Lab 3 – V1.1

Modify your LAB2 testbench for Gullfaxi to make it compliant to what is described in this document.

GOP protocol assertions

To test that Gullfaxi's three output ports respect the GOP protocol rules, the testbench program shall contain the following set of assertions. Write all of them in the testbench program, as sensitive to the negative clock edge. Add the $assertkill command before all $finish commands, so that assertions that are being evaluated are killed at the end of the simulation. All assertions shall be repeated for all ports, i.e. there shall be three versions for each assertion, with prefixes GOP0, GOP1 and GOP2, with each assertion checking the signals on one GOP port. Note that some of the assertions are already given in the example testbench. The assertions check the values of all GOP signals in all phases of the protocol: if they are all satisfied, then it is guaranteed that all GOP rules are satisfied.

The following two assertions establish the condition in which O_start is one: O_start is one if and only if there was a granted request two cycles earlier.

- GOP{0,1,2}_start1: The assertion shall check that two cycles after a granted request O_start is one. Remember that a granted request takes place if O_req was one and was suddenly granted, if O_grant was one and O_req was issued or if both signals became one at the same time.

- GOP{0,1,2}_start0: The assertion shall check that, if there is not a granted request, two cycles later O_start must be zero.

The following assertion establishes that O_end must be zero from one cycle after O_end is one to the next time in which O_start becomes one. We will later add another assertion to check the behavior of O_end between the times in which O_start becomes one and when O_end becomes one.

- GOP{0,1,2}_endOutOfTran: one cycle after O_end is one, O_start will go to zero, remain at zero for an unknown number of cycles and then rise again. The assertion shall check that O_end goes to zero one cycle after O_end was one and then remain at zero until the cycle in which O_start becomes one.

The following assertion establish the behavior of the O_req signal: it must go to zero when O_start is issued, it cannot go to zero at any other time; after O_start is one, O_req should remain at zero until at least one cycle after O_end is one.

- GOP{0,1,2}_reqNoFall: The assertion shall check that O_req cannot fall if O_start is not one.

- GOP{0,1,2}_reqFall: when O_start is one, O_end will be zero and remain at zero for an unknown number of cycles until rising. The assertion shall check that O_req falls in the same cycle in which O_start is one and that it remains zero until one cycle after O_end is one.

The following assertions check the behavior of the length signal: O_length can only change when O_req goes to one or one cycle after O_end is one; one cycle after O_end is one length must go to zero.

- GOP{0,1,2}_lengthToggle: The assertion shall check that O_length can change only when O_req goes to one or one cycle after O_end is one.

- GOP{0,1,2}_length0: one cycle after O_end is one, O_req will be zero for a certain number of cycles and then rise. The assertion shall check that O_length goes to zero one cycle after O_end is one and then remains zero until the cycle before O_req becomes one.

The following assertion checks the behavior of O_end after O_start is issued, and see if it is compatible with the value of O_length: if O_length is one, then O_end must be one at the same time in which O_start is issued, else O_end is zero and stays at zero for a number of cycles equal to O_length-1, then becomes one. You need to use a sequence with a local counter variable that is incremented on every cycle in which O_end is zero. When O_end becomes one, you must check whether the value of the counter is correct compared to the value on O_length.

- GOP{0,1,2}_endDuringTran: The assertion shall check that, when O_start is one, then either one of two following possibilities take place: (A) O_length is 1 and O_end is also 1 (packet of length 1) or (B) O_end is zero, stays at zero for O_length-1 cycles, then becomes 1.

The GOP protocol does not put any restriction on when and how O_data and O_grant should change, so you do not need to write any assertion to check their behavior. Since the assertions above check all transitions of the GOP signals in every phase, if they are all satisfied then it is guaranteed that all the GOP protocol rules are satisfied.

GIP protocol assertions

Since the company has a great trust in your verification abilities, it has decided to use you to develop assertions for the GIP protocol as well. These will be used for the verification of Andvari, Fafnir, Nagifar and Sigurd, the blocks that send their packets to Gullfaxi in the Valhalla chip (the verifiers of all four blocks have all recently been fired for lack of abilities). For the moment, you can use them to check the GIP signals generated by your testbench (although the company is sure that you won't find any mistake there given your great coding abilities).

From a certain point of view, the GIP protocol is more tricky than the GOP protocol, because it does not have a specific signal to determine when a packet transmission starts: in GOP, we knew that the first word of a packet was transmitted when O_start was one. In GIP, the header is sent when I_valid becomes one, but not every time I_valid rises corresponds to the sending of the header.

For a first thing, you need to develop a sequence startOfPacket that matches every time in which the header word of a new packet is sent.

Define a sequence startOfPacket which matches at all times in which a header word is sent. As a hint, think that a packet transmission starts when (A) reset became one, I_valid remained low for a certain amount of cycles, then it was raised, or (B) I_end was one, then I_valid remained low for at least two cycles, then it was raised. Don't forget to take care of the special case in which a sending is started in the same cycle in which reset rises.

To test if the sequence is correct, make an assertion of the type

assert (@(negedge clk) startOfPacket |-> 0) else $info("sending header");

Check on your waveforms when the assertion fails: if your sequence is correct, it will fail every time that a header word is sent and will never fail at any other times.

We do not need an endOfPacket sequence because we know the packet ends when I_end is one.

Now that you have the startOfPacket sequence, you can write the GIP assertions.

The following assertion checks that, as specified by the rules of the GIP protocol, no wait cycle can be inserted one cycle after the header byte is sent out.

- GIP_noWaitCycleAfterHeader: The assertion shall check that one cycle after the start of a packet, I_valid must be one

The following assertion checks that no transmission of packet is started if I_ready was zero in the past cycle.

- GIP_noStartIfNotReady: The assertion shall check that the start of a packet can only take place if I_ready was one in the past cycle.

The following two assertions make checks on the length and port fields in the packet headers.

- GIP_headerPortCheck: The assertion shall check that when the header byte is sent I_data[1:0] cannot be 3.

- GIP_headerLengthCheck: The assertion shall check that when the header byte is sent I_data[7:2] must be between 1 and 12.

The following assertion checks that the minimal time of two cycles between the end of the sending of a packet and the beginning of the sending of the next is respected.

- GIP_intraPacketDelay: The assertion shall check that there are at least two cycles with I_valid low following the end of a packet.

The following assertion checks the behavior of the I_end signal outside of a packet transmission. Another assertion, to check the behavior of the I_end signal during a packet transmission, is left as an optional task: if introduced, then the set of assertions is complete, if all are satisfied then all GIP protocol rules are respected.

- GIP_endOutOfTran: One cycle after the end of a packet, I_valid will go to zero and remain at zero for a certain amount of cycles, then it will rise. The assertion shall check that I_end remains zero for all the time during which I_valid is zero.

Optional tasks:

(1) Add to the set of assertions the last assertion needed to complete the checks on the GIP protocol:

- GIP_endDuringTran: The assertion shall check that, after the start of a packet, I_end shall remain at zero until all payload words have been transmitted, and go to one when the last word is sent. Hints: you can use a sequence with two internal variables, one for the expected length and the other as a counter; count only the words with I_valid=1; don't forget to take care of the special case with length=2.

(2) Modify the CRSG constraints for the generation of the A, F, N and S packets. You have to make it so that the first payload word of a packet is always 0 if the packet is an A packet, 1 if the packet is an F packet, 2 if it is an N packet and 3 if it is an S packet. Write four assertions that are triggered as soon as an A, F, N or S packet's first payload word is being sent on the four GIP ports. The assertions shall check that the values of length and outPort given in the header of the packet correspond to the constraints of the given type of packet (the A assertion shall for example fail if an A packet of odd length is being transmitted). Test these assertions on the crazy Fafnir test from the previous lab.

(3) Keep the CRSG modifications you introduced in task 2. Write an assertion that, when an F packet is received on a GIP port, checks whether or not the last payload word is the XOR of all previous payload words.