Logic Circuits
MANUSCRIPT

William F. Barnes
*†

October 4, 2023

Contents

1 Introduction

It's impossible to exist today without having heard some person assert 'computers are all ones and zeros'. While cryptic and perhaps a bit reductionist, it *is* true, at a certain level, that all of digitized electronics can be reduced down to binary 'on' or 'off' conditions represented by ones and zeros.

Of course, opening an electronic device to reveal its inner entrails will hardly betray its operation, even while the device is running. The intention here is to examine exactly *what* is turning on or off, and how this could possibly be used to manage information and build complex systems.

1.1 Basic Circuit

For a starting point, let us contrive an unspecified but still' simple' circuit using analog components as sketched in Figure 1. The region labeled *Load* is where one would place any number of resistors, inductors, capacitors, lights, fans, heaters, and so on.

Solid black lines are considered pure conductors, or *wires*. Motivating the circuit is a direct current voltage source, such as a battery, labeled *DC Source*. Connected to the positive terminal on the voltage source is a toggle switch that can *open* (turn off)

†Circuit diagrams designed at www.circuit-diagram.org

or *close* (turn on) the circuit. By convention, the *current* flow through the circuit originates from the 'high' (+) side of the voltage source, and flows back in through the 'low' (-) side. (The actual electrons go in reverse of the current.)

The total current entering the load is equal to the total current leaving the load, and this can be measured by placing an *Ammeter* in series with the load. The *Voltmeter* is a silent observer of the circuit with infinite resistance (ideally, at least), which is why it is always situated in parallel with the component(s) being measured.

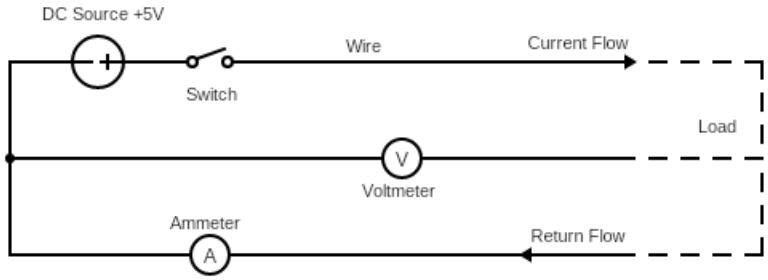


Figure 1: Basic circuit.

1.2 Digital Components

The load of an electric circuit may consist of analog components, digital components, or both. Digital components called *chips* are made from clever configurations of of silicon, carbon, and other common or uncommon materials. The end product tends to appear as small 'black box' depicted in Figure 2, which is a *74LS08 Quad 2-input AND Gate*.

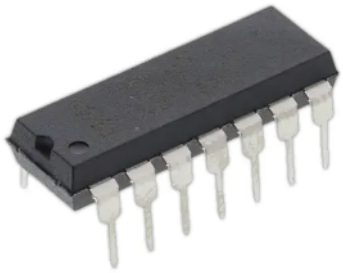


Figure 2: 74LS08 Quad 2-input AND Gate.

Making sense of the so-called 74LS08, or any chip for that matter, requires a map called a *pinout diagram* as shown in Figure 3.

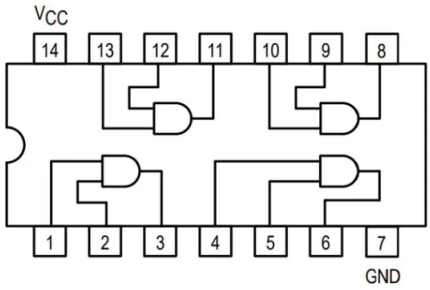


Figure 3: Pinout diagram of 74LS08 Quad 2-input AND Gate.

The pinout diagram states the purpose of each 'leg', or *pin* of the chip. For the example on hand, we see that pins 14 and 7 correspond to the high- and low voltage sources required for the chip's operation.

As a matter of program note, we will work in the regime where positive five volts ($+5\,V$) corresponds to an 'on' state, using the shorthand symbol 1. Conversely, the 'off' state symbolized by 0 corresponds to zero volts ($0\,V$).

1.3 Digital Logic

AND Gate

Studying the pinout diagram of the 74LS08, we can see an embedding of four identical circuits, each having a wire leading from one peg to one location on a D-shaped glyph. This is the symbol for the AND gate, and there are four such gates on the chip shown. Taking the AND gate connected to pins [1], [2], and [3], it follows that pins [1] and [2] correspond to the *inputs* of the gate. The third pin [3] corresponds to

the *output* of the gate. A similar comment applies to pins [4]-[6], and so on for all four gates.

The behavior of the AND gate is specified as follows:

- When both inputs are 1, the output is 1.
- When both inputs are 0, the output is 0.
- When either input is 0, the output is 0.

The same information is represented efficiently in truth table form. The left columns are inputs, the right column is output:

AND		
0	0	0
0	1	0
1	0	0
1	1	1

OR Gate

Similar to the AND gate and implemented on a similar chip is the OR gate. Jumping right to the truth table, the OR gate behaves as follows:

OR		
0	0	0
0	1	1
1	0	1
1	1	1

That is, the OR gate returns 1 when either input is 1, and only returns 0 when both inputs are 0. Notice that the truth table for the OR gate is similar to that of the AND gate, as only the output column varies.

The astute reader may notice that the output columns in the sixteen tables above express the numbers 0 to 15 in binary.

1.4 Inversion and Splitting

A different kind of component that partners with logic gates is the INV gate, equivalently called a NOT gate, also known as the *inverter*. The INV gate takes a single input and returns a single output. If the input is 1, return 0. If the input is 0, return 1. The inverter has a implies a simple truth table:

INV	
0	1
1	0

Finally, there are instances when it's necessary to 'peel apart' the voltage or current from one wire by transforming into two wires, and this is called *splitting* or *dividing*. Generically, a so-called split can be thought of as a generic logic gate in reverse as shown in Figure 5.

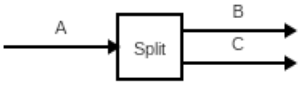


Figure 5: Generic divider.

For our purposes, the generic divider need only 'clone' the voltage from input A and assign the same value to the outputs B and C. That is, the inside of the 'Split' box is simply a junction of three wires.

In fact, the input pairs $(0,0)$, $(0,1)$, $(1,0)$, $(1,1)$ are the *only* combinations to consider when it comes to input.

Fourteen More Gates

Writing the output of the AND gate as a minimal list, and doing the same for the OR gate, we have

$$\begin{aligned} \text{AND} &= \{0,0,0,1\} \\ \text{OR} &= \{0,1,1,1\} \ . \end{aligned}$$

By inspecting the output side of each truth table, we see that the lists $\{0,0,0,1\}$, $\{0,1,1,1\}$ are just two members of *sixteen* possible lists, implying there are fourteen siblings to the gates already listed. That is, there are sixteen possible logic gates that could represent the unknown Figure 4. Throughputs are labeled by letter rather instead of by number.

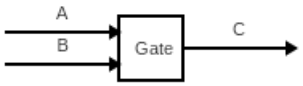


Figure 4: Generic logic gate.

All sixteen gates are represented in truth tables below. Certain logic gates are obligatory for completeness of the list but are otherwise trivial, for instance the NULL gate returns zero regardless of its inputs, whereas the A gate simply transmits whatever the first input is doing, and so on. The IMP gate (or any mentioning it) is a bit esoteric and isn't needed for now. There are four gates though, namely AND, OR, NOR, and XOR, that we'll want to keep handy.

NULL		
0	0	0
0	1	0
1	0	0
1	1	0

AND		
0	0	0
0	1	0
1	0	0
1	1	1

NIMP		
0	0	0
0	1	0
1	0	1
1	1	0

A		
0	0	0
0	1	0
1	0	1
1	1	1

-(B IMP A)		
0	0	0
0	1	1
1	0	0
1	1	0

B		
0	0	0
0	1	1
1	0	0
1	1	1

XOR		
0	0	0
0	1	1
1	0	1
1	1	0

OR		
0	0	0
0	1	1
1	0	1
1	1	1

NOR		
0	0	1
0	1	0
1	0	0
1	1	0

XNOR		
0	0	1
0	1	0
1	0	0
1	1	1

-B		
0	0	1
0	1	0
1	0	1
1	1	0

B IMP A		
0	0	1
0	1	0
1	0	1
1	1	1

-A		
0	0	1
0	1	1
1	0	0
1	1	0

IMP		
0	0	1
0	1	1
1	0	0
1	1	1

NAND		
0	0	1
0	1	1
1	0	1
1	1	0

ON		
0	0	1
0	1	1
1	0	1
1	1	1

1.5 Symbols

Logic and Flow

As seen with the AND gate, it turns out that all circuit components already have designated symbols. Several of these were shown back in Figure 1, namely the symbols for a voltage source and various meters.

An alphabet unto their own, logic synbols and related items can be interlaced using wire to spell out intricate circuits, the complexity of which would be extremely difficult to grasp without the aid of symbolic representation. We'll be making heavy use of six components depicted in Figure 6.

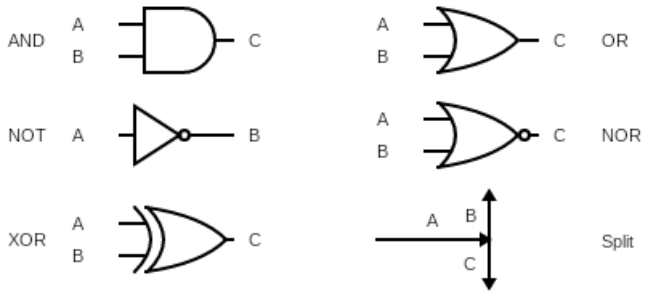


Figure 6: Symbols: AND, OR, INV, NOR, XOR, Split

Buttons and Lights

The inevitable concern of inputting information to a circuit and reading information from a circuit is answered by buttons and lights, respectively. Shown in Figure 7 are the symbols for the push switch, toggle switch, and light-emitting diode (LED). These are minimalist representations of, for instance, a keyboard and a screen.

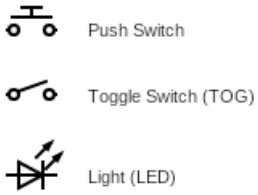


Figure 7: Symbols: Push, Toggle, LED

1.6 Logical Equivalencies

One reason for not needing *all* logic gates simultaneously is that some gates can be built from others.

Ruling out XNOR

For a swift example, the output of the XNOR gate is characterized by

$$\text{XNOR} = \{1, 0, 0, 1\} \ ,$$

which as the name XNOR may suggest, is the inversion of the XOR gate:

$$\text{XOR} = \{0, 1, 1, 0\}$$

To simulate the output XNOR, one simply need install the XOR gate and send the output through a INV gate. For this reason, it suffices to exclude a XNOR chip from the toolkit altogether, so long as there is sufficient stock of INV and XOR chips.

Another way to replicate a XNOR gate, that does not rely on XOR, is the arrangement of INV, OR, and AND gates depicted in Figure 8

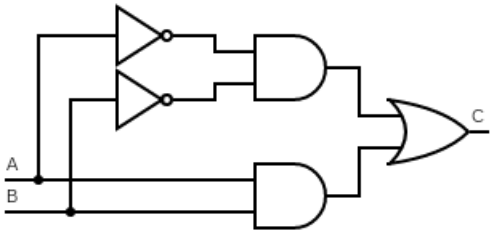


Figure 8: Equivalent XNOR gate.

OR Substitute

A case that actually saves skin in the laboratory is one that builds an OR gate from a single AND gate and three INV gates as depicted in Figure 9.

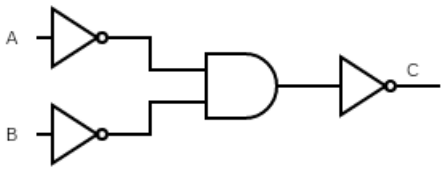


Figure 9: Poor man's OR gate.

AND Substitute

It turns out that an AND gate can be made from three NOR gates as depicted in Figure 10. Note too that the NOR gate is the linear composition of INV and OR.

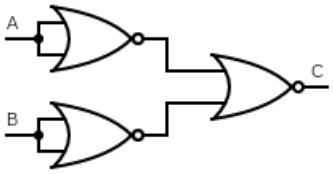


Figure 10: Pedant's AND gate.

1.7 Complete and Minimal Sets

The phenomenon of logic gate equivalencies implies a deeper pattern among logic gates than meets the eye. One question that arises, which happens to have multiple answers, concerns finding subsets of all logic gates that can reconstruct the entire table. Even more interestingly, is there a *single* logic that can do the job?

Skipping the stressful derivations, the answers to these questions go as follows:

- AND, INV form a complete set.
- OR, INV form a complete set.

- NAND forms a minimal set.
- NOR forms a minimal set.

Surprisingly, there are in fact two gates, NAND and NOR, that can alone be used (with copies of themselves) to reconstruct all other gates, even the INV gate. Figure 11 demonstrates this while introducing the NAND symbol.

Figure 11: Using NAND to make INV.

The full derivation of all logic gates from NAND alone or NOR alone is left as a research project for the reader.

1.8 Feedback

While plenty can be accomplished using logic gates in simple series and parallel configurations, a whole new world opens when we make use of *feedback*, which means using the output of a logic gate to influence its own input.

SR Latch

An important and nontrivial example of a structure that utilizes feedback is the set-reset latch, or SR latch. The latch is composed of two NOR gates mutually arranged as depicted in Figure 12. One input leading to each NOR gate comes in as 'normal' input, whereas the second input on each gate is tied to the output of its sibling.

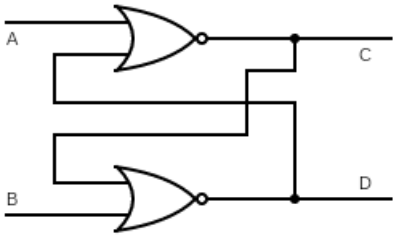


Figure 12: Set-Reset latch utilizing feedback.

As it turns out, the SR latch highlights a profound ability of logic curcuits that one wouldn't think was possible. Though, working out the truth table for feedback-driven circuits is sadly quite a chore and takes some getting used to.

1.9 Bit Storage

Starting with the SR latch depicted in Figure 12, label the inputs using A, B, and also the two outputs C, D. To be definite, associate A, C with the top half, and B, D with the bottom half.

Going with a brute force analysis, we first consider all possible states, even the wrong ones, that could possibly take place in a circuit with two inputs and two outputs. There are $4^2 = 16$ of these in total, and each can be stored in a state vector

$$\mathcal{S}_j = \{A_j, B_j, C_j, D_j\} \ ,$$

where j ranges from 0 to 15. Now the game becomes eliminating states that could not work for the SR latch on hand.

To dig in, consider the state $\mathcal{S}_0 = \{0, 0, 0, 0\}$, which has all inputs and outputs set to 'off'. Consulting the NOR truth table, we find this state to be contradictory, as the output of each NOR gate should be 'on' when both inputs are 'off'. Thus we immediately dismiss the state $\mathcal{S}_0$.

We can similarly analyze $\mathcal{S}_8 = \{1, 0, 0, 0\}$, which has one input set to 'on' with everything else 'off'. This is also a contradictory situation, as the nor gate being fed two 'off' signals results with 'on'. We can by symmetry dismiss $\mathcal{S}_4 = \{0, 1, 0, 0\}$. The states $\mathcal{S}_5 = \{0, 1, 0, 1\}$ and $\mathcal{S}_{10} = \{1, 0, 1, 0\}$ are also dispensable for containing internal contradictions.

A family of states can be knocked away by asking how it's possible for the two outputs be be 'on'. This only happens when all inputs are 'off', but outputs are touching inputs, and another contradiction is reached. It follows that any state of the form $\{A, B, 1, 1\}$ can be excluded, ruling out $\mathcal{S}_3$, $\mathcal{S}_7$, $\mathcal{S}_{11}$, $\mathcal{S}_{15}$ as viable states.

Setting both inputs to 'on' leads to undefined behavior. The states $\mathcal{S}_{13} = \{1, 1, 0, 1\}$ and $\mathcal{S}_{14} = \{1, 1, 1, 0\}$ are easily shown to be unattainable. The state $\mathcal{S}_{12} = \{1, 1, 0, 0\}$ is not forbidden, but like a coin landing on its side, is an undefined state.

After all that, there are four surviving states allowed for the SR latch:

$$\mathcal{S}_1 = \{0, 0, 0, 1\} = \text{reset}$$
$$\mathcal{S}_2 = \{0, 0, 1, 0\} = \text{set}$$
$$\mathcal{S}_6 = \{0, 1, 1, 0\} = \text{set}$$
$$\mathcal{S}_9 = \{1, 0, 0, 1\} = \text{reset}$$

SR Latch Analysis

Listing all possible SR latch states $\mathcal{S}_1$, $\mathcal{S}_2$, $\mathcal{S}_6$, $\mathcal{S}_9$ together, it stands out that the 'rest' state of the circuit, i.e. that which is no inputs coming in, corresponds to *two* different outputs. Left untouched, the circuit could be in either state $\mathcal{S}_1$ or $\mathcal{S}_2$. For this reason we see how the name 'latch' applies. The purpose of the SR latch is to 'remember' one bit of data.

The state of the SR latch can be toggled by setting one of the inputs to 1 (and optionally back to 0). Looking at states $\mathcal{S}_6$ and $\mathcal{S}_9$, we see the so-called 'set-reset' operation will toggle the output of the gate 'diagonal' from the input being toggled. This means the output state cannot be toggled by rapidly changing one input. Both inputs must be used to fully toggle the circuit.

By convention, the symbols A, B, C, D are replaced by R, S, Q, and Q', respectively. The 'set' state applies to $Q = 1$, $Q' = 0$, and the 'reset' state is the opposite of this. In summary, the entirety of the SR latch is represented in the following table:

SR Latch				
R	S	Q	Q'	
0	0	0	1	reset
0	0	1	0	set
0	1	1	0	set
1	0	0	1	reset
1	1	0	0	undefined

2 Simulating Logic Circuits

Electronic devices are widespread and inexpensive thanks to the silicon/digital revolution, and this has led to the gradual buildup of magnificent systems like modern computers.

In this study, we shall walk the path backwards by starting with a computer and writing a program to simulate what happens in simple logic circuits. The end product is to be a kind of 'virtual breadboard' designed to capture all of the phenomena described previously, including feedback and data storage. (What we are *not* shooting for is a full-blown simulator with alternating currents, inductance, and all of the detritus of electrodynamics.)

In real life, electric circuits are a rather bare-bones implementation of the laws of electromagnetism and quantum mechanics, and exactly 'where' and 'when' things occur in the circuit is a matter of the whims of nature. By contrast, our tool of choice is a *procedural programming language*, which is executed linearly and thereby unable to capture the immense simultaneity of a real electric circuit. The best we can do is model the circuit using some kind of data structure and run algorithms on the data to simulate the passage of time. By choosing the right structures and rules, we can capture the circuit's behavior.

With these considerations, the program we design must be able to:

(i) load the specification of a circuit made from components and wires.

(ii) receive real-time user input by simulating switches and buttons on the keyboard.

(iii) employ an evaluation algorithm that properly updates the circuit state.

(iv) display all outputs (including a summary of inputs) and the state of the circuit to the screen.

2.1 Labeling Convention

To make some headway on the problem, it's required to have a sufficient system for labeling the components and wires in a circuit, along with all inputs and outputs. To this end we shall use numbers as labels, thus any number used to identify a component or a wire is used solely for labeling and has no numerical purpose. Formally, such a number is called the *identity*.

For our purposes, it suffices to set up a system with the following qualities:

- A number (in square brackets) [1] to [999] identifies a single component.
- A number (not in square brackets) above 1000 identifies a wire.
- Inputs are labeled `in1`, `in2`, `in3`, etc.
- Outputs are labeled `out1`, `out2`, `out3`, etc.

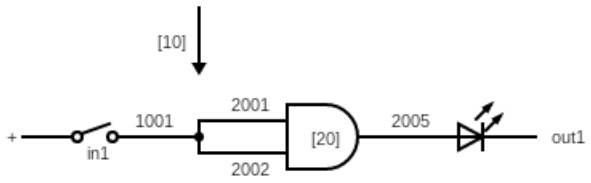


Figure 13: Example on/off light switch using labeling convention.

Figure 13 demonstrates the labeling convention to depict a simple yet over-engineered on/off switch for a single light. Importantly, notice that the junction of three wires (i.e. Split) is considered a component and has been assigned the label [10]. (The vertical arrow accompanying the [10] is to guide the eye, there is not a stray current in the figure.)

2.2 Data Structures

Begin by defining a generic data type called `Wire` with the following structure:

```
Type Wire:
    Identity // 1000 to X
    Pointer  // [1] to [999]
    Value    // 0 or 1
```

The `Identity` is the wire's own identity (1000-X). The `Pointer` field contains the identity of the downstream component touching the wire. The `value` field records the instantaneous voltage in the wire, just 1 or 0 for our purposes. Note there is not a field to store the wire's upstream connection.

Next, consider another generic type called `Element` that contains (i) the specification of a single component, and (ii) all wires touching the component.

```
Type Element:
    Species
    Identity  // [1] to [999]
    A As Wire
    B As Wire
    C As Wire
```

The `Species` field stores a text abbreviation of the component being represented. Previously we decided there should be six of these:

$$\texttt{Species} = \{\texttt{AND}, \texttt{INV}, \texttt{NOR}, \texttt{OR}, \texttt{SPL}, \texttt{XOR}\}$$

For most logic gates, the wires A, B are inputs and C is the output. The INV gate treats A as the only input and ignores wire C. The Split junction treats A as the input and B, C as output.

Note that the `Element` type does not have an extra field to store the state of the component being represented. The output wire (C and/or B) does this job.

2.3 Buttons and Lights

More structure is required to handle interaction with the circuit via buttons and lights. For this we need just one generic data type called `Interact` as follows:

```
Type Interact:
    Species
    Identity // 1000 to X
    Pointer  // [1] to [999]
    Value    // 0 or 1
```

The `Species` field defines which device is being represented. There are three kinds of buttons:

1. `PSH`: stays on when constantly depressed, off when released
2. `TOG`: traditional toggle switch
3. `PLS`: auto-pulse

Figure 7 depicts the symbolic difference between the `PSH` button and the `TOG` button. Meanwhile, there is only one kind of light, designated `LED` for *light-emitting diode*. All together, there are four possibilities for the `Species` field:

$$\texttt{Species} = \{\texttt{PSH}, \texttt{TOG}, \texttt{PLS}, \texttt{LED}\}$$

The `Identity` field applies to the wire used to connect the device to its associated component, and the `Pointer` field is that component's address. The `Value` tracks the state of the interacting device. For buttons, 1 means 'pressed'. For lights, 1 means 'on'.

2.4 Circuit as Data

Using the data types detailed above, three global arrays are used to prepare space for circuit having up to 999 components, 9 buttons, and 9 lights. The total number of wires, i.e. the upper limit in `1000 to X`, is defined to (far) exceed three times the number of components.

```
Shared Component(1 To 999) As Element
Shared Button(1 To 9) As Interact
Shared Light(1 To 9) As Interact
```

There must be a way to define and label components, buttons and lights, along with all wires connecting these. To do this, circuits are built from records containing six pieces of information:

```
NewComponent ->
  Species, Identity,
  Wire A ID, Wire B ID, Wire C ID,
  Comp B Ptr, Comp C Ptr
```

Consistent with the `Element` type, a new component is assigned a species and an identity, along with the identity of any wire touching its terminals.

The final two 'pointer' fields contain the address of any downstream component that is expected to receive information from the component being defined. This redundancy is worth the investment in the sense that the `Wire` type does not need to specify an upstream component address. When there is no downstream component, that pointer is set to -1.

Buttons and lights require less detail in their respective definitions. Following the `Interact` type, we have:

```
NewButton ->
  Species, Identity, Pointer

NewLight ->
  Species, Identity, Pointer
```

On/Off Light Switch

Using the structures above, an entire circuit and all aspects of its diagram can be encoded as data. It takes little to imagine this same scheme can support any combination of components.

Returning to the over-engineered on/off light switch first depicted in Figure 13, it turns out that the same information, including all labels, is contained in four total records:

```
Components:
  ("SPL", 10, 1001, 2001, 2002, 20, 20)
  ("AND", 20, 2001, 2002, 2005, -1, -1)
Interacts:
  (1, "TOG", 1001, 10)
  (1, "LED", 2005, 20)
```

SR Latch

A fully-labeled SR latch circuit, including two inputs and two outputs, is detailed in Figure 14.

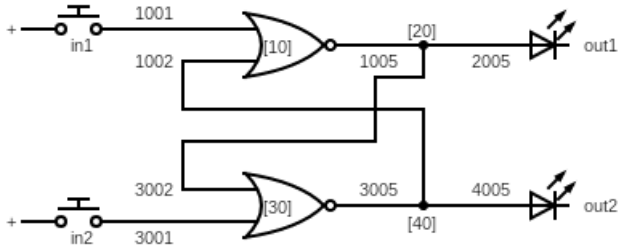


Figure 14: Example SR latch circuit diagram using labeling convention.

Interpreting the SR latch circuit diagram as records of data, we may jot down the following (abbreviated) form having four components and four interfaces:

```
Components:
  ("NOR", 10, 1001, 1002, 1005, -1, 20)
  ("NOR", 30, 3001, 3002, 3005, -1, 40)
  ("SPL", 20, 1005, 3002, 2005, 30, -1)
  ("SPL", 40, 3005, 1002, 4005, 10, -1)
Interacts:
  (1, "PSH", 1001, 10)
  (2, "PSH", 3001, 30)
  (1, "LED", 2005, 20)
  (2, "LED", 4005, 40)
```

2.5 Circuit State

The *state* of the circuit is defined as the ensemble of all `Value` fields among all wires. Note that components themselves not carry state per se, as the `Element` type intentionally lacks a `Value` field. Said another way, the state concerns only the voltages among wires in a circuit, and whatever is happening *inside* a component is irrelevant.

Ground State and Stability

The *ground* state of a circuit occurs when all voltages are off, i.e. the `Value` field for each wire stores 0. As we'll see, some can circuits remain in the ground state when 'powered on'. Other circuits cannot tolerate a static ground state, and change to a stable state as quickly as possible.

Thinking back to the analysis of the SR latch, we now say 'instability of the ground state' is responsible for the state $\{0, 0, 0, 0\}$ being forbidden.

State Record

While the computer has its own way of remembering the state of a circuit, it is still instructive to have a

text readout of the circuit's evolution. For this, define a 'state record' of the format

$$\mathcal{S} = \{\ldots [\texttt{Y}]\,\texttt{ABC}\ldots\}\ ,$$

where [Y] is the identity of a given component, and then the triplet ABC lists the state of that component's A-wire, B-wire, etc.

The complete state record contains a [Y] ABC-like entry for each component. There will be inevitable redundancies in the record, as any one wire often touches two components. The C-variable is vestigial and thus set to 0 in the case of INV.

Intermediate State

Continuing the example of the over-engineered on/off light switch and using the labeling scheme from Figure 13, the ground state of the circuit can be represented as:

$$\mathcal{S}_{off} = \{[10]\,000\,[20]\,000\}$$

The first component [10] represents the junction of three wires joining the switch `in1` to both inputs of the downstream AND gate, component [20].

Closing the switch to turn the light on, it takes little to imagine the final state replaces all 0-bits with 1-bits, as in

$$\mathcal{S}_{on} = \{[10]\,111\,[20]\,111\}\ ,$$

which is surely true, but it's worth going through how that happens despite not yet having detailed the state-updating algorithm.

Immediately after closing the switch but before stability is reached, the circuit (as defined) jumps to the intermediate state

$$\mathcal{S}_{med} = \{[10]\,100\,[20]\,000\}\ ,$$

which means the input to the junction [10] has acknowledged a 'high' signal. Recursing on the intermediate state leads to the on-state after one step.

As it turns out, the intermediate state actually depends on the order in which components are defined in memory, thus the intermediate state is more of a ghost than anything else.

Indeed, if we take the same over-engineered on/off light switch and swap address [10] with address [20], the physically-equivalent circuit is defined:

```
Components:
  ("SPL", 20, 1001, 2001, 2002, 10, 10)
  ("AND", 10, 2001, 2002, 2005, -1, -1)
Interacts:
  (1, "TOG", 1001, 20)
  (1, "LED", 2005, 10)
```

Closing the switch on this version, the intermediate state

$$\mathcal{S}_{med} = \{[10]\,000\,[20]\,100\}$$

arises as one may expect. Unlike the previous case though, the 'on' state does not come next. Instead, a secondary intermediate state gets in the way

$$\mathcal{S}_{sec} = \{[10]\,110\,[20]\,111\}\ ,$$

from which 'on' follows. Clearly then, the number and order of intermediate states depends on how the circuit is registered in memory.

Variability of the intermediate state is directly analogous to effects in real-life circuits that arise from mixing wire lengths, component brands, and so on. Unless the technician makes a significant mistake, intermediate states are often imperceptible and require tools more specialized than the ammeter and voltmeter to detect.

Equilibrium

One way to summarize intermediate state analysis is to say that *equilibrium* corresponds to the final state being reached after all intermediate states have been churned through.

2.6 Main Loop

When it comes to actually executing the simulated circuit, it's necessary to have a real-time algorithm, sometimes called a main loop or *game loop*, that will:

(i) scan for user input (including auto-pulse).

(ii) update the internal state.

(iii) display output and summary.

For the simulated to 'feel real' and achieve equilibrium quickly in response to input, the entirety of the game loop should refresh minimally at 30 Hz, i.e. 30 frames per second.

2.7 Managing Input

During the game loop, an input-polling routine must scan for keystrokes and inform the circuit whether a button has been pressed.

PSH

For buttons of the PSH species, the following pseudo-code does the job:

```
Let i = index of i'th button (1 to 9)
If key(i) pressed {q = 1}
             Else {q = 0}
r = Button(i).Identity // 1000 to X
j = Button(i).Pointer // [1] to [999]
If (r = Component(j).A.Identity) {
  Component(j).A.Value = q}
ElseIf (r = Component(j).B.Identity) {
  Component(j).B.Value = q}
ElseIf (r = Component(j).C.Identity) {
  Component(j).C.Value = q}
```

Note that the input-polling routine changes the circuit state by editing the voltage value in the wire identified by r that connects the button to the component at address j. This behavior is applied to each button type.

TOG

The TOG case is slightly more complicated:

```
Let i = index of i'th button (1 to 9)
If key(i) pressed:
r = Button(i).Identity // 1000 to X
j = Button(i).Pointer // [1] to [999]
If (r = Component(j).A.Identity) {
  q = Component(j).A.Value
  If (q = 0) {Component(j).A.Value = 1}
    Else {Component(j).A.Value = 0}}
ElseIf (r = Component(j).B.Identity) {
  q = Component(j).B.Value
  If (q = 0) {Component(j).B.Value = 1}
    Else {Component(j).B.Value = 0}}
ElseIf (r = Component(j).C.Identity) {
  q = Component(j).C.Value
  If (q = 0) {
    Component(j).C.Value = 1}
    Else {Component(j).C.Value = 0}}
```

Like its sibling, the TOG routine changes the circuit state.

PLS

The PLS button behaves automatically without requiring user interaction. In real time, the component connected to PLS will receive 2/3 seconds of 'high', and then 1/3 seconds of 'low', repeating forever. The overall frequency is 1 Hz. As pseudo-code, this looks like:

```
Let i = index of i'th button (1 to 9)
If key(i) pressed:
t = Timer (to ~ms precision)
t = t - Int(t):
If (t >= 2/3) {q = 1}
         Else {q = 0}
r = Button(i).Identity // 1000 to X
j = Button(i).Pointer // [1] to [999]
If (r = Component(j).A.Identity) {
  Component(j).A.Value = q}
ElseIf (r = Component(j).B.Identity) {
  Component(j).B.Value = q}
ElseIf (r = Component(j).C.Identity) {
  Component(j).C.Value = q}
```

2.8 Updating Internal State

As any input of the circuit is manipulated, the voltage value is immediately changed in the wire leading to the affected component. Propagation stops there momentarily, leaving a certain 'tension' on the components touching active inputs. Control then passes to the subroutine that updates internal state - the true heart of the simulation that we detail now.

Binary Cases

For each binary logic gate k in the circuit, internal state is updated in two steps:

(i) Leading into k, the values from input wires A and B are compared using the proper truth table for that component's species. The resulting 1 or 0 is written to k's output wire C.

(ii) Downstream from k is a different component j whose input A' or B' must be overwritten to reflect the change in C.

Step (i) can be implemented in a fashion captured in the following pseudo-code:

```
a = Component(k).A.Value
b = Component(k).B.Value
Switch Component(k).Species:
Case "AND"
  Component(k).C.Value = BinaryAND(a, b)
Case "NOR"
  Component(k).C.Value = BinaryNOR(a, b)
Case "OR"
  Component(k).C.Value = BinaryOR(a, b)
Case "XOR"
  Component(k).C.Value = BinaryXOR(a, b)
```

Step (ii) proceeds as follows:

```
j = Component(k).C.Pointer
If (j <> -1) {
  g = Component(k).C.Identity
  c = Component(k).C.Value
  If (Component(j).A.Identity = g) {
    Component(j).A.Value = c}
  ElseIf (Component(j).B.Identity = g) {
    Component(j).B.Value = c}}
```

Special Cases

There are two exceptions to the above, namely the INV gate and the Split (junction). Each obeys the same two-step process, however special treatment is needed for A, B, and C if used. Jumping right to pseud-code, we have, for the INV-case:

```
a = Component(k).A.Value
// step (i)
Component(k).B.Value = BinaryINV(a)
// step (ii)
j = Component(k).B.Pointer
If (j <> -1) {
  g = Component(k).B.Identity
  b = Component(k).B.Value
  If (Component(j).A.Identity = g) {
    Component(j).A.Value = b}
  ElseIf (Component(j).B.Identity = g) {
    Component(j).B.Value = b}}
```

Finally, the Split-case needs to update two downstream wires and needs its own:

```
a = Component(k).A.Value
// step (i)
Component(k).B.Value = a
Component(k).C.Value = a
// step (ii)
j = Component(k).B.Pointer
If (j <> -1) {
  g = Component(k).B.Identity
  b = Component(k).B.Value
  If (Component(j).A.Identity = g) {
    Component(j).A.Value = b}
  ElseIf (Component(j).B.Identity = g) {
    Component(j).B.Value = b}}
j = Component(k).C.Pointer
If (j <> -1) {
  g = Component(k).C.Identity
  c = Component(k).C.Value
  If (Component(j).A.Identity = g) {
    Component(j).A.Value = c}
  ElseIf (Component(j).B.Identity = g) {
    Component(j).B.Value = c}}
```

2.9 Output and Summary

With all steps thoroughly detailed, it's straightforward to write source code in a procedural programming language that simulates logic circuit behavior. Using *QB64* as a weapon of choice, the efforts of this study culminate to a single program that executes in a single window as demonstrated in Figure 15.

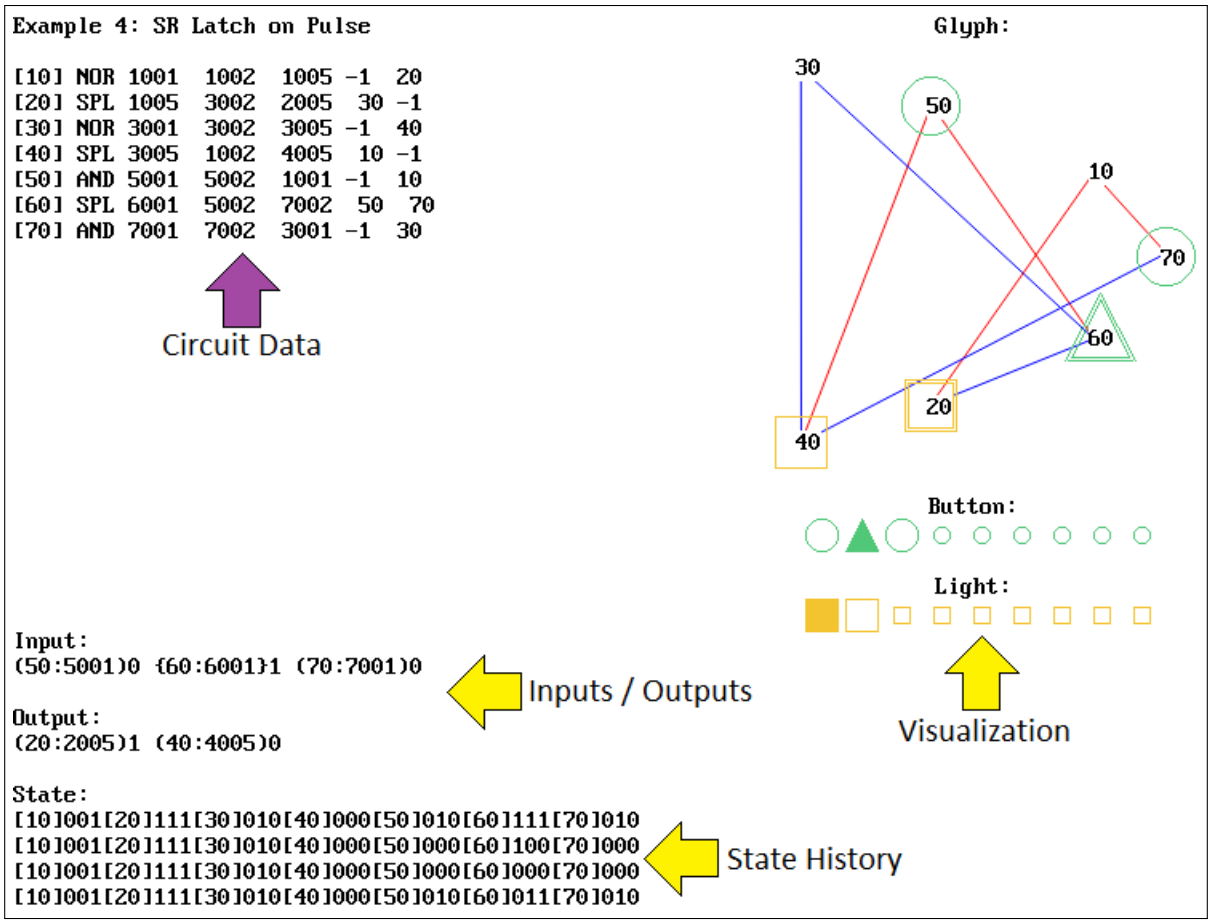


Figure 15: Sample output window showing a simulated logic circuit.

The output window has four distinct regions, one that never changes (purple arrow), and three that receive updates often (yellow arrows).

Circuit Data

In the top-left, the precise layout of the circuit is summarized in a way resembling how the circuit is defined by the programmer. All relevant relationships between components and wires can be read from the summary.

Text Readout

The bottom-left region contains a text readout of all active inputs (buttons) and outputs (lights), followed by the state of the circuit and its history.

Glyph

The right side of the Figure is for real-time visualization. In the top-right we place the so-called *glyph*, which is in essence a *bad* circuit diagram not so much unlike the real wires and components in a real circuit. Nonetheless, the glyph shows the specific relationships among components, and also indicates the voltage throughout.

Buttons and Lights

Under the glyph is a row of button and light indicators, and this is the chief output section of the circuit. On a breadboard, the lights would be a row of LEDs. All colors are decidedly yellow, with a filled square (■) denoting the 'on' state.

The keyboard numerals 1 to 9 are mapped to corresponding buttons `in1` to `in9`. For the example shown, only keys 1 and 3 are of the PSH species as indicated by a circle (●). Button 2, with its triangular indicator (▲), is of the PLS species and self-operates on a timer. Buttons 4 to 9 are unused in the example, and are displayed as diminished placeholders. Not shown (yet) are buttons of the TOG species, which appear as squares (■) when used.

3 Switching and Latching

Much ado has been made of the SR latch, which is arguably the simplest nontrivial circuit that can store either 1 or 0. From there, it takes little to imagine that the latching phenomenon can be stacked to encode complex data structures, record storage, lists of instructions, etc. - in other words, modern computing. Starting simple though, we begin here with a few modifications to the original SR latch.

3.1 SR Latch with Enable

Starting with the original SR latch circuit, suppose the input is piped through a pair of AND gates as shown in Figure 16. In addition, the unused inputs of each AND gate are joined and controlled by an additional PSH button.

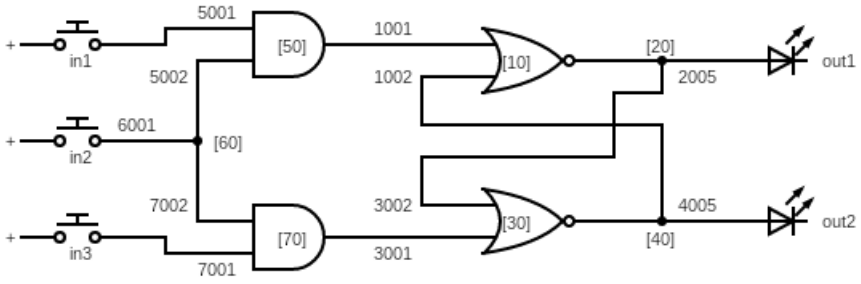


Figure 16: Enabled SR latch circuit.

The purpose of the 'enable' feature is to verify the inputs entering the circuit. That is, nothing happens unless the PSH button `in2` is depressed, thereby enabling the latch to change state.

As for circuit data, we append the specification of the SR latch with three additional components (two ANDs plus a junction), along with the 'enable' button as shown:

```
Components:
  ("NOR", 10, 1001, 1002, 1005, -1, 20)
  ("SPL", 20, 1005, 3002, 2005, 30, -1)
  ("NOR", 30, 3001, 3002, 3005, -1, 40)
  ("SPL", 40, 3005, 1002, 4005, 10, -1)
  ("AND", 50, 5001, 5002, 1001, -1, 10)
  ("SPL", 60, 6001, 5002, 7002, 50, 70)
  ("AND", 70, 7001, 7002, 3001, -1, 30)
Interacts:
  (1, "PSH", 5001, 50)
  (2, "PSH", 6001, 60)
  (3, "PSH", 7001, 70)
  (1, "LED", 2005, 20)
  (2, "LED", 4005, 40)
```

3.2 SR Latch on Pulse

The circuit represented in Figure 15 is another modified SR latch, called the 'pulsed' SR latch. This is identical to the enabled SR latch, except the manual PSH enable button is replaced by an automatic PLS species. This amounts to the following edit to the `Interacts` section:

```
Interacts:
  (2, "PLS", 6001, 60)
```

3.3 D Latch

An elegant modification to the enabled SR latch is the so-called D latch, which conjoins the two circuit inputs while inverting one of them as shown in Figure 17, or eqivalently as data:

```
Components:
  ("NOR", 1, 1001, 1002, 1005, -1, 2)
  ("SPL", 2, 1005, 3002, 2005, 3, -1)
  ("NOR", 3, 3001, 3002, 3005, -1, 4)
  ("SPL", 4, 3005, 1002, 4005, 1, -1)
  ("AND", 5, 5001, 5002, 1001, -1, 1)
  ("SPL", 6, 6001, 5002, 7002, 5, 7)
  ("AND", 7, 7001, 7002, 3001, -1, 3)
  ("SPL", 8, 8001, 7001, 9001, 7, 9)
  ("INV", 9, 9001, 5001, 0, 5, -1)
Interacts:
  (1, "PSH", 8001, 8)
  (2, "PSH", 6001, 6)
  (1, "LED", 2005, 2)
  (2, "LED", 4005, 4)
```

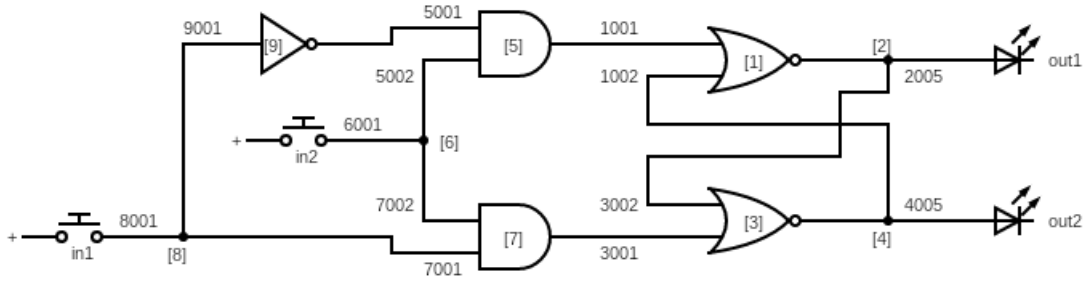


Figure 17: D latch circuit.

By convention, the outputs of the circuit labeled `out1`, `out2` are known as Q and Q', respectively. Because the output values are always in disagreement, the 'main' output is delegated to Q, whereas Q' is typically downplayed as Q's opposite. The same convention says the input `in1` is designated D (think D for 'data'), and the input in2 is designated E for 'enable'.

D Latch Analysis

Analysis of the D latch is slightly more complicated than that of the plain SR latch, but we can play the same state-listing game to understand it's behavior. Begin by writing a generic state vector for the D latch, namely

$$\mathcal{S}_j = \left\{D_j, E_j, Q_j, Q'_j\right\} \ .$$

Now, one could use a pen-and-paper method to deduce the allowed states of the D-latch. On the other hand, since we're in this to build a simulation, it should be just as good to construct the circuit and perform testing in software space. Regardless of how it is done, one should find, for allowed states:

$$\begin{aligned}
\mathcal{S}_1 &= \{0,0,0,1\} \\
\mathcal{S}_2 &= \{0,0,1,0\} \\
\mathcal{S}_5 &= \{0,1,0,1\} \\
\mathcal{S}_9 &= \{1,0,0,1\} \\
\mathcal{S}_{10} &= \{1,0,1,0\} \\
\mathcal{S}_{14} &= \{1,1,1,0\}
\end{aligned}$$

To make sense of these, note that the pair $\mathcal{S}_1$, $\mathcal{S}_2$ begin the same but have the output states swapped. The same can be said for the pair $\mathcal{S}_9$, $\mathcal{S}_{10}$. All four of these states have the E-input set to 0, meaning the circuit is not enabled, thus the state of Q, Q' remain fixed regardless of D's value.

The states $\mathcal{S}_5$, $\mathcal{S}_{14}$ correspond to $E = 1$, thus the circuit is free to exhibit SR-like behavior. When enabled, setting $D = 1$ causes $Q = 1$, whereas $D = 0$ causes $Q = 0$. All of this is summarized in the following D latch truth table:

D Latch			
D	E	Q	Q'
0	0	Q	Q'
1	0	Q	Q'
0	1	0	1
1	1	1	0

Note finally that the state $\mathcal{S}_0 = \{0,0,0,0\}$ is not supported in any SR latch or D latch. When a latch-containing circuit is initially turned on from the ground state, latches quickly find their way out of $\mathcal{S}_0$.

4 Binary Addition

The task of adding two integers can be automated using logic circuits. Given the binary nature of logic gates, numbers involved in calculations addition must be translated to a base-two format before touching a physical circuit.

Starting with the simplest possible case, consider the addition problem

$$A + B = C \,,$$

where A and B can each be either of 0 or 1. This is a simple-enough problem to list all possible outcomes, which are:

$$\begin{aligned}
0 + 0 &= 00 \\
0 + 1 &= 01 \\
1 + 0 &= 01 \\
1 + 1 &= 10
\end{aligned}$$

The output C is written in two-digit format, and the digits are named, for perhaps obvious reasons, *carry* and *sum*, respectively:

$$C = (\text{Carry}, \text{Sum})$$

4.1 Half Adder

The scenario indulged so far is called the *half adder*, and the task now is to capture its behavior in a circuit. Without knowing the components just yet, we can nonetheless see that there should be two inputs and two outputs. Further, we can write a truth table for how the circuit ought to behave:

Half Adder			
A	B	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

From this, the right-hand columns in isolation read

$$\begin{aligned} \text{Carry} &= \{0,0,0,1\} \\ \text{Sum} &= \{0,1,1,0\} \ , \end{aligned}$$

which are identical to the output columns of the AND and XOR gates, respectively:

$$\begin{aligned} \text{AND} &= \{0,0,0,1\} \\ \text{XOR} &= \{0,1,1,0\} \ . \end{aligned}$$

Evidently the truth table of the circuit maps cleanly to our arsenal of components. Thus, let us propose the half adder circuit take the form depicted in Figure 18.

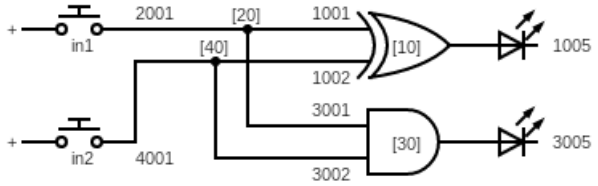


Figure 18: Half adder circuit.

As circuit data, the half adder is captured by:

```
Components:
  ("XOR", 10, 1001, 1002, 1005, -1, -1)
  ("SPL", 20, 2001, 1001, 3001, 10, 30)
  ("AND", 30, 3001, 3002, 3005, -1, -1)
  ("SPL", 40, 4001, 1002, 3002, 10, 30)
Interacts:
  (1, "PSH", 2001, 20)
  (2, "PSH", 4001, 40)
  (1, "LED", 1005, 10)
  (2, "LED", 3005, 30)
```

4.2 Full Adder

While the half adder is perfectly suited for adding a pair of one-bit numbers, it would be nice to perform addition on two-, three-, four-bit numbers and so on. From the outset, we want to do this *without* having to reason out each circuit from first principles.

To break down the task we speak of the *full adder*, which is a streamlining of the half adder suited for integration into larger circuits. In particular, we will see that addition of larger-bit numbers is accomplished by daisy-chaining identical copies of the full adder circuit in a particular arrangement.

To begin, note that the half adder never attains the output state $C = (1,1)$, which is to say the 'carry' and 'sum' values are never simultaneously high. For such a state to be attainable, a third input, thus a third button is needed, and here is the beautiful part: the third input shall be the 'carry' value from a so-far unmentioned adder-like circuit that is upstream of the one being discussed. It's purpose is to add 1 to the sum being calculated.

Without knowing which components to use or how they're arranged, we can still work out the truth table for the full adder:

Full Adder				
`in1`	`in2`	Carry-In	Carry-Out	Sum
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

When the Carry-In (synonymous with `in3`) value is set 'low', the full adder does exactly what the half adder does. With Carry-In set to 'high', we consult the bottom half of the truth table to jot down

$$\begin{aligned} \text{Carry-Out} &= \{0,1,1,1\} \\ \text{Sum} &= \{1,0,0,1\} \ . \end{aligned}$$

Comparing these to the lexicon of logic gates, we find

$$\begin{aligned} \text{OR} &= \{0, 1, 1, 1\} \\ \text{XNOR} &= \{1, 0, 0, 1\} \ , \end{aligned}$$

which is a little troublesome due to the presence of XNOR.

To skirt the problem we exploit the XNOR-equivalent arrangement depicted in Figure 8. Ignoring the pair of INV gates momentarily, we see that the bulk of the XNOR-equivalent circuit consists of a pair of AND gates leading to an OR gate, and that is the motif we shall borrow for the full adder.

The exact marriage of the half adder and the Franken-XNOR is a bit tricky to arrange, but nonetheless the full adder circuit is finished in Figure 19. An extra NOR gate has been slipped into the circuit as a necessity, and in hindsight of this we see that the full adder really is a configuration of two half adders.

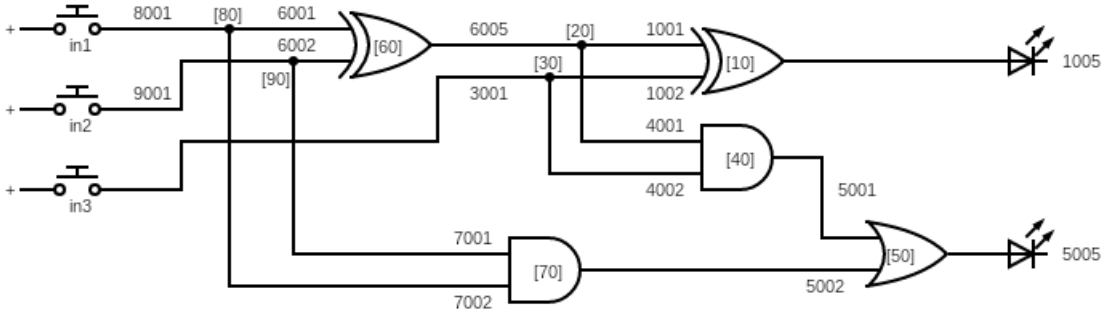


Figure 19: Full adder circuit.

Equivalently, the full adder circuit is specified by the following data:

```
Components:
  ("XOR", 10, 1001, 1002, 1005, -1, -1)
  ("SPL", 20, 6005, 1001, 4002, 10, 40)
  ("SPL", 30, 3001, 1002, 4001, 10, 40)
  ("AND", 40, 4001, 4002, 5001, -1, 50)
  ("OR", 50, 5001, 5002, 5005, 10, 30)
  ("XOR", 60, 6001, 6002, 6005, -1, 20)
  ("AND", 70, 7001, 7002, 5002, -1, 50)
  ("SPL", 80, 8001, 6001, 7002, 60, 70)
  ("SPL", 90, 9001, 6002, 7001, 60, 70)
Interacts:
  (1, "PSH", 8001, 80)  // in1
  (2, "PSH", 9001, 90)  // in2
  (3, "PSH", 3001, 30)  // carry-in
  (1, "LED", 5005, 50)  // carry-out
  (2, "LED", 1005, 10)  // sum
```

4.3 Two Bit Adder

Having so carefully prepared the full adder circuit to play nice among similar structures, the addition of higher-bit numbers becomes a matter of linking a number of identical adder circuits.

The simplest application is the *two bit adder*, which as the name suggests, adds a pair of two-bit binary numbers. Such a circuit contains five total inputs: a pair for each input, plus a carry-in bit. There are three outputs: one pair for a two-bit result, plus a carry-out bit. Letting the circuit diagram to do the rest of the talking, the two bit adder is depicted in Figure 20.

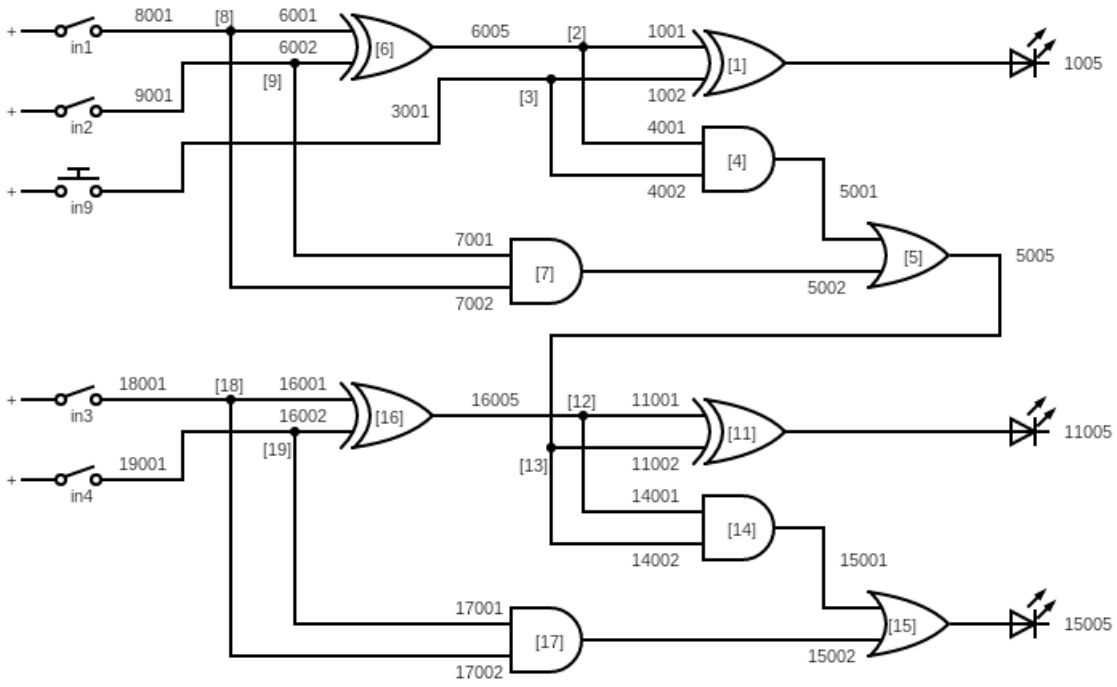


Figure 20: Two bit adder circuit.

Note that the four 'regular' inputs to the circuit are buttons of the TOG species, where as per usual, the carry-in bit is represented by a PSH button. The whole diagram and all of its labeling is equivalent to the circuit data:

```
Components:
  ("XOR", 1, 1001, 1002, 1005, -1, -1)
  ("SPL", 2, 6005, 1001, 4002, 1, 4)
  ("SPL", 3, 3001, 1002, 4001, 1, 4)
  ("AND", 4, 4001, 4002, 5001, -1, 5)
  ("OR", 5, 5001, 5002, 5005, -1, 13)
  ("XOR", 6, 6001, 6002, 6005, -1, 2)
  ("AND", 7, 7001, 7002, 5002, -1, 5)
  ("SPL", 8, 8001, 6001, 7002, 6, 7)
  ("SPL", 9, 9001, 6002, 7001, 6, 7)
  ("XOR", 11, 11001, 11002, 11005, -1, -1)
  ("SPL", 12, 16005, 11001, 14002, 11, 14)
  ("SPL", 13, 5005, 11002, 14001, 11, 14)
  ("AND", 14, 14001, 14002, 15001, -1, 15)
  ("OR", 15, 15001, 15002, 15005, -1, -1)
  ("XOR", 16, 16001, 16002, 16005, -1, 12)
  ("AND", 17, 17001, 17002, 15002, -1, 15)
  ("SPL", 18, 18001, 16001, 17002, 16, 17)
  ("SPL", 19, 19001, 16002, 17001, 16, 17)
Interacts:
  (9, "PSH", 3001, 3)    // carry in
  (1, "TOG", 8001, 8)    // a0
  (2, "TOG", 9001, 9)    // b0
  (3, "TOG", 18001, 18)  // a1
  (4, "TOG", 19001, 19)  // b1
  (1, "LED", 15005, 15)  // carry out
  (2, "LED", 11005, 11)  // sum1
  (3, "LED", 1005, 1)    // sum0
```

Two Bit Adder Analysis

Having five total inputs, there are $2^5 = 32$ distinct states of the two bit adder. The outputs are limited to $2^3 = 8$ possibilities though, thus there is significant non-uniqueness in the result as expected for addition. For this reason, coming up with a complete truth table for such a circuit is a redundant thankless chore to be avoided.

Instead, it's quite easy to predict on paper how the circuit ought to appear. The sum handled by the two bit adder can be represented by

$$N_1 + N_2 + C_{in} = C_{out} + S \,,$$

where the left side is

$$\begin{aligned} N_1 &= \{00, 01, 10, 11\} \\ N_2 &= \{00, 01, 10, 11\} \\ C_{in} &= \{0, 1\} \,, \end{aligned}$$

and the right is

$$\begin{aligned} C_{out} &= \{0, 1\} \\ S &= \{00, 01, 10, 11\} \,. \end{aligned}$$

For an example, consider the case

$$N_1 = 10$$
$$N_2 = 01$$
$$C_{in} = 0\,,$$

which represents the addition problem

$$2 + 1 + 0 = C_{out} + S\,.$$

The right side is obviously 3, which comes in the form $(0, 1, 1)$. That is, the carry-out bit is 0, and the sum contains 3 in binary, namely 11.

The same problem is represented in Figure 21 using buttons and lights. In particular, buttons 1, 3 correspond to N_1, and buttons 2, 4 correspond to N_2. The carry-in digit is delegated to the right as button 9.

Figure 21: The sum $1 + 2 + 0 = 3$.

Modifying the example, we can set the carry-in bit to 1 instead of 0, which kicks the output state to 4, represented by 100 in binary. This means the S-field contains 00, and the carry-out bit is set to 1 as depicted in Figure 22.

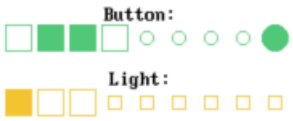


Figure 22: The sum $1 + 2 + 1 = 4$.

4.4 More Bit Adders

As demonstrated while building the two bit adder, adding yet more bits is a matter of linking yet more full adder circuits. As the adder circuits grow, precisely *how* the inputs correspond to the outputs needs further elaboration that is reserved for the four bit case below.

Three Bits

Without laboring any details and skipping to a result, Figure 23 displays a portion of the QB64 output window while running the three-bit adder.

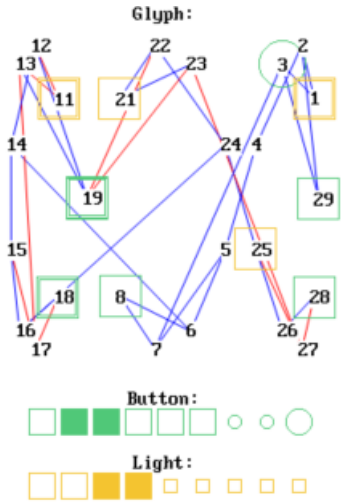


Figure 23: Partial output window showing a three bit adder circuit.

Four Bits

Figure 24 depicts the four bit adder circuit.

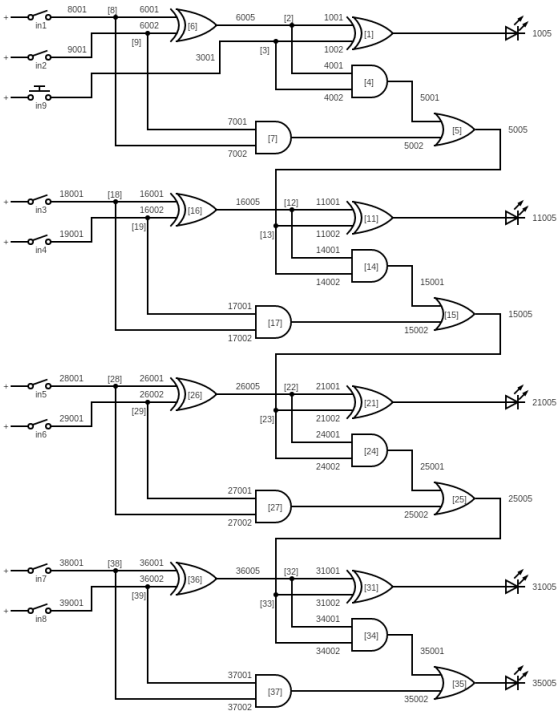


Figure 24: Four bit adder circuit.

Having nine total inputs, the four bit adder circuit uses the full range of available buttons. Like the two bit adder, we can begin analysis by writing

$$N_1 + N_2 + C_{in} = C_{out} + S\,,$$

where, now, the left side is

$$N_1 = \{0000, 0001, \ldots, 1111\}$$
$$N_2 = \{0000, 0001, \ldots, 1111\}$$
$$C_{in} = \{0, 1\}\ ,$$

and the right is

$$C_{out} = \{0, 1\}$$
$$S = \{0000, 0001, \ldots, 1111\}\ .$$

Particularly, the numbers N_1, N_2, and S are each one of sixteen possibilities, i.e. the integers 0 to 15. The greatest result the four bit adder can produce is 31, which in binary is 11111.

With the carry digit reserved to button 9 as usual, the circuit receives input as follows: The digits of N_1 are inputted using the keys 1, 3, 5, 7, and the digits of N_2 are inputted using the keys 2, 4, 6, 8. Keeping in mind the context is base two, this further means:

- Buttons 1, 2 are the **ones** column.
- Buttons 3, 4 are the **tens** column.
- Buttons 5, 6 are the **hundreds** column.
- Buttons 7, 8 are the **thousands** column.

For example, suppose we want to compute the sum

$$9 + 13 + 0 = 22\ .$$

Breaking this down, we have

$$N_1 = 1001$$
$$N_2 = 1101$$
$$C_{in} = 0$$

Since we know the answer is 22, we also have that

$$C_{out} + S = 10110\ ,$$

where the right side is 22 in binary.

As far as circuit input goes, we consult the expressions of N_1, N_2, and, reading right to left, discern that we press buttons 1, 2, 6, 7, 8 while leaving the rest alone. By doing this, the lights that go 'high' should be 1, 3, 4.

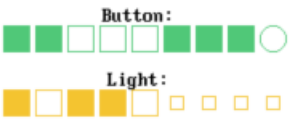


Figure 25: Partial output window demonstrating the sum $9 + 13 = 22$.

Containing the same information depicted in Figure 24, the four bit adder circuit data is listed below for completenss:

```
Components:
  ("XOR", 1, 1001, 1002, 1005, -1, -1)
  ("SPL", 2, 6005, 1001, 4002, 1, 4)
  ("SPL", 3, 3001, 1002, 4001, 1, 4)
  ("AND", 4, 4001, 4002, 5001, -1, 5)
  ("OR", 5, 5001, 5002, 5005, -1, 13)
  ("XOR", 6, 6001, 6002, 6005, -1, 2)
  ("AND", 7, 7001, 7002, 5002, -1, 5)
  ("SPL", 8, 8001, 6001, 7002, 6, 7)
  ("SPL", 9, 9001, 6002, 7001, 6, 7)
  ("XOR", 11, 11001, 11002, 11005, -1, -1)
  ("SPL", 12, 16005, 11001, 14002, 11, 14)
  ("SPL", 13, 5005, 11002, 14001, 11, 14)
  ("AND", 14, 14001, 14002, 15001, -1, 15)
  ("OR", 15, 15001, 15002, 15005, -1, 23)
  ("XOR", 16, 16001, 16002, 16005, -1, 12)
  ("AND", 17, 17001, 17002, 15002, -1, 15)
  ("SPL", 18, 18001, 16001, 17002, 16, 17)
  ("SPL", 19, 19001, 16002, 17001, 16, 17)
  ("XOR", 21, 21001, 21002, 21005, -1, -1)
  ("SPL", 22, 26005, 21001, 24002, 21, 24)
  ("SPL", 23, 15005, 21002, 24001, 21, 24)
  ("AND", 24, 24001, 24002, 25001, -1, 25)
  ("OR", 25, 25001, 25002, 25005, -1, 33)
  ("XOR", 26, 26001, 26002, 26005, -1, 22)
  ("AND", 27, 27001, 27002, 25002, -1, 25)
  ("SPL", 28, 28001, 26001, 27002, 26, 27)
  ("SPL", 29, 29001, 26002, 27001, 26, 27)
  ("XOR", 31, 31001, 31002, 31005, -1, -1)
  ("SPL", 32, 36005, 31001, 34002, 31, 34)
  ("SPL", 33, 25005, 31002, 34001, 31, 34)
  ("AND", 34, 34001, 34002, 35001, -1, 35)
  ("OR", 35, 35001, 35002, 35005, -1, -1)
  ("XOR", 36, 36001, 36002, 36005, -1, 32)
  ("AND", 37, 37001, 37002, 35002, -1, 35)
  ("SPL", 38, 38001, 36001, 37002, 36, 37)
  ("SPL", 39, 39001, 36002, 37001, 36, 37)
Interacts:
  (9, "PSH", 3001, 3)    // carry in
  (1, "TOG", 8001, 8)    // a0
  (2, "TOG", 9001, 9)    // b0
  (3, "TOG", 18001, 18)  // a1
  (4, "TOG", 19001, 19)  // b1
  (5, "TOG", 28001, 28)  // a2
  (6, "TOG", 29001, 29)  // b2
  (7, "TOG", 38001, 38)  // a3
  (8, "TOG", 39001, 39)  // b3
  (1, "LED", 35005, 35)  // carry out
  (2, "LED", 31005, 31)  // sum3
  (3, "LED", 21005, 21)  // sum2
  (4, "LED", 11005, 11)  // sum1
  (5, "LED", 1005, 1)    // sum0
```

5 Edge Detection

Here we address an important and subtle phenomenon that occurs in real-world circuits called *edge detection*. Consider a circuit where, somewhere in a given wire (either caused by a button or a component), the value (perhaps rapidly) turns on and off as depicted in Figure 26.

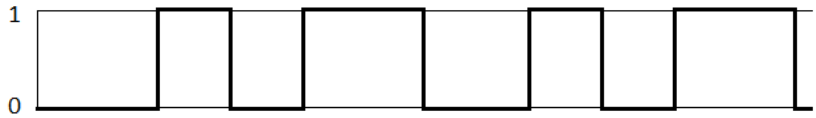


Figure 26: Changing value in a wire.

Next, suppose we are interested in capturing *only* the moments when the value changes from 0 to 1. To illustrate, Figure 27 overlays a colored edge onto the so-called 'rising edges' of the previous sketch as shown.

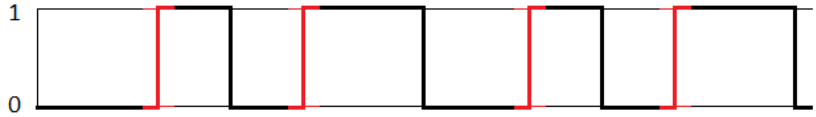


Figure 27: Rising edge overlay.

A proper *edge detector* is designed to convert rising edges into short pulses. Continuing the example on hand, this amounts to turning the red portion in the previous sketch into pulses of their own as depicted in Figure 28.

Figure 28: Pulses derived from rising edges.

5.1 Edge Detector Circuit

Astonishingly, it's possible to build an edge detector circuit from a strange but simple configuration of logic gates. The idea is to couple both inputs to an AND gate, but to invert one of them as shown in Figure 29.

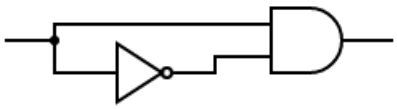


Figure 29: Edge detector circuit.

The key is to realize that the 'evaluation time' of a given logic gate is not instant, and the presence of the INV gate momentarily lags down the signal reaching the second AND gate input. From this we see suddenly switching the input to 1 causes a brief moment where both inputs on the AND gate are 1, and a quick high pulse is emitted.

Extended Edge Detector

The particular make and model of the INV gate chosen for an edge detector circuit, along with the quality of the wires involved, has direct influence over its proper functionality. That is, some chips evaluate very quickly, sending a very narrow pulse downstream that is too slender to use.

This is compensated for by adding an additional INV gates to the circuit in even pairs. as shown in Figure 30.

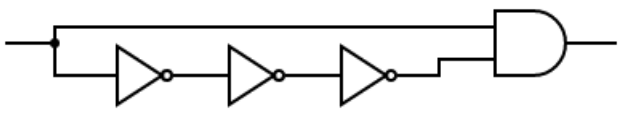


Figure 30: Extended edge detector circuit.

The extra pair of INV gates helps to slow the signal reaching the low AND gate input, thus making the detected edge last longer. As it turns out, this idea must also be exploited in simulation because the order in which components are defined has an effect on internal state. In order to guarantee a 'proper' pulse from the AND gate, the safer option is the extended edge detector. Our version of the extended edge detector takes the form:

```
Components:
  ("SPL", 1, 1001, 2002, 3001, 2, 3)
  ("INV", 2, 2002, 4001, 0, 4, -1)
  ("AND", 3, 3001, 3002, 3010, -1, -1)
  ("INV", 4, 4001, 5001, 0, 5, -1)
  ("INV", 5, 5001, 3002, 0, 3, -1)
Interacts:
  (1, "PSH", 1001, 1)
  (1, "LED", 3010, 3)
```

It's worth having a precise look at the state of the extended edge detector. Starting with the ground state, we first have

$$\mathcal{S}_{off} = \{[1]\,000\,[2]\,000\,[3]\,000\,[4]\,000\,[5]\,000\}\ ,$$

which is in fact unstable, and the circuit jumps quickly to the rest state:

$$\mathcal{S}_{rest} = \{[1]\,000\,[2]\,010\,[3]\,010\,[4]\,100\,[5]\,010\}$$

From here, let us examine what happens by pressing and holding button 1. Doing so causes the intermediate state

$$\mathcal{S}_{med} = \{[1]\,100\,[2]\,010\,[3]\,010\,[4]\,100\,[5]\,010\}\ .$$

Immediately after this, the circuit jumps to the curious intermediate state

$$\mathcal{S}_{edge} = \{[1]\,111\,[2]\,100\,[3]\,101\,[4]\,010\,[5]\,100\}\ ,$$

which momentarily flashes the circuit's only light. This indicates that the so-called edge has been successfully detected and any downstream components are now aware. Note the off-color behavior of the AND gate [3] at this moment: the inputs disagree, but the output is high.

Finally, with the button still pushed and held, the state becomes

$$\mathcal{S}_{hold} = \{[1]\,111\,[2]\,100\,[3]\,100\,[4]\,010\,[5]\,100\}\ .$$

Letting go of the button, the circuit returns to the rest state.

5.2 D Flip-Flop

Quickly reviewing the story of bit storage, we know that a clever use of feedback gives rise to the SR latch, and then the inputs to the circuit can be edited to give the 'enabled' SR latch. Modifying the input further produces the elegant D latch.

Here we continue the SR latch evolution story by incorporating edge detection. In particular, begin with the D latch and place an extended edge detector circuit immediately after the 'enable' button as shown in Figure 31.

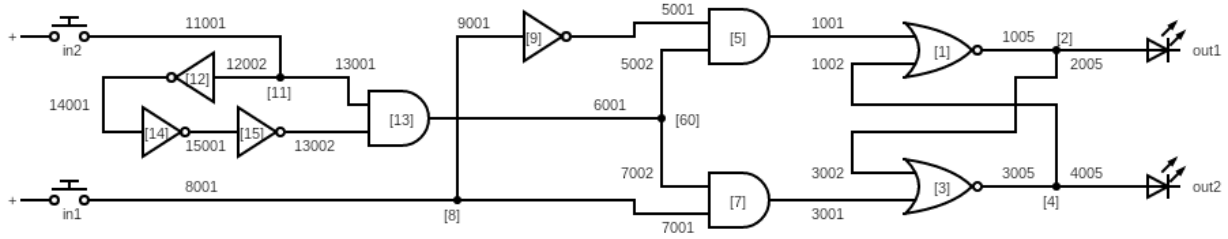


Figure 31: D flip-flop circuit.

This construct is called a *D flip-flop*. Different from the D latch, the 'enable' function only triggers on the leading edge of the enable button (`in2`). That is, the circuit is only enabled in pulses, not continuously. Internally, the D flip-flop does the same job as the D latch and obeys the same truth table.

The corresponding data for the D flip-flop circuit is as follows:

```
Components:
  ("NOR", 1, 1001, 1002, 1005, -1, 2)
  ("SPL", 2, 1005, 3002, 2005, 3, -1)
  ("NOR", 3, 3001, 3002, 3005, -1, 4)
  ("SPL", 4, 3005, 1002, 4005, 1, -1)
  ("AND", 5, 5001, 5002, 1001, -1, 1)
  ("SPL", 6, 6001, 5002, 7002, 5, 7)
  ("AND", 7, 7001, 7002, 3001, -1, 3)
  ("SPL", 8, 8001, 7001, 9001, 7, 9)
```

```
  ("INV", 9, 9001, 5001, 0, 5, -1)
  ("SPL", 11, 11001, 12002, 13001, 12, 13)
  ("INV", 12, 12002, 14001, 0, 14, -1)
  ("AND", 13, 13001, 13002, 6001, -1, 6)
  ("INV", 14, 14001, 15001, 0, 15, -1)
  ("INV", 15, 15001, 13002, 0, 13, -1)
Interacts:
  (1, "PSH", 8001, 8)    // D
  (2, "PSH", 11001, 11)  // E
  (1, "LED", 2005, 2)    // Q
  (2, "LED", 4005, 4)    // Q'
```

D Flip-Flop Symbol

As it turns out, the D flip-flop is popular enough to have its own symbol as shown in Figure 32. The input D, along with outputs Q, Q' are labeled accordingly (with Q' equivalent to $\overline{Q}$). The 'set' (S) and 'reset' (R) pins can be ignored for our purposes, which leaves the strange triangular symbol on the left for the 'enable' function. This input is by convention called the *clock*.

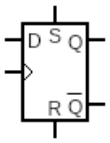


Figure 32: D flip-flop symbol.

6 Counters

The D flip-flop can be put to use in a variety of interesting ways, and we scratch the surface here by building a few counters.

6.1 One Bit Counter

Begin with the D flip-flip, and perform the strange surgery of removing the D-input, and re-routing its wire to the Q' output as shown in Figure 33. Also ignore the Q-output for a moment and only look at Q'.

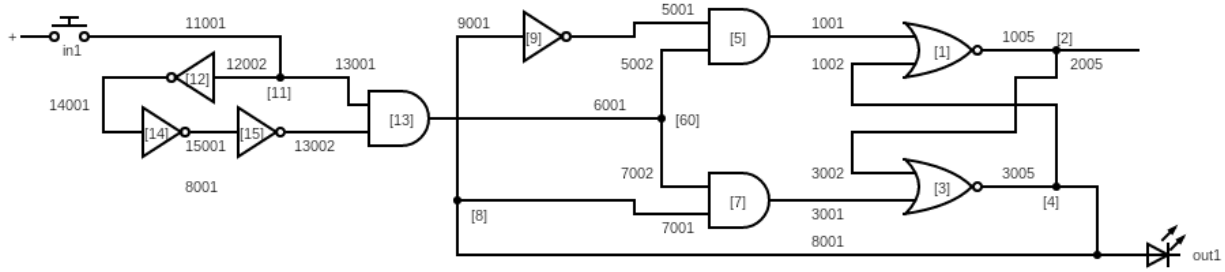


Figure 33: One bit counter circuit.

Such a modified D flip-flop is called a *one bit counter*. To 'count' a one bit number isn't very exciting, as the circuit merely amounts to a button that turns a light on or off. The subtlety is, the circuit only responds to the moment the button is pressed, i.e. on the rising edge of the input.

Evolution of State

Slightly less interesting than the states allowed in circuit is the order in which they occur. From the ground state $\mathcal{S}_{00}$, pushing and holding the input leads to the state $\mathcal{S}_{11}$. Releasing the button gives $\mathcal{S}_{01}$. From there, pressing the button again gives $\mathcal{S}_{10}$, and releasing it a second time gets back to $\mathcal{S}_{00}$. In summary, we find the sequence of states to be:

$$\mathcal{S}_{00} \to \mathcal{S}_{11} \to \mathcal{S}_{01} \to \mathcal{S}_{10} \to \mathcal{S}_{00} \to \dots$$

In other words, the light stays on after the button is released. It takes a second push to turn the light off.

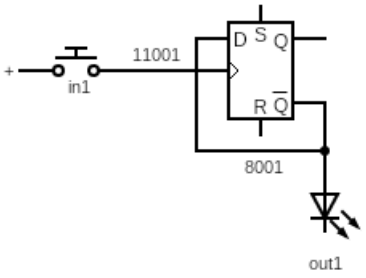


Figure 34: One bit counter circuit in terms of D flip-flop symbol.

In terms of the D flip-flop symbol, the one bit counter is represented in Figure 34. The equivalent as circuit data reads:

```
Components:
```

```
  ("NOR", 1, 1001, 1002, 1005, -1, 2)
  ("SPL", 2, 1005, 3002, 2005, 3, -1)
  ("NOR", 3, 3001, 3002, 3005, -1, 4)
  ("SPL", 4, 3005, 1002, 8001, 1, 8)
  ("AND", 5, 5001, 5002, 1001, -1, 1)
  ("SPL", 6, 6001, 5002, 7002, 5, 7)
  ("AND", 7, 7001, 7002, 3001, -1, 3)
  ("SPL", 8, 8001, 7001, 9001, 7, 9)
  ("INV", 9, 9001, 5001, 0, 5, -1)
  ("SPL", 11, 11001, 12002, 13001, 12, 13)
  ("INV", 12, 12002, 14001, 0, 14, -1)
  ("AND", 13, 13001, 13002, 6001, -1, 6)
  ("INV", 14, 14001, 15001, 0, 15, -1)
  ("INV", 15, 15001, 13002, 0, 13, -1)
Interacts:
  (1, "PSH", 11001, 11)  // Clock
  (1, "LED", 8001, 8)    // Q'
```

6.2 Two Bit Counter

One can straightforwardly build a two bit counter by linking a pair of one bit counters in series as shown in Figure 35.

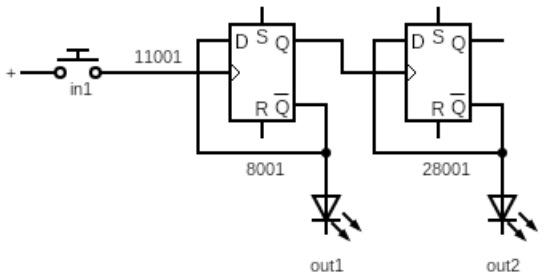


Figure 35: Two bit counter circuit in terms of D flip-flop symbol.

For clarity we have employed the symbolic notation for the D flip-flop, but keep in mind the circuit being described isn't precisely trivial. For completeness, the corresponding circuit data reads:

```
Components:
  ("NOR", 1, 1001, 1002, 1005, -1, 2)
  ("SPL", 2, 1005, 3002, 31001, 3, 31)
  ("NOR", 3, 3001, 3002, 3005, -1, 4)
  ("SPL", 4, 3005, 1002, 8001, 1, 8)
  ("AND", 5, 5001, 5002, 1001, -1, 1)
  ("SPL", 6, 6001, 5002, 7002, 5, 7)
  ("AND", 7, 7001, 7002, 3001, -1, 3)
  ("SPL", 8, 8001, 7001, 9001, 7, 9)
  ("INV", 9, 9001, 5001, 0, 5, -1)
  ("SPL", 11, 11001, 12002, 13001, 12, 13)
  ("INV", 12, 12002, 14001, 0, 14, -1)
  ("AND", 13, 13001, 13002, 6001, -1, 6)
  ("INV", 14, 14001, 15001, 0, 15, -1)
  ("INV", 15, 15001, 13002, 0, 13, -1)
  ("NOR", 21, 21001, 21002, 21005, -1, 22)
  ("SPL", 22, 21005, 23002, 22005, 23, -1)
  ("NOR", 23, 23001, 23002, 23005, -1, 24)
  ("SPL", 24, 23005, 21002, 28001, 21, 28)
  ("AND", 25, 25001, 25002, 21001, -1, 21)
  ("SPL", 26, 26001, 25002, 27002, 25, 27)
  ("AND", 27, 27001, 27002, 23001, -1, 23)
  ("SPL", 28, 28001, 27001, 29001, 27, 29)
  ("INV", 29, 29001, 25001, 0, 25, -1)
  ("SPL", 31, 31001, 32002, 33001, 32, 33)
  ("INV", 32, 32002, 34001, 0, 34, -1)
  ("AND", 33, 33001, 33002, 26001, -1, 26)
  ("INV", 34, 34001, 35001, 0, 35, -1)
  ("INV", 35, 35001, 33002, 0, 33, -1)
Interacts:
  (1, "PSH", 11001, 11)
  (1, "LED", 8001, 8)
  (2, "LED", 28001, 28)
```

The two bit counter still operates on a single input. The so-far unused Q output of the first counter is sent as the input to the next, and the state of the counter is read from the pair Q' outputs.

The two bit counter's qualitative behavior extends that of the one bit, namely that the output state changes only when the input is initially pressed. Starting from the ground state and listing only the output values, the two bit counter yields the output sequence

$$00 \to 01 \to 10 \to 11 \to 00 \to \dots ,$$

or in base ten,

$$0 \to 1 \to 2 \to 3 \to 0 \to \dots .$$

That is, the circuit counts from 0 to 3 and repeats forever.

6.3 Four Bit Counter

Given the plug-and-play flexibility the one bit adder circuit, it's very easy to extend the two bit counter to add three, four, or any number of bits. Granted, it becomes more difficult to manage the such circuits, which is why D flip-flip circuits are often printed to chips.

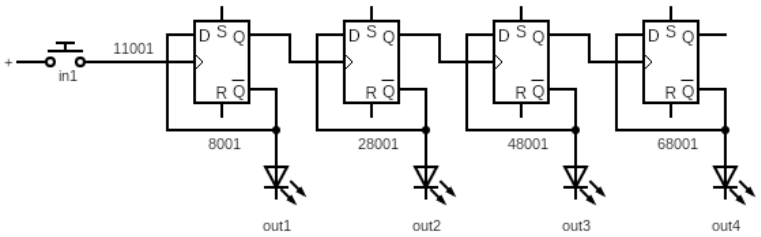


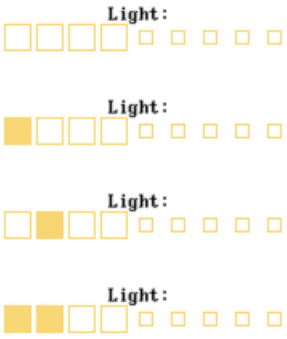
Figure 36: Four bit counter circuit.

A four bit counter has been prepared and is shown in Figure 36, which looks rather tame in terms of the D flip-flop symbol. Such a device uses four outputs to cycle through the integers 0000 to 1111, also known as zero to fifteen. As circuit data, this one is a monster:

```
Components:
  ("NOR", 1, 1001, 1002, 1005, -1, 2)
  ("SPL", 2, 1005, 3002, 31001, 3, 31)
  ("NOR", 3, 3001, 3002, 3005, -1, 4)
  ("SPL", 4, 3005, 1002, 8001, 1, 8)
  ("AND", 5, 5001, 5002, 1001, -1, 1)
  ("SPL", 6, 6001, 5002, 7002, 5, 7)
  ("AND", 7, 7001, 7002, 3001, -1, 3)
  ("SPL", 8, 8001, 7001, 9001, 7, 9)
  ("INV", 9, 9001, 5001, 0, 5, -1)
  ("SPL", 11, 11001, 12002, 13001, 12, 13)
  ("INV", 12, 12002, 14001, 0, 14, -1)
  ("AND", 13, 13001, 13002, 6001, -1, 6)
  ("INV", 14, 14001, 15001, 0, 15, -1)
  ("INV", 15, 15001, 13002, 0, 13, -1)
  ("NOR", 21, 21001, 21002, 21005, -1, 22)
  ("SPL", 22, 21005, 23002, 51001, 23, 51)
  ("NOR", 23, 23001, 23002, 23005, -1, 24)
  ("SPL", 24, 23005, 21002, 28001, 21, 28)
  ("AND", 25, 25001, 25002, 21001, -1, 21)
  ("SPL", 26, 26001, 25002, 27002, 25, 27)
  ("AND", 27, 27001, 27002, 23001, -1, 23)
  ("SPL", 28, 28001, 27001, 29001, 27, 29)
  ("INV", 29, 29001, 25001, 0, 25, -1)
  ("SPL", 31, 31001, 32002, 33001, 32, 33)
  ("INV", 32, 32002, 34001, 0, 34, -1)
  ("AND", 33, 33001, 33002, 26001, -1, 26)
  ("INV", 34, 34001, 35001, 0, 35, -1)
  ("INV", 35, 35001, 33002, 0, 33, -1)
  ("NOR", 41, 41001, 41002, 41005, -1, 42)
  ("SPL", 42, 41005, 43002, 71001, 43, 71)
  ("NOR", 43, 43001, 43002, 43005, -1, 44)
  ("SPL", 44, 43005, 41002, 48001, 41, 48)
  ("AND", 45, 45001, 45002, 41001, -1, 41)
  ("SPL", 46, 46001, 45002, 47002, 45, 47)
  ("AND", 47, 47001, 47002, 43001, -1, 43)
  ("SPL", 48, 48001, 47001, 49001, 47, 49)
  ("INV", 49, 49001, 45001, 0, 45, -1)
  ("SPL", 51, 51001, 52002, 53001, 52, 53)
  ("INV", 52, 52002, 54001, 0, 54, -1)
  ("AND", 53, 53001, 53002, 46001, -1, 46)
  ("INV", 54, 54001, 55001, 0, 55, -1)
  ("INV", 55, 55001, 53002, 0, 53, -1)
  ("NOR", 61, 61001, 61002, 61005, -1, 62)
  ("SPL", 62, 61005, 63002, 62005, 63, -1)
  ("NOR", 63, 63001, 63002, 63005, -1, 64)
  ("SPL", 64, 63005, 61002, 68001, 61, 68)
  ("AND", 65, 65001, 65002, 61001, -1, 61)
  ("SPL", 66, 66001, 65002, 67002, 65, 67)
  ("AND", 67, 67001, 67002, 63001, -1, 63)
  ("SPL", 68, 68001, 67001, 69001, 67, 69)
  ("INV", 69, 69001, 65001, 0, 65, -1)
  ("SPL", 71, 71001, 72002, 73001, 72, 73)
  ("INV", 72, 72002, 74001, 0, 74, -1)
  ("AND", 73, 73001, 73002, 66001, -1, 66)
  ("INV", 74, 74001, 75001, 0, 75, -1)
  ("INV", 75, 75001, 73002, 0, 73, -1)
Interacts:
  (1, "PLS", 11001, 11)
  (1, "LED", 8001, 8)
  (2, "LED", 28001, 28)
  (3, "LED", 48001, 48)
  (4, "LED", 68001, 68)
```

Output

As a reward for patience, the PSH button has been upgraded to species PLS, freeing the user's hand. Letting the circuit run, the output section cycles through the numbers 0 to 15 as follows:




Light:

Light:

Light:

Light:

Light:

Light:

Light:

Light:

Light:

Light:

Light:

Light: