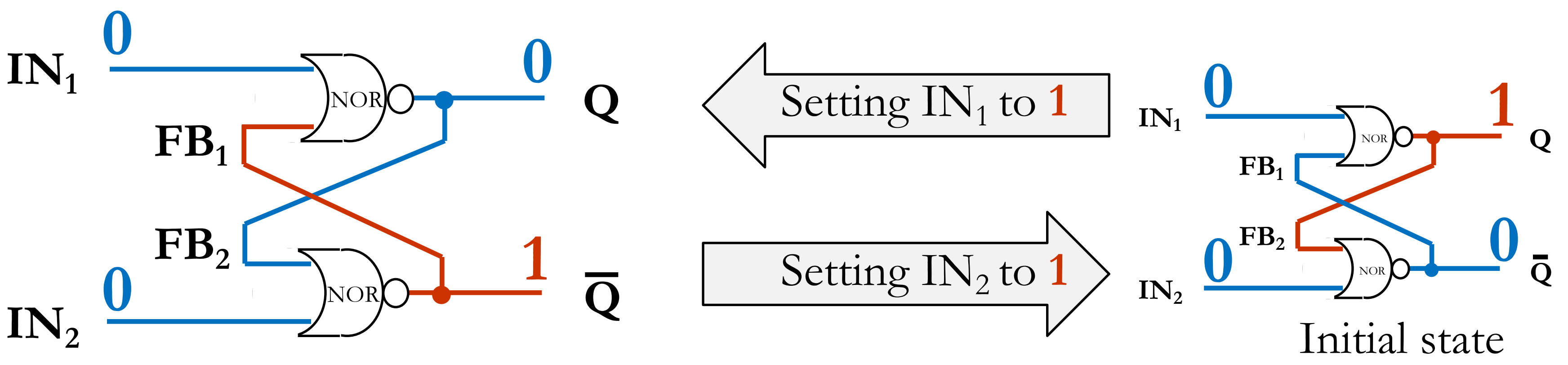
- Two "mirrored" stable states
- Setting one of the inputs to 1 changes the state

More practical sequential circuits



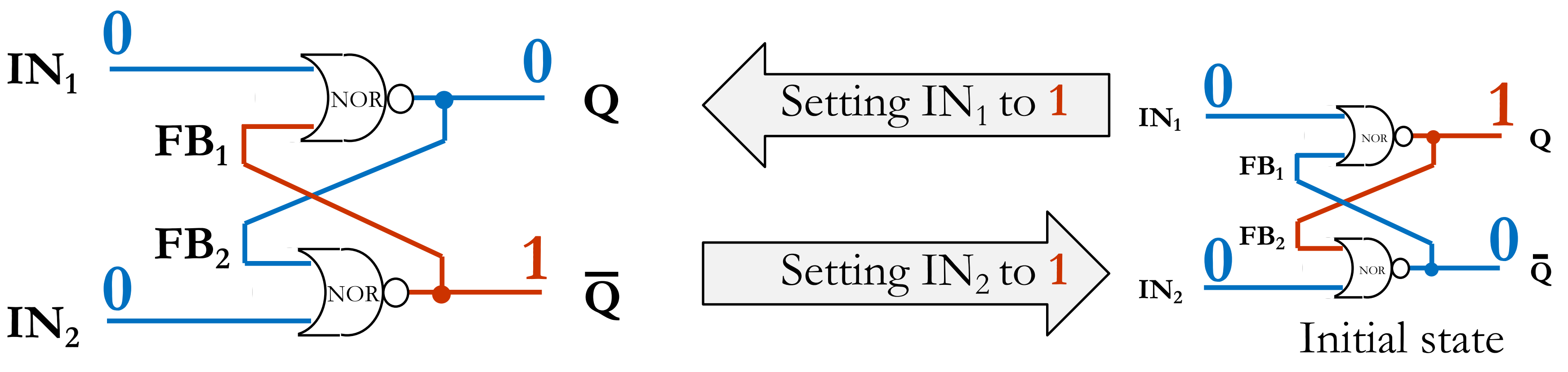
- Two “mirrored” stable states
- Setting one of the inputs to 1 changes the state
- Outputs are inverse of each other ($\text{OUT}_1 = \overline{\text{OUT}_2} = \text{Q}$)

More practical sequential circuits



- Two "mirrored" stable states
- Setting one of the inputs to 1 changes the state
- Outputs are inverse of each other ($OUT_1 = \overline{OUT_2} = Q$)

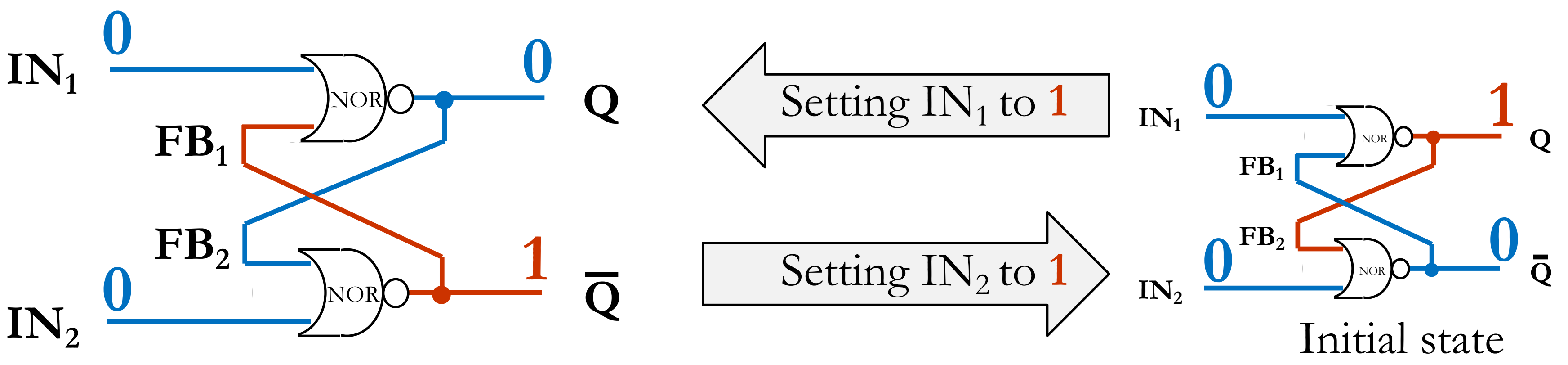
More practical sequential circuits



- Two "mirrored" stable states
- Setting one of the inputs to 1 changes the state
- Outputs are inverse of each other ($OUT_1 = \overline{OUT_2} = Q$)

By setting one of the inputs, Q can be set to either 1 or 0

More practical sequential circuits

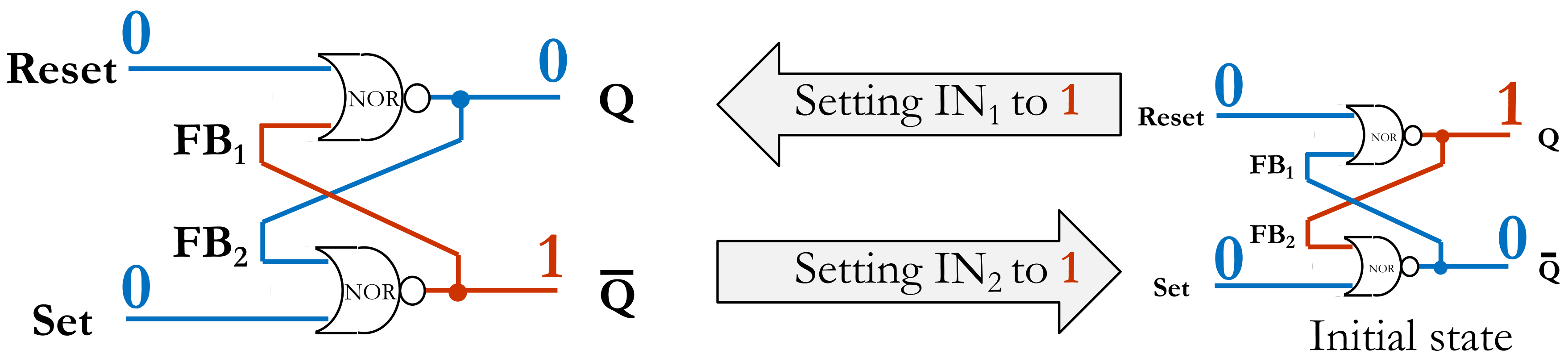


- Two "mirrored" stable states
- Setting one of the inputs to 1 changes the state
- Outputs are inverse of each other ($OUT_1 = \overline{OUT_2} = Q$)

By setting one of the inputs, Q can be set to either 1 or 0

- Setting IN$_1$ to **1** *resets* the value Q (**1 -> 0**)
- Setting IN$_2$ to **1** *sets* the value Q (**0 -> 1**)

More practical sequential circuits

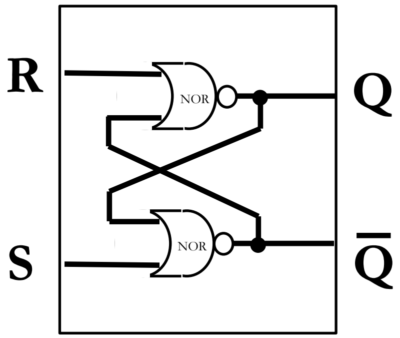


- Two "mirrored" stable states
- Setting one of the inputs to 1 changes the state
- Outputs are inverse of each other ($OUT_1 = \overline{OUT_2} = Q$)

By setting one of the inputs, Q can be set to either 1 or 0

- Setting IN$_1$ to **1** *resets* the value Q (**1 → 0**)
- Setting IN$_2$ to **1** *sets* the value Q (**0 → 1**)

SR latch



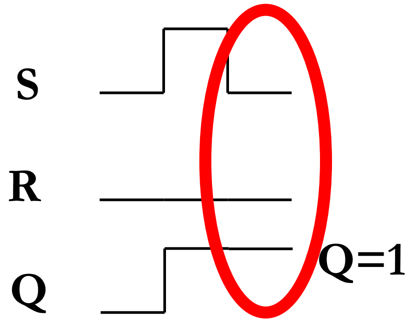
S	R	Q_i
0	0	Q_{i-1}
0	1	0
1	0	1
1	1	inv

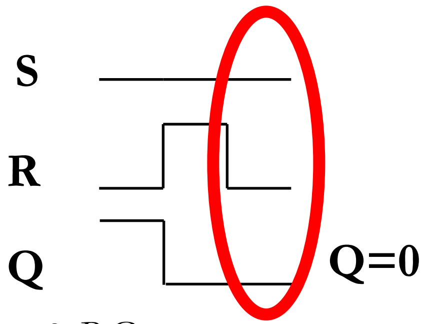
Previous value

inv=invalid

Usage: 1-bit memory

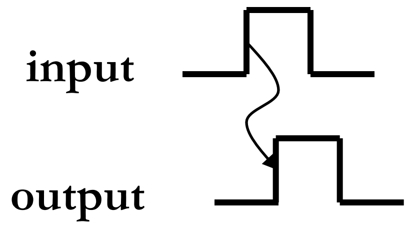
- Keep the value in memory by maintaining S=0 and R=0
- Set the value in memory to 0 (or 1) by setting R=1 (or S=1) for a short time





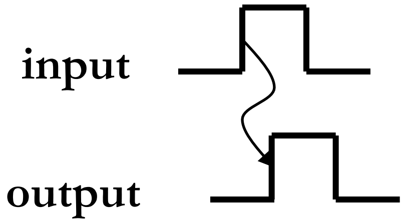
Timing of events

- Asynchronous sequential logic
  - State (and possibly output) of circuit changes whenever inputs change



Timing of events

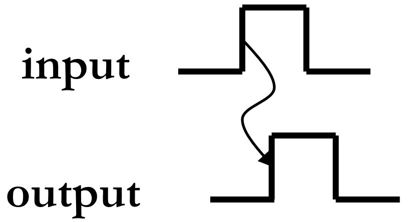
- Asynchronous sequential logic
  - State (and possibly output) of circuit changes whenever inputs change



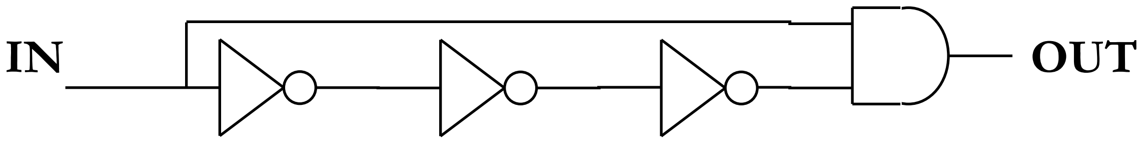
Problem: different delays on different circuit paths

Timing of events

- Asynchronous sequential logic
  - State (and possibly output) of circuit changes whenever inputs change

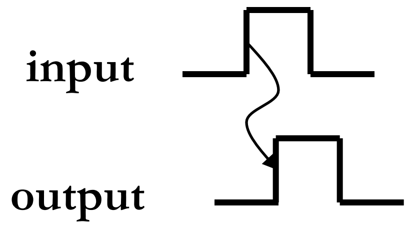


Problem: different delays on different circuit paths

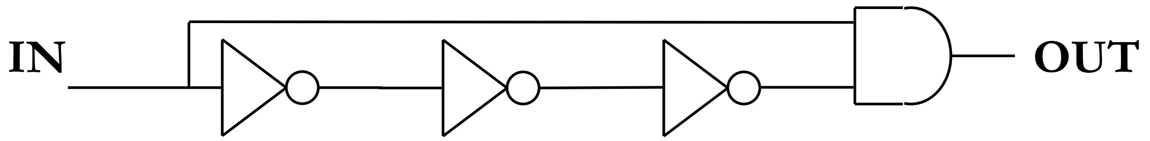


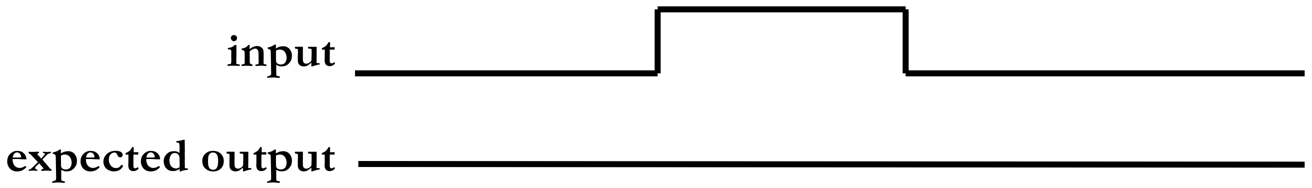
Timing of events

- Asynchronous sequential logic
  - State (and possibly output) of circuit changes whenever inputs change



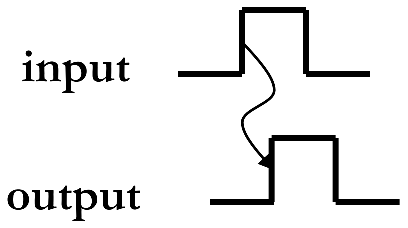
Problem: different delays on different circuit paths



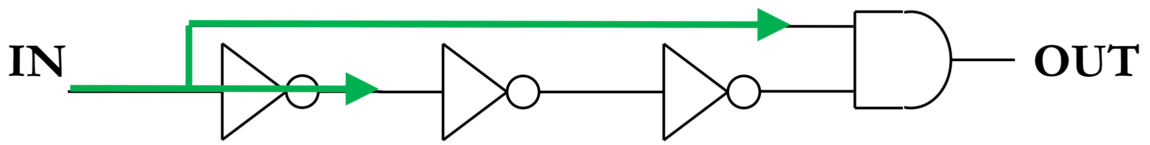


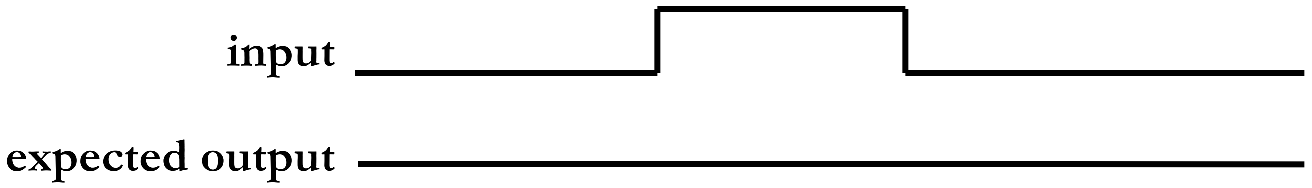
Timing of events

- Asynchronous sequential logic
  - State (and possibly output) of circuit changes whenever inputs change



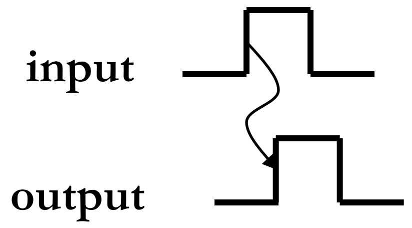
Problem: different delays on different circuit paths



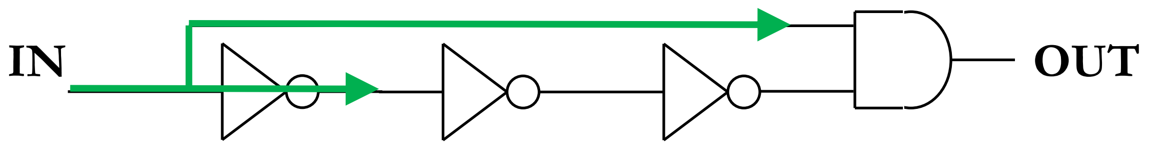


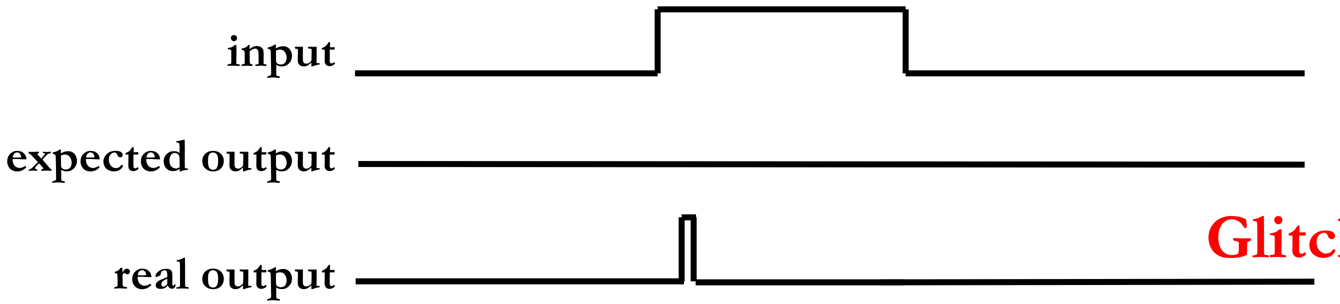
Timing of events

- **Asynchronous** sequential logic
  - State (and possibly output) of circuit changes whenever inputs change



Problem: different delays on different circuit paths

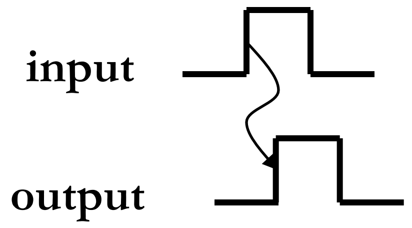




Glitches!

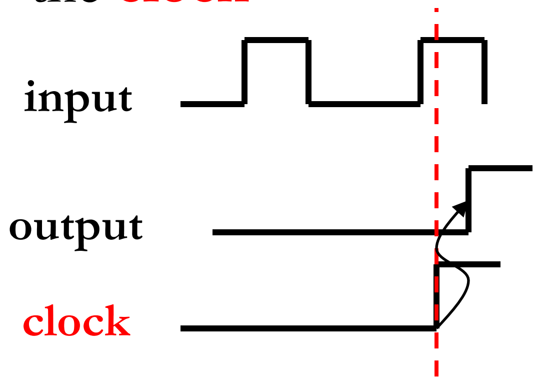
Timing of events

- **Asynchronous** sequential logic
  - State (and possibly output) of circuit changes whenever inputs change



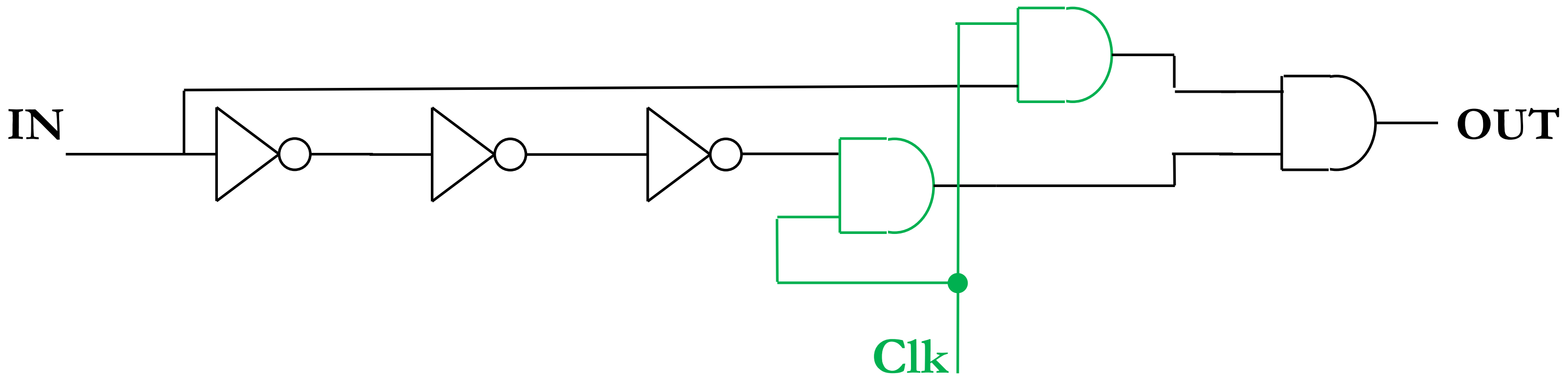
Problem: different delays on different circuit paths

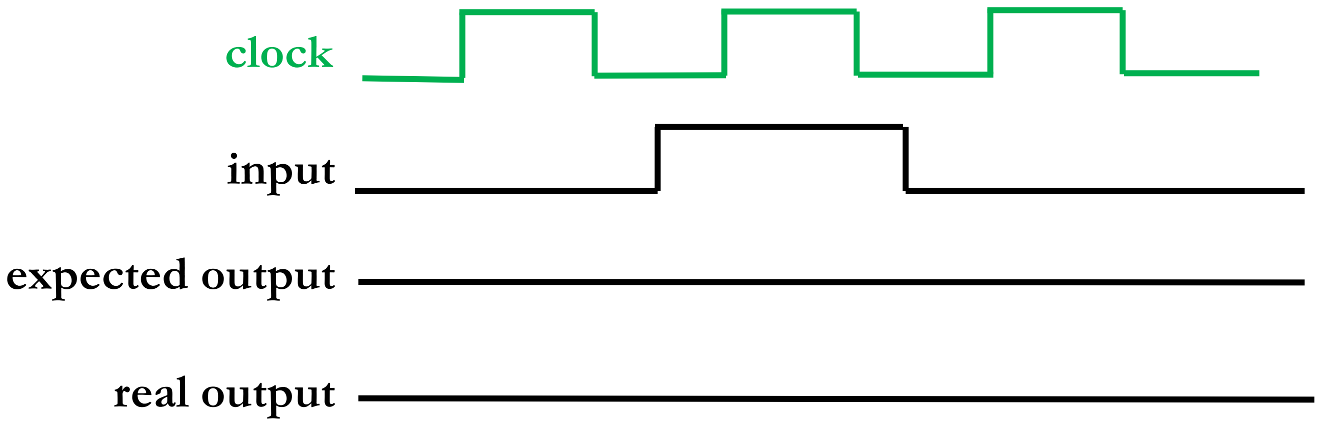
- **Solution**: **Synchronous** sequential logic
  - State (and possibly output) can only change at times synchronized to an external signal → the **clock**



Timing of events

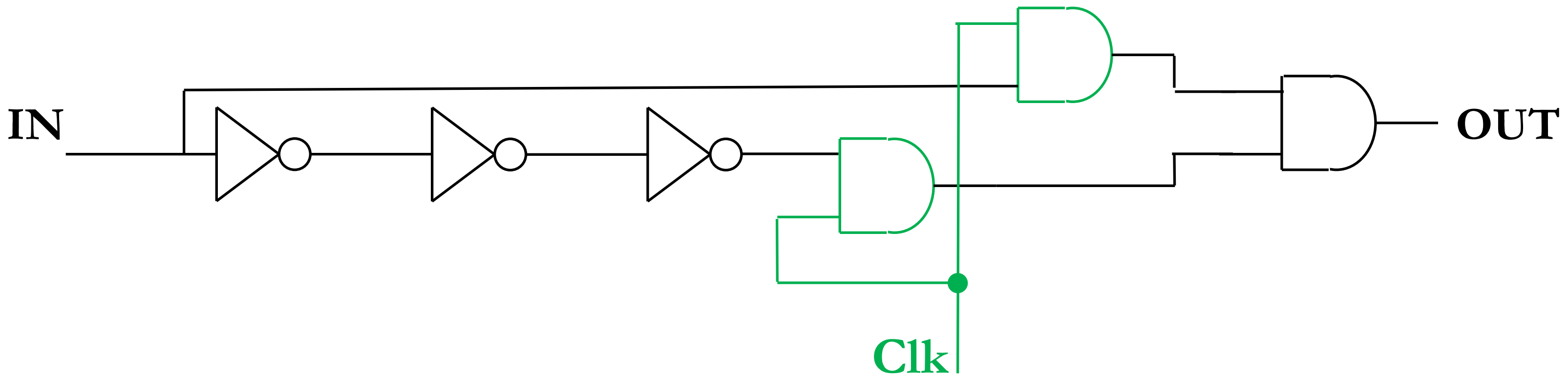
Synchronous sequential logic

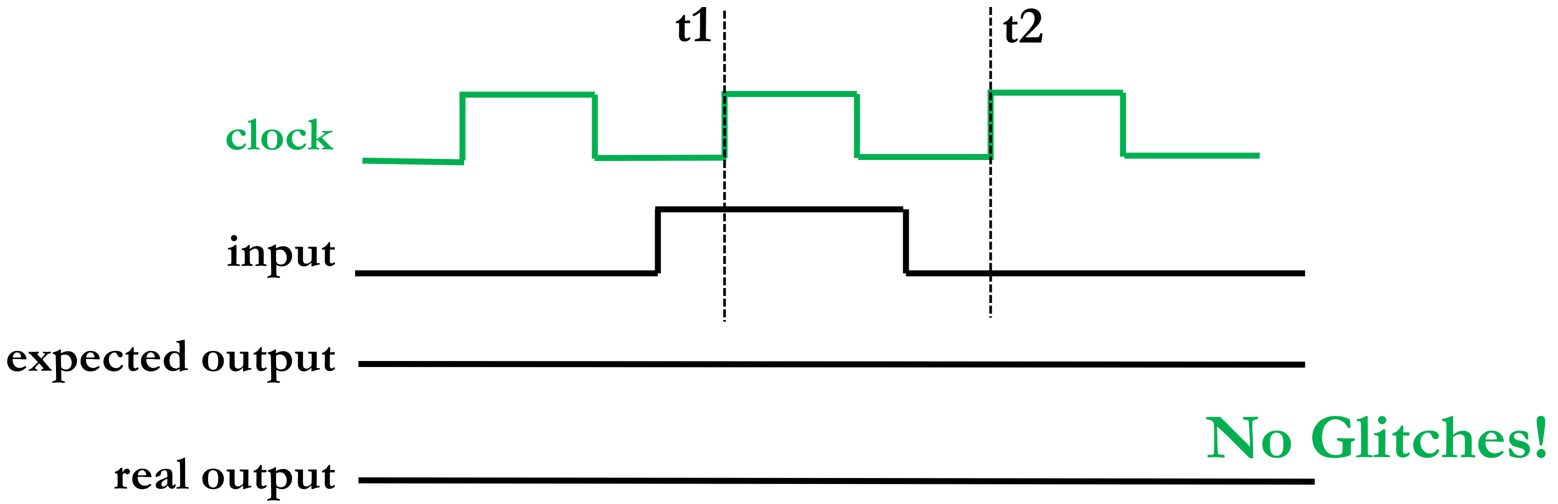




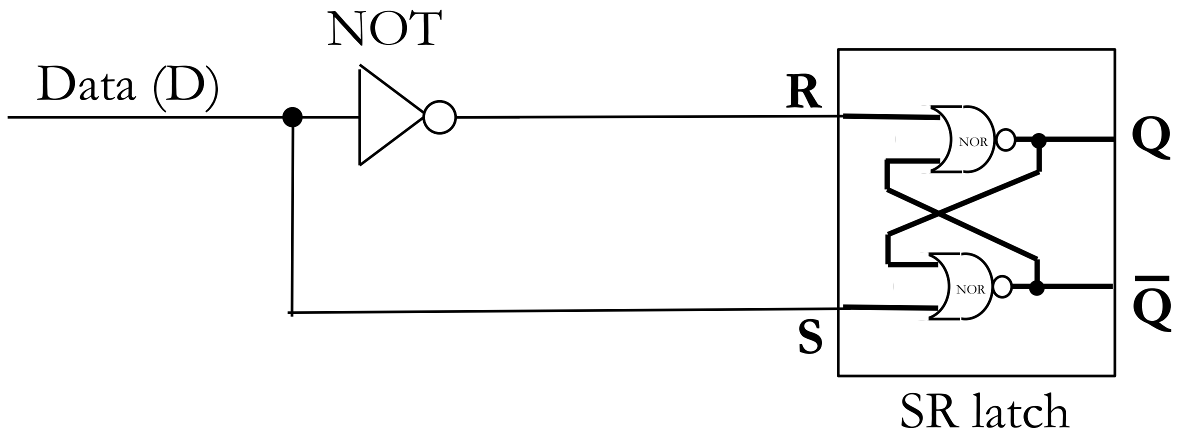
Timing of events

Synchronous sequential logic





Clocked SR latch



S	R	Q_i
0	0	Q_{i-1}
0	1	0
1	0	1
1	1	inv

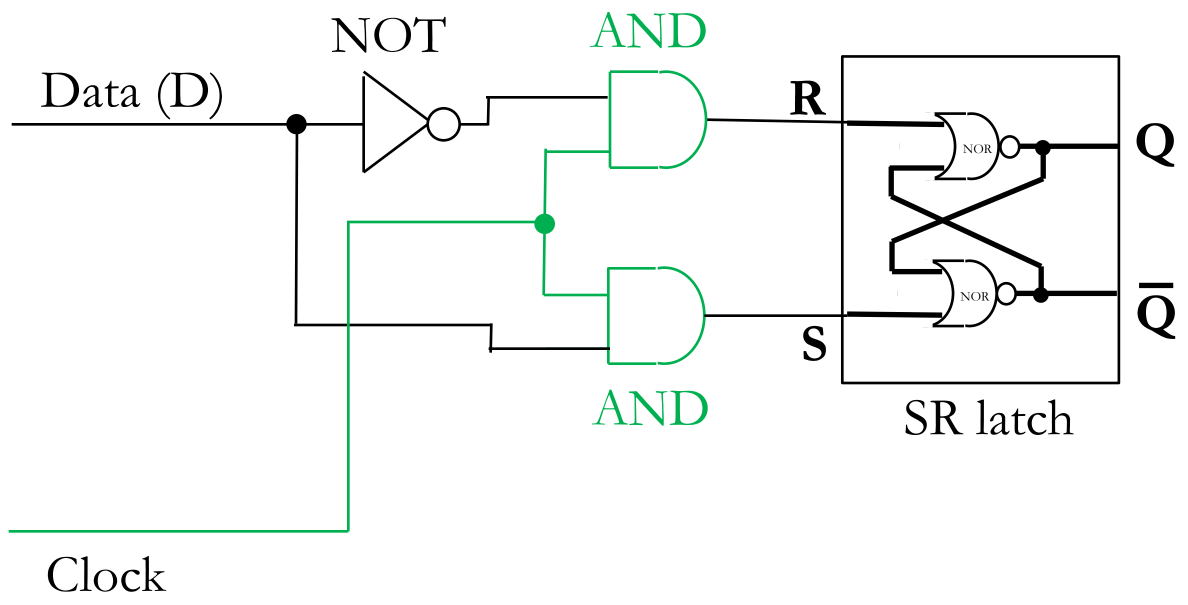
Some logic computes the data (1 bit)

We want to 'save' the data in an SR latch when the data is ready

Problem: glitches

Solution: add clock!

Clocked SR latch



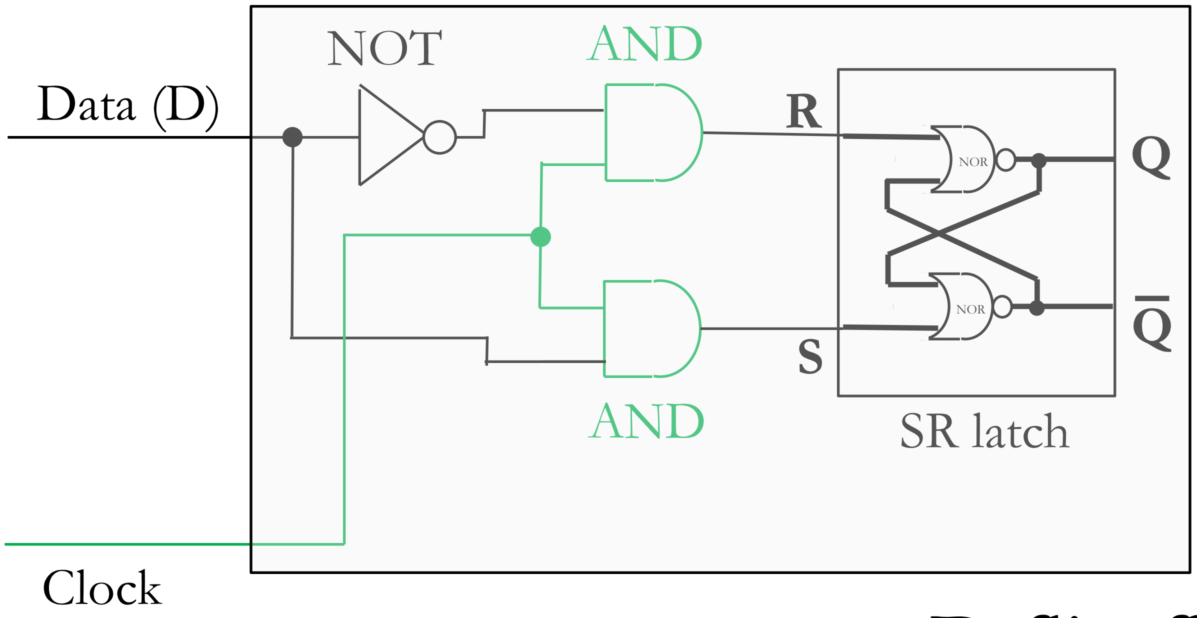
S	R	Q_i
0	0	Q_{i-1}
0	1	0
1	0	1
1	1	inv

Some logic computes the data (1 bit)

We want to 'save' the data in an SR latch when the data is ready

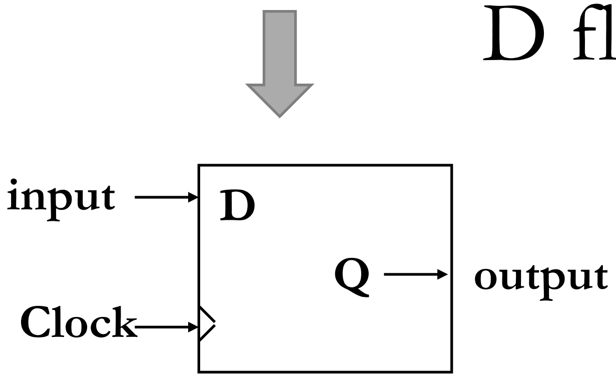
Set Q to D when Clock is 1; Ignore D if Clock is 0

Clocked SR latch: D flip-flop

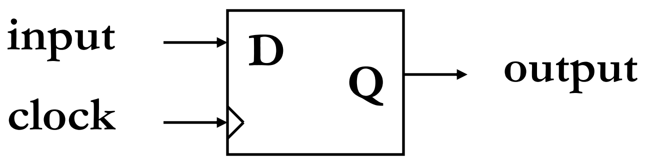


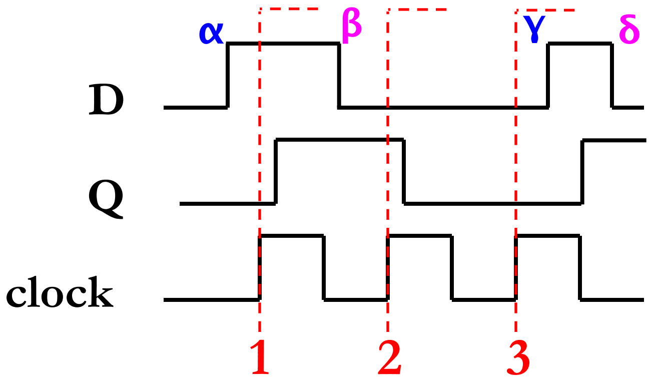
S	R	Q_i
0	0	Q_{i-1}
0	1	0
1	0	1
1	1	inv

D flip-flop



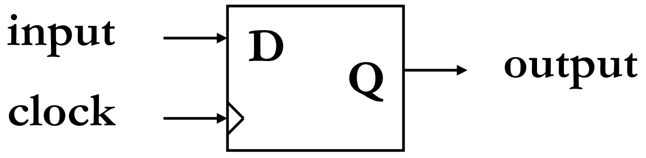
Using clock to build a memory element

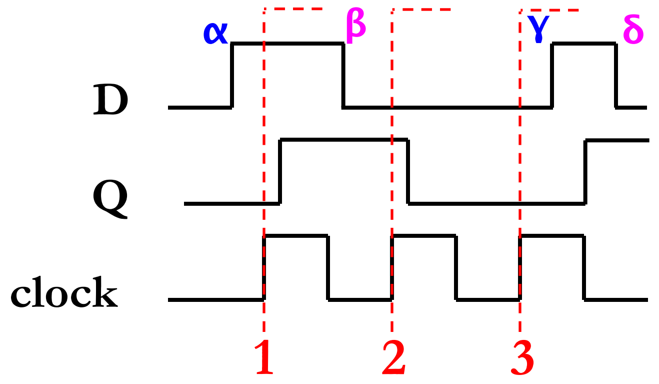




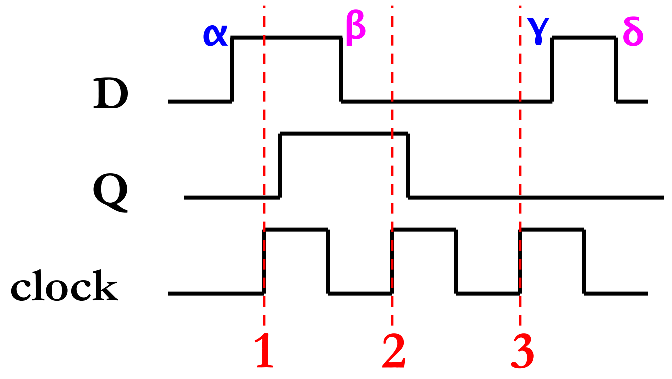
- Level-triggered D flip-flop: whenever clock is 1, D is propagated to Q

Using clock to build a memory element



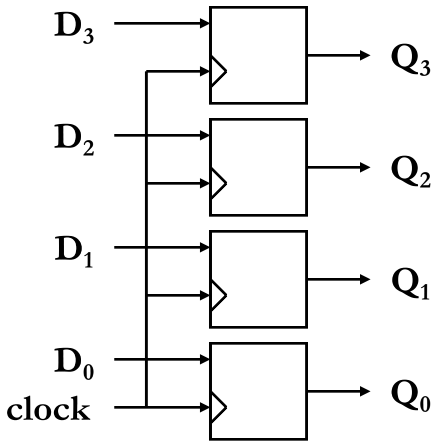


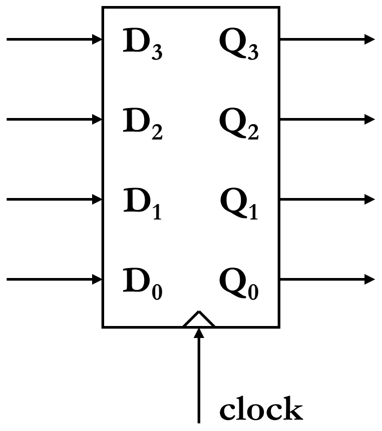
- Level-triggered D flip-flop: whenever clock is 1, D is propagated to Q
- Edge-triggered D flip-flop: on a positive clock edge, D is propagated to Q

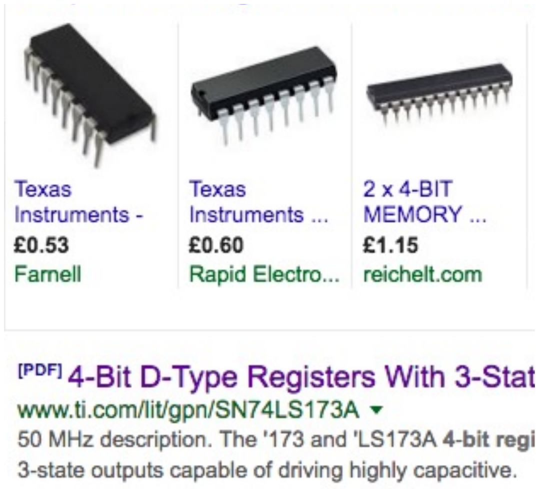


Register

- Tie multiple D flip-flops together using a common clock
- E.g., 4-bit register:



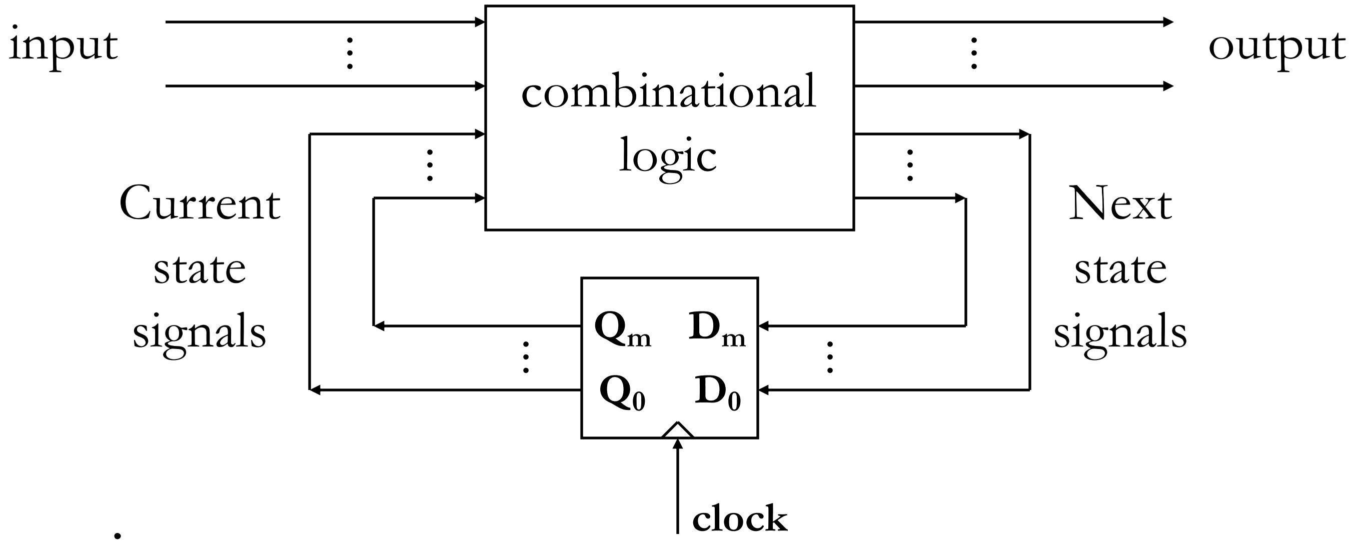




Texas Instruments -	Texas Instruments ...	2 x 4-BIT MEMORY ...
£0.53	£0.60	£1.15
Farnell	Rapid Electro...	reichelt.com

[PDF] 4-Bit D-Type Registers With 3-Stat
www.ti.com/lit/gpn/SN74LS173A ▾
50 MHz description. The '173 and 'LS173A 4-bit regi
3-state outputs capable of driving highly capacitive.

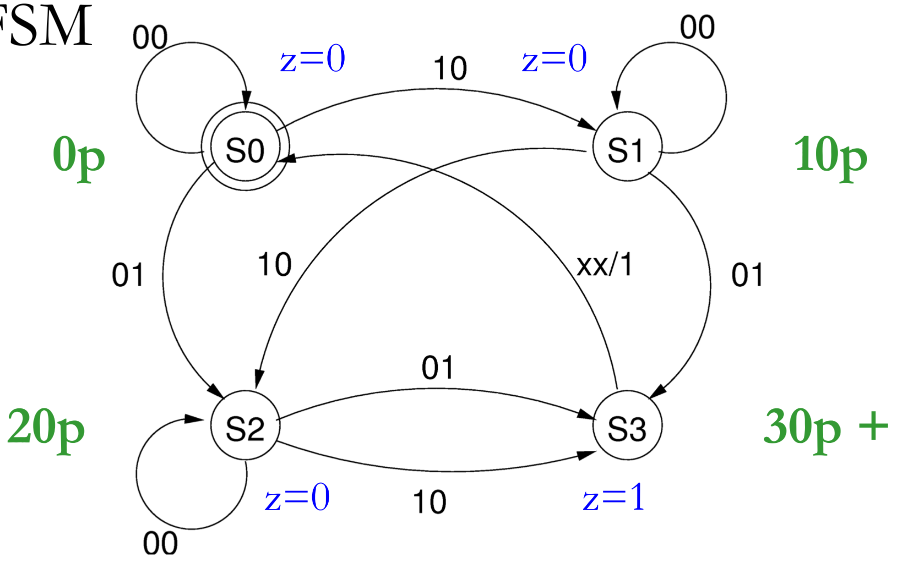
General sequential logic circuit



- Operation:
  - At every rising clock edge, next state signals are propagated to current state signals
  - Current state signals plus inputs work through combinational logic and generate output and next state signals

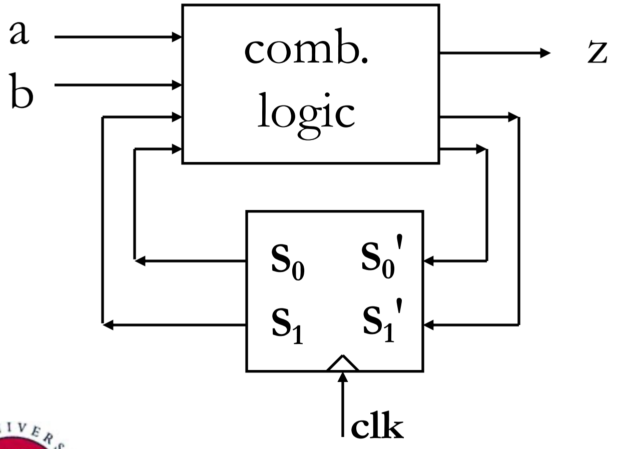
Hardware FSM

- A sequential circuit is a (deterministic) Finite State Machine – FSM
- Example: Vending machine
  - Accepts 10p, 20p coins, sells one product costing 30p, no change given
  - Coin reader has 2 signals: *a, b* for 10p, 20p coins respectively. These are the inputs to our FSM
  - Output *z* asserted when 30p or more has been paid in



FSM implementation

- Methodology:
  - Choose encoding for states, e.g S0=00, …, S3=11
  - Build truth table for the next state s_1', s_0' and output z
  - Generate logic equations for s_1', s_0', z
  - Design comb logic from logic equations and add state-holding register



s_1	s_0	a	b	s_1'	s_0'	z
0	0	0	0	0	0	0
0	0	0	1	1	0	0
0	0	1	0	0	1	0
0	1	0	0	0	1	0
0	1	0	1	1	1	1
0	1	1	0	1	0	0
●●●●●●●●				●●●●●●●●		