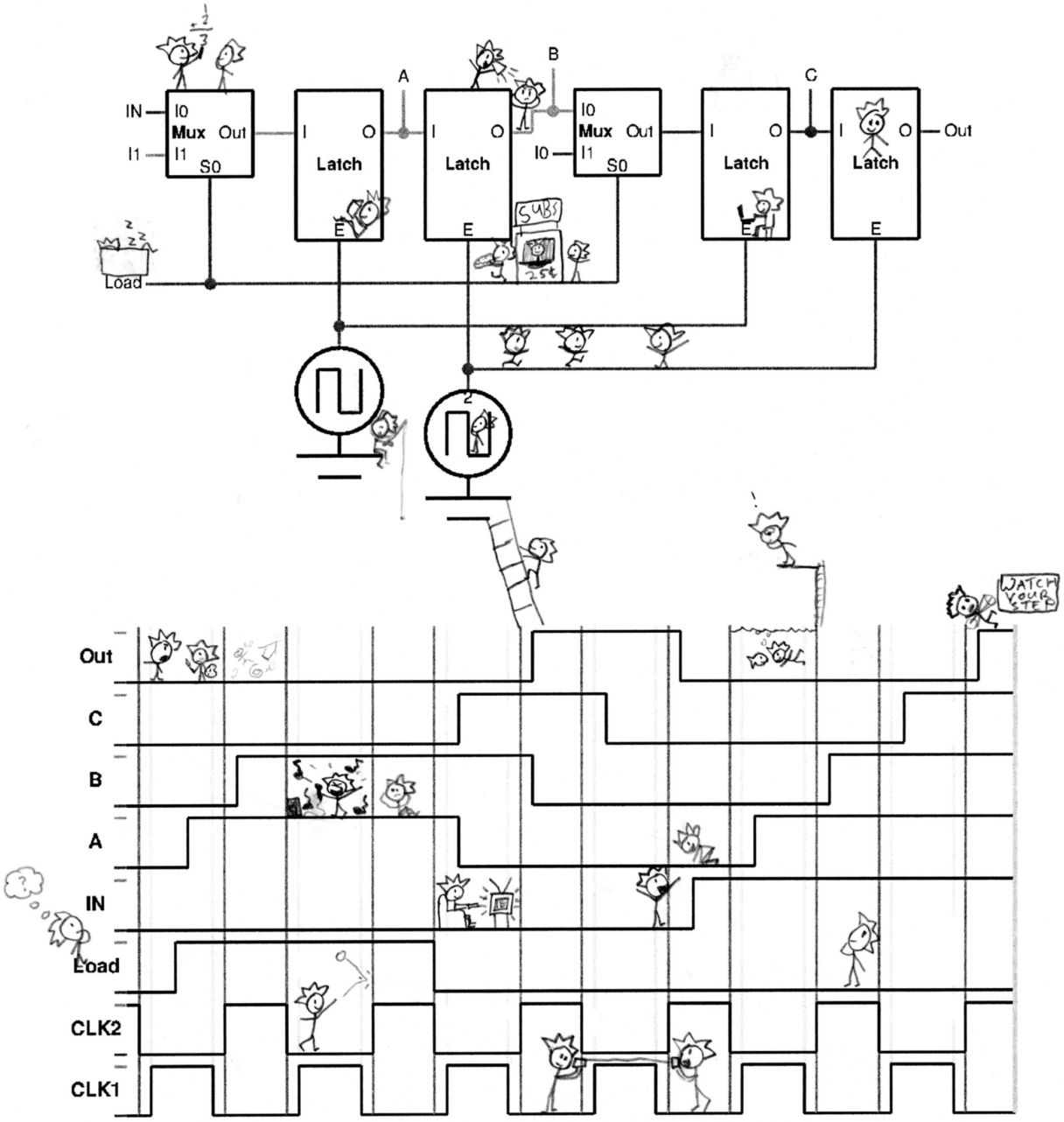
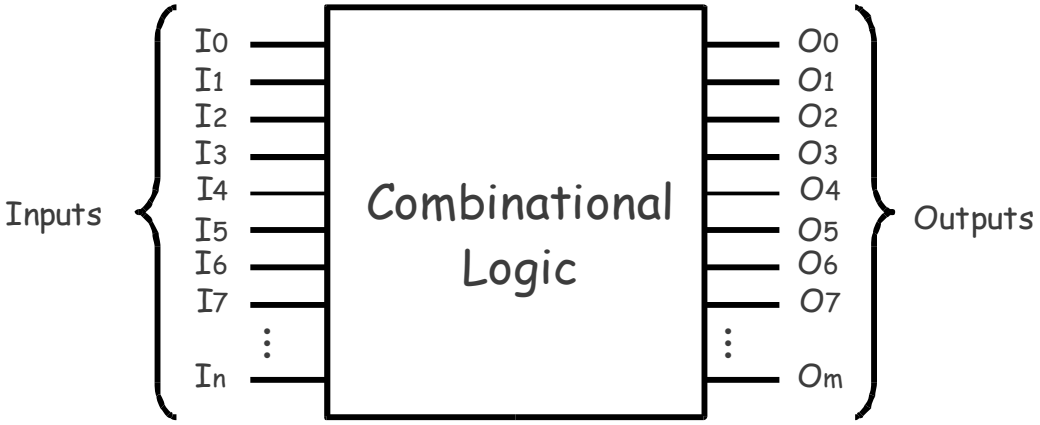
Latches and Registers



Latches and Registers

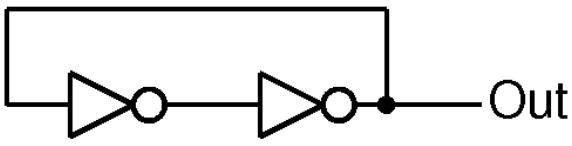
The potential to design functional blocks using switches and wire appears limitless. The analog quantities of our world can be represented using multi-bit words. Operations on these values, defined a Boolean expressions, can be constructed as *combinational logic* of unbound complexity. So what is missing? … *history.*



No matter how complex the implemented function is, it has no memory of previous values or results. All data to be processed must be presented to the combinational logic as inputs. Combinational logic has no state.

Building even the simplest digital system is impossible without state. Consider a basic four function calculator (a four banger). Although it can process big numbers rapidly, how can it solve a simple expression "5 + 3 =" without state? It would require the input keys "5", "+", "3", and "=" to be held simultaneously while the answer is displayed. Multi-digit math is out of the question.

One Bit Store: For useful digital computing systems, a simple block is needed that can store a bit of data for an extended period of time. That way data available now will persist, and can be used later. To best exploit the technology, this bit store must be built with switches and wire. And since many bits are needed, it must be implemented simply. Here's a starting point:



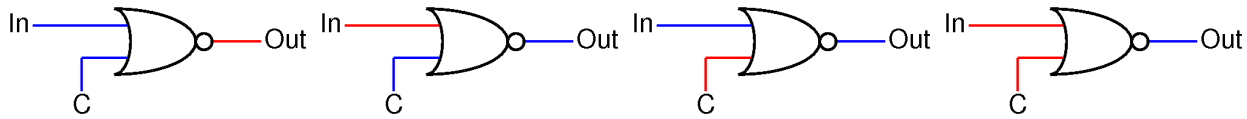
It certainly is simple, only two inverters (four switches). But why would somethings this simple have the ability to store data? The wire looping from the output to the

input is unusual: its uses its own output as an input. Analyzing this simple block is a challenge since, although it has only two nodes, it has no input values. So what are the nodes values?

Surprisingly, there are two answers. The output could be low (0) with the short wire between the inverters high (1). Or the output could be high with the short wire low. Which is right? They both are! This circuit is stable in two states; its *bi-stable*. Since a one bit store maintains one of two states, this simple block is ideal … except it has no input.

RS Latch: In order to use this bit store, it needs inputs that can set the output high, or reset the output low. But it still must retain the ability to hold a state, high or low, for an indefinite period. *Cross-coupled* inverting gates, where the output of each inverting gate is an input to the other, provides bi-stability. But inputs are need to Reset and Set it to a known state.

Consider a familiar gate, seen in a new way: a two input NOR. Assume one of the gate's input is a boolean variable In. The other is a control variable C.



C = 0 C = 1

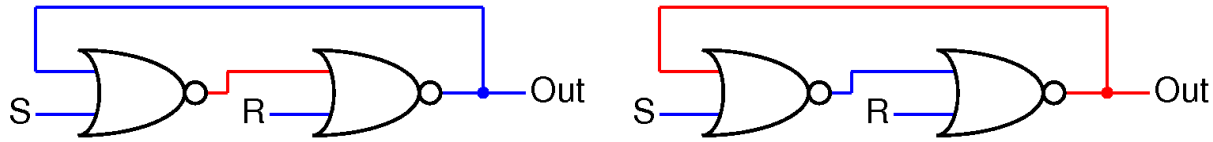
The control signal C determines whether output is related to In. If C is low, the output is the complement of In (i.e., it is an inverter). If C is high, the output is low no matter what the value of In is.

IN	C	Out	
A	0	$\overline{A}$	Out = $\overline{In}$
X	1	0	Out = 0

This is just what is needed. The heart of a bit store is two cross-coupled inverters. To force the output into one of two states, the inverter can be preempted. Using cross-coupled NOR gates, the control inputs turn off one of the two inverters, creating a known output (low or high).

When both R (reset) and S (set) are low, both NOR gates act as inverters. Their other input is complemented to become the output. So when Reset and Set are low,

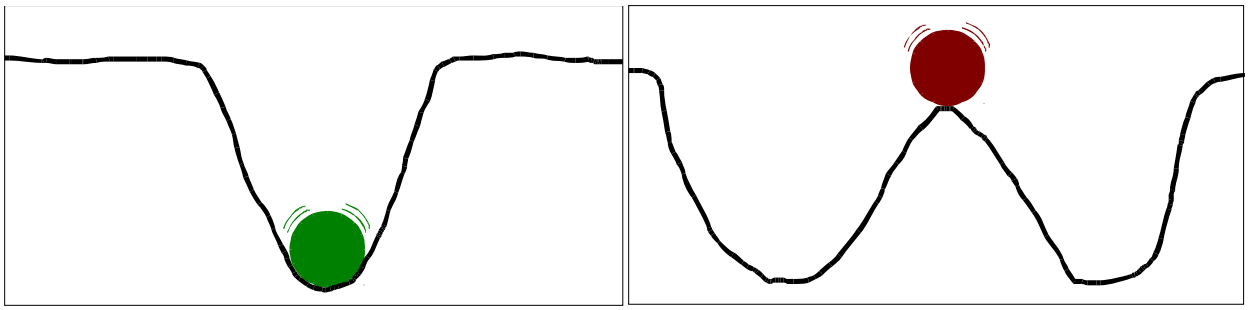
cross-coupled NOR gates become cross-coupled inverters used in the previous bit store implementation.



When either the Reset or Set inputs is temporary asserted (set high), it turns off an inverters, forcing the output into a known state. When Reset is high, Out is low independent of the first NOR gate's output. Since Out is low, the first NOR gate (acting as an inverter) makes the ignored input to the second NOR gate high. So when Reset returns low, and the second NOR gate becomes an inverter, Out remains low.

When Set is asserted (with Reset low), the output of the first NOR gate is low. With Reset low, the first NOR gate's output is inverted by the second NOR gate, setting Out high. Since this also sets the other input of the first NOR gate, the gates retain this state when Set is deasserted (set low).

Meta-Stability: Reset and Set can be asserted individually to force the RS latch into one of its two stable states. When both Reset and Set are low, this stable state remains unchanged. But what happens when Reset and Set are asserted simultaneously? Both NOR gates have low outputs. While this is logically correct, it can lead to an unpredictable state when Reset and Set are deasserted simultaneously. Which of the two stable states will it become? This condition is call *meta-stable* because the state of the latch is unknowable. It is also not knowable how long it will take for the latch to return to one of the two states.



"stable" "meta-stable"

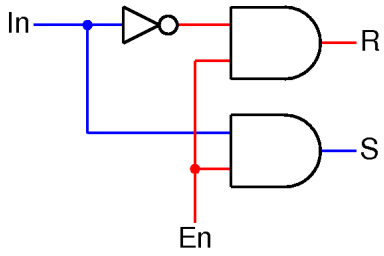
To appreciate the difference between stable and meta-stable, consider a ball in a valley versus a ball balanced in a peak. Small forces on the stable ball will not change its state. In contrast, a small force applied to the meta-stable ball will cause a significant state change. Needless to say, meta-stability should be avoided

when predictable storage is desired. So Reset and Set are only asserted individually.

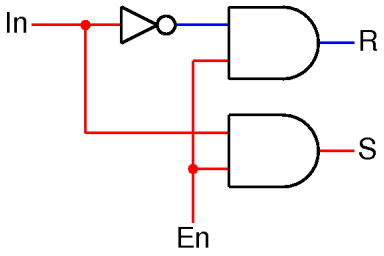
R	S	Out	
0	0	Q_o	hold
1	0	0	reset
0	1	1	set
1	1	0	avoid

The hold state employs a new symbol: Q_o to represent a previously defined state. It can be either 0 or 1, depending on whether Reset or Set was last asserted.

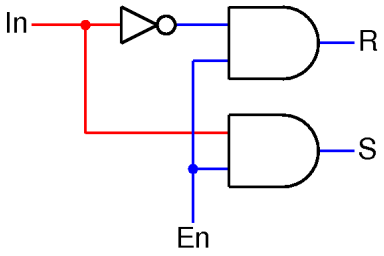
Transparent Latch: An RS latch can provide needed state. But it requires specific control signals to define and hold the storage. When a 0 is being stored, Reset is asserted and Set remains low. When a 1 is being stored, Set is asserted while R is low. When the value is held, neither Reset nor Set are asserted. This can be generated from an input In and an enable En that allows the input to be captured.



assert R

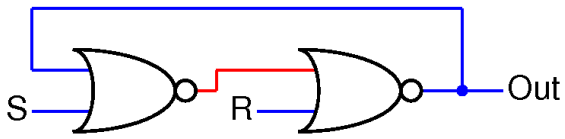


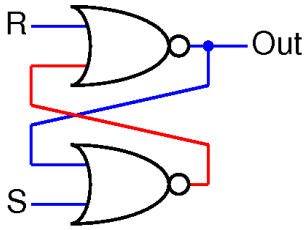
assert S



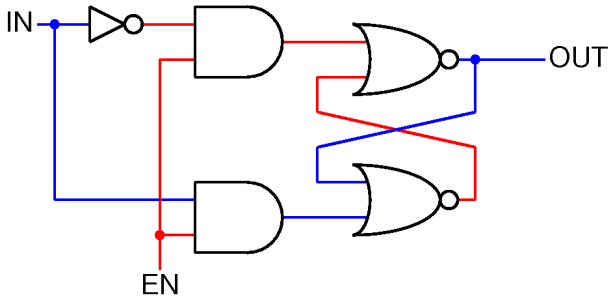
hold

Before connecting this circuity, the RS latch must be rearranged.

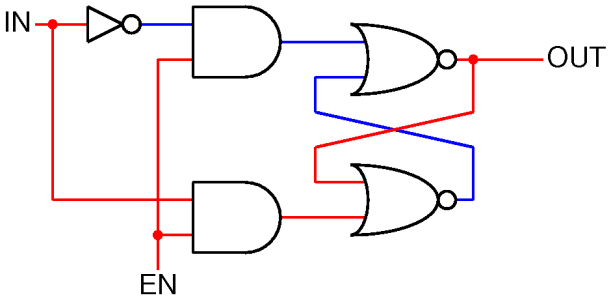




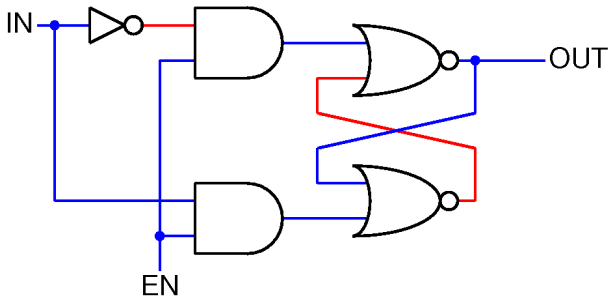
Combining both circuits produces a *transparent latch*. This transparent latch, shown below, can be in one of four different cases. Two are completely defined by the inputs: *storing 0* and *storing 1*. Two are defined by En and the internal state: *holding 0* and *holding 1*. In the second two cases, IN is ignored; it doesn't matter whether its one or zero because its masked by the AND gates.



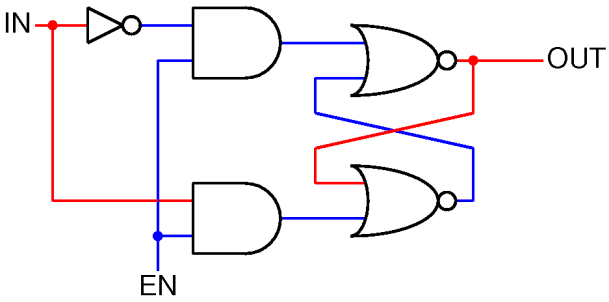
storing 0



storing 1



holding 0



holding 1

Here's the functional behavior of a transparent latch.

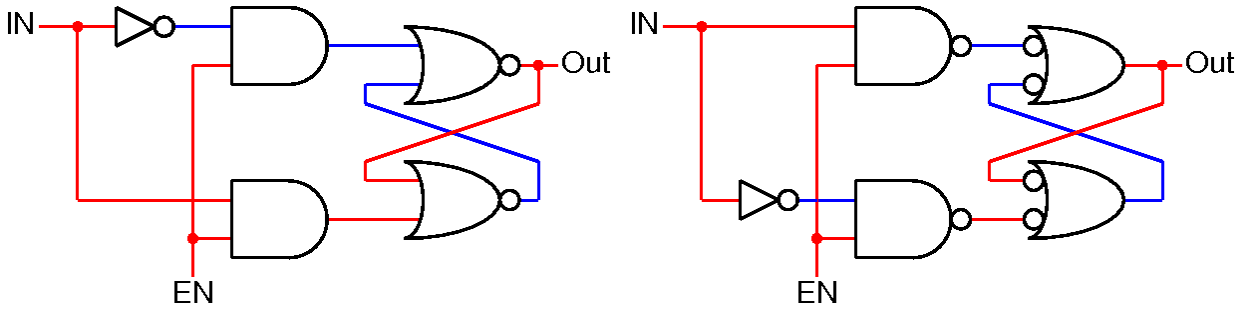
IN	En	Out	
X	0	Q_o	latch
A	1	A	transparent

When enable is asserted, the output follows the input so the latch becomes transparent. When enable is not asserted, the latch maintains the stored state on the output, independent of the input. Its value was defined at the last moment of transparency.

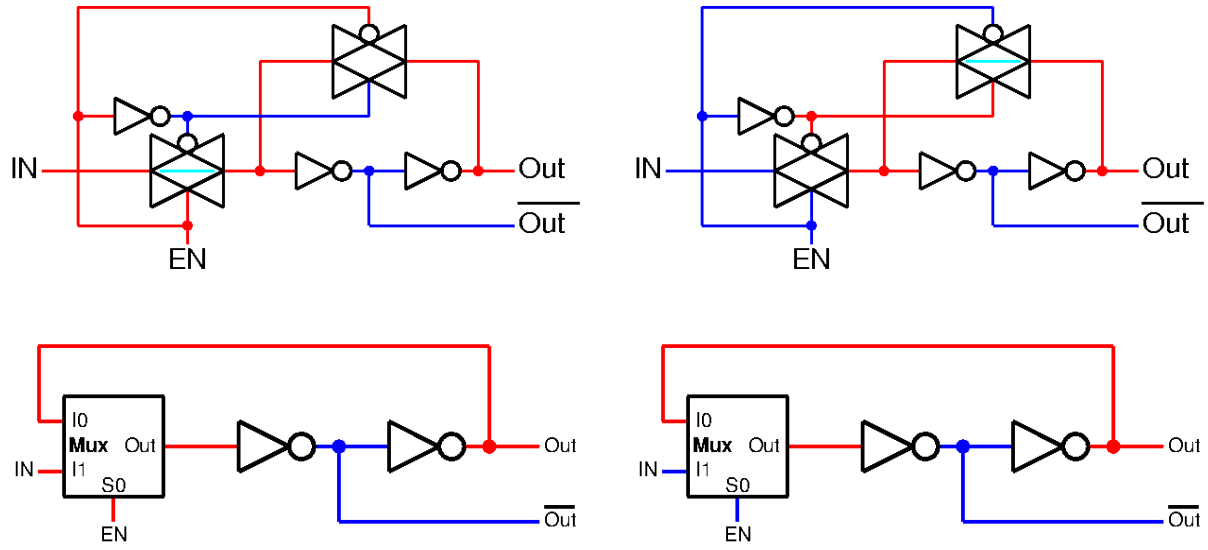
Implementation Costs: When implemented with switches and wire, this transparent latch requires two NOR gates (2 x 4 switches), two AND gates (2 x 6 switches), and one inverter (2 switches) for a total of 22 switches. Not bad. But digital systems require a lot of storage bits. Is there a cheaper implementation of this behavior using switches and wire?

Some tricks from mixed logic can help. If the bubbles on the NOR gates slide arround to the inputs and an extra pair of bubbles is added between the AND and OR, this implementation is transforms from two NOR and two AND gates to four NAND gates. Here the Set and Reset signal become active low (they are asserted when low, unasserted when high). Generating these signals requires the inverter to move down. But this latch implementation realizes the transparent latch behavior

with four NAND gates (4 x 4 switch) and one inverter (2 switches) for a total of 18 switches.



But can the count still be lowered? Let's go back to basics. Two cross coupled inverters are capable of storing a bit, but lack an input. Suppose pass gates are used to selective configure these inverters into either a transparent mode (where the the input is connected and the feedback pass is removed), or a hold mode (where the feedback path is connected and the input is removed). These pass gates serve as a two to one mux. It may be easier to understand when drawn as a mux.

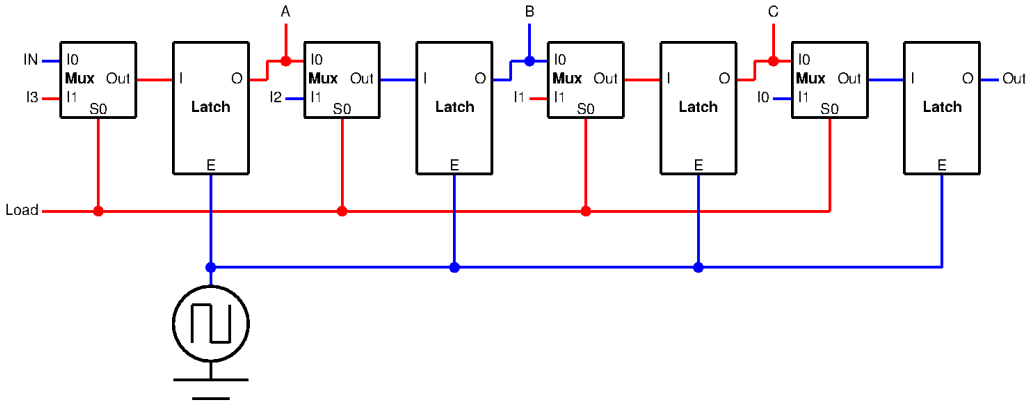


transparent mode hold mode

This implementation employs three inverters (2 x 2 switches) and two pass gates (2 x 2 switches) for a total of 10 switches. It is called a ten transistor latch and is the significant storage element in digital computation. Can we do better than ten transistors? Yes, but at a cost in speed, and only in dense arrays. More on this follows in the memory chapter.

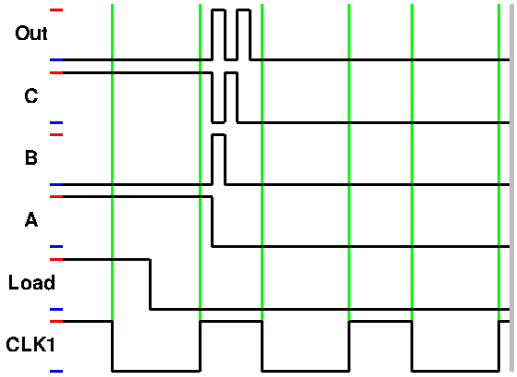
Latch Limitations: Latches can store a single bit of data, but with limitations. Consider a parallel to serial shift register. This is a device that can take parallel

word (in this case, four bits), and shift it down a serial wire in an orderly way. We use these devices to transfer data between our digital products. USB is short for "Universal Serial Bus". SATA means "Serial Advanced Technology Attachment". Strange as it may seem, serial buses often transfer data faster than parallel buses. So the need to load a parallel word into a clocked serial bus interface is widespread. Here's a first attempt.



Notice we are using latches to hold the word (I3:I0) when the Load signal is high. Then we use a simple clock to move bits along to Out.

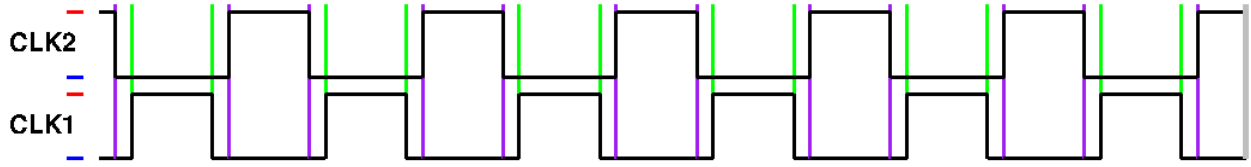
What is a Clock: In physics, the most used clock signal is a sinusoidal waveform. But in the digital world, everything is a one or a zero. A clock is a square wave that alternates between high and low at a defined period. A timing diagram shows the behavior as the load signal goes low and the data move serially through the latches. Time advances from left to right. Each signal is stacked with high and low values indicated by the red and blue marks. Note clock alternates between zero and one.



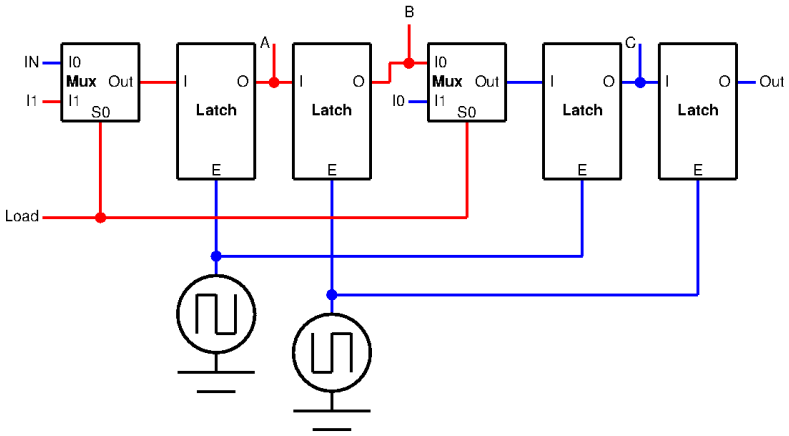
The problem occurs when the clock goes high (after Load goes low). All the enables on all of the latches go high and all latches become transparent. The stored data does travel to the output, but not in an orderly fashion. Instead bit race through

the latches independent of the clock. Not good. There is no way to capture and reconstruct the parallel word on the other end of the serial bus.

Two Phase, Non-Overlapping Clock: The problem with a latch is that is has no storage when its transparent. It can't hold an old value and accept a new value at the same time. To do accomplish this, there needs to be two latches and a special clocking scheme that allows one latch to hold the current value while a new value is being captured. The clocking scheme must allow each latch to be transparent independently, with a brief period in between where both latches are holding their value. Here's the clock that produces this behavior:

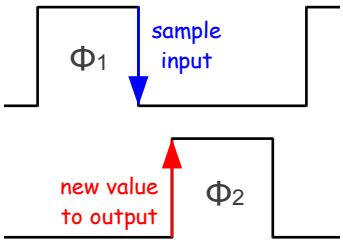


Both clocks have the same shape, including a small asymmetry of being low longer than being high. But the second clock is phase shifted by 180 degrees. Note also that the two clock are never high at the same time. This creates a two phase, non-overlapping clock. The clock signals are often named phi1 (Φ_1) and phi2 (Φ_2). It is widely used in digital computation where phase periods are set large enough to accommodate the gate depth x gate delay, and non-overlap periods are large enough to accommodate anticipated clock skew. This scheme will help create a workable shift register.

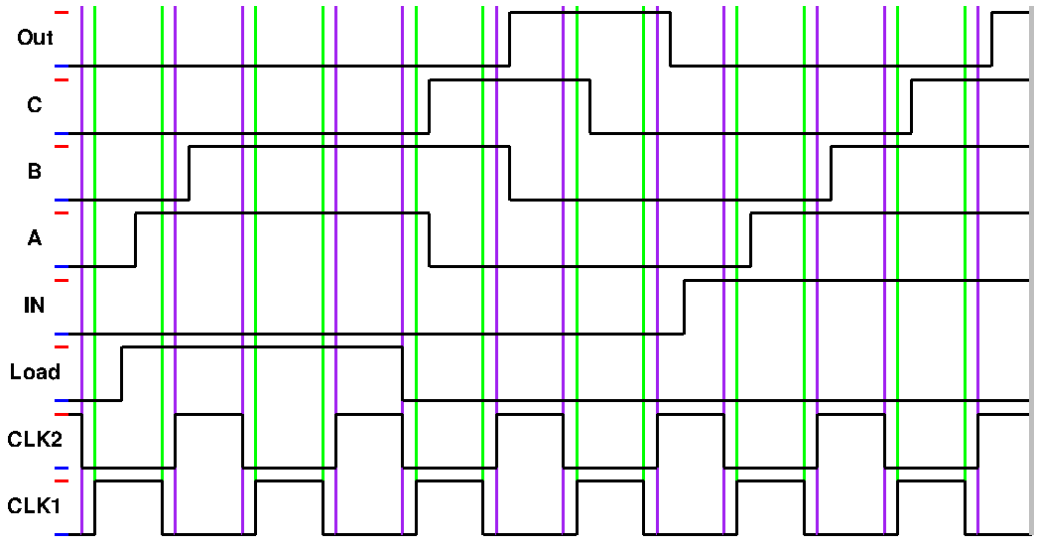


After data is loaded, it is advanced in the shift register in time with the clock frequency. This orderly movement is defined as the latch alternate between transparent and hold modes. During Φ_1, the first latch is transparent while new data is sampled. The falling edge of Φ_1 defines the sample point. Then on the rising

edge of Φ_2, this new value moves forward through the second (now transparent) latch.

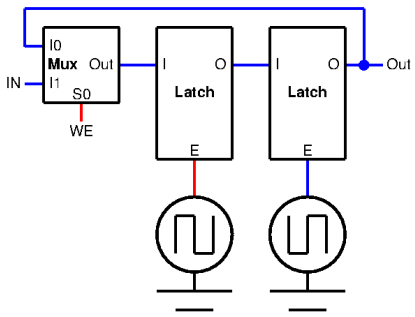


These two critical moments (the falling edge of Φ_1 and the rising edge of Φ_2) define this clock scheme behavior. Here's the timing diagram of this functional shift register:



Note the movement of ones and zeros through the monitor points A, B, C and Out. Of course, this four latch shift register can only maintain two bits. Half of the latches are transparent and cannot hold values.

Register: Two latches, plus the multiplexer form the core of a register.

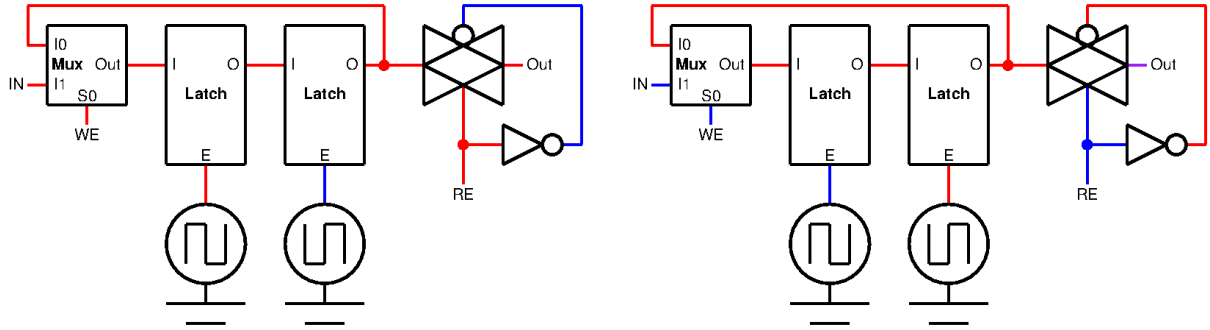


IN	WE	Clk	Out
X	0	↑ ↓	Q_0
A	1	↑ ↓	A

By connecting the output through a 2 to 1 mux to the input of the first latch, the ability to selectively write (or preserve) a register's value can be controlled. Like

the enable signal on the transparent latch, the write enable (WE) signal either selects a new input, or recycle the current output as the input. However this is a synchronous behavior in that the changing or preserving of a stored value is in sync with the clock signals. This selective write is call a *write port*.

Read Port: What would a *read port* be? Writing means changing the register state. Reading (or not) has no effect on its value. So what does a read port accomplish? Often registers are read onto a shared bus. Since only one value can be *read* onto the bus, a read port is a method of passing the register's contents onto a write (or not). This has been explored in demultiplexers, and is efficiently accomplished using a pass gate.



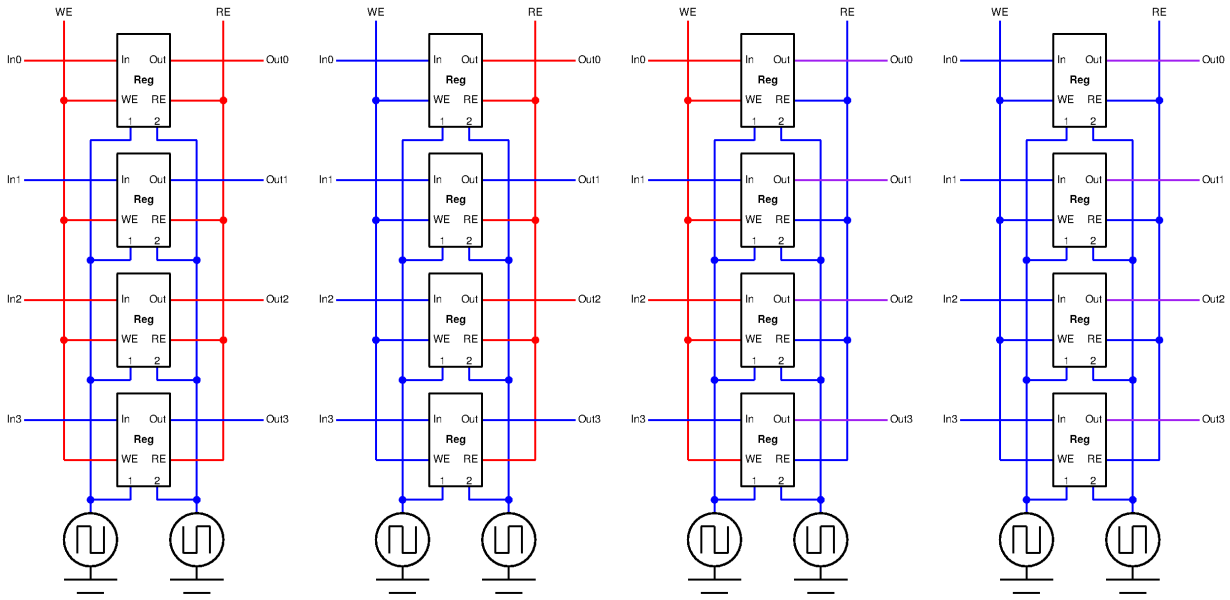
WE = RE = 1 WE = RE = 0

The register on the left is being written and read. The register on the right is holding a value that is not being read (Out is floating). The behavior of this widely used storage element is shown below. Note that read and write are independent operations. Even when the register is not read (the output is floating), write operations can be performed. And when neither write or read operations are performed, a bit is still be stored.

IN	WE	RE	Clk	Out	
X	0	0	↑ ↓	Z_0	hold
A	1	0	↑ ↓	Z_0	write
X	0	1	↑ ↓	Q_0	read
A	1	1	↑ ↓	A	write & read

Word-Wide Register: Once a one bit register is designed, it can be replicated to create a word wide register to store multi-bit values. These parallel registers are stored and loaded using multiple bit values. In this example, the word size is four bits. Control signals and clocks to read and write the register are shared. Individual lines for each input and out bit position are connected separately. Read

and write operations can be performed independently, something latches cannot do. In this example, the stored value of the register (0101) is unchanged in the four examples.



read & write | read | write | hold

Summary: Here's a summary of key points in digital storage:

- Storage is needed for digital systems. It can created using simple cross-coupled inverting gates in a circuit that is bi-stable.
- A *Transparent Latch* is can store a bit of data, but it cannot hold data when a new bit is being stored. The 10T latch is the workhorse in digital systems.
- A *Register* can simultaneously be read and written. It is built of two latches, one to hold the current value while the other receives the new value.
- A *two-phase non-overlapping clock* provides necessary timing for digital systems. Its parameters (depth and non-overlap delay) determine performance of the digital system.
- A *shift register* shifts parallel words over a serial bus, often at high speeds. Serial interfaces are widely used in digital systems (USB, SATA, etc.).
- A timing diagram shows how sequential systems evolve in time. Behavioral tables cannot fully capture this information.

