

Inf2C - Computer Systems

Lecture 8-9

Logic Design

Boris Grot

School of Informatics
University of Edinburgh



Reminder

- **Coursework 1: due Wed @ 4pm**
- Do:
 - Have correct code
 - Compiles/builds/runs without warnings or errors
 - MIPS & C syntax & semantics are followed
 - Have well-structured code
 - Use functions; no goto's
 - Have readable code
 - Meaningful comments
 - Meaningful names for functions, labels, C variables, etc.



Reminder

- Coursework 1: due Wed @ 4pm
- Don't:
 - Be late!
 - Ask me for extensions
 - UG2 organizer – Dr. Sharon Goldwater – handles these
 - Plagiarize!

I WILL NOT PLAGIARIZE ANOTHER'S WORK
I WILL NOT PLAGIARIZE ANOTHER'S WORK
I WILL NOT PLAGIARIZE ANOTHER'S WORK
I WILL NOT PLAGIARIZE ANOTHER'S WORK
I WILL NOT PLAGIARIZE ANOTHER'S WORK
I WILL NOT PLAGIARIZE

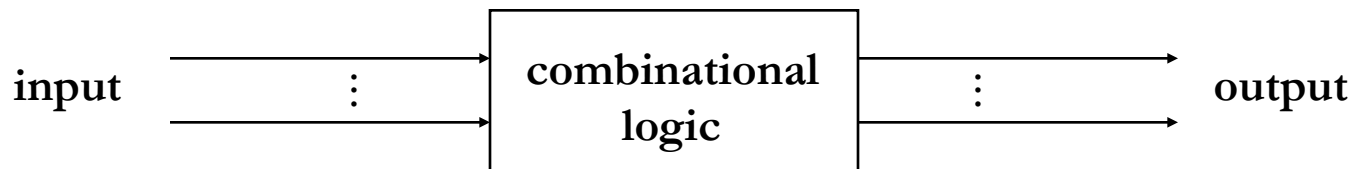


Logic design overview

Binary digital logic circuits:

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

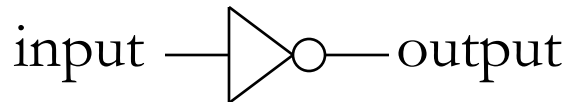
Combinational logic: output depends only on the current inputs
(no memory of past inputs)



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

Combinational logic circuits

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



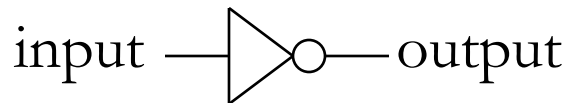
IN	OUT
0	1
1	0

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

Truth table

Combinational logic circuits

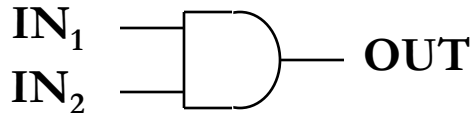
- Inverter (or NOT gate): 1 input and 1 output
“invert the input signal”



IN	OUT
0	1
1	0

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

- AND gate: 2 inputs and 1 output
“output 1 only if both inputs are 1”

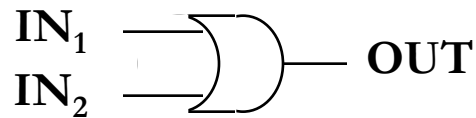


IN_1	IN_2	OUT
0	0	0
0	1	0
1	0	0
1	1	1

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

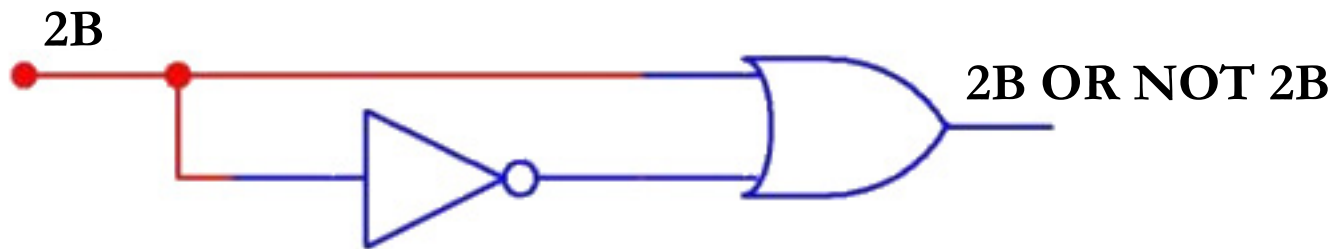
Combinational logic circuits

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



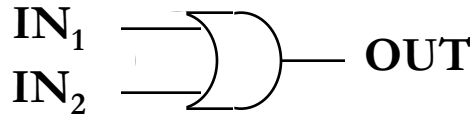
IN ₁	IN ₂	OUT
0	0	0
0	1	1
1	0	1
1	1	1

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



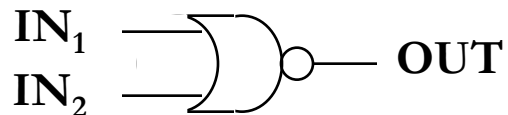
Combinational logic circuits

- 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 = IN₁ + IN₂

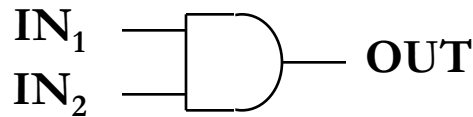
- NOR gate: “output 1 if no input is 1” (NOT OR)

	IN ₁	IN ₂	OUT
	0	0	1
	0	1	0
	1	0	0
	1	1	0

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

Combinational logic circuits

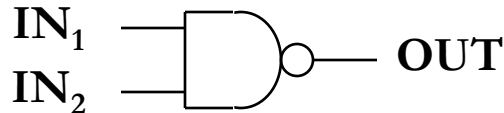
- AND gate: “output 1 if both inputs are 1”



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

$$OUT = IN_1 \cdot IN_2$$

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

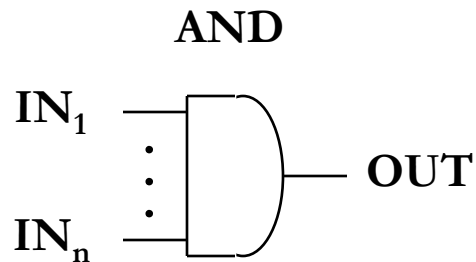


IN_1	IN_2	OUT
0	0	1
0	1	1
1	0	1
1	1	0

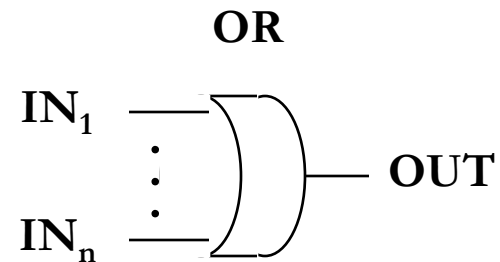
$$OUT = \overline{IN_1 \cdot IN_2}$$

Combinational logic circuits

- Multiple-input gates:



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



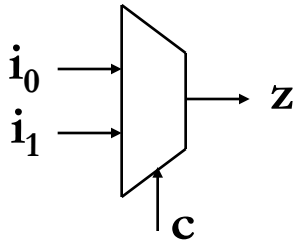
$\text{OUT} = 1$ if any $\text{IN}_i = 1$

Combinational logic circuits

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

Multiplexer (mux)

- Multiplexer: a circuit for selecting one of multiple inputs



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

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

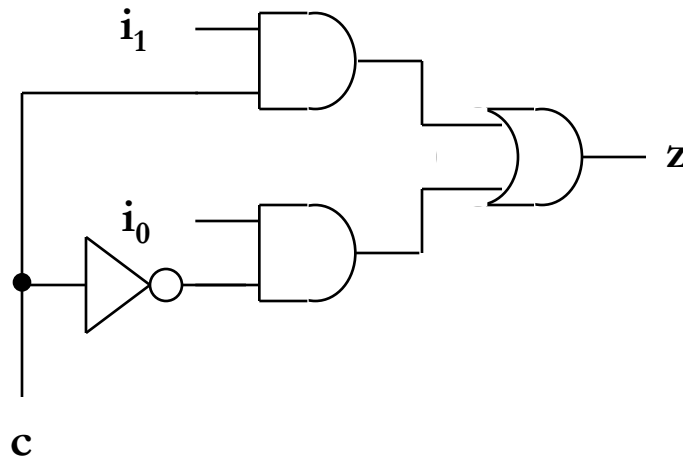
$$\begin{aligned} z &= \bar{c}.i_0.\bar{i}_1 + \bar{c}.i_0.i_1 + c.\bar{i}_0.i_1 + c.i_0.i_1 \\ &= \bar{c}.i_0.(\bar{i}_1 + i_1) + c.(\bar{i}_0 + i_0).i_1 \\ &= \bar{c}.i_0 + c.i_1 \end{aligned}$$

minimized

“sum of products form”

A multiplexer implementation

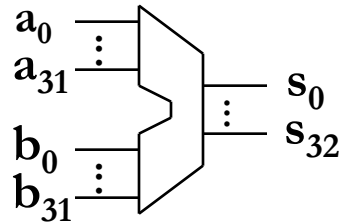
- Sum of products form: $i_1 \cdot c + i_0 \cdot \overline{c}$
 - Can be implemented with 1 inverter, 2 AND gates & 1 OR gate:



- 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



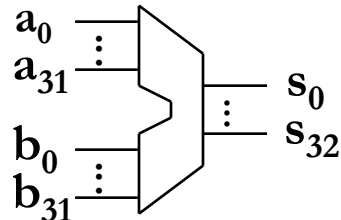
64 inputs \rightarrow too complex for
sum of products

- Idea: modularize!

- Design a generic 1-bit adder block
- Replicate it N number of times for an N-bit adder

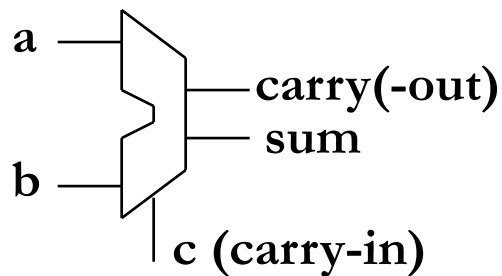
Arithmetic circuits

- 32-bit adder



64 inputs \rightarrow too complex for
sum of products

- Full adder:



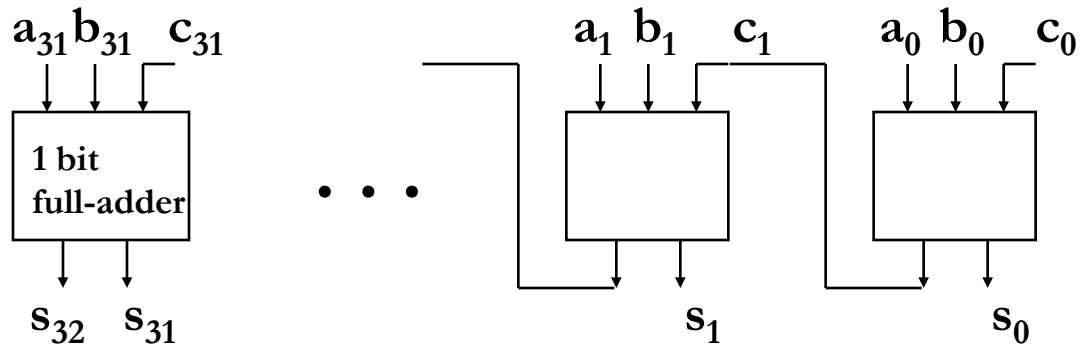
$$\text{sum} = \overline{a}.\overline{b}.c + \overline{a}.b.\overline{c} + a.\overline{b}.\overline{c} + a.b.c$$

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

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:
 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 \rightarrow AND \rightarrow OR \Rightarrow 3 gate delays. Fast!
- What's the delay of a 32-bit ripple carry adder?
 - 65 gate delays \rightarrow slow
 - AND2 + OR3 for each of 31 carries to propagate;
followed by NOT + AND3 + OR4 for S_{31}

Practice problem:

Design a circuit that, given a 4-bit unsigned input, outputs:

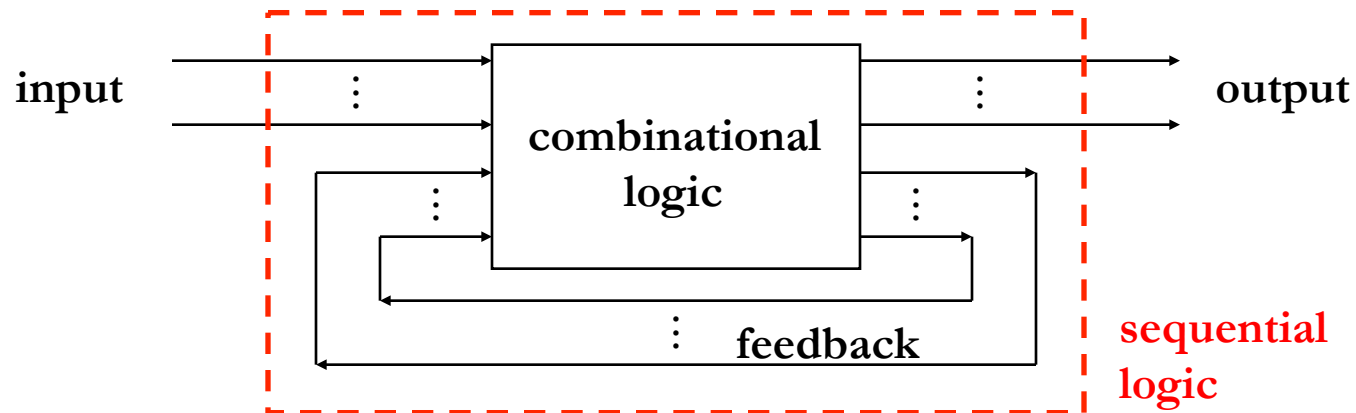
1 - if the input value is ≥ 7

0 - otherwise

What is the propagation delay of the circuit?

What is the delay if only 1- and 2-input gates are allowed?

Sequential logic circuits

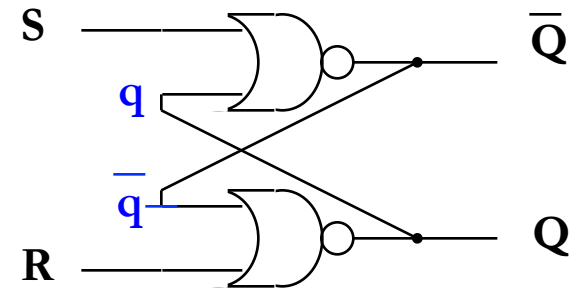


- Output depends on current AND past inputs
 - The circuit has memory
- Sequences of inputs generate sequences of outputs \Rightarrow sequential logic
 - With n feedback signals \rightarrow up to 2^n stable states

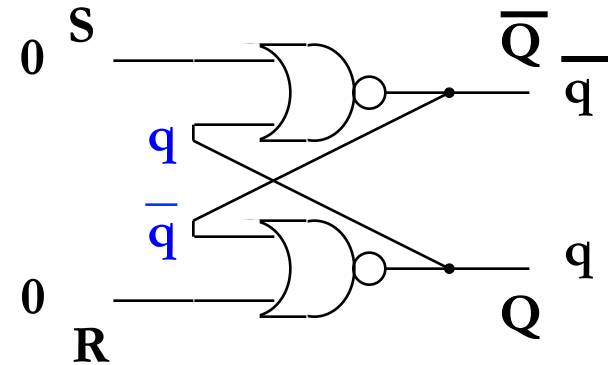
SR Latch: the basic sequential circuit

SR latch

- Inputs: R, S
- Feedback: q, \bar{q}
- Output: Q



SR Latch

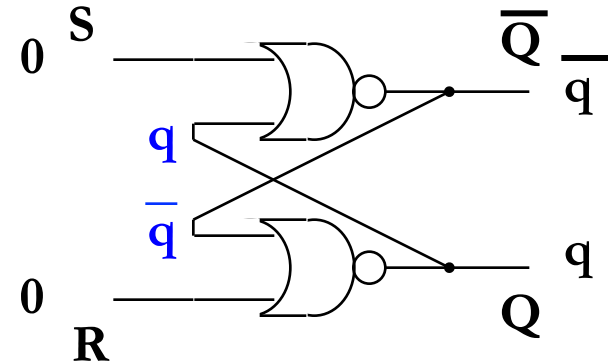


SR Latch

■ Truth table:

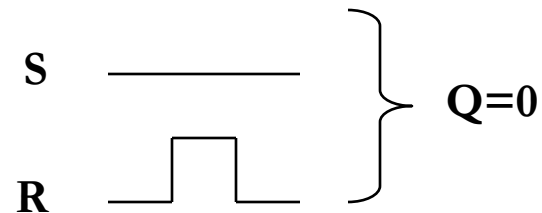
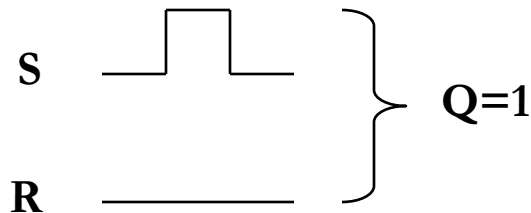
S	R	Q_i
0	0	Q_{i-1}
0	1	0
1	0	1
1	1	inv

inv=invalid



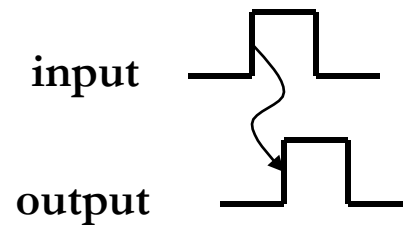
■ Usage: 1-bit memory

- 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

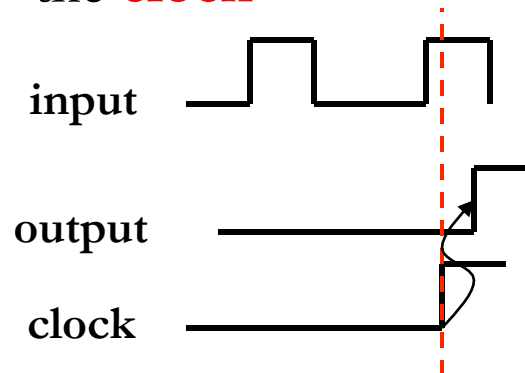


Timing of events

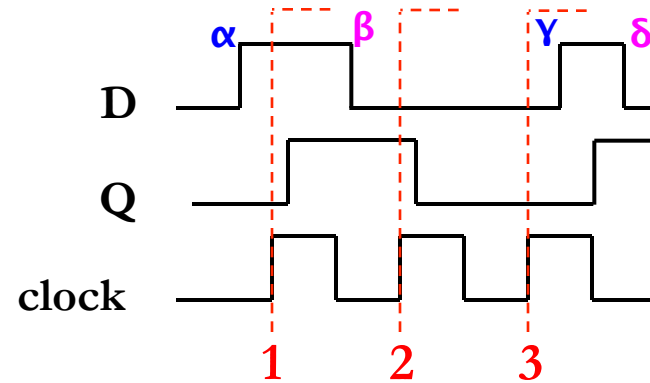
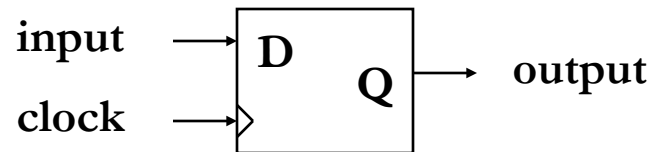
- Asynchronous sequential logic
 - State (and possibly output) of circuit changes whenever inputs change



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

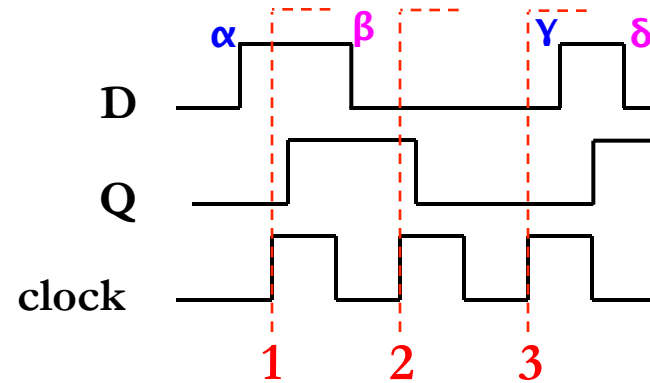
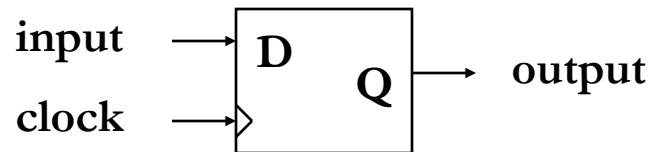


Using clock to build a memory element

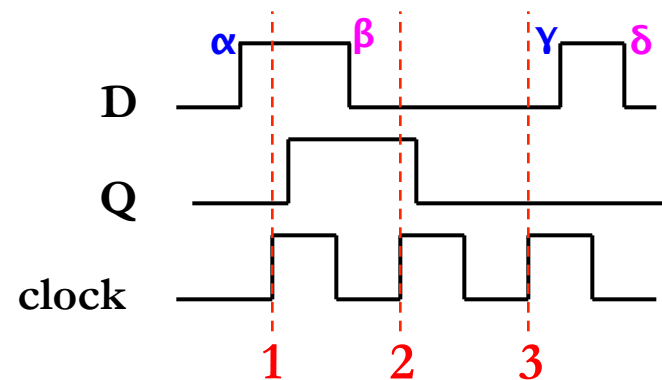


- **Level-triggered D latch:** whenever clock is 1, D is propagated to Q

Using clock to build a memory element

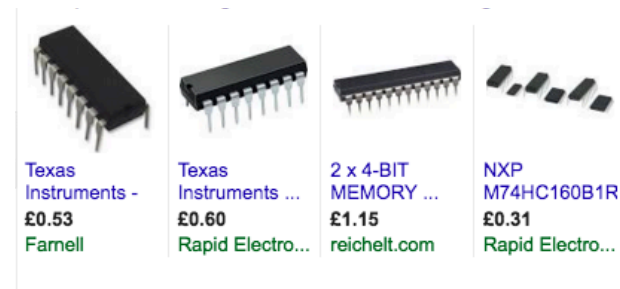
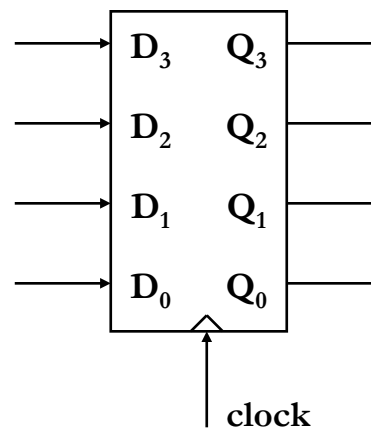
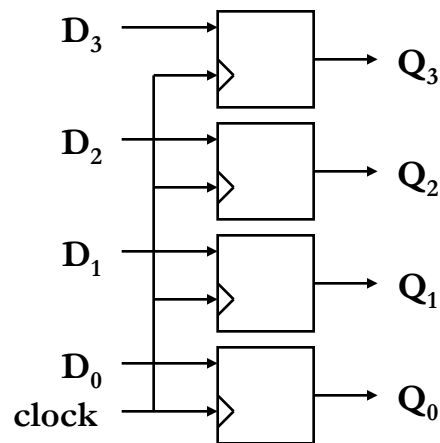


- **Level-triggered D latch:** whenever clock is 1, D is propagated to Q
- **Edge-triggered D flip-flop:** on a **positive clock edge**, D is propagated to Q



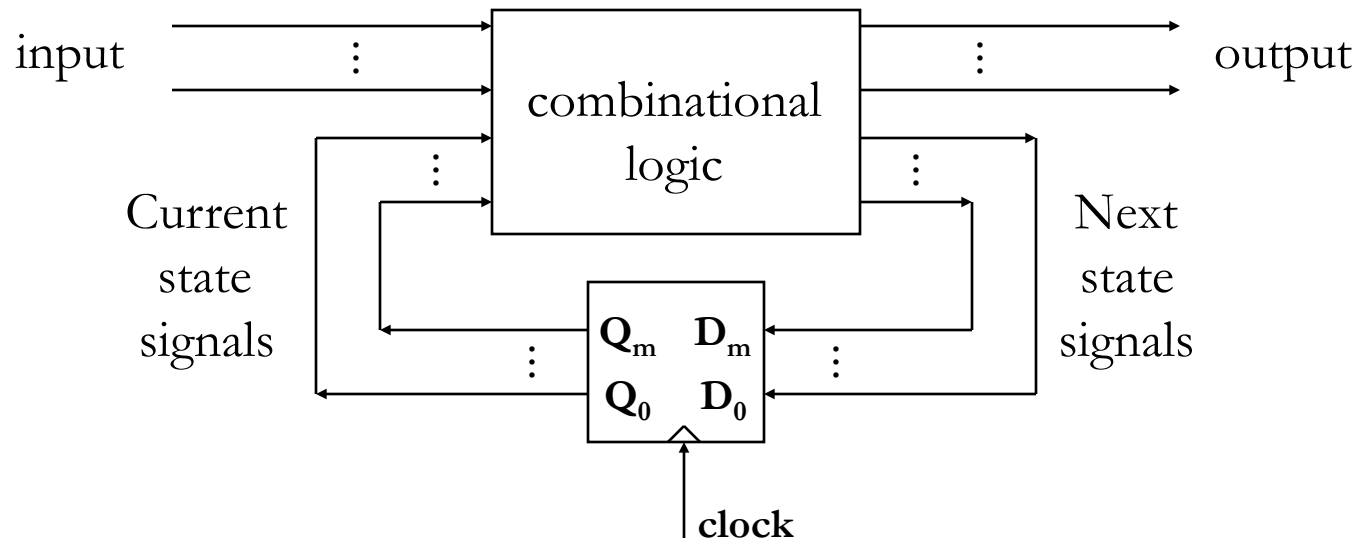
Register

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



[PDF] [4-Bit D-Type Registers With 3-State Outputs \(F](http://www.ti.com/lit/gpn/SN74LS173A)
www.ti.com/lit/gpn/SN74LS173A ▼
50 MHz description. The '173 and 'LS173A 4-bit registers include. D-
3-state outputs capable of driving highly capacitive.

General sequential logic circuit

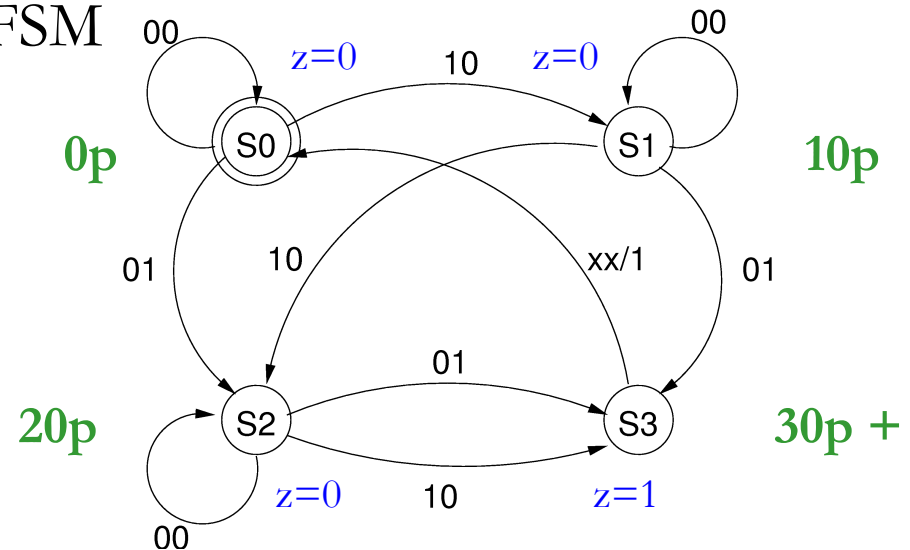


■ Operation:

- 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

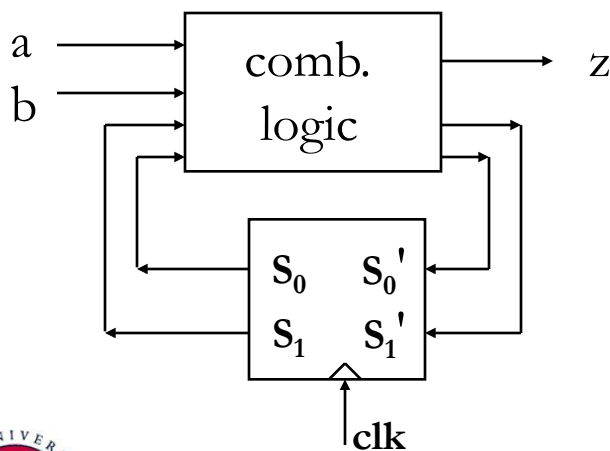
Hardware FSM

- 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 $S_0=00$, ..., $S_3=11$
 - 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	

.....