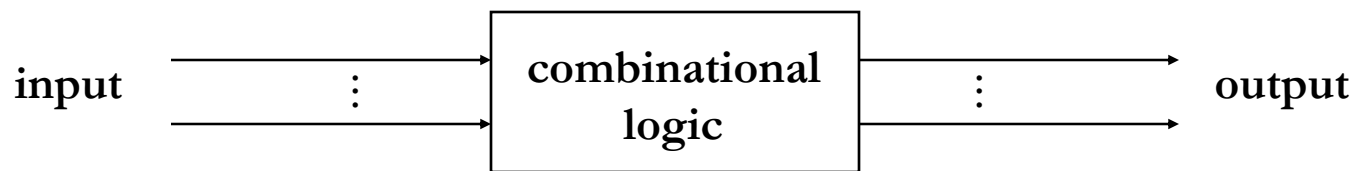


Lecture 7: Logic design

- Binary digital logic circuits:
 - Two voltage levels (ground and supply voltage) for 0 and 1
 - Built from transistors used as on/off switches
 - Analog circuits not very suitable for generic computing
 - Digital logic with more than two states is not practical

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

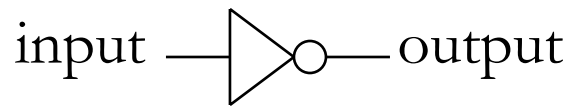


Sequential logic: output depends on the current inputs as well as
(some) previous inputs



Combinational logic circuits

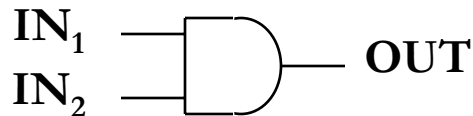
- Inverter (or NOT gate): 1 input and 1 output
“invert the input signal”



IN	OUT
0	1
1	0

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

- AND gate: minimum 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

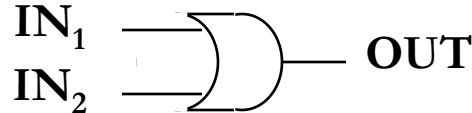
$$\text{OUT} = \text{IN}_1 \cdot \text{IN}_2$$



Combinational logic circuits

- OR gate:

- “output 1 if at least one input is 1”

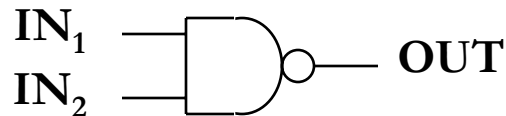


IN_1	IN_2	OUT
0	0	0
0	1	1
1	0	1
1	1	1

$$OUT = IN_1 + IN_2$$

- NAND gate:

- “output 1 if both inputs are not 1” (NOT AND)



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

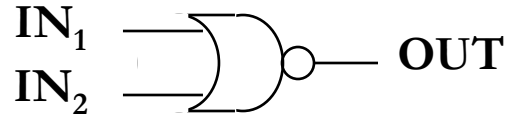
$$OUT = \overline{IN_1 \cdot IN_2}$$



Combinational logic circuits

- NOR gate:

“output 1 if no input is 1” (NOT OR)

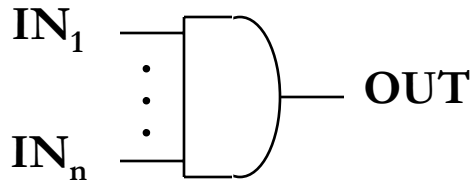


IN ₁	IN ₂	OUT
0	0	1
0	1	0
1	0	0
1	1	0

$$\text{OUT} = \overline{\text{IN}_1 + \text{IN}_2}$$

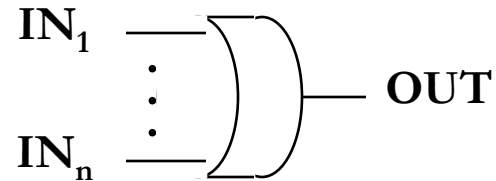
- Multiple-input gates:

AND



$$\text{OUT} = 1 \text{ if } \underline{\text{all}} \text{ } \text{IN}_i = 1$$

OR

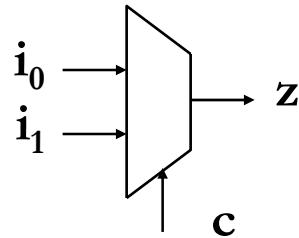


$$\text{OUT} = 1 \text{ if } \underline{\text{any}} \text{ } \text{IN}_i = 1$$



Multiplexer

- Multiplexer: a circuit for selecting one of many inputs



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

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

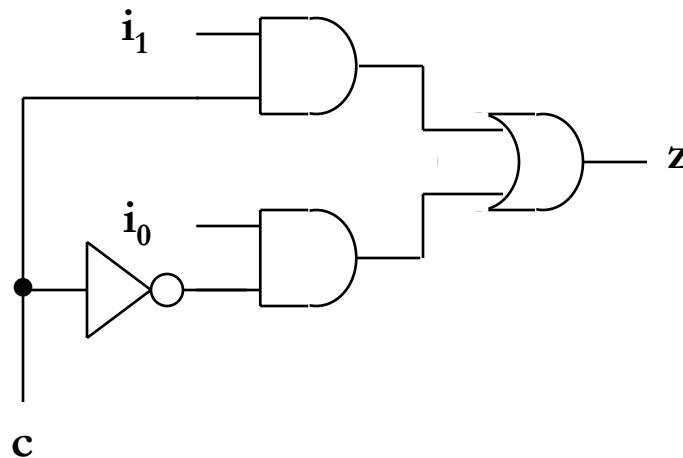
$$\begin{aligned}
 z &= \bar{i}_0.i_1.c + i_0.\bar{i}_1.\bar{c} + i_0.i_1.\bar{c} + i_0.i_1.c \\
 &= \bar{i}_0.i_1.c + i_0.i_1.c + i_0.i_1.\bar{c} + i_0.i_1.c \\
 &= (\bar{i}_0 + i_0).i_1.c + i_0.(i_1 + i_1).\bar{c} \\
 &= i_1.c + i_0.\bar{c}
 \end{aligned}$$

“sum of products form”



A multiplexer implementation

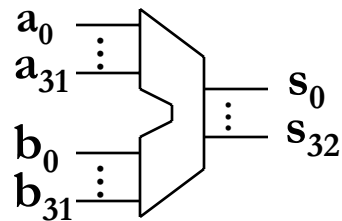
- Sum of products form: $i_1 \cdot c + i_0 \cdot \bar{c}$
 - Can be implemented with 1 inverter, 2 AND gates and 1 OR gate:



- 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

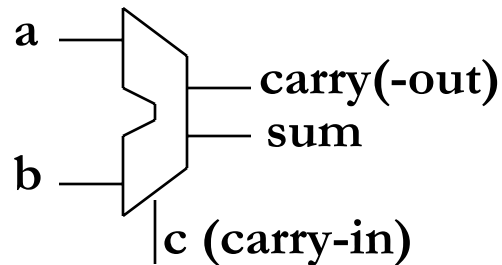
Arithmetic circuits

- 32-bit adder



64 inputs \rightarrow too complex for sum of products

- Full adder:



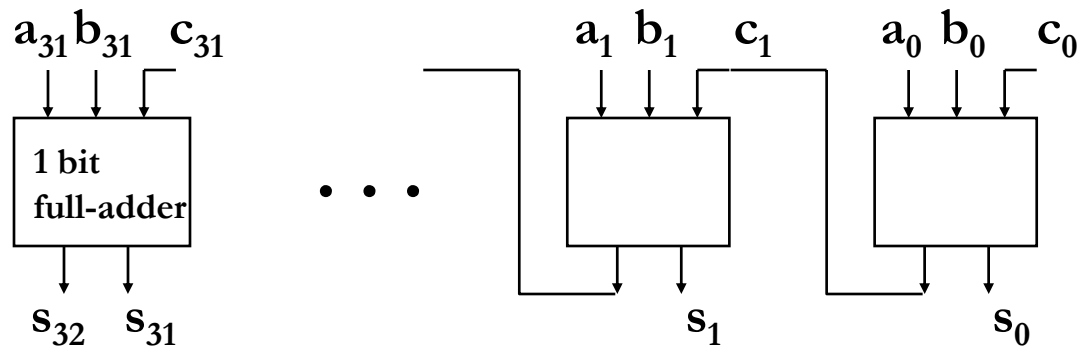
$$\text{sum} = \bar{a}.\bar{b}.c + \bar{a}.b.\bar{c} + a.\bar{b}.\bar{c} + a.b.c$$

$$\text{carry} = b.c + a.c + a.b$$

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



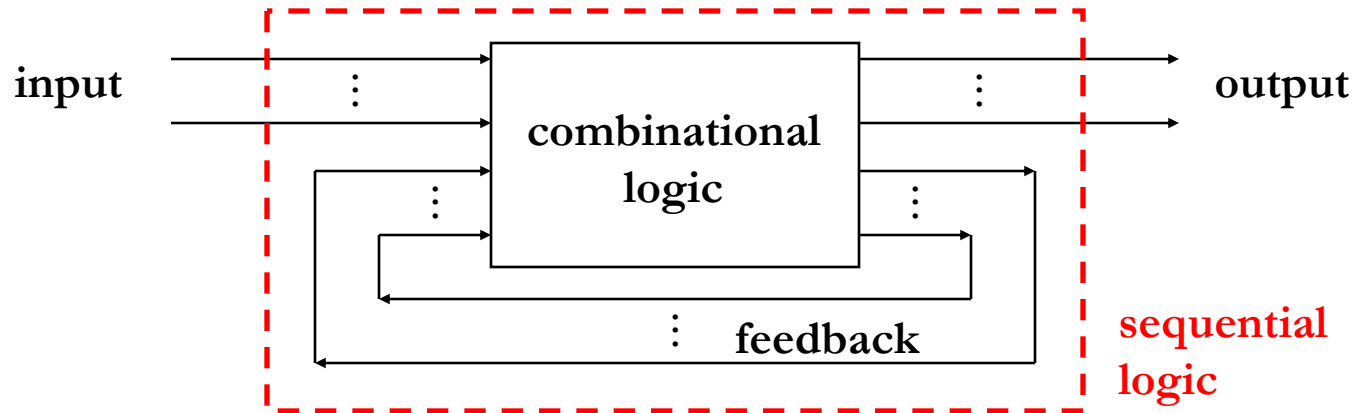
- Carry bits c_i are computed in sequence c_1, c_2, \dots, c_{32} (where $c_{32} = s_{32}$), as c_i depends on c_{i-1}
- Since sum bits s_i also depend on c_i , they too are computed in sequence

Propagation Delays

- Propagation delay = time delay between input signal change and output signal change at the other end
- Delay depends on technology (transistor, wire capacitance, etc.) and number of gates driven by the gate's output (**fan out**)
- e.g.: Sum of products circuits: 3 2-input gate delays (inverter, AND, OR) → very fast!
- e.g.: 32-bit ripple carry adder: 65 2-input gate delays (1 AND + 1 OR for each of 31 carries to propagate; 1 inverter + 1 AND + 1 OR for S_{31}) → slow



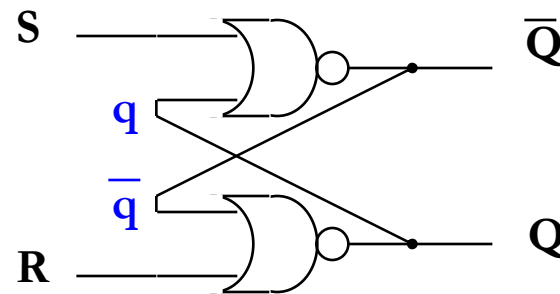
Sequential logic circuits



- Output depends on current inputs as well as past inputs
 - The circuit has memory
- Sequences of inputs generate sequences of outputs \Rightarrow sequential logic

Sequential logic circuits

- For a fixed input and n feedback signals, the circuit can have 2^n possible different behaviours (states)
 - E.g. $n=1 \rightarrow$ one state if feedback signal = 0
one state if feedback signal = 1
- Example: SR latch
 - Inputs: R, S
 - Feedback: q
 - Output: Q

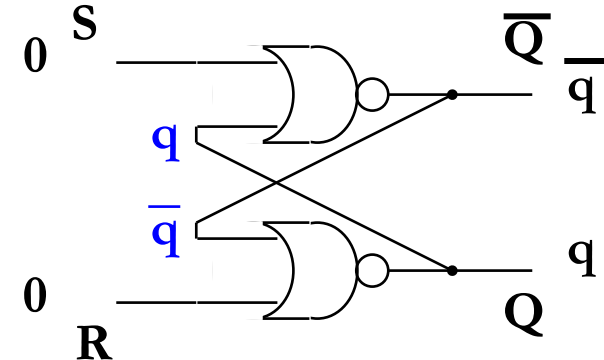


SR Latch

■ Truth table:

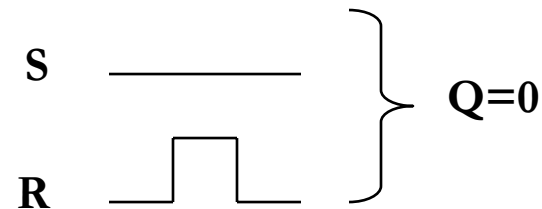
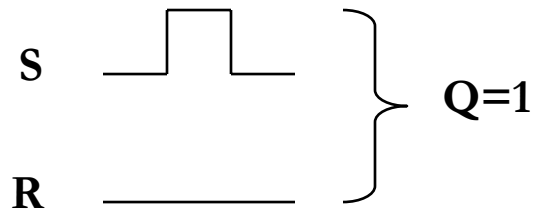
S	R	Q_i
0	0	Q_{i-1}
0	1	0
1	0	1
1	1	u

u=unused



■ Usage: 1-bit memory

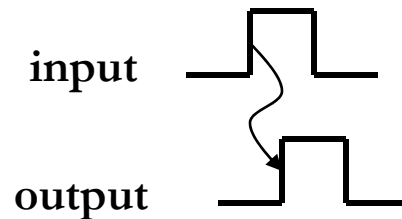
- Keep the value in memory by maintaining $S=0$ and $R=0$
- Set the value in memory to 0 (or 1) by setting $R=1$ (or $S=1$) for a short time



Timing of events

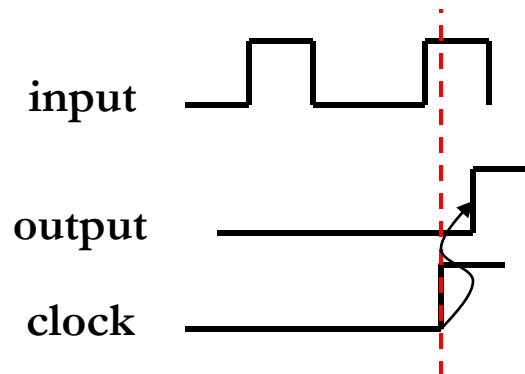
- Asynchronous sequential logic

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

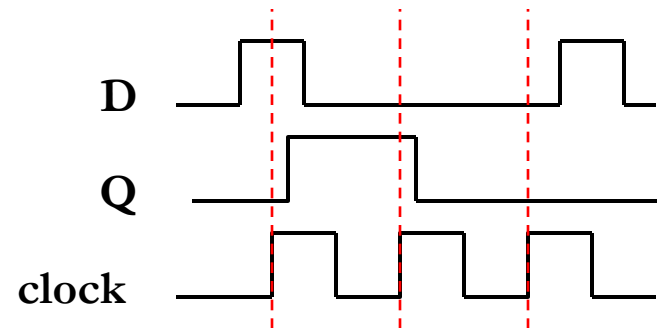
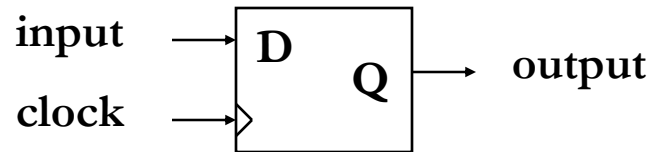


- Synchronous sequential logic

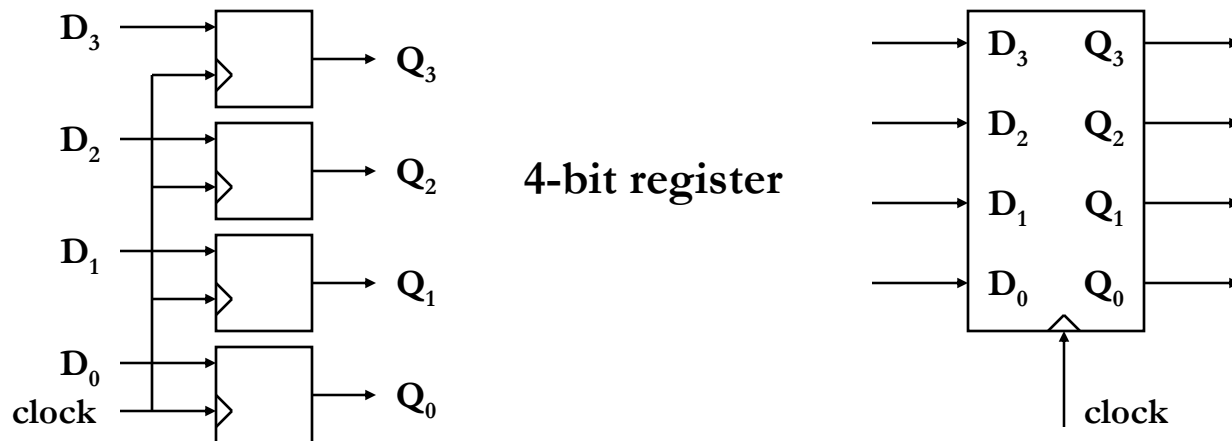
- State (and possibly output) can only change at times synchronized to an external signal → the **clock**



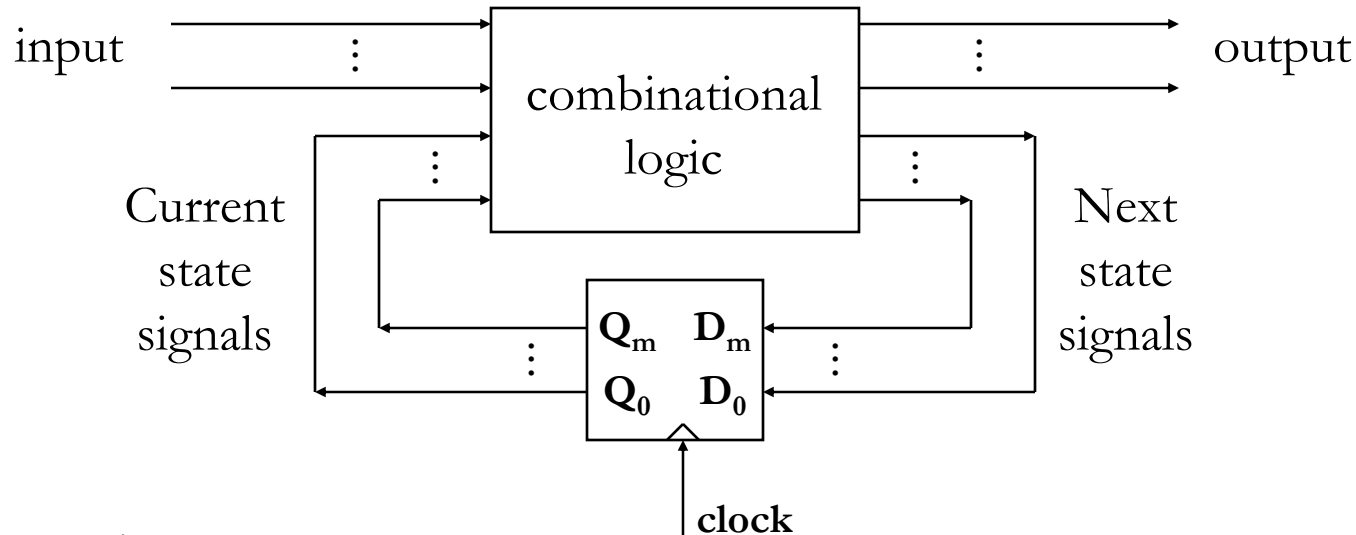
D flip-flop



- **Edge-triggered** flip-flop: on a +ve clock edge, D is copied to Q
- Can be used to build registers:



General sequential logic circuit



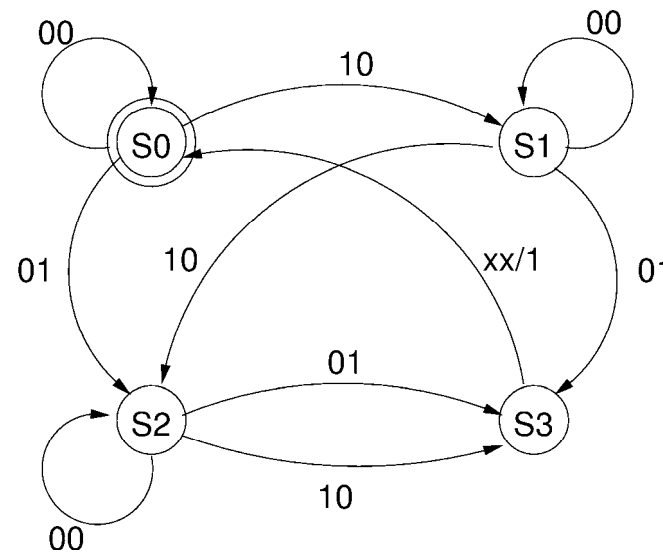
■ Operation:

- 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



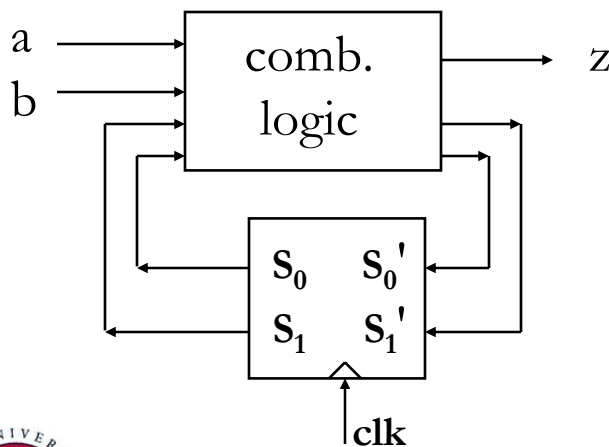
Hardware FSM

- 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 outputs: a,b for 10p, 20p coins respectively
 - Output z asserted when 30p or more has been paid in



FSM implementation

- Methodology:
 - Choose encoding for states, e.g $S_0=00$, ..., $S_3=11$
 - 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	1	1	0	0
0	0	1	0	0	1	
0	1	0	0	0	1	
0	1	0	1	1	1	0
0	1	1	0	1	0	
.....					