Introduction to Sequential Logic

Antoine Lavault[1][2]

[1]Apeira Technologies

[2]UMR CNRS 9912 STMS, IRCAM, Sorbonne Université

January 19, 2024

Contents

1 Introductory Problem
- A lamp. And a button.
- Building a memory

2 Sequential Logic Basic Circuits

3 State Machines

4 Asynchronous behavior with a synchronous system

5 Memory
- Shift Registers

Contents

1 Introductory Problem
- A lamp. And a button.
- Building a memory

2 Sequential Logic Basic Circuits

3 State Machines

4 Asynchronous behavior with a synchronous system

5 Memory

Outline

1 Introductory Problem
- A lamp. And a button.
- Building a memory

Introductory example

Yesterday, we did things. That's good.
Today, we will do other things. Like lighting up a lamp

Introductory example

Yesterday, we did things. That's good.
Today, we will do other things. Like lighting up a lamp

A lamp and its push button

Let's build a system such that when a button is pushed :

- The lamp turns on if it is off
- The lamp turns off if it is on

Good luck in building this with combinational logic.
But why?

A lamp and its push button

The system's input is the button, and the output is the lamp. So far, so good. But the biggest difference is that the device's output is not a function of the device's current input value.
The behavior when the button is pushed depends on what has happened in the past: it remembers. Somehow.
Devices that remember something about the history of their inputs are said to have state.

A lamp and its push button. Continued.

There is a second difference. But it is more subtle.
The push of the button marks an event in time: we speak of the state before the push (e.g., "the light is on") and state after the push (in this case, "the light is off").
It's the transition of the button from un-pushed to pushed that we're interested in, not whether the button is currently pushed.
Some sort of rising or falling edge...

Really? Still a lamp and a button...

The device's internal state allows it to produce different outputs despite receiving the same input. A combinational device can't exhibit this behavior since its outputs depend only on the input's current values. If we were to use combinational logic here, the light would be on when the button is pressed, but would instantly turn off when the button is released (unless it is a latch button) Let's see how we'll incorporate the notion of device state into our circuitry.

Outline

1 Introductory Problem
- A lamp. And a button.
- Building a memory

Sequential circuit

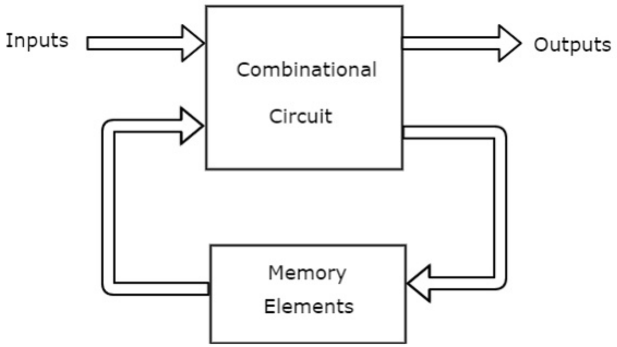

Figure: Sequential circuit

There is feedback. And states.

Memories.

We'll introduce a new abstraction of a memory component that will store the current state of the digital system we want to build.
The memory component stores one or more bits that encode the current state of the system. These bits are available as digital values on the memory component's outputs and are wired to the inputs of the combinational circuit.
The current state and the current input values are the inputs to a block of combinational logic that produces two sets of outputs. One set of outputs is the next state of the device, encoded using the same number of bits as the current state.
The other set of outputs are the signals that serve as the outputs of the digital system. The functional specification for the combinational logic (truth table, boolean equations) specifies how the next state and system outputs are related to the current state and current inputs.
How to build a memory?

A crude memory: the capacitor

If we encode information using voltage, capacitors can "hold" the information.
Well, bits are voltages, so using a capacitor to hold the voltage makes sense.
On the other hand, we would need something to act as a "switch" to charge/discharge the capacitor...
While this kind of memory exists, it is slow, clunky, and requires a lot of interfacing...
Also, it loses charge because of the imperfections of the FET...
Maybe feedback could help keep the charge, at least?

Two NOT gates back to back are very stable.

Two NOT gates back to back are a memory. When there is (positive) feedback, at least.
Two stable states and one metastable (if we consider the voltage characteristics, this is where *wereVIN* $= VOUT$).
Great! We can now store things. But how do we put something in it?

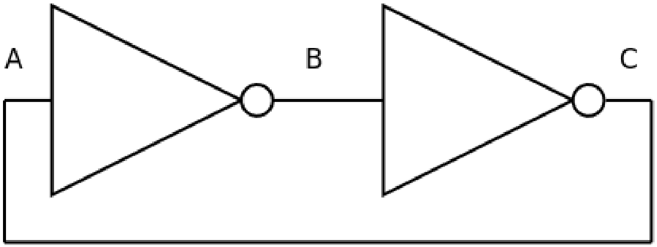

Figure: Bistable with NOT gates and feedback

Something to remember.

Using a multiplexer (remember this one ? We saw it long ago.)
Here, the multiplexer is controlled by a clock, which will alternatively have the value 0 or 1.

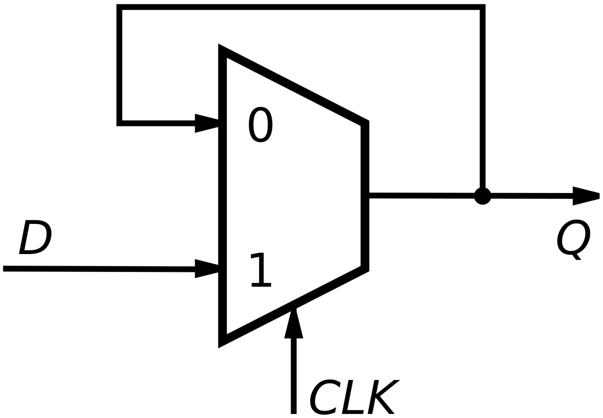

Figure: Latching with a multiplexer

Truth table of the multiplexer latch

Clk	D	Q_n	Q_{n+1}
0	–	0	0
0	–	1	1
1	0	–	0
1	1	–	1

Table: Caption

Explanation :

- If $Clk = 0$, the bit 0 will be selected. With the feedback loop, the stability is there.
- if $Clk = 1$, the bit 1 will be selected. No feedback, direct input to output.

We actually built a D-latch, or just latch

By the way...

This is how you build a latch with NOT bistables and MOS to synchronize with the clock.

Note

An NMOS is a "closed switch" when its gate is high. A PMOS is a "closed switch" when its gate is low.

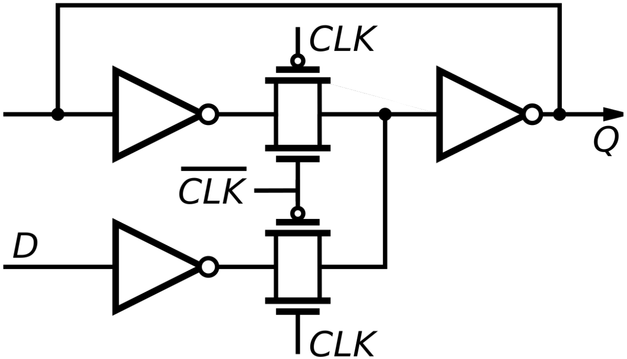

Figure: Latch

Contents

1 Introductory Problem

2 Sequential Logic Basic Circuits

3 State Machines

4 Asynchronous behavior with a synchronous system

5 Memory

Sequential Logic

We have seen in the introductory example that a sequential system has a state and some feedback.
In other words, the output at time t is a function of the input at time t, the output at time $t - 1$, and the internal state.
Sequential logic is considered asynchronous if the state change is not bound to a clock. In other words, the outputs of the circuit change directly in response to changes in inputs.

Flip-flops

We have seen a latch before, and a clock's value triggered it. We define a flip-flop as a device similar in function to a latch triggered by the rising edge of a clock.
A latch is said to be level-sensitive while a flip-flop is said to be edge-sensitive.

D flip-flop

The D flip-flop follows the following equation :

$$Q_{n+1} = D_n \tag{1}$$

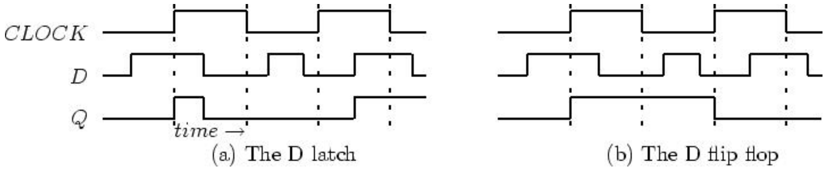

Figure: Chronogram of latch vs. flip-flop

The T flip-flop

If the T input is high, the T flip-flop changes state ("toggles")[27] whenever the clock input is strobed. If the T input is low, the flip-flop holds the previous value. The characteristic equation describes this behavior:

$$Q_{n+1} = Q_n \bar{T}_n + \bar{Q}_n T_n \tag{2}$$

The SR flip-flop/latch

When using static gates as building blocks, the most fundamental latch is the simple SR latch, where S and R stand for set and reset. It can be constructed from cross-coupled NOR or NAND logic gates. The stored bit is present on the output marked Q.
Problem: if $S = R = 1$, we get an indefinite state...

JK Flip-flop

A better SR flip-flop where the forbidden state in the SR FF makes the JK FF toggle its output.
The truth table for the JK FF is :

J	K	Q_{n+1}	Comment
0	0	Q	No change
0	1	0	Reset
1	0	1	Set
1	1	$\bar{Q}$	Toggle

An example: the clock divider by 2

Assuming we have the following D flip-flop :

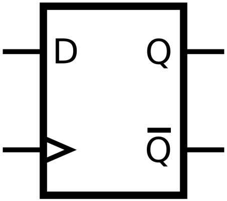

Figure: D flip-flop

How can we implement a clock divider by 2?

Contents

1 Introductory Problem

2 Sequential Logic Basic Circuits

3 State Machines

4 Asynchronous behavior with a synchronous system

5 Memory

So far...

- We know that a D flip-flop can hold data.
- That sequential system needs some memory to keep its state
- It has a combinational part in addition to the sequential state memory.

Another example...

Let's make a combination lock as an example of a sequential system. The lock has :

- 1-bit input signal (the combination is a sequence of bits)
- 1-bit output called UNLOCK: 1 if the combination is correct, 0 otherwise.

In this example, we want to assert UNLOCK, i.e., set UNLOCK to 1, when the last four input values are the sequence 0110.
How many state bits do we need?

How many bits to encode the state ?

Do we have to remember the last four input bits? $\rightarrow$ 4 state bits
Can we remember less information and still do our job? Yes, because we don't need to remember the full history. We only need to know if the most recent entries represent some part of a partially entered correct combination.

Finite State Machines

We can characterize the behavior of a sequential system using a new abstraction called a finite state machine (FSM).
The FSM abstraction aims to describe the input/output behavior of the sequential logic independent of its actual implementation.
One of the interesting challenges in designing an FSM is determining the required number of states since there's often a tradeoff between the number of state bits and the complexity of the internal combinational logic required to compute the next state and outputs.

Back to the lock

Let's make the FSM!

- We need an initial state *SX*
- We need a state for each step of the proper entry sequence.
- We need to handle the rest

Now, it's whiteboard time. For some time.

Valid state diagram

A valid state diagram has the following properties :

- Arcs leaving a state are mutually exclusive, i.e., you can't have two choices for a given input value
- every state must specify what happens for every input combination. "Nothing happens" means an arc looping back to the state.

We can also define two types of FSM :

- Moore machines: an FSM where the outputs are a function of the current state only
- Mealy machines: an FSM where the outputs are functions of both the current state and the current inputs.

FSM and truth tables

An FSM can be transformed into a truth table. Back to the lock :

In	Current State	Next State	UNLOCK
0	SX = 000	S0 = 001	0
1	SX = 000	S0	0
0	S0 = 001	S0	0
1	S0 = 001	S01	0
0	S01 = 010	S0	0
1	S01 = 010	S011	0
0	S011= 011	S0110	0
1	S011 = 011	SX	0
0	S0110 = 100	S0 = 001	1
1	S0110 = 100	S01	1

- A state can be encoded into binary. This can reduce the size of the state representation.
- We can make a further reduction with combinational logic methods.

Put it together !

We can use ROM encoding. The states are fixed, and so are transitions.
We can also use a shift register and a logic gate.
We can also use gates instead of the ROM (with logic gates, it will be fun...).
Note: the ROM would act as a look-up table here, whose inputs are the input, the state, its output, the next state, and UNLOCK. Just like... something for the next classes.

Contents

1 Introductory Problem

2 Sequential Logic Basic Circuits

3 State Machines

4 Asynchronous behavior with a synchronous system

5 Memory

One clock to rule not much

It’s finally time to investigate issues caused by asynchronous inputs to a sequential logic circuit.
By "asynchronous," we mean that the timing of transitions on the input is completely independent of the timing of the sequential logic clock. This situation arises when the inputs arrive from the outside world where the timing of events is not under our control.
We know that logic circuits have propagation delays and setup time, which is not great if the asynchronous input goes haywire.
Maybe we can develop a synchronizer circuit that takes an unsynchronized input signal and produces a synchronized signal that only changes shortly after the rising edge of the clock.
We would use a synchronizer on each asynchronous input and solve our timing problems that way!

Bounded-time synchronizer

Here's a detailed specification for our synchronizer.

- The synchronizer has two inputs, IN and CLK, which have transitions at time t_{in} and t_C, respectively.
- If IN's transition happens sufficiently before CLK's transition, we want the synchronizer to output a 1 within some bounded time t_d (decision time) after CLK's transition.
- If CLK's transition happens sufficiently before IN's transition, we want the synchronizer to output a 0 within time t_d after CLK's transition.
- Finally, if the two transitions are closer together than some specified interval t_e (allowable error), the synchronizer can output either a 0 or a 1 within time t_d of CLK's transition. Either answer is fine if it's a stable digital 0 or digital 1 by the specified deadline.

No dice.

This turns out to be an unsolvable problem! For no finite values, can we build a synchronizer that's guaranteed to meet this specification even when using components that are 100% reliable?

"But a D flip-flop will do the job !"

No. Because of propagation delays. And because a D flip-flop is actually two D-latches...

`https://assets.nexperia.com/documents/data-sheet/74AHC_AHCT74.pdf`

Back to the bistable

Remember the bistable? Well, it has two stable states and one metastable.

In the real world...

Stack D flip-flops. This will smooth out the bumps and will work most of the time.
Why? We have seen that we can use synchronizing registers to quarantine potentially metastable signals for some time. Since the probability of still being metastable decreases exponentially with the quarantine time, we can reduce the failure probability to any desired level. It is not 100% guaranteed, but it is close enough that metastability is not a practical issue if we use our quarantine strategy of stack D flip-flops.

Contents

1 Introductory Problem

2 Sequential Logic Basic Circuits

3 State Machines

4 Asynchronous behavior with a synchronous system

5 Memory
- Shift Registers

Outline

5 Memory
- Shift Registers

Shift registers

There are a few which are made by chaining D latches.

- Serial-in to Parallel-out (SIPO)
- Serial-in to Serial-out (SISO)
- Parallel-in to Serial-out (PISO)
- Parallel-in to Parallel-out (PIPO)

Which can be used for :

- SIPO/PISO: serial communication, i.e., UART, modems...
- SISO: Digital delay line i.e., building block of digital filters
- PIPO: usage for bit-shifting operation (exponentiation by 2)

SIPO shift register

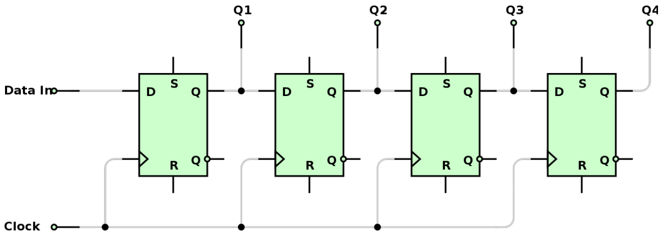

Figure: Serial In-Parallel Out Shift Register