Lab 2 – V1.2

Modify your LAB1 testbench for Gullfaxi to make it compliant to what is described in this document.

Background: Gullfaxi Drivers

We have recently received input from the system designers of the Valhalla chip, the upcoming super-powerful chip in which Gullfaxi will be used, and we now know better how Gullfaxi will be used. It is now time to upgrade Gullfaxi's testbench so that the packets injected by the testbench correspond better to what Gullfaxi will have to deal with in the real use cases that it will encounter once implemented in the Valhalla chip.

On its GIP port, Gullfaxi will receive packets coming from four different blocks: Andvari, Fafnir, Nagifar and Sigurd. The packets generated by the four blocks are called A packets (for Andvari), F packets (for Fafnir), N packets (for Nagifar) and S packets (for Sigurd).

High-level simulations of the Valhalla chip show that Gullfaxi receives on average:

- 12.5% A packets
- 37.5% F packets
- 25% N packets
- 25% S packets

"A" packets can only have payload length 2, 4, 6, 8 or 10 (with uniform probability distribution). In 90% of the cases, they are destined for port 1, in 10% of the cases they are destined for port 2. A packets are never destined for port 0.

"F" packets can only have payload length 3, 5, 7, 9 or 11 (with uniform probability distribution). In 90% of the cases, they are destined for port 1, in 10% of the cases they are destined for port 2. F packets are never destined for port 0. The last word in an F-packet is an error-correction word which is obtained by XORing bit-by-bit all the other payload words. For example, if an F packet of length 3 has the first two words equal to 8'hA8 and 8'h2F, the last (third) payload word is obtained by doing the bit-by-bit XOR between 8'hA8 and 8'h2F ('h87).

"N" packets can be destined to all three output ports of Gullfaxi (with uniform probability distribution). The length of an N packet is between 5 and 12 payload bytes. All packet lengths have the same probability to occur except the value 5, which occurs twice as often as the other values.

"S" packets can be destined only to output ports 1 or 2 of Gullfaxi (with uniform probability distribution). If an S packet is destined for port 1, then its payload length is between 1 and 8; if it is destined for port 2, then its payload length is between 1 and 10.

Random Constrained Generation of the Packets

Modify your LAB 1 testbench, taking inspiration from the CRSG/coverage example. At time 0, a process shall generate a queue of classes. Every class shall have the following rand members:

- pktType, an enumerated data type that can take the values A, F, N, S.
- length, a 6-bit array
- outPort, a 2-bit array
- payload, a dynamic vector of bytes

Every time the class is randomized, it shall generate a legal packet of type A, F, N or S based on the constraints given above. Randomize the variables in the following order: pktType, outPort, length, payload. Payload bytes can be generated randomly without any constraint. In case of an F packet, the payload shall first be generated randomly; then, after the random generation, the last payload word shall be overwritten in the post_randomize() function of the class, i.e. the post_randomize() function shall look at all the randomly-generated bytes except the last and overwrite the randomly-generated last word with the calculated error-correction code. The class shall contain "dist", "ordering" and other conditional and non-conditional constraints to ensure that the A, F, N and S packets are generated with the properties and probabilities given above (the ratio between A, F, N and S packets shall for example be the one given above). In case the packet is an A or F packet, then a conditional dist construct shall make sure that the probabilities in the selection of the output ports are 90% and 10%.

At time 0, in a loop, a class object shall be randomized to generate legal packets of type A, F, N and S based on all the constraints given above and shall be stored into two queues of classes, one for sending and the other one for receiving.

Injection and checking of the packets

The GIP driver process used for LAB1 shall be modified to pop class objects from the sending queue and drive them to the DUT on the GIP port. The payload words shall be taken from the class member payload. The GOP consumer processes shall be modified so that they pop classes from the receiving queue and check if the port and the length of the GOP transactions are as expected. All other features of the LAB1 testbench, such as insertion of wait cycles and parameters, shall be retained. There is no need to do any check on the payload in the GOP controller, it is enough to check that the length and the port are correct.

Functional Coverage on Packets

Add to the testbench program three variables:

- pktType, an enumerated data type that can take the values A, F, N, S.
- length, a 6-bit array
- outPort, a 2-bit array

Add to the testbench program the definition of a covergroup cg with no triggering event, with the following coverpoints:

- pktType
- length

- outPort
- cross between pktType and outPort
- cross between pktType and length
- cross between length and outPort

Define as ignore bins all the values that you know will never occur due to the constraints (for example, you know that length==15 will never occur; outPort=3 will also never occur; pktType=A and length=1 in the pktTypeXlength cross will never occur; length=3 and outPort=0 in the lengthXoutPort cross will also never occur, ...)
Make another coverpoint on pktType that you will use for transition bins, to check if we tested Gullfaxi for some special input sequences that, according to what the system designers say, will play a very important role in the upcoming Valhalla chip. On this coverpoint, make bins to record how many times we had:

- four A packets in a row
- four F packets in a row
- eight A packets in a row
- eight F packets in a row
- an A packet followed by 3 F packets followed by an A packet
- an A packet followed by 6 F packets followed by an A packet
- an A packet followed by 6 packets of any type followed by an A packet
- any number between 4 and 8 N packets followed by an A packet
- any A packet followed by a F packet and any F packet followed by an A packet (the bin shall increment for both an A => F transition and for an F => A transition)
- any A or F packet followed by three consecutive N packets followed by an S packet
- any sequence of 10 consecutive packets which are all of the type A or F only (it shall for example increment if there is a sequence A => F => F => A => A)

In the process that generates the packet queues at time 0, after each randomization, store the values of pktType, length and outPort to the three variables pktType, length and outPort, and then trigger a sampling of the covergroup using the sample() built-in function.

Functional Coverage on DUT Signals

Generate a second covergroup with the negative clock edge as triggering event. Add two coverpoints on I_ready and a coverpoint on the O_req port, three coverpoints on O0_start, O1_start and O2_start, three coverpoints on O0_end, O1_end and O2_end, a cross between O0_start and O0_end, a cross between O1_start and O1_end, a cross between O2_start and O2_end.

- define as illegal bin all bins in which O_req is one in more than one output port at the same time: if this happens then Gullfaxi must have some serious bug!
- leave the first coverpoint on I_ready with automatic bins. We can use this coverpoint to check whether Gullfaxi's buffer ever became near-to-full.
- On the second coverpoint on I_ready, add the following transition bins:
    - I_ready goes to 0, stays at 0 for a single cycle, then goes back to 1 (1-0-1)
    - I_ready goes to 0, stays at 0 for two cycles, then goes back to 1 (1-0-0-1)
    - I_ready goes to 0, stays at 0 for three cycle, then goes back to 1 (1-0-0-0-1)

  - I_ready goes to 0, stays at 0 for a number of cycles between 4 and 12, then goes back to 1.
  - I_ready stays at 0 for 100 consecutive cycles
  - I_ready stays at 1 for 100 consecutive cycles

Tests

One of the main advantages of CRSG is that once a system of constraints is in place, it is easy to make new tests (testbench variations) by adding constraints and/or deactivating existing constraints.

Without adding any constraint to the class, using only the ”randomize() with” statement and the selective turning off of constraints in the process that generates the class queues, make the following tests, each time saving coverage data in a new subdirectory:

- generic test with no additional constraints compared to those that were already given
- test in which only packets of type A, F and N are generated
- test in which all packets are bound for port 0
- test in which the distribution of the packets is: 25% type A, 50% type F, 10% type N, 15% type S (here you need to deactivate the dist constraint in the class)
- test in which all packets are bound for ports 0 or 2 and have length higher than 5
- test in which all N packets are bound for port 0 (other packets can go anywhere).
- test in which all packets have length smaller than 4 and are all bound for port 0… Interesting, huh?
- “crazy Fafnir” test: Our preliminary tests have shown that Fafnir has a serious RTL bug and sometimes enters a highly unstable state called “crazy Fafnir state”. When Fafnir is in the crazy Fafnir state it sends out packets at its maximum throughput, bound for a random port chosen randomly among the three Gullfaxi output ports. Unfortunately, we need to check that Gullfaxi can deal with the crazy Fafnir state, because the bug will most likely not be solved before the final delivery of the Valhalla chip (Fafnir’s designer has been recently fired for lack of abilities and nobody has yet taken his place). In this test the testbench shall change the ratio of packets generated to: 5% for type A, 75% for type F, 10% for type N and 10% for type S to simulate the erratic behavior of Fafnir. Leave the length constraints and payload constraints on F packets but don’t forget to make it so that F packets are sent to any random port.

Hopefully now you understand how easy it is to generate tests using CRSG.

Optional tasks

(1) Without modifying the class, only by adding constraints to the testbench program during packet randomization and changing the parameters of the testbench (such as time between packet sendings, time before grant is given, etc.), write a series of tests whose functional coverage, when merged, is 100%.

(2) Without modifying the class, modifying only the packet-generating initial block, use “randomize with”, to ensure that there are never more than two consecutive packets bound for port 0 (after two packets bound for port 0, the next packet shall be bound for either port 1 or 2).

(3) Make a test in which Fafnir starts in a normal state, then enters the crazy Fafnir state after a random time between 5000 and 10000 cycles and stays in the crazy Fafnir state until the end of the simulation. Note: for this task, it is not possible to generate all packets at time zero, because the way that packets should be generated depends on information only known at run-time. So, you should modify the structure of the testbench so that packets are generated on-the-fly. The recommended structure is the following: use an initial block to toggle a crazy Fafnir flag after a random number of cycles. Every time the GIP driver process can send a new packet, it shall generate it randomly through CRSG and push the new packet in the packet receiving queue (there is no need of a packet sending queue). When generating a new packet, the GIP driver process shall look at whether Fafnir is crazy or normal and change the randomization constraints accordingly.