**Problem 1:**

Design parse and match action logic in VHDL using Xilinx ISE. Write a testbench that include all possible test cases in order to verify your design using simulation. Implement your verified design on the ATLYS board to verify the proper functionality of your design on hardware.

**Multistage parser:**

Top-Level block diagram for your design is given in Fig.1. The descriptions of interfaces are given in Table-I. For design of your logic you need to use only the given interfaces at the top level of the design. Internally you can have signals of your choice.



Fig. 1. Parse and match action logic top level interfaces.

Table I. Parse and match action interface descriptions

|  |  |  |
| --- | --- | --- |
| Interface | Direction | Description |
| iData\_av | Input | This is a 1-bit signal. A logic high represents the availability of the data for this module at the remote interface. |
| iRd\_Data | Output | This is a 1-bit signal. This signal is driven by your logic for reading data from remote interface. |
| iData | Input | This is a 144 bit signal. This is the incoming data to your logic. You receive a valid data for each clock cycle when you assert i\_Rd\_Data signal in your logic. |
| oData\_av | Output | This is a 1-bit signal. A logic high represents the availability of the data at the output interface of your logic. This signal is driven by your logic. |
| oData\_rd | Input | This is a 1-bit signal. This signal is driven by remote logic for reading data from your logic. |
| oData | Output | This is a 144 bit signal. This is the outgoing data from your logic. You provide a valid data for each clock cycle when you assert oData\_rd signal is high. |
| Lkup\_data | Output | This is 128 bit signal. You provide this data based on the information extracted from the incoming data to your logic. |
| Lkup\_valid | Output | This is a 1-bit signal. You assert this signal high for 1-clock cycle after you have provided lkup\_data. |
| Lkup\_Flow\_miss | Input | This is a 1-bit signal. In response to the Lkup\_data you receive this input. Where  0🡪 no match in lookup table.  1🡪 a successful match in lookup table. |
| Lkup\_Flow\_priority | Input | This is a 3-bit signal. In response to the Lkup\_data you receive this input. This indicates the priority of the flow “000” indicates the lowest priority whereas “111” indicates the highest priority. |
| Lkup\_Flow\_id | Input | This is an 8-bit signal. In response to the Lkup\_data you receive this input. This represents an 8-bit unique flow id which is used by the rate limit logic. |
| Lkup\_Flow\_info | input | This is a 256-bit signal. In response to the Lkup\_data you receive this input. This represents the information related to the flow. |
| Lkup\_Flow\_instruction | input | This is an 8-bit signal. In response to the Lkup\_data you receive this input. This represents the instruction need to be executed on the data. |
| Lkup\_Flow\_info\_valid | Input | This is an 8-bit signal. This signal represents that the information available as a response of lookup is valid. |
| Flow\_id | Input | This is an 8-bit signal. this indicate the unique flow which should be configured for burst size and flow rate. |
| Burst\_Size | Input | This is a 16-bit signal. This indicate the number of back to back packets can be supported for a particular flow. |
| Flow\_rate | Input | This is a 16-bit signal. This indicate the information rate(Mbps) that can be allowed for a particular flow. |
| configure | Input | This is a 1-bit signal. A logic high on this signal indicates that we need to store the burst\_size and flow\_rate information for a flow mentioned using Flow\_id. |
| I\_Offset | Input | This is a 16-bit signal. This indicates a location in the data. |
| I\_Length | Input | This is a 3-bit signal. This indicates a length in bytes. |
| I\_Instruction | Input | This is a n 8-bit signal. This indicates the instruction to be executed on the incoming data. |
| O\_Offset | Input | This is a 16-bit signal. This indicates a location in the data. |
| O\_Length | Input | This is a 3-bit signal. This indicates a length in bytes. |
| O\_Instruction | Input | This is a n 8-bit signal. This indicates the instruction to be executed on the incoming data. |
| clk | Input | This is a 1-bit clock signal for the logic. |
| rst | Input | This is a 1-bit reset signal for the logic. |

**Design description:**

Incoming data will be in the format as shown in Fig. 2, where each 8-bit data is preceded by a valid signal (dv). dv indicates whether 8-bit data followed by dv is valid or not. dv🡪’1’ indicates a valid data and dv🡪’0’ indicates an invalid data. dv🡪’0’ in incoming data marks the end of packet for the current data.



Fig. 2. Incoming data format.

Your design should start fetching/reading data (iRd\_data=’1’) only when iData\_av input signal to your design is asserted to ‘1’ (logic High). Once your design asserts the iRd\_data, you will be able to get 144-bit data on each clock cycle.

Tasks to be performed on an incoming data packet:

1. Identify the end of packet marker.
2. Start counting number of bytes you received.
3. Identify a location given by the I\_offset in the incoming data (using your bytes count).
4. Extract I\_Length bytes of data from identified location.
5. Now provide this extracted I\_Length bytes to Lkup\_data signal as input.
   1. In case, size of I\_Length bytes is less than the size of Lkup\_data, append ‘1’s to extracted I\_Length bytes of data before giving this as input to Lkup\_data.
6. Assert Lkup\_data\_valid signal for one clock cycle once you have given I\_Length bytes of extracted data input to Lkup\_data.
7. Wait for the Lkup\_Flow\_info\_valid signal to go to logic high (from ‘0’🡪 ’1’).
   1. Once Lkup\_Flow\_info\_valid is high, store the other flow related information such as Lkup\_Flow\_miss, Lkup\_Flow\_priority, Lkup\_Flow\_id, Lkup\_Flow\_info, Lkup\_Flow\_instruction in respective registers.
8. In case, you receive Flow\_miss signal high, drop all the data till the end of packet marker and reset the counter once end of packet marker is detected.
9. Otherwise, give the flow\_id information to rate\_limit logic and check if sufficient tokens are available.
10. If sufficient tokens are available execute the Flow\_instruction as given in table-II on the data and store the data in output FIFO.
11. Extract the O\_offset, O\_length and O\_instruction fields from the Lkup\_Flow\_info and provide these information’s as output of your logic.
12. Create a Finite state machine (FSM) based on the tasks described above.



Fig. 3. Lkup\_Flow\_info signal format.

Lkup\_Flow\_info data format is shown in Fig. 3.

Table II. Instruction list.

|  |  |
| --- | --- |
| Instruction | Description |
| 00000000 | **Bypass:** provide input data as output without any processing |
| 00000001 | **Extract:** provide *Length*, *Offset* and *Instruction* fields of *Lkup\_Flow\_info* as *O\_offset, O\_length and O\_instruction* output respectively. |
| 00000010 | **Add Tag:** add the tag type and tag values at the *offset* location specified in *Lkup\_Flow\_info.* |
| 00000011 | **Swap:** swap the tag value of specified *Length* at *offset* location specified in *Lkup\_Flow\_info.* |
| 00000100 | **Drop:** drop the packet. |

You also need to design a logic for leaky bucket algorithm, we call it rate limit logic. In this logic you need to store the flow rate for 256 flows. (A flow is identified by a distinct flow\_id). Bucket will keep the token count of all 256 flows. Based on Lkup\_flow\_id you will extract token count for identified flow and start decreasing the token count based on number of bytes of data for that particular flow has been forwarded by your logic. You need to refill the token bucket for all the flows after a fixed time interval (your design choice) by appropriate amount. The amount by which you need to refill the token for a particular flow is provided to you in form of Flow\_id, Burst\_size and Flow\_rate at the time of flow setup. You need to store this information related to that particular flow in some register or block RAM. At the time of flow setup configure signal is asserted for storing this information.

Expected Outcome:

Commented synthesizable VHDL code of your design, which functions as described in problem statement.

Testbench written in VHDL for simulating the different possible test cases for your design.

**Problem 2:**

Design path control logic in VHDL using Xilinx ISE. Write a testbench that include all possible test cases in order to verify your design using simulation. Implement your verified design on the ATLYS board to verify the proper functionality of your design on hardware.

Top-Level block diagram for your design is shown in Fig.5. The descriptions of the interfaces are given in Table-III. For design of your logic you need to use only the given interfaces at the top level of the design. Internally you can have signals of your choice.

**Design description:**

Incoming data will be in the format as shown in Fig. 4, where each 8-bit data is preceded by a valid signal (dv). dv indicates whether 8-bit data followed by dv is valid or not. dv🡪’1’ indicates a valid data and dv🡪’0’ indicates an invalid data. dv🡪’0’ in incoming data marks the end of packet for the current data.



Fig. 4. Incoming data format.

Your design should start fetching/reading data (iRd\_data=’1’) only when iData\_av input signal to your design is asserted to ‘1’ (logic High). Once your design asserts the iRd\_data, you will be able to get 144-bit data on each clock cycle.



Fig. 5. top level interface for Path control logic.

Table III. Signal Description

|  |  |  |
| --- | --- | --- |
| Interface | Direction | Description |
| iData\_av | Input | This is a 1-bit signal. A logic high represents the availability of the data for this module at the remote interface. |
| iData\_rd | Output | This is a 1-bit signal. This signal is driven by your logic for reading data from remote interface. |
| iData | Input | This is a 144 bit signal. This is the incoming data to your logic. You receive a valid data for each clock cycle when you assert iData\_rd signal in your logic. |
| Exp\_Data\_av | Output | This is a 1-bit signal. A logic high represents the availability of the data at the output interface of your logic. This signal is driven by your logic. |
| Exp\_Data\_rd | Input | This is a 1-bit signal. This signal is driven by remote logic for reading data from your logic. |
| Exp\_Data | Output | This is a 144 bit signal. This is the outgoing data from your logic. You provide a valid data for each clock cycle when you assert Exp\_Data\_rd signal is high. |
| Buf\_Data\_av | Output | This is a 1-bit signal. A logic high represents the availability of the data at the output interface of your logic. This signal is driven by your logic. |
| Buf\_Data\_rd | Input | This is a 1-bit signal. This signal is driven by remote logic for reading data from your logic. |
| Buf\_Data | Output | This is a 144 bit signal. This is the outgoing data from your logic. You provide a valid data for each clock cycle when you assert Buf\_Data\_rd signal is high. |
| Output\_port | Output | This is an 8-bit signal, which represents the output port for the packet. |
| Output\_port\_valid | Output | This is a 1-bit signal, indicate the Output\_port **s**ignal is valid. |
| Wr\_cmd\_av | Output | This is a 1-bit signal indicating that the data is available to be written in memory. |
| Wr\_cmd\_rd | input | This is 1-bit signal, from remote logic to read the command. |
| Wr\_cmd | Output | This is a 32-bit signal providing the write command to remote logic. Write command consist of the start address of the memory where data need to be written. |
| Wr\_data | Output | This is a 144-bit signal carrying the data to be written in memory. |
| Wr\_data\_valid | Output | This is a 1-bit signal indicates that the data present on Wr\_datais valid and it should be written in memory**.**. |
| rd\_cmd\_av | Output | This is a 1-bit signal indicating that the data is available to be written in memory. |
| rd\_cmd\_rd | input | This is 1-bit signal, from remote logic to read the command. |
| rd\_cmd | Output | This is a 32-bit signal providing the write command to remote logic. Read command consist of the start address of the memory and size of the data to be read from memory. |
| rd\_data | Input | This is a 144-bit signal carrying the read data from memory. |
| rd\_data\_valid | Input | This is a 1-bit signal indicates that the data present on rd\_datais valid. |
| clk | Input | This is a 1-bit clock signal for the logic. |
| rst | Input | This is a 1-bit reset signal for the logic. |

Task to be performed on an incoming packet:

1. Identify the output port based on the port information in the data.
2. Identify the end of packet marker.
3. Based on the output port information decide where to send the data:
   1. On an express path
   2. On a buffered path
   3. To external memory
4. Identify the memory address in case data to be sent to an external memory.
5. Append a 144-bit header to the packet pertaining to the memory address where data need to be stored.
6. Count the size of the data to be written in the memory.
7. Once the data pertaining to the packet is completely transferred (written) to memory, store the packet size.
8. Once sufficient buffer is available, create a read command to read a packet from the memory.
9. For reading packets, reading is done based on the priority.
   1. There are 4 priority queues and weights are assigned to each priority queue
      1. Q3🡪64
      2. Q2🡪32
      3. Q1🡪16
      4. Q0🡪8
   2. These weights indicate the number of packets to be read for each priority. 1st we read 64 packets for Q3 then 32 packets of Q2, then 16 packets of Q1 and finally 8 packets of Q0.
   3. In case number of available packets are lesser than the weights value, we need to move to next weight after reading all packets.
   4. Repeat the cycle for reading.
10. For reading packets from memory, identify the address to read based on priority weights and give it as output.
11. Make sure if packet for a particular output port and particular priority is available in memory or in buffer all subsequent packets need to follow same path.

Expected Outcome:

Commented synthesizable VHDL code of your design, which functions as described in problem statement.

Testbench written in VHDL for simulating the different possible test cases for your design.

**Problem statement 3:**

Design Match table lookup logic in VHDL using Xilinx ISE. Write a testbench that include all possible test cases in order to verify your design using simulation. Implement your verified design on the ATLYS board to verify the proper functionality of your design on hardware.

Top-Level block diagram for your design is shown in Fig.6. The descriptions of the interfaces are given in Table-IV. For design of your logic you need to use only the given interfaces at the top level of the design. Internally you can have temporary signals of your choice.



Fig. 6. Top level interfaces of Match Table lookup logic.

Table IV. Signal description.

|  |  |  |
| --- | --- | --- |
| Interface | Direction | Description |
| Lkup\_data | Input | This is an N X 128 bit signal. Where N represents the number of ports on the switch. This is the key for 1st stage lookup. |
| Lkup\_valid | Input | This is a N X 1-bit signal. A high login on one of the N input represents the valid lookup data at repective Nx128 lkup\_data. |
| Lkup\_Flow\_miss | output | This is a 1-bit signal. In response to the Lkup\_data you received. Where  0🡪 no match in lookup table.  1🡪 a successful match in lookup table. |
| Lkup\_Flow\_priority | Output | This is a 3-bit signal. This indicates the priority of the matched flow “000” indicates the lowest priority whereas “111” indicates the highest priority. |
| Lkup\_Flow\_id | Output | This is an 8-bit signal. This represents an 8-bit unique matched flow id pertaining to a particular port, which is used by the rate limit logic. |
| Matched\_flow\_tag | Output | This is a 256-bit signal. This represents the information related to the matched flow. |
| Matched\_flow\_tag\_valid | Output | This is an 8-bit signal. This signal represents that the information available on Matched\_flow\_tag is valid. |
| Flow\_addr | Input | This is a 10-bit signal representing the address in the table to write flow information. |
| Flow\_lkup\_info | Input | This is the 128-bit lookup information to be written in table. |
| Lkup\_info\_wr | Input | This is a 1-bit signal indicates that Flow\_lkup\_infoshould be written at theFlow\_addr**.** |
| Flow\_tag | Input | This is a 256-bit signal represent the tag to be written in 2nd stage table. |
| Flow\_tag\_wr | Input | 1-bit signal indicate that the available data on Flow\_tag is valid. |
| Active\_flow\_addr | Input | 10-bit Address of the active flow. |
| Active\_flow | Input | 1-bit Active flow information. |
| Active\_flow\_wr | Input | 1-bit to assert write active\_flow and active\_flow\_addr. |
| clk | Input | This is a 1-bit clock signal for the logic. |
| rst | Input | This is a 1-bit reset signal for the logic. |

**Design description:**

This design is required for the lookup process against a list of keys coming from the parse and match action logic. You need to perform the lookup process in two stages. In first stage you will match the provided key in the table. Once you find a valid match in the table, it will provide you the offset (address) for the second stage. Format of first stage table is shown in Fig.7. Using this offset you can get the final lookup information. The format of the lookup information is shown in Fig.8. you need to provide this lookup information to respective parse and match action logic.



Fig. 7. First stage table format



Fig. 8. Lookup output Format

Tasks to perform:

1. Since, multiple parse and match action logic are trying to initiate lookup in same table. You need to design an arbiter to schedule the lookups.
2. You need to split your 1st stage table in multiple tables so that parallel lookup can be done in multiple tables.
3. Once a match is found in a table, you need to terminate other parallel lookups against the same key.
4. You can have single table for second stage lookup.
5. You need to create a proper interface from 1st stage to second stage such that second stage can be accessed by only one of the matched offset. Other matched offset need to be stored and should wait for a lookup to complete.
6. Based on the offset, extract the lookup information and give it as output.
7. Write Flow\_lookup\_info at the flow\_addr in 1st stage lookup table when Lkup\_info\_wr is at logic high.
8. Write flow\_tag at flow\_addr in 2nd stage lookup table when Flow\_tag\_wr is at logic high.
9. For each flow two entries are written in the 2nd stage table. One corresponds to primary path and 2nd corresponds to protection path. But your lookup will give only one information which is active.
10. Once active\_flow\_wr is asserted, you need to update the active flow information at address active\_flow\_adress by active\_flow.
    1. Active\_flow
       1. ‘0’🡪Primary path active
       2. ‘1’🡪Protection path active

Expected Outcome:

Commented synthesizable VHDL code of your design, which functions as described in problem statement.

Testbench written in VHDL for simulating the different possible test cases for your design.

**Problem statement 4:**

Design Contention Resolution and switching logic in VHDL using Xilinx ISE. Write a testbench that include all possible test cases in order to verify your design using simulation. Implement your verified design on the ATLYS board to verify the proper functionality of your design on hardware.

Top-Level block diagram for your design is shown in Fig.9. The descriptions of the interfaces are given in Table-V. For design of your logic you need to use only the given interfaces at the top level of the design. Internally you can have temporary signals of your choice.



Fig. 9. Top level interfaces of contention resolution and switching logic.

Table V. Signal description.

|  |  |  |
| --- | --- | --- |
| Interface | Direction | Description |
| Outport\_Data\_av | Output | This is a Nx1-bit signal. Where N represents the number of ports in the switch. A logic high represents the availability of the data at your logic for respective remote interface logic. |
| Outport\_Data\_rd | Input | This is a Nx1-bit signal. This signal is driven by remote interface logic for reading data for respective remote interface. |
| Outport\_Data | Output | This is a Nx144 bit signal. This is the 144-bit outgoing data from your logic to respective N-ports. Remote interface gets a valid data for each clock cycle when it assert Exp\_Data\_rd signal to your logic. |
| Exp\_Data\_av | Input | This is a Nx1-bit signal. Where N represents the number of ports in the switch. A logic high represents the availability of the data at the respective remote interface logic. |
| Exp\_Data\_rd | Output | This is a Nx1-bit signal. This signal is driven by your logic for reading data from the respective remote interface. |
| Exp\_Data | Input | This is a Nx144 bit signal. This is the incoming data to your logic from N-ports. You receive a valid data for each clock cycle when you assert respective Exp\_Data\_rd signal in your logic. |
| Buf\_Data\_av | Input | This is a Nx1-bit signal. Where N represents the number of ports in the switch. A logic high represents the availability of the data at the respective remote interface logic. |
| Buf\_Data\_rd | Output | This is a Nx1-bit signal. This signal is driven by your logic for reading data from the respective remote interface. |
| Buf\_Data | Input | This is a Nx144 bit signal. This is the incoming data to your logic from N-ports. You receive a valid data for each clock cycle when you assert respective Buf\_Data\_rd signal in your logic. |
| Output\_port | Output | This is an 8-bit signal, which represents the output port for the packet. |
| Output\_port\_valid | Output | This is a 1-bit signal, indicate the Output\_port **s**ignal is valid. |
| clk | Input | This is a 1-bit clock signal for the logic. |
| rst | Input | This is a 1-bit reset signal for the logic. |

**Design description:**

There are N input and N output ports in the switch. A packet coming from a port can go to one of the N output port. In your design you need to create a virtual output queueing logic, where each input have N output buffers corresponding to N output port. This logic helps in avoiding head of line blocking. This way you will have total NxN buffers in your logic and your output port scheduling logic will read packets from these N buffers. There is possibility that packets coming at two different input port are destined to same output port at the same time. Any output port can only be accessed by one of the input port at any given instance of the time. Therefore, you also need to design a scheduling logic such that each output port runs their individual scheduling logic and allow access to appropriate input port.

For example, a packet coming at input port 3 which have its destination at output port 5, then you need to store packet in buffer number 5 of N buffers allocated to input port 3. Similarly, if a packet is coming at input port 8, which have its destination at output port 5, then you need to store packet in buffer number 5 of N buffers allocated to input port 8. Port-5 faces contention from Input port-3 and port-8. Therefore, there is a requirement of scheduling mechanism to allow only one port to transmit data to output port at a time.

Tasks to perform:

1. Create NxN VOQs.
2. Read packet over exp\_data or buf\_data signal.
3. Based on the input port and requested output port, write packet in respective VOQ.
4. Design an arbiter to schedule packet transfer from N inputs to a particular output port.
5. Arbiter should connect only one input port to a output port at any given instance of time.
6. Identify the end\_of\_packet marker. Once end\_of\_packet marker detected allow next input port to transfer packet.
7. Create N instances of such arbiter one for each output port and run them in parallel.

Expected Outcome:

Commented synthesizable VHDL code of your design, which functions as described in problem statement.

Testbench written in VHDL for simulating the different possible test cases for your design.

**Problem statement 5:**

Design classify\_packet logic in VHDL using Xilinx ISE. Write a testbench that include all possible test cases in order to verify your design using simulation. Implement your verified design on the ATLYS board to verify the proper functionality of your design on hardware.

Top-Level block diagram for your design is shown in Fig.10. The descriptions of the interfaces are given in Table VI. For design of your logic you need to use only the given interfaces at the top level of the design. Internally you can have temporary signals of your choice.



Fig. 10. Classify packet logic's top-level interfaces.

Table VI. signal Description.

|  |  |  |
| --- | --- | --- |
| Interface | Direction | Description |
| iData\_av | Input | This is a N+2-bits signal. A logic high represents the availability of the data for this module at the remote interface. |
| iRd\_Data | Output | This is a N+2-bits signal. This signal is driven by your logic for reading data from remote interface. At a time only one of the N+2-bits can be high. |
| iData | Input | This is a (N+2)x144 bit signal. This is the incoming data to your logic. You receive a valid data for each clock cycle when you assert respective iRd\_Data signal in your logic. |
| oData\_av | Output | This is a 1-bit signal. A logic high represents the availability of the data at the output interface of your logic. This signal is driven by your logic. |
| oData\_rd | Input | This is a 1-bit signal. This signal is driven by remote logic for reading data from your logic. |
| oData | Output | This is a 144 bit signal. This is the outgoing data from your logic. You provide a valid data for each clock cycle when you assert oData\_rd signal is high. |
| MAC | Input | This is a 48-bit signal. This represents the MAC address of your design. |
| Opcode | Output | This is a 3-bits signal. This represents classification of incoming packets based on your logic. |
| Rd\_opcode | Input | This is a 1-bit signal to read the opcode from remote logic. |
| Input\_port | Output | This is a 5-bit signal representing the selected input port |
| Port\_mask | Output | This is a 32-bit signal, where index of each bit represents an output port. Respective index bit is made high by the designed logic representing the destination port for the packet. otherwise it remains at logic low. |
| Edge\_ports | Output | This is a 32-bit signal, a logic high on a bit represents the respective ports as edge port. |
| Core\_Ports | Output | This is a 32-bit signal, a logic high on a bit represents the respective ports as core port. |
| clk | Input | This is a 1-bit clock signal for the logic. |
| rst | Input | This is a 1-bit reset signal for the logic. |

Design description:

There will be inputs from N+2 ports. You need to select an input from N+2 ports and read the available data on selected input port. Therefore, you need to design an arbiter to select an input port. Based on the packet’s Ethertype and Ether value you need to classify the packet and provide the classification as output “Opcode” and selected input port as “Input\_Port”. You need to extract the path pointer and path vectors from the packet. Using these path vectors you need to forward the packet at the destination port. Few frames need to be forwarded as well processed by the logic.i.e. Hello frame. Frame format and Ether values of different frames are shown in Fig. 11.



Fig. 11. Frame format and different opcodes

Tasks to perform:

1. Design an arbiter to select from N+2 input port.
2. Extract the **Ethertype** and **Ether value** from a packet.
3. Extract the path pointer and path vector information from the packet.
4. Extract the **target MAC address** form the packet.
5. Match the extracted **target MAC address** with **MAC** address of the logic, in case no other valid path-vector present.
   1. In case, **MAC** address matches
      1. Classify the packet based on the **Ether value**.
   2. In case the **MAC** address does not match
      1. drop this packet.
6. In case a valid path-vector present in packet, identify the output port based on the path pointer and path vector.
7. Mark **opcode** as “forward” in order to forward the packet to destination port.
8. Make the destination port index in “**Port\_mask”** high**.**
9. Increment the path pointer to next value.
10. In case, the packet is of type **inter\_switch** you need to mark the input port of the packet as core port and reset the counter.
11. The counter should count 1 sec.
    1. In case, timer gets timed out and you don’t receive an **inter\_switch** packet on a port, you need to mark that port as **edge port**.
12. Create an FSM for the above-mentioned tasks.

**Problem statement 6:**

Design write logic in VHDL using Xilinx ISE. Write a testbench that include all possible test cases in order to verify your design using simulation. Implement your verified design on the ATLYS board to verify the proper functionality of your design on hardware.

Top-Level block diagram for your design is shown in Fig.12. The descriptions of the interfaces are given in table-VII. For design of your logic you need to use only the given interfaces at the top level of the design. Internally you can have temporary signals of your choice.



Fig. 12. Write logic's top-level interfaces.

Table VII. signal Description.

|  |  |  |
| --- | --- | --- |
| Interface | Direction | Description |
| iData\_av | Input | This is a N+2-bits signal. A logic high represents the availability of the data for this module at the remote interface. |
| iRd\_Data | Output | This is a N+2-bits signal. This signal is driven by your logic for reading data from remote interface. At a time only one of the N+2-bits can be high. |
| iData | Input | This is a (N+2)x144 bit signal. This is the incoming data to your logic. You receive a valid data for each clock cycle when you assert respective iRd\_Data signal in your logic. |
| opcode | Input | This is a 3-bits signal, it represents the type of packet which is available at the iData\_av. |
| Rd\_opcode | output | This is 1-bit signal. your logic drive this signal high in case the opcode=3**.** |
| Wr\_valid | Output | This is a 1-bit signal. A logic high represents the availability of the extracted data at the output interface of your logic. This signal is driven by your logic. |
| Wr\_complete | Input | This is a 1-bit signal. This signal is driven high by remote logic once the extracted data is written in respective registers. |
| Write\_Data | Output | This is a 512 bit signal. This is the outgoing extracted data aligned to LSB from your logic. Extracted data is valid when the wr\_valid is at logic\_high. |
| Input\_port | Input | This is a 5-bit signal. this represents the port from where packet is coming to logic. |
| Table\_number | Output | This is an 8-bit signal. This represents the table number where registers need to be written. |
| Table\_address | Output | This is a 16-bits signal. This represents the offset location in table to write the registers |
| Sequence\_number | Output | This is 16-bit signal, it represents the extracted sequence number of the packet. |
| Output\_port | output | This is 5-bit signal, this represents the port from where packet is received in order to send an acknowledgement. |
| Send\_wr\_ack | output | This is 1-bit signal, Once write\_complete is asserted this logic should assert send\_wr\_ack signal. |
| Ack\_sent | Input | This is 1-bit signal, Once send\_wr\_ack is asserted is driven by the remote interface to deassert the send\_wr\_ack from the logic. |
| clk | Input | This is a 1-bit clock signal for the logic. |
| rst | Input | This is a 1-bit reset signal for the logic. |

Design description:

In this design, you need to extract the relevant data from the incoming packet based on the table number. You also need to assert the send\_wr\_ack signal once the data is extracted and written in the respective registers. Write packet format is shown in Fig. 13.



Fig. 13. Write Frame format

Tasks to perform:

1. Check the idata\_av signal for logic high.
2. Check if wr\_complete signal is logic high.
3. Check if opcode input to logic is = 3.
   1. In case opcode=0x“3” start reading the packet by asserting the iRd\_Data =’1’;
4. Extract the frame sequence number.
5. Extract the table number and table address information from the packet.
6. Identify the data present in the packet.
7. Based on the table number extract the relevant information from the data present in packet.
8. Give the extracted data, table no and table address as output.
9. Once you get wr\_complete=’1’, assert send\_ACK signal and give frame sequence number and output\_port as output of your logic.
10. Identify the end of the packet marker. Once you find the end of packet marker assert “Rd\_opcode**”** signal for 1 clock cycle.
11. Wait for wr\_complete and Ack\_sent to be asserted.
12. Create the FSM based on above tasks.

**Problem statement 7:**

Design CFM logic in VHDL using Xilinx ISE. Write a testbench that include all possible test cases in order to verify your design using simulation. Implement your verified design on the ATLYS board to verify the proper functionality of your design on hardware.

Top-Level block diagram for your design is shown in Fig.14. The descriptions of the interfaces are given in Table-VIII. For design of your logic you need to use only the given interfaces at the top level of the design. Internally you can have temporary signals of your choice.



Fig. 14.CFM logic's top-level interfaces.

Table VIII. signal Description.

|  |  |  |
| --- | --- | --- |
| Interface | Direction | Description |
| iData\_av | Input | This is a N+2-bits signal. A logic high represents the availability of the data for this module at the remote interface. |
| iRd\_Data | Output | This is a N+2-bits signal. This signal is driven by your logic for reading data from remote interface. At a time only one of the N+2-bits can be high. |
| iData | Input | This is a (N+2)x144 bit signal. This is the incoming data to your logic. You receive a valid data for each clock cycle when you assert respective iRd\_Data signal in your logic. |
| opcode | Input | This is a 3-bits signal, it represents the type of packet which is available at the iData\_av. |
| Rd\_opcode | output | This is 1-bit signal. your logic drives this signal high in case the opcode=5**.** |
| Wr\_valid | Intput | This is a 1-bit signal. A logic high represents the availability of the extracted data at the intput interface of your logic. |
| Wr\_complete | Output | This is a 1-bit signal. This signal is driven high by your logic once the extracted data is written in the table. |
| Write\_Data | Input | This is a 512 bit signal. This is the incoming extracted data aligned to LSB from remote logic. Extracted data is valid when the wr\_valid is at logic\_high. |
| Input\_port | Input | This is a 5-bit signal. this represents the port from where packet is coming to logic. |
| Table\_number | Input | This is an 8-bit signal. This represents the table number, where registers need to be written. |
| Table\_address | Intput | This is a 16-bits signal. This represents the offset location in table to write the registers |
| Sequence\_number | Output | This is 16-bit signal, it represents the sequence number for the packet to be generated. |
| Output\_port | Output | This is 5-bit signal, this represents the port where generated CFM m. |
| CFM\_Flow\_id | Output | This is a 16-bit signal, this represents the entry offset in CFM table. |
| CFM\_set\_active | Output | This is a 1-bit signal, this is asserted to set the entry as active path which is present at CFM\_flow\_id offset. |
| CFM\_Information | Output | This is a 256-bit, information, based on the CFM entry present in the table. This information is used by the generate frame logic for packet generation. |
| Gen\_CFM | Output | This is 1-bit signal, Once write\_complete is asserted this logic should assert send\_wr\_ack signal. |
| CFM\_sent | Input | This is 1-bit signal, Once send\_wr\_ack is asserted is driven by the remote interface to deassert the send\_wr\_ack from the logic. |
| clk | Input | This is a 1-bit clock signal for the logic. |
| rst | Input | This is a 1-bit reset signal for the logic. |

Design description:

There are two CFM table entries corresponding to each Forwarding lookup Table entry – one for the primary path and other for the protection path. These entries in the CFM table have the primary and protection path information’s. You need to send a CFM messages for each valid entry present in CFM table at regular interval of 3ms. As a response you will also get a CFM message at an interval of 3ms. Your logic needs to keep a counter for each entry in the CFM table. Your logic will increment the counter value by 1 in every 3ms and this counter value must be decremented on each received CFM message. Once your counter value for any particular entry reaches to a value of 3, then you need to switch the path from primary-to-protection or protection-to-primary. A CFM frame format is shown in Fig. 15



Fig. 15. CFM Frame format

Tasks to perform:

1. Start 3ms counter and generate a tick for 3ms count.
2. Generate a CFM\_message counter for each of the CFM entry.
3. Check the idata\_av signal.
4. If it is high and FSM\_bsy signal of your design is low, check the opcode.
5. In case, opcode=0x”5” start reading the packet by asserting the ird\_data=’1’;
6. Extract the frame sequence number.
7. Extract the flow id.
8. Decrement the CFM\_message counter for an entry at location flow\_id in CFM table by 1.
9. In case, CFM message counter value of any of the entry becomes 3, switch the path.
10. Assert gen\_CFM for every entry present in CFM table once in every 3ms.
11. When Wr\_valid **signal is high**
    1. Check if Table\_number=6
       1. Write the relevant data from Write\_data signal.
       2. Write extracted data at the table\_address location in CFM table.
       3. Assert wr\_complete signal for 1 clock cycle.
12. Create the FSM based on above tasks.

**Problem statement 8:**

Design frame\_generate and forward logic in VHDL using Xilinx ISE. Write a testbench that include all possible test cases in order to verify your design using simulation. Implement your verified design on the ATLYS board to verify the proper functionality of your design on hardware.

Top-Level block diagram for your design is shown in Fig.16. The descriptions of the interfaces are given in table-IX. For design of your logic you need to use only the given interfaces at the top level of the design. Internally you can have temporary signals of your choice.



Fig. 16. Frame\_generate\_Forward top level interface.

Table IX. Signal Description

|  |  |  |
| --- | --- | --- |
| Interface | Direction | Description |
| iData\_av | Input | This is a1-bit signal. A logic high represents the availability of the data for this module at the remote interface. |
| iRd\_Data | Output | This is a -bit signal. This signal is driven by your logic for reading data from remote interface. |
| iData | Input | This is a 144 bit signal. This is the incoming data to your logic. You receive a valid data for each clock cycle when you assert iRd\_Data signal in your logic. |
| oData\_av | Output | This is a 1-bit signal. A logic high represents the availability of the data at the output interface of your logic. This signal is driven by your logic. |
| oData\_rd | Input | This is a 1-bit signal. This signal is driven by remote logic for reading data from your logic. |
| oData | Output | This is a 144 bit signal. This is the outgoing data from your logic. You provide a valid data for each clock cycle when you assert oData\_rd signal is high. |
| Output\_port\_mask | Output | This is a 32-bit signal. Each bit represents an output port. If a bit is asserted high, available data should be sent on the respective output port. |
| MAC | Input | This is a 48-bit signal. This represents the MAC address of your design. |
| Opcode | Input | This is a 3-bits signal. This represents classification of incoming packet. |
| Rd\_opcode | Output | This is a 1-bit signal to read the opcode from your logic. |
| Input\_port | Input | This is a 5-bit signal representing the input port of the packet. |
| Port\_mask | Input | This is a 32-bit signal, where index of each bit represents an output port. Respective index bit is made high by the remote logic representing the destination port for the packet. otherwise it remains at logic low. |
| Port\_number | Output | This is 5-bit signal, indicating the port number for reading the port’s status. |
| Rd\_port\_info | Output | This is 1-bit signal. This is asserted to request a port’s status where port number is available on Port\_number signal. |
| Port\_info | Input | This is an 80-bit signal. This provides status of the requested port. |
| Port\_info\_valid | Input | This is a 1-bit signal, it represents that the status available on Port\_info signal is valid. |
| CFM\_Sequence\_number | Input | This is 16-bit signal, it represents the sequence number for the CFM packet to be generated. |
| Output\_port | Input | This is 5-bit signal, this represents the port where generated CFM m to be sent. |
| CFM\_Information | Input | This is a 256-bit, information. This information is used to generate the CFM frame. |
| Gen\_CFM | Input | This is 1-bit signal, input to this logic. This signal represents a request to generate the CFM frame. |
| CFM\_sent | Output | This is 1-bit signal, it is sent in response to **Gen\_CFM**. Once the CFM frame is generated and sent to respective output port. |
| Wr\_Sequence\_number | Input | This is 16-bit signal, it represents the sequence number of the wr ack packet. |
| Output\_port | Input | This is 5-bit signal, this represents the port, where generated wr\_ack packet need to be sent. |
| Send\_wr\_ack | output | This is 1-bit signal, This signal is request to this logic to generate the Wr\_ack message. |
| Ack\_sent | Input | This is 1-bit signal, In response to Wr\_ack message this signal is asserted by this logic to indicate that ack message is sent.. |
| Core\_ports | Input | This is a 32 bit signal. If a bit is high in this signal, it represents that the respective port is marked as core port. |
| MAC | Input | This is 32-bit signal. It is the MAC address of the design. |
| clk | Input | This is a 1-bit clock signal for the logic. |
| rst | Input | This is a 1-bit reset signal for the logic. |

Design description:

In this design, you need to generate various messages and send them to desired port. This logic is also used for forwarding an incoming packet to desired output port. Output port is provided by masking the index of output port in Output\_port\_mask signal. Generated messages are stored in a FIFO. Remote logic read the message from output fifo based on the availability of the data in fifo. Format of different messages to be generated are shown in Fig 17-19.



Fig. 17. CFM Frame format



Fig. 18. write ack message format.



Fig. 19. Hello ack message format

Tasks to perform:

1. Check the idata\_av signal for logic high.
2. Check if FSM\_bsy signal of your design is low.
3. Check if opcode input to logic is = 1.
   1. In case opcode=0x“1” start reading the packet by asserting the iRd\_Data =’1’;
   2. Mark the FSM\_bsy=’1’;
   3. Extract the sequence number from the packet.
   4. Forward this packet to all the core ports except the input port.
   5. Generate a hello ack frame using the extracted sequence number.
   6. Send this generated reply frame on the output\_port by providing output\_port\_mask(input\_port)=’1’.
   7. Identify the end of the packet marker. Once you find the end of packet marker assert “Rd\_opcode**”** signal for 1 clock cycle.
   8. Mark FSM\_bsy=’0’;
4. In case no data is available on idata\_av.
5. Check if send\_wr\_ack or gen\_cfm signals are high.
   1. If the signals are high, Mark the FSM\_bsy=’1’;
   2. Use the respective sequence number and other relevant information to generate the message.
   3. Provide the output port by masking respective bit in output\_port\_mask signal to ‘1’
   4. Mark FSM\_bsy=’0’;
6. Create the FSM based on above tasks.