Programmable Wave Generator Verification Environment

XILINX®

Programmable Wave Generator Verification Environment

Verification Environment Description

The “Programmable Wave Generator (wave_gen)” design used in the FPGA courses was designed with an extensive verification environment. This environment allows the verification of the full design, as well as various sub-blocks on which it is based. This verification is an extensible, self checking environment that allows for the rapid implementation of many test suites.

The verification environment has two main functions. The first is to ensure that the complete design, “wave_gen” functions correctly. This design is the underlying code base for the majority of the labs in the ISE Design Entry, Fundamentals of FPGA Design, Designing for Performance, Advanced FPGA Implementation, and Chipscope classes. As each generation of these classes is released, it is essential that the code base used for the labs not only meet the educational needs of the labs, but also be correct RTL code that accomplishes its stated purpose. As a result, this verification environment will include a set of tests that rigorously verifies the functionality of the full wave_gen design, as described in the design_descript_doc. While the HDL code for wave_gen is available in both Verilog and VHDL for lab use, the complete design environment is only available in Verilog. This is acceptable since the complete verification environment is intended primarily for internal use only (not for use in class labs), and hence can be used to verify both the Verilog and VHDL RTL code using mixed language simulation.

In addition, a portion of the verification environment itself will be used as the basis of several labs. The complete design and verification environment may be too complex for some classes, and hence a portion of the verification environment can be used to verify the UART receiver (uart_rx) portion of the design. This portion will use some of the same sub-modules as the full verification environment, wrapped in a smaller testbench, but will be available in both Verilog and VHDL.

The testbench module (tb_wave_gen) implements much of the functionality required to support tests. It consists of drivers and monitors for the pins of the UUT, and higher level functionality that allows the implementation of self checking tests. Figure 1 shows the architecture of the testbench. The clock generation is procedurally implemented in the testbench, and implements a simple periodic clock.

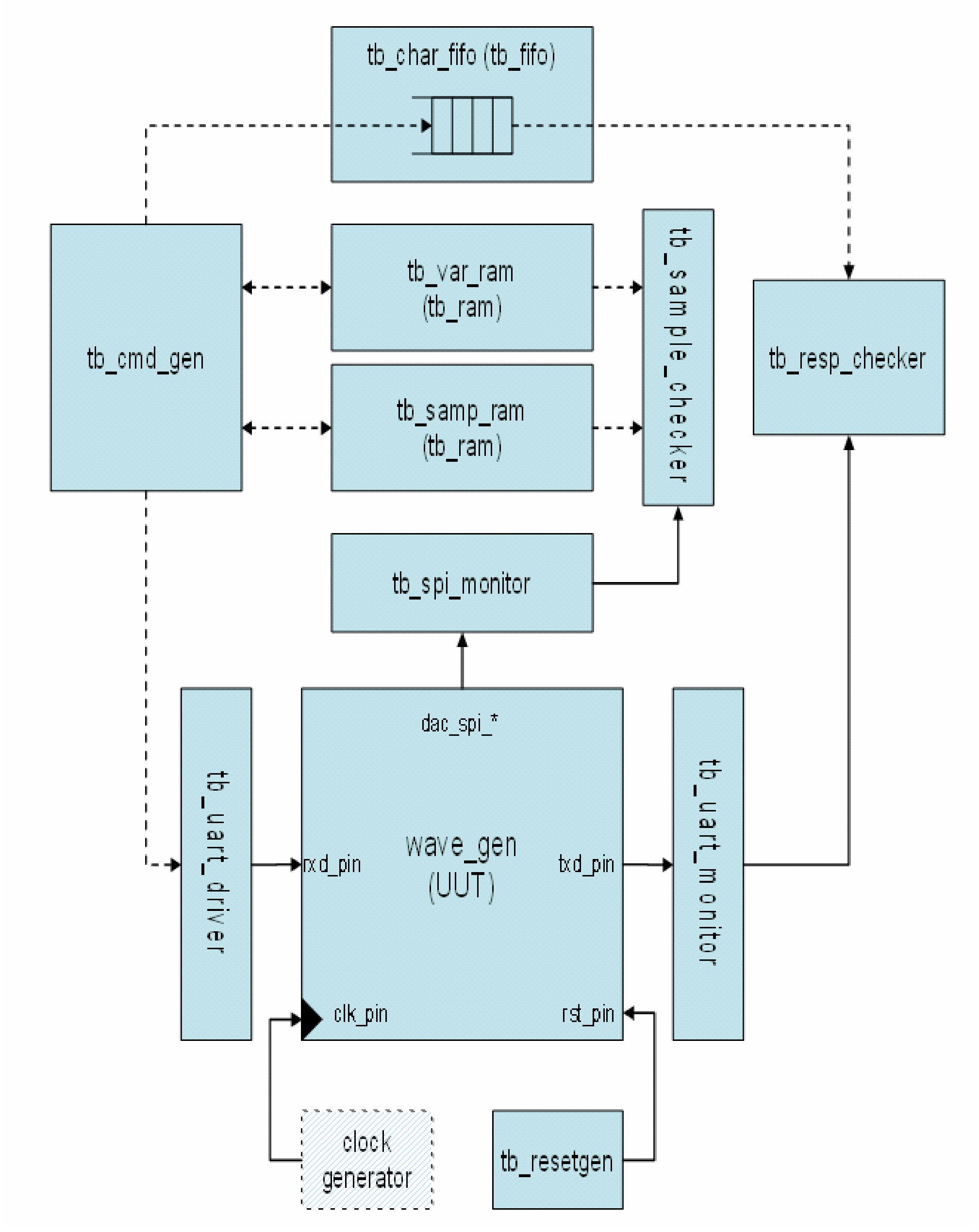

Figure 1: Testbench Architecture for the Programmable Wave Generator (tb_wave_gen)

The testbench is typically instantiated in a test, which can then interact with the sub-modules of the testbench. For this design, the normal interaction with the testbench is at the level of the commands; the same commands that the user of the Wave Generator has for controlling its functions.

For each command listed in table 1 of design_descript_lab.pdf, the module tb_cmd_gen has a task for implementing the command. In general, this involves the following steps

- Generating the set of ASCII characters that forms the command to send to the Wave Generator (UUT)

- Checking the validity of the arguments to determine which type of response the UUT will generate (error, acknowledgement, or data response).
- When necessary, querying the internal state of the testbench, managed by tb_samp_ram and tb_var_ram, to determine the appropriate data response
- When necessary, modifying the internal state of the testbench
- Generating the set of ASCII characters which is the expected response from the command, and placing them in the tb_char_fifo for comparison with the UUTs response

The test module is a module for implementing a specific sequence of operations to the UUT through the testbench. In general, the test consists of one main block of procedural code (an initial statement), which invokes a set of commands. These commands should exhaustively exercise the functionality available through the commands.

Additionally, the test may bypass the normal command generation process, and generate sequences of input characters and expected output characters, which can (for example) be used for verifying the responses to parser errors (illegal commands, illegal characters in commands, etc…). Legal commands should not be issued by this mechanism, since commands issued this way will not query or update the internal state managed by the testbench.

The next sections will document the various testbench modules and the tasks used to control them. The testbench module itself contains a single task:

Initialize the testbench structures: init()

This task will initialize all the testbench structures, namely the two data RAMs, including setting the programmable minimum and maximum values for the variables in the tb_var_ram. It will then assert the reset for 20 clocks, then deassert it and enable all the checkers. This task should be called at the beginning of all tests.

Reset Generator (tb_resetgen)

This module controls the reset input (***rst_pin***) to the UUT, based on the testbench clock. It is controlled by a single task, **assert_reset(num_clk)**, which asserts the resets for **num_clk** clock cycles and then de-asserts **reset**. The reset generated is active high.

UART Driver (tb_uart_driver)

This module generates the RS232 input to the UUT. It has a single output port ***data_out*** which is the serial data stream. In essence, this module is a behavioral implementation of a UART transmitter. There are several tasks that can be invoked for sending serial data.

Send a Character at a Specified Bit Rate: send_char_bitper(char, bit_per)

This task is the most general task in the module. As its name implies, it sends the character ***char*** at the bit rate specified by ***bit_per***. The ***char*** input is an 8 bit character to be sent over the serial line, and is sent according to the RS232 protocol; Start bit (0), eight data bits (LSbit first), and stop bit (1). The bits are sent using the bit period specified by the ***bit_per*** input; which is measured in increments of the base time period specified by the `timescale for this module (which is 1ns).

Most of the other tasks in this module call this task, but this task can be invoked directly to (for example) verify that UUT responds properly to characters sent at slightly different bit periods (since the RS232 protocol allows for bit rate variations of up to 2.5%).

Send a Character at the Default Bit Period: send_char(char)

This task simply invokes the **send_char_bitper** task using the default bit period. This avoids needing to pass the ***bit_per*** input on each call. The default bit period is set using the **set_bitper** task.

Set the Default Bit Period: set_bitper(new_bit_per)

Sets the default bit period to ***new_bit_per***. This period is then used in all subsequent calls to the **send_char** task.

Send a Character and Push it into the FIFO: send_char_push(char)

This task is not used in the full verification environment; it is only used in the **uart_rx** verification environment. This task calls the **send_char** task, and then pushes the character sent into the **tb_char_fifo**. In the full verification environment, all data in the character FIFO is managed by the ***cmd_gen*** module.

UART Monitor (tb_uart_monitor)

This module receives RS232 format serial data on a single input **data_in**, and converts it to characters for use by the rest of the test bench. In essence, this module is a behavioral implementation of a UART receiver. When a character is received, it is placed on the **char** output of this module and the **char_val** output is pulsed to for one time unit (1ns). The output of this module is fed to the response checker.

This module also does the (minimal) protocol checking permitted by the RS232 format; framing error checking. If the 9th bit received after the START condition is not a 0 (indicating a valid STOP bit), then an error message will be printed.

The baud rate used by this module is set by the parameter BAUD_RATE, and is 57,600 by default.

This module has only one task:

Start RS232 processing: start()

This task has no inputs, and should be invoked at the beginning of a test. The value of the data_in pin will be ignored until the **start** task is invoked. This task is used to ensure that RS232 protocol checking and conversion start properly at the beginning of the simulation.

SPI Monitor (tb_spi_monitor)

This module receives SPI format serial data on the spi_mosi, spi_clk, and spi_csb pins, and converts them to a set of outputs. In essence, this is a behavioral implementation of an SPI receiver. The outputs of this module are:

- out_val: pulsed for 1ns when a new SPI sequence has been received

- out_cmd[3:0]: the command of the SPI sequence
- out_addr[3:0]: the address of the SPI sequence
- out_data[15:0]: the data of the SPI sequence

The output of this module is fed to the sample checker.

This module also checks that the SPI protocol on each sequence. Namely, this module verifies that each sequence has the proper number of bits (controlled by a parameter, but should be set to 32 for this design); since SPI accesses can be back to back, this essentially means that the chip select can only be de-asserted after a multiple of 32 bits. If the protocol is violated, an error message is printed.

This module has only one task:

Start SPI processing: start()

This task has no inputs, and should be invoked at the beginning of a test. The SPI input pins will be ignored until the **start** task is invoked. This task is used to ensure that SPI protocol checking and conversion start properly at the beginning of the simulation.

Character FIFO (tb_char_fifo)

This module is an instantiation of the module tb_fifo, which is a generic testbench FIFO. The FIFO uses the parameters WIDTH and DEPTH to configure the width of the data it can hold, and the maximum depth (number of entries) that the FIFO can hold. The actual number of entries that the FIFO can hold is DEPTH-1. The data in the FIFO is held in an internal array (Verilog memory or VHDL array), and is accessed in a circular manner (modulo DEPTH) using a head and tail pointer that is also held internally. Data enters the FIFO at the head and leaves at the tail.

Accesses to the FIFO are managed by functions and tasks:

Calculate the "next" address: next_ptr(ptr)

This function calculates the value of the next pointer. Since the pointers are circular indexes into a behavioral memory, this function simply returns (ptr + 1) % DEPTH.

Check for FIFO empty: is_empty(ignore)

This function returns 1 if the FIFO is empty, and 0 if it is not. The FIFO is known to be empty when the head pointer is equal to the tail pointer.

Since all verilog functions must have at least one input (for syntactical correctness), this module has an input called **ignore**, which is ignored (not used by the function). When the function is invoked, any value can be passed in to the **ignore** input.

Check for FIFO full: is_full(ignore)

This function returns 1 if the FIFO is full, and 0 if it is not. The FIFO is known to be empty when the act of incrementing the head pointer would make it equal to the tail pointer (essentially having the FIFO overflow). This would happen when next head pointer is equal to the tail pointer; in other words, when next_ptr(head) == tail.

This function also has a single input, which it ignores.

Pop a value from the FIFO: pop(ignore)

This function pops the value at the tail of the FIFO, and returns that value. If the FIFO is empty when this function is invoked, an error message is printed, and X is returned. Otherwise the value in the internal memory at address **tail** is returned, and the tail pointer is incremented; tail = next_ptr(tail).

Push a value into the FIFO: push(new_data)

This task pushes the value **new_value** into the head of the FIFO. If the FIFO is full when the task is invoked, an error message is printed, and **new_data** is discarded. Otherwise **new_data** is placed into the internal memory at address **head**, and the head pointer is incremented; head = next_ptr(head).

Sample RAM (tb_samp_ram) and Variable RAM (tb_var_ram)

These instances are both instantiations of the module tb_ram. The tb_ram module is a generic paramaterizable memory that can be used in testbenches to manage internal state. The RAM itself is a behavioral memory that can be parameterized as to both width (number of bits per entry) and depth (number of entries), using the **WIDTH** and **DEPTH** parameters respectively. Internally, it is implemented as three verilog memories; one for the actual values stored (**val**), one for the maximum allowable values for each value stored (**max_val)**, and one for the minimum allowable value for each value stored (**min_val)**. All three memories are the same size.

The tb_samp_ram instantiation is used to model the sample memory in the UUT. It is 16 bits wide and holds 1024 entries (like the sample RAM in the RTL). Samples stored in this RAM can have any legal 16 bit value, hence the min and max for this memory should be 0 and 65,535 respectively.

The tb_var_ram instantiation is used to model the three internal variables of the UUT; nsamp, prescale, and speed. These will be stored at locations 0, 1, and 2, respectively, and the memory will be configured with a WIDTH of 16, and a DEPTH of 3 (for the 3 variables). The min and max allowable values for these variables are described by the design_descript_lab document, but should be set as follows:

- nsamp: 1 to 1024
- prescale: 32 to 65,535
- speed: 1 to 65,535

The initialization of the content memory as well as the minimums and maximums will be done at time 0 by the testbench (thus alleviating the need of each and every testcase initializing the memory).

Access to the memory is provided via the following tasks:

Initialize the array: init()

This task sets all values of the memory to 0, all **min_val** values to 0, and all **max_val** values to 2^WIDTH-1. It takes no inputs or outputs.

Set the minimum and maximum value: set_min_max(addr, min_val, max_val)

This task sets the minimum and maximum values for the cell at address **addr** to the specified minimum and maximum values. These values are used by the **write** function to do range checking before accepting a new value (see the **write** function).

Write a value to the array: write(addr, value)

This function first tests that **value** falls within the legal range for the location at address **addr**, by comparing it against **min_val(addr)** and **max_val(addr)**. If it is, this task writes **value** to **val(addr)** and returns the return code 1'b1, otherwise it discards the value, and returns the return code 1'b0.

Read a value from the array: read(addr)

This function returns the value of the array at location **addr**.

Command Generator (tb_cmd_gen)

This module is the main interface for the testcase to the testbench; most of the operations done in a typical testcase will be done through the tasks of this module. This module implements the tasks that represent the Wave Generator command, and, as described in the introduction, do range checking, generation of the characters to send to the UUT (the command), queries the internal state to determine the response expected from the UUT, creates the expected output response to push into the character FIFO, and updates the internal state as required.

This module has several tasks:

Do a Command/Response pair: do_cmd(cmd,rsp)

This task manages the lower level string manipulation for sending a command and expecting a response. The string **cmd** is sent to the UUT (through tb_uart_driver), and the concatenation of **cmd** and **rsp** are pushed into the character FIFO as the expected response from the UUT (the UUT will echo the **cmd** to the output, and then append its **rsp**.

This task is invoked by all the specific commands, but can be called directly to verify syntax error parsing. For example, sending the command "*x" (which is a syntax error), will result in the response "-err\n"; thus the correct invocation to test this would be do_cmd("*x","-ERR\n").

Write to Sample RAM: write(addr,val)

This task implements the "*W" command.

When invoked, this task will first check that the **addr** argument is legal (in the range of 0 to 1023). If it is outside this range, it is known that the UUT will reject this command, and issue an error response. Therefore, this task will send the command to the UUT and expect the error response.

If **addr** is in the legal range, we know the UUT will update the sample RAM at address **addr** with the value **val** and return an acknowledgement response. Therefore, this task will modify the internal state of the testbench sample RAM by invoking tb_samp_ram.write(addr,val), send the command to the UUT and expect an acknowledgement response.

In either case, the do_cmd task will be invoked with the command being ("*W*aaaavvvv*", where *aaaa* is the hexadecimal representation of **addr**, and *vvvv* is the hexadecimal representation of **val**), and the expected response to be either the acknowledgement ("-OK\n"), or the error response ("-ERR\n"), depending on whether **addr** is in the valid range.

Read from Sample RAM: read(addr)

This task implements the "*R" command, and verifies that the UUT returns the correct result, as determined by the state of the testbench sample RAM.

If **addr** is in the legal range, this task will query the internal state of the testbench RAM to determine what the sample value at that location is currently set to. It will then issue the proper command ("*R*aaaa*", where *aaaa* is the hexadecimal representation of **addr**) to the UUT and will expect the command to be echoed and the response to be sent; if **addr** is in the legal range, then the reponse will be the data response ("-*hhhh ddddd*\n" where *hhhh* is the hexadecimal representation of the sample value and *ddddd* is the decimal representation), otherwise the error response will be expected.

Set nsamp, prescale, or speed: set_nsamp(val), set_prescale(val), set_speed(val)

All three of these tasks simply invoke the set_var task shown below with the **var** parameter set to 0, 1, or 2 respectively.

Set a variable: set_var(var, val)

This task implements the *N, *P, or *S commands. When invoked, this task first attempts to update the testbench internal state of the specified variable (**var**) with the new value (**val**). This is done by invoking the tb_var_ram_i0.write(var,val) function. The return code from this function indicates whether the **val** is a legal value for the specified value.

Regardless of the return code from tb_var_ram, the character sequence for the command is sent to the UUT, either "*N*vvvv*", "*P*vvvv*", or "*S*vvvv*", where *vvvv* is the hexadecimal representation of **val**. The expected response will either be the error response if the return code from tb_var_ram is 1'b0, or the acknowledgement response if the return code is 1'b1.

Get nsamp, prescale, or speed: get_nsamp(), get_prescale(), get_speed()

All three of these tasks simply invoke the get_var task shown below with the **var** parameter set to 0, 1, or 2 respectively.

Get a variable: get_var(var)

This task implements the *n, *p, or *s commands. It issues the appropriate command to the UUT, and expects a data response that is determined by the current value of the variable in the testbench internal state.

Sample Generator Control: samp_go(), samp_cont(), samp_halt()

These tasks control the sample checker. When the samp_go task is invoked, the **enable** pin to the sample checker is pulsed for 1ns. The samp_cont and samp_halt sets the **enable** pin high or low respectively.

Response Checker (tb_resp_check)

This module compares the characters coming out of the UUT against the characters stored in the character FIFO in the testbench. It has three inputs, **char, char_val**, and **frm_err**. On the rising edge of **char_val**, the 8 bit data on the **char** port is compared against the next expected byte from the testbench FIFO. If the data matches, an informational message is shown indicating the match. If the data does not match, and error message is printed, showing the expected and received byte. A parameter controls the delay between the rising edge of **char_val** and the actual check; this delay should be set to 0 for RTL simulations, but may be necessary to support post place and route simulations.

This module is intended to be used both in the full verification environment, as well as in the verification environment for uart_rx. In the uart_rx environment, the frm_err signal is monitored for a rising edge while checking is enabled, and prints an error if one is detected. In the full environment, the framing error checking is done by the tb_uart_monitor, hence the **frm_err** signal of this module should be tied to 1'b0.

This module has only one task:

Start data checking: start_chk(new_enabled)

This task controls the enabling of the checking mechanism. If called with the argument **new_enabled** set to 1'b0, the checking is disabled, otherwise the checking is enabled. No data checking will take place when the checker is disabled, which is the default condition. This task is normally called at the beginning of the testcase to enable checking once the UUT is out of reset, to ensure that no false error messages are generated at the beginning of the simulation.

Sample Checker (tb_samp_check)

This module checks the samples coming from the dac_spi_monitor, to determine if the expected samples are sent correctly.

When enabled (by a rising edge of the **enable** signal from the command generator), this module will expect a sequence of samples to be issued. Once enabled, the checker will query the internal state of nsamp, speed and prescale, to determine which samples are to be expected, and how fast they should be coming. As each sample is received, it is checked against the next expected sample, which this module reads from the testbench tb_samp_ram.

The arrival rate is also checked, as, once started, each sample should be received at intervals of clk_per * **speed** * **prescale**, where clk_per is 1e9/CLOCK_RATE, which is the clock period in nanoseconds; CLOCK_RATE is a parameter to this module. In order to accommodate sub-clock period variations, this module will allow up to ½ clock period of "slop" in the arrival time of the sample. Similarly, the expected arrival time of the first sample is monitored; since the latency of the command sequence is not specified, this module expects the first sample to arrive within a reasonable amount of time from when the **enable** signal is activated. This time is also parameterized, but should be set to approximately 40 clock periods – 8 clock periods for internal latency, and 32 clock periods for processing of the first sample by the dac_spi_monitor.

Utility Tasks (tb_util)

[These tasks are not currently used – if we decide to use them, then all $display statements in the design will be replaced accordingly]

This module has no inputs or outputs, but contains a number of tasks for managing the simulation and simulation messages. In order to provide a consistent message display mechanism, all test and testbench messages are managed through the tb_util tasks.

Set the simulation mode: set_sim(verbose, stop_on_error)

This task configures the behavior of some of the other tasks. The **verbose** input controls the verbosity of the output; output messages with higher values (hence lower priority) than **verbose** are not printed. The minimum value for **verbose** is 0, which means that only errors (including fatal errors) and information messages with a priority of 0 are printed; all lower priority messages are discarded. See the **error, warning,** and **info** tasks for more details on message priority.

If **stop_on_error** is set to 1, the simulation will end whenever the first error is encountered. When set to 0, the error message will be printed, but simulation will continue.

Display error message: error(str)

When called, the message in the string **str** will be printed to the console (along with the current time). If the **stop_on_error** flag is set, the simulation will end (by calling the **sim_end** task described below).

Display fatal error message: error_fatal(str)

When called, the message in the string **str** will be printed to the console, and the simulation will end unconditionally (regardless of **stop_on_error**).

Display warning: warning(str)

When called, the message in the string **str** will be printed on the console. Warning messages have a priority of 1, and hence will be printed as long as the verbosity is set to at least 1.

Display informational message: info(level,str)

When called, the message in the string **str** will be printed on the console as long as the current verbosity level is not larger than **level**. As mentioned previously, the level indicates the priority of the message; higher levels mean lower priority. In the Channel FIFO testbench, the meaning of various levels are shown below.

Level	**Meaning**
0	Errors and priority 0 informational messages only. Very terse.
1	Errors, Warnings, and priority 0&1 informational messages only. Quite terse.
2-9	Average verbosity
10-19	Quite verbose
20+	Very verbose. Generally only testbench debugging messages are set with priorities of 20 and greater

Table 1: Verbosity Levels

Print a complete frame: print_frm(level,channel,frm)

When called, a complete frame (all FRM_LEN bytes) of the frame **frm** will be printed on the console as long as the current verbosity level is not larger than **level**. The task should be invoked with **channel** equal to the channel number.

End the simulation: sim_end

This task has no inputs. When invoked, it will print out a message indicating that the simulation is ending and the status of the simulation. If no calls to **error** or **error_fatal** occurred before the invocation of this task, the message “Simulation PASSED” will be printed, otherwise, the message “Simulation FAILED” will be printed. In addition, this task will print the number of calls to the error and warning tasks. Note: errors are counted whenever an invocation of **error** or **error_fatal** occurs, and warnings are counted whenever **warning** is called, regardless of whether the warning message was printed (due to the verbosity level).