Generate Eth packets, drive to DUT according to interface protocol and capture the driven packets in a monitor to display

Packet Generator

1. Generate Ethernet packets with following layers
    a. L2
        i. Eth header - DA: 48b, SA: 48b
        ii. VLAN may or may not be present: 32b
        iii. Eth type: 16b
    b. L3: IP header should be one of the following types
        i. IPv4: 20B
        ii. IPv6: 40B
    c. L4: Should be one of the following types
        i. TCP: 20B
        ii. UDP: 8B
    d. TCP/UDP ports (both source and destination) should be equally distributed among the following protocols
        i. FTP data (20)
        ii. HTTP (80)
        iii. HTTPS (443)
        iv. PPTP (1723)
        v. Other
    e. Payload
2. Packet size 64B up to 4KB

Driver and Monitor

Driver gets the packet from generator and drives to DUT according to interface protocol. Monitor captures the packets from DUT interface and display them.

Interface Protocol

The protocol is a simple valid-ready type interface where valid qualifies data and other control signals.

1. TB drives valid, data, sop and eop. DUT drives ready
2. Data transfer happens at every cycle where valid and ready are high
3. Data transfer stops if either valid or ready is low.
4. DUT is free to de-assert ready at any time during the packet transfer and keep ready low for any number of cycles
5. TB can de-assert valid at any time during the packet transfer and keep valid low for any number of cycles.

6. Assertion of valid should not depend on ready to be high
7. Packet is bounded by sop and eop
8. ‘bc’ indicates number of valid bytes at the last flit of the packet. All other flits must be complete flits without any invalid byte.

Simple packet with Byte Count

Interface signals

Signal	Direction (from TB)	Width (bit)	Description
valid	Out	1	If high, indicates sop, data and eop are valid
ready	In	1	If high, indicates receiver is ready to accept data
sop	Out	1	If set, indicates start fleet of the packet
data	Out	1024	Packet data
eop	Out	1	If set, indicates end fleet of the packet
bc	Out	7	Indicates valid bytes count when eop is high. Must be 0 otherwise. • 0: Indicates 128B • 1-127: Valid byte count

Implementation note

1. Use SV constraint block to generate different packet types
2. Implement packet functions to assemble header and payload bytes into an array
   a. Packets represented as byte arrays is useful for the driver
3. Use SV interface between TB and DUT
4. Communication between Generator and Driver should be formal. No method call between components.

Implementation Steps

1. Packet Generation Method

1. **Ethernet Header object generation**
   - destination MAC addresses generated using randomization.

- source MAC addresses generated using randomization.
- EtherType randomized with constraint value inside {**0x0800**, **0x86DD, 0x8100**}
  - if generated EtherType is **0x8100**, indicates VLAN Tag is present and another 16 bits random value added for the Tag Control Information (TCI).
  - otherwise, ignore.
- EType: 16 bits; EType randomized with constraint value inside {**0x0800**, **0x86DD**}
- post_randomization() generates ipv4 or ipv6 object based on the EType
- IP4 Header: IPv4 Object if EType == **0x0800**
- IP6 Header: IPv6 Object if EType == **0x86DD**
- l4_header: post_randomization() generates l4_header based on the ipv4 protocol or ipv6 next_header.
- if ipv4 protocol == 8'h06, generate TCP header object
- if ipv4 protocol == 8'h11, generate UDP header object
- if ipv6 next_header == 8'h06, generate TCP header object
- if ipv6 next_header == 8'h16, generate UDP header object

2. **IPv4 Header object generation**
   - IPv4 header is an object defined with the following fields:
     - version : 4 bits; assigned default value 4
     - ip header length : 4 bits; generated using randomization
     - type of service : 8 bits; generated using randomization
     - total_length : 16 bits; generated using randomization
     - ip identification number : 16 bits; generated using randomization
     - flags : 3 bits; generated using randomization
     - fragment_offset : 13 bits; generated using randomization
     - ttl : 8 bits; generated using randomization
     - protocol : 8 bits; **generated using randomization with constraint inside {0x06, 0x11}**
     - header_checksum : 16 bits; generated using randomization
     - source_ip : 32 bits; generated using randomization
     - dest_ip : 32 bits; generated using randomization
3. **IPv6 Header object generation**
   - IPv6 header is an object with the following fields:
     - version : 4 bits; assigned default value 6
     - traffic_class : 8 bits; generated using randomization
     - flow_label : 20 bits; generated using randomization
     - payload_length : 16 bits; generated using randomization
     - next_header : 8 bits; **generated using randomization with constraint inside {0x06, 0x11}**
     - hop_limit : 8 bits; generated using randomization

- source_ip : 128 bits; generated using randomization
- destination_ip : 128 bit; generated using randomization

4. **TCP header (l4_header) generation:**
   - TCP header is implemented as a class object the following fields:
     - source_port; 16 bits // Source Port equally distributed among the following protocols
     - dest_port; 16 bits // Destination Port equally distributed among the following protocols
     - seq_num; 32 bits // Sequence Number generated through randomization
     - ack_num; 32 bits // Acknowledgment Number generated through randomization
     - header_length; 4 bits generated through randomization
     - reserved; 4 bit // Reserved bits generated through randomization
     - flags; 8 bits // Control Flags generated through randomization
     - window_size; 16 bits // Window Size generated through randomization
     - checksum; 16 bits // Checksum generated through randomization
     - urgent_pointer; 16 bits // Urgent Pointer generated through randomization
5. **UDP header (l4_header) generation:**
   - TCP header is implemented as a class object the following fields:
     - source_port; 16 bits // Source Port equally distributed among the following protocols
     - dest_port; 16 bits // Destination Port equally distributed among the following protocols
     - length; 16 bits // Length of UDP Header and Data generated through randomization
     - checksum; 16 bits // Checksum generated through randomization
6. **Port Distribution Logic**
   - **Weighting**:
     - Assigned a weight of 1/5 to each of the specified ports (20, 80, 443, 1723).
     - Assigned a weight of 1/5 to “Other,” which can be any valid 16-bit port number.
   - **Random generation of source_port and dest_port** :
     - implemented as a distribution constraint of ports. dist { 20 :=1, 80 := 1, 443 := 1, 1723 := 1, default :/65532};
7. **Payloads generation:**
   a. Based on the packet size (64 B - 4 KB) a variable size random bytes will be added as payload. the size should be constrained.
8. **Assembling the ethernet header, ipv4 header/ipv6 header, l4 header and payloads into packets:**
   - pack() function has the following duties:
     - Combining the generated Ethernet header object, IPv4 or IPv6 header object and TCP/UDP objects
     - Adding any additional payload data based on the packet size.

2. Generator Implementation

`Generator - Generates (create and randomize) the ethernet packets and send to driver through mailbox.`

Generator class has the following members:

- Packet object,
- mailbox handle,
- `put the packet into the mailbox`

Mailbox.put() method is called to place the packet in the mailbox.

3. Driver Implementation

- Driver implemented as a class has the following function:
  - Driver gets the packet from the mailbox
  - breaks the packet into fleets,
  - drive fleets to the signals in the interface modport
  - Virtual interface is declared inside driver class

4. Mailbox Implementation

- Mailboxes are used as a communication mechanism to transfer data between a packet generator to a driver.
  - Mailbox object is instantiated in the top module.
  - mailbox handle is passed to the generator and driver.

5. Monitor Implementation

- Monitor implemented as a class. has the following function:
  - Monitor receive signals from the DUT interface
  - interprets the signal,
  - converts signal into byte array
  - convert the byte array into packet object: unpack() function
  - display the packet object.
- Virtual interface and packet object handle is declared inside Monitor class

6. Interface Implementation

- Interface defines the signals (valid, ready, sop, data, eop, bc, clk) and modports
  - modport monitor
  - modport dut
  - modport driver

6. DUT Implementation

will be provided

7. TB Implementation

- At the top module packet generator, driver, monitor, mailbox, interface objects are instantiated.
- Clock is created for sampling
- bounded mailbox of depth 1 is implemented
- Three parallel processes are forked:
  a. generator to generate packets and put it into the mailbox
  b. driver to get a packet from mailbox to drive it to the modport of the interface
  c. monitor to captures the packets from DUT interface and display them
- vpd file generated to see the waveform.