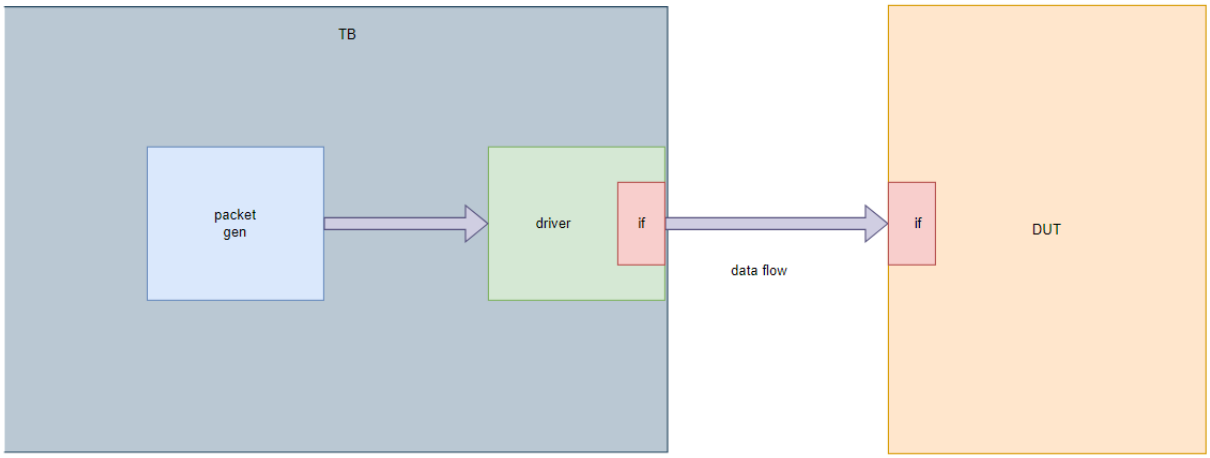
Generate Eth packets and drive to DUT according to interface protocol

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
2. Packet size 64B up to 4KB

Driver

Driver gets the packet from generator and drives to DUT according to interface protocol

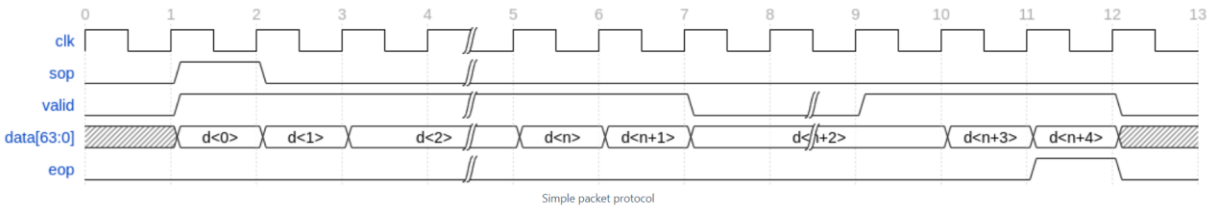


Simple packet driver

Interface Protocol

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

1. Data transfer happens at every cycle, valid is high
2. Data transfer stops if valid is low. Valid can be low for any number of times during packet transfer and may remain low for any number of cycles
3. Packet is bounded by SOP and EOP



Simple packet protocol

Interface signals

Signal	Width (bit)	Description
sop	1	If set, indicates start of the packet
valid	1	If high, indicates all other signals in the interface is valid. All signals should hold the
data	64	Packet data
eop	1	If set, indicates end of the packet

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 to dive the packets

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, l4_header = 20 Bytes TCP header generated as random values.
- if ipv4 protocol == 8'h11, l4_header = 8 Bytes UDP header generated as random values.
- if ipv6 next_header == 8'h06, l4_header = 20 Bytes TCP header generated as random values.
- if ipv6 next_header == 8'h16, l4_header = 8 Bytes UDP header generated as random values.

2. **IPv4 Header object generation**
   - IPv4 header is an object of 20 bytes 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 of 40 bytes 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. **Transport Layer header (l4_header) generation:**

It is implemented as a byte array. The protocol field in the **IPv4 Header** or the next_header field in the **IPv6 Header** specifies whether it is TCP or UDP:

if protocol field in the **IPv4 Header** is **0x06**, 20 bytes are appended as TCP header with random values.

if protocol field in the **IPv4 Header** is **0x11**, 8 bytes are appended as UDP header with random values.

similarly,

if next_header field in the **IPv6 Header** is **0x06**, 20 bytes are appended as TCP header with random values.

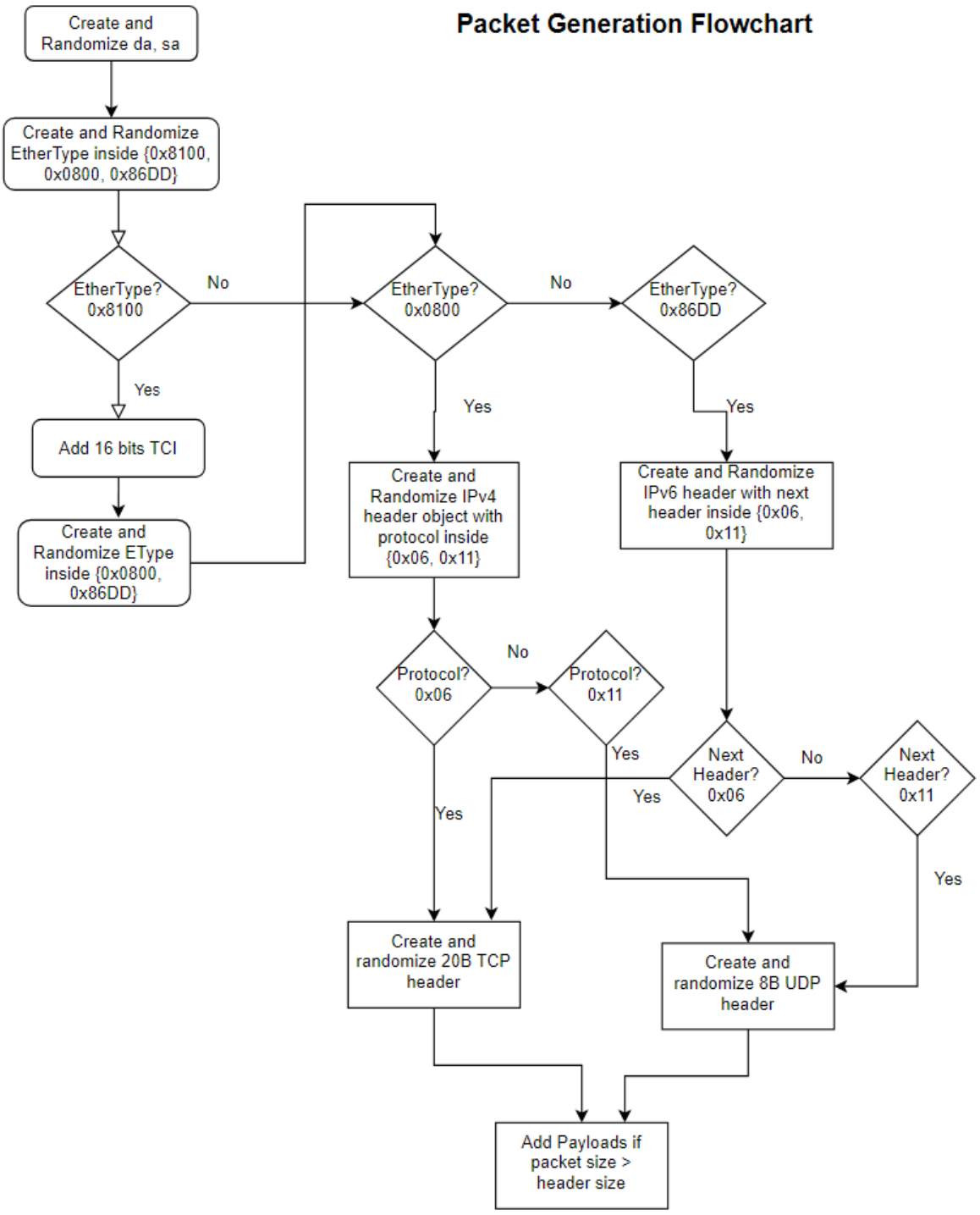
if next_header field in the **IPv6 Header** is **0x11**, 8 bytes are appended as UDP header with random values.

5. **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.
6. **Assembling the ethernet header, ipv4 header/ipv6 header, l4 header and payloads into packets: pack() function:**

Combining the generated Ethernet header object, IPv4 or IPv6 header object and transport layer byte array.

Adding any additional payload data based on the packet size.

A pack function assembles a byte array into a packet.



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. 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

4. Driver Implementation

Driver gets the packet from the mailbox, breaks it into fleets, and drive fleets to the signals in the interface modport.

Driver Gets the packet from generator using mailbox.

Driver implemented as a class, has the following members:

- virtual Interface
- Packet handle
- mailbox handle
- `Method to drive packet in fleets`

Mailbox.get() method is called to retrieve a packet from the mailbox.

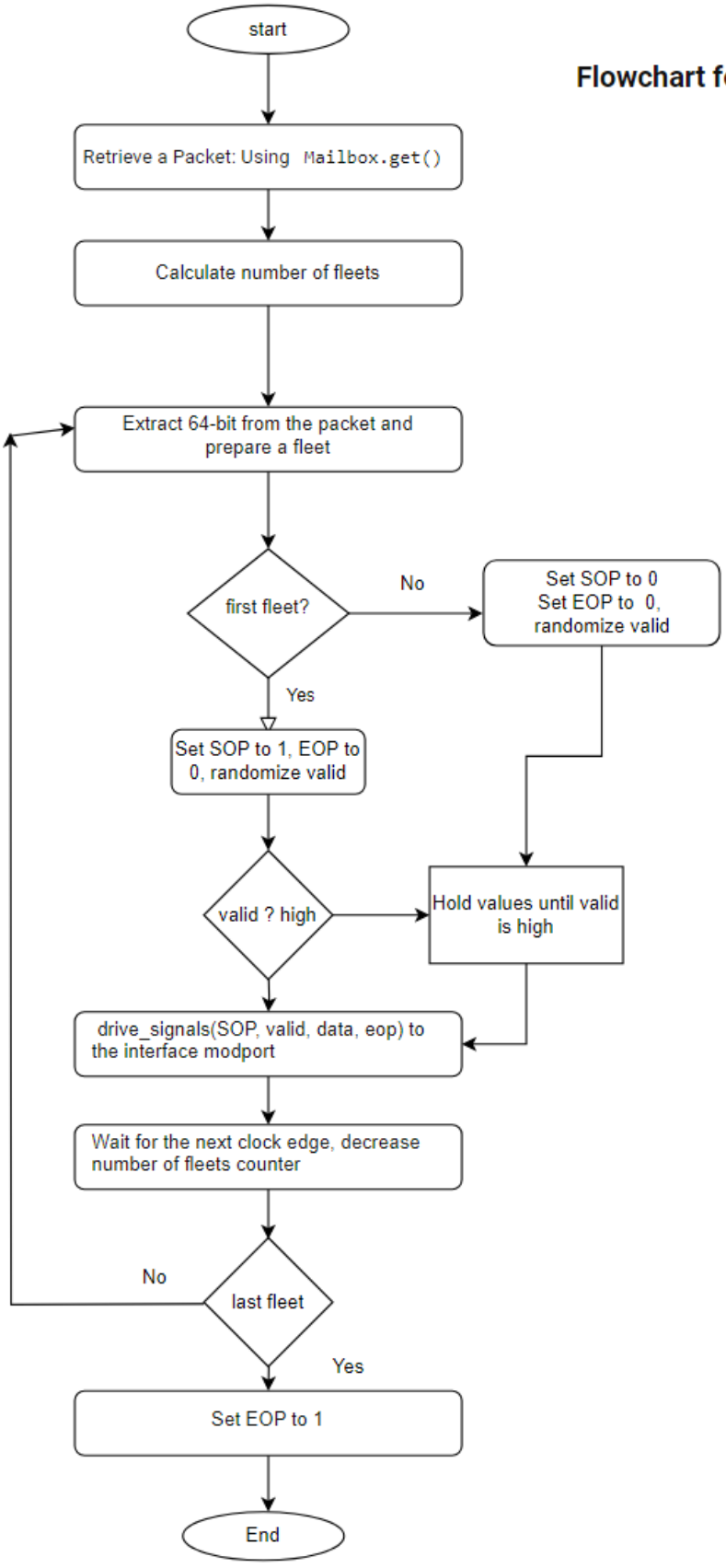
Flowchart that describes the process of transferring a packet in chunks from the driver to the interface:

1. **Start**: Begin the transfer process.
2. **Retrieve Packet**: Use `Mailbox.get()` to retrieve the packet from the mailbox.
3. Calculate the number of fleets needed to drive a packet.
4. **While Loop**: Continue transferring fleets until the entire packet is transferred.
    - **Prepare next fleet**: Extract the next 64-bits from the packet.

- **Set SOP**: If it's the first chunk, set `SOP` (Start of Packet) to 1; otherwise, set it to 0.
- **Set eop**: If it's the last chunk, set `eop` (End of Packet) to 1; otherwise, set it to 0.
- **Drive signal to the interface**: drive signals (`SOP`, `valid`, `data`, `eop`) according to the protocol. if valid is low, hold values etc.
- **Wait for Clock Edge**: Wait for the next clock edge to synchronize the transfer.

5. **End**: Complete the transfer process.

Flowchart for Drive Packet



5. Interface Implementation

Interface Defines the signals (`clock`, `SOP`, `valid`, `data`, `eop`) and amodport

Interface is instantiated inside the top module.

6. Top Module

- At the top module packet generator, driver, mailbox, interface objects are instantiated. mailbox handle is passed to the generator and driver class.
- Clock is created for sampling
- bounded mailbox of depth 1 is implemented
- Two 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
- vpd file generated to see the waveform.