I2C Wishbone Slave Verification and RVfpga VeeR EL2 SoC Integration

# Overview

The I2C (Inter-Integrated Circuit) protocol lets a master device communicate with multiple addressed slave devices over an I2C bus using a data line (sda) and a clock line (scl). The integration of an I2C master allows for peripheral devices such as sensors or other IC’s to be interfaced with the SoC.

The I2C device is built across three modules:

* **i2c\_core.sv** - Handles the fundamentals of the I2C protocol and has individual ports for each I/O signal.
* **i2c\_memory\_map.sv** - Responsible for grouping the i2c\_core I/O into registers. For example, single bit inputs such as the enable, start, stop and read/write bits are grouped into a single byte register. This is helps consolidate the memory space that the device occupies.
* **i2c\_wishbone.sv** - i2c\_memory\_map is wrapped in a wishbone interface (i2c\_wishbone.sv). The wishbone is a standardised lightweight interface designed for communication between modules on a chip.

I2C\_wishbone

I2C\_memory\_map

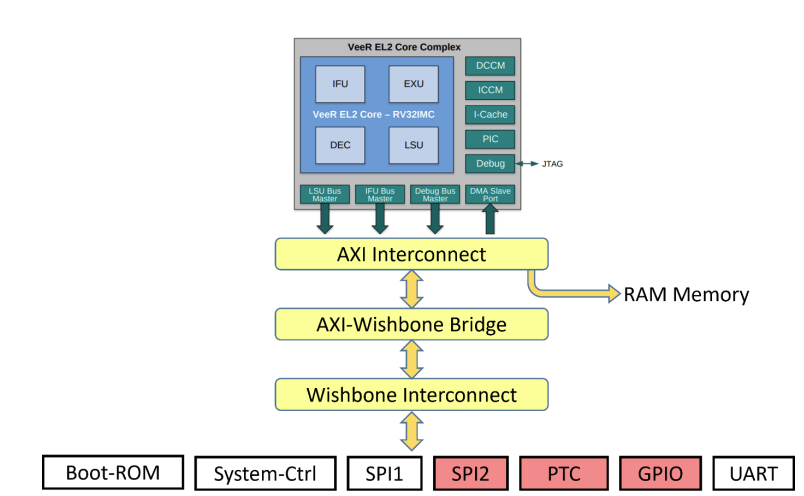
I2C\_core

Wishbone Interconnect

I2C Module

Splitting the design across three layers hierarchical layers gives the I2C design some degree of modularity and re-usability. For example, the i2c\_core and i2c\_memory\_map could be re-used in a different on-chip communications interface.

The wishbone interface allows the VeeR EL2 processor core to interact with the I2C module. Read the RVfpgaEL2 System Overview section of the Rvfpga\_getting\_started\_guide to understand how the wishbone peripherals physical interface with the processor.



I2C

# Wishbone Slave

Whereas the I2C protocol is designed to enable communication between IC’s, the wishbone is specifically designed to be a lightweight interface between modules **within** an SoC. In this case, the wishbone interface enables communications to the VeeR EL2 processor which uses the Wishbone bus to access memory-mapped registers, allowing the processor to read and write to a peripheral as if it were accessing registers within its own address space.

Wishbone Interface Signals:

* **CLK (Clock)**: Synchronises data transfer and operations across the Wishbone bus.
* **RST (Reset)**: Resets the Wishbone interface and connected components to their initial state.
* **ADR\_O (Address Output)**: Specifies the memory or register address being accessed.
* **DAT\_I (Data Input)**: Carries data from a slave to the master during a read operation.
* **DAT\_O (Data Output)**: Carries data from the master to a slave during a write operation.
* **WE\_O (Write Enable Output)**: Indicates whether the current operation is a write (when high).
* **STB\_O (Strobe Output)**: Signals that the master is requesting a data transfer.
* **ACK\_I (Acknowledge Input)**: Indicates that the slave has completed the requested operation.
* **CYC\_O (Cycle Output)**: Marks the duration of a valid data transfer on the bus.

A screen shot of a computer

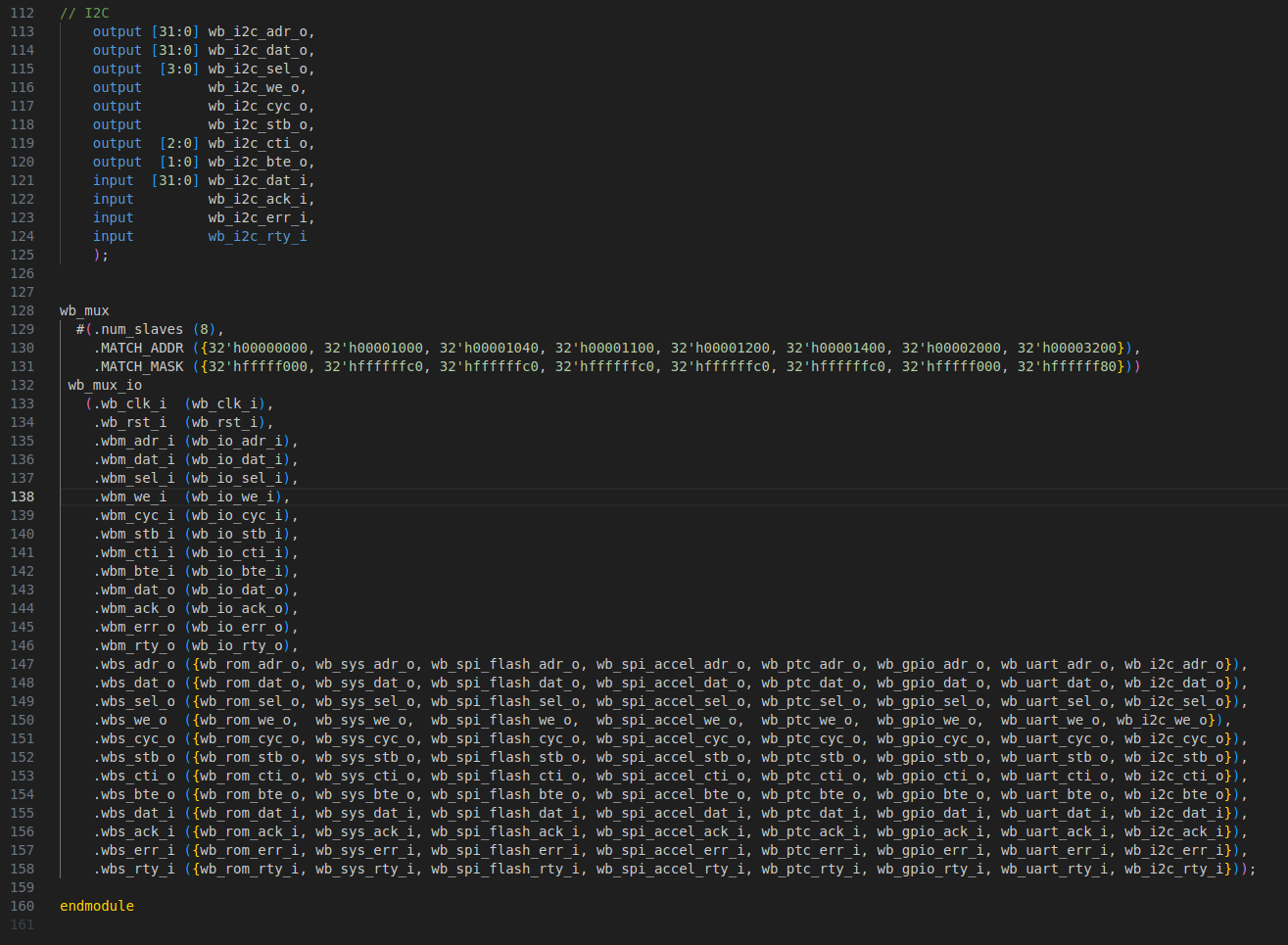
Description automatically generatedLooking at the I2C wishbone module (src/VeeRwolf/Peripherals/i2c/ i2c\_wishbone.sv), we can see that all these signals are defined in the I/O list.

Then looking at the src/VeeRwolf/Interconnect/WishboneInterconnect folder, we can see that two wishbone interconnect files exist. ‘wb\_intercon.v’ defines the wb\_intercon module which acts as a wrapper to connect multiple peripheral devices to a large wishbone multiplexer (wb\_mux) which is responsible for directing the wishbone signals to the correct device. The mux has a set of wishbone master signals (prefixed with “wbm”) connected to wishbone signals labelled ‘io’, these are connected to the AXI-Wishbone bridge. The mux also has slave signals (prefixed with wbs) which are connected to a concatenation of wishbone signals from different peripheral devices.

wb\_intercon.vh is a Verilog header file that instantiates an instance of the wb\_intercon module as well as defining a bunch of peripheral signals. These signals are connected to the wb\_intercon during its instantiation. The header file is included in the veerwolf\_core.v, exposing the same signals connected to the wb\_intercon, allowing them to also be connected to peripheral devices instantiated in the veerwolf\_core module.

To add the I2C module to the wishbone interface, we must first modify the wb\_intercon module inside the wb\_intercon.v file. We must add additional ports to the modules I/O list and then add these signals to the concatenation of peripheral wishbone signals connected to the slave ports of the wb\_mux.

In the wb\_mux parameters, we can also increase the number of slave devices by one. In the next chapter discussing memory mapping, we will also modify the MATCH\_ADDR and MATCH\_MASK parameters.



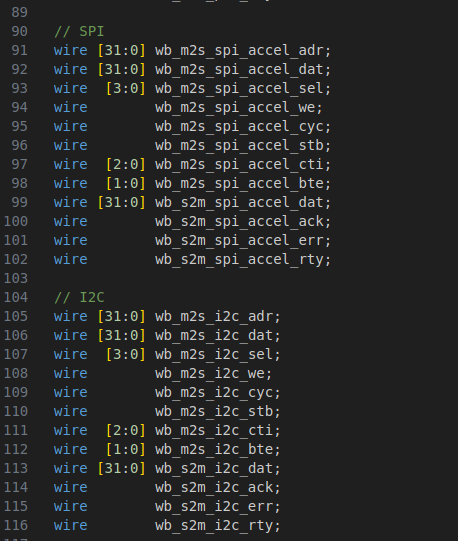
New ports added to the I/O list for the I2C module. These are exactly the same as the ports defined for the other wishbone peripherals but use a different label, in this case ‘i2c

Increased number of slave devices from 7 to 8

Wishbone master signals. In the veerwolf\_core module, these signals are connected to the AXI-Wishbone bridge. Remember the VeeR EL2 core doesn’t have a direct wishbone interface and instead uses an AXI interface to control the wishbone bus.

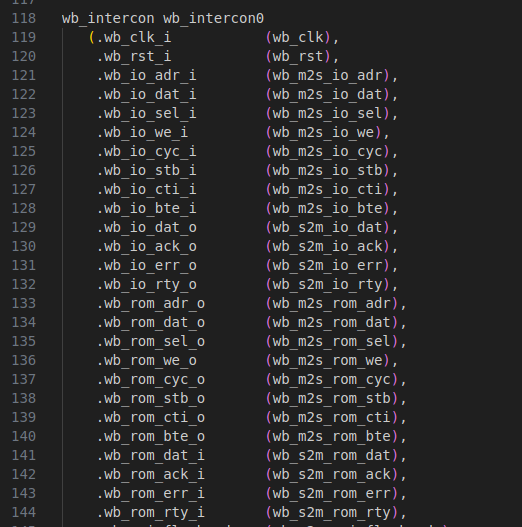
Wishbone slave signals. Add the new peripheral signals defined in the I/O list to the end of the concatenation

Now we need to modify the instantiation of the wb\_intercon module inside the wb\_intercon.vh header file.

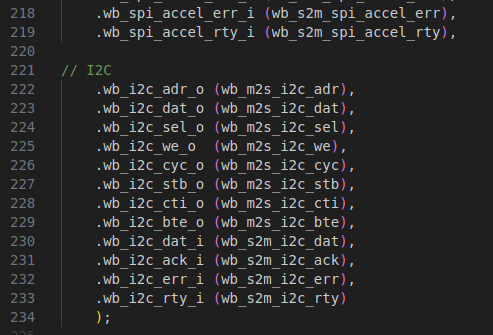


Create a new set of wishbone interface signals. Again, these are identical to the signals for the other peripheral devices so you can copy-paste but remember to change the label. In this case the label is ‘i2c’

Now, still working in the wb\_intercon.vh file, scroll down to the wb\_intercon instantiation and connect these new signals to the new ports we defined in the wb\_intercon module.



Scroll to bottom of instantiation

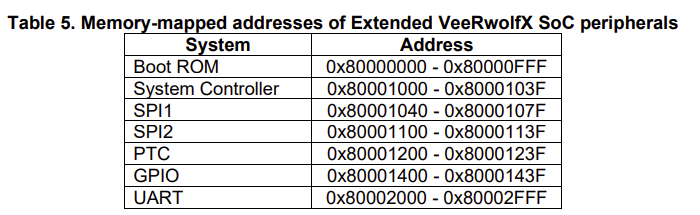


Finally, the I2C module can be instantiated in the veerwolf\_core module and connected using the wishbone signals defined in the wb\_intercon.vh file. Since the I2C master module is designed to function using an 8-bit wishbone bus (the SoC uses a 32-bit wishbone bus) the data in and data out signals can be truncated to just 8 bits. The number of required address bits depends on the number of registers that need to be addressed. In this case only 6 address bits are required, this will be explained in the next chapter.

A screen shot of a computer program

Description automatically generated

# Memory Map

In order to use the wishbone bus to manipulate the internal registers of the I2C module, we assign addresses to individual registers. We want the processor to be able to manipulate these registers, therefore the addresses we assign must fit into the processors address space without any collisions. The VeeR EL2 reserves addresses from 0x80000000 – 0x8FFFFFFF for peripheral devices. The Rvfpga\_GettingStartedGuide describes the address spaces of the original wishbone peripheral devices.

We can choose to allocate any base address that does not coincide with the address space of an existing peripheral. For the I2C device the base address was set at 0x80003200. Since the wishbone interface uses 32-bit data lines, every time a transaction is initiated, the wishbone will attempt to read or write to 32 bits (4 bytes) of memory starting at the address set on the wishbone bus address line (adr\_i).

For example, if a write to slave transaction is initiated with an address of 0x80003200, the data line will be filled with 32-bits from 0x80003200, 0x80003201, 0x80003202 and 0x80003203. This means that despite the internal registers only occupying 8-bits (1 byte) we must still separate them into 4-byte chunks. (Note: The wishbone interface does include a SEL signal that can be used to determine which byte is actually selected. This is useful in a memory confined system, but we have lots of memory to play with so separating the addresses is the simplest solution).

The table below shows the memory map of the I2C device, note how the addresses are separated by 4 bytes. This includes all the internal I2C registers we want to access via the processor.

A table with text on it

Description automatically generated

We therefore require an address space that spans from 0x80003200 to 0x80003214, a total of (0x14 + 0x3 = 0x17) bytes.

We can then add the base address (0x80003200) to the MATCH\_ADDR parameter concatenation in the wb\_intercon.v file. The wishbone interface knows that all peripheral devices must be within the address space 0x80000000 – 0x8FFFFFFF, so we can omit the leading 8.

The wishbone multiplexer uses the match address to determine which slave device to select during a wishbone transaction. This is done by comparing the address set on the address line with the addresses in the MATCH\_ADDR field. However, there is only one address in the MATCH\_ADDR field per peripheral and the I2C device has 7 different addressable registers! This is where the MATCH\_MASK is used. It determines which bits are considered during the address comparison by performing a logical AND operation between the MATCH\_MASK and the address on the wishbone. It compares the resulting value with the MATCH\_ADDR to determine which peripheral the multiplexer should select.

If you want to be able to address 8 bytes then you would set the MATCH\_MASK to 0xFFFFFF8. For our case we will need a minimum of 0x17 or 23 bytes so we will set the MATCH\_MASK to 0xFFFFF80 to give a total addressable space of 0x80 or 128 bytes. Again, the leading 8 can be omitted from the MATCH\_ADDR and MATCH\_MASK.

A screen shot of a computer program

Description automatically generatedSince we only need 5 bits to represent the 0x17 byte address space we can truncate the wishbone address line being passed to the I2C module during the instantiation.

Now we have a way of directing read/write transactions to the I2C peripheral, but we need a method to allocate the contents of the wishbone data bus to the correct internal registers inside the I2C module during a write transaction and perform the reverse during a read transaction. This is most easily handled in RTL using a case statement.

The next image shows the handling of the wishbone interface in the i2c\_wishbone.sv file.

A screen shot of a computer program

Description automatically generated

A screen shot of a computer screen

Description automatically generatedFirstly, the stb and cyc signals are checked, if both are asserted then a valid wishbone transaction is occurring. Next the write enable bit (we) is sampled to check whether the operation is a read or write. In both cases the address on the wishbone address bus is compared with different macros. These macros are defined at the top of the i2c\_wishbone.sv file.

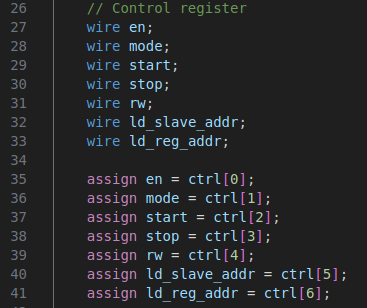
The macros are used to define the addresses assigned to the internal registers, notice how they match the final two digits of the addresses in the register map table. (Using macros to define the addresses allows them to be easily reused in a testbench later).

Note that the wishbone address is being compared with each of the addresses assigned to different registers in the memory map. In the case of a read operation (we = 0), if the address matches, the relevant register is broadcast onto the wishbone data out bus (dat\_o). In the case of a write operation (we = 1), if the address matches, the dat\_i contents is assigned to the corresponding internal register.

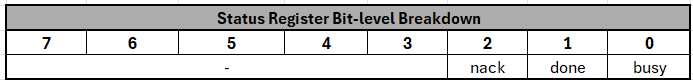
The i2c\_memory\_map module is responsible for dissecting the control input register into its individual signals to pass to the i2c\_core module.

A table with text and numbers

Description automatically generated



Similarly, the i2c\_memory\_map module combines the output signals from the i2c\_core module to make up the status register.



A screen shot of a computer

Description automatically generated

# Firmware

Add section of creating new platformIO project

Now we’ll see how we can use the memory mapping to manipulate the internal registers and control the I2C module using programmatic code that will run on the VeeR EL2. In this example we will use C code although the same could be accomplished using assembly only much more tedious. Inside firmware/src there are three C files, main.c, simple\_i2c\_hal.c and simple\_i2c\_hal.h. main.c is the code that is compiled and loaded into RAM, I2C\_HAL.c contains functions that abstract out the operation of the I2C hardware, hence the name I2C hardware abstraction layer. The simple\_i2c\_hal.h is a header file that is included in main.c to provide access to the HAL functions. The I2C\_HAL is used in the main function to achieve higher level tasks.

Let’s examine the simple\_i2c\_hal.h file. At the top, we define the register map using macros. Note how these addresses match with the memory map shown in the previous chapter. We also define bit masks for both the control and status registers. This provides a simple and verbose way to access particular bits within a register.

A screenshot of a computer program

Description automatically generated

The value of the mask corresponds to the position of the bit. For example, the I2C enable bit is the 7th bit of the control register (indexed from zero). Therefore, the enable mask will have a hex value of 2^7 = 0x80.

We can also define macros to perform read/write operations to/from registers.



Finally, we prototype the functions we declare in the simple\_i2c\_hal.c file.

A black background with blue and white text

Description automatically generated

Now looking at the simple\_i2c\_hal.c file, we first import the header file to give access to the address, mask and instruction macros.  


To read an address,



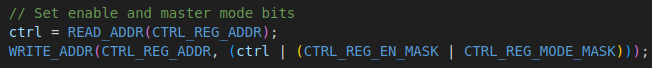
(remember the wishbone bus is 32-bit, so the size of the variable type (uint8\_t) can be increased if you need the extra length).

To write to an address,

A black background with blue and green text

Description automatically generated

However, if we want to write a sub-section of the register (i.e. a single bit), we must ensure that the remaining bits are preserved. To do this we must use the read, modify, write method.



In this case we want to set the enable and mode bits of the control register. First, we read the control register and store it to a variable called ctrl. Then we combine the enable and mode bit masks using a logical OR operation (CTRL\_REG\_EN\_MASK | CTRL\_REG\_MODE\_MASK). This creates a new bit mask with both the enable and mode bits set. We can then perform another logical OR with the new bit mask and the value that we read from the control register. This gives a control register value with the enable and mode bits set and preserves the state of all the other bits. Finally, this value is written back to the control register.

We can also check whether a particular bit is set by performing a logical AND operation with the relevant bit mask.

A screen shot of a computer screen

Description automatically generated

This do-while statement will block the program until the done bit has been set by the I2C device. While this method works, it is inefficient as it will block the processor. Instead, it’s a good idea to use interrupts to signal to the processor when the I2C transaction is complete, and the RX register can be read but this will require addition hardware.

These basic building blocks can be used to build software control functions for wishbone peripheral devices.

# FPGA Pinout

In order to allow physical devices to be connected to the SoC, the FPGA pinout must be modified to accommodate the sda and scl lines. The Nexys boards connects a number of general-purpose pins from the A7 FPGA to physical Pmod connectors.

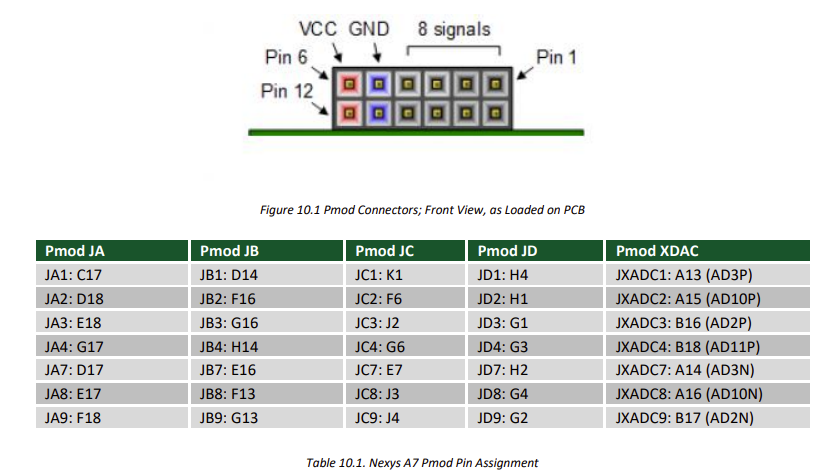


Figure 1:https://digilent.com/reference/\_media/reference/programmable-logic/nexys-a7/nexys-a7\_rm.pdf

The FPGA pinout is defined in the rvfpganexys.xdc constraints file. To connect the I2C bus lines to pins 3 (G1) and 4 (G3) of Pmod JD, the following lines can be added to the constraints file.



Figure 2: I2C ports defined in the rvfpganexys.xdc constraints file

I/O ports must also be defined at the top level of the HDL design (rvfpganexys.sv). Since I2C lines are bidirectional, the ports must be defined using the ‘inout’ specifier type. First define the I2C ports in the I/O list of the top module (rvfpganexys.sv) using the same names that are used in the rvfpganexys.xdc, as this is how the top level I/O is connected to the physical FPGA pins.

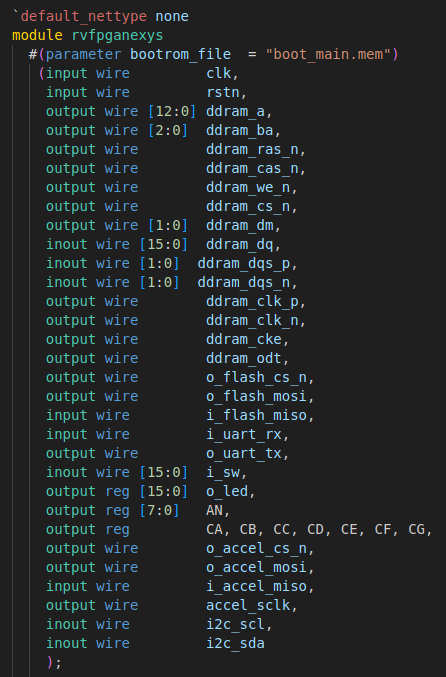


Figure 3: I2C ports added to I/O list of project top module (rvfpganexys.sv)

Since the I2C module will be defined in the VeeR EL2 processor top module (VeeRwolf/veerwolf\_core.v), I2C ports must be added to this module as well.

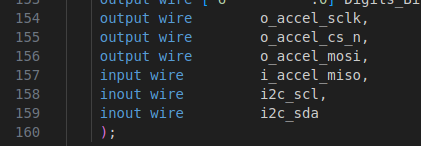
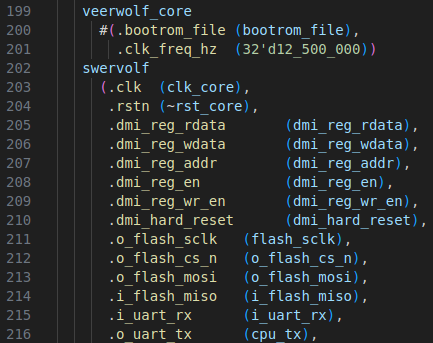


Figure 4: I2C ports added to I/O list of VeeR EL2 top module (veerwolf\_core.v)

Back in the rvfpganexys.sv file, the I2C ports from the top-level module can be connected to the VeeR core. Find the instantiation of the veerwolf\_core module inside the rvfpganexys module and connect the I2C wires.



Scroll to bottom of instantiation



Now the I2c lines have been connected all the way from the physical FPGA pins to the VeeR core and are ready to be connected to the I2C Master module. But first, the wishbone interface needs to be considered.

# Simulation

**I2C Core Testbench (**simulation/i2c\_core**)**

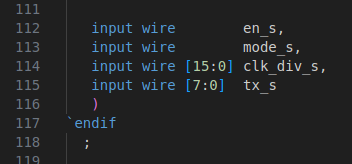
**I2C Wishbone Testbench (**simulation/i2c\_wishbone**)**

**RVfpga SoC Simulation (**simulation/rvfpga\_soc\_sim**)**

The RVfpga package ships with a Verilator testbench that allows the complete integrated system-on-chip to be simulated using custom software running on the simulated VeeR EL2 processor. This allows us to verify the wishbone integration and the C code. For this to work, make sure you have installed Verilator, you can follow the installation instructions in the RVfpga getting started guide.

For simulation the RVfpga project uses a different top module (SimulationSources/rvfpgasim.v). We will first modify this top module to add an additional slave I2C core module, similar to what we did in the previous two testbenches.

We will need add the slave control signals to the rvfpgasim I/O list. Make sure you add ports outside of the ifdef statements.



Scroll to bottom of rvfpgasim I/O list

Above the veerwolf core instance, we can define the i2c signals and simulate the weak pullup resistors.

A screen shot of a computer program

Description automatically generated

Inside the veerwolf\_core instantiation, we can connect the i2c wires to the i2c ports we added to the veerwolf core module in chapter 5.

A screen shot of a computer program

Description automatically generated

Now at the bottom of the rvfpgasim module we can instantiate an instance of the i2c\_core module to act as the slave device and connect the control signals we just created in the rvfpgasim I/O list. We can also define and attach some additional signals that will be useful for debugging.

A screen shot of a computer program

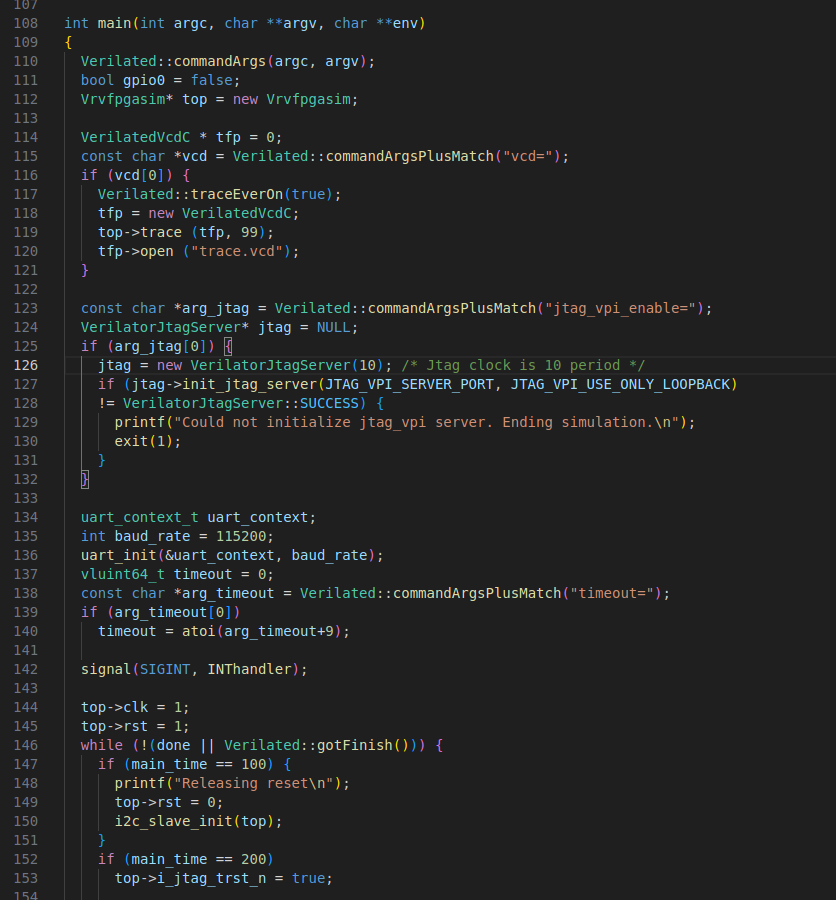
Description automatically generated

Now we have modified the top-level module used in the Verilator testbench. Verilator is an open-source RTL simulator built using C++. When we build the simulation in a later step, Verilator will convert the top-level Verilog module into a C++ class (Vrvfpgasim). The simulation testbench is a C++ file called tb.cpp that uses an instance of the Vrvfpgasim class called ‘top’. At the start of the simulation, the reset signal is held high while simulated ROM module is loaded with compiled code. This is the program cod that will execute on the simulated EL2 core. Once the ROM is initialised the reset is released and the testbench executes.

A screen shot of a computer code

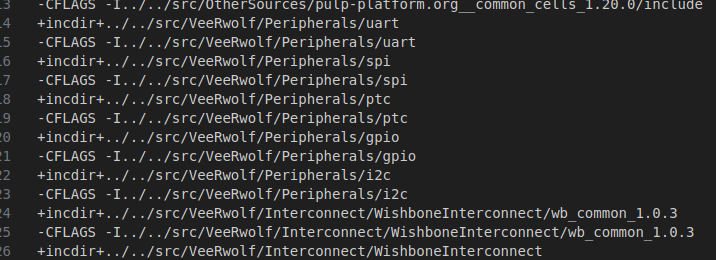
Description automatically generatedWe now need to modify the Verilator C++ testbench (tb.cpp) to configure the slave device once the reset signal is released. To do this we can create a new function inside tb.cpp. This function will take a pointer to the Vrvfpgasim class object (this object is called ‘top’). Vrvfpgasim object contains an attribute for each of the ports to the rvfpgasim module which can be accessed using the ‘->’ operator.

We then need to call this function from within the main function after the reset is released.

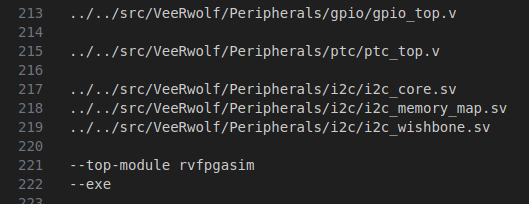


Now we have modified the top simulation module to add an additional slave I2C device that we can use to verify the I2C device integrated into the SoC. We have also modified the Verilator testbench to initialise the slave I2C core after the reset has been released.

The Verilator simulation uses the rvfpgasim.vc to define all the Verilog and C++ simulation sources. We need to add the additional I2C files to make sure they are included in the simulation build. First add additional +incdir+ and -CFLAGS lines for the i2c peripheral directory.



The towards the bottom of the rvfpgasim.vc file add the relative paths to each of the i2c design files.



Now we can build the Verilator simulation executable file. To do this open a terminal in the rvfpga\_soc\_sim directory and enter the following commands:

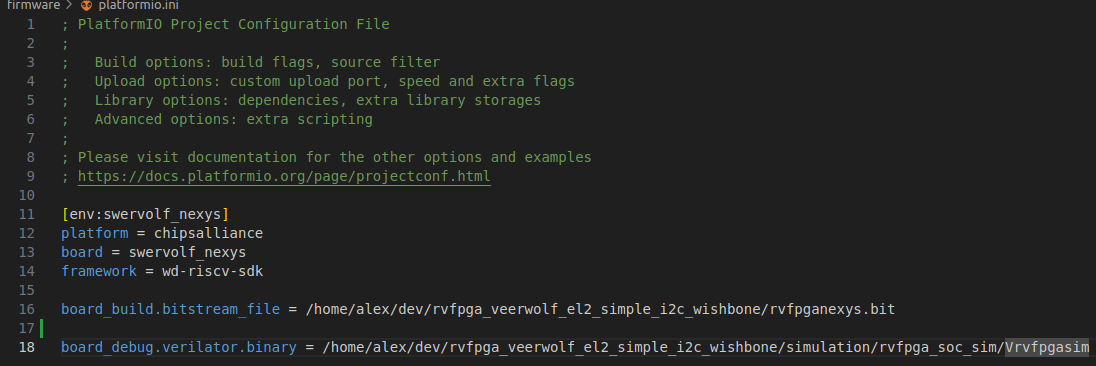
make clean

make

These two commands interact with the Makefile inside the rvfpga\_soc\_sim. The first cleans the directory of any already built files. If you make changes to the design/simulation sources or the C++ testbench, its important you use this command to clean the directory before rebuilding.

The make command first generates a bunch of C++ files from the Verilog design sources before compiling them into an executable called Vrvfpgasim.

Now navigating back to the VSCode platformIO project containing the C code firmware, open the platformio.ini file and add the following verilator variable definition with the absolute path to the Vrvfpgasim executable we just built.



Now in the platformIO: project tasks window, click ‘generate trace’. Pressing this button performs a number of back-to-back tasks. First the source code is compiled into a .elf file using the riscv64 toolchain. The .elf file is then used to generate a hex dump in the form of a Verilog header file (.vh). Both of these files can be found inside the .pio/build/swervolf\_nexys folder. Finally, the verilator simulation is run by calling the Vrvfpgasim executable with the .vh file passed as an argument. The simulation will generate a trace.vcd file (value change dump) which can be loaded using GTKWave to view the simulation waveforms.

To view the waveforms using GTKWave, run the following command from the platformIO project directory

gtkwave .pio/build/swervolf\_nexys/trace.vcd