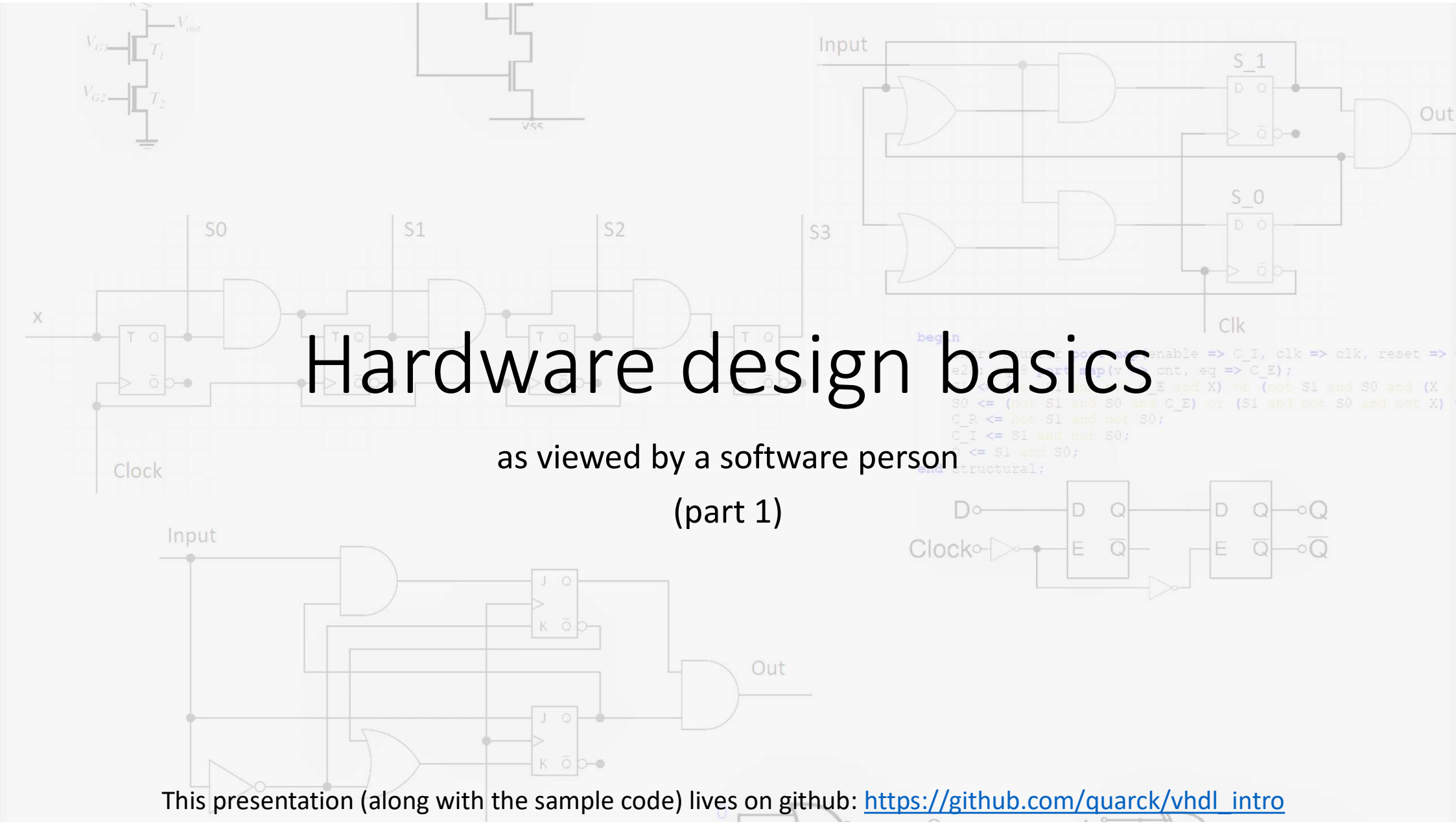


Hardware design basics

as viewed by a software person

(part 1)

This presentation (along with the sample code) lives on github: https://github.com/quarck/vhdl_intro

Motivation?

- Expected words about FPGA and how we use it…
- From the software developer perspective, CPU is a black-box that is:
  - Executing instructions
  - Loading data from the memory
  - Storing results back to memory
  - Accessing IO devices
  - Etc
  - What's happening inside a black-box is a MAGIC

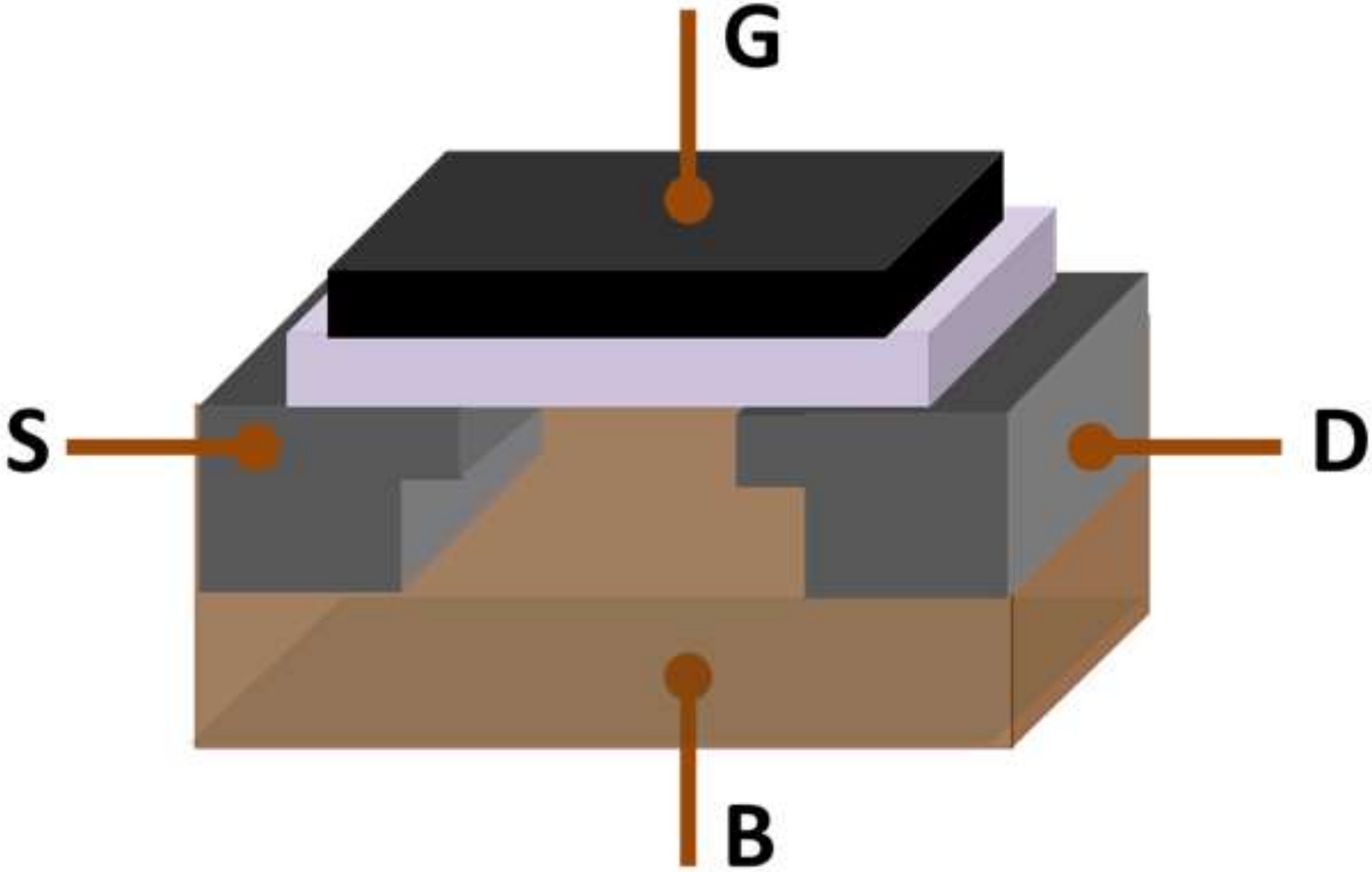

CPU HARDWARE IS A MAGIC!
xor eax, eax
cmpxchg
add edx, ecx
lea edx, [edi+358h]
mov eax, ebx

Digital circuits - basics

- Works with two distinguishable signal levels: LOW and HIGH
- LOW represents logical 0, HIGH represent logical 1
- The whole circuit is a majestic play of zeroes and ones
- TTL typical voltage levels:
  - voltage between 0V and 1.5V represents LOW
  - voltage between 3.5V and 5V represents HIGH
  - Anything in between 1.5V and 3.5V is invalid (and can cause circuit damage)
- CMOS typical voltage levels:
  - voltage between 0V and 0.8V represents LOW
  - voltage between 2V and 3.3V represents HIGH
  - Anything in between 0.8V and 2V is invalid (and can cause circuit damage)

Transistor – the basic element of the circuit

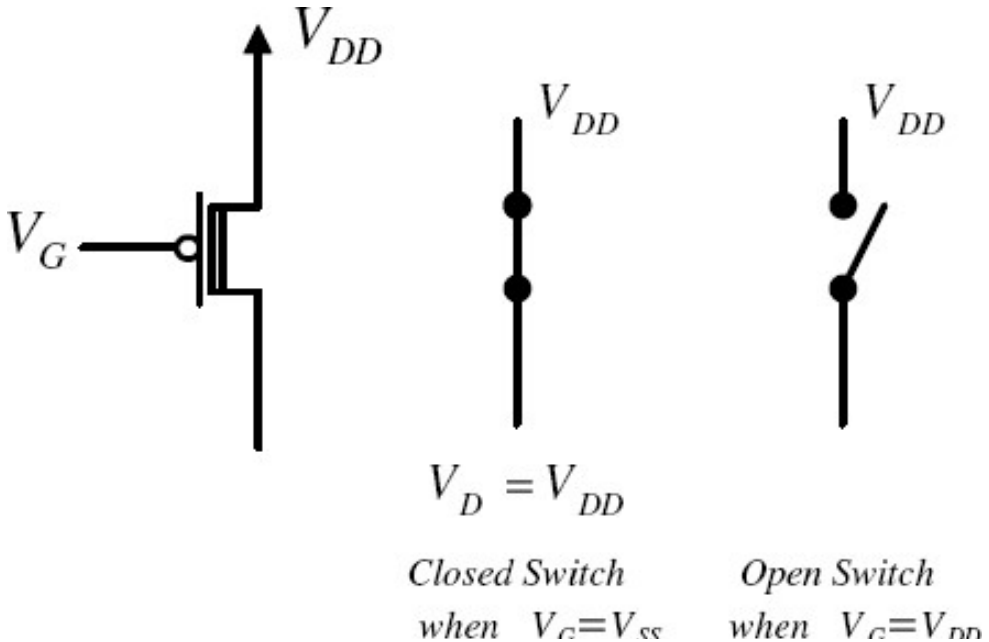
- In a nutshell transistor is an electronically controlled “switch”
- Multiple types of transistors do exist
- MOSFET (metal oxide semiconductor field effect transistor) transistors are the most commonly used in a modern day electronics

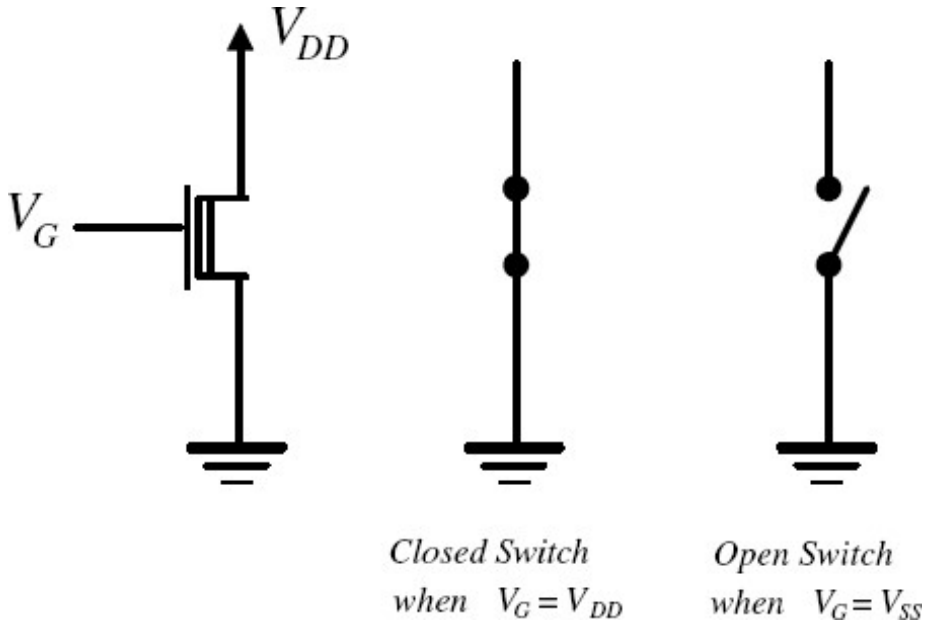


MOSFET transistor © Wikipedia

MOSFET

- There are two sub-types of MOSFETs:
- pMOS
- nMOS





One level of abstraction up – logical gates

- On a higher level, digital circuits are built of logical gates
- Logical gates are built from transistors (and other elements in some cases)
- Logical gates have one or more inputs
- Let’s use letters A, B… (or a, b, …) to represent values of the inputs
- Logical gates have just one output, let’s represent its value with ‘F’

Logical gates

Gate:	Function	Symbol(s)
NOT	F = **not** A	
AND	F = A **and** B **and** …	
OR	F = A **or** B **or** …	
XOR	F = A **xor** B **xor** …	
NAND	F = **not** (A **and** B **and** …)	
NOR	F = **not** (A **or** B **or** …)	
XNOR	F = **not** (A **xor** B **xor** …)	

Image credits – Wikipedia

How are gates implemented?

CMOS NOT gate

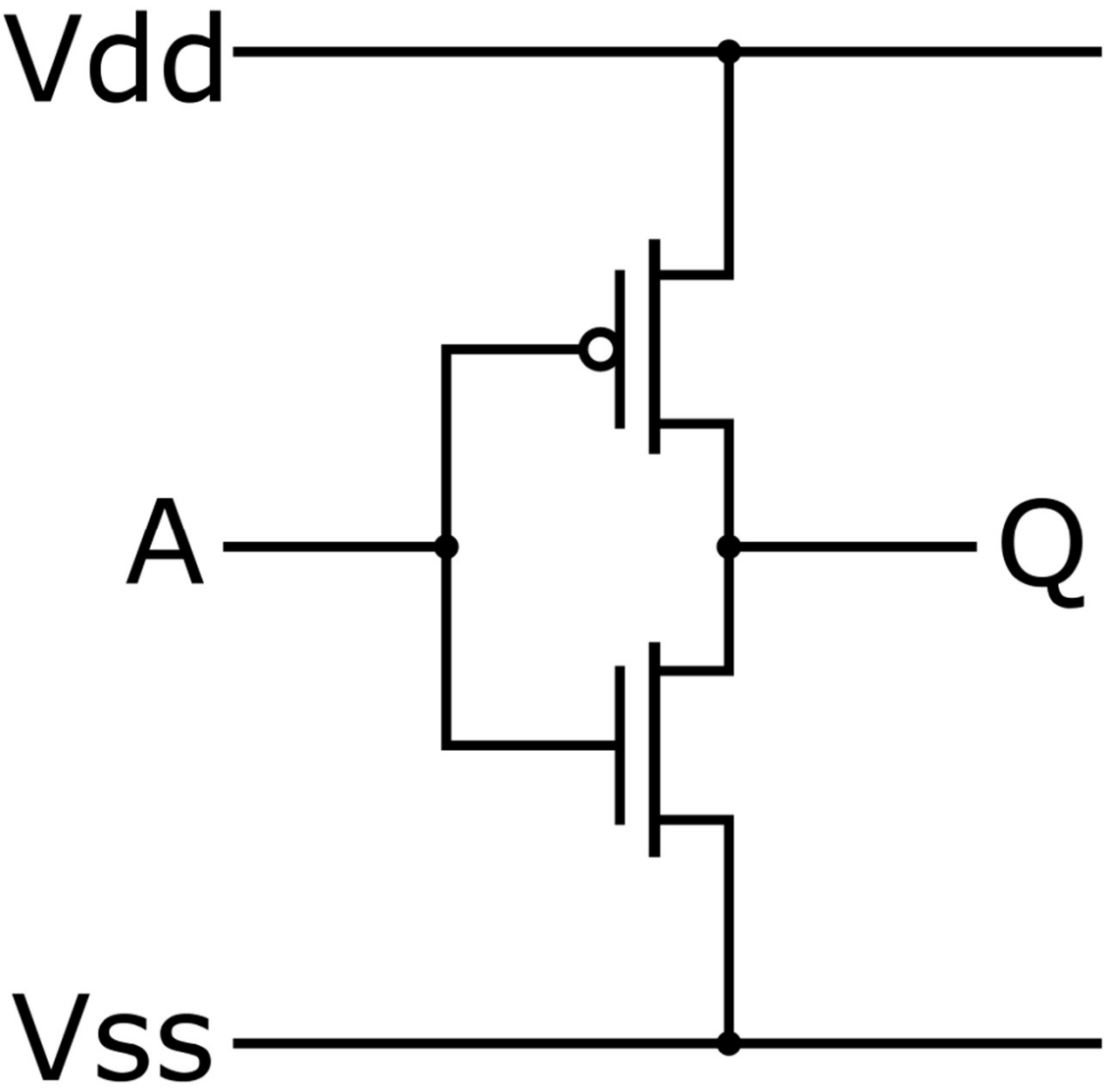
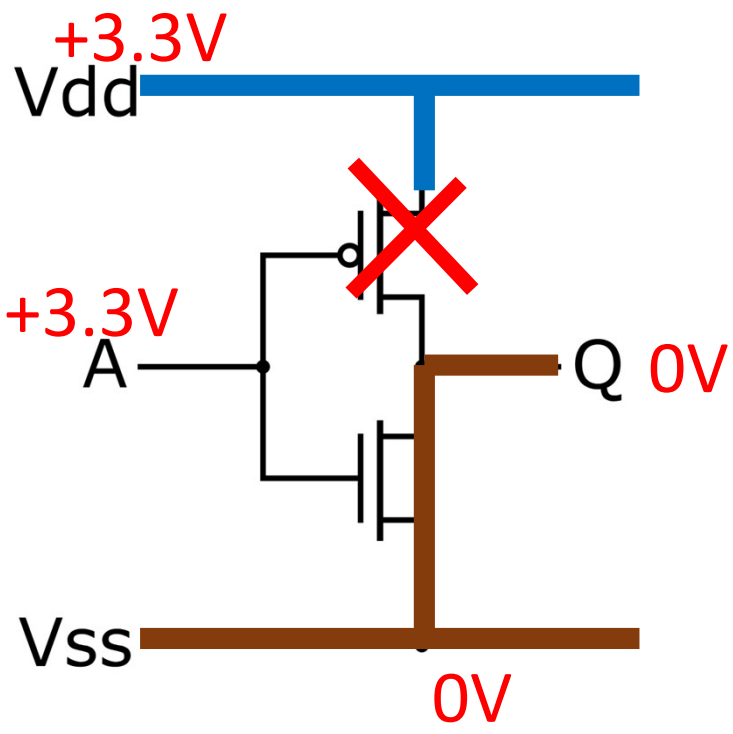
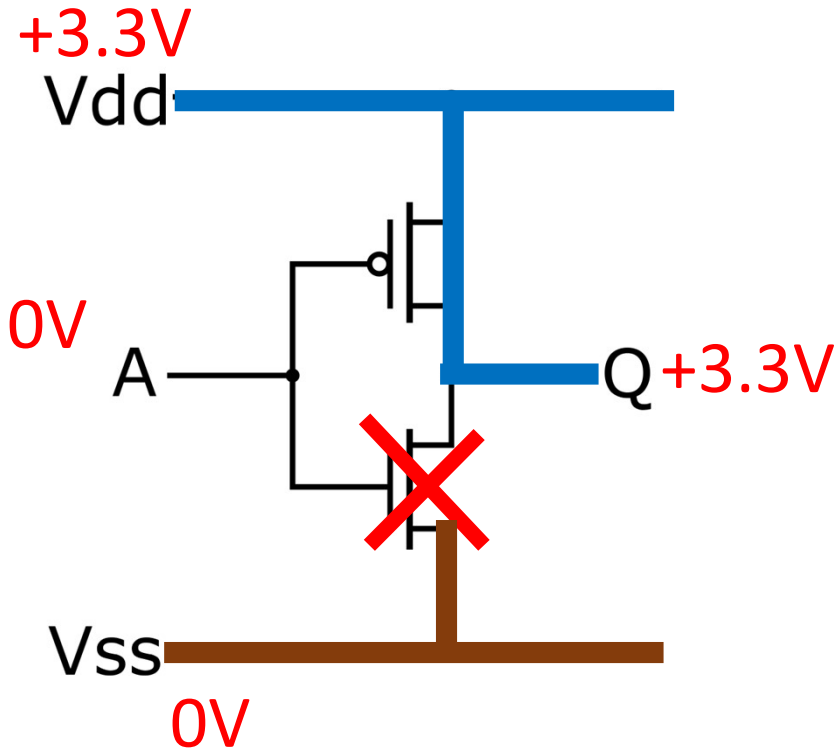


Image credits – Wikipedia

CMOS NOT gate – operation explained





Simulation video: https://www.youtube.com/watch?v=RJP3yPLblQ4

Image credits – Wikipedia

CMOS NAND gate

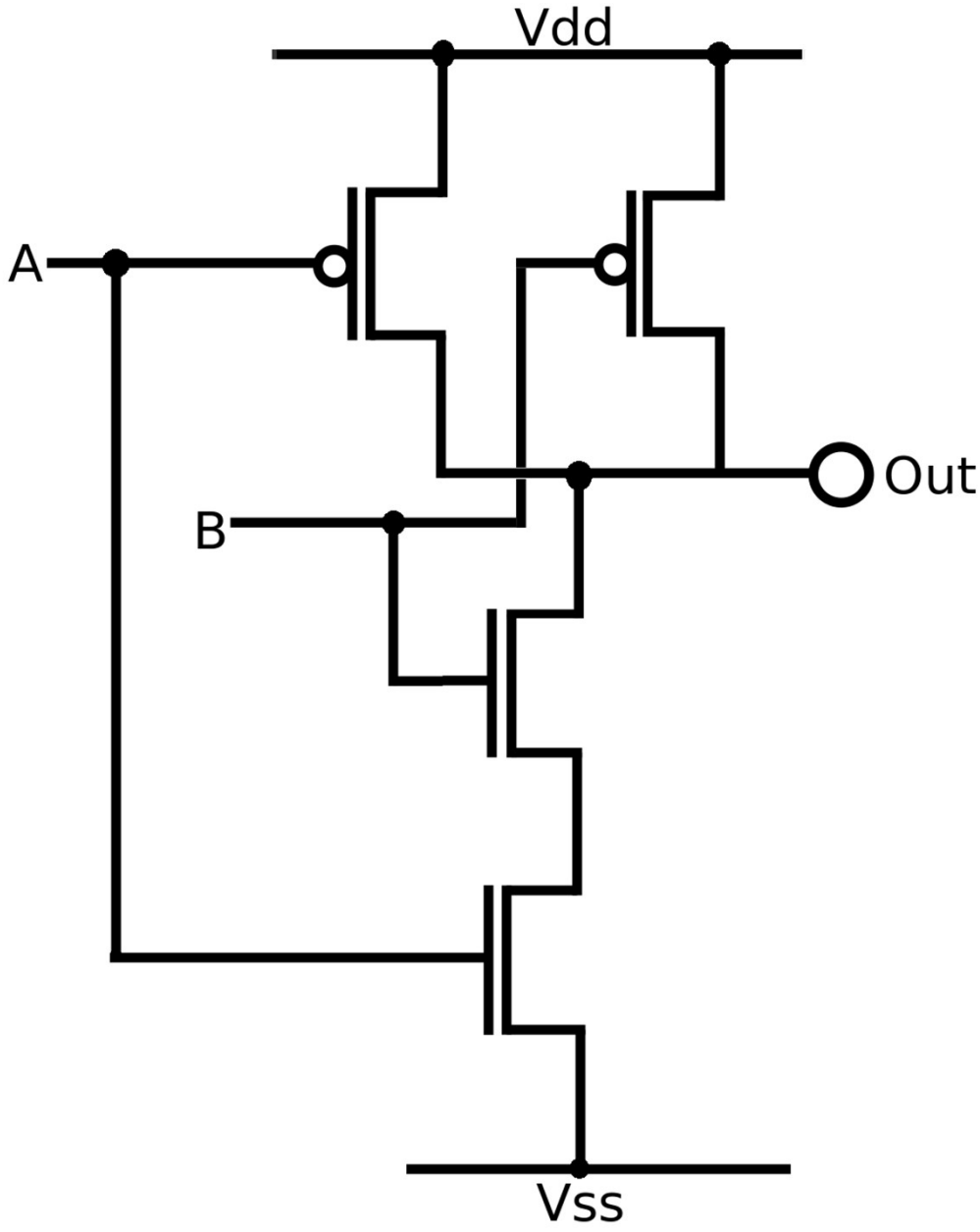


Image credits – Wikipedia

CMOS NAND gate – operation explained

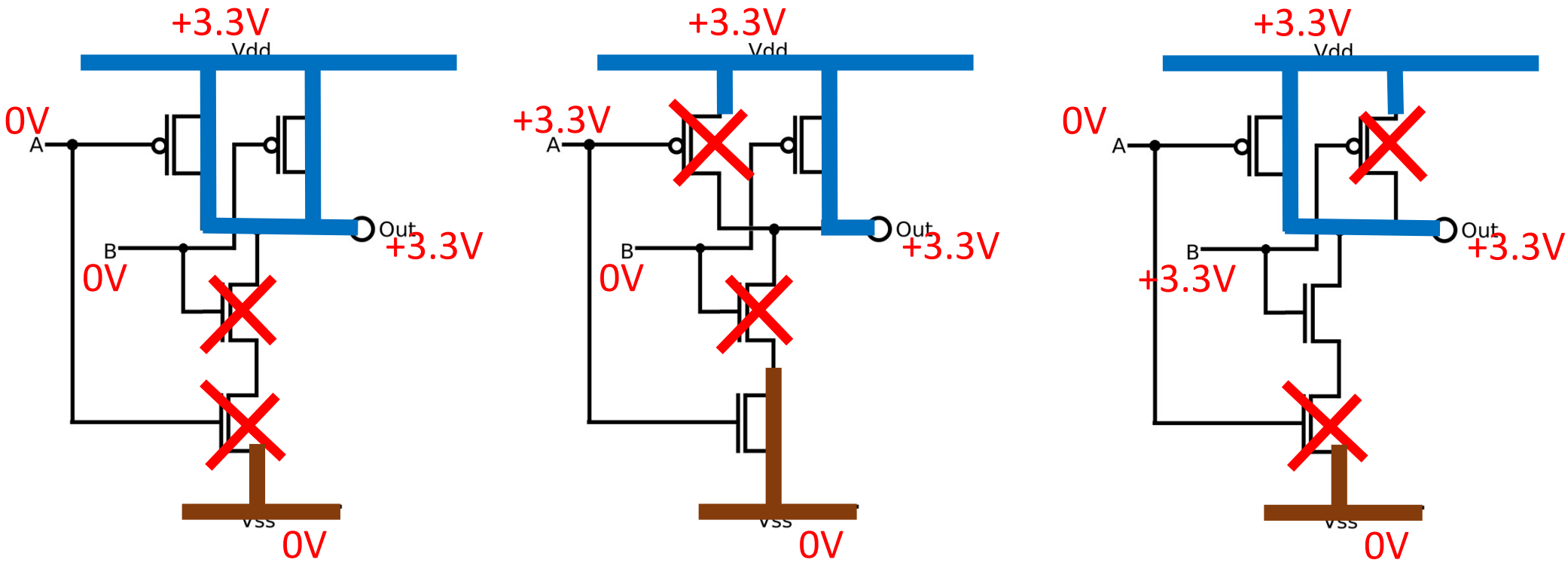
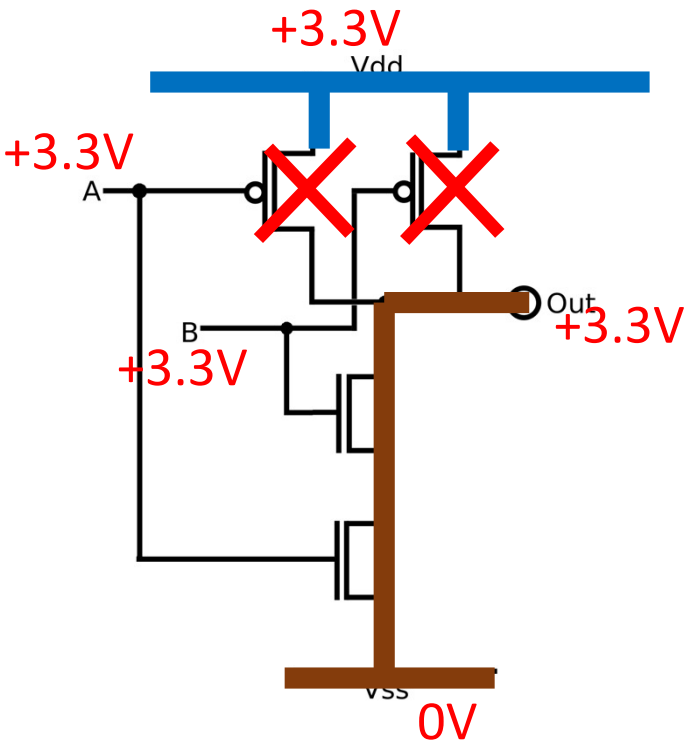


Image credits – Wikipedia

CMOS NAND gate – operation explained



Simulation video: https://www.youtube.com/watch?v=Yjwf5rUbDxs

Image credits – Wikipedia

NAND/NOR logic

- Any other gate type could be represented only with NAND gates only
- Similar applies to NOR gates
- NAND/NOR gates are easier to manufacture in hardware
- Thus, it is often easier to build the whole chip with only NAND or with only NOR gates

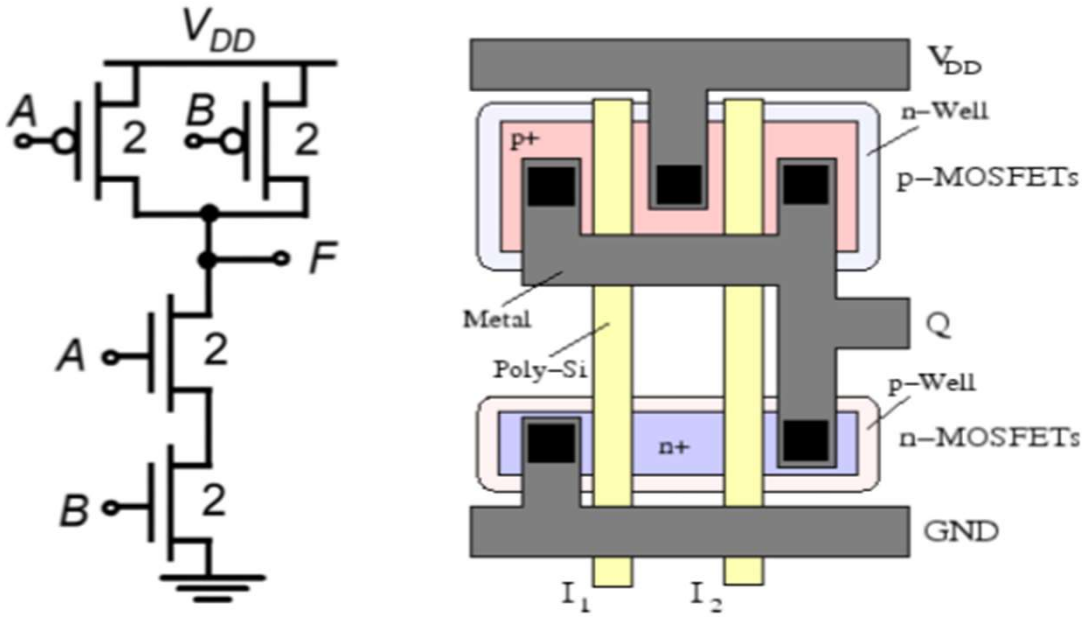
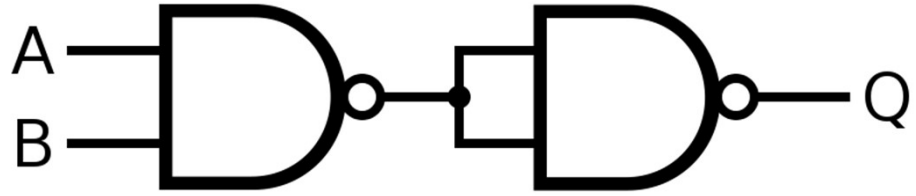
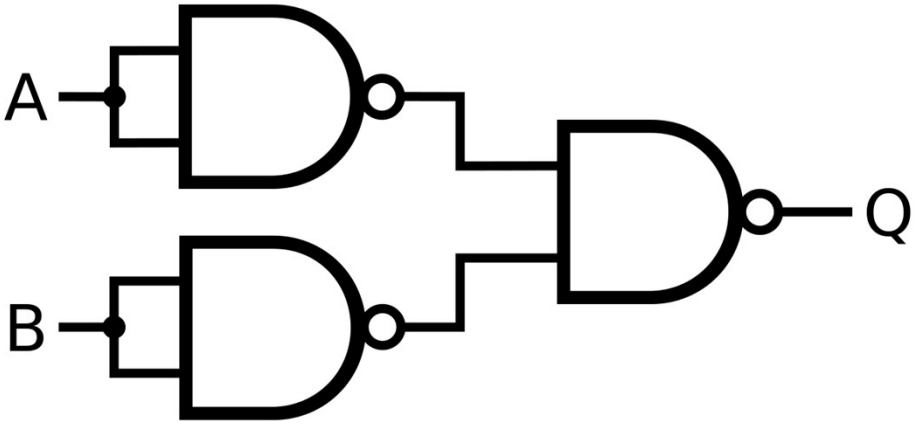


Image credit: https://www.tutorialspoint.com/vlsi_design/vlsi_design_combinational_mos_logic_circuits.htm

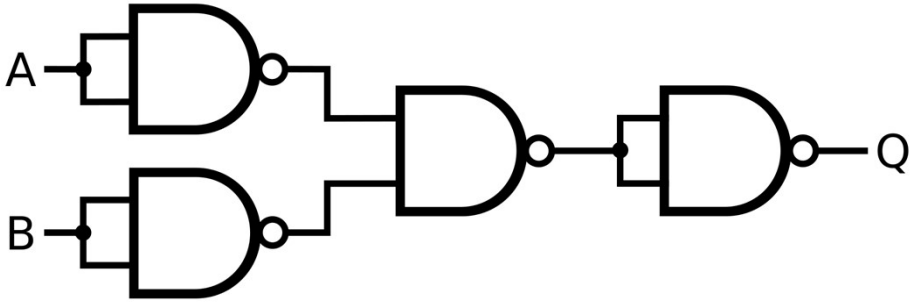


==AND

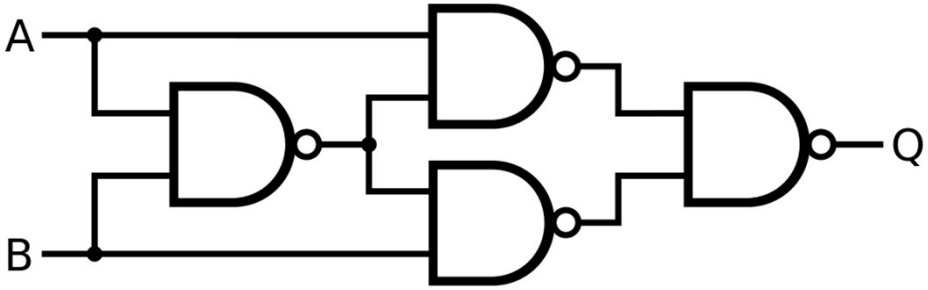


==OR

Image credits: Wikipedia: https://en.wikipedia.org/wiki/NAND_logic



==NOR



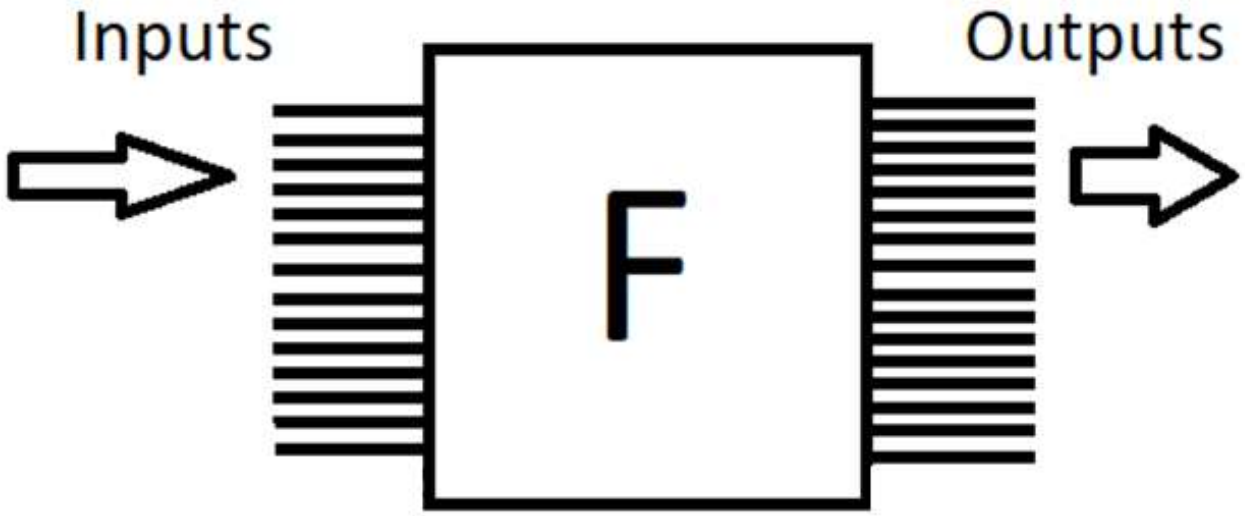
==XOR

Image credits: Wikipedia: https://en.wikipedia.org/wiki/NAND_logic

Combination circuits

Combinational circuits

- A “black box” implementing certain logical function
- Output depends only on the current inputs (after the signal propagates through the gates)



Combinational circuits

- Combinational circuit can be fully defined in a table form
- Let’s consider the following function defined by the table:

A	B	C	X	Y
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

- For every combination of input values (A, B, C), we explicitly define the output values X and Y.

A	B	C	X	Y
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

In a programming language word we would have defined this function as something like:

```
X = ((A==0 && B==0 && C == 1) ||
     (A==0 && B==1 && C == 0) ||
     (A==1 && B==0 && C == 0) ||
     (A==1 && B==1 && C == 1)) ? 1 : 0;
```

Or treating A, B, C, X, Y as Booleans (and using math notation):

$$X = \bar{A}\bar{B}C + \bar{A}B\bar{C} + A\bar{B}\bar{C} + ABC$$

And similar way we can write Y:

$$Y = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$$

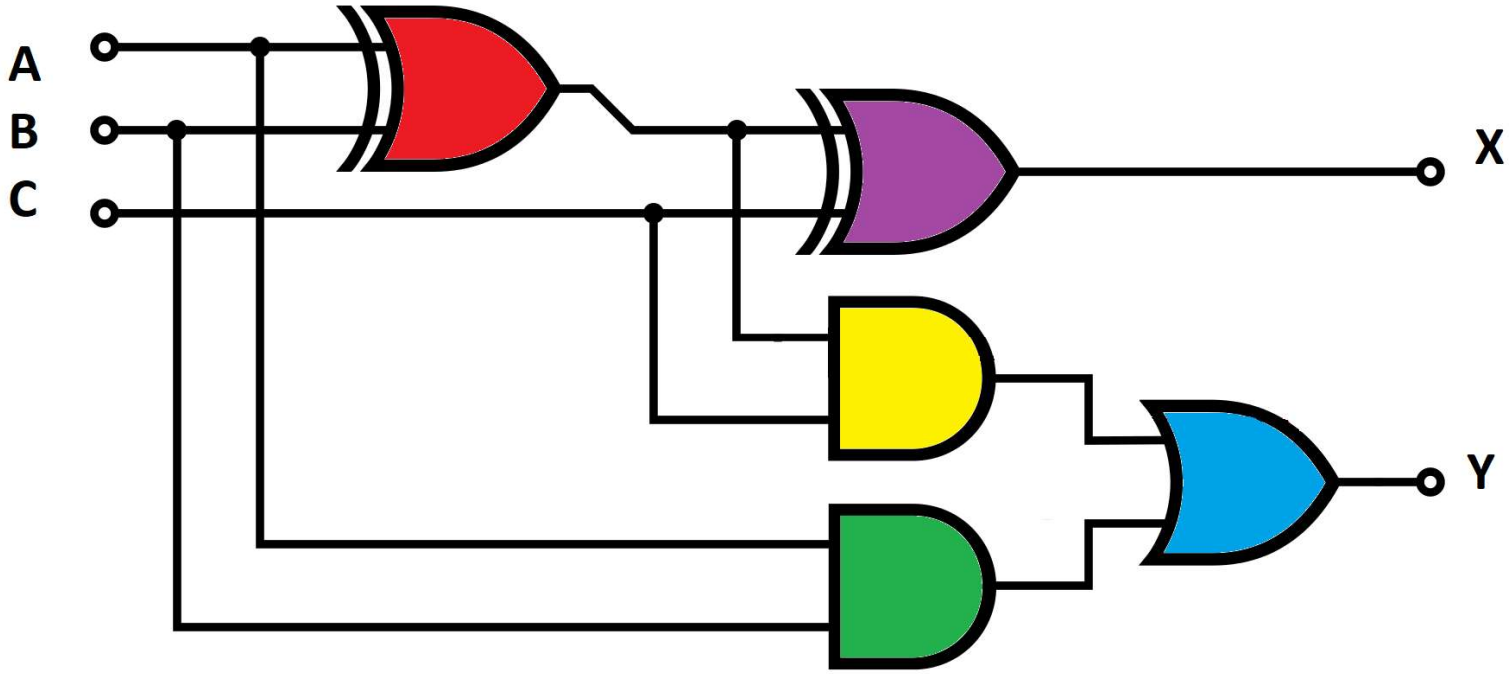
- We can already build the circuit based on equations above directly mapping and / or functions into gates
- But these are sub-optimal and would result inefficient circuit
- We must optimize the expressions first
- Many ways do exist, not going to deep details, let's just use the simple formula transformation for our simple case:

$$
\begin{aligned}
X &= \bar{A}\bar{B}C + \bar{A}B\bar{C} + A\bar{B}\bar{C} + ABC \\
&= (\bar{A}\bar{B} + AB)C + (\bar{A}B + A\bar{B})\bar{C} \\
&= \left(\overline{A \oplus B}\right)C + (A \oplus B)\bar{C} \\
&= A \oplus B \oplus C
\end{aligned}
$$

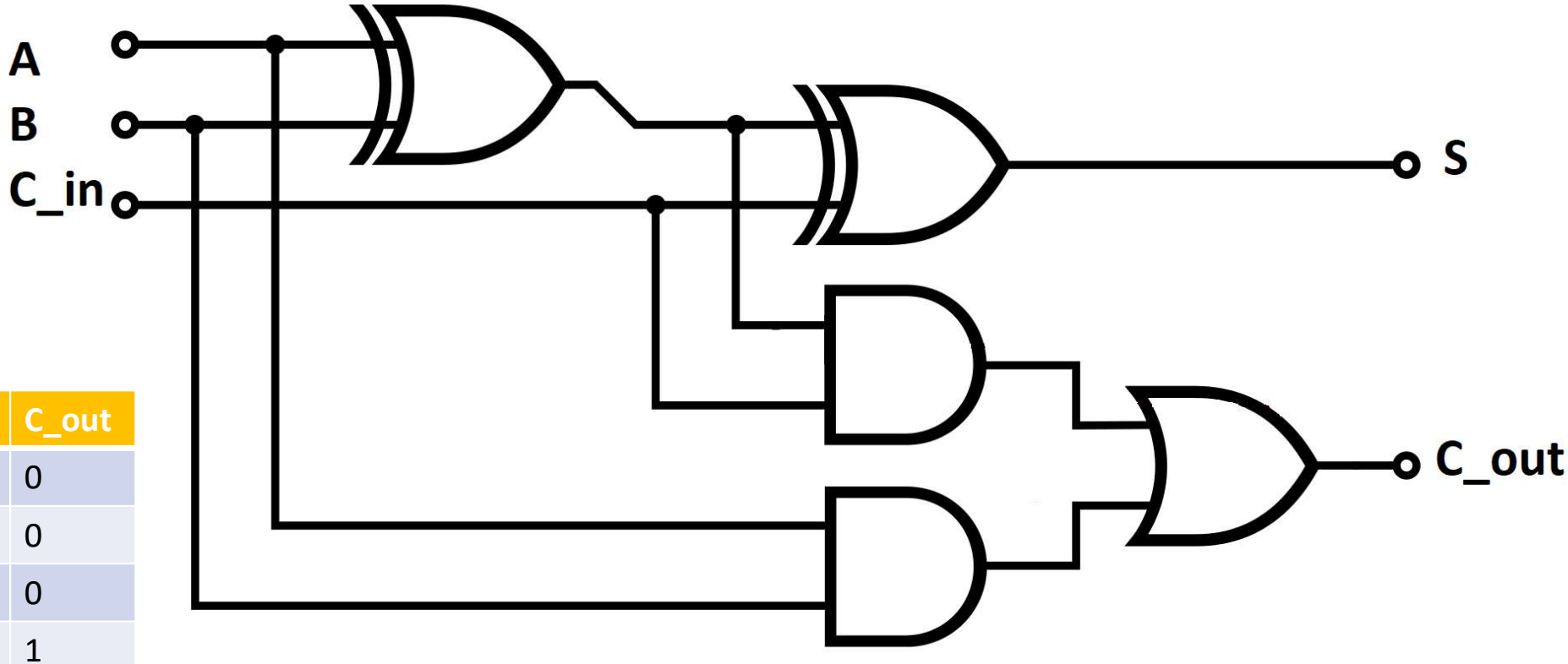
$$Y = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$$
$$= (\bar{A}B + A\bar{B})C + AB(\bar{C} + C)$$
$$= (A \oplus B)C + AB$$

$$X = (A \oplus B) \oplus C$$

$$Y = (A \oplus B) and C + A and B$$



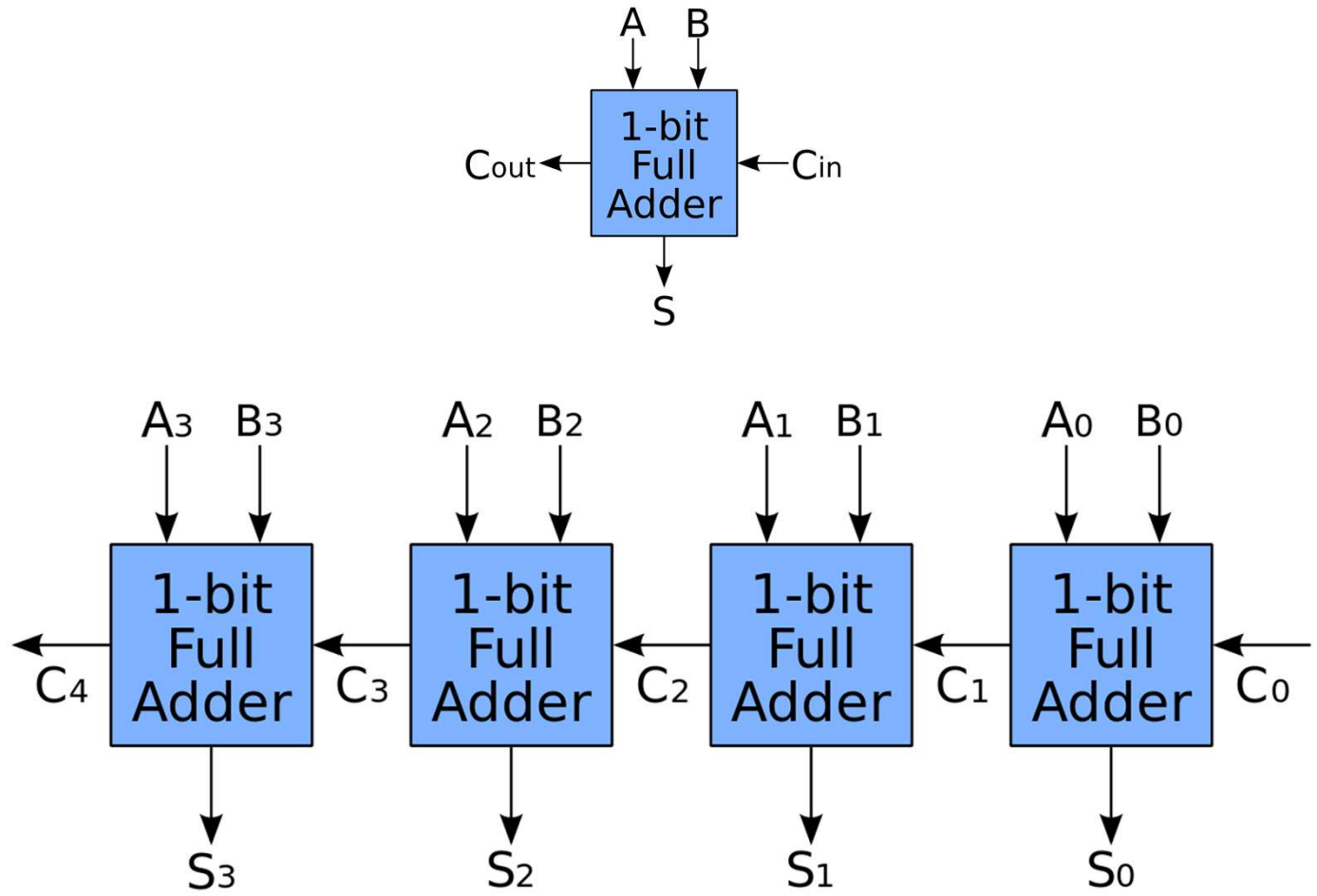
We've just created a binary full adder circuit!



A	B	C_in	S	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

$$C\_out,S = A + B + C\_in$$

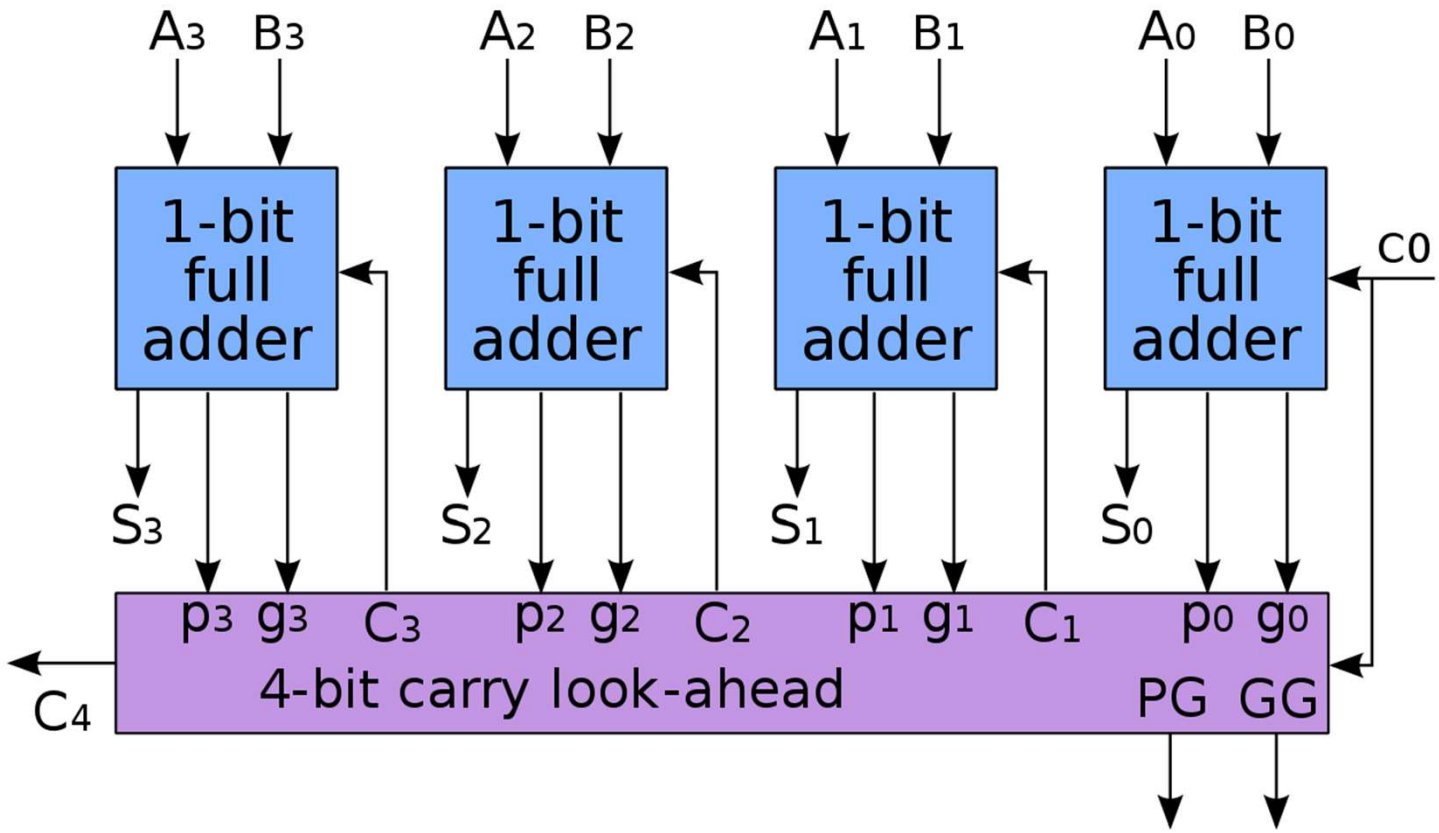
Combining 1-bit adders and N-bit



It is slow

Image credits: https://en.wikipedia.org/wiki/Adder_(electronics)#Ripple-carry_adder

Carry look ahead adder



It is faster

Image credits: https://en.wikipedia.org/wiki/Carry-lookahead_adder

Another example: multiplexer - MUX

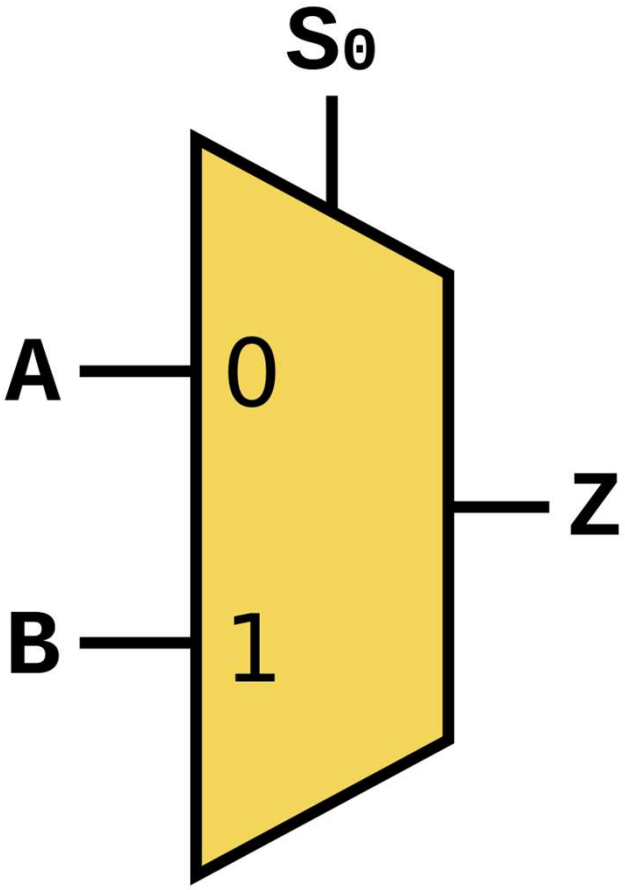
Let's define a combinatory circuit that would:

- Has three inputs: signals A, B and S
- Has one output Z, depending on S:
- If S == 0, output equals A
- If S == 1, output equals B

Such circuit has its own given name:

multiplexer or **MUX**

And it has its own symbol



MUX

- In table form we can define the MUX operation as follows:

A	B	S	Z
0	-	0	0
1	-	0	1
-	0	1	0
-	1	1	1

- Note that we only have 4 rows instead of expected 8, and there are also subtly coloured “-” cells,
- Symbol “-” means “don’t care”, or “any value of signal would match this line”

MUX

A	B	S	Z
0	-	0	0
1	-	0	1
-	0	1	0
-	1	1	1

In a similar way to binary adder, we can derive the logical expression for the circuit output from the table:

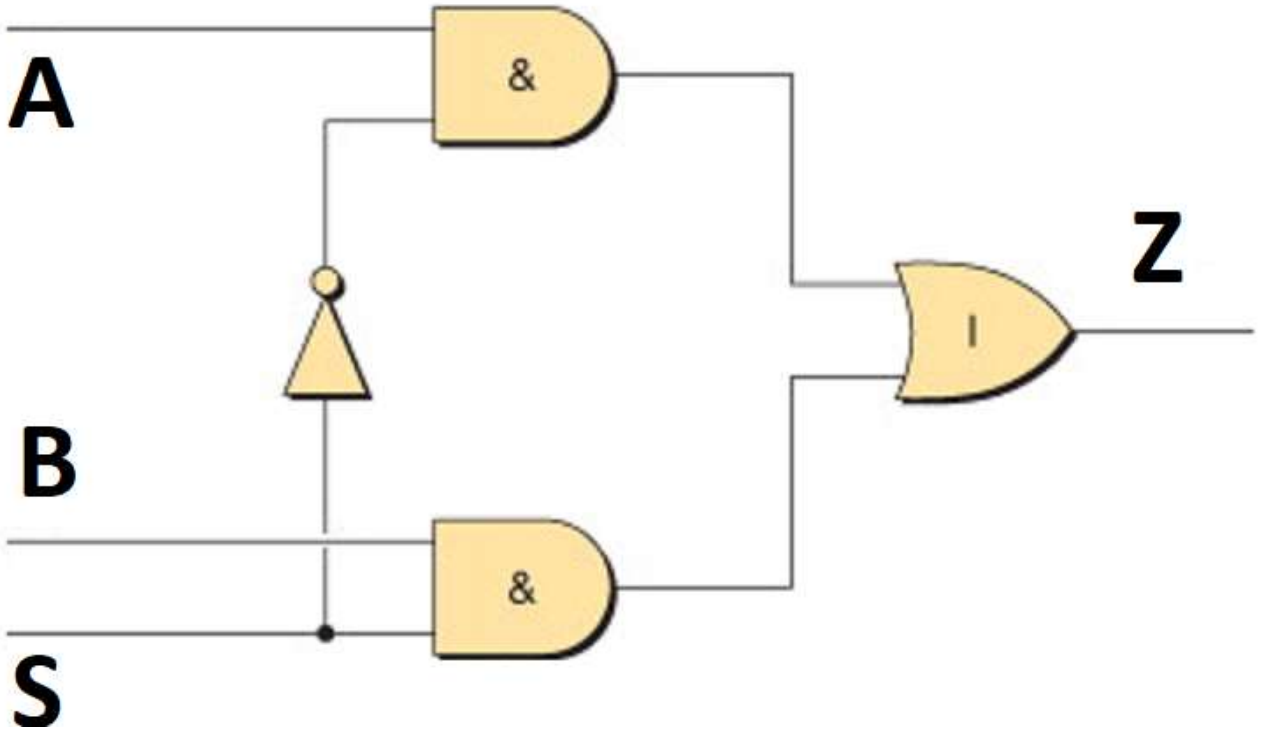
$$Z = A\bar{S} + BS$$

Note that "-" values are ignored, this is an equivalent to writing the full equations as both pairs of "do not care" values were present and then simplifying the equation:

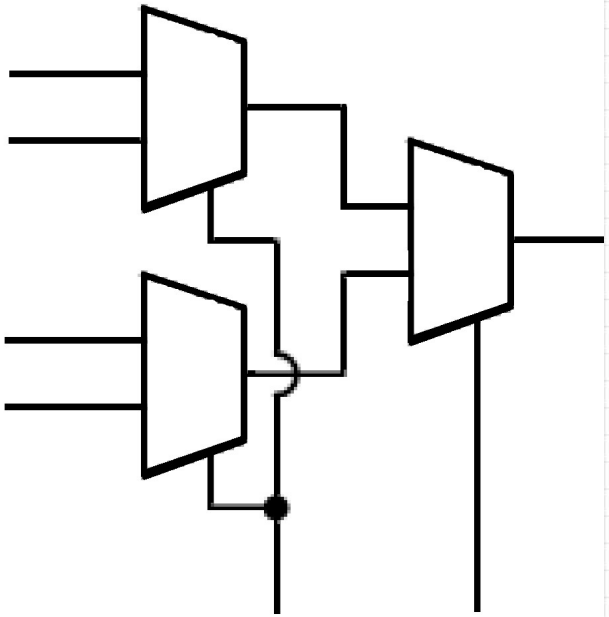
$$Z = AB\bar{S} + A\bar{B}\bar{S} + ABS + \bar{A}BS = A\bar{S}(B + \bar{B}) + (A + \bar{A})BS$$
$$= A\bar{S} + BS$$

MUX

$Z = A\bar{S} + BS$



4-1 MUX



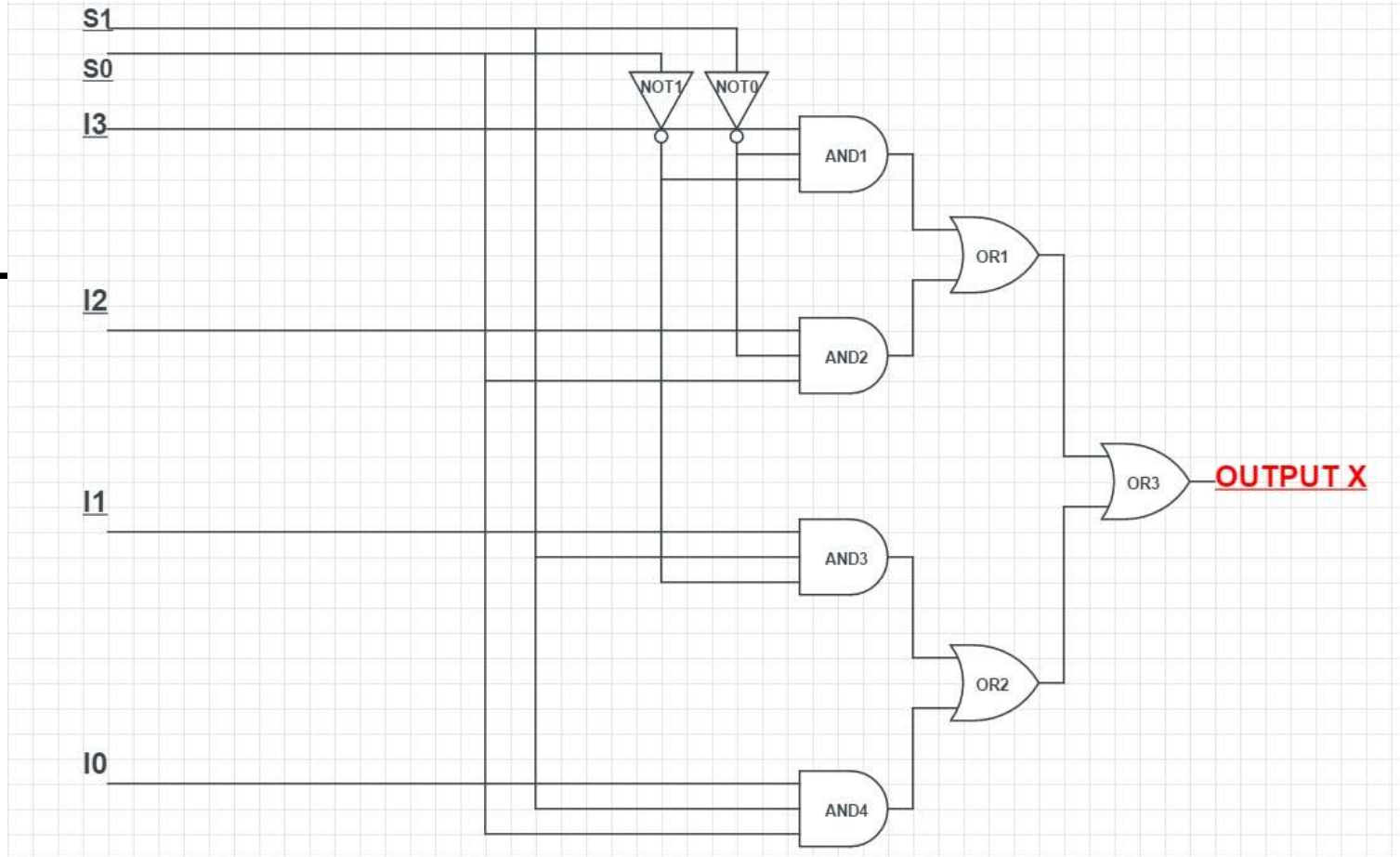


Image credit: Wikipedia

A bit of VHDL

(That’s why we are all here, right?)

VHDL: brief intro

Let's introduce VHDL by giving some example code and explaining what it does, but first…

The most important thing you need to know about VHDL:

VHDL abbreviation stands for:

- Very
- High
- Speed
- Integrated
- Circuit
- Hardware
- Description
- Language

(there is nested abbreviation: **V**HDL == **VHSIC** HDL)

VHDL: brief intro

- Developed by U.S. Department of Defence with the initial purpose of documenting of circuit behaviour
- Later adopter for other uses, e.g. synthesis of circuit from the definition
- Syntax is based on ADA programming language (another beautiful child of DoD, no irony here)
- Mandatory Hello World in ADA:
- Both ADA and VHDL are super-strictly typed **military grade languages**
  (start fearing now!)

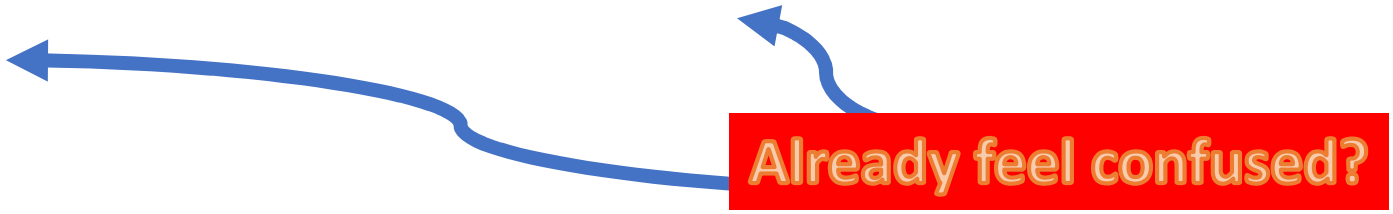
```
with Ada.Text_IO; use Ada.Text_IO;
procedure Hello is
begin
   Put_Line ("Hello, world!");
end Hello;
```

VHDL brief intro

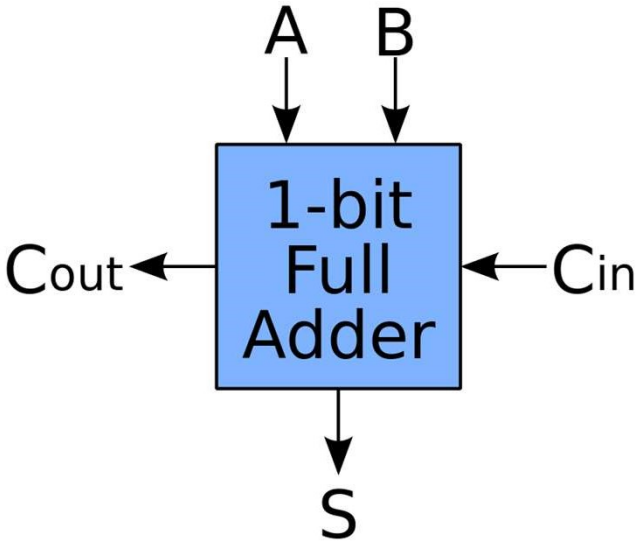
- Far not the only Hardware Definition Language in existence or use, other alternatives:
  - Verilog
  - SystemVerilog
  - C-dialects (!)
  - Others
- Rumours are that VHDL is more popular amongst FPDA developers, Verilog is more popular in ASIC (Application Specific Integrated Circuit) world
- VHDL Language itself is quite big and complex, but we don’t have to learn it all to understand the basics

VHDL terminology by contrast with SW

- Design – FPGA engineers convert caffeine into design
- Synthesis – “compilation”, process of converting VHDL code into a netlist
- Signal – “variable” (not quite exactly, but very-very roughly)
- Variable – “variable”
- Entity – “interface”
- Architecture – “implementation of the interface”
- Component == Entity + it’s architecture



VHDL by example



- Lets get back to our 1-bit adder
- It is a classical example of “entity”
- Entity has a “port” - number of input / output / bidirectional signals it uses to interface with the rest of the world
- Those signals are the only way it interfaces with the outer world
- In our adder example we have signals A, B, C_in, C_out and S

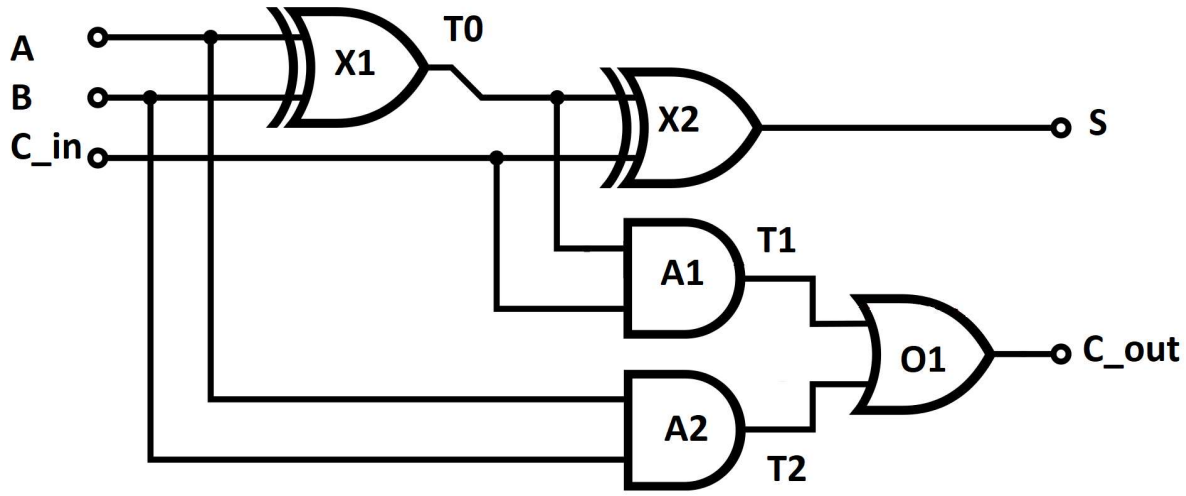
```
entity full_adder is
    port (
        A, B, C_in      :   in  std_logic;
        S, C_out        :   out std_logic );
end full_adder;
```

VHDL - One bit adder – dataflow (or RTL) model implementation

```
library ieee;
use ieee.std_logic_1164.all;

entity full_adder is
    port (
        A, B, C_in      :   in  std_logic;
        S, C_out        :   out std_logic );
end full_adder;

architecture dataflow of full_adder is
begin
    S <= A xor B xor C_in;
    C_out <= ((A xor B) and C_in) or (A and B);
end dataflow;
```

VHDL - One bit adder – structural model implementation

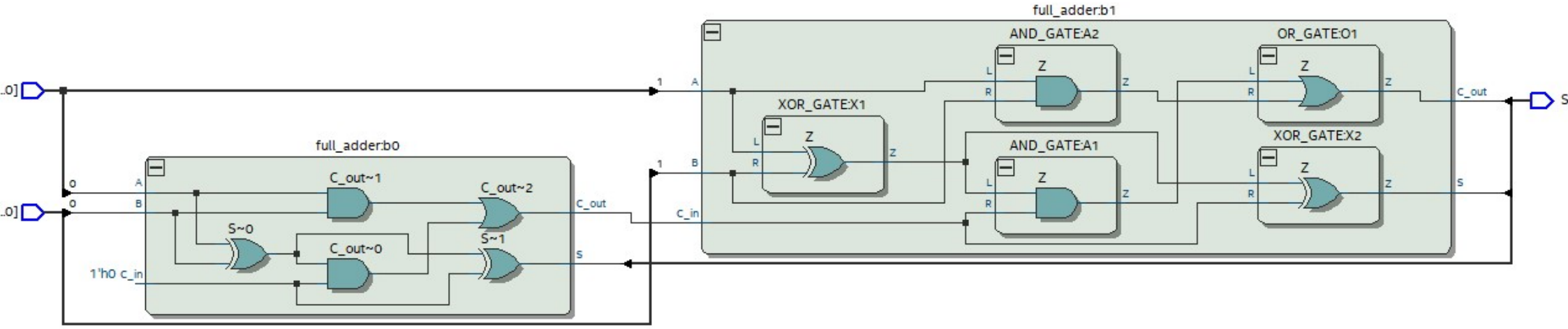
```
architecture structural of full_adder is
    component AND_GATE
        port(L, R: in std_logic;
             Z : out std_logic);
    end component;
    component OR_GATE
        port(L, R: in std_logic;
             Z : out std_logic);
    end component;
    component XOR_GATE
        port(L, R: in std_logic;
             Z : out std_logic);
    end component;

    signal T0, T1, T2: std_logic;
begin
    X1: XOR_GATE port map(L => A, R => B, Z => T0);
    X2: XOR_GATE port map(L => T0, R => C_in, Z => S);
    A1: AND_GATE port map(T0, C_in, T1);
    A2: AND_GATE port map(A, B, T2);
    O1: OR_GATE port map(T1, T2, C_out);
end structural;
```



VHDL – both adders together

```
entity Adders is
    port (
        A : in std_logic_vector(1 downto 0);
        B : in std_logic_vector(1 downto 0);
        S: out std_logic_vector(2 downto 0)
    );
end Adders;

architecture dataflow of Adders is
    signal carries: std_logic_vector(1 downto 0);
begin
    -- first bit is using dataflow implementation
    b0: entity work.full_adder(dataflow)
        port map(A => A(0), B => B(0), C_in => '0',
        S => S(0), C_out => carries(0));

    -- second bit is using structural
    b1: entity work.full_adder(structural)
        port map(A => A(1), B => B(1), C_in => carries(0),
        S => S(1), C_out => S(2));

end dataflow;
```

VHDL – both adders together - synthesis

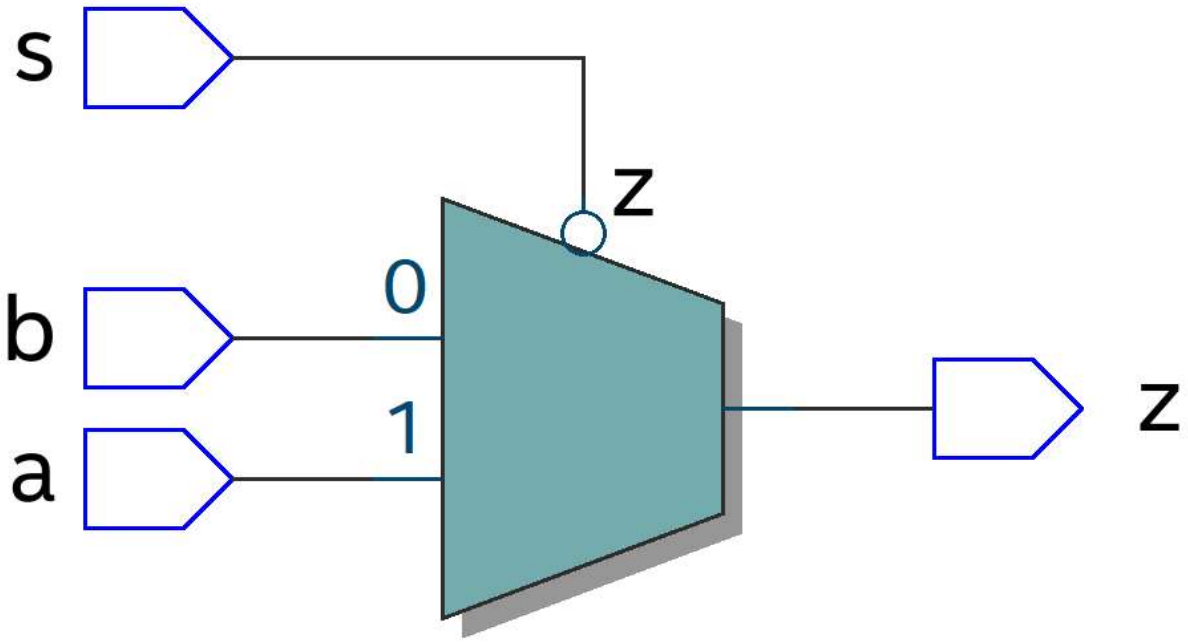


Side note: how you should actually do addition in VHDL

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity RealWorldAdder is
    port (
        A       : in std_logic_vector(15 downto 0);
        B       : in std_logic_vector(15 downto 0);
        S       : out std_logic_vector(15 downto 0)
        );
end RealWorldAdder;

architecture dataflow of RealWorldAdder is
begin
    S <= A + B;
end;
```

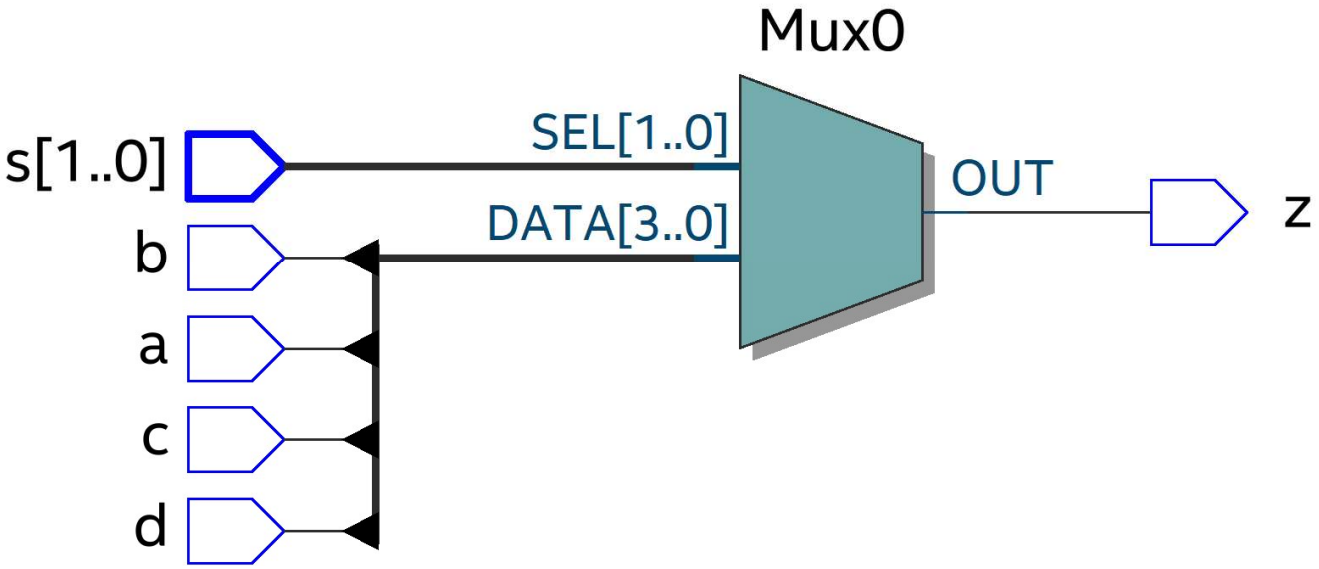
VHDL – MUX2

```
library IEEE;
use IEEE.std_logic_1164.all;

entity mux2 is
port(
   a        : in  std_logic;
   b        : in  std_logic;
   s        : in  std_logic;
   z        : out std_logic);
end mux2;

architecture rtl of mux2 is
begin
   with s select
      z <= a when '0',
           b when others;
end rtl;
```



VHDL – MUX4

```
library IEEE;
use IEEE.std_logic_1164.all;

entity mux4 is
port(
   a         : in  std_logic;
   b         : in  std_logic;
   c         : in  std_logic;
   d         : in  std_logic;
   s         : in  std_logic_vector(1 downto 0);
   z         : out std_logic);
end mux4;

architecture rtl of mux4 is
begin
   with s select
      z <= a when "00",
           b when "01",
           c when "10",
           d when others;
end rtl;
```

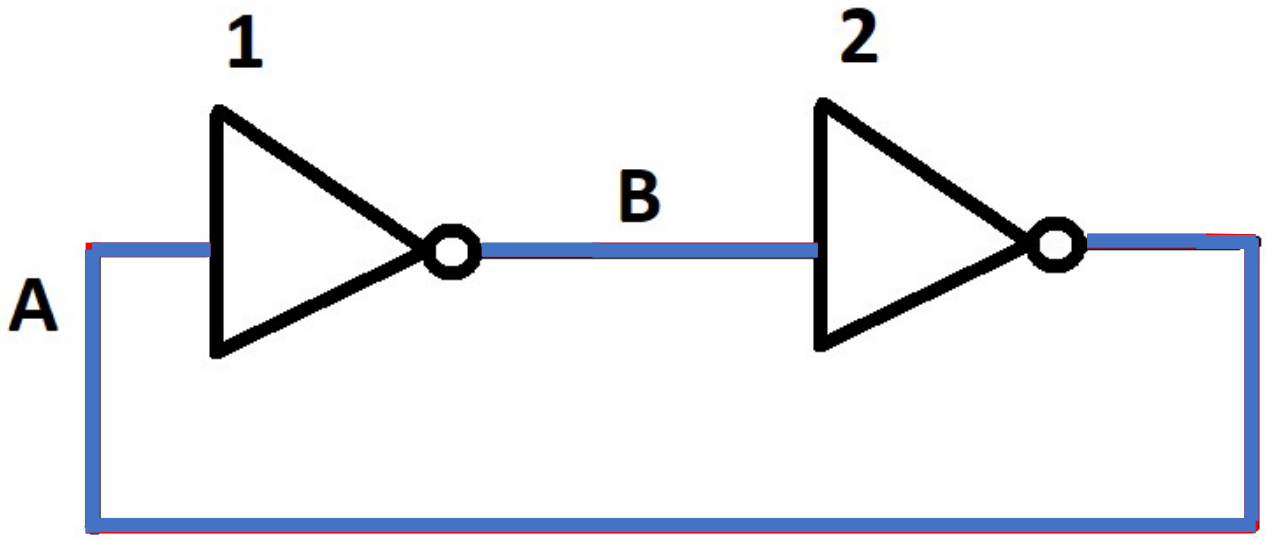


Bistables

When the state is important..

Bistables – Triggers and Flip-Flops

- Consider the circuit:



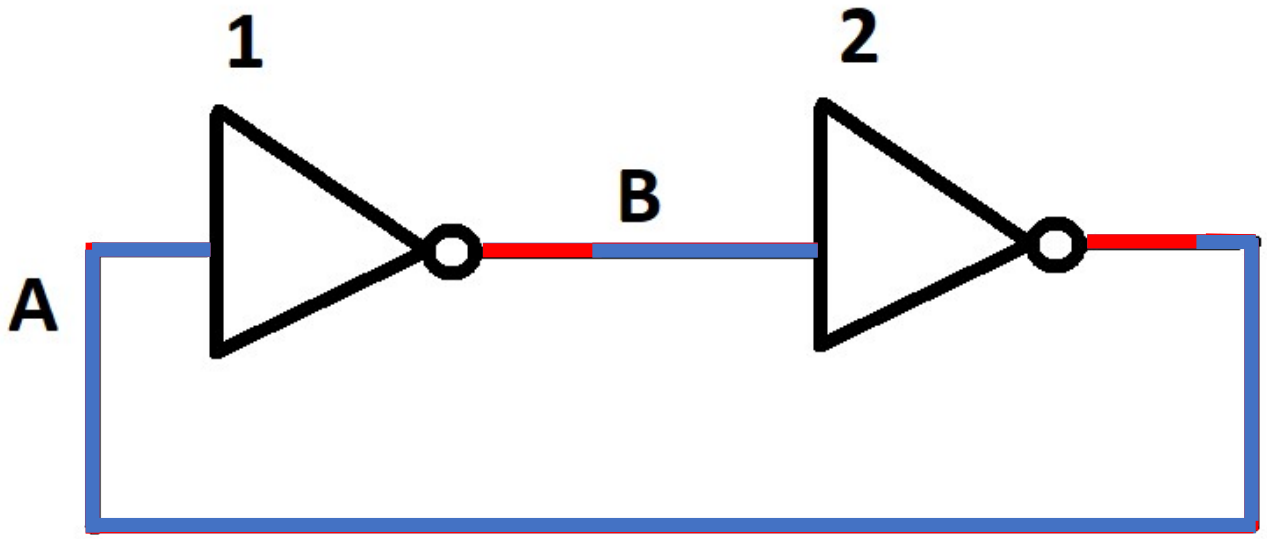
- What would be the values of signals A and B?

0

1

Bistables – Triggers and Flip-Flops

- Consider the circuit:



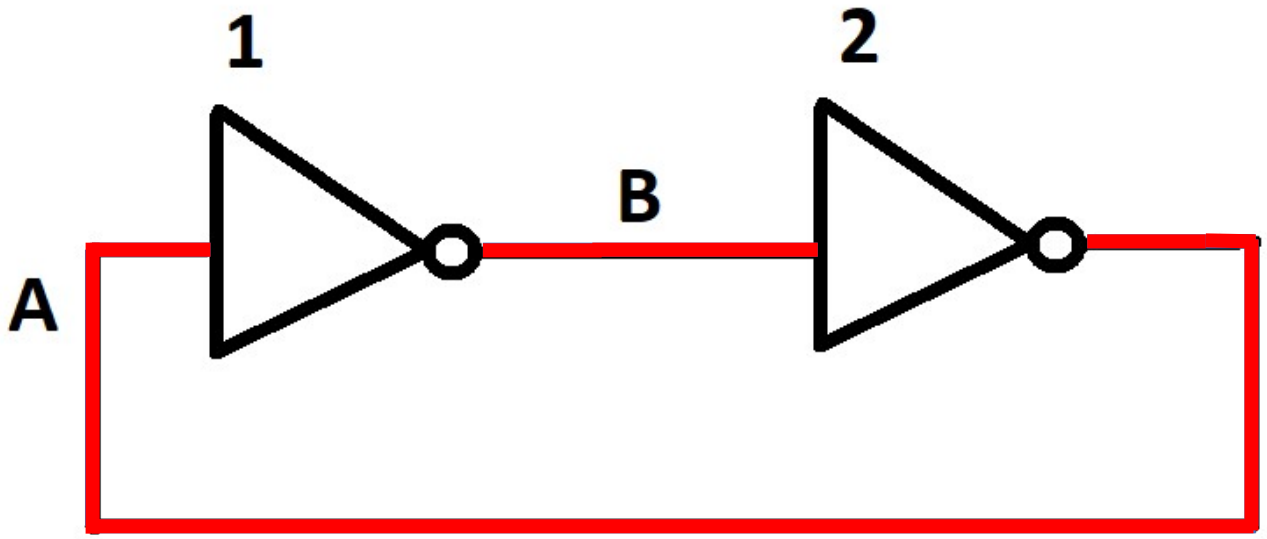
- What would be the values of signals A and B?

0

1

Bistables – Triggers and Flip-Flops

- Consider the circuit:



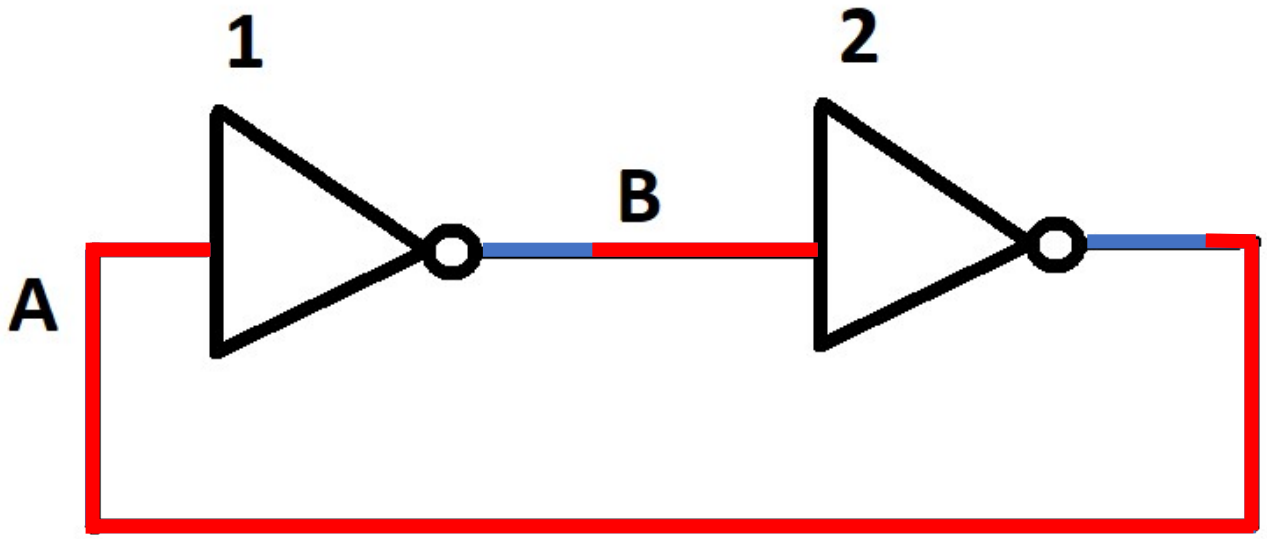
- What would be the values of signals A and B?

0

1

Bistables – Triggers and Flip-Flops

- Consider the circuit:



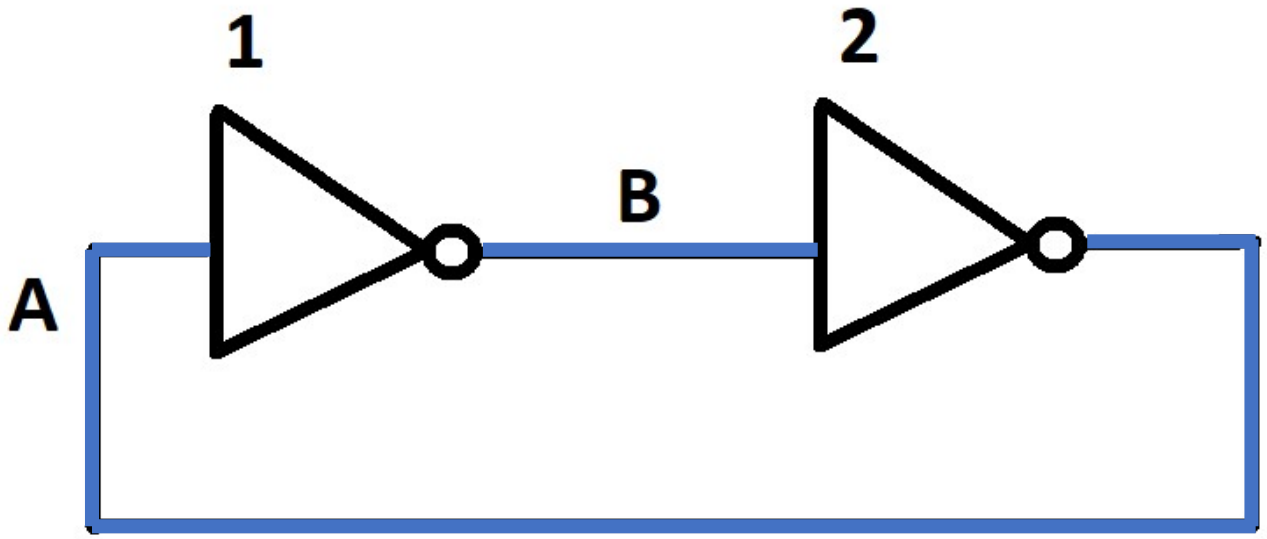
- What would be the values of signals A and B?

0

1

Bistables – Triggers and Flip-Flops

- Consider the circuit:



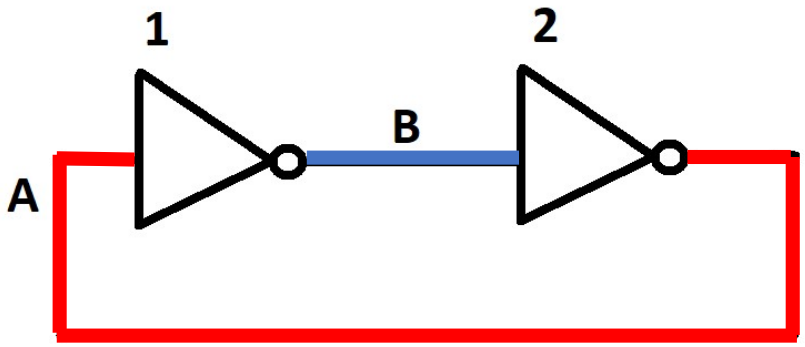
- What would be the values of signals A and B?

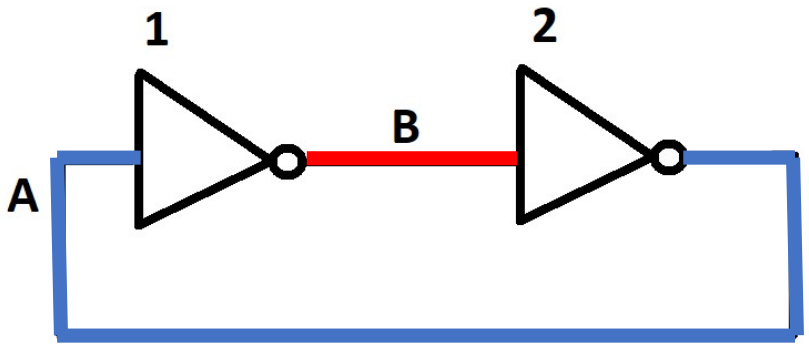
0

1

Bistables – Triggers and Flip-Flops

- Real world is not symmetrical, thus circuit would eventually resolve into one of the stable states:





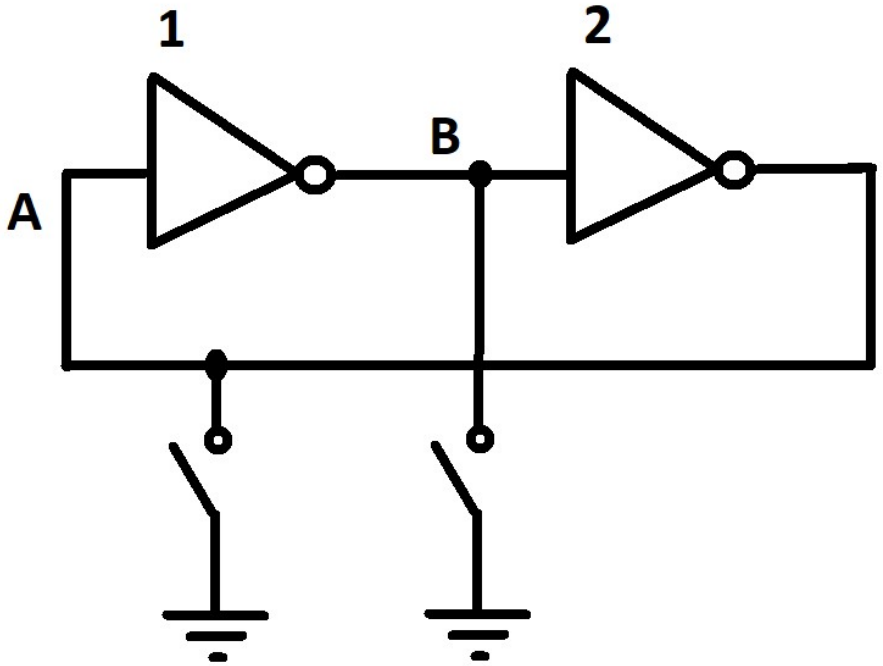
- These states are stable and self-supporting

0

1

Bistables – Triggers and Flip-Flops

- Let’s now add two switches connecting to zero signals A and B when pressed
- This way we can now control the state by forcing one or another side into zero level
- It is a bit brutal (as an opposite to byte brutal)
- It is how SRAM (static RAM) works



Simulation videos:

https://www.youtube.com/watch?v=ytM9lqsXwxk

https://www.youtube.com/watch?v=Si_8qexyEfU

Latches

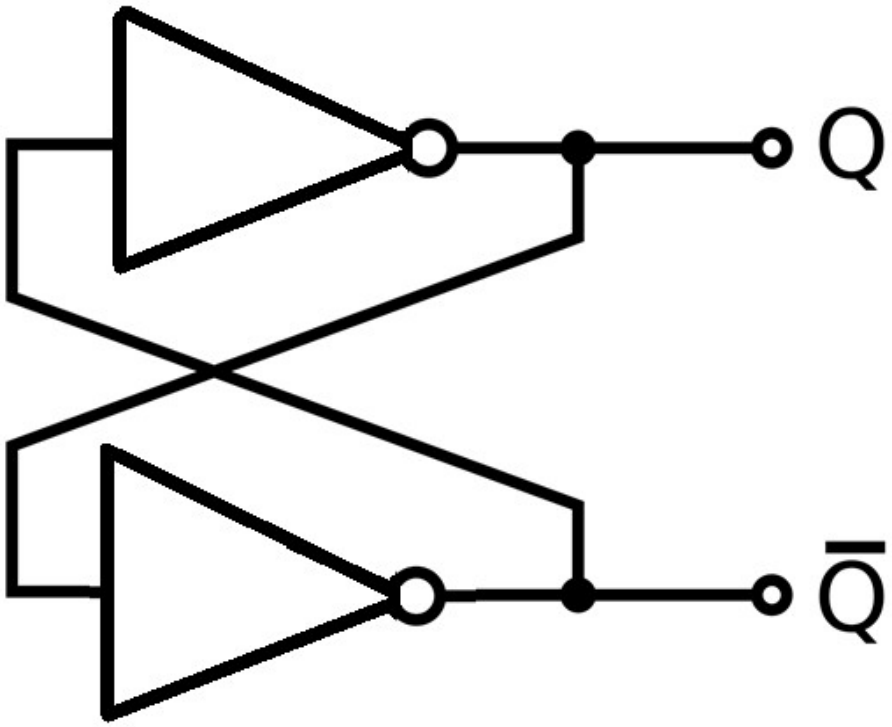


Image credits: https://en.wikibooks.org/wiki/Electronics/Flip_Flops

Latches

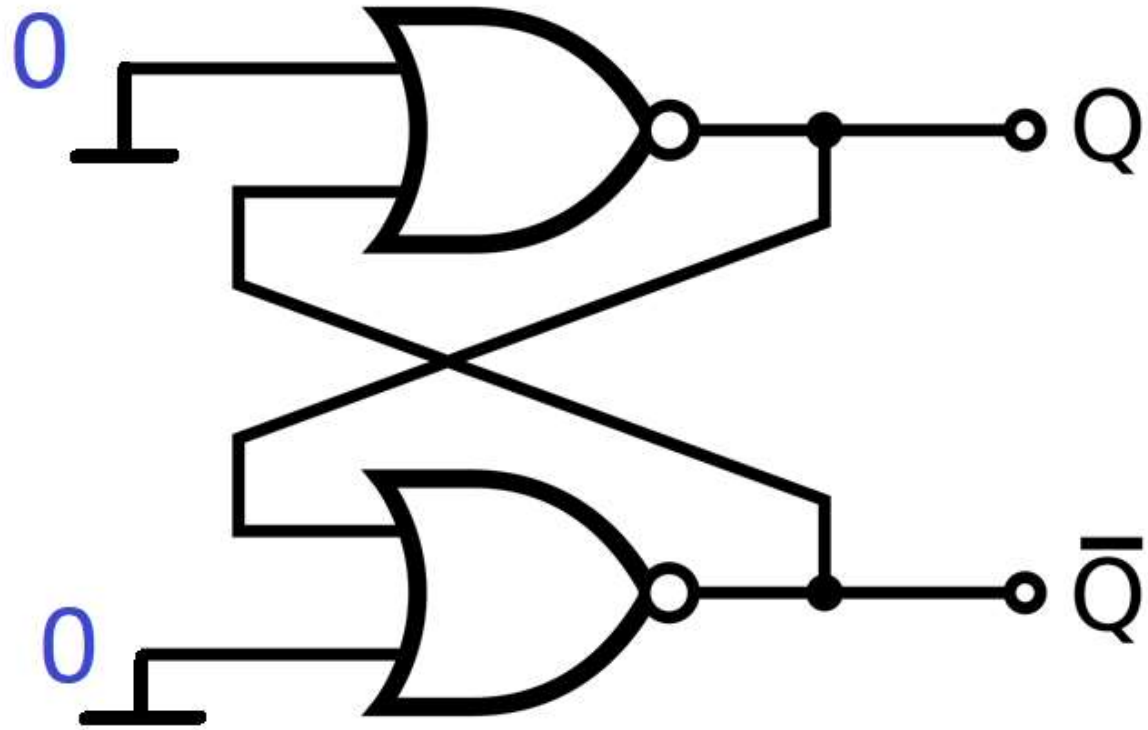


Image credits: https://en.wikibooks.org/wiki/Electronics/Flip_Flops

SR latch

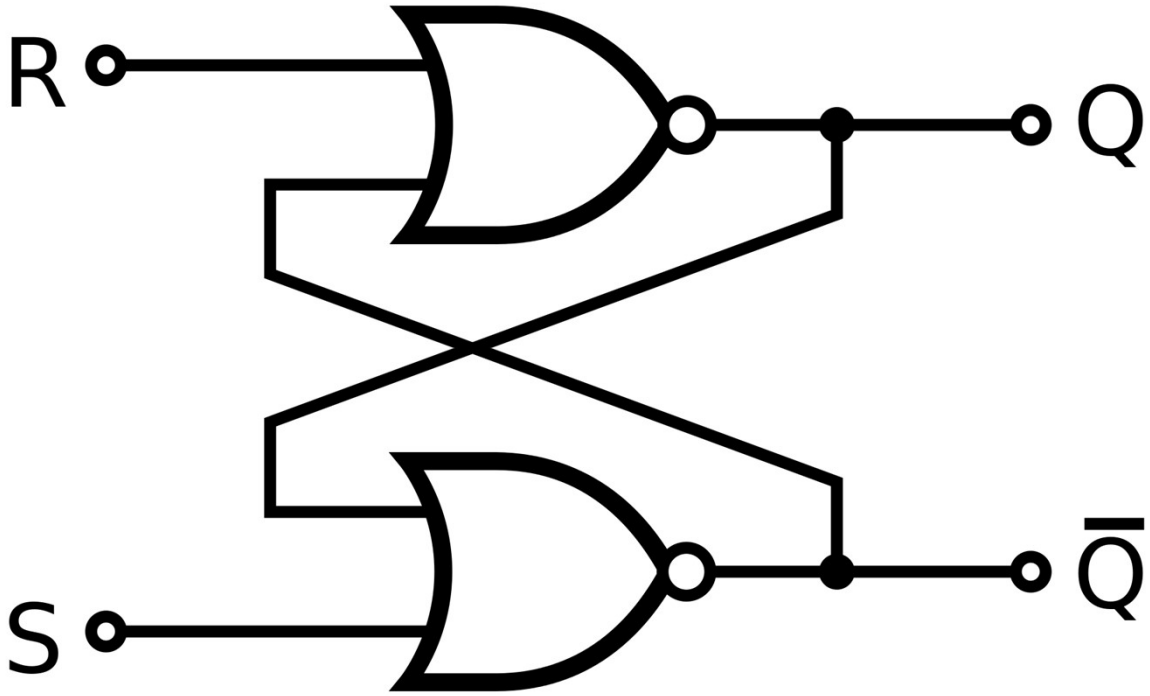
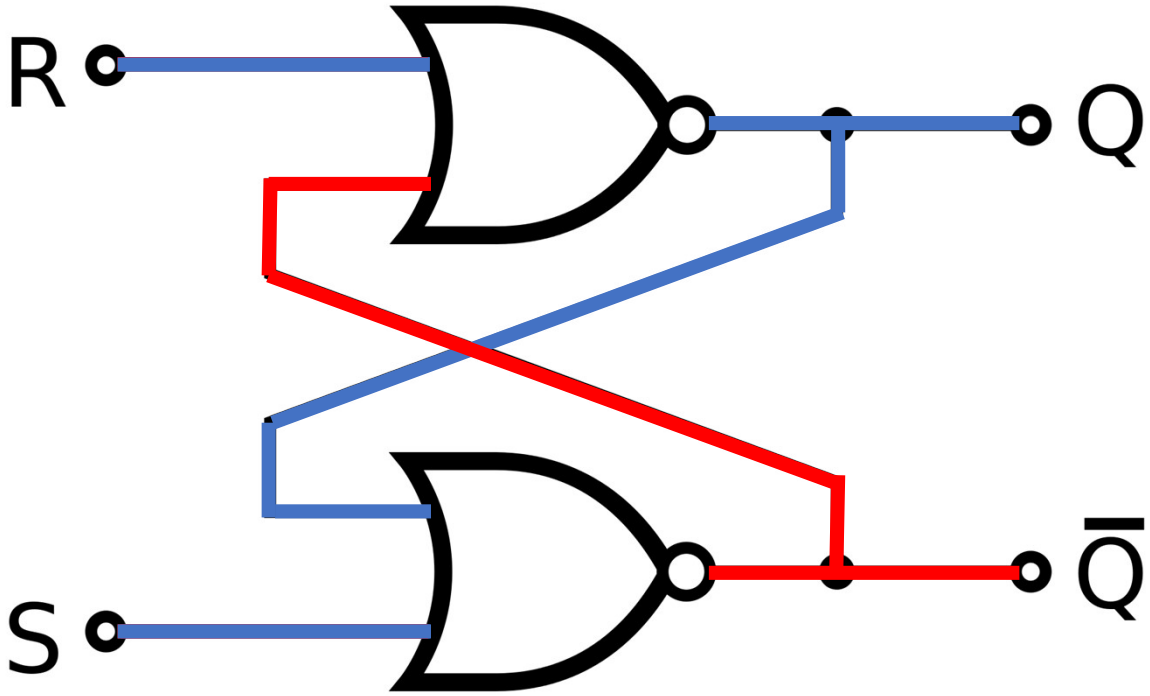
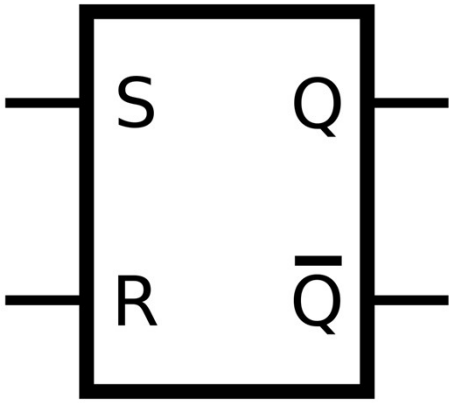


Image credits: https://en.wikibooks.org/wiki/Electronics/Flip_Flops

SR latch - operation





The circuit is stable

0

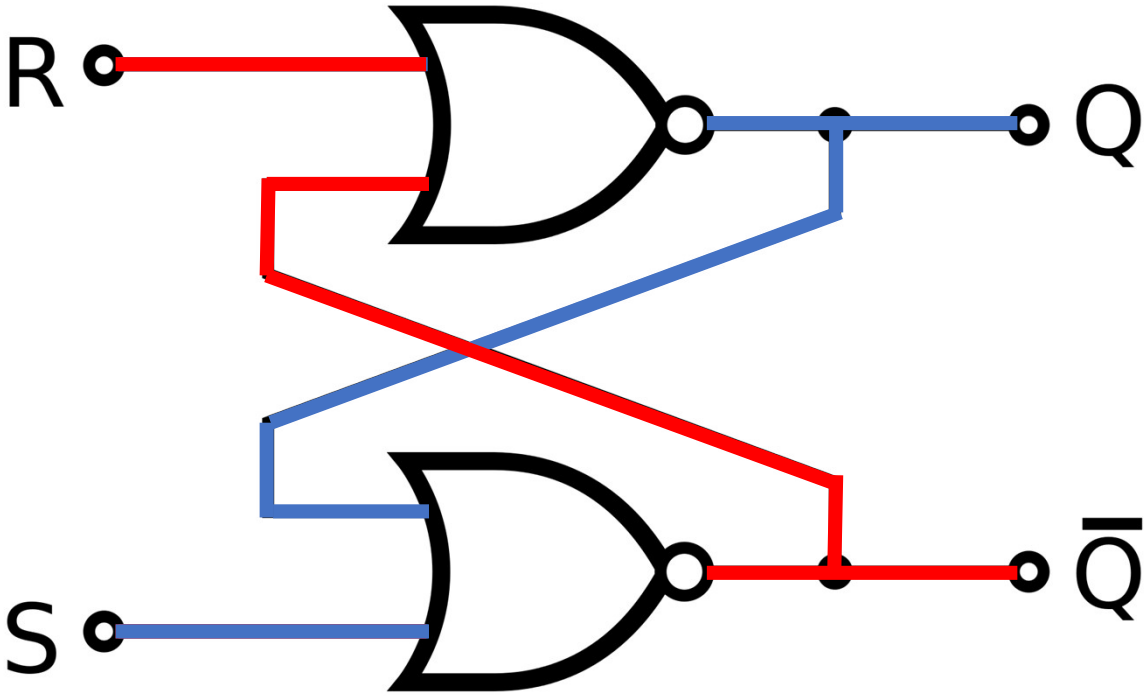
1

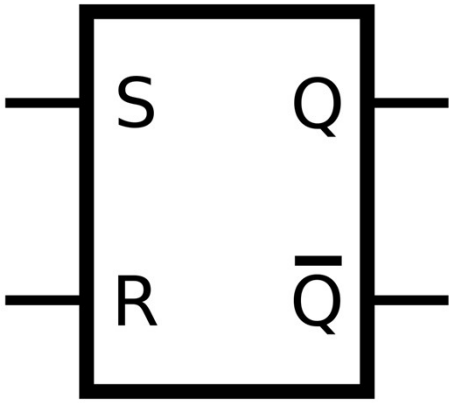
Simulation video: https://www.youtube.com/watch?v=mKkE1hGsxHE

Image credits: https://en.wikibooks.org/wiki/Electronics/Flip_Flops

SR latch - operation

- R = 1 changes nothing





The circuit is stable

0

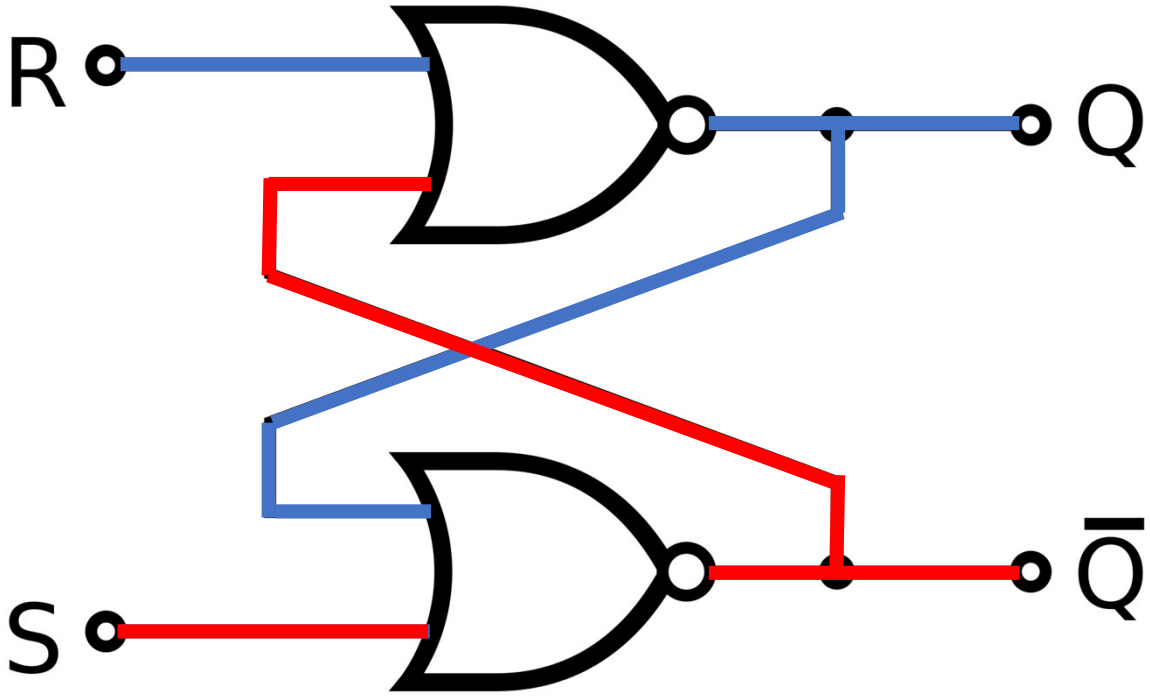
1

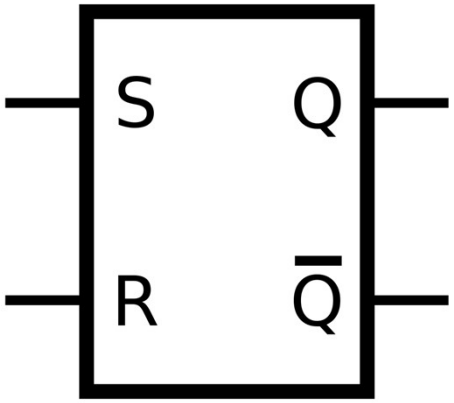
Simulation video: https://www.youtube.com/watch?v=mKkE1hGsxHE

Image credits: https://en.wikibooks.org/wiki/Electronics/Flip_Flops

SR latch - operation

- R = 1 changes nothing
- Now R is back to 0, and S =1





0

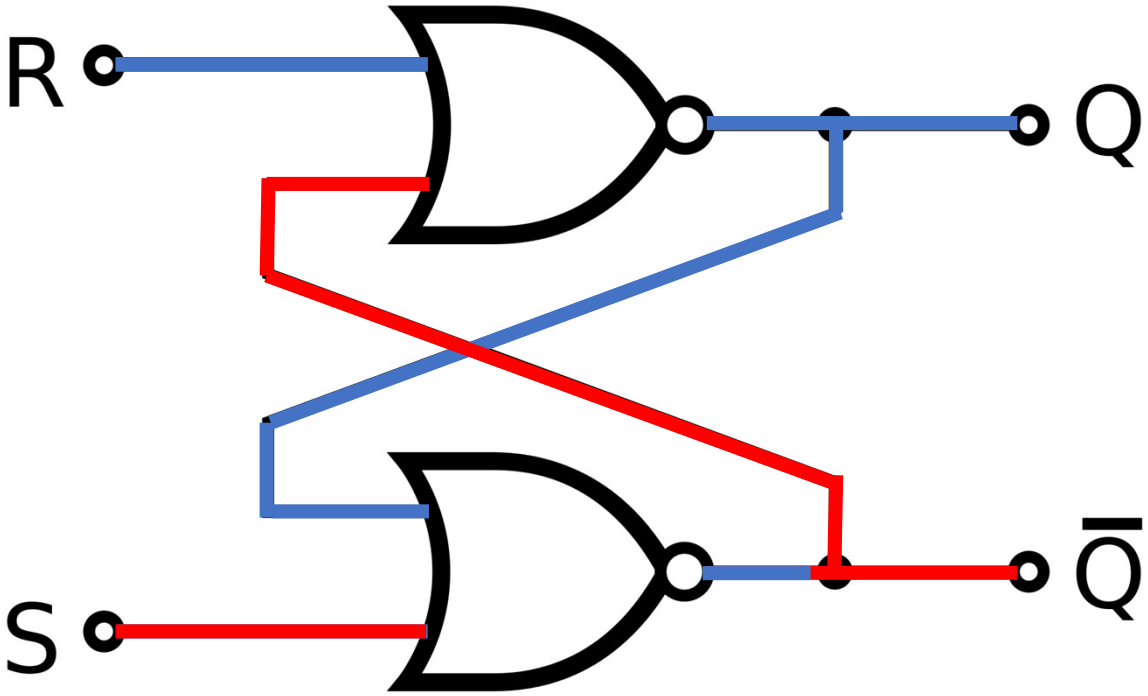
1

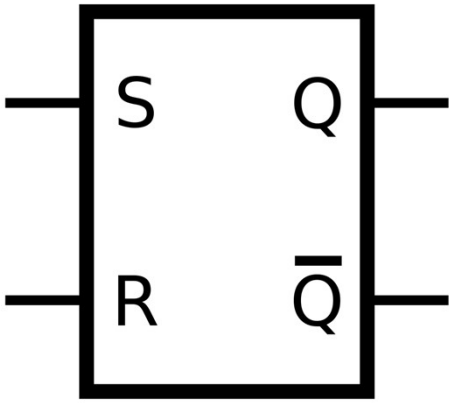
Simulation video: https://www.youtube.com/watch?v=mKkE1hGsxHE

Image credits: https://en.wikibooks.org/wiki/Electronics/Flip_Flops

SR latch - operation

- R = 1 changes nothing
- Now R is back to 0, and S =1
- Bottom gate out flips to zero





0

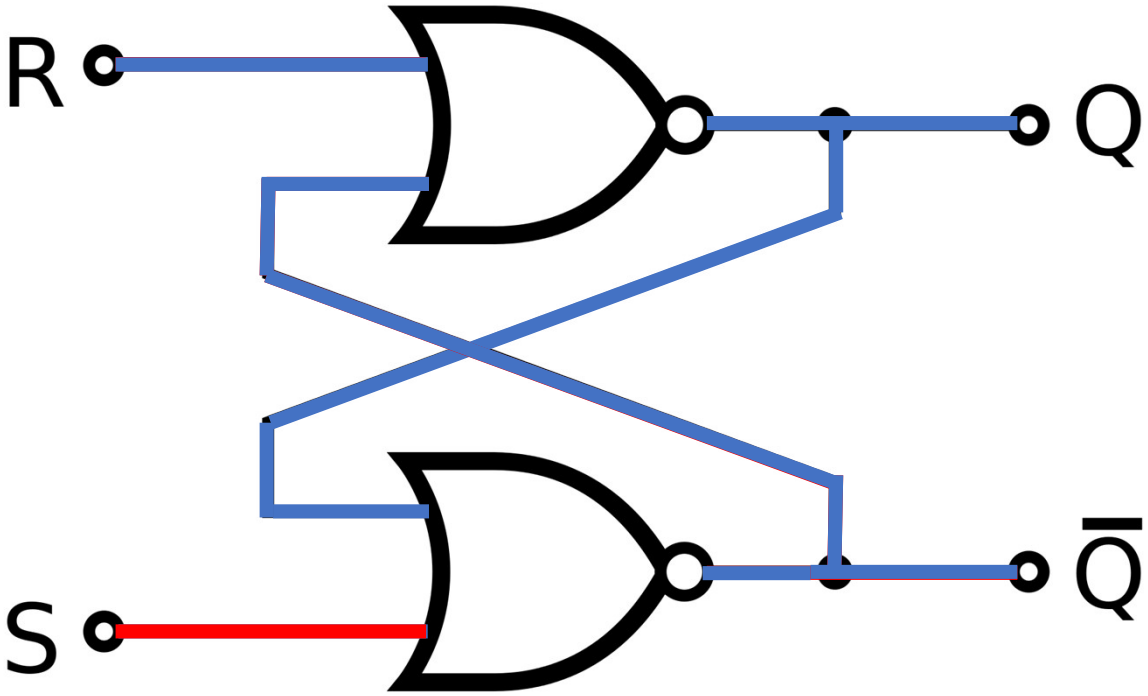
1

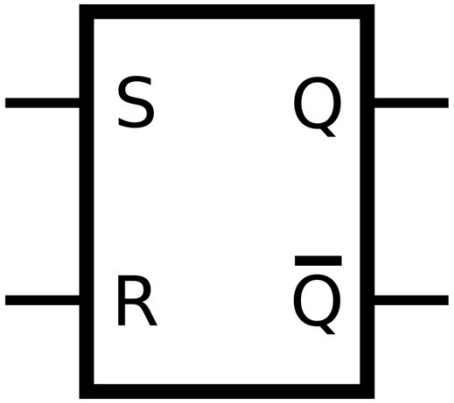
Simulation video: https://www.youtube.com/watch?v=mKkE1hGsxHE

Image credits: https://en.wikibooks.org/wiki/Electronics/Flip_Flops

SR latch - operation

- R = 1 changes nothing
- Now R is back to 0, and S =1
- Bottom gate out flips to zero
- Zero propagates…





0

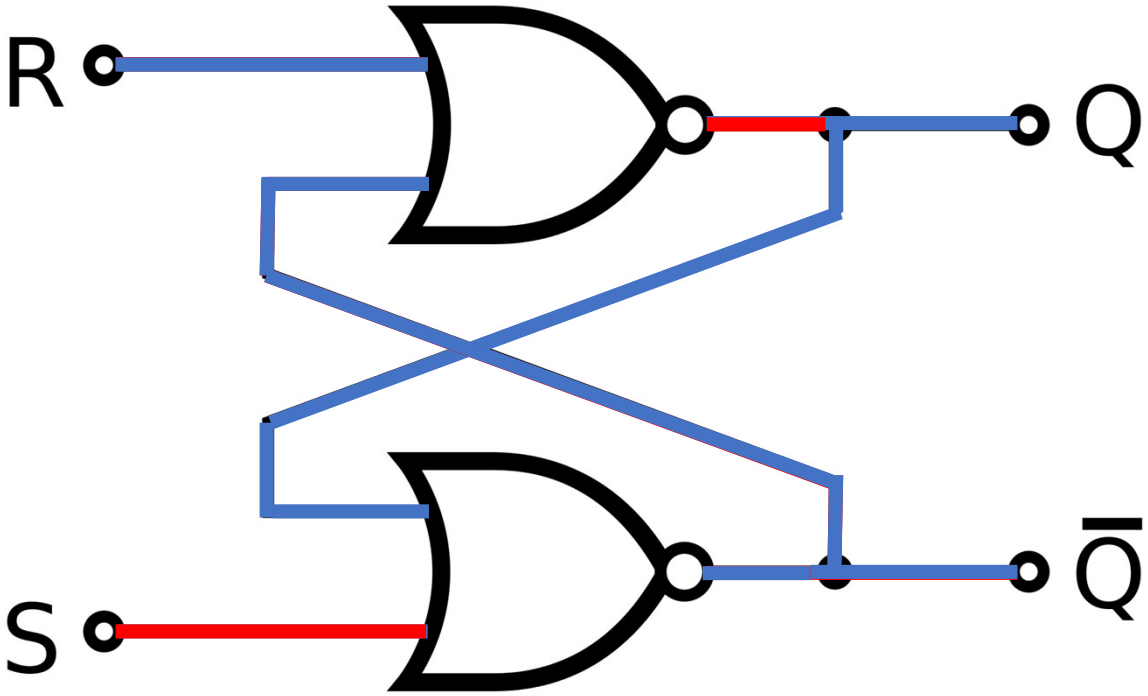
1

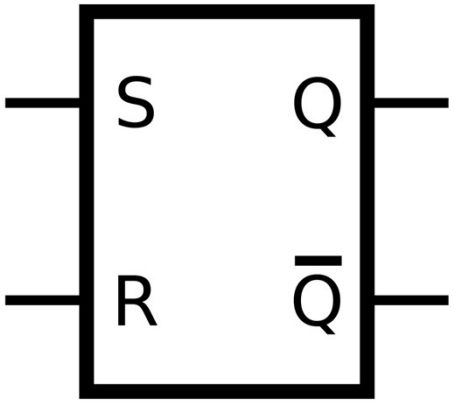
Simulation video: https://www.youtube.com/watch?v=mKkE1hGsxHE

Image credits: https://en.wikibooks.org/wiki/Electronics/Flip_Flops

SR latch - operation

- R = 1 changes nothing
- Now R is back to 0, and S =1
- Bottom gate out flips to zero
- Zero propagates…
- Upper gates flips to one





0

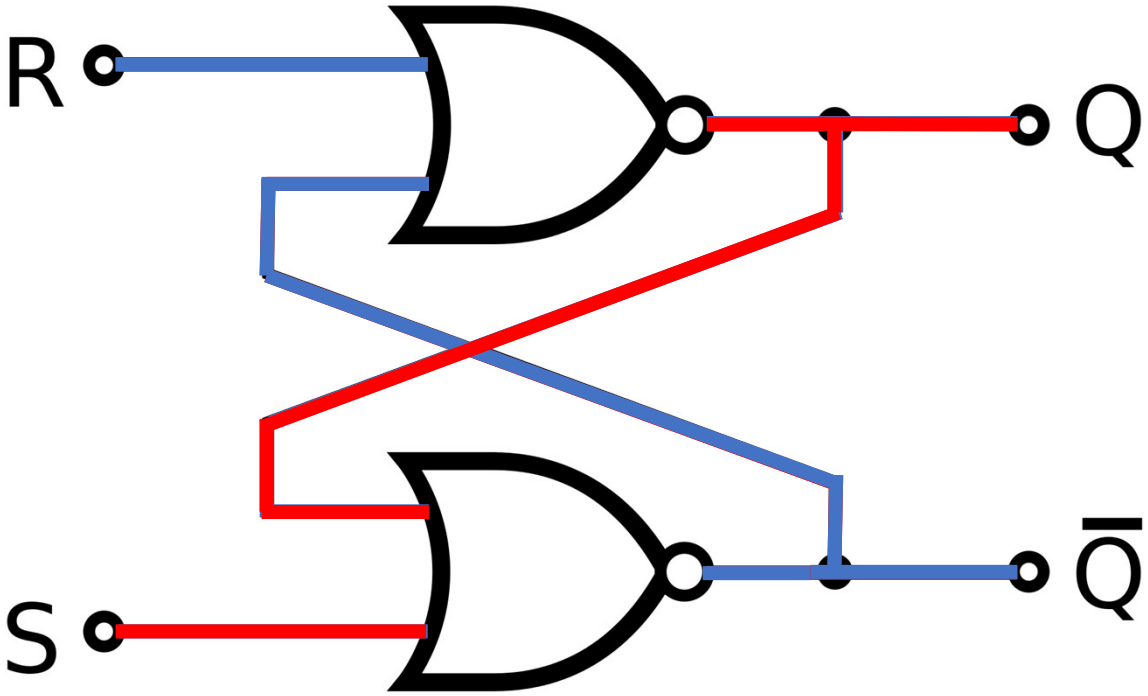
1

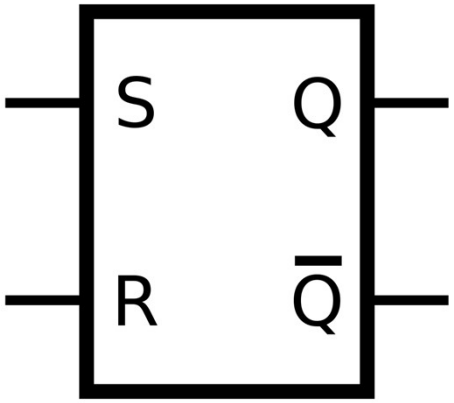
Simulation video: https://www.youtube.com/watch?v=mKkE1hGsxHE

Image credits: https://en.wikibooks.org/wiki/Electronics/Flip_Flops

SR latch - operation

- R = 1 changes nothing
- Now R is back to 0, and S =1
- Bottom gate out flips to zero
- Zero propagates…
- Upper gates flips to one
- And it propagates too





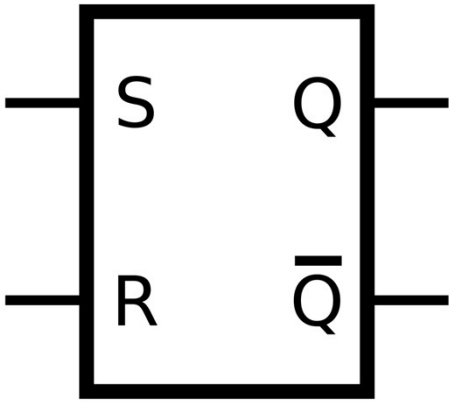
0

1

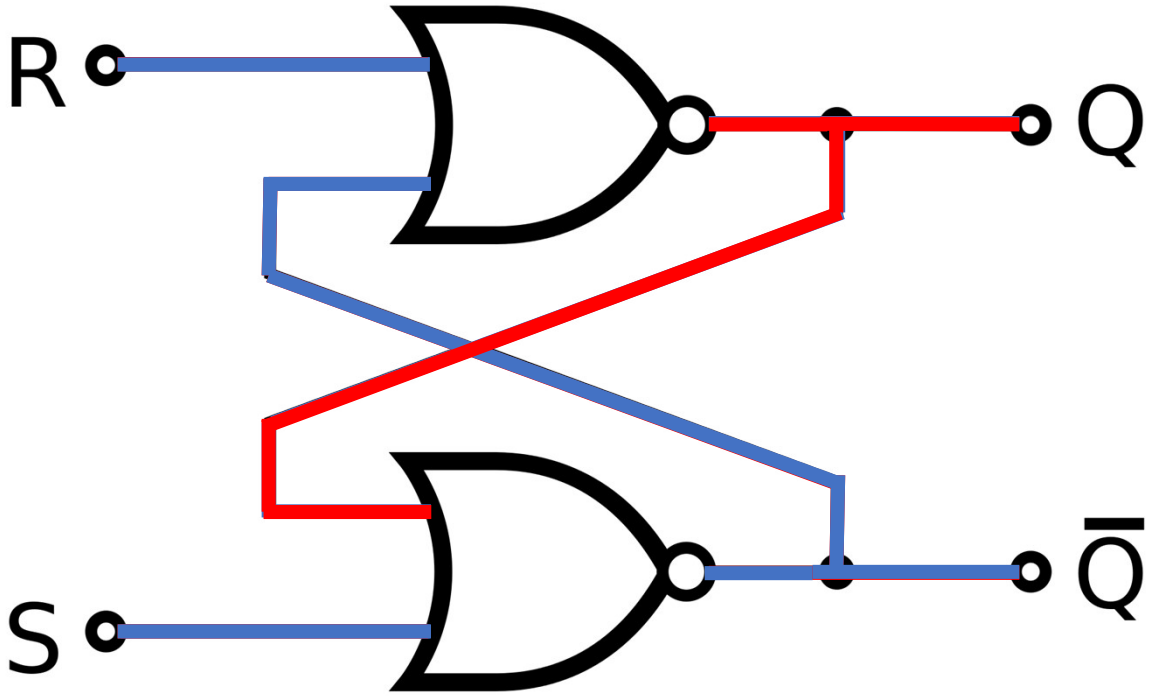
Simulation video: https://www.youtube.com/watch?v=mKkE1hGsxHE

Image credits: https://en.wikibooks.org/wiki/Electronics/Flip_Flops

SR latch - operation



- R = 1 changes nothing
- Now R is back to 0, and S =1
- Bottom gate out flips to zero
- Zero propagates…
- Upper gates flips to one
- And it propagates too
- Now if we remove S=1, circuit stays in it state



The circuit is now stable

0

1

Simulation video: https://www.youtube.com/watch?v=mKkE1hGsxHE

Image credits: https://en.wikibooks.org/wiki/Electronics/Flip_Flops

Gated SR latch

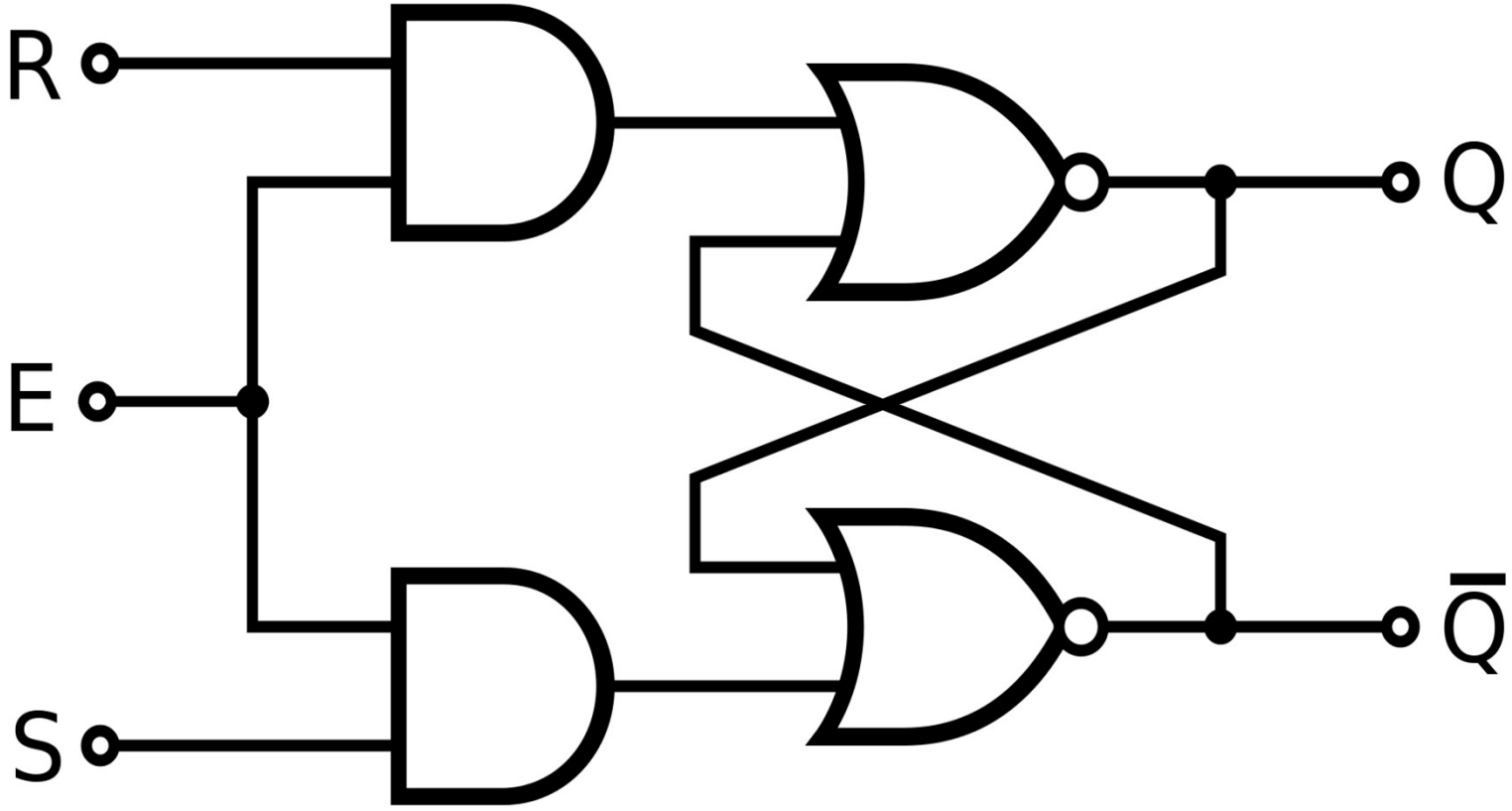


Image credits: https://en.wikibooks.org/wiki/Electronics/Flip_Flops

D latch

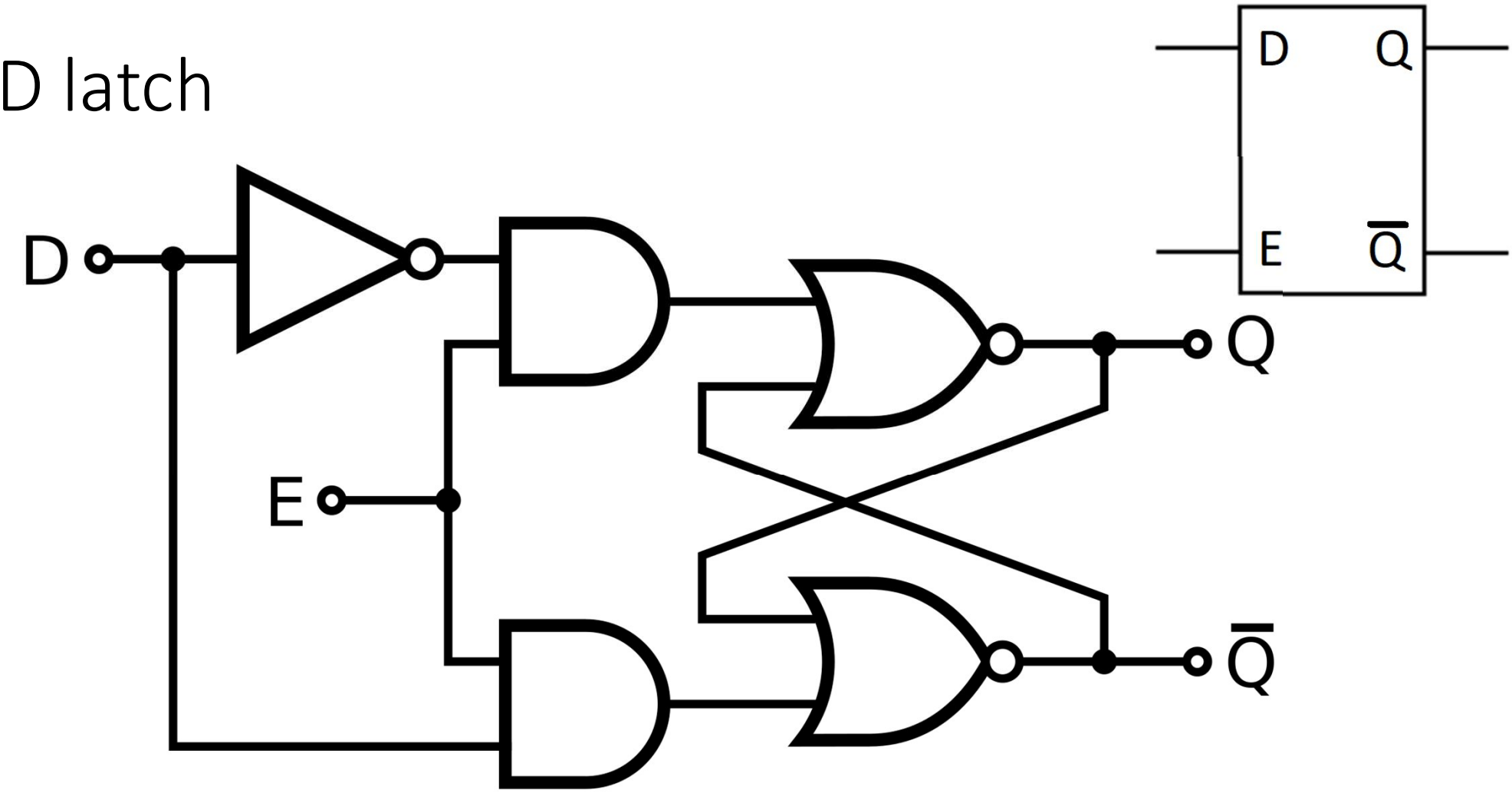
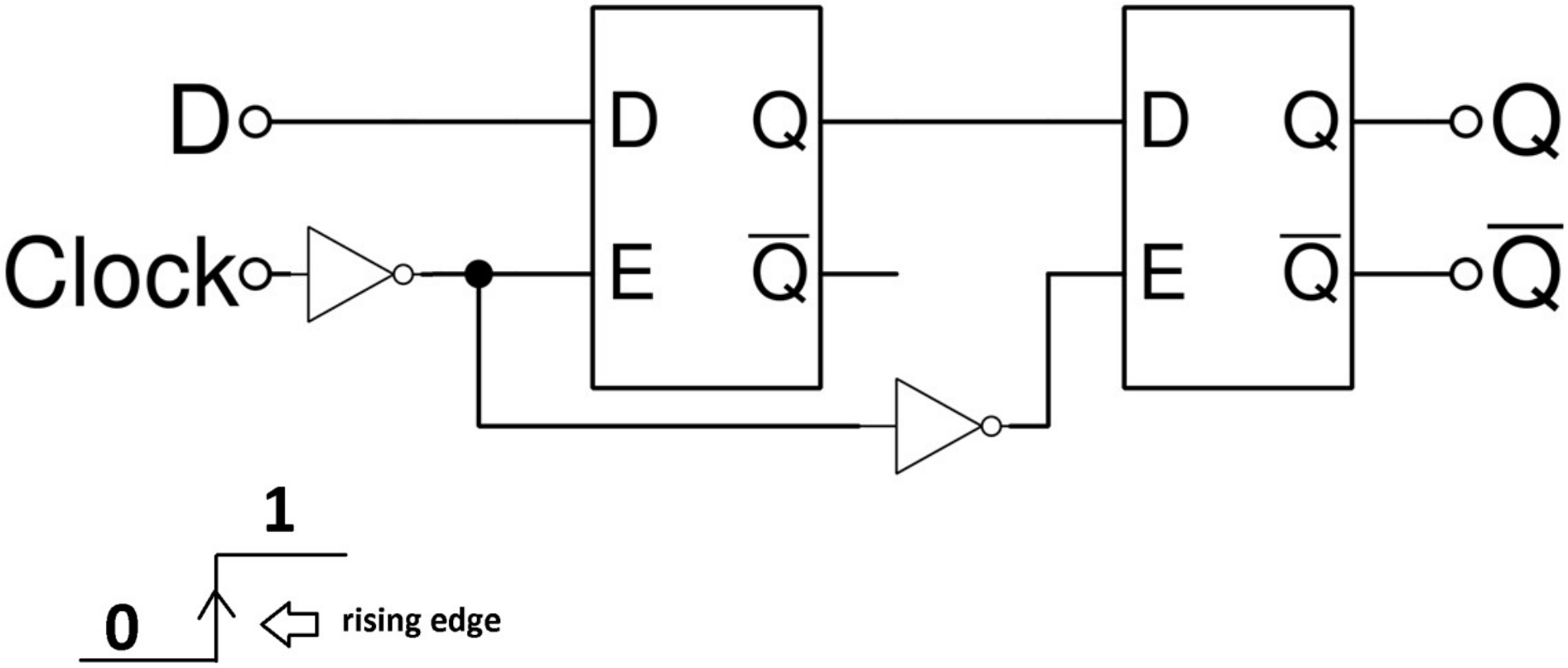


Image credits: https://en.wikibooks.org/wiki/Electronics/Flip_Flops

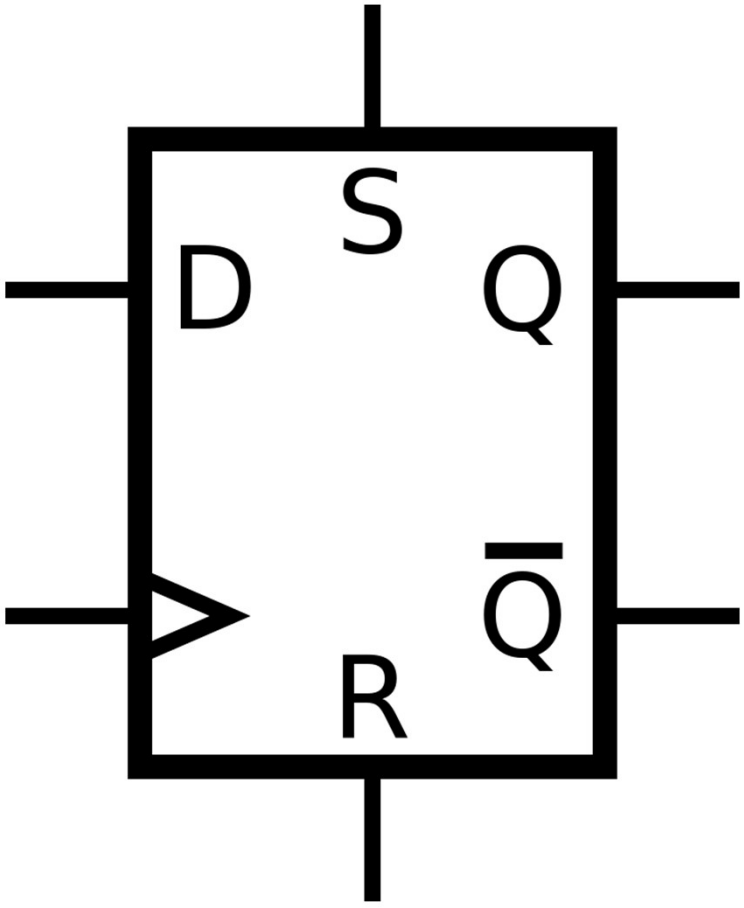
Simulation video: https://www.youtube.com/watch?v=PnkENqnPqd8

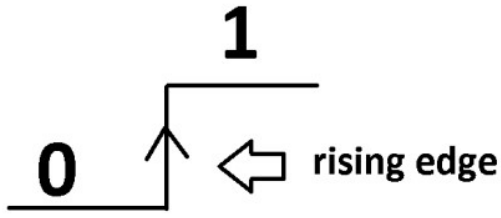
Flip-flops: D flip flop



Simulation video: https://www.youtube.com/watch?v=fain2HVHdp4

D flip-flop: 1 bit register



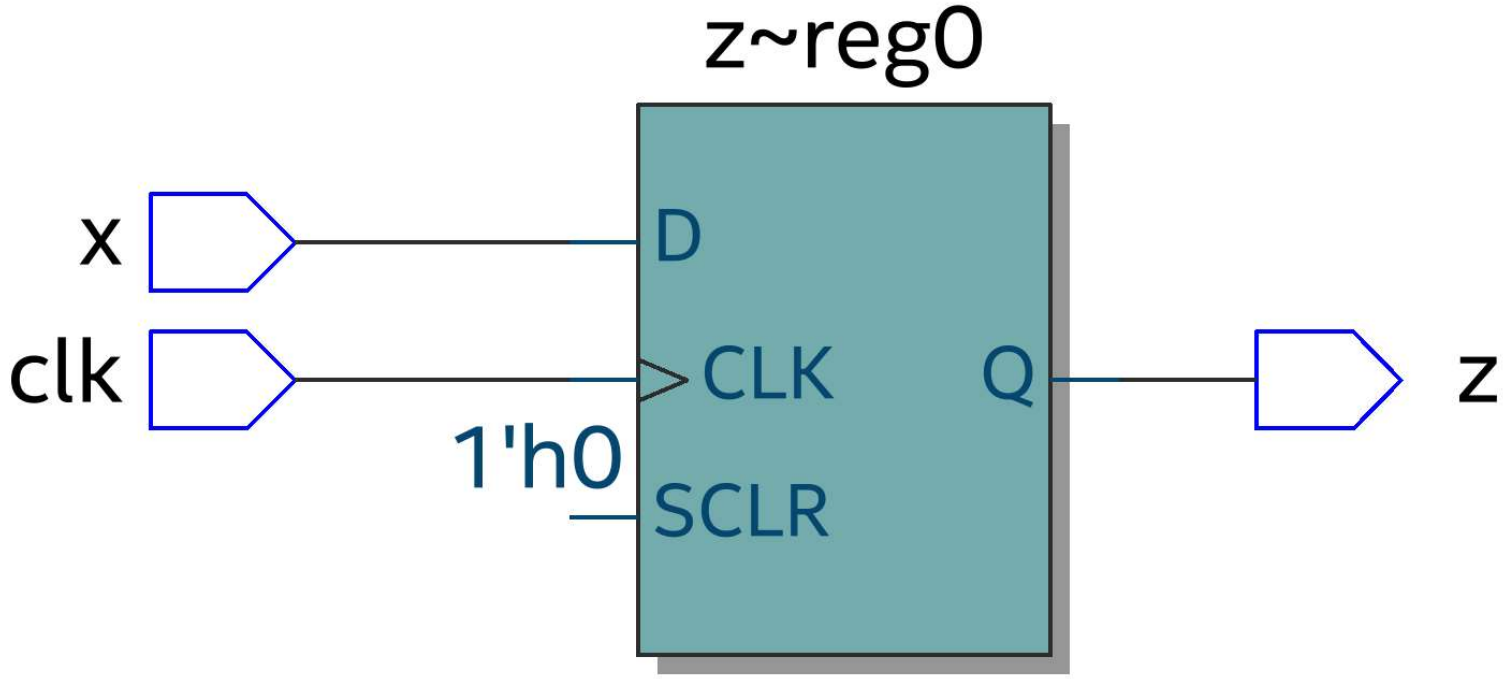


D flip flop in VHDL?

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity d_ff is
    port (
        x, clk  :   in  std_logic;
        z       :   out std_logic
);
end d_ff;

architecture behavior of d_ff is
begin
    process (x, clk)
    begin
        if clk'event and clk = '1' then
            z <= x;
        end if;
    end process;
end behavior;
```

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity d_ff is
    port (
        x, clk  :   in  std_logic;
        z       :   out std_logic
);
end d_ff;

architecture rtl of d_ff is
begin
    z <= x when clk = '1' and clk'event;
end rtl;
```

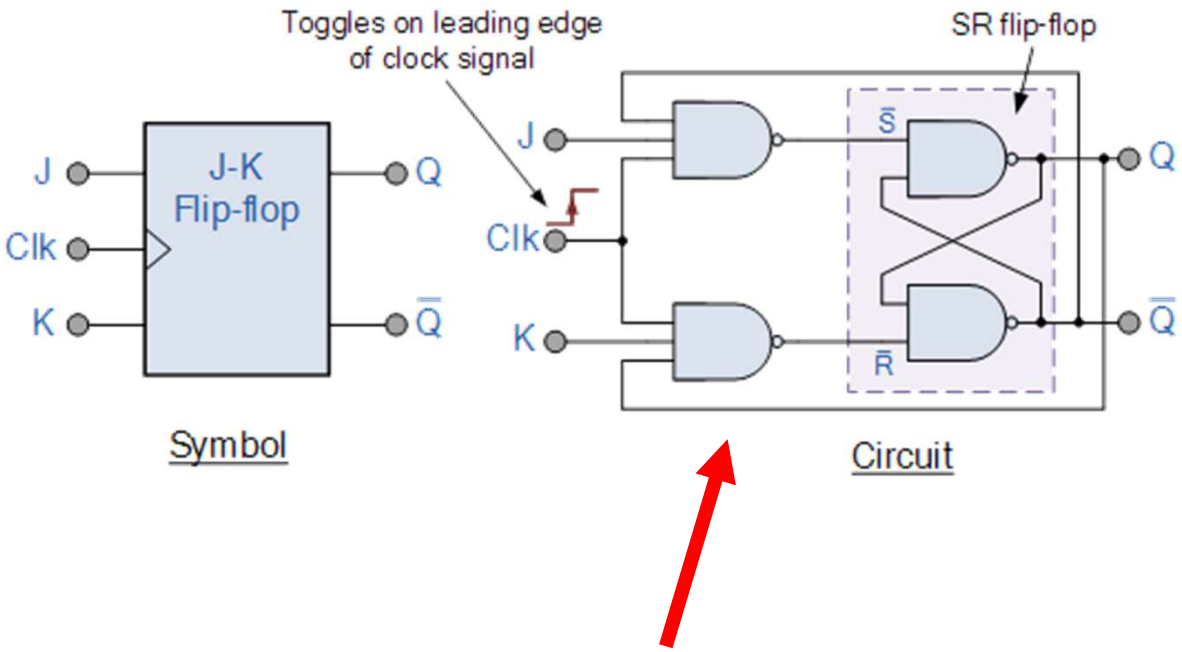
D flip flop in VHDL?



Other flip-flop types

JK flip-flop

- Similar to SR, but inputs named J and K:
- J=1, K=0 sets output to 0
- J=0, K=1 sets output to 0
- But J=1 and K=1 inverts current output on the next clock



T flip flop

- One input – T
- If T=0, nothing happens
- If T=1, output is inverted on the next clock edge
- Commonly implemented as JK flip flop with signals J and K connected to each other

NOTE: Timing-sensitive design.
Clock pulse must be short, otherwise FF would trigger again for J=K=1

Image source and further reading: https://www.electronics-tutorials.ws/sequential/seq_2.html

The Master-Slave JK Flip Flop

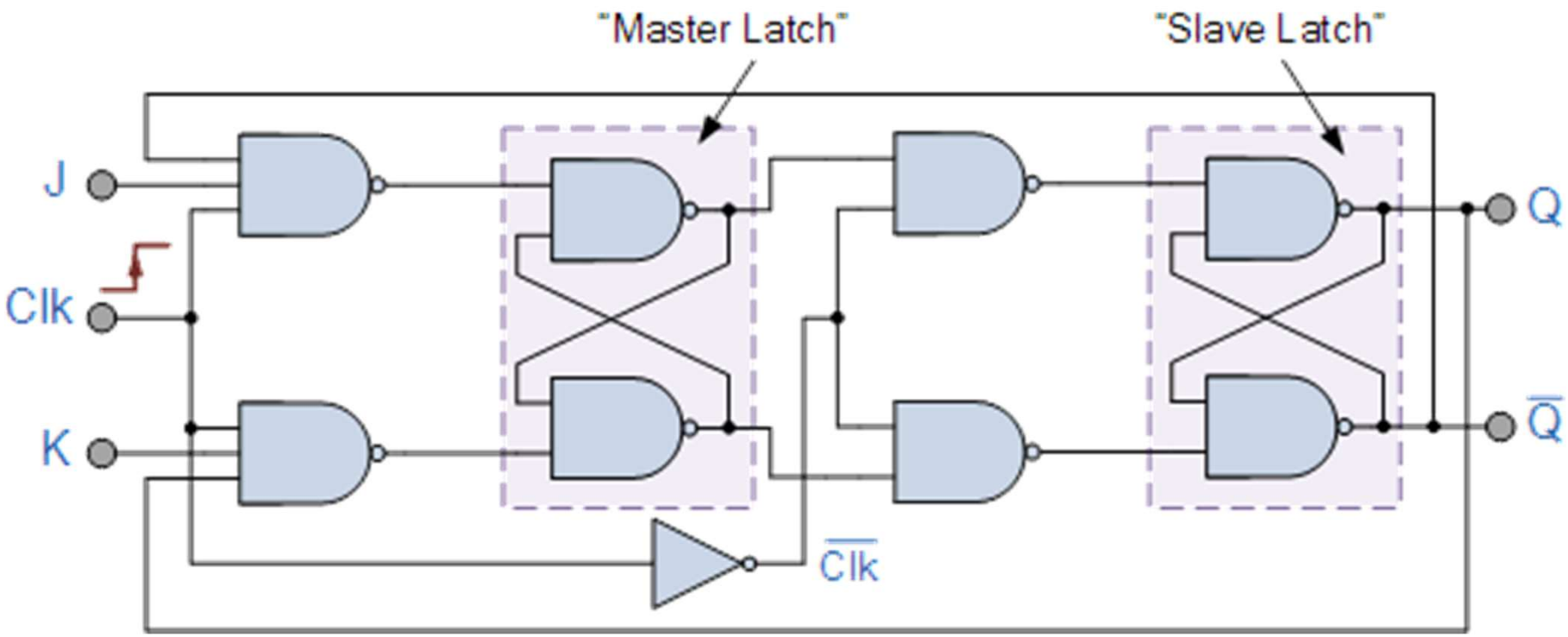
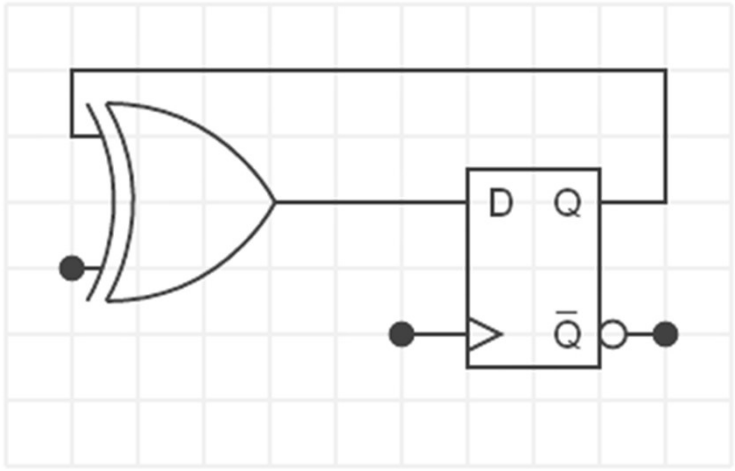


Image source and further reading: https://www.electronics-tutorials.ws/sequential/seq_2.html

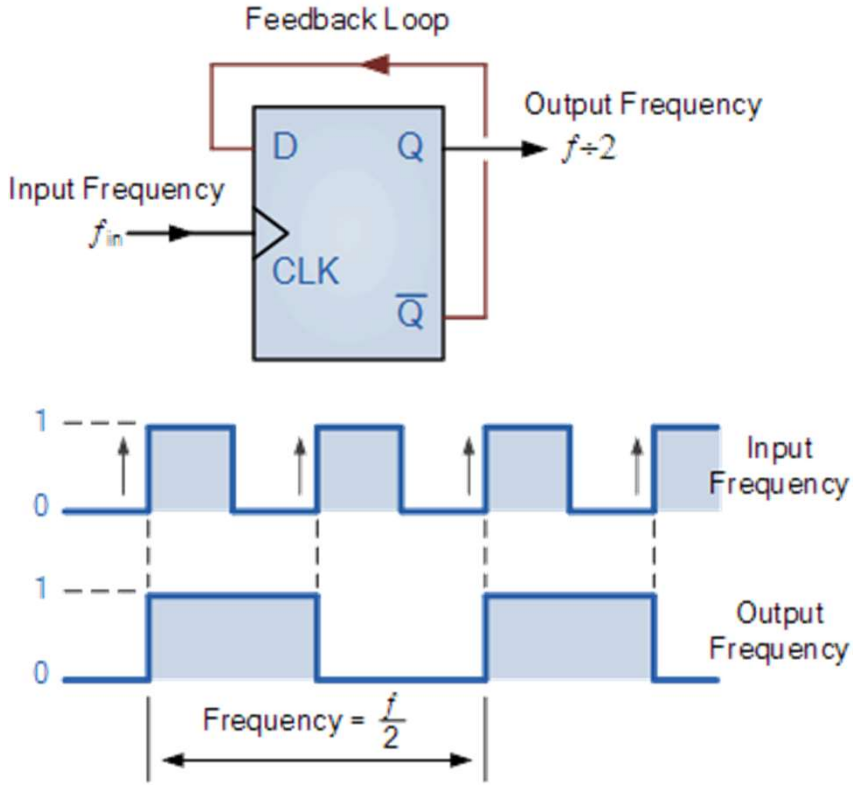
Flip Flop Conversion

- Example: T from D



Further reading: https://www.allaboutcircuits.com/technical-articles/conversion-of-flip-flops-part-iv-d-flip-flops/

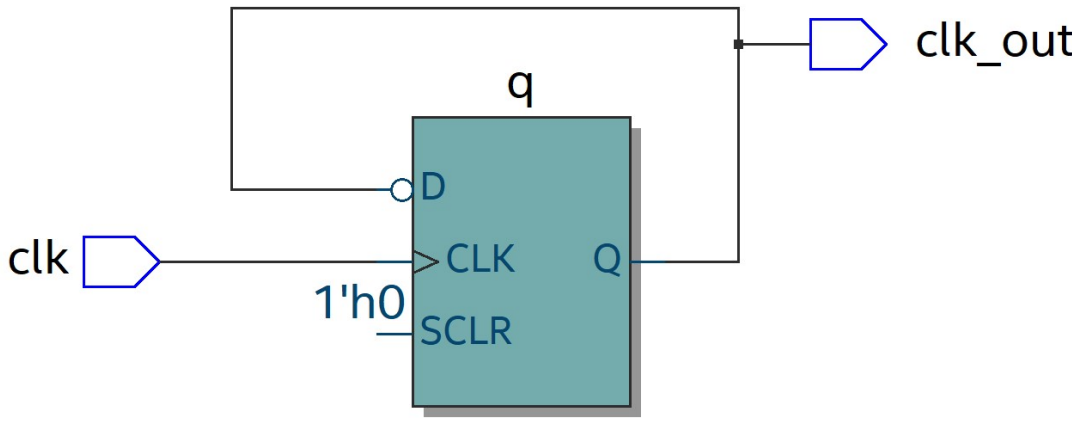
Flip flop usage examples: freq. division (by 2)



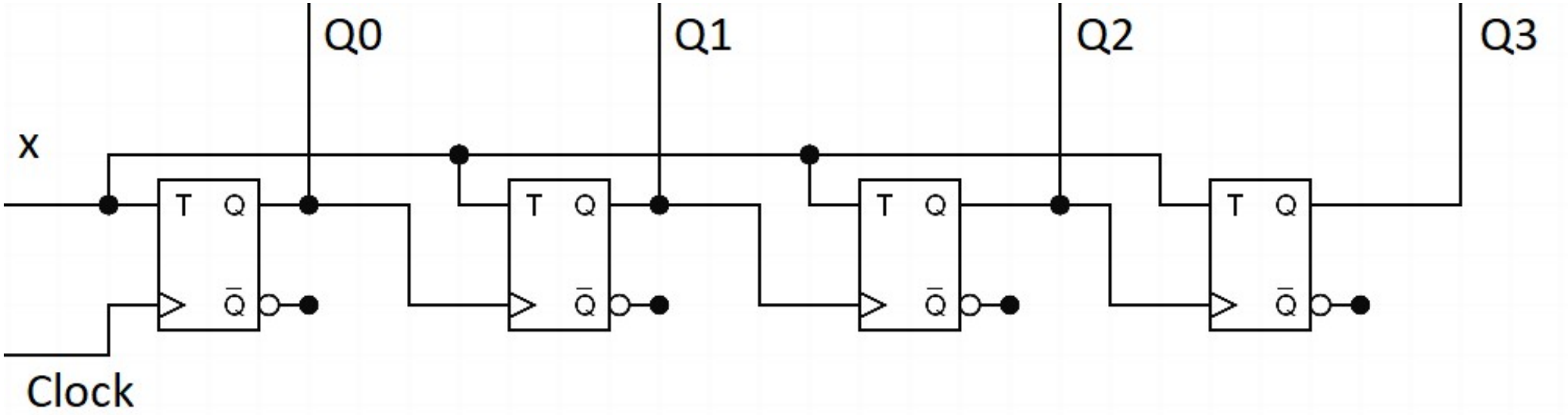
Source and further reading: https://www.electronics-tutorials.ws/counter/count_1.html

VHDL implementation: freq. division (by 2)

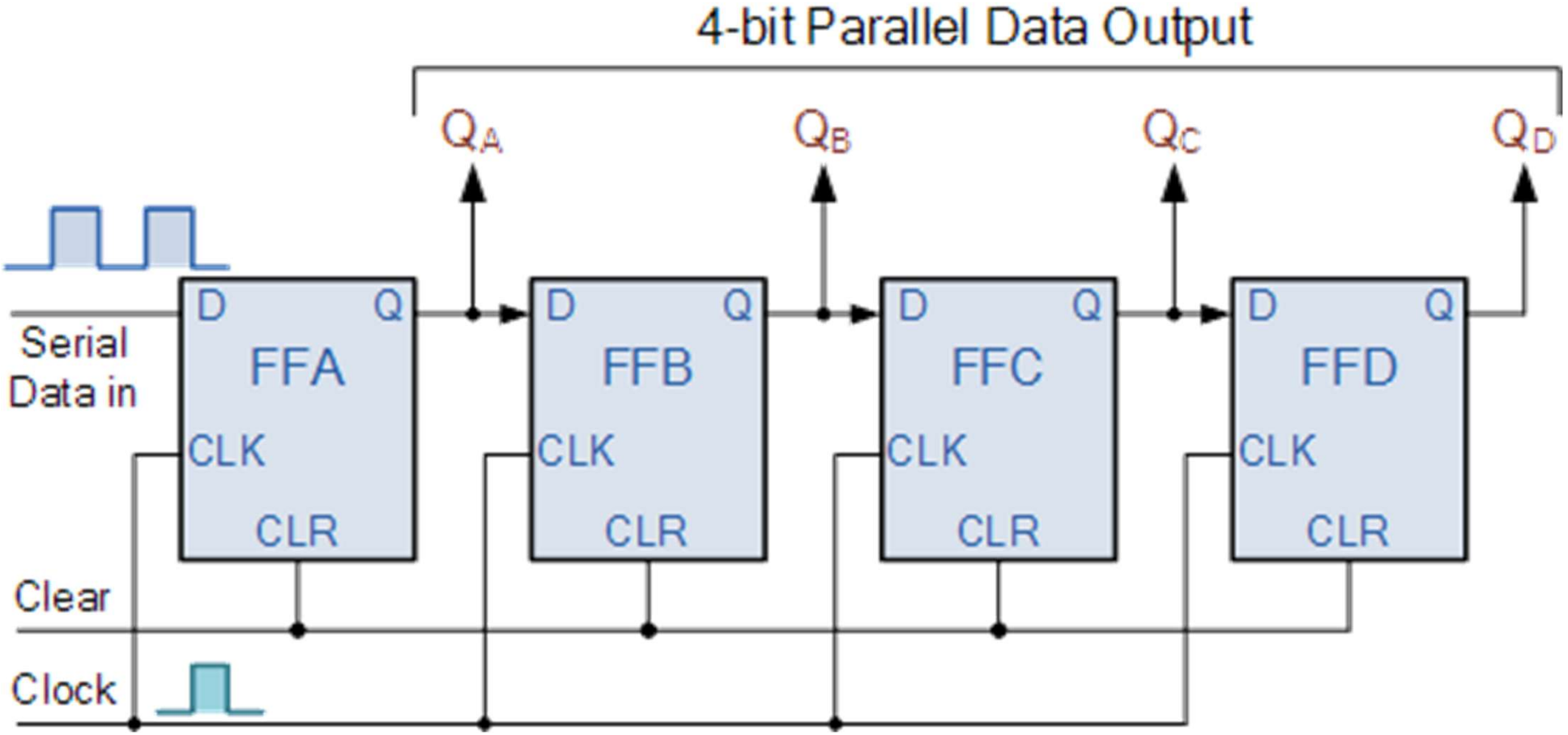
```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

entity clk_div2 is
port(
  clk             : in  std_logic;
  clk_out         : out std_logic);
end clk_div2;

architecture rtl of clk_div2 is
 signal q: std_logic := '0';
begin
 q <= not q when rising_edge(clk);
 clk_out <= q;
end rtl;
```



Flip flop usage examples: Async counter



Flip flop usage examples: shift register



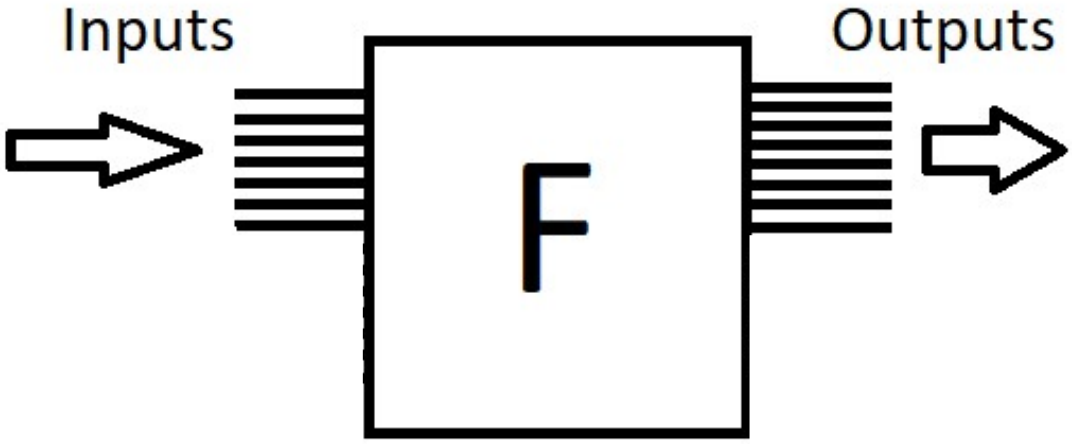
Source and further reading: https://www.electronics-tutorials.ws/sequential/seq_5.html

Sequential circuits

(Finally some real stuff!)

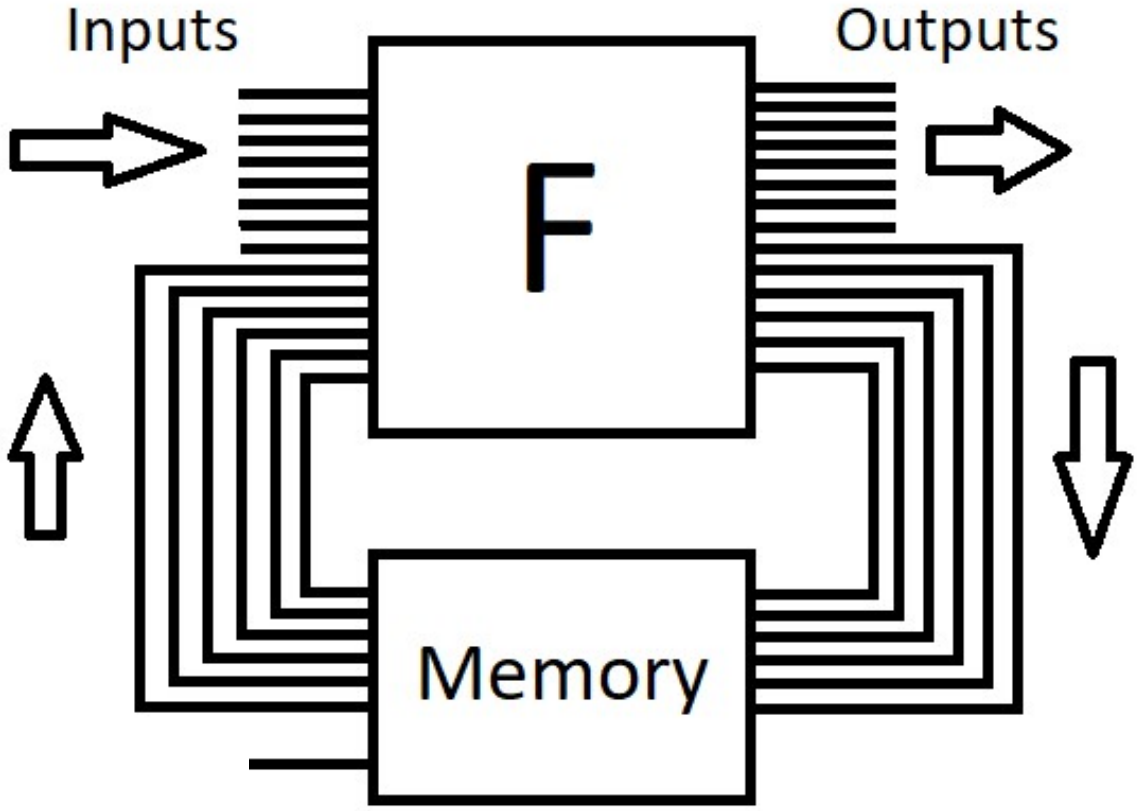
Sequential circuits

- Consider our combinatory circuit black box



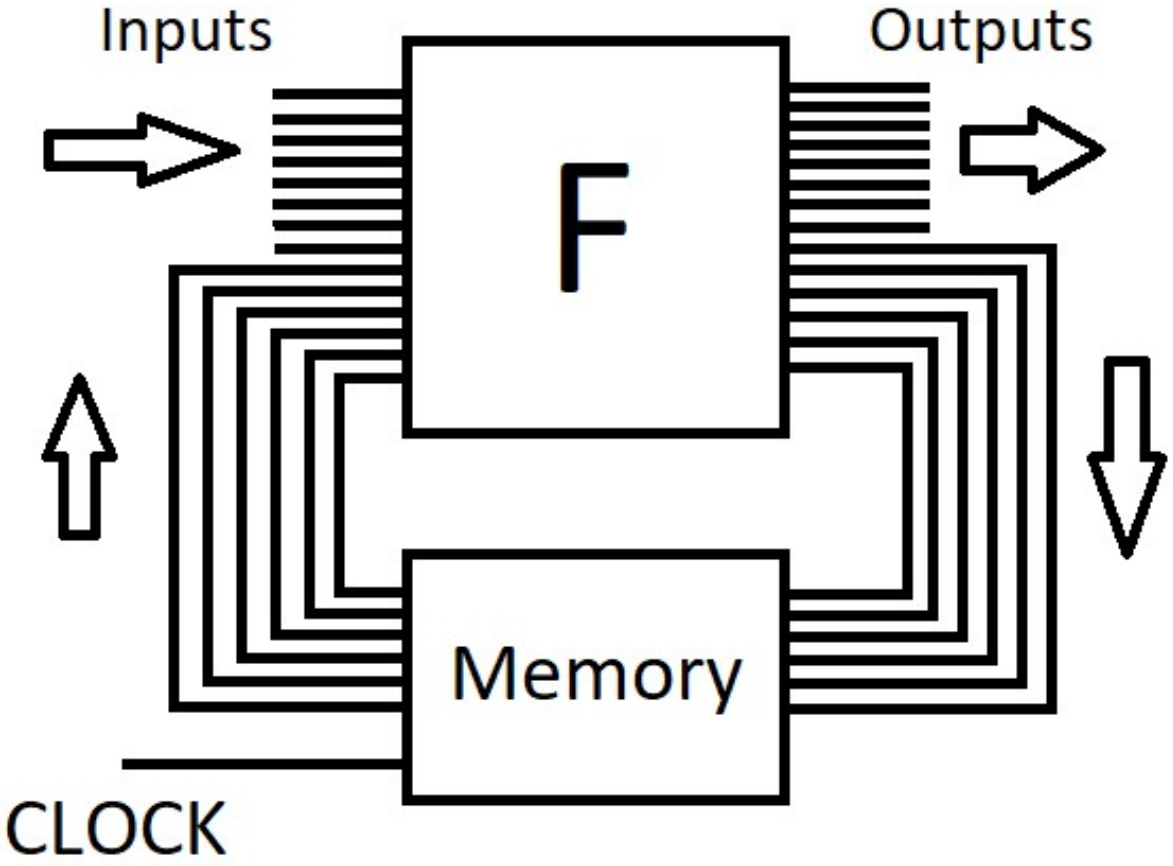
Sequential circuits

- Consider our combinatory circuit black box

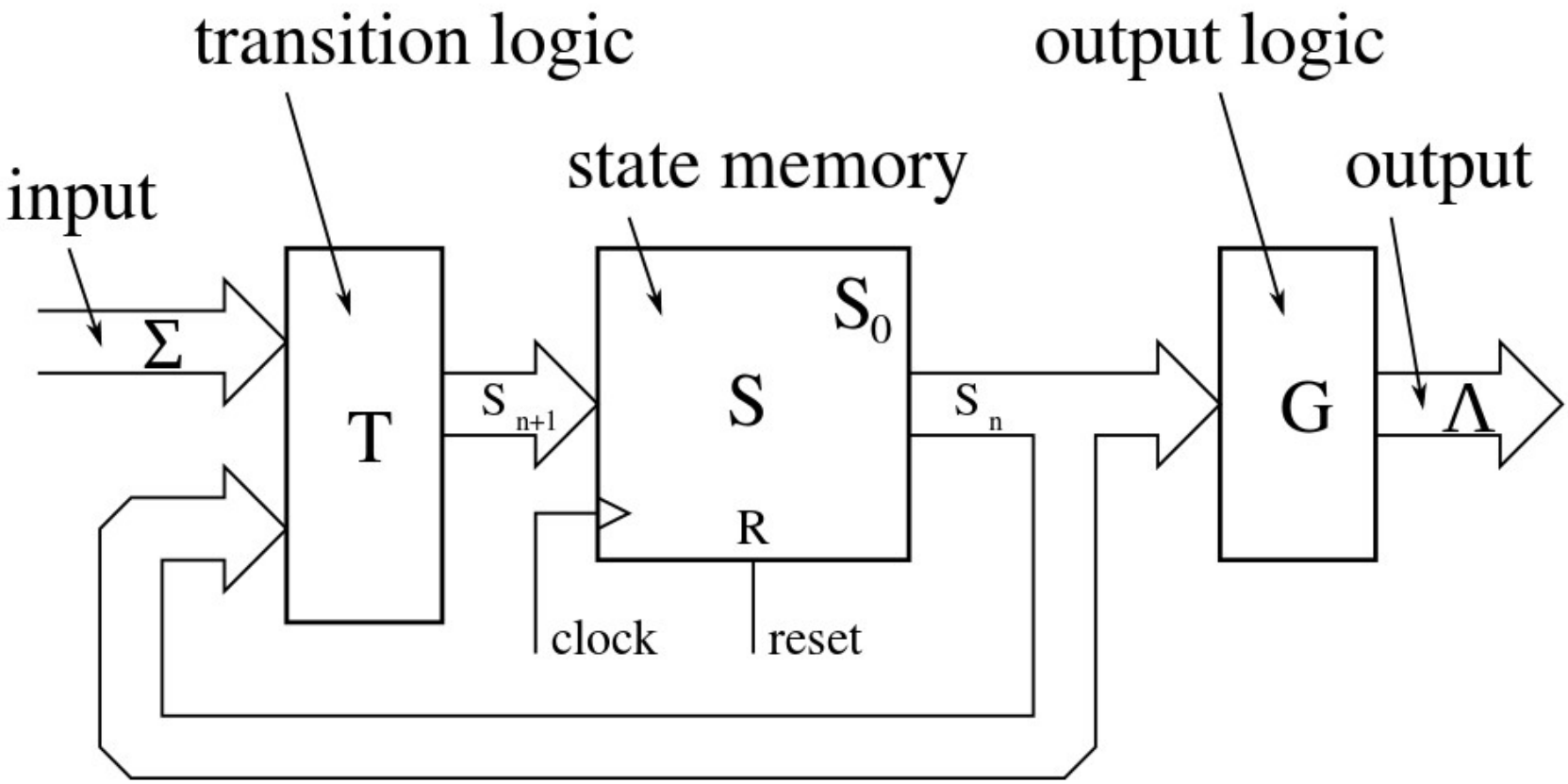


Sequential circuits

- Consider our combinatory circuit black box



Finite State machines: Moore



Finite State machines: Moore

- A finite set of states S
- A start state S0
- A finite set of inputs
- A finite set of outputs
- A transition function T
- An output function G

Finite State machines: Mealy

- Very similar to Moore, but output is a function of both current state and current input
- Less popular in FPGA design as it is harder to verify / validate

Moore by example

- Let's design a state machine that would detect a sequential "111" bit pattern and start outputting '1' after detecting such.
- We would need four states:
  - S0 – initial
  - S1 – seen one '1'
  - S2 – seen two '1's
  - S3 – seen three or more '1's

Current State, S	Next state if I == 0	Next state if I == 1	Circuit output, O
S0	S0	S1	0
S1	S0	S2	0
S2	S0	S3	0
S3	S0	S3	1

Moore by example

- We would need two bit's to encode the state as we have four states, so now change states to values:
  - S0 – 00
  - S1 – 01
  - S2 – 10
  - S3 – 11

S[1]	S[0]	I == 0, next val		I == 1, next val		O
		Sn[1]	Sn[0]	Sn[1]	Sn[0]	
0	0	0	0	0	1	0
0	1	0	0	1	0	0
1	0	0	0	1	1	0
1	1	0	0	1	1	1

- This is pretty much a combinatory function definition table

Moore by example

Sn[1] table:

S[1]	S[0]	I	Sn[1]
0	0	0	0
0	1	0	0
1	0	0	0
1	1	0	0
0	0	1	0
0	1	1	1
1	0	1	1
1	1	1	1

Sn[0] table:

S[1]	S[0]	I	Sn[0]
0	0	0	0
0	1	0	0
1	0	0	0
1	1	0	0
0	0	1	1
0	1	1	0
1	0	1	1
1	1	1	1

S[1]	S[0]	I == 0, next val		I == 1, next val		O
		Sn[1]	Sn [0]	Sn[1]	Sn[0]	
0	0	0	0	0	1	0
0	1	0	0	1	0	0
1	0	0	0	1	1	0
1	1	0	0	1	1	1

Output table

S[1]	S[0]	0
0	0	0
0	1	0
1	0	0
1	1	1

Moore by example

Sn[1] table:

S[1]	S[0]	I	Sn[1]
0	0	0	0
0	1	0	0
1	0	0	0
1	1	0	0
0	0	1	0
0	1	1	1
1	0	1	1
1	1	1	1

$$Sn_1 = \overline{S_1}S_0I + S_1\overline{S_0}I + S_1S_0I$$
$$= (\overline{S_1}S_0 + S_1(\overline{S_0} + S_0))I$$
$$= (\overline{S_1}S_0 + S_1)I$$
$$= (S_0 + S_1)I$$

Note: this can be done easier using Karnaugh map: https://en.wikipedia.org/wiki/Karnaugh_map

Moore by example

Sn[0] table:

S[1]	S[0]	I	Sn[0]
0	0	0	0
0	1	0	0
1	0	0	0
1	1	0	0
0	0	1	1
0	1	1	0
1	0	1	1
1	1	1	1

$$Sn_0 = \overline{S_1}\,\overline{S_0}I + S_1\overline{S_0}I + S_1S_0I$$
$$= (\overline{S_1}\,\overline{S_0} + S_1\overline{S_0} + S_1S_0)I$$
$$= (\overline{S_0} + S_1S_0)I$$
$$= (\overline{S_0} + S_1)I$$

Note: this can be done easier using Karnaugh map: https://en.wikipedia.org/wiki/Karnaugh_map

Moore by example

Output table

S[1]	S[0]	O
0	0	0
0	1	0
1	0	0
1	1	1

$$Out = S_1 S_0$$

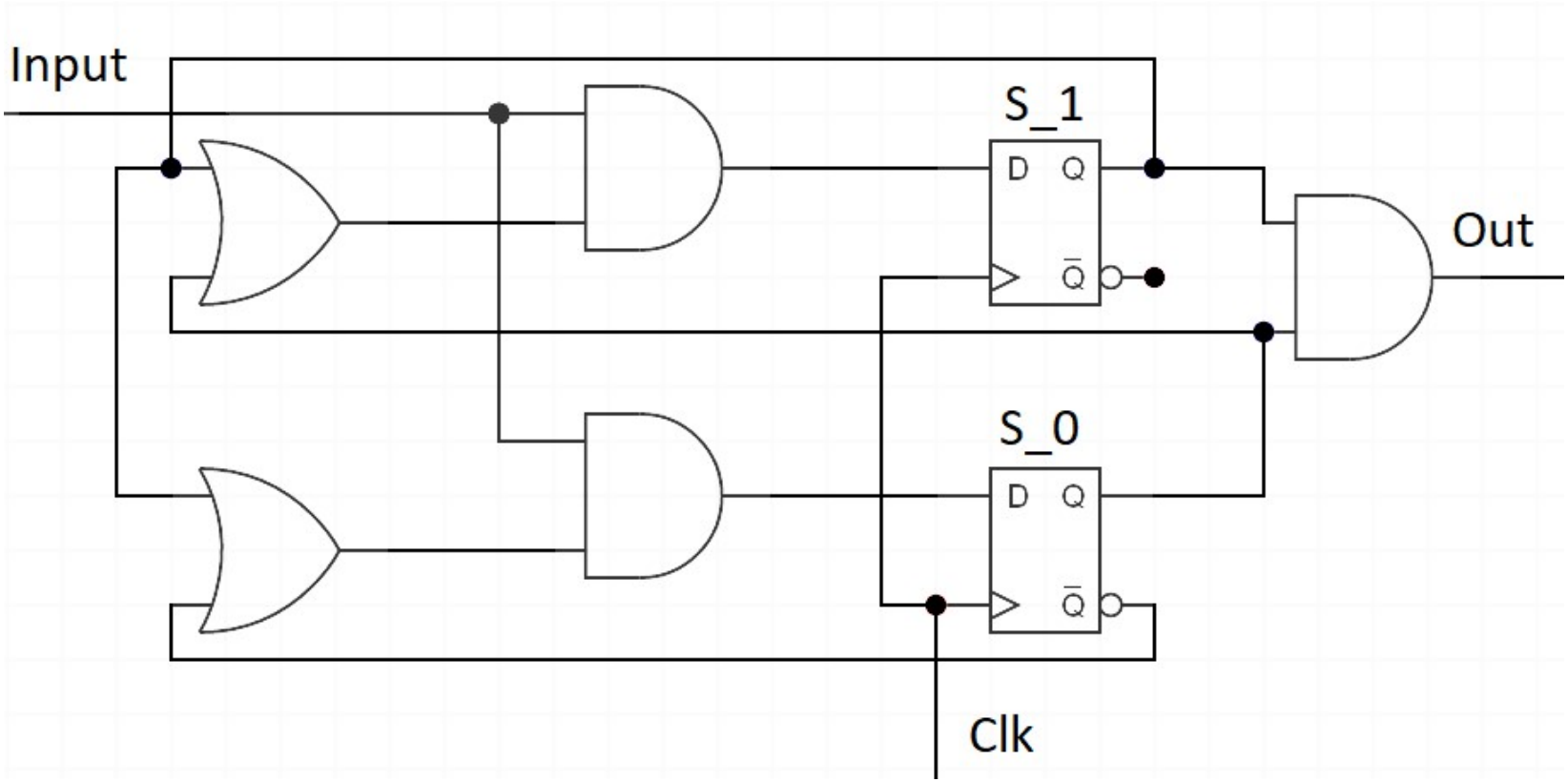
Note: this can't be done easier using Karnaugh map: https://en.wikipedia.org/wiki/Karnaugh_map

Moore by example

$Sn_1 = (S_1 + S_0)I$

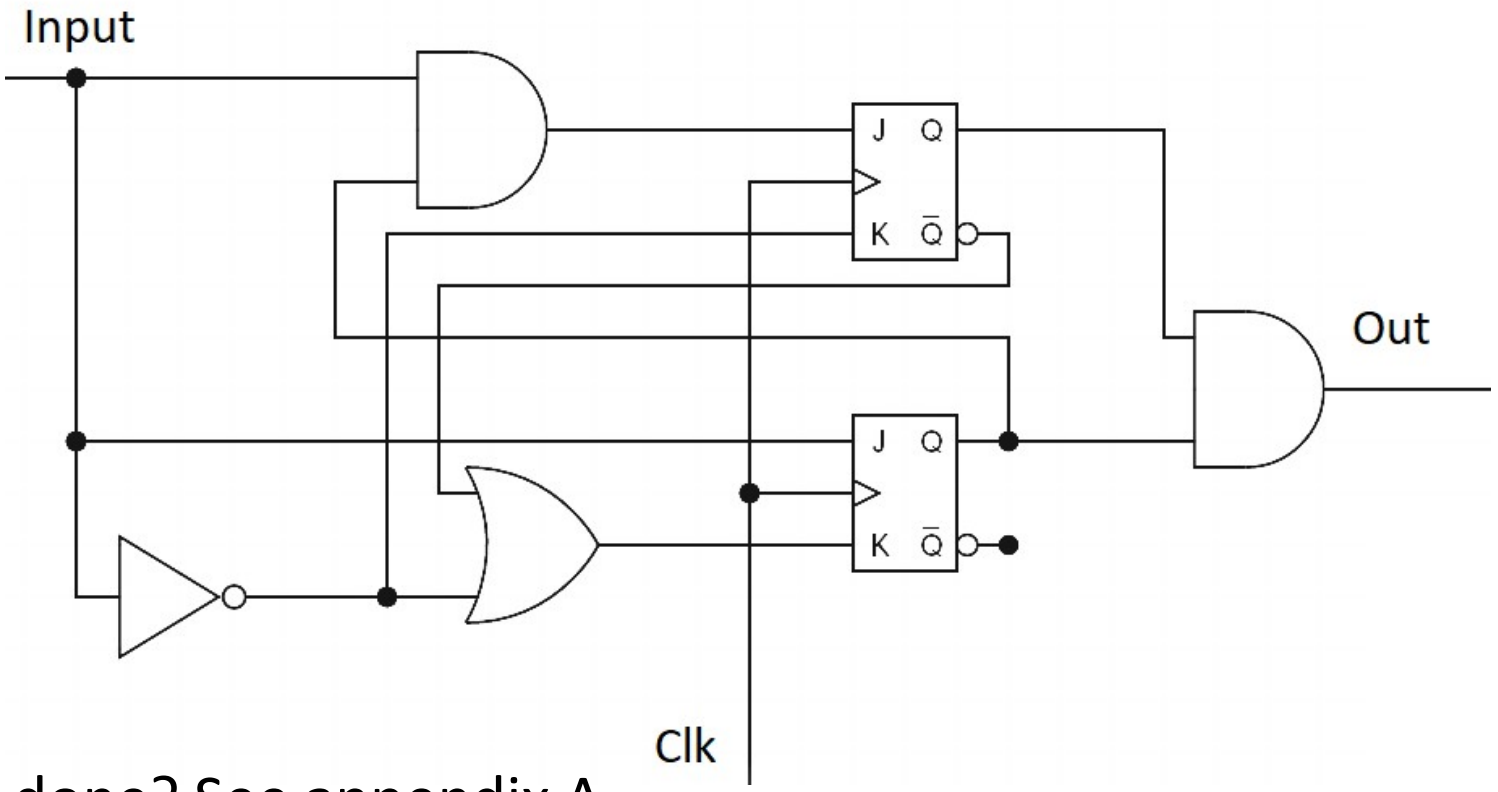
$Sn_0 = (\overline{S_0} + S_1)I$

$Out = S_1 S_0$



Alternative implementation with JK Flip Flop

- Different types of flip flops can be more or less efficient for particular purpose



- How is it done? See appendix A.

FSM: state encoding matters

- There are multiple ways our states S0…SN can be encoded in binary form, some produce more optimal gate networks, using more or less flip flops (ultimate case – “one hot” using N flip flops for N states)
- Some other types of flip flops can also produce more optimal gate networks
- Important: since we write VHDL and not designing circuits by hands, we don’t have to worry about the most efficient way of encoding, synthesis tool would optimise it for us.

Further reading: ch.9 of *“Introduction to Digital Systems: Modeling, Synthesis, and Simulation Using VHDL”* by Mohammed Ferdjallah, ISBN: 9780470900550, available in Safari Books

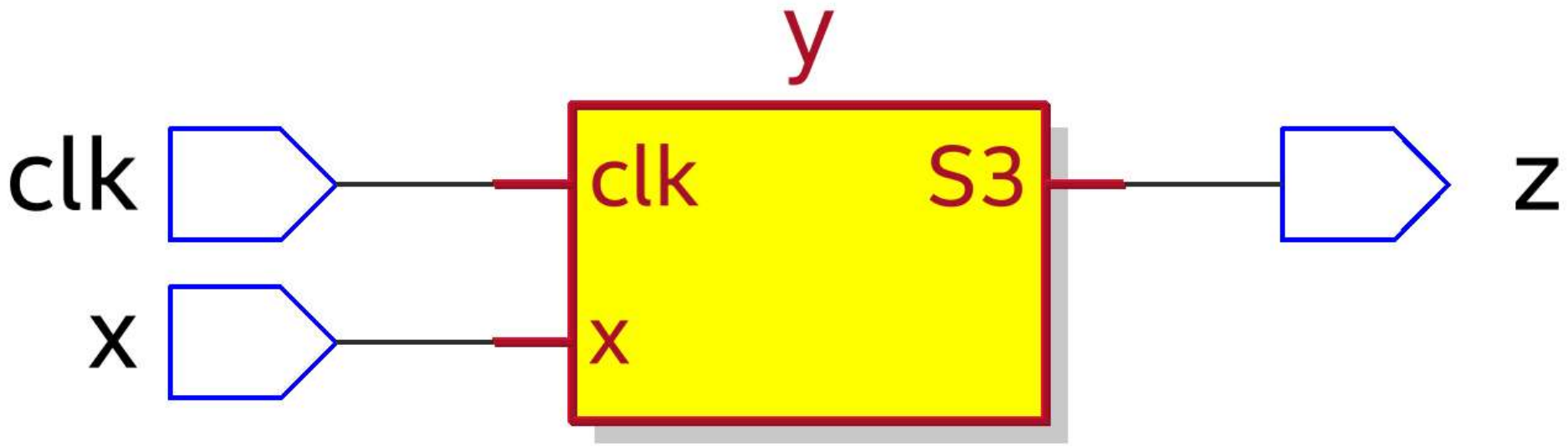
Moore by example – VHDL implementation

```
entity fsm is
    port (
        x, clk  :   in  std_logic;
        z       :   out std_logic)
end fsm;
```

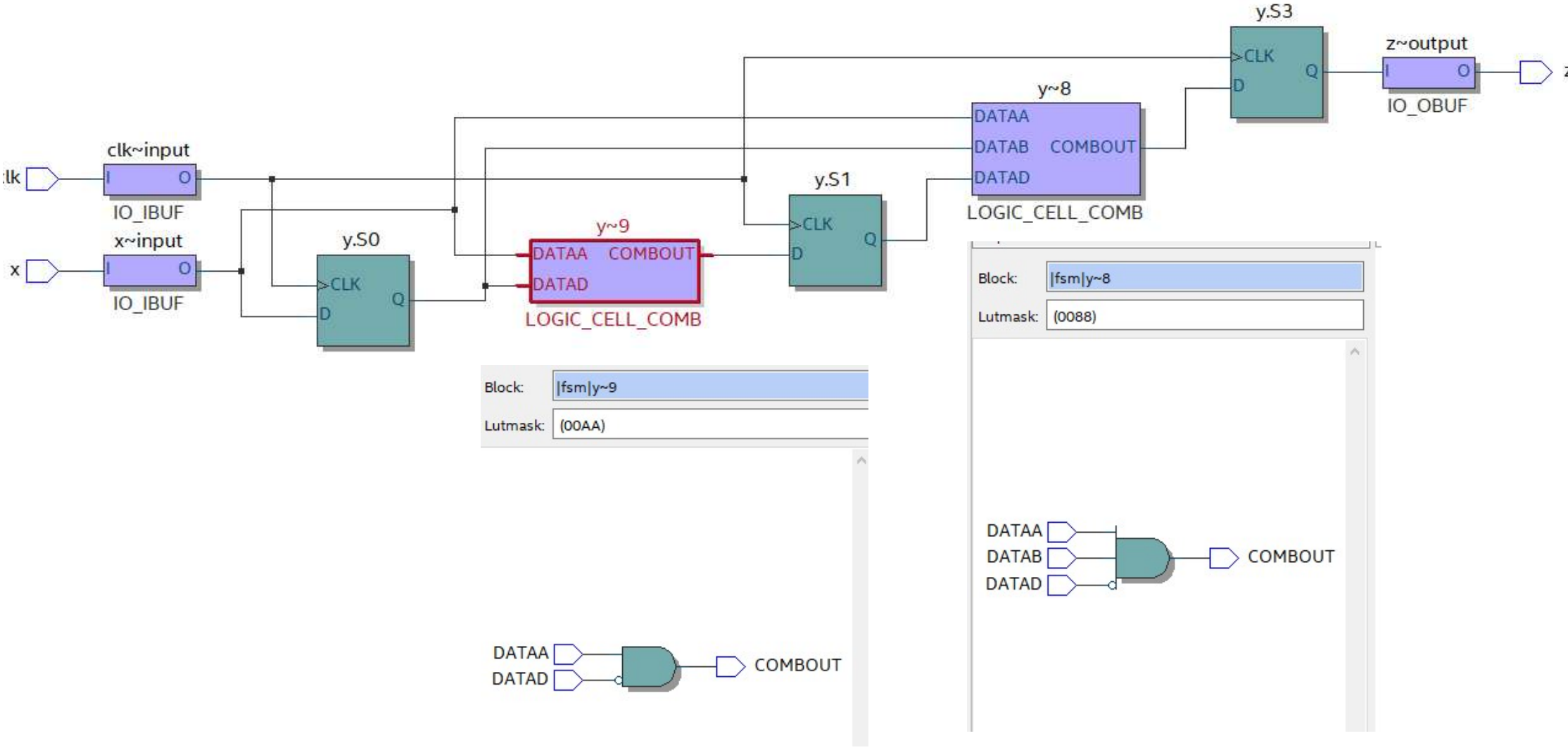
Moore by example – VHDL implementation

```
architecture fsm_behavior of fsm is
    type state_type is (S0, S1, S2, S3);
    signal y: state_type;
begin
    process (clk , x)
    begin
        if clk'event and clk = '1' then
            case y is
                when S0 =>
                    if x = '1' then y <= S1;
                    else y <= S0;
                    end if;
                when S1 =>
                    if x = '1' then y <= S2;
                    else y <= S0;
                    end if;
                when S2 =>
                    if x = '1' then y <= S3;
                    else y <= S0;
                    end if;
                when S3 =>
                    if x = '1' then y <= S3;
                    else y <= S0;
                    end if;
            end case;
        end if;
    end process;

    z <= '1' when y = s3 else '0';
end fsm_behavior;
```

VHDL synthesis - Intel

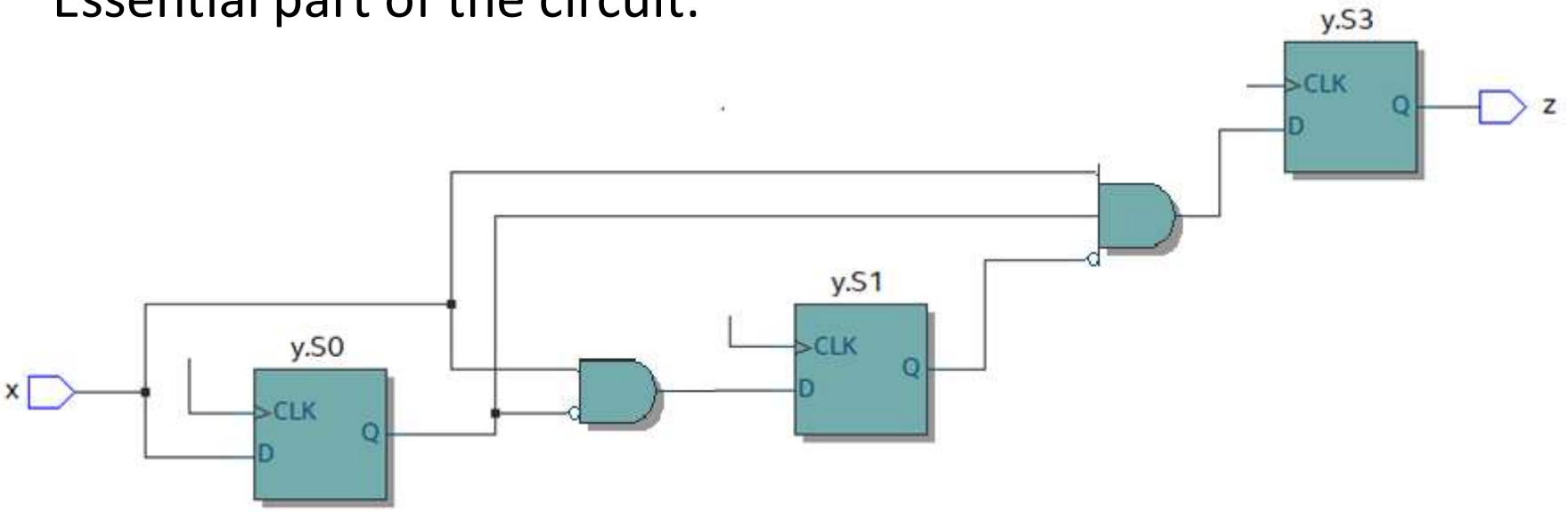


Synthesis –Technology Map – Intel



Synthesis –Technology Map – Intel

Essential part of the circuit:



$Sn_0 = x$ $Sn_1 = \overline{S_0}x$ $Sn_3 = \overline{S_1}S_0x$ $Out = S_3$

Synthesis –Technology Map – Intel

$Sn_0 = x$ $Sn_1 = \overline{S_0}x$ $Sn_3 = \overline{S_1}S_0x$ $Out = S_3$

Note: at any state, x=0 sets S_0, S_1 and S_3 into 0, so let's consider the state progression for a x=1 for a number of consecutives ticks:

Time	S0	S1	S3	x	Sn0	Sn1	Sn3	Out
0	0	0	0	1	1	1	0	0
1	1	1	0	1	1	0	0	0
2	1	0	0	1	1	0	1	0
3	1	0	1	0	0	0	0	1

Key point: synthesis tool can heavily optimise the circuit out, but the resulting behaviour remains the same

(not a flying spaghetti monster)

Another FSM example: 4-bit counter equivalent

- Let's now try building the state machine with 16 states S0…S15 and one input x
- When x=0, machine state doesn't change
- When x=1, machine state moves to the next one: S0 to S1, .. S7 to S8, etc. If state was S15, machine goes back to S0 and process moves on
- Output would be set equal to the current state encoded as binary
- We would be using T Flip Flops – apparently this type of FSM can be very efficiently implemented using T FF.
- Similar way to JK implementation, gate network must produce value for T flip flop input that would cause correct resulting state: T=1 if we need to toggle current value of a state bit, T=0 if we must keep it.

FSM design example: 4-bit counter equivalent

S	Sn or x=0	Sn for x=1
S0	S0	S1
S1	S1	S2
S2	S2	S3
S3	S3	S4
S4	S4	S5
S5	S5	S6
S6	S6	S7
S7	S7	S8
S8	S8	S9
S9	S9	S10
S10	S10	S11
S11	S11	S12
S12	S12	S13
S13	S13	S14
S14	S14	S15
S15	S15	S0

S	Sn or x=0	T for x=0	Sn for x=1	T for x=1
0000	0000		0001	
0001	0001		0010	
0010	0010		0011	
0011	0011		0100	
0100	0100		0101	
0101	0101		0110	
0110	0110		0111	
0111	0111		1000	
1000	1000		1001	
1001	1001		1010	
1010	1010		1011	
1011	1011		1100	
1100	1100		1101	
1101	1101		1110	
1110	1110		1111	
1111	1111		0000	

FSM design example: 4-bit counter equivalent

S	Sn or x=0	Sn for x=1
S0	S0	S1
S1	S1	S2
S2	S2	S3
S3	S3	S4
S4	S4	S5
S5	S5	S6
S6	S6	S7
S7	S7	S8
S8	S8	S9
S9	S9	S10
S10	S10	S11
S11	S11	S12
S12	S12	S13
S13	S13	S14
S14	S14	S15
S15	S15	S0

S	Sn or x=0	T for x=0	Sn for x=1	T for x=1
0000	0000	0000	0001	0001
0001	0001	0000	0010	0011
0010	0010	0000	0011	0001
0011	0011	0000	0100	0111
0100	0100	0000	0101	0001
0101	0101	0000	0110	0011
0110	0110	0000	0111	0001
0111	0111	0000	1000	1111
1000	1000	0000	1001	0001
1001	1001	0000	1010	0011
1010	1010	0000	1011	0001
1011	1011	0000	1100	0111
1100	1100	0000	1101	0001
1101	1101	0000	1110	0011
1110	1110	0000	1111	0001
1111	1111	0000	0000	1111

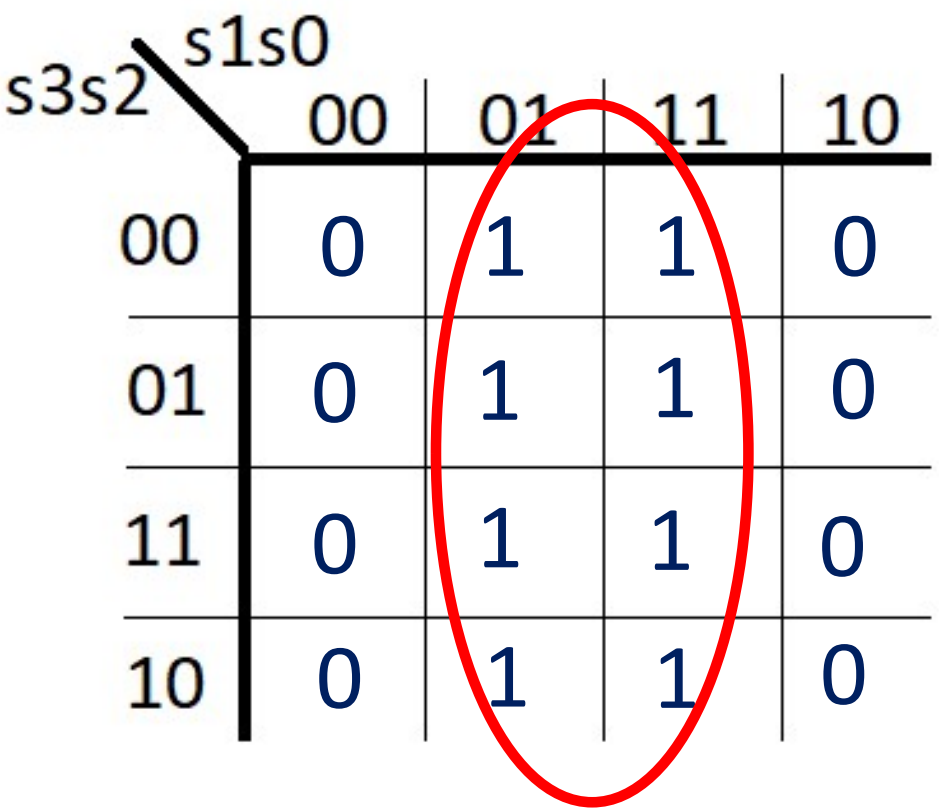
FSM design example: 4-bit counter equivalent: T0

S	T for x=1
0000	0001
0001	0011
0010	0001
0011	0111
0100	0001
0101	0011
0110	0001
0111	1111
1000	0001
1001	0011
1010	0001
1011	0111
1100	0001
1101	0011
1110	0001
1111	1111

$$T_0 = x$$

FSM design example: 4-bit counter equivalent: T1

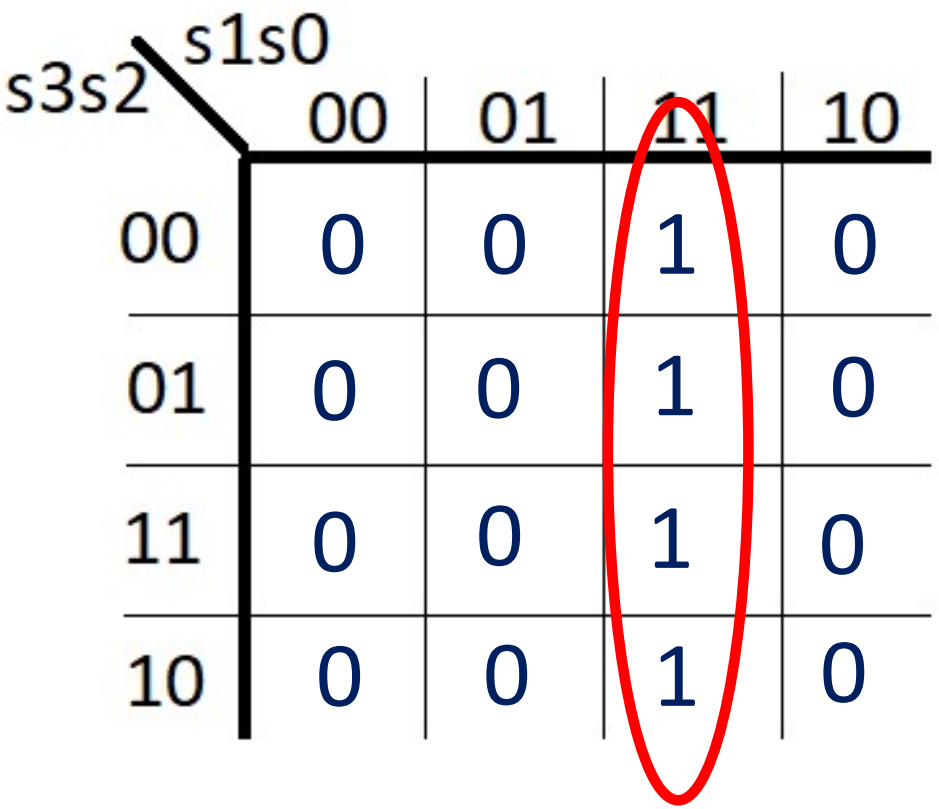
S	T for x=1
0000	0001
0001	0011
0010	0001
0011	0111
0100	0001
0101	0011
0110	0001
0111	1111
1000	0001
1001	0011
1010	0001
1011	0111
1100	0001
1101	0011
1110	0001
1111	1111



$$T_1 = S_0 x$$

FSM design example: 4-bit counter equivalent: T2

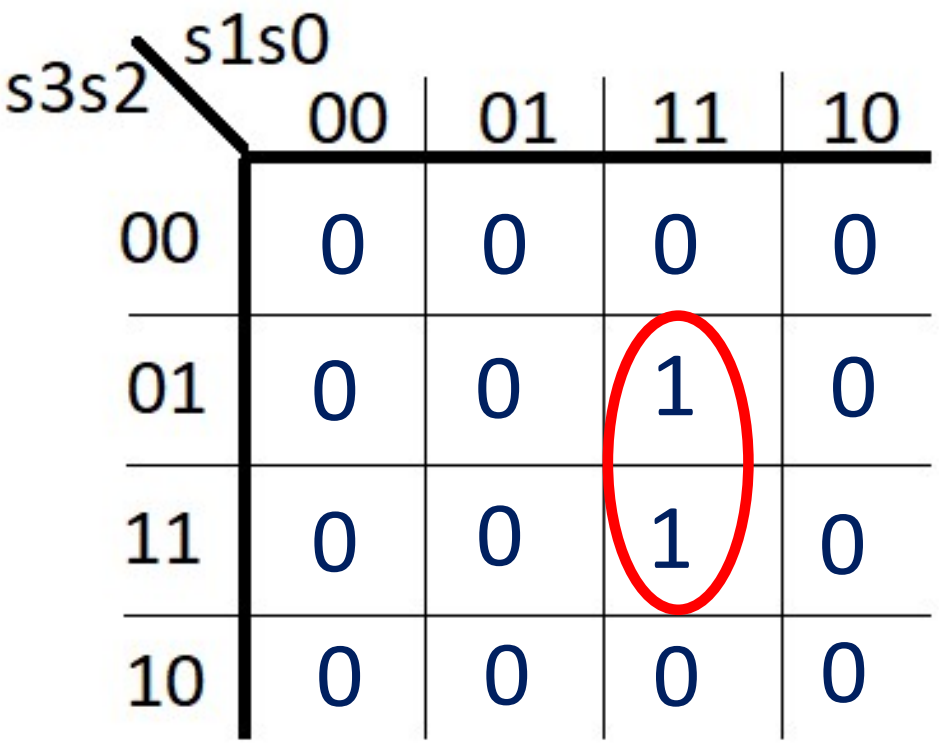
S	T for x=1
0000	0001
0001	0011
0010	0001
0011	0111
0100	0001
0101	0011
0110	0001
0111	1111
1000	0001
1001	0011
1010	0001
1011	0111
1100	0001
1101	0011
1110	0001
1111	1111



$$T_2 = S_1 S_0 x$$

FSM design example: 4-bit counter equivalent: T3

S	T for x=1
0000	0001
0001	0011
0010	0001
0011	0111
0100	0001
0101	0011
0110	0001
0111	1111
1000	0001
1001	0011
1010	0001
1011	0111
1100	0001
1101	0011
1110	0001
1111	1111

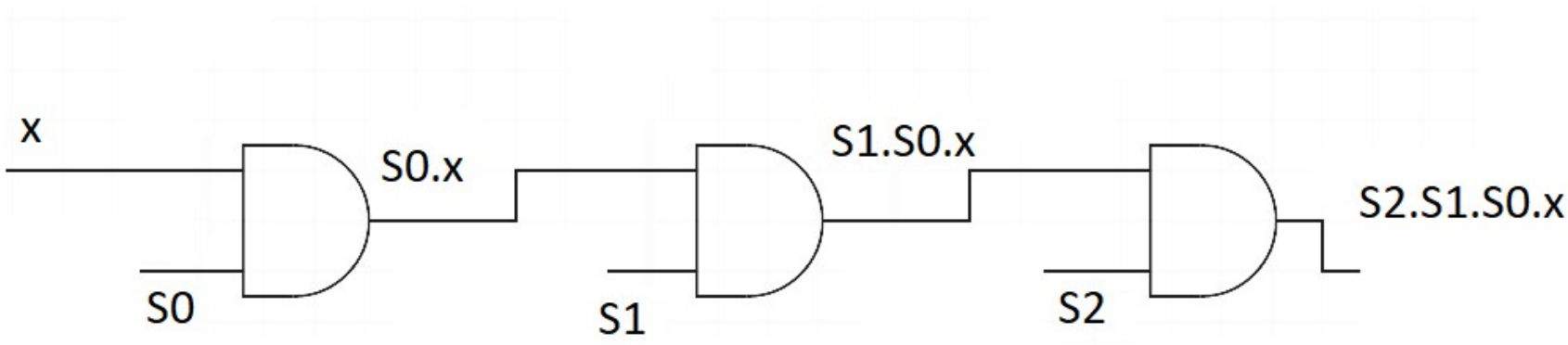


$$T_3 = S_2 S_1 S_0 x$$

FSM design example: 4-bit counter equivalent: summary

$$T_0 = x \qquad T_2 = S_1 S_0 x$$

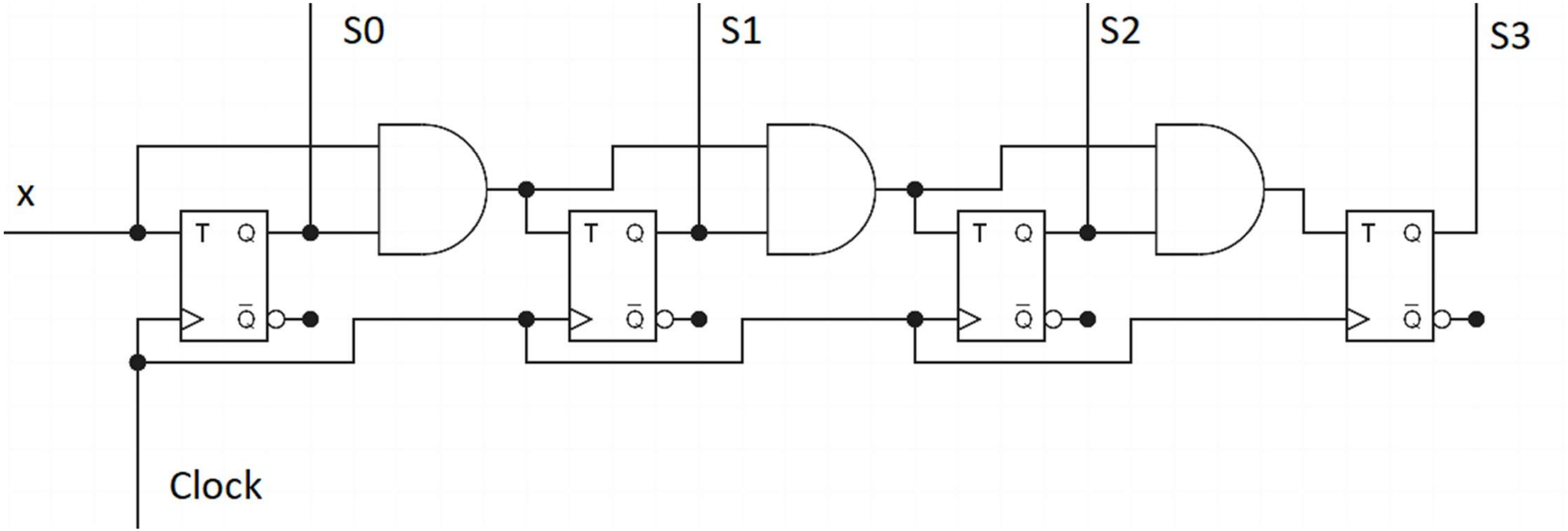
$$T_1 = S_0 x \qquad T_3 = S_2 S_1 S_0 x$$



FSM design example: 4-bit counter equivalent: summary

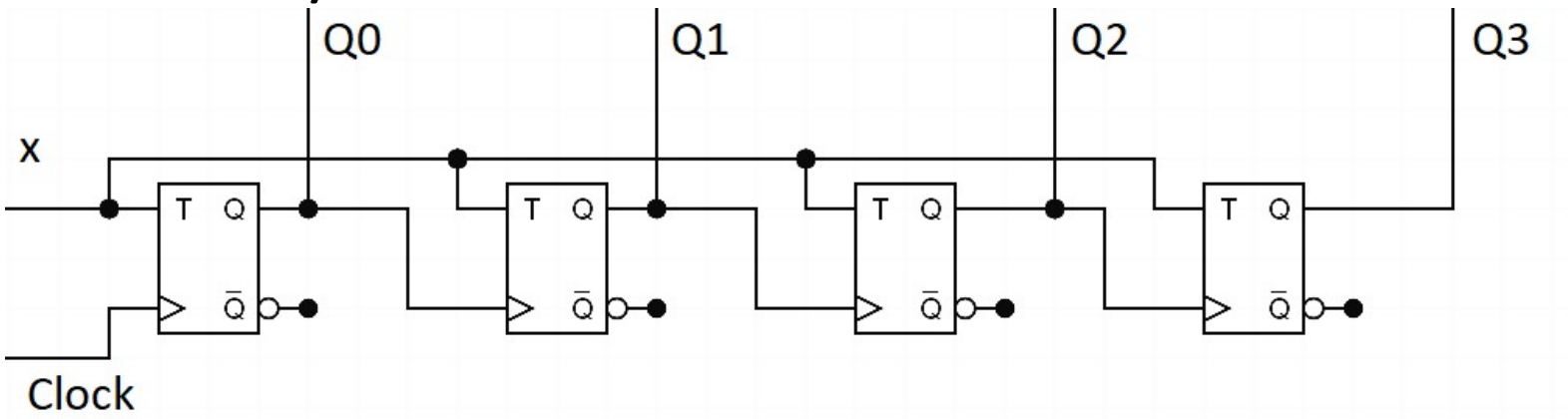
$$T_0 = x \qquad T_2 = S_1 S_0 x$$

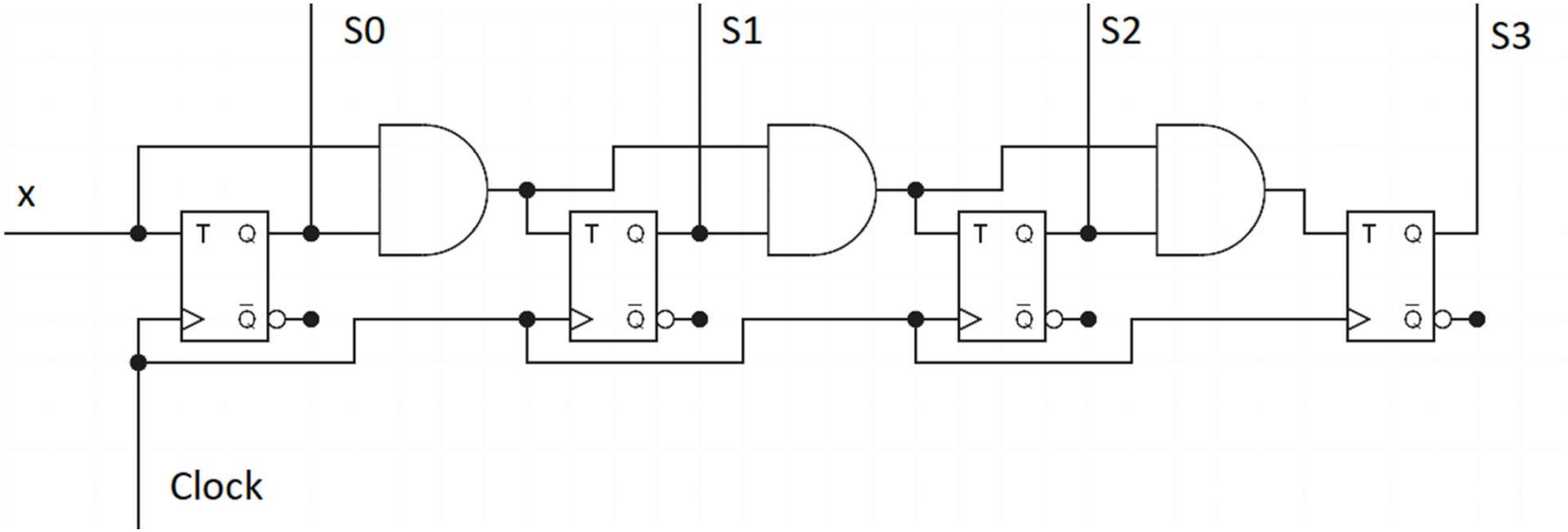
$$T_1 = S_0 x \qquad T_3 = S_2 S_1 S_0 x$$



This circuit is called “synchronous counter”

Sync vs Async counters





Time to get serious

Tired? Would certainly be soon!

Ethernet Preamble

- Now let's try creating a design that would detect a pattern of 56 bits of alternating 1s and 0s
- When such pattern is detected – output single '1' and return into waiting state
- If pattern is broken mid-pattern – for simplicity we return into wait / idle state
- Creating 56-ish different states for this FSM would be impractical
- Thus, we would need variables, or more precisely – counters, to minimise total number of states
- Note: a sequence of 56 alternating 1s and 0s is a preamble of the ethernet frame

```
enum State { IDLE, SEEN_0, SEEN_1, DETECTED_PATTERN };
int st = IDLE;
int num_pairs = 0;
void run_on_clk(int input, int* output) {
    switch (state) {
        case IDLE:
            num_pairs = 0;
            if (x != 0) state = SEEN_1;
            break;
        case SEEN_1:
            if (x == 0) {
                state = SEEN_0;
                num_pairs += 1;
            }
            else
                state = IDLE;
            break;
        case SEEN_0:
            if (num_pairs == 26) state = DETECTED_PATTERN;
            else if (x != 0) state = SEEN_1;
            else state = IDLE;
            break;
        case DETECTED_PATTERN:
            state = IDLE;
            break;
    }
    *out = (state == DETECTED_PATTERN);
}
```

Approximate software-ish implementation

Main design principles

- Proceed with implementing a Moore FSM
- Add an external counter module
- Treating the current counter value as an external input into the FSM
- FSM must generate control signals to increment / clear counter values when appropriate/necessary

FSM: eth preamble

- State transition table would be as follows.
- ‘x’ stands for the current value of the input, and ‘cnt’ for the current counter value:

State	Next State				Main output, O	Increment counter enable, C_I	Reset counter enable, C_R
	cnt /= 26 x=0	cnt /= 26 x=1	cnt = 26 x=0	cnt = 26 x=1			
IDLE	IDLE	SEEN_1	IDLE	IDLE	0	0	1
SEEN_0	IDLE	SEEN_1	DET_PATTERN	DET_PATTERN	0	0	0
SEEN_1	SEEN_0	IDLE	SEEN_0	IDLE	0	1	0
DET_PATTERN	IDLE	IDLE	IDLE	IDLE	1	0	0

- For cnt we need 5-bit register, as its value is not exceeding 31.
- We compare for == 28 (11100 in binary) by simply doing:

$$C_E = c_4 c_3 c_2 \overline{c_1}\,\overline{c_0}$$

(where c_n are individual bits of the counter, and C_E is a “summary” signal)

FSM: eth preamble

Current State		Next State								Main output, O	Increment counter enable, C_I	Reset counter enable, C_R
		cnt /= 26 (C_E=0) x=0		cnt /= 26 (C_E=0) x=1		cnt = 26 (C_E=1) x=0		cnt = 26 (C_E=1) x=1				
S_1	S_0	Sn_1	Sn_0	Sn_1	Sn_0	Sn_1	Sn_0	Sn_1	Sn_0			
0	0	0	0	1	0	0	0	0	0	0	0	1
0	1	0	0	1	0	1	1	1	1	0	0	0
1	0	0	1	0	0	0	1	0	0	0	1	0
1	1	0	0	0	0	0	0	0	0	1	0	0

Sn_1:

S_1S_0 \ C_E,X	00	01	11	10
00	0	1	0	0
01	0	1	1	1
11	0	0	0	0
10	0	0	0	0

$$Sn_1 = \overline{S_1}\,\overline{C_E}X + \overline{S_1}S_0X + \overline{S_1}S_0C_E$$
$$= \overline{(S_1 + C_E)}X + \overline{S_1}S_0(X + C_E)$$

FSM: eth preamble

Current State		Next State								Main output, O	Increment counter enable, C_I	Reset counter enable, C_R
		cnt /= 26 (C_E=0) x=0		cnt /= 26 (C_E=0) x=1		cnt = 26 (C_E=1) x=0		cnt = 26 (C_E=1) x=1				
S_1	S_0	Sn_1	Sn_0	Sn_1	Sn_0	Sn_1	Sn_0	Sn_1	Sn_0			
0	0	0	0	1	0	0	0	0	0	0	0	1
0	1	0	0	1	0	1	1	1	1	0	0	0
1	0	0	1	0	0	0	1	0	0	0	1	0
1	1	0	0	0	0	0	0	0	0	1	0	0

Sn_0:

S_1S_0 \ C_E,X	00	01	11	10
00	0	0	0	0
01	0	0	1	1
11	0	0	0	0
10	1	0	0	1

$$Sn_0 = \overline{S_1} S_0 C_E + S_1 \overline{S_0} \overline{X}$$

FSM: eth preamble

Current State		Next State								Main output, O	Increment counter enable, C_I	Reset counter enable, C_R
		cnt /= 26 (C_E=0) x=0		cnt /= 26 (C_E=0) x=1		cnt = 26 (C_E=1) x=0		cnt = 26 (C_E=1) x=1				
S_1	S_0	Sn_1	Sn_0	Sn_1	Sn_0	Sn_1	Sn_0	Sn_1	Sn_0			
0	0	0	0	1	0	0	0	0	0	0	0	1
0	1	0	0	1	0	1	1	1	1	0	0	0
1	0	0	1	0	0	0	1	0	0	0	1	0
1	1	0	0	0	0	0	0	0	0	1	0	0

$$O = S_1 S_0$$

FSM: eth preamble

Current State		Next State								Main output, O	Increment counter enable, C_I	Reset counter enable, C_R
		cnt /= 26 (C_E=0) x=0		cnt /= 26 (C_E=0) x=1		cnt = 26 (C_E=1) x=0		cnt = 26 (C_E=1) x=1				
S_1	S_0	Sn_1	Sn_0	Sn_1	Sn_0	Sn_1	Sn_0	Sn_1	Sn_0			
0	0	0	0	1	0	0	0	0	0	0	0	1
0	1	0	0	1	0	1	1	1	1	0	0	0
1	0	0	1	0	0	0	1	0	0	0	1	0
1	1	0	0	0	0	0	0	0	0	1	0	0

$$C_I = S_1\overline{S_0}$$

FSM: eth preamble

Current State		Next State								Main output, O	Increment counter enable, C_I	Reset counter enable, C_R
		cnt /= 26 (C_E=0) x=0		cnt /= 26 (C_E=0) x=1		cnt = 26 (C_E=1) x=0		cnt = 26 (C_E=1) x=1				
S_1	S_0	Sn_1	Sn_0	Sn_1	Sn_0	Sn_1	Sn_0	Sn_1	Sn_0			
0	0	0	0	1	0	0	0	0	0	0	0	1
0	1	0	0	1	0	1	1	1	1	0	0	0
1	0	0	1	0	0	0	1	0	0	0	1	0
1	1	0	0	0	0	0	0	0	0	1	0	0

$$C_R = \overline{S_1}\,\overline{S_0}$$

FSM: ethernet preamble

$$Sn_1 = \overline{(S_1 + C_E)}X + \overline{S_1}S_0(X + C_E)$$

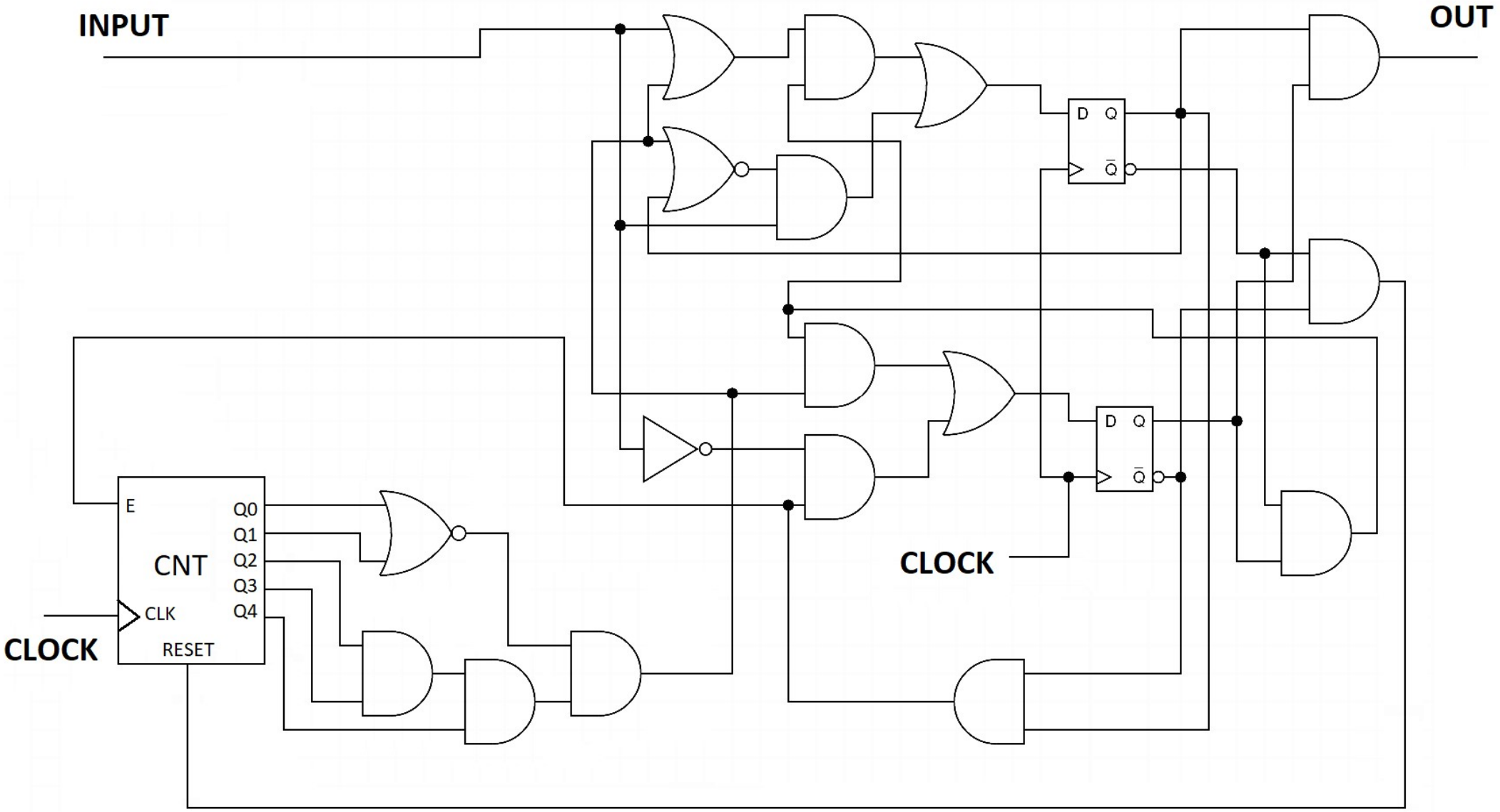
$$Sn_0 = \overline{S_1}S_0C_E + S_1\overline{S_0}\overline{X}$$

$$C_R = \overline{S_1}\,\overline{S_0}$$

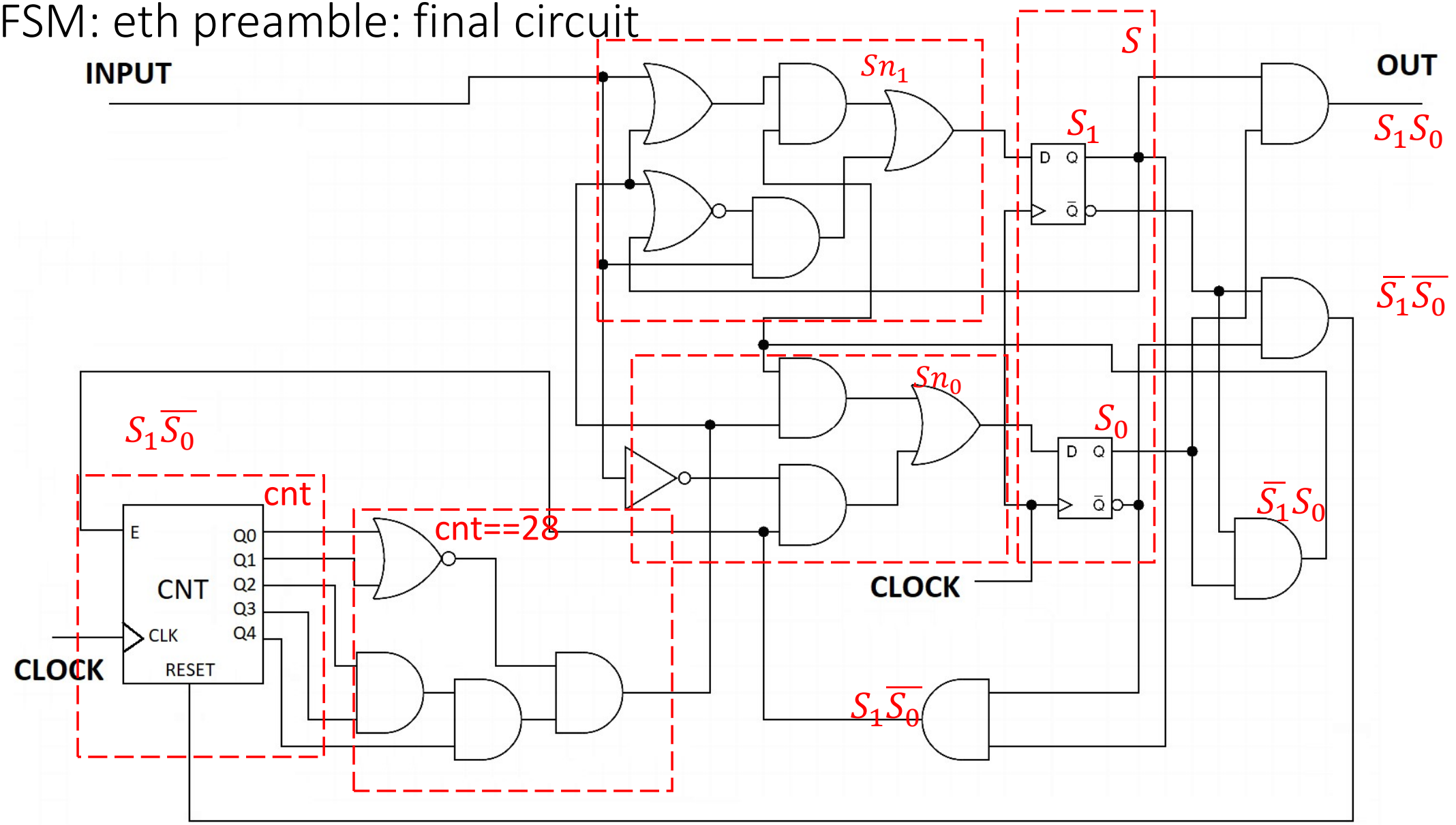
$$C_I = S_1\overline{S_0}$$

$$O = S_1S_0$$

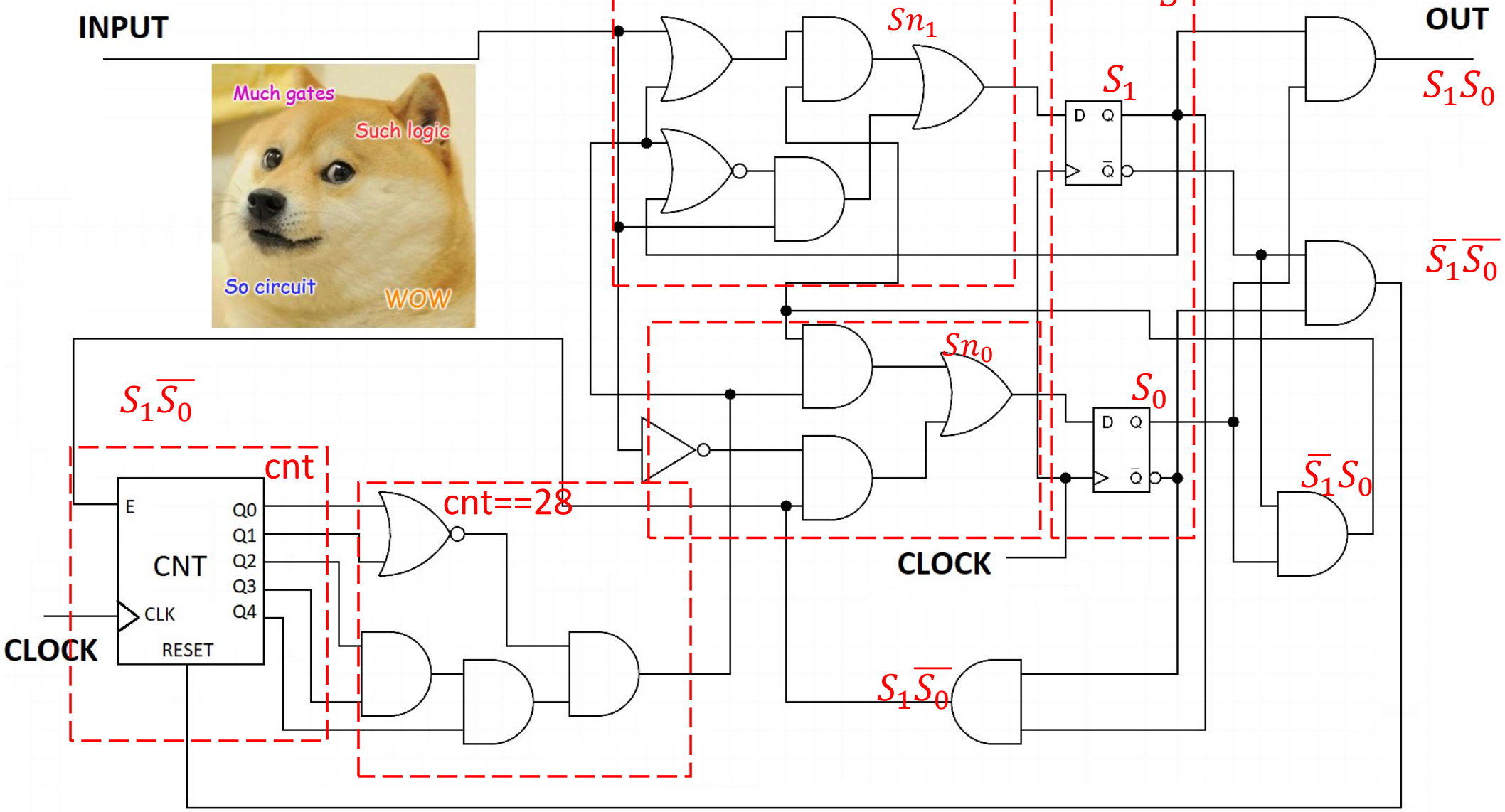
FSM: eth preamble: final circuit



FSM: eth preamble: final circuit



FSM: eth preamble: final circuit



And it works!

https://www.youtube.com/watch?v=Mn2j9D9xkUQ

Note: created with http://everycircuit.com/

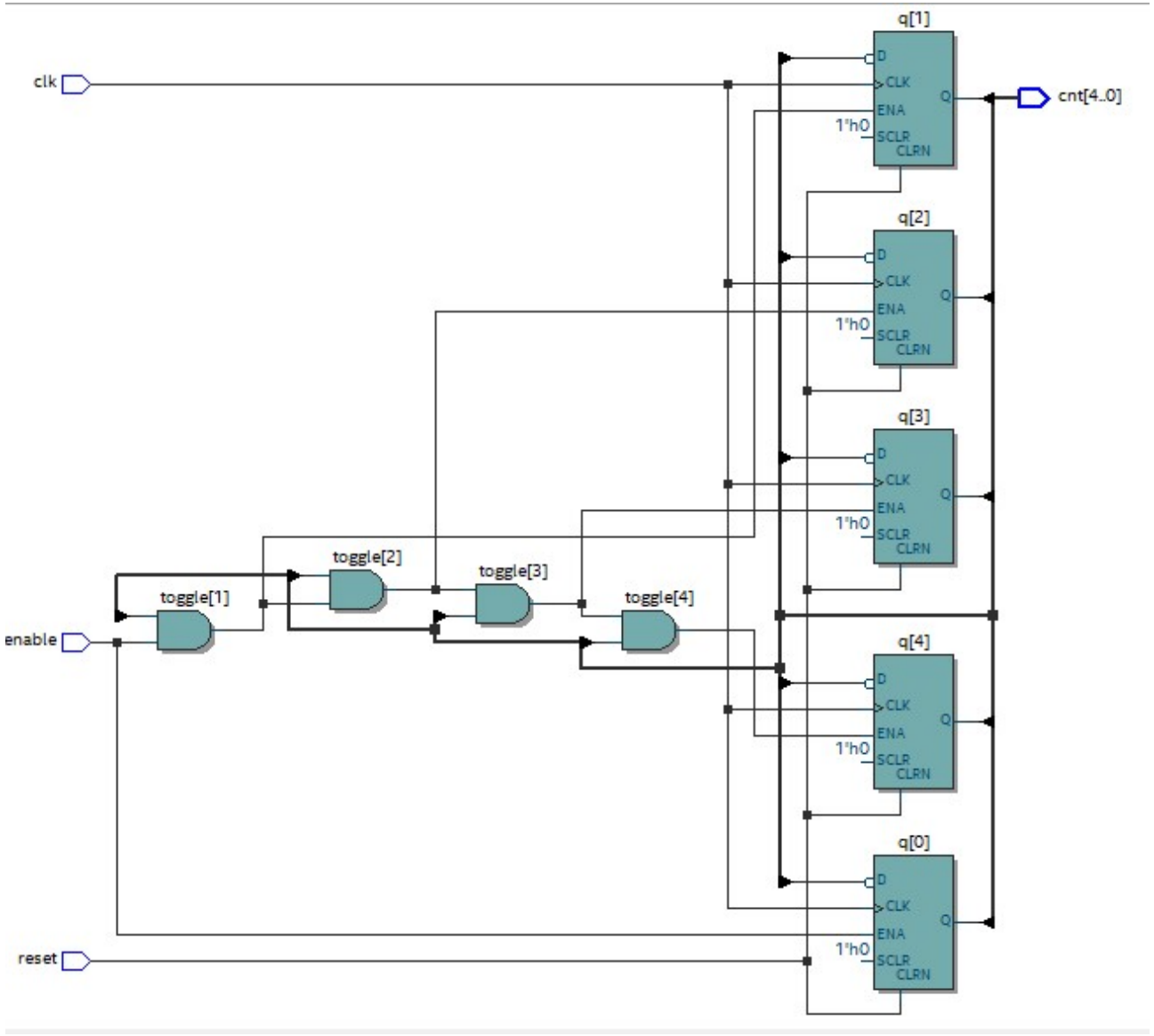
Let’s now implement it in VHDL

- We would implemented two separate versions
- First version would be the exact replica of what our manual circuit does – very low structural level
- Second version would be a more high level VHDL implementation
- After that we would run both in parallel to compare outputs

First version, counter module

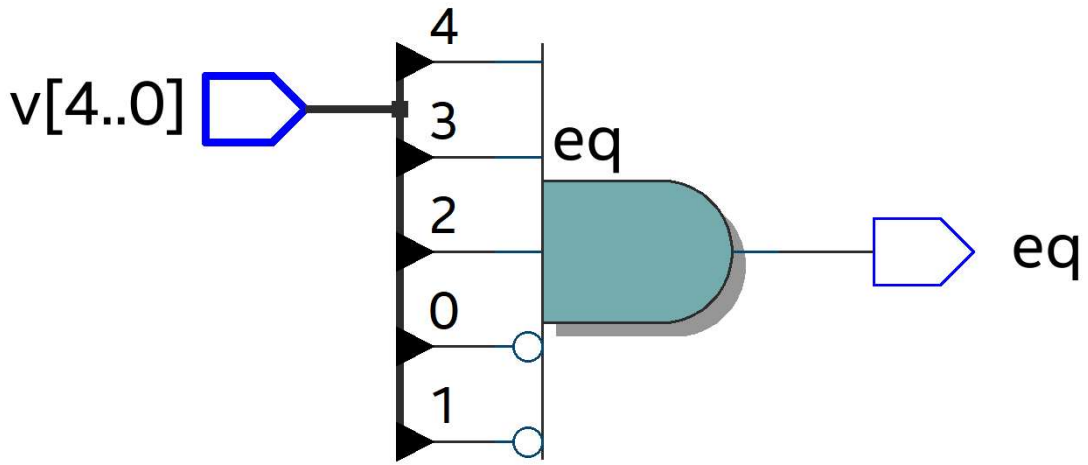
```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity counter is
    port (
        enable      : in std_logic;
        clk         : in std_logic;
        reset       : in std_logic;
        cnt         : out std_logic_vector(4 downto 0));
end counter;
architecture dataflow of counter is
    signal q        : std_logic_vector(4 downto 0) := "00000";
    signal toggle   : std_logic_vector(4 downto 0) := "00000";
begin
    toggle(0) <= enable;
    -- The "for-generate" statement would have the same effect as
    -- a sequence of individual assignments:
    -- toggle(1) <= toggle(0) and q(0);
    -- toggle(2) <= toggle(0) and q(0) and q(1);
    -- toggle(3) <= toggle(0) and q(0) and q(1) and q(2);
    -- toggle(4) <= toggle(0) and q(0) and q(1) and q(2) and q(3);
    gen_toggle:
    for i in 1 to 4 generate
        toggle(i) <= toggle(i-1) and q(i-1);
    end generate gen_toggle;
    gen_cnt: -- would normally have a blank line, but this is powerpoint!
    for i in 0 to 4 generate
        q(i) <= '0' when reset = '1' else
                (q(i) xor toggle(i)) when rising_edge(clk);
    end generate gen_cnt;
    cnt <= q;
end;
```

Ethernet preamble: first version, counter module



Ethernet preamble: first version, “== 28” module

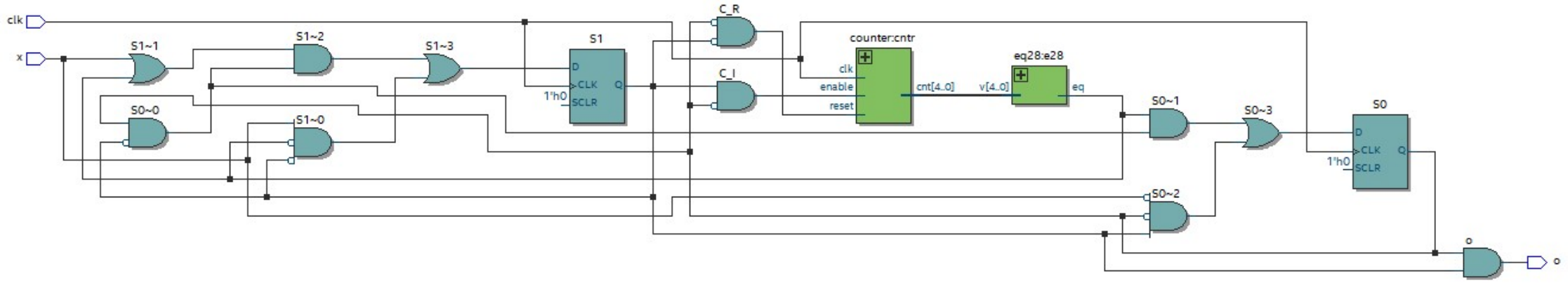
```
library ieee;
use ieee.std_logic_1164.all;
entity eq28 is
    port (
        v       : in std_logic_vector(4 downto 0);
        eq      : out std_logic
        );
end eq28;
architecture dataflow of eq28 is
begin
    eq <= (not v(0)) and (not v(1)) and v(2) and v(3) and v(4);
end;
```



Ethernet preamble: first version, main module

```
library ieee;
use ieee.std_logic_1164.all;
entity ethp is
    port (
        x, clk  :   in  std_logic;
        o       :   out std_logic);
end ethp;
architecture structural of ethp is
    component counter is
        port (
            enable      : in std_logic;
            clk         : in std_logic;
            reset       : in std_logic;
            cnt         : out std_logic_vector(4 downto 0));
    end component;
    component eq28 is
        port ( v : in std_logic_vector(4 downto 0); eq : out std_logic);
    end component;
    signal S1 : std_logic := '0';
    signal S0 : std_logic := '0';
    signal C_I, C_R, C_E : std_logic;
    signal cnt  : std_logic_vector (4 downto 0);
begin
    cntr: counter port map(enable => C_I, clk => clk, reset => C_R, cnt => cnt);
    e28: eq28 port map(v => cnt, eq => C_E);
    S1 <= (not S1 and not C_E and X) or (not S1 and S0 and (X or C_E)) when rising_edge(clk);
    S0 <= (not S1 and S0 and C_E) or (S1 and not S0 and not X) when rising_edge(clk);
    C_R <= not S1 and not S0;
    C_I <= S1 and not S0;
    O <= S1 and S0;
end structural;
```

Ethernet preamble: synthesis



Ethernet preamble: second version, high level, declarations

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity ethp_hl is
    port (
        x, clk  :   in  std_logic;
        o       :   out std_logic);
end ethp_hl;
architecture beh of ethp_hl is
    type machine_state is (IDLE, SEEN_1, SEEN_0, DETECTED_PATTERN);
    signal state    : machine_state;
    signal counter  : std_logic_vector(4 downto 0) := "00000";
begin
    process (clk, x)
    begin
        if rising_edge(clk) then
```

Ethernet preamble: second version, high level, process

```
begin
    process (clk, x)
    begin
        if rising_edge(clk) then
            case state is
                when IDLE =>
                    counter <= "00000";
                    if x = '1' then state <= SEEN_1; end if;
                when SEEN_1 =>
                    if x = '0' then
                        state <= SEEN_0;
                        counter <= counter + 1;
                    else
                        state <= IDLE;
                    end if;
                when SEEN_0 =>
                    if counter = 28 then
                        state <= DETECTED_PATTERN;
                    elsif x = '1' then
                        state <= SEEN_1;
                    else
                        state <= IDLE;
                    end if;
                when DETECTED_PATTERN =>
                    state <= IDLE;
            end case;
        end if;
    end process;
    o <= '1' when state = DETECTED_PATTERN else '0';
end beh;
```

EDIT:

There is a slight inconsistency between this high-level and low-level implementations here:

low-level implementation does unconditional counter increment, but here we only do it if x is equals to ‘0’.

In this particular case of FSM this logical difference doesn’t play a significant role, thus we can either modify this high level code to be closer to low-level by moving line outside the if statement or modify low level code to include requirement for x to be equal ‘0’:

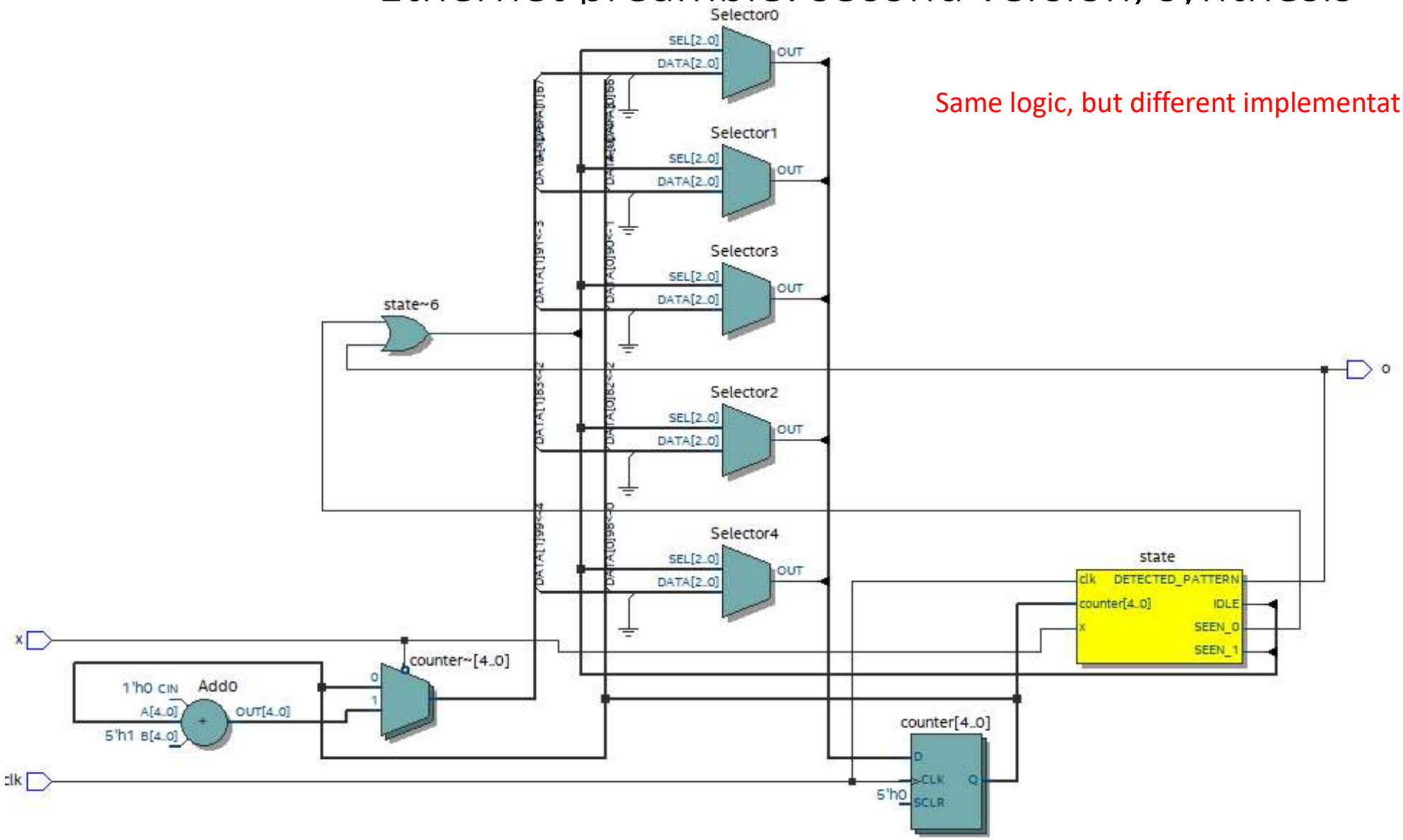
C_I <= S1 and not S0 and not x;

In all cases, this particular FSM produces the same result.

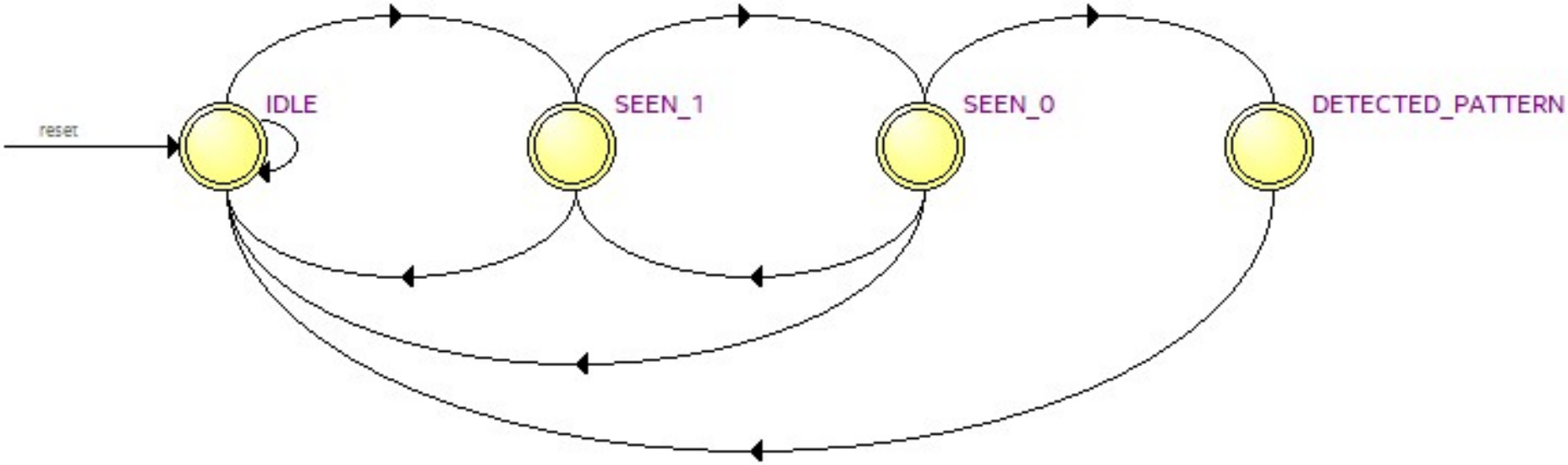
Note: since C_I is formally an output of FSM, this is now Mealy FSM.

Ethernet preamble: second version, synthesis

Same logic, but different implementation

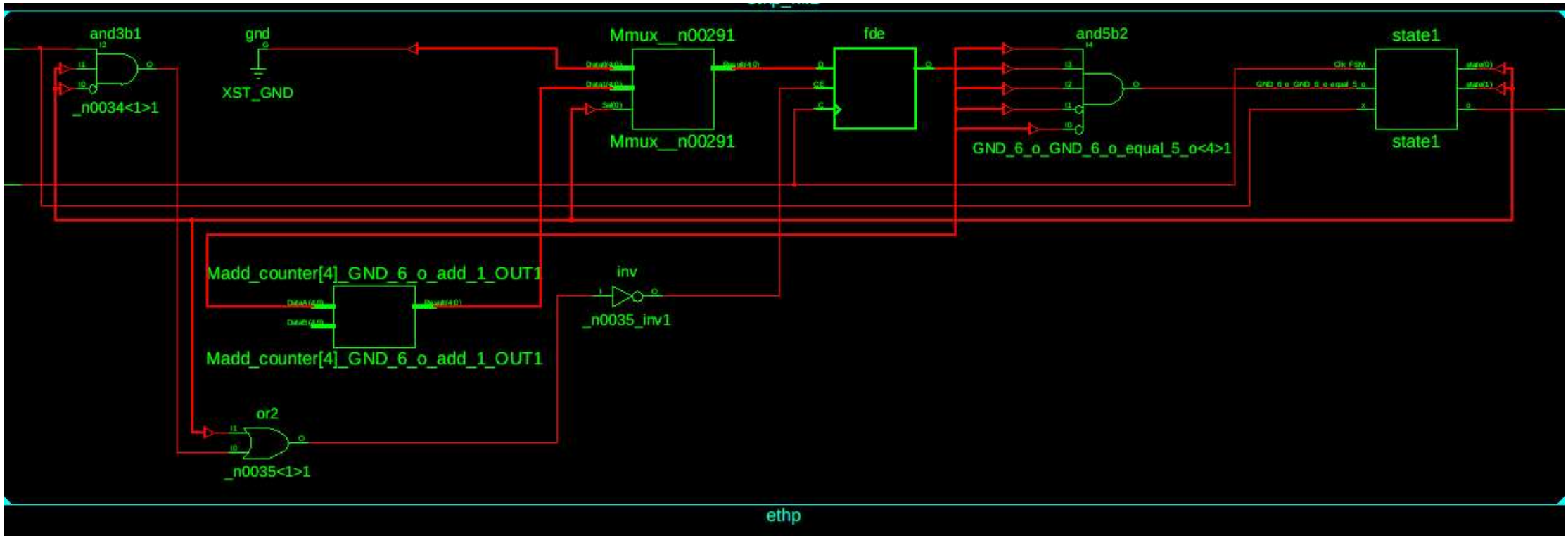


Ethernet preamble: second version, state machine



Ethernet preamble: second version, synthesis, Xilinx

Same logic, but different implementation



LEVELUP!!

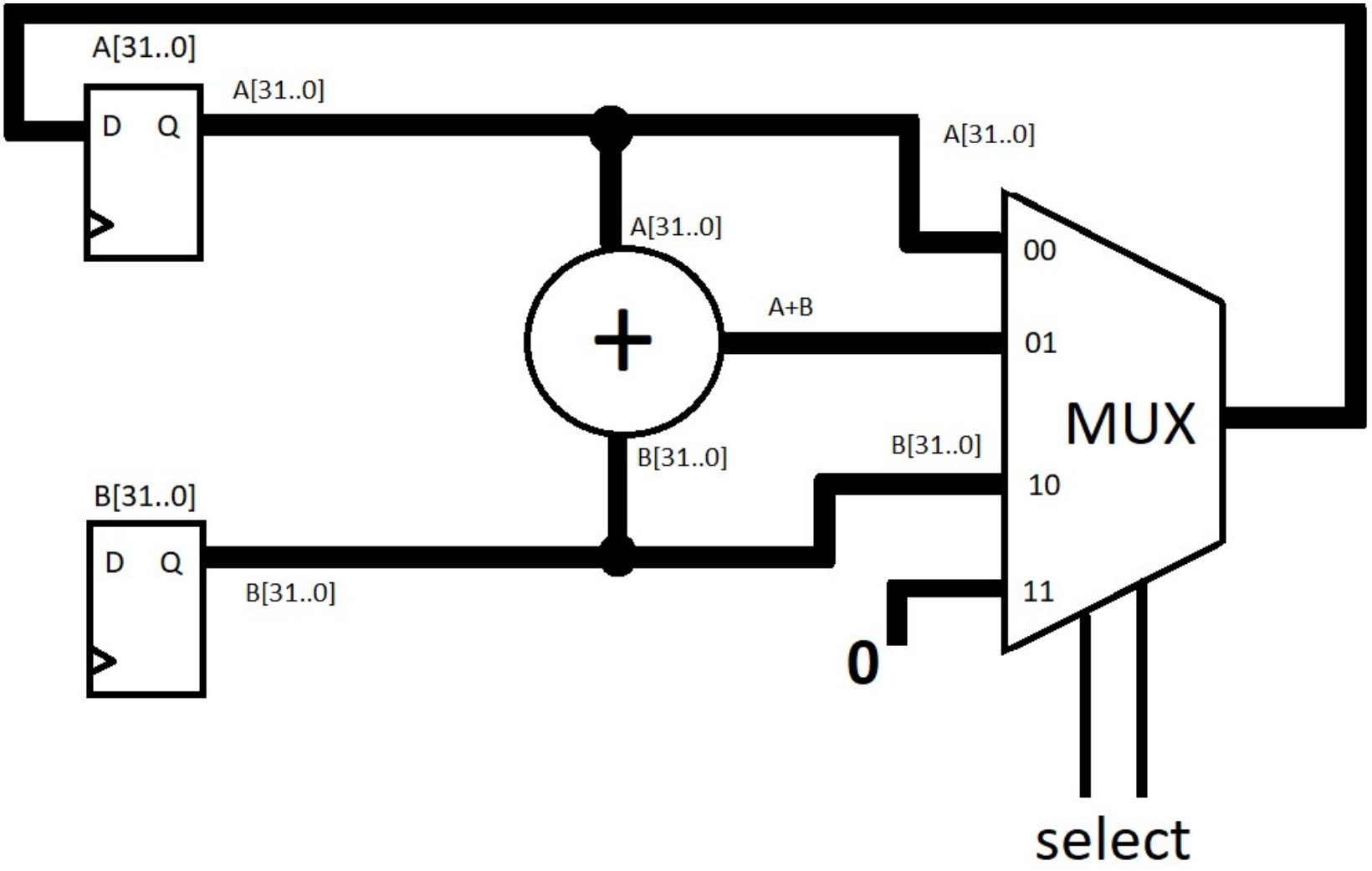
WE'VE UNLOCKED

VARIABLES

Variables – more advanced control example

```
if condition_1 then
    A <= A + B;  -- select <= "01"
elsif condition_2 then
    A <= B;      -- select <= "10"
elsif condition_3 then
    A <= 0;      -- select <= "11"
else
    -- no change -- select <= "00"
end if;
```

Variables – more advanced control example



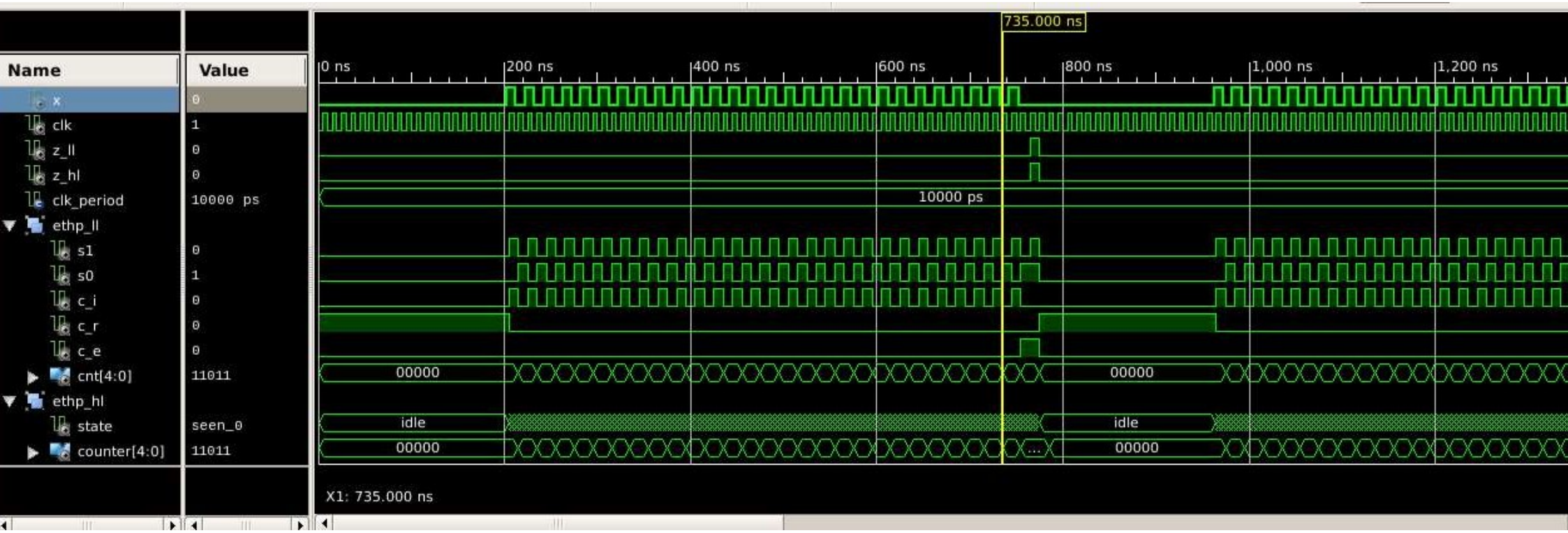
Testbench: declarations

```
library ieee;
use ieee.std_logic_1164.all;

entity TB is
end TB;

architecture behavior of TB is
    -- Component Declaration for the Unit Under Test (UUT)
   component ethp_hl is
        port (
            x, clk  :   in  std_logic;
            o       :   out std_logic);
    end component;
    component ethp is
        port (
            x, clk  :   in  std_logic;
            o       :   out std_logic);
    end component;
    --Inputs
    signal x : std_logic := '0';
    signal clk : std_logic := '0';
    signal z_ll : std_logic := '0';
    signal z_hl : std_logic := '0';

   -- Clock period definitions
   constant clk_period : time := 10 ns;
begin
```

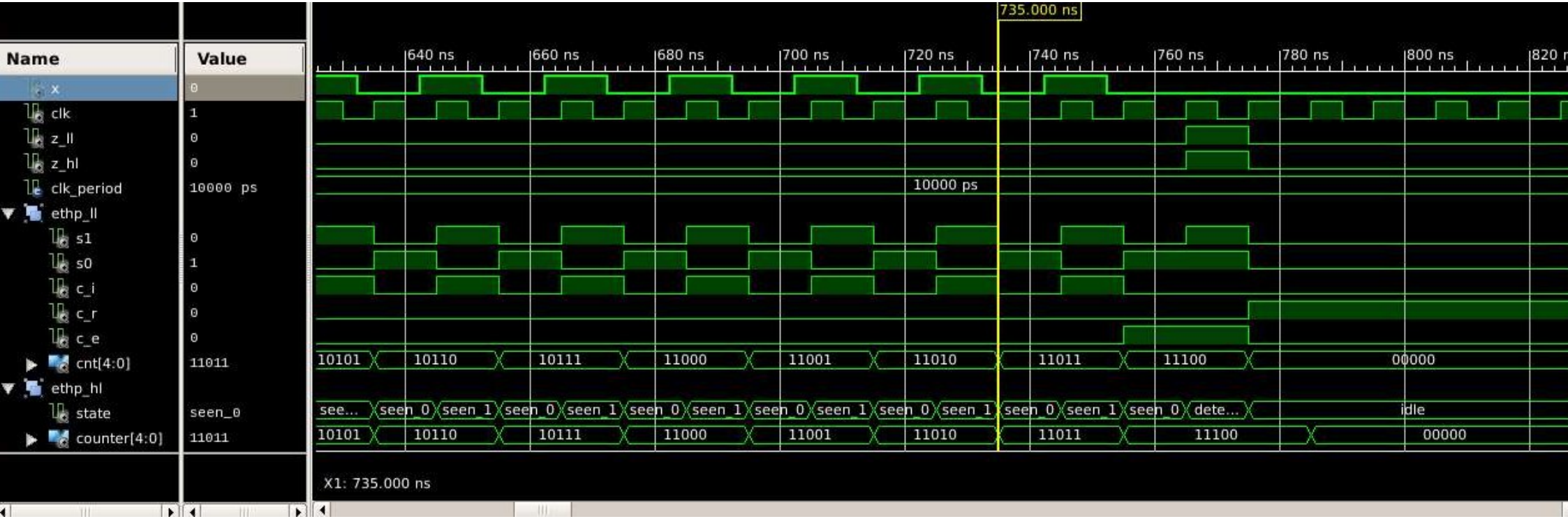
Testbench: code

```
begin
    -- Instantiate the Unit(s) Under Test (UUT)
    uut0: ethp_hl PORT MAP (x => x, clk => clk, o => z_hl); -- high level impl
    uut1: ethp PORT MAP (x => x, clk => clk, o => z_ll); -- low level impl

    clock_process: process -- clock generator process
    begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
    end process;

   stim_proc: process   -- Stimulus process
   begin
      -- hold reset state for 100 ns.
        wait for 100 ns;
        wait for clk_period*10; -- and a few more
        wait for clk_period/4;  -- offset from the clock edge
        for repeat in 1 to 2 loop
            for i in 1 to 28 loop
                x <= '1';
                wait for clk_period;
                x <= '0';
                wait for clk_period;
            end loop;
            for i in 1 to 20 loop
                x <= '0';
                wait for clk_period;
            end loop;
        end loop;
      wait;
   end process;
end;
```

Testbench: RUN



Testbench: RUN



Further reading and other materials

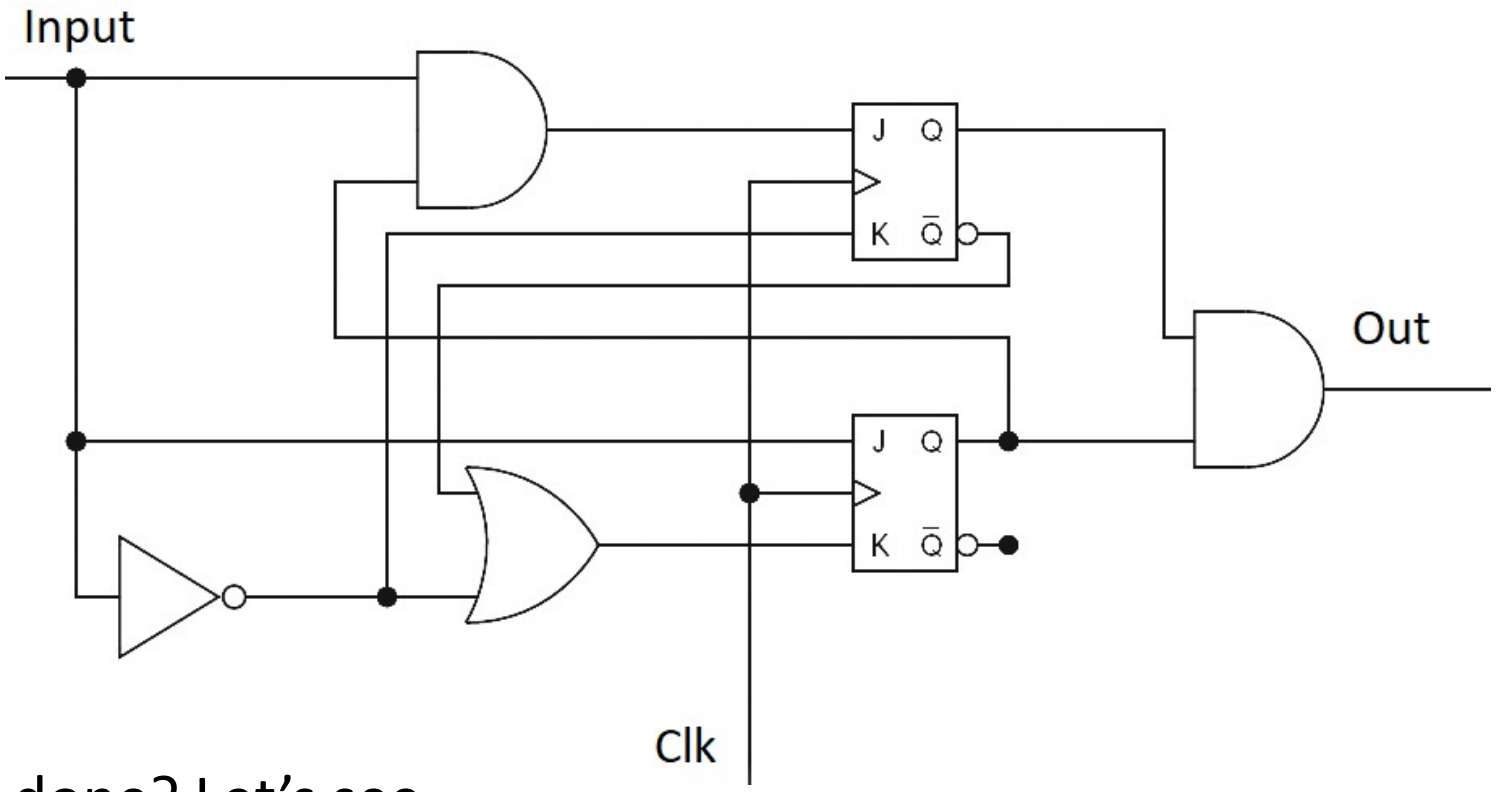
- *"VHDL 101"* by William Kafig, ISBN: 9780080959399 (available in Safari Books)
- *"Introduction to Digital Systems: Modeling, Synthesis, and Simulation Using VHDL"* by Mohammed Ferdjallah, ISBN: 9780470900550 (available in Safari Books)
- *"Digital Systems: From Logic Gates to Processors" by Universitat Autònoma de Barcelona* – free coursera course, https://www.coursera.org/learn/digital-systems/home/welcome
- *"Introduction to FPGA Design for Embedded Systems"* by University of Colorado Boulder -- free coursera course, mainly as a manual how to use Intel FPGA tools, https://www.coursera.org/learn/intro-fpga-design-embedded-systems

Appendix A.
JK-flip-flop implementation for ‘111’ sequence detecting Finite State Machine

NOTE: SKIP UNLESS EVERYONE IS EXCITED…

Alternative implementation with JK Flip Flop for ‘111’ sequence detection

- Different types of flip flops can be more or less efficient for particular purpose



- How is it done? Let’s see

NOTE: SKIP UNLESS EVERYONE IS EXCITED….

Alternative implementation with JK Flip Flop

- Same as before: do the state transition table, encode states as bits
- After finding out how bits should change in each particular transition, we must calculate flip-flop inputs to toggle correct flip flop value toggle.
  - E.g. gate network must produce correct values for J and K inputs, rather just ‘D’ inputs
- The same can be done with T Flip Flops.

Further reading on FSM: ch.9 of *“Introduction to Digital Systems: Modeling, Synthesis, and Simulation Using VHDL”* by Mohammed Ferdjallah, ISBN: 9780470900550, available in Safari Books

NOTE: SKIP UNLESS EVERYONE IS EXCITED….

Alternative implementation with JK Flip Flop

- Let's examine all the possible bit transitions and find out what J and K values has to be applied, again "-" means "do not care":

From	To	J	K	Comment
0	0	0	-	If K=0, FF state stays the same. If K=1, FF is again set to 0, which in fact is not changing its state
0	1	1	-	If K=0, FF is set to '1' by J=1, if K=1 FF is toggled by J=K=1, result is again 1
1	0	-	1	If J=0, FF state is set to 0 by K=1, if J=1, FF state is toggled by J=K=1, result is 0
1	1	-	0	If J=0, FF state didn't change. If J=1, FF state is set to 1, that was already the value of the FF

Further reading on FSM: ch.9 of *"Introduction to Digital Systems: Modeling, Synthesis, and Simulation Using VHDL"* by Mohammed Ferdjallah, ISBN: 9780470900550, available in Safari Books

NOTE: SKIP UNLESS EVERYONE IS EXCITED….

Alternative implementation with JK Flip Flop

From	To	J	K
0	0	0	-
0	1	1	-
1	0	-	1
1	1	-	0

- Taking the bit transition / J/K values table, lets update our original state transition table with values for J and K:

S[1]	S[0]	I == 0, next val						I == 1, next val						O
		Sn[1]	J[1]	K[1]	Sn[0]	J[0]	K[0]	Sn[1]	J[1]	K[1]	Sn[0]	J[0]	K[0]	
0	0	0	0	-	0	0	-	0	0	-	1	1	-	0
0	1	0	0	-	0	-	1	1	1	-	0	-	1	0
1	0	0	-	1	0	0	-	1	-	0	1	1	-	0
1	1	0	-	1	0	-	1	1	-	0	1	-	0	1

- The beauty of “-” aka “do not care” cells is that we can pick any value that simplifies our circuit / formula the best way.
- This is best to be done using Karnaugh map: https://en.wikipedia.org/wiki/Karnaugh_map

NOTE: SKIP UNLESS EVERYONE IS EXCITED….

Alternative implementation with JK Flip Flop

- Finally, Karnaugh maps…

S[1]	S[0]	I == 0, next val						I == 1, next val						O
		Sn[1]	J[1]	K[1]	Sn[0]	J[0]	K[0]	Sn[1]	J[1]	K[1]	Sn[0]	J[0]	K[0]	
0	0	0	0	-	0	0	-	0	0	-	1	1	-	0
0	1	0	0	-	0	-	1	1	1	-	0	-	1	0
1	0	0	-	1	0	0	-	1	-	0	1	1	-	0
1	1	0	-	1	0	-	1	1	-	0	1	-	0	1

$$J_1 = S_0 I$$

I \ S_1, S_0	00	01	11	10
0	0	0	-	-
1	0	1	-	-

NOTE: SKIP UNLESS EVERYONE IS EXCITED….

Alternative implementation with JK Flip Flop

- Finally, Karnaugh maps…

S[1]	S[0]	I == 0, next val						I == 1, next val						O
		Sn[1]	J[1]	K[1]	Sn[0]	J[0]	K[0]	Sn[1]	J[1]	K[1]	Sn[0]	J[0]	K[0]	
0	0	0	0	-	0	0	-	0	0	-	1	1	-	0
0	1	0	0	-	0	-	1	1	1	-	0	-	1	0
1	0	0	-	1	0	0	-	1	-	0	1	1	-	0
1	1	0	-	1	0	-	1	1	-	0	1	-	0	1

$$J_0 = I$$

I \ S_1, S_0	00	01	11	10
0	0	-	-	0
1	1	-	-	1

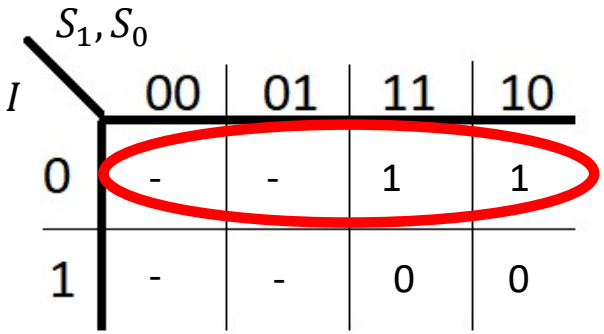
NOTE: SKIP UNLESS EVERYONE IS EXCITED….

Alternative implementation with JK Flip Flop

- Finally, Karnaugh maps…

S[1]	S[0]	I == 0, next val						I == 1, next val						O
		Sn[1]	J[1]	K[1]	Sn[0]	J[0]	K[0]	Sn[1]	J[1]	K[1]	Sn[0]	J[0]	K[0]	
0	0	0	0	-	0	0	-	0	0	-	1	1	-	0
0	1	0	0	-	0	-	1	1	1	-	0	-	1	0
1	0	0	-	1	0	0	-	1	-	0	1	1	-	0
1	1	0	-	1	0	-	1	1	-	0	1	-	0	1

$$K_1 = \bar{I}$$



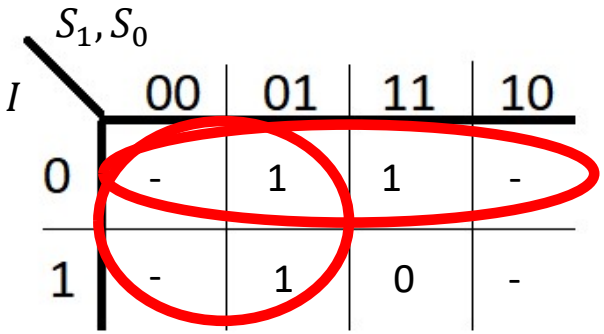
NOTE: SKIP UNLESS EVERYONE IS EXCITED….

Alternative implementation with JK Flip Flop

- Finally, Karnaugh maps…

S[1]	S[0]	I == 0, next val						I == 1, next val						O
		Sn[1]	J[1]	K[1]	Sn[0]	J[0]	K[0]	Sn[1]	J[1]	K[1]	Sn[0]	J[0]	K[0]	
0	0	0	0	-	0	0	-	0	0	-	1	1	-	0
0	1	0	0	-	0	-	1	1	1	-	0	-	1	0
1	0	0	-	1	0	0	-	1	-	0	1	1	-	0
1	1	0	-	1	0	-	1	1	-	0	1	-	0	1

$$K_0 = \bar{I} + \bar{S_1}$$

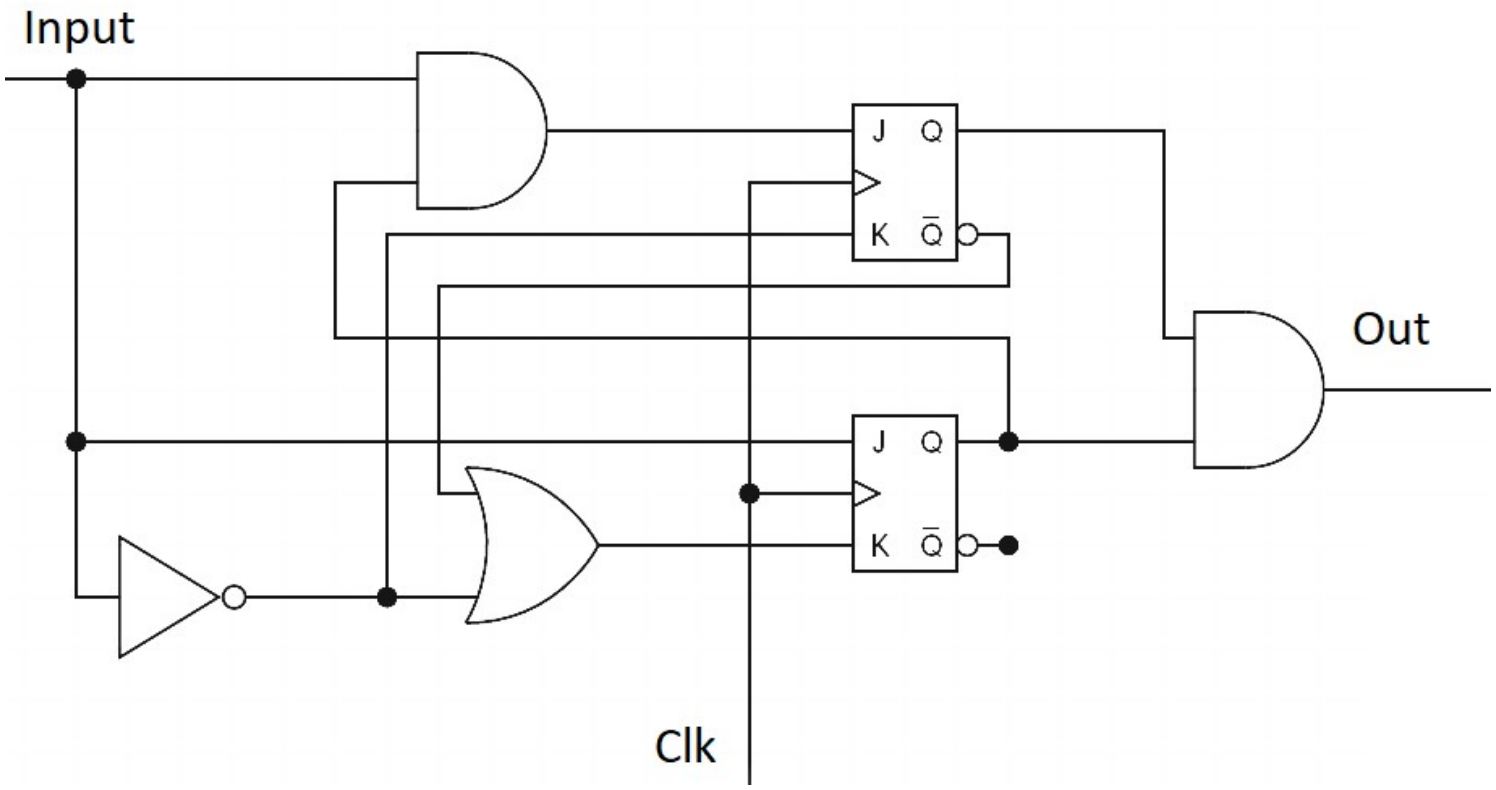


NOTE: SKIP UNLESS EVERYONE IS EXCITED….

Alternative implementation with JK Flip Flop

$$J_1 = S_0 I \qquad J_0 = I$$

$$K_1 = \bar{I} \qquad K_0 = \bar{I} + \overline{S_1}$$



Appendix B. Tools and IDEs

- Everycircuit - http://everycircuit.com/app/
- Xilinx ISE (legacy) - https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/design-tools.html
- Intel Quartus Prime Lite Edition - http://fpgasoftware.intel.com/18.1/?edition=lite