Design of Digital Circuits

Frühjahrssemester 2019 – Lecture Notes

Sven Pfiffner

Davos|Zürich April 19

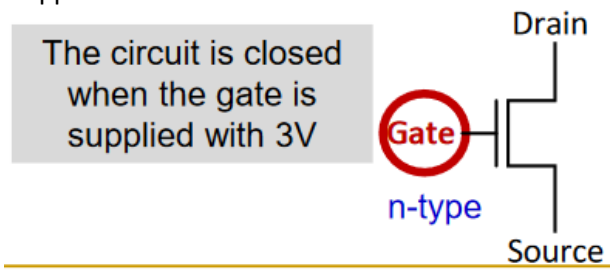
Content

Combinational Logic

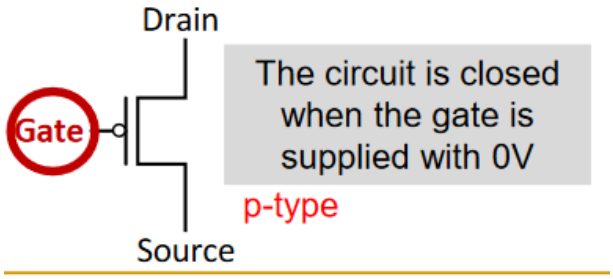
Computers are built from very large numbers of very simple structures. Probably the most important ones are the so called MOS-Transistors (**M**etal, **O**xide, **S**emiconductors). We can combine MOS-Transistors to realize simple logic gates, which we'll then use to implement logic to our machine.

Essentially, there are two types of MOS-Transistors:

- The n-type MOS opens when supplied with a high voltage. The connection from source to drain acts like a piece of wire when the gate is supplied with 0.3V to 3V (depending on the technology) and closes when supplied with 0V



- The p-type MOS opens when supplied with a low voltage. The connection from source to drain acts like a piece of wire when the gate is supplied with 0V and closes when supplied with 0.3V to 3V (depending on the technology) and closes when supplied with 0V

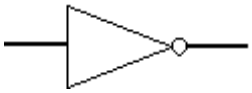


Modern Computers use both n-type and p-type transistors, i.e. Complementary MOS (CMOS) technology ⇒ nMOS + pMOS = CMOS

But how does this help us to realize some logic in our machine? The answer is relatively simple: We construct basic logic structures, so called logic-gates, out of individual MOS transistors. We then can go one level higher in the abstraction and use those gates as fundamental building blocks to implement the boolean logic and arithmetic that we want. Let's take a deeper look at commonly used gates in modern computers

NOT

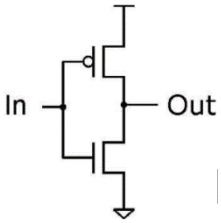
Symbol



Logic Table

Input 1	Output
0	1
1	0

Transistor Level



AND

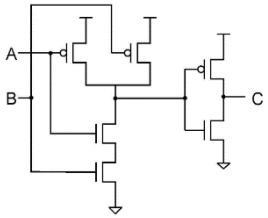
The AND-Gate acts in the same way as the logical ∧ operator.

Symbol

Logic Table

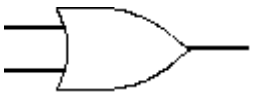
Input 1	Input 2	Output
0	0	0
0	1	0
1	1	1
1	0	0

Transistor Level



OR

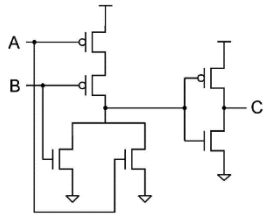
Symbol



Logic Table

Input 1	Input 2	Output
0	0	0
0	1	1
1	1	1
1	0	1

Transistor Level



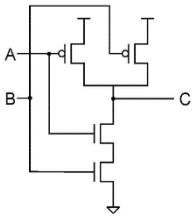
NAND

Symbol

Logic Table

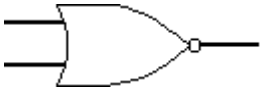
Input 1	Input 2	Output
0	0	1
0	1	1
1	1	0
1	0	1

Transistor Level



NOR

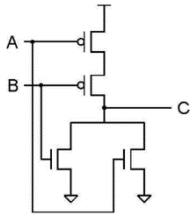
Symbol



Logic Table

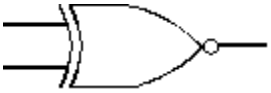
Input 1	Input 2	Output
0	0	1
0	1	0
1	1	0
1	0	0

Transistor Level



XNOR

Symbol

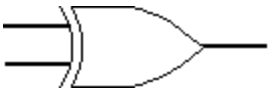


Logic Table

Input 1	Input 2	Output
0	0	1
0	1	0
1	1	1
1	0	0

XOR

Symbol

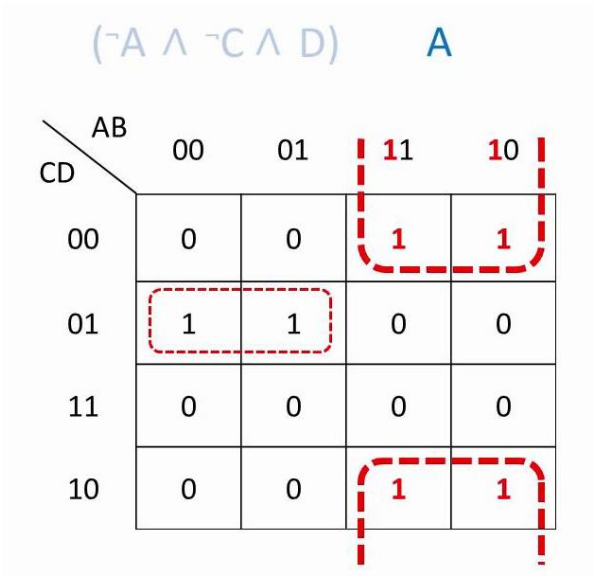


Logic Table

Input 1	Input 2	Output
0	0	0
0	1	1
1	1	0
1	0	1

Karnaugh Maps

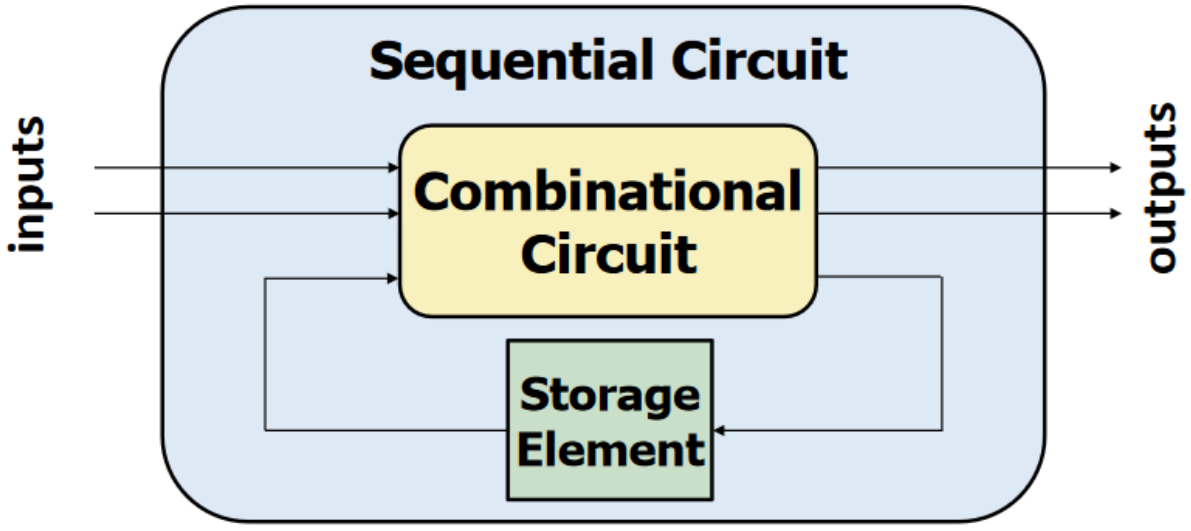
The Karnaugh map is a method of simplifying Boolean algebra expressions. This can help us to design efficient circuits with as few gates as possible. The required Boolean results are transferred from a truth table onto a two-dimensional grid where, in Karnaugh maps, the cells are ordered in Gray code, and each cell position represents one combination of input conditions, while each cell value represents the corresponding output value. Optimal groups of 1s or 0s are identified, which represents the terms of a canonical form of the logic in the original truth table. These terms can be used to write a minimal Boolean expression representing the required logic.



Example of usage of a Karnaugh Map

Sequential Logic Design

The circuits that we can build with the gates that we have seen so far depend only on current input. But what if we want circuits that produce output depending on current and past input values – circuits with memory? How can we design a circuit that stores information?

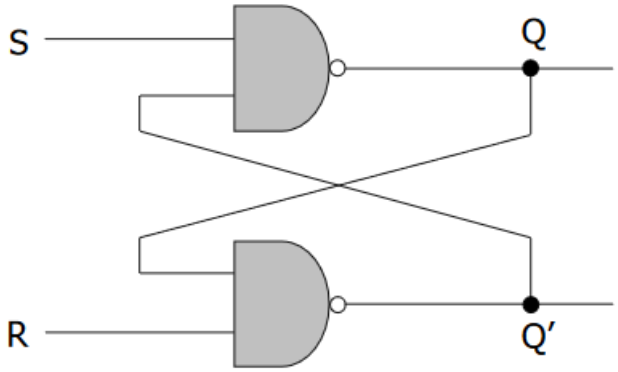


Structure of a sequential circuit

S-R Latch

A S-R Latch consists of two cross-coupled NAND gates and has two stable states. Typically, one state is referred to as set and the other as reset, hence the name S-R Latch.

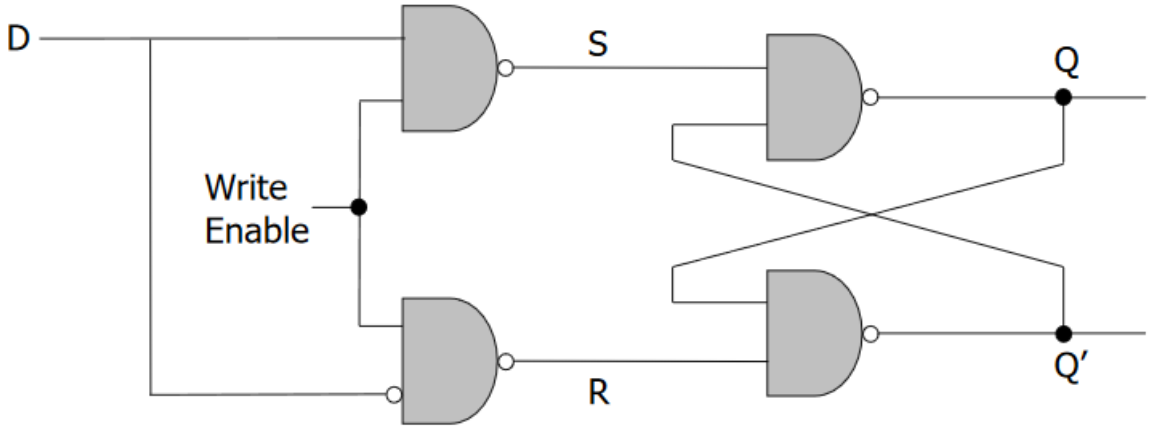
To create an S-R latch, we can wire two NAND gates in such a way that the output of one feeds back to the input of another, and vice versa, like this:



Note that S and R should never both be 0 at the same time since Q and Q' will both settle to 1, which breaks our invariant that $Q = !\,Q'$

Gated D Latch

We can easily modify the S-R Latch to guarantee correct operation (prevent that the latch enters illegal state by receiving 0 at both S and R) by adding two more NAND gates to the circuit.

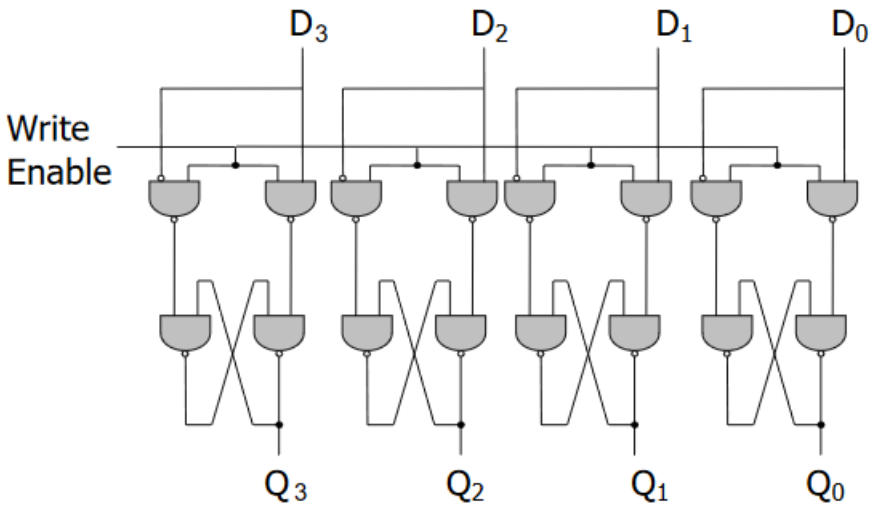


As one sees, Q takes the value of D, when write enable is set to 1 and most importantly, S and R can never be 0 at the same time! This concept is called a Gated D Latch (because a gate keeps us safe from the illegal state)

Register

We have seen how to store a single bit using gated D Latches. One thing that one might be wondering now is how to store more data. The answer is quite simple. If we want to store one bit, we use a single D Latch; if we want to store multiple bits, we use multiple D Latches.

Since we normally want to write to all bits of the data at once, a single WE (write enable) signal for all latches should be enough and allows simultaneous writes.



Here we have a register, or a structure that stores more than one bit and can be read from and written to. This register holds 4 bits (one per latch), and its data is referenced as $Q[3:0]$.

Memory

Memory is comprised of locations that can be written to or read from. Every unique location in memory is indexed with a unique address. The number of bits of information stored in each location is called the addressability and the set of unique locations is referred to as the address space of the memory

Addr(00):	0100 1001	**Addr**(01):	0100 1011
Addr(10):	0010 0010	**Addr**(11):	1100 1001

Example memory array with 4 locations and addressability 8

State

The state of a system is a snapshot of all relevant elements of the system at the moment of the snap-shot.

Example: Lets think of a standard traffic light which has 4 states

1. Light is Green
2. Light is Yellow
3. Light is Red
4. Light is both Red and Yellow

The sequence of these states is always as follows: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 1 \rightarrow \cdots$

The question that one may ask is when the lights should change from one state to another, or more generally, who or what controls when a state of any given system should change. In order to time such state changes, we use so called clocks.

A clock is a general mechanism that triggers transition from one state to another in a sequential circuit. It synchronizes state changes across many sequential circuit elements by generating a so-called clock signal that alternates between 0 and 1. At the start of a clock cycle (the so called rising-edge), the system state changes. In the traffic light example, we are assuming the traffic light stays in each state an equal amount of time, but generally we can use logic to evaluate for rising-edges and realize certain delays.

A typical clock signal (10 rising-edges can be seen)

Finite State Machines

A finite state machine (FSM) is a mathematical model of computation. It is an abstract machine that can be in exactly one of a finite number of states at any given time. The FSM can change from one state to another in response to some external inputs. An FSM is defined by a list of its states, its initial state, and the conditions for each transition.

Finite State Machines enable us to pictorially think of a stateful system using simple diagrams

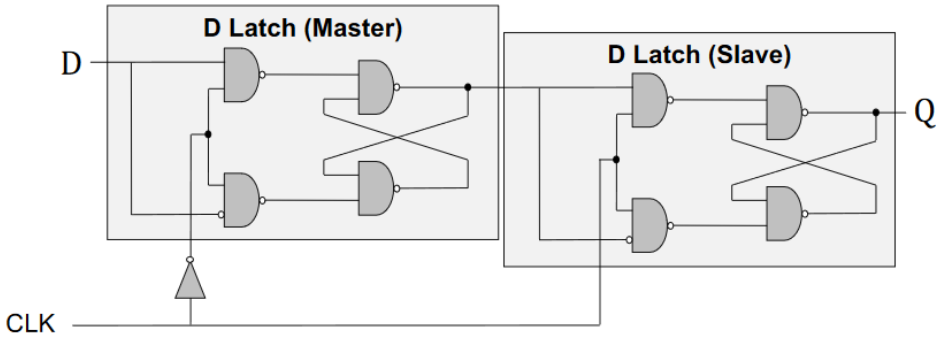
In order to implement a finite state machine, we need:

- **Sequential Circuits:** So-called state registers store the current state and load the next state at the clock edge
- **Combinational Circuits:** Next state logic determines what the next state will be
- **Output logic:** Generates the desired output

D Flip-Flop

When implementing our own FSM – Sequential Circuits we may think of using Gated D Latches. But the problem that arises is that we cannot simply wire a clock to the Write-Enabled signal of a latch. Whenever the clock is high, the latch enables write from input to output. What we want however, is to only allow write to output at the rising edge of the clock while reading from output should be available for the full clock cycle.

Fortunately, there is a circuit that allows such operations, and it's called the D Flip-Flop.



D Flip-Flop Circuit

Basically, the D Flip-Flop consists of two D Latches, where one serves as the so called "master" and the other as "Slave"

- **Master:** We can only write to master as long as the clock signal is low, so at the point of "clock-rising" the last input before that is stored to its output. The stored value will not change until the clock signal becomes low again.
- **Slave:** The slave is writable during high clock signals. Since its input is the master's output, it will store whatever value was fed into the master at the clocks rising-edge. The stored value will not change until the clock signal becomes high again.

Hardware description Languages

A hardware description language (HDL) is a specialized computer language used to describe the structure and behavior of electronic circuit. A hardware description language enables a precise, formal description that allows for the automated analysis and simulation of an electronic circuit. It also allows for the synthesis of an HDL description into a netlist, which can then be used to create an integrated circuit

Timing and Verification

Tradeoffs in Circuit Design

Of course, it is not possible to create fast, reliable, small, efficient and cheap circuits all at the same time. We always have to make trade-offs between certain things when designing circuits. The most important things that one must consider are

- Area: Generally, we want to make our circuits as small as possible, and what's even more important: use as few modules as possible. The area of a circuit is proportional to the cost of the device.
- Speed/Throughput: More speed and throughput mean that our circuit can achieve more in the same time. Therefore, we want faster, more capable circuits whenever possible
- Power/Engergy: It is crucial to think about this aspect in advance, because it's importance varies based on the target devices of our circuits. While high performance devices dissipate more than $100\frac{W}{cm^2}$ mobile devices need to work with a limited power supply.
- Design Time: Designers are expensive in time and money and the competition will not wait. It is important to consider the time one needs to design circuits.

Timing

Until now, we only investigated logical functionality of circuits. But what about timing? How can we determine how fast a circuit is? How can we make a circuit faster and what happens if we run a circuit too fast?

Combinational Circuit Timing

Up until now we have only dealt with the general functionality of our circuits. We have automatically used a convenient abstraction and relied on the output changing immediately with the input. In reality however, outputs are delayed from inputs because CMOS transistors take a finite amount of time to switch. Delay is fundamentally caused by capacitance and resistance in a circuit and the finite speed of light and there are many things that can increase these delays. Some of them are:

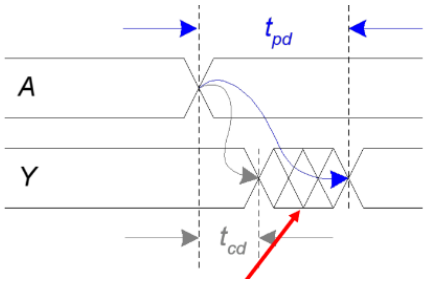
- Aging of the circuit
- Changes in environment (temperature, humidity…)
- Different types of inputs
- …

We have a range of possible delays from input to output

Contamination and propagation delay

The contamination delay (t_{cd}) is the minimum amount of time from when an input changes until any outputs starts to change its value.

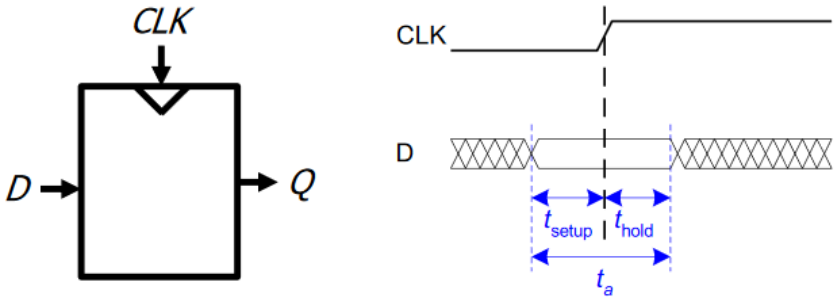
The propagation delay (t_{pd}) is the length of time which starts when the input to a logic gate becomes stable and valid to change, to the time that the output of that logic gate is stable and valid to change.



Sequential Circuit Timing

Like combinational circuits, when sequential circuits, such as edge-triggered flip-flops, are physically implemented, they exhibit certain timing characteristics. Unlike combinational circuits, these characteristics are specified in relation to the clock input. Since flip-flops only change value in response to a change in the clock value, timing parameters can be specified in relation to the rising or falling clock edge. The following parameters specify sequential circuit behavior. Unless otherwise specified, the following descriptions pertain to positive edge-triggered circuits. Similar definitions can be made for negative edge-triggered circuits.

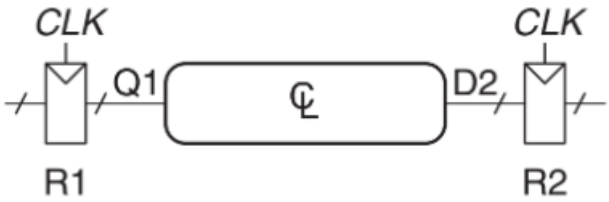
- Setup time (t_{setup}): This value indicates the amount of time before the clock edge that data input must be stable.
- Hold time (t_h): This value indicates the amount of time after the clock edge that data input must be held stable.
- Aperture time (t_a): This value indicates the amount of time around the clock edge that data input must be held stable



Setup, hold and aperture time of a clock

Ensuring correct sequential operation

A circuit must be designed so that the D Flip-Flop input signal arrives at least t_s time units before the clock edge and does not change until at least t_h time units after the clock edge. If either of these restrictions are violated for any of the Flip-Flops in the circuit, the circuit will not operate correctly. These restrictions limit the maximum clock frequency at which the circuit can operate. If the rising clock edges are too close together, data will not have enough time to propagate through the circuits to the Flip-Flop input and arrive t_s time units before the rising clock edge. Let's take a look at the following circuit:



In order to guarantee correct sequential operation, we must ensure correct input timing on $R2$. Specifically, $D2$ must be stable at least t_{setup} before the clock edge and at least until t_{hold} after the clock edge. This means that there is both a minimum and maximum delay between two Flip-Flops. If the clock signal is too fast, then $R2$ violates t_{hold}; if it is too slow, then $R2$ violates t_{setup}.

Circuit verification

Von Neumann Model

Instruction Set Architecture

LC-3

MIPS

Assembly

Microarchitecture

Pipelining

Reorder Buffer

Out-of-Order Execution