Universal Verification Methodology

Assignment Manual

Table of Contents

Overview

In these assignments you will be integrating a verification environment for a YAPP router design. Save all the work as it will be used in the preceding parts of assignments and next assignments as well. You will be integrating multiple UVC components provided to you and will use multichannel sequences and TLM.

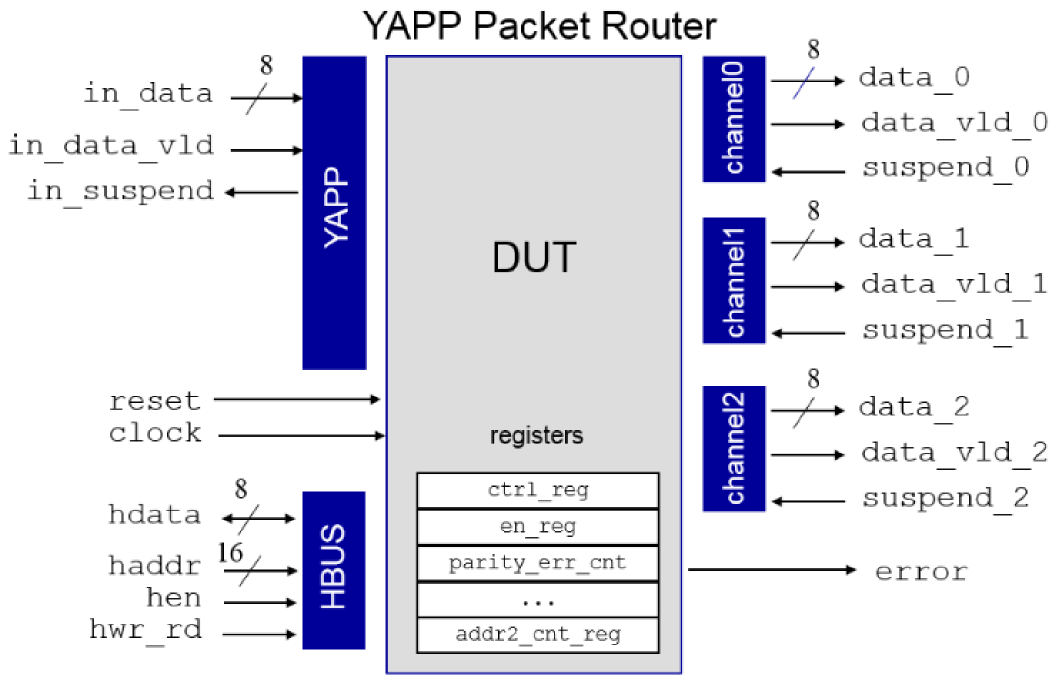
YAPP Router Specification

YAPP Router Description

The YAPP router accepts data packets on a single input port, `in_data`, and routes the packets to one of three output channels: `channel0`, `channel1` or `channel2`. The input and output ports have slightly different signal protocols.

The router also has an HBUS host interface for programming registers that are described in the next section.

High-Level Diagram – YAPP Router (Yet Another Packet Protocol)

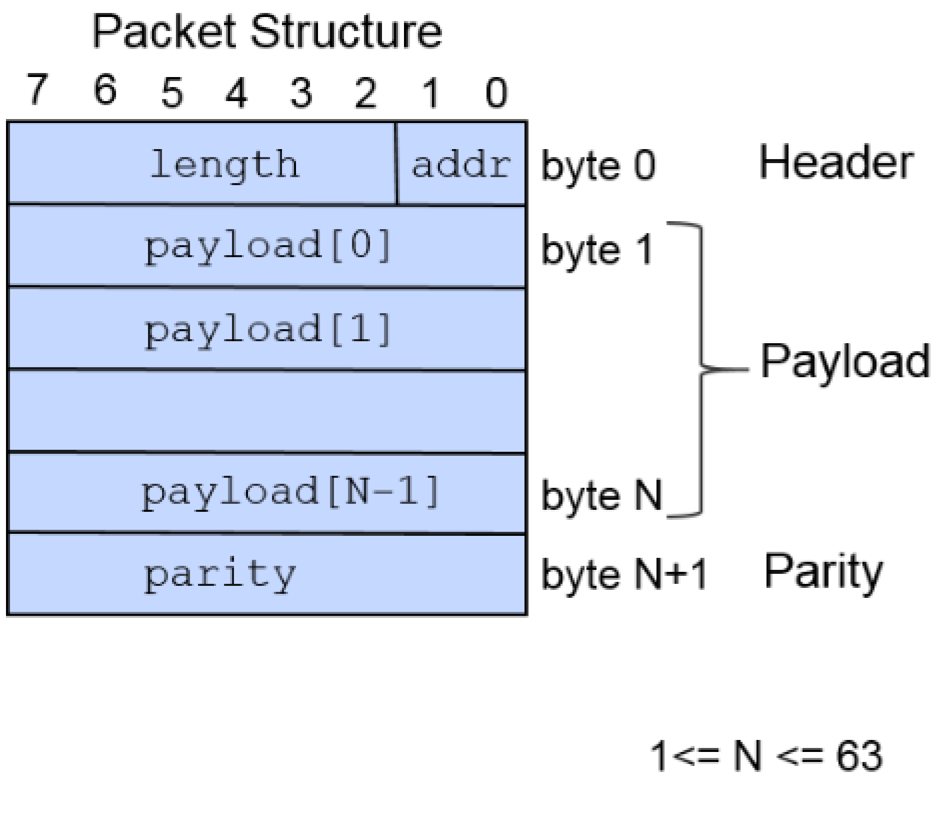


Packet Data Specification

A packet is a sequence of bytes with the first byte containing a header, the next variable set of bytes containing payload, and the last byte containing parity.

The header consists of a 2-bit address field and a 6-bit length field. The address field is used to determine which output channel the packet should be routed to, with the address 3 being illegal. The length field specifies the number of data bytes (payload).

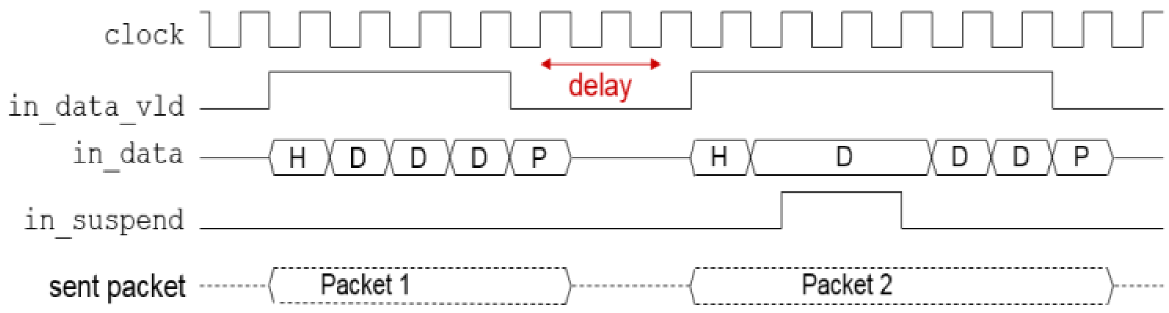
A packet can have a minimum payload size of 1 byte and a maximum size of 63 bytes.
The parity should be a byte of even, bitwise parity, calculated over the header and payload bytes of the packet.



Input Port Protocol

All input signals are active high and are to be driven on the *falling* edge of the clock. The `in_data_vld` signal must be asserted on the same clock when the first byte of a packet (the header byte) is driven onto the `in_data` bus. As the header byte contains the address, this tells the router to which output channel the packet needs to be routed. Each subsequent byte of data needs to be driven on the data bus with each new falling clock.

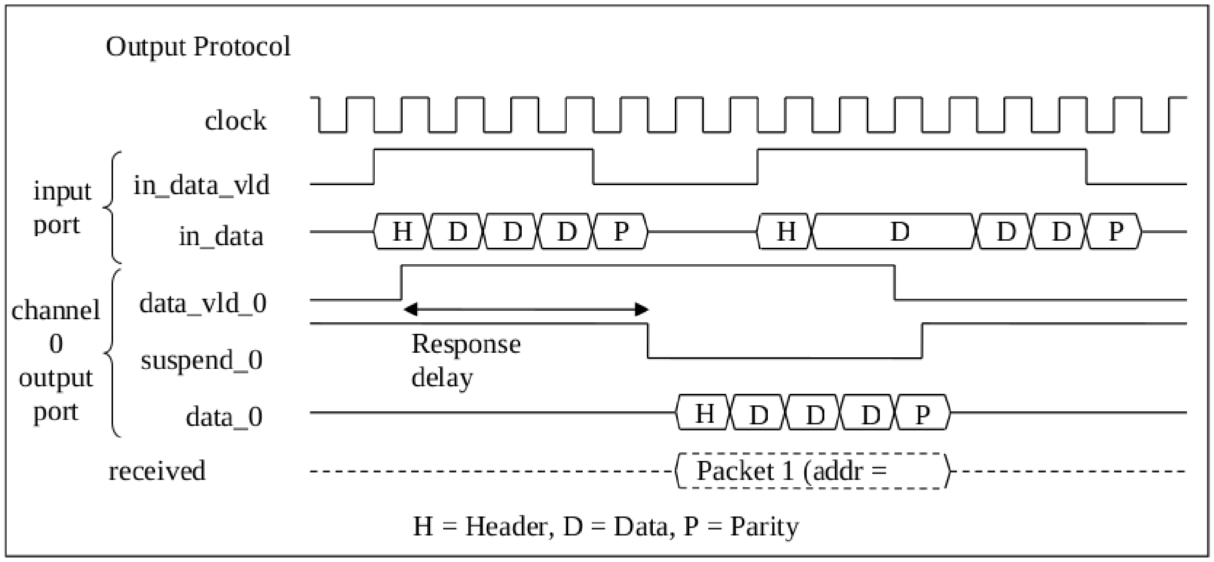
After the last payload byte has been driven, on the next falling clock, the `in_data_vld` signal must be de-asserted, and the packet parity byte needs to be driven. The input data cannot change while `in_suspend` signal is active (indicating FIFO full). The `error` signal asserts when a packet with bad parity is detected, within 1 to 10 cycles.



H = Header, D = Data, P = Parity

Output Port Protocol (Channel Ports)

All output signals are active high and are to be sampled on the *falling* edge of the clock. Each output port is internally buffered by a FIFO of depth 16 and a width of 1 byte. The router asserts the `data_vld_x` signal when valid data appears on the `data_x` output bus. The `suspend_x` input signal must then be de-asserted on the falling clock edge in which data is read from the `data_x` bus. As long as the `suspend_x` signal remains inactive, the `data_x` bus drives a new valid packet byte on each rising clock edge.



Packet Router DUT Registers

The packet router contains internal registers that hold configuration information. These registers are accessed through the host interface port. Register characteristics are as follows:

address	register	reset	field	field name	policy	description
0x1000	ctrl_reg	0x3f	5:0	maxpktsize	RW	Maximum packet length
			7:6		RW	Unused
0x1001	en_reg	0x01	0	router_en	RW	Router enable
			1	parity_err_cnt_en	RW	Parity error count enable
			2	oversized_pkt_cnt_en	RW	Length error count enable
			3	[reserved]	RW	Not implemented
			4	addr0_cnt_en	RW	Address 0 packet count enable
			5	addr1_cnt_en	RW	Address 1 packet count enable
			6	addr2_cnt_en	RW	Address 2 packet count enable
			7	addr3_cnt_en	RW	Address 3 packet count enable
0x1004	parity_err_cnt_reg	0x00	7:0		RO	Packet parity error count
0x1005	oversized_pkt_cnt_reg	0x00	7:0		RO	Packet length error count
0x1006	addr3_cnt_reg	0x00	7:0		RO	Address 3 packet count
0x1009	addr0_cnt_reg	0x00	7:0		RO	Address 0 packet count
0x100a	addr1_cnt_reg	0x00	7:0		RO	Address 1 packet count
0x100b	addr2_cnt_reg	0x00	7:0		RO	Address 2 packet count

If the input packet length is greater than the `maxpktsize` field of the `ctrl_reg` register, then the router drops the entire packet.

The `router_en` field of the `en_reg` register controls the enabling and disabling of the router. A disabled router drops all packets. Enabling or disabling the router during packet transmission will yield to unpredictable behavior.

The router counters are enabled by individual bits in `en_reg`. The router specification says that if these bits are changed while the router is processing a packet, then the router behavior is undefined.

The router counters are defined as follows:

`parity_err_cnt_reg` – incremented when a bad parity packet is received

`oversized_pkt_cnt_reg` – incremented when packet with length greater than `maxpktsize` is received

`addr3_cnt_reg` – incremented when a packet with an illegal address (3) is received

`addr0_cnt_reg` – incremented when a packet with address 0 is received

`addr1_cnt_reg` – incremented when a packet with address 1 is received

`addr2_cnt_reg` – incremented when a packet with address 2 is received

Router Memories

The router contains two memory blocks as follows:

Start Address	Name	Size	Policy	Description
0x1010	yapp_pkt_mem	[0:63]	RO	Stores the bytes of the last packet received by the yapp router.
0x1100	yapp_mem	[0:255]	RW	“Scratch” memory

Host Interface Port Protocol (HBUS)

All input signals are active high and are to be driven on the *falling* edge of the clock. The host port provides synchronous read/write access to program the router.

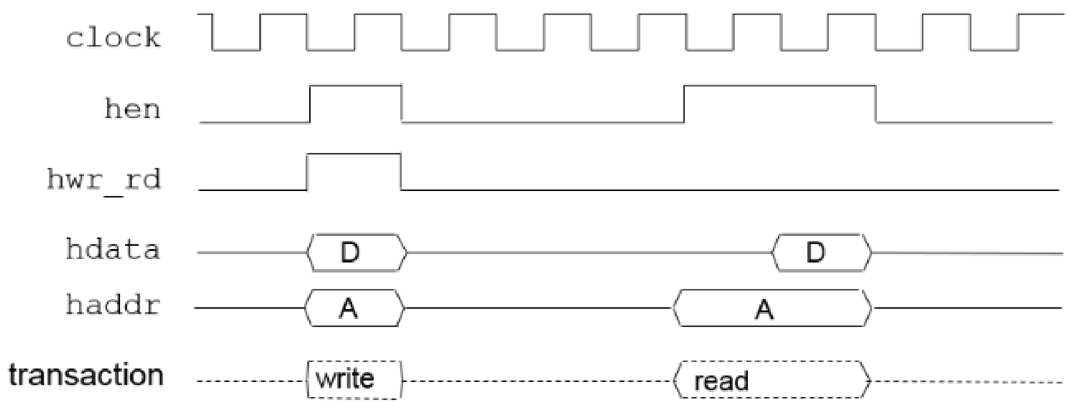
A WRITE operation takes one clock cycle as follows:

- `hwr_rd` and `hen` must be 1. Data on `hdata` is then clocked on the next rising clock edge into the register based on `haddr` decode.
- `hen` is driven to 0 in the next cycle.

A READ operation takes two clock cycles as follows:

- `hwr_rd` must be 0 and `hen` must be 1. In the first clock cycle, `haddr` is sampled and `hdata` is driven by the design under test (DUT) in the second clock cycle.
- `hen` is then driven low after cycle 2 ends. This will cause the DUT to tri-state the `hdata` bus.

The HBUS port provides synchronous read/write access to program the router.



A = Address, D = Data

Note:

File names, component names, and instance names are suggested for many of the labs. You are not required to use these names but if you do not, you may need to edit code provided to match your path and instance names.

These labs do not include step-by-step instructions, and do not tell you exactly what you need to type.

Assignment 1: Integrating Multiple UVCs and writing Multichannel Sequences and System-Level Tests

Objective: To connect and configure the HBUS UVC, Clock and Reset UVC and three output Channel UVCs. And to build and connect multi channel sequences to your testbench

---

Part-1

Understanding Specification and supplied code

Review the YAPP Router specification and understand the protocol. Also review the supplied code for this assignment.

For this lab, you will connect the HBUS, Clock and Reset and Channel UVCs to the router DUT.

All three UVCs are provided. None of the UVCs use configuration objects.

These are the directories we will be using for this and subsequent labs:

```
hbus/sv HBUS UVC files
channel/sv Channel UVC files
clock_reset/sv Clock and Reset UVC files
yapp/sv YAPP input UVC
router_rtl Router DUT
assignment1_integ Your working directory for this lab.
```

Setting Up the Directory Structure

We will be working on the testbench, testclass, and top files. Work in the *assignment1-integ/tb* directory.

Testbench: Channel UVC

1. Update your testbench, `router_tb.sv`, to add the Channel UVCs.

   a. Add three handles of the Channel UVC (`channel_env`) and create the instances in the `build_phase()` method using factory calls.

   b. Use a configuration set method to set the `channel_id` property of each Channel instance. The Channel instance for address 0 should have a `channel_id` of 0, the Channel instance for address 1 should have a `channel_id` of 1and the Channel instance for address 2 should have a `channel_id` of 2. For example,

   ```
   uvm_config_int::set(this, "chan0", "channel_id", 0)
   ```

Testbench: HBUS UVC

2. Update your testbench, `router_tb.sv`, to add the HBUS UVCs.

   a. Add a handle of the HBUS UVC (`hbus_env`) and create the instances in the `build_phase()` method using factory calls.

   b. Use configuration set methods to set the num_masters property of the HBUS UVC to 1, and the num_slaves property to 0. The HBUS UVC has both master and slave agents. For the router testing, we only need the master agent.

Testbench: Clock and Reset UVC

3. Update your testbench, `router_tb.sv`, to add a handle of the Clock and Reset UVC (`clock_and_reset_env`) and create the instance in the `build_phase()` method using a factory call. This UVC requires no configuration.

Hardware Top Module hw_top

4. Update `hw_top.sv` as follows:

   a. Add an interface instantiation for the Clock and Reset. The interface file can be found in the `clock_and_reset/sv` directory. Map the `clock, reset, run_clock` and `clock_period` interface ports to the local signals of the same name.

   b. Connect the `clkgen` module instance to the Clock and Reset interface instance by replacing the `run_clock` and `clock_period` literal port mappings with the local signals of the same name.

   c. Add interface instantiations for the HBUS and all three Channels. The interface files can be found in the sv directory of each UVC directory. Map the ports of the interfaces to the local `clock` and `reset` signals of the same name.

   d. As the `reset` will now be generated by the Clock and Reset UVC, delete the `initial` block which generates the reset waveform.

   e. Update the port mapping of the router instantiation to connect the Channel and HBUS interface signals.

      Warning – The HBUS interface contains a bi-directional signal `hdata`. When you connect the HBUS interface signals to the router DUT, you must use the wire net `hdata_w`, in the port mapping, not the `logic` variable `hdata`.

UVM Top Module tb_top

5. Update `tb_top.sv` as follows:

   a. Add imports for the Channel, Clock and Reset and HBUS UVC package files. The packages can be found in the sv directory of each UVC.

b. Set the HBUS, Clock and Reset and Channel UVC virtual interfaces to the correct interface. (*Hint*: The UVC header files contain `typedefs` for each interface.)

   Use wildcards in the pathname to update all UVC components with a single statement.

   Use an absolute hierarchical pathname for the value to select the correct interface instance from the `hw_top` module.

Running Base Test

6. Run a simulation with `base_test` only. Check the topology report carefully to make sure all of your UVCs are instantiated and configured correctly. Copy the topology report into a new file for future reference.

Test Library

7. Add a new test class, simple_test, in `router_test_lib.sv` as follows (copy from existing tests). Sequencer pathnames can be read from the topology report.
   a. Set the YAPP UVC to create short YAPP packets with a `set_type_override`.
   b. Set the default sequence of the YAPP UVC to `yapp_012_seq`.
   c. Set the default sequence of each Channel UVC to `channel_rx_resp_seq`.

      Hint: you can set all three Channel UVCs with a single statement.
   d. Set the default sequence of the Clock and Reset UVC to `clk10_rst5_seq`.
   e. Do not define a default sequence for the HBUS UVC.
   f. (Optional) Now might be a good time to clean up the test library and remove the older tests. Delete or comment out all the other test classes besides `base_test` and `simple_test`.

Running Simple Test

8. Run a simulation on EDA Playground using `simple_test` and verify as follows:
   a. Use the waveform viewer to confirm that packets are passed correctly through the router and collected at the right channel.

Further Integration Testing (Optional)

9. Write a new YAPP sequence in the `yapp/sv/yapp_tx_seqs.sv` file to generate packets for all four channels (including the illegal address 3). The packets should have incrementing payload sizes from 1 to 22 and a parity distribution of 20% bad parity (88 packets in total).

Hint: You could create packets using nested loops for address and payload.

10. Create a new test, `test_uvc_integration`, in the `router_test_lib.sv` file to perform the following:

    a. Set the `run_phase` default sequence of the YAPP UVC to the sequence created above.

    b. Set the `run_phase` default sequence of the HBUS UVC to set up the router with register field `maxpktsize` = 20 and enable the router (register field `router_en` = 1).

    Hint: There is a sequence defined for this in the HBUS master sequences `hbus_master_seqs.sv`.

    Hint: The hierarchical path name for the HBUS configuration setting can be read from the topology report.

11. Run a simulation and check the results to see that the three channels are properly addressed, that there is an error signal when parity is wrong, and that packets are dropped if bigger than `maxpktsize` or have illegal addresses.

Part-2

For the next part of the lab, you will build and connect a multichannel sequencer for the router, and create multichannel sequences to coordinate the activity of the three router UVCs.

Copy the files from *assignment1-integ/tb* into *part2_mcseq/tb*. Work in the *part2_mcseq/tb* directory.

1. Complete the multichannel sequencer component `router_mcsequencer.sv`.

    a. Add a component macro and constructor.

    b. Add the references for the HBUS and YAPP UVC sequencer classes.

    The Channel UVCs continuously execute a single response sequence, so they do not need to be controlled by the multichannel sequencer.

    The Clock and Reset UVC could be controlled by the multichannel sequencer if, for example, we wanted to initiate reset during packet transmission. However, for simplicity we'll leave Clock and Reset out of the multichannel sequencer.

2. Create a multichannel sequence library file, `router_mcseqs_lib.sv` and define a single multichannel sequence, `router_simple_mcseq`, as follows:

    a. Add an object macro and constructor.

    b. Add a `` `uvm_declare_p_sequencer `` macro to access the multichannel sequencer references.

c. Using the sequences defined in the YAPP and HBUS UVC sequence libraries, create a multichannel sequence to:

   - Raise an objection on `starting_phase`.
   - Set the router to accept small packets (payload length < 21) and enable it.
   - Read the router `MAXPKTSIZE` register to make sure it has been correctly set.
   - Send six consecutive YAPP packets to addresses 0, 1, 2 using `yapp_012_seq`.
   - Set the router to accept large packets (payload length < 64).
   - Read the router `MAXPKTSIZE` register to make sure it has been correctly set.
   - Send a random sequence of six YAPP packets.
   - Drop the objection on `starting_phase`.

There are pre-defined sequences in the UVC libraries for all the above operations.

3. Modify the `router_tb.sv` testbench to instantiate, build, and connect the multichannel sequencer.

   Hint: Examine a topology report to find the reference for the HBUS sequencer. Remember the connections for the multichannel sequencer references are hierarchical pathnames, not configuration instance name strings, therefore you cannot use wildcard characters in the connection.

4. Create a new test in `router_test_lib.sv` to achieve the following:

   a. Set a type override for short packets only.

   b. Set the default sequence of all output channel sequencers to `channel_rx_resp_seq` (copy from a previous test).

   c. Set the default sequence of the Clock and Reset sequencer to `clk10_rst5_seq` (copy from a previous test).

   d. Set the default sequence of the multichannel sequencer to the `router_simple_mcseq` sequence declared above.

   e. Do **not** set a default sequence for the YAPP or HBUS sequencer. Control is now solely from the multichannel sequencer.

5. Add `include` statements to your top module to reference the new files.

   Make sure the includes are in the correct order, for example the multichannel sequencer must be included before the testbench file, as the testbench creates an instance of the multichannel sequencer.

6. Run a test and check your results. If you open the simulator log file (Download files after run) in an editor, you should be able to track packets through the router and see the HBUS read and write transactions.

   If necessary, you can insert extra delays between the YAPP and HBUS sequences in the multichannel sequence to clearly separate transactions on the different interfaces.