

## Chapter 2: Digital Logic

### Contents

Chapter 2: Digital Logic.....	1
Chapter overview .....	3
Learning objectives.....	3
Combinational Logic .....	4
Concept: Logic Gates .....	4
Lab: Inverter .....	5
Demo: Inverter .....	6
TL-Verilog Syntax: Philosophy .....	6
TL-Verilog Syntax: Boolean Logic Expressions .....	6
TL-Verilog Syntax: Indentation and Signal Names .....	7
Indentation .....	7
Signal Names .....	7
Lab: Logic Gates .....	8
Demo: Logic Gates .....	9
Arithmetic Logic.....	10
Concept: Arithmetic Logic .....	10
TL-Verilog Syntax .....	11
Lab: Arithmetic Operators .....	11
Demo: Arithmetic Operators .....	12
Multiplexers.....	13
Concept: Multiplexers .....	13
TL-Verilog Syntax .....	13
Lab: Calculator .....	14
Demo: Calculator .....	15
Literals and Concatenation.....	16

Concept and Syntax .....	16
Literals .....	16
Concatenation .....	16
Lab: Calculator Stimulus .....	16
Demo: Calculator Stimulus .....	17
Visual Debug.....	18
Lab: Visual Debug .....	18
Demo: Visual Debug .....	18
File Structure and Tool Flow.....	19
Concepts .....	19
Video: File Structure and Tool Flow Concepts.....	20
Motivation .....	21
Sequential Logic.....	22
Concept: Clock and Flip-Flops.....	22
Sequential Logic Example and Reset .....	22
TL-Verilog Syntax .....	24
Concept: Generalization of Sequential Logic.....	25
Lab: Counter .....	25
Demo: Counter .....	26
Lab: Recirculating Calculator .....	26
Demo: Recirculating Calculator .....	27
List of sources .....	28

## **Chapter overview**

This chapter provides an opportunity to explore basic circuits within the Makerchip IDE. It establishes a baseline understanding of the fundamental principles of digital logic design, and how to design digital logic using TL-Verilog and the Makerchip IDE.

## **Learning objectives**

By the end of this chapter, you should:

- be familiar with basic logic gates;
- be comfortable composing logic gates into higher-order combinational logic functions, including multiplexers and arithmetic circuits;
- understand the TL-Verilog language syntax for expressing combinational and arithmetic logic functions;
- comprehend sequential logic and how to express sequential logic in TL-Verilog;
- have gained experience debugging combinational circuits in Makerchip, including the use of Visual Debug capabilities, unique to the Makerchip platform.

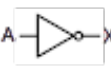
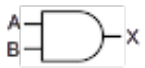





## Combinational Logic

### Concept: Logic Gates

If you are already familiar with logic gates, feel free to skip this concept.

In digital circuits, wires stabilize to one of two voltages: a high voltage (VDD) or a low voltage (VSS or ground). So, a wire carries a boolean value, where high and low voltages can be viewed as 1/0, true/false, asserted/deasserted, on/off, etc. This provides an important abstraction for composing higher-order logic functions with predictable behavior.

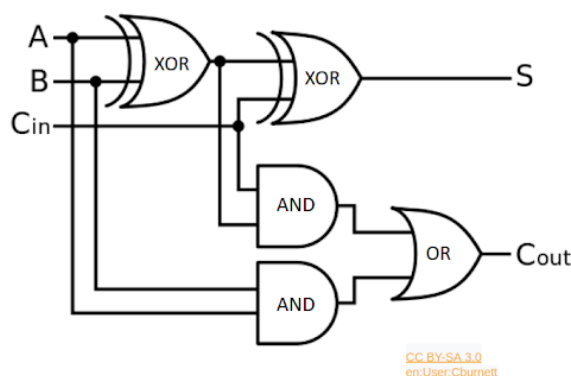
Logic gates are the basic building blocks for implementing logic functions. The table below shows basic logic gates. Their function is defined by the "truth tables", which show, for each combination of input values (A & B), what the output value (X) will be. Be sure to understand the behavior of each gate.

Name	NOT	AND	OR	XOR	NAND	NOR	XNOR																																																																																																
Symbol																																																																																																							
Truth Table	<table><tr><th>A</th><th>X</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	X	0	1	1	0	<table><tr><th>A</th><th>B</th><th>X</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	X	0	0	0	0	1	0	1	0	0	1	1	1	<table><tr><th>A</th><th>B</th><th>X</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	X	0	0	0	0	1	1	1	0	1	1	1	1	<table><tr><th>A</th><th>B</th><th>X</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	X	0	0	0	0	1	1	1	0	1	1	1	0	<table><tr><th>A</th><th>B</th><th>X</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	X	0	0	1	0	1	1	1	0	1	1	1	0	<table><tr><th>A</th><th>B</th><th>X</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	X	0	0	1	0	1	0	1	0	0	1	1	0	<table><tr><th>A</th><th>B</th><th>X</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	X	0	0	1	0	1	0	1	0	0	1	1	1
A	X																																																																																																						
0	1																																																																																																						
1	0																																																																																																						
A	B	X																																																																																																					
0	0	0																																																																																																					
0	1	0																																																																																																					
1	0	0																																																																																																					
1	1	1																																																																																																					
A	B	X																																																																																																					
0	0	0																																																																																																					
0	1	1																																																																																																					
1	0	1																																																																																																					
1	1	1																																																																																																					
A	B	X																																																																																																					
0	0	0																																																																																																					
0	1	1																																																																																																					
1	0	1																																																																																																					
1	1	0																																																																																																					
A	B	X																																																																																																					
0	0	1																																																																																																					
0	1	1																																																																																																					
1	0	1																																																																																																					
1	1	0																																																																																																					
A	B	X																																																																																																					
0	0	1																																																																																																					
0	1	0																																																																																																					
1	0	0																																																																																																					
1	1	0																																																																																																					
A	B	X																																																																																																					
0	0	1																																																																																																					
0	1	0																																																																																																					
1	0	0																																																																																																					
1	1	1																																																																																																					

Picture 1. Logic Gates.

Note:

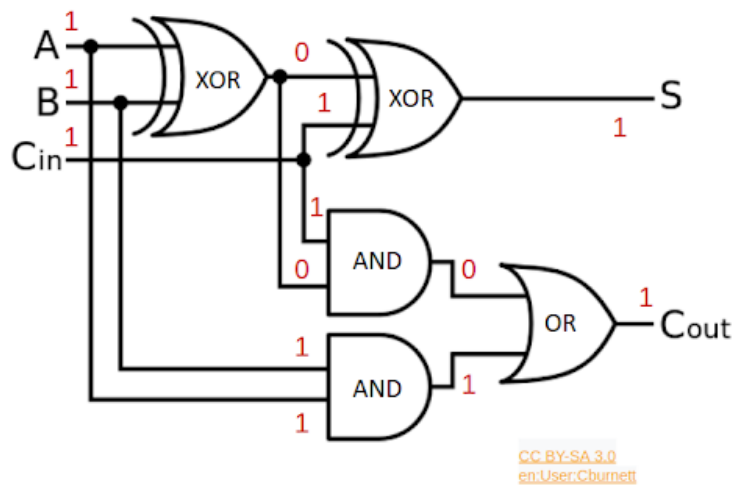
- AND and OR gates follow their English meanings;
- the small circle (or "bubble") on the output of some gates indicates an inverted output;
- XOR and XNOR are "exclusive" OR and NOR, where "exclusive" means "but not both";
- logic gates can be composed to generate higher-order logic functions, as in the circuit below.  
(This happens to be a circuit known as a "full adder".)



CC BY-SA 3.0  
en User:Cburnett

Picture 2. Full Adder Circuit.

For a given set of input values, such as the ones depicted below, we will get a given set of output values.



Picture 3. Example of Boolean Value Propagation through Gates.

### Lab: Inverter

Let's try coding an inverter (a **NOT** gate) in Makerchip. As you go through this lab exercise, check the box for each step when done, to ensure you perform all required steps.

- ◆ Reload the Makerchip IDE to begin with the default code template. (You could also use **Ctrl-Z** in the Editor to restore to the default template or load the default template from the Examples page.)
- ◆ The first line of the source file specifies the TL-Verilog language version. If it is other than "1d", it may be necessary to revert the language version to be consistent with this course. In this case, check the GitHub repository [1] for guidance.
- ◆ In place of "//...", type "`$out = ! $in1;`". Be sure to preserve the 3-spaces of indentation, similar to the surrounding expressions. This is an inverter.
- ◆ Compile and simulate (under the Editor's "E" menu, or **Ctrl-Enter**). If any red X's appear on the tabs (vs. green checkmarks), make sure you followed the instructions properly and try to resolve the issue. Use the LOG to debug if necessary, or use the video on the next page in times of desperation.

Exploring the results, we make a few key observations.

- Unlike Verilog, there was no need to declare your signals (wires) (`$out` and `$in1`). In TL-Verilog, your assignment statement acts as the declaration of its output signal(s).

- This circuit has a dangling input signal and a dangling output signal. It also probably has a dangling `$reset` signal that was provided in the template. These result in non-fatal warning/error conditions. They are reported, but they do not prevent simulation.
- ◆ Observe the non-fatal errors in the LOG and in the corresponding mouse-over pop-ups in NAV-TLV.
- There was no need to write a *test bench* to provide stimulus (input) to your inverter. Makerchip provides random stimulus for dangling inputs. As your designs mature, you will want to avoid dangling signals and provide more-targeted stimulus, but, while your code is under development, automatic stimulus can be a real convenience.
- ◆ See in the WAVEFORM that the inverter's output is indeed the inverse of its input.
- ◆ Find the logic expression for your inverter in the DIAGRAM in a mouse-over pop-up. Select the expression in the DIAGRAM and observe the highlighting in other panes.

### Demo: Inverter

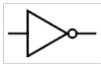



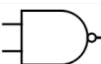


In case you had trouble with the Inverter lab, [here](#) is a screen capture of the steps you had to perform.

### TL-Verilog Syntax: Philosophy

TL-Verilog defines syntax very rigidly. This can be a source of annoyance for newcomers with their own coding style preferences. But this rigidity has an important benefit. In industry, code is touched by many hands from many teams, and this rigidity enforces consistency. The TL-Verilog Syntax Specification can be found from the Makerchip "Help" menu, but we will cover the necessary points as we go.

### TL-Verilog Syntax: Boolean Logic Expressions

Boolean logic has taken on various notations in different fields of study. The following chart shows some of these mathematical notations, as well as TL-Verilog operators (which are the same as Verilog) for basic logic gates.

Op	Bool Arith	Bool Calc	Verilog	Gate
NOT	$\overline{A}$	$\neg A$	$\sim A$ (or $!A$ )	
AND	$A \cdot B$	$A \wedge B$	$A \& B$ (or $\&\&$ )	
OR	$A + B$	$A \vee B$	$A   B$ (or $  $ )	
XOR	$A \oplus B$	$A \oplus B$	$A \wedge B$	
NAND	$\overline{A \cdot B}$	$\neg (A \wedge B)$	$!(A \& B)$	
NOR	$\overline{A + B}$	$\neg (A \vee B)$	$!(A   B)$	
XNOR	$\overline{A \oplus B}$	$\neg (A \oplus B)$	$!(A \wedge B)$	

Picture 4. Boolean Logic: Mathematical Notations and TL-Verilog Operators for Basic Logic Gates.

You can use parentheses to group expressions to form more complex logic functions. If a statement is extended to multiple lines, these lines must have greater indentation than the first line. Statements must always end with a semicolon. Always have a space before and after the "=". For example:

```
$foo = ( $val1 && $val2) ||
        (! $val1 && ! $val2);
```

## TL-Verilog Syntax: Indentation and Signal Names

### Indentation

In TL-Verilog (within \TLV code blocks), indentation and whitespace are meaningful. Tabs (which have no consistently-defined behavior) are not permitted. Each level of indentation is 3 spaces (and the Makerchip editor helps with this).

### Signal Names

As long as you stick with the suggested signal names throughout this course, you won't have any trouble, but, for those who might wish to veer off from the script a bit, TL-Verilog is picky about signal names too. While languages typically leave choices like camel-case vs. underscore delimitation up to coding conventions, TL-Verilog enforces these choices.

Naming restrictions serve several purposes:

- They enforce consistency.
- They distinguish types.

- As TL-Verilog is processed into Verilog, auto-generated logic can use Verilog signal names that cannot conflict with those named by the coder.

Specifically, TL-Verilog signal names:

- Are prefixed with "\$".
- Are composed of tokens delimited by underscores, where each token is a string of lower-case characters followed by zero or more digits .
- Begin with at least two alphabetic (lower-case) characters.

So, for example, these are legal signal names:

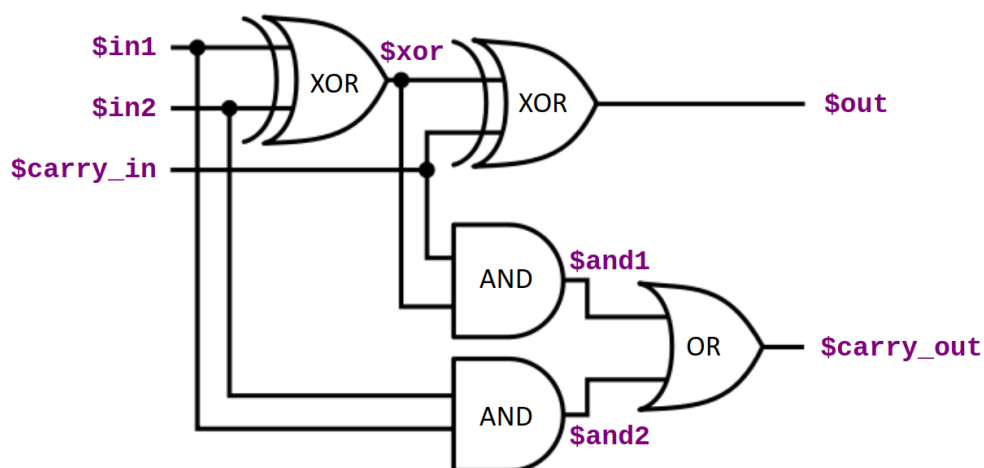
- \$my\_sig;
- \$val1.

And these are not:

- \$a;
- \$Sig (this is actually a "state signal", which we will not use in this course);
- \$val\_1;

You may elsewhere see the term "*pipesignal*" referring to these TL-Verilog signals. The distinction between Verilog signals and TL-Verilog pipesignals is not relevant in this course, and we will simply use the short-hand term "*signals*" throughout.

### Lab: Logic Gates



CC BY-SA 3.0  
en:User:Cburnett

Picture 5. Full Adder Circuit for Reference.

As you go through the labs of this course, you can use the checkboxes to keep track of your completion of each step.

- As you did with the inverter, try other single-gate logic expressions.



- If you are new to hardware description languages (HDLs), try coding the full adder circuit as depicted above. Try it first with each logic gate as a separate statement, then try combining the three gates producing `$carry_out` into a single statement with parentheses to group subexpressions.

### **Demo: Logic Gates**

In case you had trouble with the lab, [here](#) is a screen capture of the steps you had to perform.

## Arithmetic Logic

### Concept: Arithmetic Logic

If you have prior experience with hardware description languages (HDLs), and are comfortable with binary and hexadecimal, you can safely skip this concept.

While individual wires (or "bits") hold one of two values in a digital circuit, we can have a collection of N wires (called a "vector") that represent up to  $2^N$  possible values.

We are all used to representing numbers in base ten, or decimal. In decimal, we use ten digits, 0-9, and when we count past the last available digit, 9, we wrap back to 0 and increment the next place value, which is worth ten. Base ten, unfortunately, is very awkward for digital logic. Base two or any power of two (4, 8, 16) is much more natural. In base two, or binary, we have digits 0 and 1. Each digit can be represented by a bit. Base sixteen, or hexadecimal, is also very common. In hexadecimal, the digits are 0-9 and A-F (for ten through fifteen). A single hexadecimal digit can be represented by four bits. The table below shows values zero through twenty in decimal, binary, and hexadecimal.

Decimal (base 10)	Binary (base 2)	Hexadecimal (base 16)
00	00000	00
01	00001	01
02	00010	02
03	00011	03
04	00100	04
05	00101	05
06	00110	06
07	00111	07
08	01000	08
09	01001	09
10	01010	0A
11	01011	0B
12	01100	0C
13	01101	0D
14	01110	0E
15	01111	0F
16	10000	10
17	10001	11
18	10010	12
19	10011	13
20	10100	14

Picture 6. Decimal, binary, and hexadecimal number systems.

HDLs support "vector" signals that hold multiple bits. Vectors can be used to represent binary-encoded (signed or unsigned) integer values. For example, a 5-bit vector could hold the value 13 as bit values 01101. HDLs support arithmetic operations, such as addition, that operate on these bit vector values.

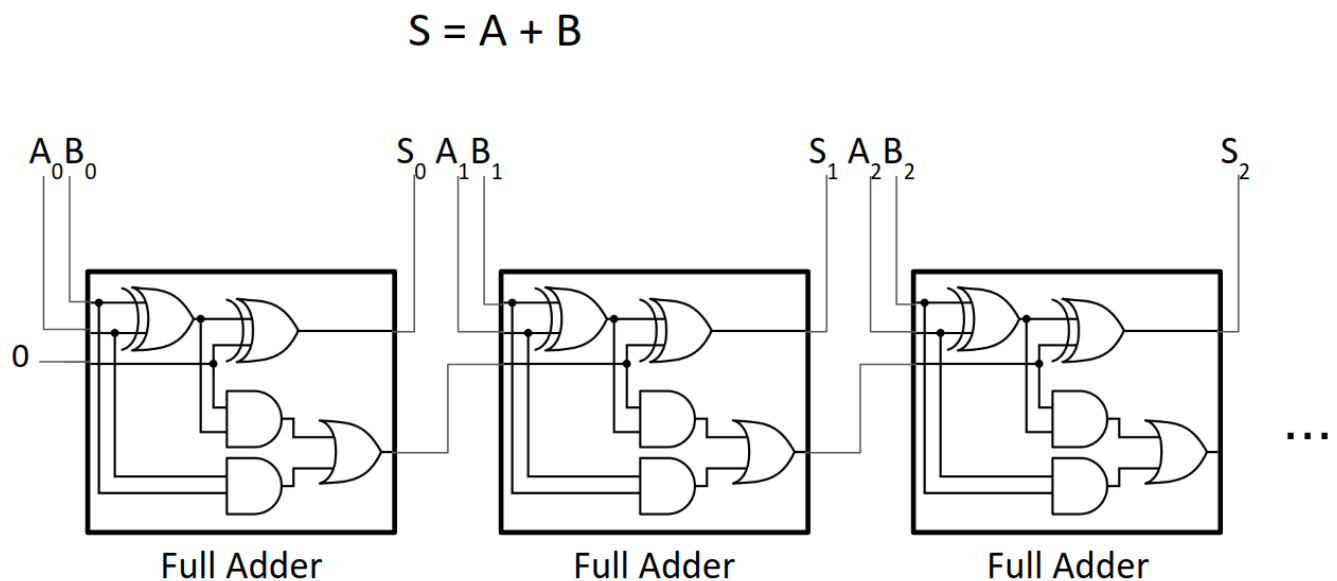
### TL-Verilog Syntax

In TL-Verilog, the most common data types are booleans (as you used in the previous lab) and bit vectors. A vector is declared by providing a bit range in its assignment as so:

```
$vect[7:0] = ....;
```

Bit ranges are generally not required on the right-hand side of an expression. When they are used, they extract a subrange of bits from a vector signal.

In Verilog and TL-Verilog, arithmetic operators, like +, -, \*, /, and % (modulo) can be used on vectors. Without these operators, an adder circuit would have to be constructed by replicating the full adder circuit we looked at earlier for each bit position in the adder to create the "ripple-carry adder" circuit depicted below.



Picture 7. Ripple-carry adder circuit composed of full adders.

Other vector operators are supported, including comparison operators like ==, !=, >, >=, <, <=. We will only cover the operators needed for this course.

### Lab: Arithmetic Operators

Starting again from the default template, or first deleting your previous logic, now try some arithmetic expressions.

Note:

As with our earlier examples, you will, as a convenience, not bother declare the input vectors. As such, there is no definition of their width. In this circumstance, you can define their widths by using bit ranges on the right-hand side as such:

```
$out[7:0] = $in1[6:0] + $in2[6:0];
```

Values of vector signals are represented in the waveform viewer in hexadecimal. There are many online conversion tools, such as RapidTables if you need help finding a decimal or binary equivalent.

This lab is a simple one; don't forget to check the box when you completed the step.

- Code various arithmetic expressions and comparisons to become comfortable with their use.

### **Demo: Arithmetic Operators**

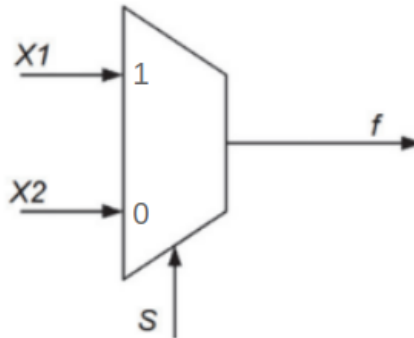
In case you had trouble with the Arithmetic Operators lab, [here](#) is a screen capture of the steps you had to perform.

## Multiplexers

### Concept: Multiplexers

If you are familiar with multiplexers, you may safely skip this concept.

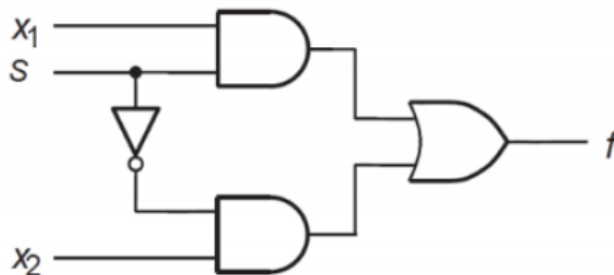
One of the most important logic functions is a multiplexer (or MUX), depicted below.



Picture 8. A two-way single-bit multiplexer with one-bit (encoded) select.

A multiplexer selects between two or more inputs (which can be binary values, vectors, or any other data type). The select line(s) identify the input to drive to the output. Most often, the select will be either a binary-encoded input index or a "one-hot" vector in which each bit of the vector corresponds to an input. One and only one of the bits will be asserted to select the corresponding input value.

The MUX depicted in the "Two-way single-bit multiplexer" graphic above can be constructed from basic logic gates, as seen below. We might read this implementation as "assert the output if  $x_1$  is asserted and selected (by  $S == 1$ ) OR  $x_2$  is asserted and selected (by  $S == 0$ )".



Picture 9. Gate-level implementation of a multiplexer.

### TL-Verilog Syntax

Verilog provides no less than six reasonable syntaxes for coding a multiplexer, each with pros and cons. TL-Verilog favors the use of the ternary ( $? :$ ) operator, and we will stick with this throughout the course. In its simplest form, the ternary operator is:

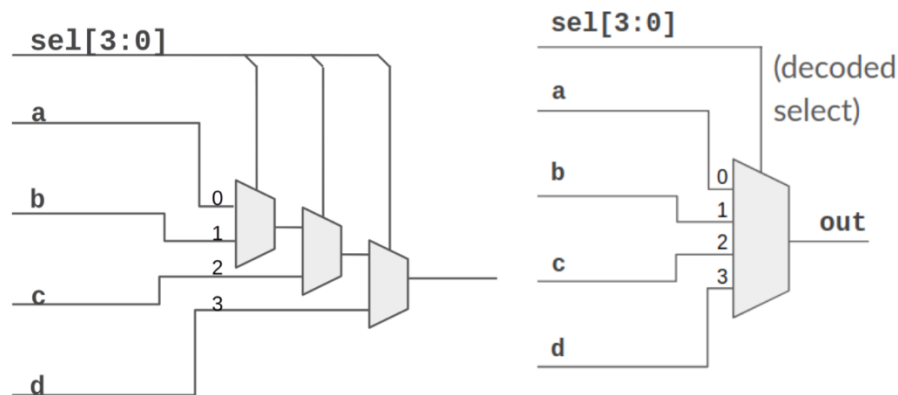
```
$out = $sel ? $in1 : $in0;
```

This can be read, "\$out is: if \$sel then \$in1 otherwise \$in0."

The ternary operator can be chained to implement multiplexers with more than two input values from which to select. And these inputs can be vectors. We will use very specific code formatting for consistency, illustrated below for a four-way, 8-bit wide multiplexer with a one-hot select. (Here, \$in0-3 must be 8-bit vectors.)

```
$out[7:0] =  
    $sel[3]  
        ? $in3 :  
    $sel[2]  
        ? $in2 :  
    $sel[1]  
        ? $in1 :  
    //default  
    $in0;
```

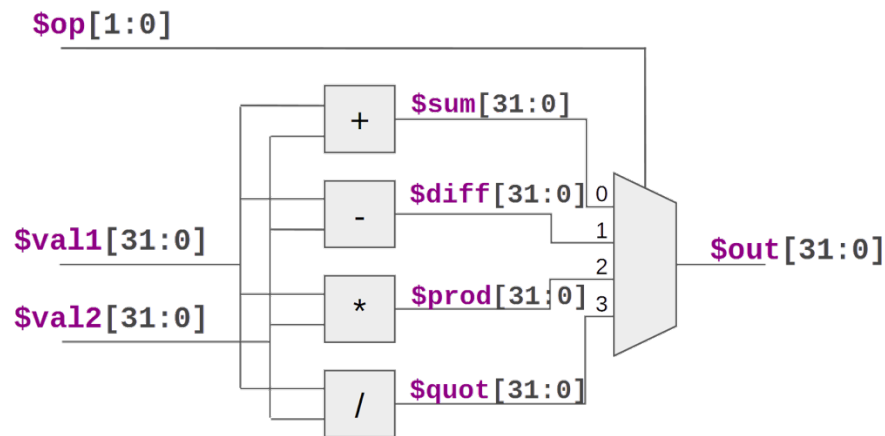
This expression prioritizes top-to-bottom. So, if \$sel[3] is asserted, \$in3 will be driven on the output regardless of the other \$sel bits. Its literal interpretation is depicted below, along with its single-gate representation (which is ambiguous about the priority).



Picture 10. A four-input multiplexer, represented as a chain of two-input multiplexers and as a single gate.

### Lab: Calculator

Next, you will try coding the circuit below. This circuit implements a calculator that can perform +, -, \*, or / on two input values. Provide an expression for each of the signals named below, and be sure to use the exact names shown and the select encodings shown on the MUX inputs. This will be important later.



Picture 11. Combinational calculator circuit.

Note that this circuit uses an encoded (aka binary) select, where two  $\$op[1:0]$  bits select from four possible MUX inputs.  $\$op$  can be decoded by using expressions like  $\$op[1:0] == 2$  to select  $\$prod$ , for example. As you go through this lab exercise, check the box for each step when done, to ensure you perform all required steps.

- Since we wouldn't want you to lose your work, click "Save as New Project" in the upper right. This will create a project that should auto-save and will update the URL to reflect this new project. Bookmark this new URL and name your bookmark so you can return to your work later (or perhaps just rely on your browser to remember the URL in its history). The status text in the upper right will confirm that your file is auto-saving.
- Code the circuit.
- Be sure your LOG contains only expected errors/warnings about dangling signals.
- Inspect the circuit diagram to make sure it looks right. (Don't worry about the waveforms just yet, we will debug this circuit in the next two labs.)

### Demo: Calculator

In case you had trouble with the Calculator lab, [here](#) is a screen capture of the steps you had to perform.

## Literals and Concatenation

### Concept and Syntax

#### Literals

If you are familiar with Verilog expression syntax, you may safely skip this "Concept and Syntax" section.

This expression:

```
$foo[7:0] = 6;
```

defines `$foo` to hold a constant value of 6. In this case, the 6 is coerced to eight bits by the assignment.

Often, it is necessary to be explicit about the width of a literal:

```
$foo[7:0] = 8'd6;
```

explicitly assigns `$foo` to an 8-bit decimal ("d") value of 6. (To be clear, the `'` is the single-quote character.) Equivalently, we could have written:

```
$foo[7:0] = 8'b110;    // 8-bit binary six
```

or

```
$foo[7:0] = 8'h6;      // 8-bit hexadecimal
```

#### Concatenation

Concatenation of bit vectors is simply the combining of two bit vectors one after the other to form a wider bit vector. The syntax is clear from this example:

```
$word[15:0] = {$upper_byte, $lower_byte};
```

### Lab: Calculator Stimulus

Your calculator circuit is driven with random 32-bit inputs. You may have observed that the computation often underflows or overflows, meaning the correct result value would be too large ( $> 2^{32}$ ) or too small ( $< 0$ ) to express in the 32-bit `$out` signal.

It will be easier to visually verify the simulation behavior if we use smaller input values. So, let's randomize only the lower bits of `$val1` and `$val2`. To reduce the chance of underflow, let's use a smaller value for `$val2` than `$val1`. As you go through this lab exercise, check the box for each step when done, to ensure you perform all required steps.

- Return to your calculator project.
- Assign `$val1` such that its upper 26 bits are zero and its remaining 6 bits are random. To do so, you'll use a literal and a concatenation. For the random bits use a new unassigned signal, `$val1_rand[5:0]`.



- Assign `$val2` similarly, but randomizing only the lower 4 bits.

The default code template you started with in Makerchip compiles without strict bit-width checking. Let's enable strict checking to make sure you got this right.

- Below `"m4_makerchip_module"`, add this on its own line: `/* verilator lint_on WIDTH */`, and press **Ctrl-Enter** to compile and simulate. Debug LOG messages as necessary.
- Now, it should be easier to understand the waveforms, so visually verify the operation of your calculator. Verify that it saved (message in upper right).

### Demo: Calculator Stimulus

In case you had trouble with the Calculator Stimulus lab, [here](#) is a screen capture of the steps you had to perform.

## Visual Debug

### Lab: Visual Debug

Waveform viewers have been the standard debug tool for circuit design since dinosaurs roamed the earth. But Makerchip supports a better debug methodology as well. We have prepared some custom visualization to help with the debug of your calculator. As always, check the box for each step when done, to ensure you perform all required steps.

To include this visualization:

- Paste this single line below the "**m4\_makerchip\_module**" line to include the visualization library:  

```
m4_include_lib(['https://raw.githubusercontent.com/stevehoover/LF-Building-a-RISC-V-CPU-Core/main/lib/calc_viz.tlv']).
```

It may be necessary to correct the single-quote characters by retyping them after cut-and-pasting.
- Add this line as the last line in the **\TLV** region: "**m4+calc\_viz()**" to instantiate the visualization. Press **<Ctrl>-Enter**.
- You should now see a calculator in the VIZ pane. If necessary, debug the LOG. In NAV-TLV, the **m4\_include\_lib** line should have turned into a comment, and the **m4+calc\_viz()** macro instantiation should have expanded to a block of "**\viz**" code.
- Lay out your IDE so you can see both VIZ and WAVEFORM. Step through VIZ to see the operations performed by your calculator. Note that your calculator, like the waveform, is showing values in hexadecimal. Relate what you see in VIZ to what you see in the waveform. If you notice incorrect behavior, debug it by isolating the faulty logic and fixing it.
- Be sure your work is saved.

Typically, you would create your own custom visualizations as you develop your circuit, so you can see the big picture simulation behavior more easily. In this course, you will focus on the hardware logic, and we provide visualization for you. If you are curious, though, you can see the visualization code in the NAV-TLV pane, where macros are expanded.

### Demo: Visual Debug

In case you had trouble with the Visual Debug lab, [here](#) is a screen capture of the steps you had to perform.

## File Structure and Tool Flow

### Concepts

This *File Structure and Tool Flow* topic is entirely optional. It describes the TL-Verilog file structure used within Makerchip and the motivations for it. It describes the use of the M4 macro preprocessor and other aspects of the tool flow used behind the scenes when you compile and simulate. Feel free to skip over it, but we know some of you will be curious, especially those who have experience with Verilog.

**Note:** to get more insight on this topic, you can EITHER follow the self-guided tour on this page, OR you can watch the video exploring this topic in a bit more detail on the next page.

First, in the IDE's "Help" menu, select "Help", and read through this help page.

Your TL-Verilog source file is first processed by the M4 macro preprocessor. This is how we imported and used the calculator visualization. The resulting TL-Verilog file is processed into SystemVerilog or Verilog by Redwood EDA's SandPiper™ tool. You can see how your code is processed into Verilog by SandPiper by opening "Show Verilog" under the Editor's "E" menu.

Verilator is an open-source tool used to compile your Verilog code into a C++ simulator. This simulator is run to produce the trace data that you can view in the waveform viewer.

The LOG tab shows output from all these tools. Output from M4 and SandPiper is in blue, and output from Verilator and its resulting simulation are in black.

TL-Verilog features are used to define the logic within a (System)Verilog module. Code comments below explain the parts of a TL-Verilog source file structured for use within Makerchip.

```
\m4_TLV_version 1d --bestsv: tl-x.org
// This version line specifies:
//   o that macro preprocessing using M4 is enabled
//   o the language version in use (1d)
//   o optionally command-line arguments
//   o a link to docs.
\SV
// This region contains SystemVerilog (or Verilog) code.
// SandPiper passes this code through to the Verilog file without
// any processing.
m4_makerchip_module
// This M4 macro expands to a Verilog module definition with
// an interface that is required by the Makerchip platform.
// This module interface provides the communication between
```

```

// Makerchip and your design.
//
// It includes global clock and reset input signals
// via this interface and its output signals "passed"
// and "failed" can be driven to end the simulation with a
// "Simulation Passed" or "Simulation FAILED" message in the
// LOG.
//
// To see the expansion of this macro, look in the NAV-TLV
// pane. This macro also provides a random vector that can
// be used for stimulus and it provides some Verilator
// configuration.
\TLV
// TL-Verilog syntax is enabled in this region to express your
// logic. In this course, we'll always declare our logic
// here within the m4_makerchip_module, but you could also
// put your TLV logic in a separate module with an interface
// that is yours to define.
$reset = *reset;
    // In \TLV context, *reset references the (System)Verilog
    // reset signal. Here we simply connect it to a TL-Verilog
    // $reset pipesignal.
// YOUR LOGIC HERE
*passed = ...;
*failed = ...;
    // Assert either of these to end simulation (before Makerchip
    // cycle limit).
\SV
// Back to SystemVerilog context to end the module.
endmodule

```

### Video: File Structure and Tool Flow Concepts

Sit back and watch [this](#) nine-minute video for full details.

## Motivation

For those curious about the motivation for this file structure, it is necessary to understand the strategy for evolving TL-Verilog from Verilog. The ultimate goal is to eventually introduce a new modeling language philosophically different from Verilog in all respects. This will play out over the next decade or decades. In the meantime, we work toward this incrementally, layering on Verilog as a working starting point, with TL-Verilog as a language extension to Verilog. This layering also provides an essential and incremental migration path. And, as tools mature, it is always possible to fall back on Verilog.

Also noteworthy is the fact that TL-Verilog is really a Verilog implementation of TL-X, a language extension defined to layer atop any HDL to extend it with transaction-level features. So there is a migration path from any supported HDL (and, as of this writing, Verilog is the only one).

By using TL-Verilog syntax only within module definitions, Verilog-based tools that are used to stitch the interconnections between modules can remain blissfully unaware of TL-Verilog. Within the Verilog module, other forms of modularity and hierarchy, particular to TL-Verilog can be employed.

Everything we will do in this course will be inside the `\TLV` region, but now you understand how this connects with Verilog-based tools.

**Note:** everything we do in this course could be done just as well in Verilog, but, by using TL-Verilog, you will be poised for further learning of the true power of transaction-level design. You will be able to follow this course up with others where you will pipeline your design with ease. (Also, the Makerchip DIAGRAM tab only works with TL-Verilog.)

## Sequential Logic

### Concept: Clock and Flip-Flops

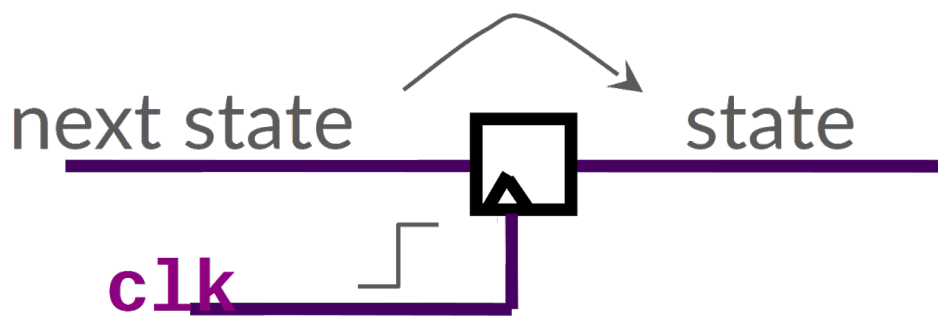
If you are familiar with sequential logic and flip-flops, you can safely skip this concept.

Sequential logic introduces a clock signal.



Picture 12. Clock waveform.

The clock is driven throughout the circuit to "flip-flops" which sequence the logic. Flip-flops come in various flavors, but the simplest and most common type of flip-flop, and the only one we will concern ourselves with, is called a "positive-edge-triggered D-type flip-flop". These drive the value at their input to their output, but only when the clock rises. They hold their output value until the next rising edge of their clock input.

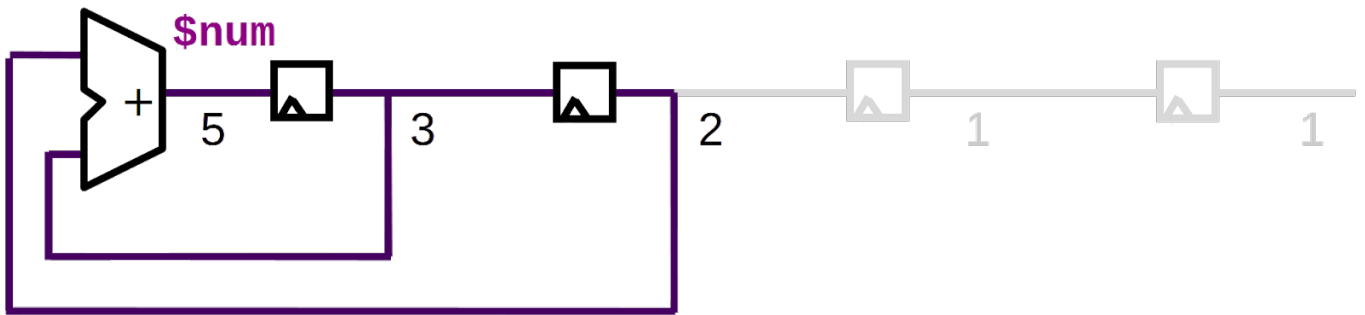


Picture 13. A flip-flop.

Although there are also flip-flops that act on the falling edge of the clock, our circuits will operate only on the rising edge. Additionally, there are flip-flops that incorporate logic functions. Tools can choose to implement our designs using these flip-flops even though we will not be explicit about doing so in our source code. Since we will use only D flip-flops, we will henceforth refer to them simply as flip-flops, or even just "flops".

### Sequential Logic Example and Reset

Before getting too theoretical about sequential logic, let's look at an example. Let's look at a circuit that computes the Fibonacci sequence. Each number in the Fibonacci sequence is the sum of the previous two numbers: 1, 1, 2, 3, 5, 8, 13, ... This circuit will perpetually compute the next number in the sequence:



Picture 14. Fibonacci circuit (incomplete).

With each rising clock edge, the values will propagate through the flops, shifting one flop to the right, producing this waveform.

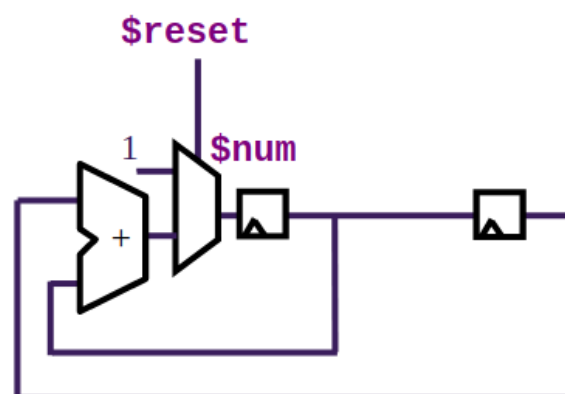


This circuit however is incomplete. What is missing? Reset. We need a way to initialize our circuit with two 1s to begin the sequence.

Unlike a combinational circuit, where output values are purely a function of the input values, sequential circuits have internal state. Every sequential circuit needs the ability to get to a known "reset" state. And therefore, every sequential circuit will have a "reset" signal responsible for accomplishing this. The circuit must be designed such that, if reset is asserted for many cycles, the logic will stabilize to a known reset state.

The easiest approach to providing a reset capability is to force every flip-flop to a reset value when reset is asserted. This methodology is used by some design teams, but if area and power are a concern we can do better.

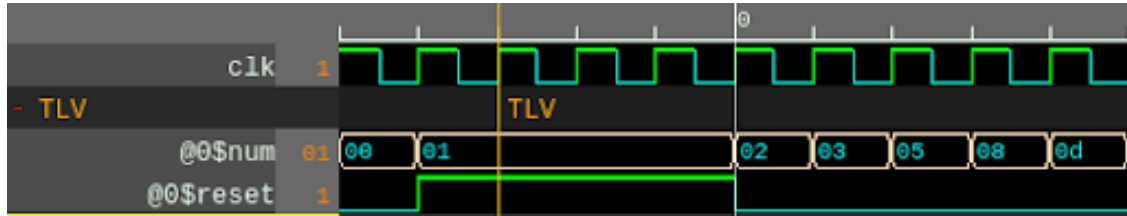
In our Fibonacci circuit, we only need to reset the first flip-flop.



Picture 15. Fibonacci series circuit.

While `$reset` is asserted, a 1 value is driven into `$num`. The clock continues to toggle during reset, and the 1 value propagates through both flops, resetting them both to a 1 value.

This provides the initial two 1s in the sequence. When `$reset` is deasserted, `$num` takes on a value of 2, and the circuit continues to produce a new value in the sequence in each subsequent clock cycle.



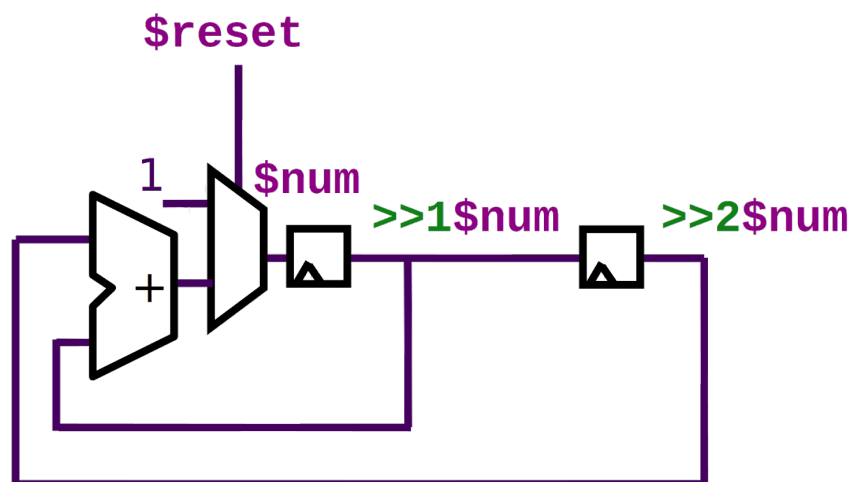
Picture 16. Fibonacci sequence circuit waveforms.

**Note:** it is common for `$reset` to be provided as a negatively asserted signal, perhaps named `rstn`, meaning reset occurs when `rstn` is 0, and `rstn` is 1 during normal operation. Though the motivation for this is rarely relevant with modern logic synthesis tools, you will still commonly see this in practice. We will stick with a positively asserted `$reset`.

**Note:** often, the reset capability is physically incorporated into the flip-flop itself. We will provide our reset logic explicitly. Logic synthesis tools can choose to implement the behavior using a "reset flip-flop".

### TL-Verilog Syntax

In TL-Verilog, we can reference the previous and previous-previous versions of `$num` as `>>1$num` and `>>2$num`. Unlike RTL, in TL design we need not assign these explicitly. They are implicitly available for use, and the need for flip-flops is implied.



Picture 17. Fibonacci circuit labeled with staged signal references.

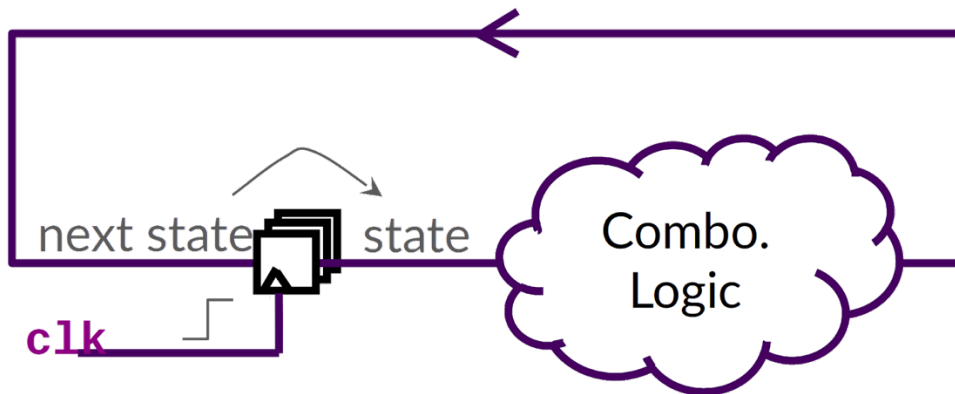
So this example can be coded as in the following Fibonacci sequence circuit expression:

```
$num[31:0] = $reset ? 1 : (>>1$num + >>2$num);
```



### Concept: Generalization of Sequential Logic

A sequential circuit, containing flops and combinational logic can be viewed as follows:

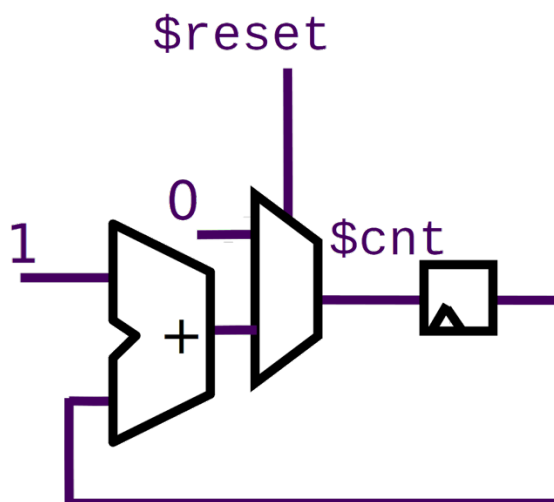


Picture 18. Generalized view of a sequential circuit.

During each cycle of the clock, the combinational logic evaluates, then the clock rises and next state becomes state, and the process continues.

### Lab: Counter

Now, it's time for you to try it. Similar to the Fibonacci sequence circuit, you'll create a 16-bit free running counter, depicted below.



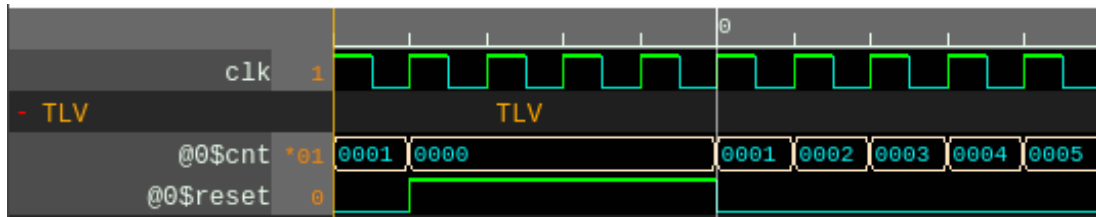
Picture 19. A counter circuit.

`$cnt` resets to zero (`16'b0`) and, after reset, begins incrementing by one (`16'b1`) each cycle.

As you go through this lab exercise, check the box when you completed the step.

- Reload or reset Makerchip to begin again from the default template.
- Confirm that the default template is providing you with a `$reset` signal as:  
`$reset = *reset;`
- Create the single-statement expression for this circuit. When you get the waveform below, you've got it right. For reference, this is the expression for the Fibonacci circuit:

```
$num[31:0] = $reset ? 1 : (>>1$num + >>2$num);
```



Picture 20. Correct waveform for counter circuit lab.

### Demo: Counter

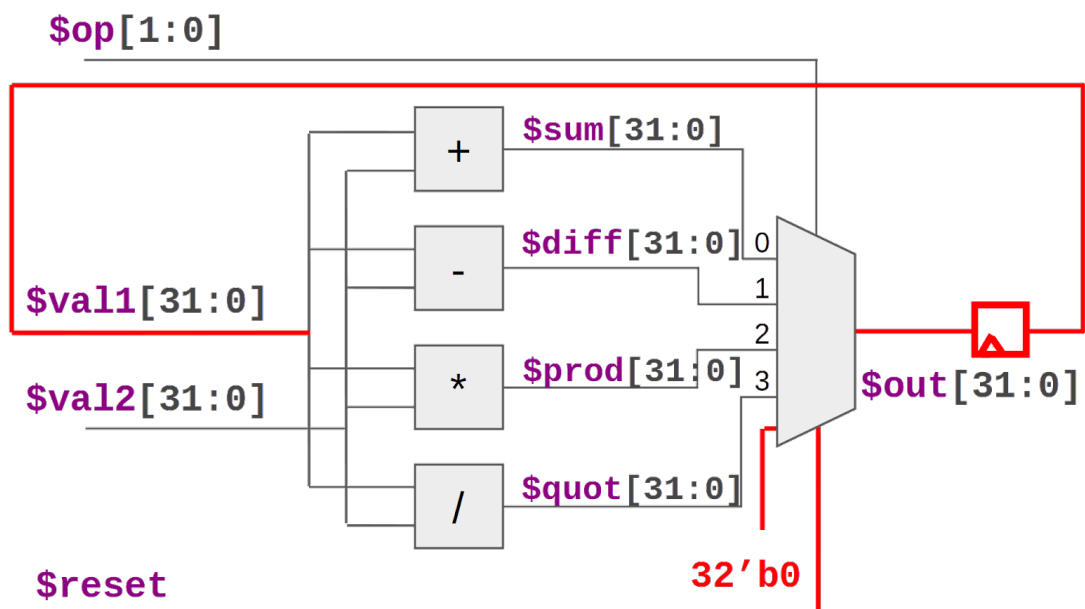
In case you had trouble with the Counter lab, [here](#) is a screen capture of the steps you had to perform.

### Lab: Recirculating Calculator

A real (old-school) calculator displays the result of each calculation. It holds onto this result value and uses it as the first operand in the next computation. If you enter "+ 3" in the calculator, it adds three to the previous result. Let's update our calculator to act like this. Each cycle, we'll perform a new calculation, based on the previous result.

This previous result is state. And wherever we have state, we must have a `$reset` that will set that state to a known value. As in a real calculator, we will reset the value to zero.

To recirculate the result (`$out`), and reset it to zero, we would have:



Picture 21. Logic modifications for this lab to sequentialize the calculator circuit.

As you go through this lab exercise, check the box when you completed the step.

- Return to your combinational calculator project.
- Assign `$val1[31:0]` to the previous value of `$out` (replacing its old assignment).

- Add a `$reset` signal and a new (highest priority) MUX input to reset `$out` to zero.
- Visually confirm proper operation in VIZ and WAVEFORM. Note that negative values will be represented with upper bits equal to 1 (so "fff..." in hexadecimal. (You could disable subtraction to prevent negative values.)
- You may want to save your work outside of Makerchip.

### **Demo: Recirculating Calculator**

In case you had trouble with the Recirculating Calculator lab, [here](#) is a screen capture of the steps you had to perform.

## List of sources

1. Building a RISC-V CPU Core // GitHub. URL: <https://github.com/stevehoover/LF-Building-a-RISC-V-CPU-Core>.