# Scratch VHDL User Guide

## Introduction

Scratch VHDL has been written as an education tool. It provides people with experience of the full design process for FPGAs without them having to learn VHDL. It uses freely available tools, ModelSim and Vivado, to avoid commercial barriers to getting started, and works within the limits of those free tools (e.g. number of VHDL statements, variety of devices).

The suggested examples are deliberately simple, in order to introduce basic concepts. You will light up only 4 LEDs in different ways using only 4 buttons. This still provides the opportunity to explore basic logic gates, finite state machines for different tasks, and finally a 4-bit RISC CPU! The original reason though is to demonstrate the synthesis of logic from code and provide the satisfaction of programming a real FPGA device. Hence the whole process becomes tangible through some hands-on familiarity.

## Basic Scratch

This section covers building designs using VHDL to infer logic. The next section will cover doing the equivalent designs using a custom RISC processor.

### Design Entry

1. Open VSCode in the Scratch VHDL root directory where the open source project was cloned, this will be referred to ${scratch\_vhdl} from now on.
2. Graphical user interface, application

   Description automatically generatedOpen “design\_entry.vhdl” to edit your design.
3. Drag and drop the library blocks to form your code. For this you will need to refer to the online “Demonstration Designs” list at <https://house-of-abbey.github.io/scratch_vhdl/#demonstration-designs>. You will also have tutors on hand to explain and guide you here.

**Increment:**

There is the internal concept of an “increment”. The external clock has and 8 ns period, or frequency of 125 MHz. Neither the LEDs nor our eyes can keep pace with logic changes that fast, so a signal is derived such that is pulses for one clock cycle every 0.5 s, synchronous to that clock, and hence using the condition that incr = ‘1’ will slow the changes down to be visible to the human eye. In simulation this is sped up to every 10 clock cycles as simulation is slower than real life.

1. To generate, right click to see the following menu:

Graphical user interface

Description automatically generated with medium confidenceFrom here you can “View Code” to see the VHDL generated by the Scratch editor, and “Build” to compile it ready for simulation in ModelSim.

1. Check the compile transcript for errors to correct.

**Advanced Tip:**

During simulation you will discover that the toolbar used to generate stimulus has multiple control tabs. To select the default tab you want to use can set the constant button\_tab\_c to a positive integer referring to the *n*th tab. To do this choose “Edit constants”.

### Simulation

1. To start the simulation tool for the first time run the batch file ${scratch\_vhdl}\design\run\_sim.cmd. You do not need to do this if ModelSim is already open. A command shell should appear to confirm the application is being opened.
2. Initially click the “Autostep” checkbox in order to emulate your design interactively whilst you press the buttons on the controls and watch the LEDs change.
3. The take a look at the simulation window. Useful controls in here are
   1. ‘f’/’F’ - Display the whole trace (full)
   2. ‘I’/’L’ - Zoom in
   3. ‘o’/’O’ - Zoom out
4. Verify the operation of your design and use the wave window to debug unexpected interactions. Once you have changed your code, remember to:
   1. “Right click “Build”
   2. Click “Resim” on the Modelsim custom toolbar to reload and restart the re-compiled design.

When you are satisfied, move on to the Synthesis step.

**Custom Simulation Controls:**

Graphical user interface, application

Description automatically generatedThree tabs provide the following interfaces:

| Tab | Purpose | button\_tab\_c |
| --- | --- | --- |
| Push Switch | LEDs 3:0 with press for on, release for off buttons 3:0 | 1 |
| Toggle Switch | LEDs 3:0 with press for on, press for off buttons 3:0 | 2 |
| Traffic | LEDs laid out for traffic lights with push buttons | 3 |

The “Sim Controls” are as follows:

| Sim Controls | Purpose |
| --- | --- |
| Reload | Open the Control panel (if it has been closed). |
| Change ASM | Restart the simulation from 0 ns. |
| Step | Scroll the displayed simulation to show the currently selected cursor. |
| Autostep | Setup the triggers to update the displayed controls. |
| At Cursor | Disconnect the triggers from the displayed controls. |

Text

Description automatically generated**Custom ModelSim Toolbar:**

| Button | Purpose |
| --- | --- |
| Controls | Open the Control panel (if it has been closed). |
| Resim | Restart the simulation from 0 ns. |
| Goto Cursor | Scroll the displayed simulation to show the currently selected cursor. |
| Start Monitor | Setup the triggers to update the displayed controls. |
| Stop Monitor | Disconnect the triggers from the displayed controls. |

### Synthesis

1. To start the synthesis tool for the first time run the batch file ${scratch\_vhdl}\design\ run\_vivado.cmd. You do not need to do this if Vivado is already open. Vivado may take a while to open, so check the Task Manager for no Vivado process before running the batch file again. Opening multiple instances slows the opening process down even more! A command shell should appear to confirm the application is being opened.
2. Make sure that “zybo\_scratch” is the default top level under “Design Sources”, if not right click it and select “Set as Top”.[[1]](#footnote-1)
   1. The alternative choice is “zybo\_risc\_cpu” which is covered later and this should not yet be chosen.
   2. Ignore all other options, they are possible lower levels of hierarchy that should not be selected.
   3. These two designs will include more code to wrap the led4\_button4 design you have created, to manage design aspects that you should not need to care about for the purposes of this exercise.
3. Click on the “E” button on the custom toolbar to elaborate the design. Give the tool time to throw algorithms about. When it is finished, the *generic* gates used to realise your design from VHDL will be shown for inspection.
   1. Click on a gate and press F7 to display the VHDL code used to infer the gate.
4. Diagram

   Description automatically generatedClick the “S” button on the custom toolbar to synthesise the design to library of gates available on the required device. When it is finished, the *mapped* gates used to realise your design from VHDL will be shown for inspection.
   1. All combinatorial logic is packed into 6-input LUTs. You can verify the truth table used in any LUT by checking the “Cell Properties” and then the “Truth Table”.
   2. Click on a gate and press F7 to display the VHDL code used to infer the gate.
5. Click the “i” button on the custom toolbar to implement or “place and route” the design on the required device. When it is finishedA picture containing diagram

   Description automatically generated a diagram similar to the synthesis results will be displayed, only now each gate has a location on the FPGA “sea of gates”.
   1. You can verify this by checking the “Cell Properties” (right click menu) then selecting the “General” tab and viewing the “Site” and “Tile” properties displayed there.
   2. Click on a gate and press F7 to display the VHDL code used to infer the gate.
   3. Check if you met timing in the “Timing” tab below the schematic viewer. “Worst Negative Slack (WNS)” should be positive.
   4. Graphical user interface, application

      Description automatically generatedClick the number to the right of WNS to see the list of 10 worth paths.
      1. Right click the path and select “View Path Report” for the timing details required for an understanding of which propagation delays are involved. This is used when failing to meet timing in order to decide how logic should be changed.
      2. Select a path and press “F4” to show the path in the schematic viewer to see which gates are involved in the report.
6. Click the “P” button on the custom toolbar to programme the FPGA board connected via the USB cable. Make sure the board has been powered on by the switch above the micro-USB connector.
   1. Graphical user interface, application, table

      Description automatically generatedThis display appears showing the board and the separate devices that can be programmed on it. The Zynq FPGA contains an ARM processor within it, but we ignore its presence completely. The FPGA logic is show here as “Programmed”.
   2. Now use the buttons on the development board to operate your design as you did in the simulation step.
7. You have completed putting your FPGA design through the whole design process, and hopefully avoided needing to learn VHDL.

**Custom Vivado Toolbar:**

| Button | Purpose |
| --- | --- |
| E | Elaborate, generic gates from VHDL code. |
| S | Synthesis, mapping generic gates to an actual device library to produce a netlist. |
| i | Implementation, place and route the netlist across the FPGA fabric. |
| P | Programme, download the resulting design to the FPGA development board. |
| A | Specify the assembled instruction file to be used with the RISC CPU. |

## RISC 4-Bit CPU

### Design Entry

You can design your own RISC CPU and instruction set, but for now use the one provided. The provided design has 8 internal registers as shown in the table below. Remember there is no RAM made available to the CPU, only ROM for the code that is fixed at compile time, hence a few more registers are provided than you might think necessary for a 4-bit CPU!

| Register | Purpose |
| --- | --- |
| r0..r5 | 6 general purpose registers. |
| btns or r6 | Read the button state |
| leds or r7 | Write the LEDs state |

### Simulation

1. Load the example “risc\_cpu.vhdl” design into VSCode.
2. Right click “Build” to compile it for simulation.
3. In ModelSim select “Change ASM” from the control display and select an assembled instruction file.
   1. The files should have been assembled as part of the project installation and can be found in %USERPROFILE%\ModelSim\projects\button\_leds\instr\_files\\*.o. The original source files are located in ${scratch\_vhdl}\design\demos\asm\\*.asm for inspection.
   2. Now use the controls to stimulate the design as before by pressing the “Autostep” button and then the 4 buttons provided and verify execution of the code via the LEDs and waveform window.

### Synthesis

1. To start the synthesis tool for the first time run the batch file ${scratch\_vhdl}\design\ run\_vivado.cmd. You do not need to do this if Vivado is already open. Vivado may take a while to open, so check the Task Manager for no Vivado process before running the batch file again. Opening multiple instances slows the opening process down even more! A command shell should appear to confirm the application is being opened.
2. Make sure that “zybo\_risc\_cpu” is the default top level under “Design Sources”, if not right click it and select “Set as Top”.[[2]](#footnote-2)
3. We are going to compile the design with the assembled instruction code in a read-only memory. The assembled code is provided to the design through a VHDL generic value which we need to specify.
   1. Graphical user interface, application

      Description automatically generatedClick the “A” button on the custom toolbar and specify the file to use with no path nor extension, e.g. “traffic\_lights”.[[3]](#footnote-3)
      1. NB. This dialogue box may appear too small to show the available field and value boxes, so the dialogue box may need to be resized.
      2. The files should have been assembled as part of the project installation and can be found in %USERPROFILE%\ModelSim\projects\button\_leds\instr\_files\\*.o. The original source files are located in ${scratch\_vhdl}\design\demos\asm\\*.asm for inspection.
   2. Click the “i” button on the custom toolbar, followed by the “P” button.

Repeat the process with your own instruction file.

1. Open instructions.asm in VSCode and write your instructions.
2. Assemble by running asm\_compile.cmd.
3. Select the assembled code in ModelSim and Vivado to simulate and synthesis the design.

**Instruction Set:**

The assembly mnemonics used here are based on a custom assembler created by [hlorenzi](https://hlorenzi.com/), and the documentation can be found here <https://github.com/hlorenzi/customasm/wiki/Getting-started>. Using this we have created an assembler for the 4-bit RISC CPU as defined in the following table, provided as a reference. The easiest way to learn the instruction set is to review the examples provided in ${scratch\_vhdl}\design\demos\asm.

| Mnemonic | Meaning |
| --- | --- |
| noop | No operation. |
| r5 <- {0..15} | Assign the selected output register a 4-bit value. |
| r5 <- r0 | Assign the selected output register from the select input register. |
| r5 <- r0 OP r1 | Assign the selected output register the result of an operation on two input registers, where OP can be one of and, or, +, -. E.g. r3 <= r2 and r3, where r2 = “1010” and r3 = “0011” will assign r3 to “0010”. |
| r5 <- not r0 | Assign the selected output register bitwise not of the select input register. E.g. r4 <- not r1 when r1 = “0100” assigned r4 to “1011”. |
| r5 <- {0|1} > r0 | Shift the selected input register right one bit and append a 0 or 1 to the left. E.g. if r0 is “1111”, then r5 <- 0 > r0 gives r5 of “0111”. |
| r4 <- r1 < {0|1} | Shift the selected input register left one bit and append a 0 or 1 to the right. E.g. if r1 is “0000”, then r4 <- r1 < 1 gives r4 of “0001”. |
| if r5[3] | Conditional branch when register 5’s bit 3 is set. |
| if r0 eq r1 | Conditional branch a == b, See example below. |
| if r0 gt r1 | Conditional branch a > b, See example below. |
| if r0 ge r1 | Conditional branch a >= b, See example below. |
| wincr | Wait for one increment. |
| wincr <*n*> | Wait for *n* increments, *n*=0..29-1. |
| goto <*n*> | Goto code line, *n*=0..29-1. Typically this is a named label in the code so that the 9-bit programme counter can be inserted for you by the assembler.  The character $ evaluates to the current programme counter (line number) to allow relative addressing, e.g. “$-2” for two instructions previous. |

Integers can be specified in decimal or other bases like binary or hexadecimal. E.g. 10 = 0b1010 = 0xa. Note that integers must be constrained to the natural or non-negative range, i.e. zero or more, and most of the time in the range 0..15, except for the last two row of the table above.

**Conditional Branch Example:**

op\_ifgt:

if r1 gt r0 ; 1 > 0

leds <- 0b1000 ; pass

goto fail ; fail

fail:

leds <- 0b0111

goto $

The capabilities of the assembler exceed that which we really need for our limited processor. The original code was forked on GitHub in order to add new output formats to be friendly with reading by VHDL test benches.

**Advanced Tip:**

The ScratchVHDL version of the RISC CPU is limited to using a bit vector to represent the “op codes”. This makes decoding the instruction hard. By changing to the non-Scratch version an enumerated type is used for the op codes, make it easier to debug. To make this change, in ModelSim execute a command like the following with your desired instruction file:

change\_asm {F:\Compile\ModelSim\projects\button\_leds\instr\_files\knight\_rider\_start\_stop.o} 1

## Installation

### Board Support

The code will currently work with two revisions of the Zybo board.

1. Zybo Rev B, requires Zybo-Master.xdc
   * <https://www.digikey.co.uk/en/products/detail/digilent-inc/410-279/4840864> (Obsolete)
2. Zybo Z7 Rev D, requires Zybo-Z7-Master.xdc
   * <https://www.digikey.co.uk/en/products/detail/digilent-inc/410-351-10/7652757> (Current)

To correctly compile for a board, we provide the TCL function to run in Vivado as shown next. This function will swap the constraints file over so that the correct FPGA pins are used.

#### Rev B:

switch\_board "Zybo-Master"

switch\_board "Rev B"

#### Rev D:

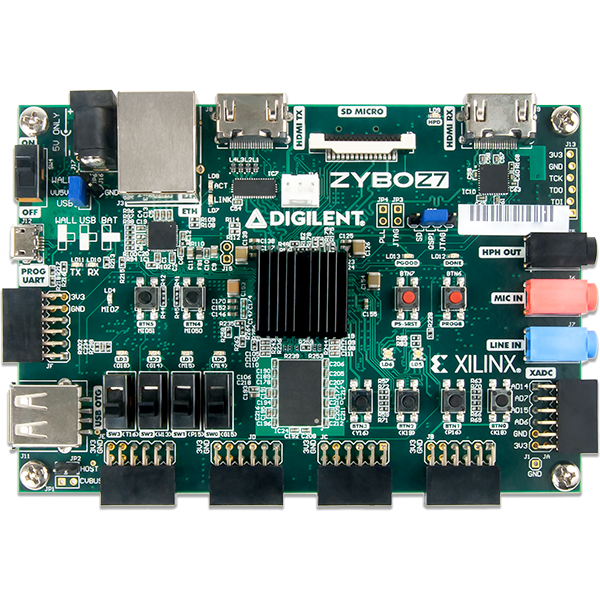
switch\_board "Zybo-Z7-Master"

switch\_board "Rev D"

### Hardware Connectivity

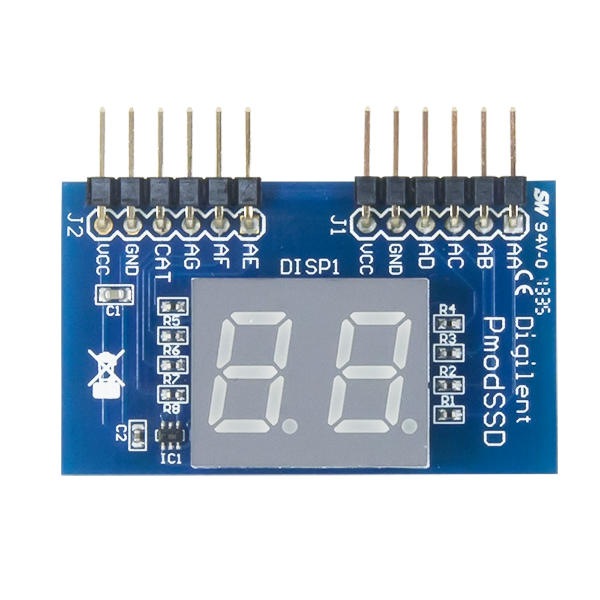
* USB Cable
* 7 Segment Display is optional and provides a number to reflect the LED status. It is connected to the Pmod Connectors JD & JE, upper row of pins.

Pmod connector pin out:



**JD**

**JE**

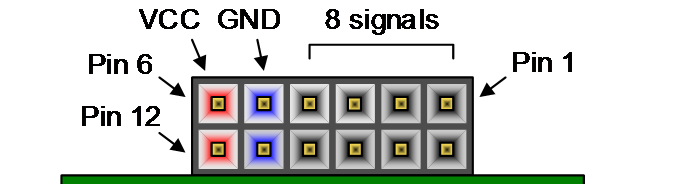


**USBJB**

**Clear FPGA**

**FPGA Programmed**

**Power OK**



### Software Installation

These FPGA tool versions have been shown to work in the Scratch VHDL project.

1. Microsoft’s VSCode, latest update
2. Modelsim Version 2020.1 (The *free* Intel FPGA Starter Edition)
3. Vivado Version 2022.1

### Scratch VHDL Project Installation

Once these steps haves been run on a clean clone there should be no need to repeat them. From a clean clone:

1. Fetch the latest assembler with fetch\_bin.cmd.
2. Install the VSCode extensions the Scratch VHDL project relies on with install\_extension.cmd.
3. Rename ${scratch\_vhdl}\design\config.cmd.editme to config.cmd and edit the file in a text editor to provide the paths used in your local tool installations.
4. Execute ${scratch\_vhdl}\design\asm\_compile.cmd to assemble the example instruction files so they can be selected for use when compiling the RISC CPU.
5. Run ${scratch\_vhdl}\fpga\VHDL\Local\modelsim\_compile.cmd to compile a required test bench library.
6. Vivado Project creation using vivado\_project.cmd.

## The Authors

Scratch VHDL was co-created by Joseph and Philip Abbey as an Open Source software project with documentation at <https://house-of-abbey.github.io/scratch_vhdl/> and source code at <https://github.com/house-of-abbey/scratch_vhdl>.

1. Equivalent to TCL “set\_property top zybo\_ scratch [current\_fileset]”. [↑](#footnote-ref-1)
2. Equivalent to TCL “set\_property top zybo\_risc\_cpu [current\_fileset]”. [↑](#footnote-ref-2)
3. Equivalent to TCL “set\_asm\_file {traffic\_lights}”. [↑](#footnote-ref-3)