Hyper-Transport eVC
User Guide v1.0

Table of Contents

Overview

The Hyper-Transport (HT) test-bench is an ***e*** environment designed to provide verification and modeling functionality for the Hyper-Transport bus protocol.
The test-bench has two modes of operation:

Standalone Mode – This is a standalone environment with no Verilog simulator attached. This mode is useful for the design of test cases, evolution of the test bench and for traffic profiling. Clocking the design and traffic passing between devices is handled by ***e*** environment.

Test Mode – This mode of operation requires a DUT and an HDL simulator to be connected to the test environment. An external clock needs to be created and DUT wires must be used to connect to each ***e*** device in the test bench. This is the verification mode of operation.

Installation and Integration

Installing the HT Model

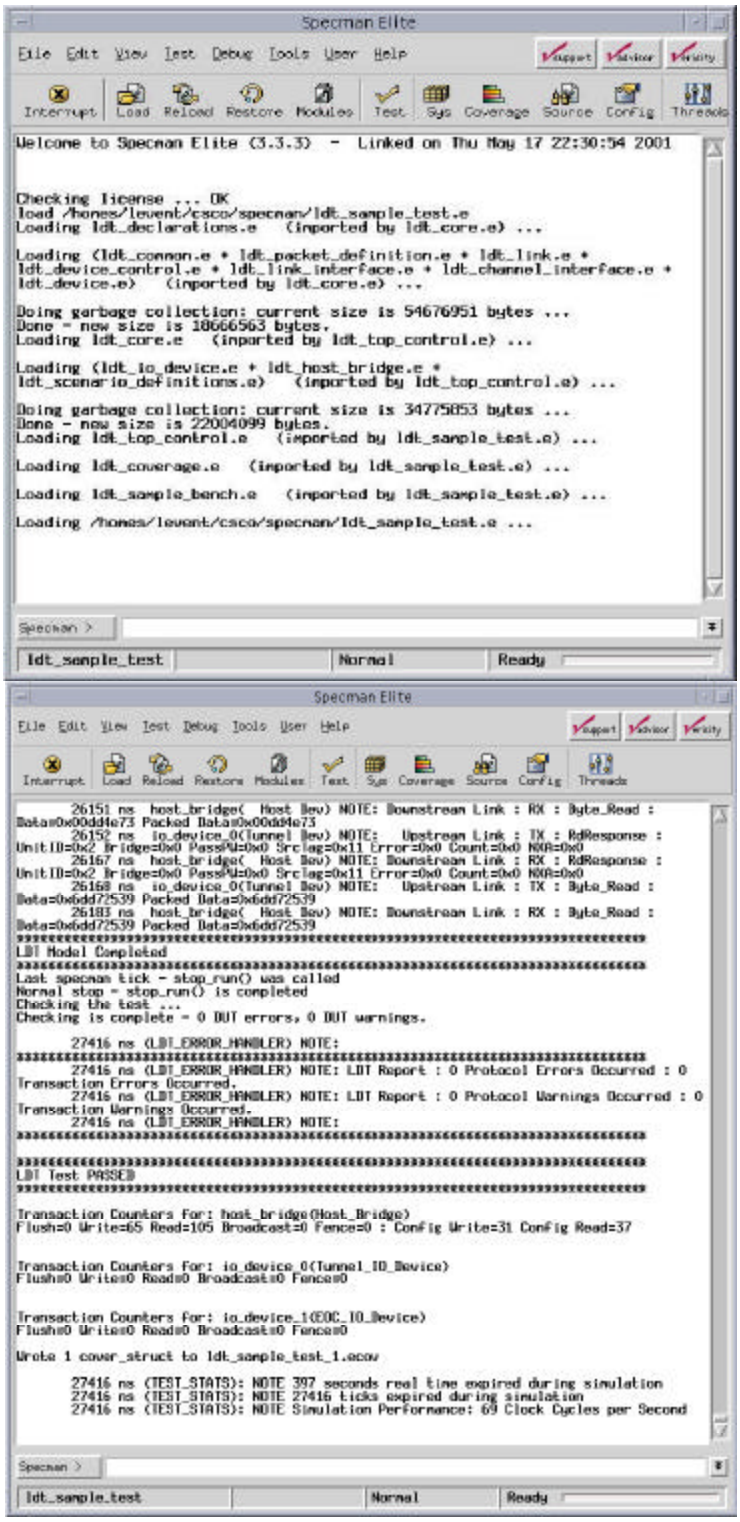
To install the HT model, follow the steps outlined below:

1. Download the latest Hyper-Transport model from Vault.
2. Extract the contents of the tar file to your installation directory.
   This creates the directory *Hyper Transport* and it contains all the necessary files to run the HT model.
   A sample test is also provided in this directory.

Running the HT in Standalone Mode

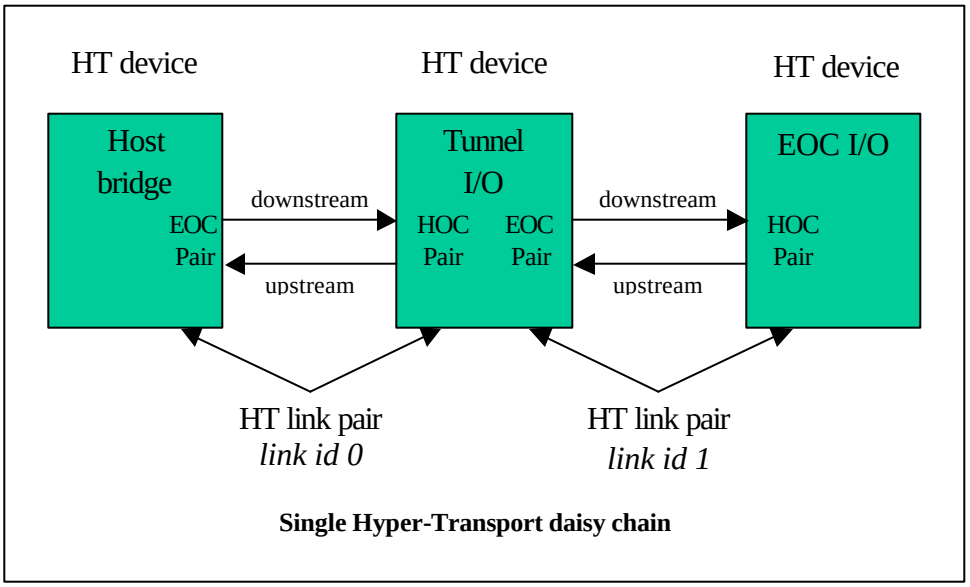
1. Make sure that Specman Elite is installed in your environment.
2. Add the *Hyper Transport* directory to the environment variable $SPECMAN_PATH.
3. Load <install_directory>/HyperTransport/ht_sample_test.e
4. Execute *"test"*

Upon the successful completion of the session, the HT model will print out the **"HT Test Passed"** message.



Successful completion of this test ensures the correct installation of the HT model.

Hyper-Transport Environment Configuration



Single Hyper-Transport daisy chain

A single HT daisy chain, from host-bridge to the end of chain device, is referred as a single bus. Each link pair within a bus has a specific link id. '0' is the link id connecting the host bridge to the first IO device and the devices down the link will receive incremented link ids.

Basics of Hyper-Transport Structure and Terminology

HOC: Head of chain
EOC: End of chain
Upstream: Traffic traveling towards the host bridge.
Downstream: Traffic traveling towards the end of chain.
Upstream interface: Interface closest to the Host Bridge (i.e. the link with a TX upstream path).
Downstream interface: Interface pointing away from the Host Bridge (i.e. the link with a TX downstream path).
Bit time: A bit time is the time window necessary to transmit a slice of data on a given HT link. Different HT links can work on different clock frequencies.

Logical Hyper-Transport Structure

The HT model is divided into several logical parts:

Link Interface is the lowest logical level of the HT model and is the part that actually interacts with the bus at a wire level. It is the single point of contact for the model to interact with the DUT and responsible for framing an HT packet into the appropriate number of bit times and generating/checking the CRC for each bit lane.

The Channel is the logical interface to the HT link. It handles all HT traffic at the packet level and manages the inbound and outbound packet ordering. It takes the HT packets and helps to assemble/disassemble the packet for interaction with the link interface. The channel allows the device to present packets to be transmitted and handles the scheduling. Additionally the channel will present received packets to the device.

The Device is the main control node sitting on the HT chain. It is responsible for generating and managing traffic at a high level. It will then pass the desired traffic to the HT Channel for interaction with the actual HT protocol/bus.

Test Writing Interface

The test file "ht_sample_test.e", included in the installation, contains all of the constraints that the user should adjust to customize their test environment and the test to be run.

Note: Any generation constraint outside of this list is intended to keep the generator from creating illegal/invalid test cases.

Defining the System Topology

The flowing examples define a Host Bridge and an EOC Device implemented by the HT model. These two devices interact with the DUT which is a Tunnel Device implemented in Verilog HDL.

Defining HT devices:

The following steps are necessary in order to set up the HT protocol devices and the system topology. More detailed information can be found in the "ht_sample_test.e" file, where set up includes an "*e*" Host Bridge and EOC device surrounding a Verilog Tunnel device (DUT).

Defining a Host Bridge Device

Instantiate a basic Host Bridge device:

```
example_host : ht_device is instance;
keep example_host.device_function == Host_Bridge;
```

Set up the HDL path to the interface point for this device:

```
keep example_host.hdl_path() == "testbench";
```

Connect system wide Power and Reset signals:

```
keep example_host.Reset_L == "HT_RESET_NEG";
keep example_host.PowerOk == "HT_POWEROK";
```

Define the signal interface for the link:

```
keep for each (interface) in example_host.ht_interfaces {
```

Define the basic clock for the link interface:

```
interface.ht_link.ht_link_fundamental_clock_signal == appendf("HT_CLK");
```

Define the signals for the DUT's receive path (The 'e' device's TX path):

```
    for each in interface.ht_link.clk_tx_H {
      it == appendf("HT_RX_CLK_H");
    };
    for each in interface.ht_link.clk_tx_L {
      it == appendf("HT_RX_CLK_L");
    };
```

```
interface.ht_link.ctl_tx_H == "HT_RX_CTL_H";
interface.ht_link.ctl_tx_L == "HT_RX_CTL_L";
for each in interface.ht_link.cad_tx_H {
  it == appendf("HT_RX_CAD%d_H", index);
};
for each in interface.ht_link.cad_tx_L {
  it == appendf("HT_RX_CAD%d_L", index);
};
```

Define the signals for the DUT's transmit path (The 'e' device's RX path):

```
for each in interface.ht_link.clk_rx_H {
  it == appendf("HT_TX_CLK_H");
};
for each in interface.ht_link.clk_rx_L {
  it == appendf("HT_TX_CLK_L");
};
interface.ht_link.ctl_rx_H == "HT_TX_CTL_H";
interface.ht_link.ctl_rx_L == "HT_TX_CTL_L";
for each in interface.ht_link.cad_rx_H {
  it == appendf("HT_TX_CAD%d_H", index);
};
for each in interface.ht_link.cad_rx_L {
  it == appendf("HT_TX_CAD%d_L", index);
};
```

Defining a Tunnel IO Device

Instantiate a basic Tunnel IO device:

```
example_tunnel : ht_device is instance;
keep example_tunnel.device_function == Tunnel_IO_Device;
```

Set up the HDL path to the interface point for this device:

```
keep example_tunnel.hdl_path() == "wasabi_tb";
```

Connect system wide Power and Reset signal:

```
keep example_tunnel.Reset_L == "HT_RESET_NEG";
keep example_tunnel.PowerOk == "HT_POWEROK";
```

Define the signal interface for the link: A tunnel has two interfaces so the definition of signals is more complicated. The preface of "index == 0 =>" implies that the first interface should have the specified value and the preface of "index == 1 =>" that the second interface should have the specified value:

```
keep for each (interface) using index (interface_index) in
example_tunnel.ht_interfaces
```

Define the basic clock for the link interface:

```
interface.ht_link.ht_link_fundamental_clock_signal == appendf("HT_CLK");
```

Define the signals for the DUT's receive path (The 'e' device's TX path):

```
for each in interface.ht_link.clk_tx_H {
  interface_index == 0 => it == appendf("HT_0_RX_CLK_H");
  interface_index == 1 => it == appendf("HT_1_RX_CLK_H");
};
for each in interface.ht_link.clk_tx_L {
  interface_index == 0 => it == appendf("HT_0_RX_CLK_L");
  interface_index == 1 => it == appendf("HT_1_RX_CLK_L");
};
interface_index == 0 => interface.ht_link.ctl_tx_H == "HT_0_RX_CTL_H";
```

```
interface_index == 1 => interface.ht_link.ctl_tx_H == "HT_1_RX_CTL_H";
interface_index == 0 => interface.ht_link.ctl_tx_L == "HT_0_RX_CTL_L";
interface_index == 1 => interface.ht_link.ctl_tx_L == "HT_1_RX_CTL_L";
for each in interface.ht_link.cad_tx_H {
  interface_index == 0 => it == appendf("HT_0_RX_CAD%d_H", index);
  interface_index == 1 => it == appendf("HT_1_RX_CAD%d_H", index);
};
for each in interface.ht_link.cad_tx_L {
  interface_index == 0 => it == appendf("HT_0_RX_CAD%d_L", index);
  interface_index == 1 => it == appendf("HT_1_RX_CAD%d_L", index);
};
```

Define the signals for the DUT's transmit path (The 'e' device's RX path):

```
for each in interface.ht_link.clk_rx_H {
  interface_index == 0 => it == appendf("HT_0_TX_CLK_H");
  interface_index == 1 => it == appendf("HT_1_TX_CLK_H");
};
for each in interface.ht_link.clk_rx_L {
  interface_index == 0 => it == appendf("HT_0_TX_CLK_L");
  interface_index == 1 => it == appendf("HT_1_TX_CLK_L");
};
interface_index == 0 => interface.ht_link.ctl_rx_H == "HT_0_TX_CTL_H";
interface_index == 1 => interface.ht_link.ctl_rx_H == "HT_1_TX_CTL_H";
interface_index == 0 => interface.ht_link.ctl_rx_L == "HT_0_TX_CTL_L";
interface_index == 1 => interface.ht_link.ctl_rx_L == "HT_1_TX_CTL_L";
for each in interface.ht_link.cad_rx_H {
  interface_index == 0 => it == appendf("HT_0_TX_CAD%d_H", index);
  interface_index == 1 => it == appendf("HT_1_TX_CAD%d_H", index);
};
for each in interface.ht_link.cad_rx_L {
  interface_index == 0 => it == appendf("HT_0_TX_CAD%d_L", index);
  interface_index == 1 => it == appendf("HT_1_TX_CAD%d_L", index);
};
```

Defining an EOC IO Device

Instantiate a basic EOC IO device:

```
example_EOC : ht_device is instance;
keep example_EOC.device_function == EOC_IO_Device;
```

Set up the HDL path to the interface point for this device:

```
keep example_EOC.hdl_path() == "testbench";
```

Connect system wide Power and Reset signals:

```
keep example_EOC.Reset_L == "HT_RESET_NEG";
keep example_EOC.PowerOk == "HT_POWEROK";
```

Define the signal interface for the link::

```
keep for each (interface) in example_EOC.ht_interfaces {
```

Define the basic clock for the link interface:

```
interface.ht_link.ht_link_fundamental_clock_signal == appendf("HT_CLK");
```

Define the signals for the DUT's receive path (The 'e' device's TX path):

```
for each in interface.ht_link.clk_tx_H {
  it == appendf("HT_RX_CLK_H");
```

```
};
for each in interface.ht_link.clk_tx_L {
  it == appendf("HT_RX_CLK_L");
};
interface.ht_link.ctl_tx_H == "HT_RX_CTL_H";
interface.ht_link.ctl_tx_L == "HT_RX_CTL_L";
for each in interface.ht_link.cad_tx_H {
  it == appendf("HT_RX_CAD%d_H", index);
};
for each in interface.ht_link.cad_tx_L {
  it == appendf("HT_RX_CAD%d_L", index);
};
```

Define the signals for the DUT's transmit path (The 'e' device's RX path):

```
for each in interface.ht_link.clk_rx_H {
  it == appendf("HT_TX_CLK_H");
};
for each in interface.ht_link.clk_rx_L {
  it == appendf("HT_TX_CLK_L");
};
interface.ht_link.ctl_rx_H == "HT_TX_CTL_H";
interface.ht_link.ctl_rx_L == "HT_TX_CTL_L";
for each in interface.ht_link.cad_rx_H {
  it == appendf("HT_TX_CAD%d_H", index);
};
for each in interface.ht_link.cad_rx_L {
  it == appendf("HT_TX_CAD%d_L", index);
};
```

General parameters to the global environment

In order to define the directed traffic for a Specman device, a new scenario must be created. First define a name for the scenario in the ht_scenario_kind type. Then extend the ht_scenario object to create a new when subtype that defines the generate_transactions method. After that is done, extend the ht_address_region type to create a target memory region and extend the ht_address_map object to define the region. Use the schedule_packet() method to queue the request and use the directed methods to generate the request. The following is an example for generating a sequence of 4 packets.

```
extend ht_address_region : [USER_REGION];
extend USER_REGION'region ht_address_map {
keep soft base_addr == 40'h00_1000_0000;
keep top_addr == 40'h00_1100_0000;
};

extend ht_scenario_kind : [USER_SCENE];
extend USER_SCENE'kind ht_scenario {
keep addr_map.region == USER_REGION;
```

Don't self generate target addresses

```
keep target_address_gen_complete == TRUE;

generate_transactions()@parent_device.ht_device_clock is {
    var delay : uint (bits:4);
    gen delay;
    wait [delay] * cycle;
    schedule_transaction(parent_device.gen_directed_write(DoubleWord_Write,
    NonPosted, NoPassPW, 4'b1000,  %{32'hCAFE}, 40'h1111100000));
    gen delay;
    wait [delay] * cycle;
    schedule_transaction(parent_device.gen_directed_write(Byte_Write, NonPosted,
    NoPassPW, 4'b1001, %{32'hDEAD, 32'h0003}, 40'h2222200000));
    gen delay;
    wait [delay] * cycle;
```

```
    schedule_transaction(parent_device.gen_directed_read(DoubleWord_Read,
    NoPassPW, PassPW, 4'b1000, 4'b0000, 40'h1111100000));
    gen delay;
    wait [delay] * cycle;
    schedule_transaction(parent_device.gen_directed_read(Byte_Read, PassPW,
    NoPassPW, 4'b1001, 4'b1111, 40'h2222200000));
    done = TRUE;
};
};
```

Host Bridge Device Parameters:

Max_memory_latency: Defines the maximum memory latency when writing or reading to the devices memory.Max_memory_latency can be configured to be between 0x0 and 0xffffffff inclusive. The default latency is 10 cycles.

Max_io_delay: Defines the maximum latency I/O when writing or reading to the device's PCI I/O region. Valid Max_io_delay range from 0 to 0xfffffffff. However, the currently the HT model does not support accesses to the PCI I/O space. The default delay is 10 cycles.

Max_tunneling_delay: Defines the maximum number of cycles to wait before forwarding packets that are not destined for the Host Bridge. Max_tunneling_delay can be set between 0 and 0xffffffff. The default setting is10 cycles.

ht_interfaces: This is a list of links or interfaces associated with a device. In the case of the Host Bridge, it only has 1 ht_interface.

generate_bad_CRCs: This is a boolean flag to define whether random CRC error should be injected into the system. The default setting is FALSE.

The following are virtual channel queue weights that control the priority and selection frequency of the scheduled packets to be transmitted based on its type. All VC weights must be positive and the default weight is 1. Along with the VC_queue_weight, the schedule_timing_weight (for control packets) and data_scheduling_weight for data packets) are also factored into the priority and selection process. The final weighting for a packet is calculated as follows:

for control packets:
packet_weight = (VC_queue_weight + schedule_timing_weight*(max_num_queable_items - index));
where index is the index where the packet is in the scheduling queue

for data packets:
packet_weight = VC_queue_weight + data_scheduling weight.

Note, the VC_queue_weight represents either the Posted_VC_queue_weight, onPosted_VC_queue_weight or Repsonse_VC_queue_weight depending on the packet's virtual channel type.

For control packets, the scheduling_timing_index affects the probability that items earlier in the queue will be serviced earlier.

Posted_VC_queue_weight: The weight that is factored into a Posted packet's weight. The default setting is 1 and the weight must be valid.

NonPosted_VC_queue_weight: The weight that is factored into a NonPosted packet's weight. The default setting is 1 and the weight must be valid.

Response_VC_queue_weight: The weight that is factored into a Response packet's weight. The default setting is 1 and the weight must be valid.

scheduling_timing_weight: This represents the priority of packets earlier in the tc_scheduling_queue to be selected for transmission. The default weight is 1. The scheduling_timing_weight can be from 0 to 0xffffffff.

data_scheduling_weight: The default weight is 1. The scheduling_timing_weight can be from 0 to 0xffffffff.

The following are the set of knobs that allow the user to determine the exact ordering requirements of the HT IO traffic. They are generated from the ordering section 7.1 from the 1.0 HT specification: The default setting allow for the most aggressive rescheduling of packets.

Posted_Request_with_PassPW_pass_Posted_Request: A boolean flag allowing posted request packets with the PassPW bit set to pass posted request packets. Default setting is TRUE.

NonPosted_Request_with_PassPW_pass_Posted_Request: A boolean flag enabling whether a non-posted request packet with the PassPW bit set to be transmitted before a posted request packet. Default setting is TRUE.

NonPosted_Request_pass_NonPosted_Request: A boolean flag enabling whether a non-posted request packet can be sent before a non-posted request. Default setting is TRUE.

NonPosted_Request_pass_Response: A boolean flag defining whether a non-posted request packet can be transmitted before a response. Default setting is TRUE.

Response_with_PassPW_pass_Posted_Request: A boolean flag defining whether a response packet with the PassPW bit set can be transmitted before a posted request. Default setting is TRUE.

Response_pass_Response: A boolean flag enabling response packets to pass other response packets. Default setting is TRUE.

create_target_starvation: A boolean flag to control whether the receive link should be starved or not. The default value is FALSE.

starvation_buffer_release_percentage: The default value is 10 and valid configurations range between 0 and 100 inclusive.

starvation_cycle_count: Defines the number of cycles to wait before releasing buffers once the receive link is starved. Currently the default and only valid configuration for this parameter is 0.

nop_bandwidth_percent: Defines the percentage of bandwidth consumed by Nop packets. Valid configurations are 1 to 99% and the default setting is 25%.

nop_maximum_buffer_release: Represents a boolean flag whether the Nop packet indicates that the maximum number of buffers is available in the sender. The default setting is TRUE. If nop_maximum_buffer_release is FALSE, a random number of buffers will be released.

interrupt_tx_packet_percentage: Defines the probability of a packet that is being transmitted to be interrupted on a given 4-byte window. The default value is 25% and the valid ranges are 0 to 100.

Debugging Flags

Link Level Flags and Channel Level Flags can be customized on a per device basis, but that is considered to be an advanced feature as it is more difficult to do.

Link Level Flags

These are the flags to control what information to be printed on the screen.

debug_link: Flag to print link level information to the user interface.

debug_crc: Flag to control printing of CRC information to the user interface.

debug_initialization: Flag to control printing initialization information to the user interface.

record_link_traffic : Flag to control the link interface to record every bit time that is transmitted and received.

Channel Level Flags

display_channel_messages: Flag to control massage printing when a non-NOP packet is transmitted or received.

debug_channel: Flag to control massage printing when a packet other than an empty Nop is transmitted or received.

tx_scheduling_queue_watermark: Flag to monitor the level of the tx_scheduling_queue. If the watermark level is exceeded print a message.

The following are the reset sequencing flags:

initiate_sync_sequence_delay: Number of cycles to wait after RESET is deasserted before initiating the SYNC SEQUENCE.

sync_sequence_length: Number of cycles for the sync sequence minimu m number of cycles

Device Level Flags

record_traffic: Flag to control recording of traffic at the channel level

record_all_nop: Flag to control recording of nop with no information. However, if that information

development_test : Extending the system architecture for Standalone Mode of operation, remove this for standard DUT.

Defining Specman Traffic

Defining new address regions for testing

Address Region	Device Type	Base Address	Top Address
test1'region ht_address_map	Host Bridge	40'h00_0000_0000	40'h00_0000_003f
test2'region ht_address_map	Host Bridge	40'h00_2000_0000	40'h00_2000_003f
test3'region ht_address_map	Tunnel Device	40'h00_4000_0000	40'h00_4000_003f
test4'region ht_address_map	EOC	40'h00_6000_0000	40'h00_6000_003f
test4'region ht_address_map	EOC	40'h00_6001_0000	40'h00_6001_003f

Base address configuration

The target_memory_map is a structure used for a device when it is generating random traffic. This memory map enables the device to know what memory regions it can access. This field is only defined for Tunnel and EOC devices as the Host will generate it's own target_memory_map based on chain sizing.

INIT_REGION'region ht_address_map : base_addr 40'h00_0000_0000;

Methods

The following are the methods used for users to create directed HT requests. They allow a user to specify the major fields of the packet without worrying about the guts of an HT request. Each method will return a newly created packet.

schedule_transaction(packet : ht_packet)

This method will take a packet that is to be transmitted and place it in the channels scheduling queue. In addition, it will also maintain the appropriate scoreboards and counters for the transmit path. This method should be the only means of accessing the channel on the transmit side.

gen_directed_flush(passing_rule : ht_passing_rules, SeqID : uint (bits:4))

passing Rule: Dictates if the PassPW bit will be set. NoPassPW or PassPW are the possible choices.
SeqID: The sequence number to be used for this request.

gen_directed_write(data_type : ht_packet_encoding, vc : ht_virtual_channel, passing_rule : ht_passing_rules, SeqID : uint (bits:4), data : list of bit, addr : uint (bits:40))

data_type: Specifies if the request is to be a DoubleWord_Write or Byte_Write. Double Word or Byte is the choices.
vc: Selects either the Posted or NonPosted virtual channels. Posted or NonPosted are the choices.
passing_rule: Dictates if the PassPW bit will be set. NoPassPW or PassPW are the possible choices.
SeqID: The sequence number to be used for this request.
data: The data to be written.
addr: The 40 bit address

gen_directed_read(data_type : ht_packet_encoding, resp_passing_rule : ht_passing_rules, passing_rule : ht_passing_rules, SeqID : uint (bits:4), data_size : uint (bits:4), addr : uint (bits:40))

data_type: Specifies if the request is to be a DoubleWord_Read or Byte_Read. Double Word or Byte is the choices.
resp_passing_rule: Dictates if the PassPW bit will be set for the response. NoPassPW or PassPW are the possible choices.
passing_rule: Dictates if the PassPW bit will be set. NoPassPW or PassPW are the possible choices.
SeqID: The sequence number to be used for this request.
data_size: Either the count field for Double Word request or the byte mask field for Byte requests.
addr: The 40 bit address

gen_directed_broadcast(vc : ht_virtual_channel, passing_rule : ht_passing_rules, SeqID : uint (bits:4), addr : uint (bits:40))

vc: Selects either the Posted or NonPosted virtual channels. Posted or NonPosted are the choices.
passing_rule: Dictates if the PassPW bit will be set. NoPassPW or PassPW are the possible choices.
SeqID: The sequence number to be used for this request.
addr: The 40 bit address

gen_directed_fence(passing_rule : ht_passing_rules, SeqID : uint (bits:4))

passing_rule: Dictates if the PassPW bit will be set. NoPassPW or PassPW are the possible choices.
SeqID: The sequence number to be used for this request.

Pre-Configuration Example for a Specman HT Device

Preloading the CSR's for each device :

```
extend ht_device_control_block {
pre_load_csr() is only {
if(device_function == Host_Bridge) {
// Base_Address_Register[0]
device_header.write(6'h10, 4'hF, 32'h00000000);
// Base_Address_Register[1]
device_header.write(6'h14, 4'hF, 32'h00000000);
// Base_Address_Register[2]
device_header.write(6'h18, 4'hF, 32'h40000000);
// Base_Address_Register[3]
device_header.write(6'h1C, 4'hF, 32'h00000000);
// Base_Address_Register[4]
device_header.write(6'h20, 4'hF, 32'h80000000);
// Base_Address_Register[5]
device_header.write(6'h24, 4'hF, 32'h00000000);
};

if(device_function == Tunnel_IO_Device) {
// Base_Address_Register[0]
device_header.write(6'h10, 4'hF, 32'hC0000000);
// Base_Address_Register[1]
device_header.write(6'h14, 4'hF, 32'h00000000);
// Base_Address_Register[2]
device_header.write(6'h18, 4'hF, 32'h00000000);
// Base_Address_Register[3]
device_header.write(6'h1C, 4'hF, 32'h00000001);
// Base_Address_Register[4]
device_header.write(6'h20, 4'hF, 32'h40000000);
// Base_Address_Register[5]
device_header.write(6'h24, 4'hF, 32'h00000001);

// Capability Command Register set the UnitID to 1
capability_register.write(6'h00, 4'hF, 32'h00010000);
};

if(device_function == EOC_IO_Device) {
// Base_Address_Register[0]
device_header.write(6'h10, 4'hF, 32'h80000000);
// Base_Address_Register[1]
device_header.write(6'h14, 4'hF, 32'h00000001);
// Base_Address_Register[2]
device_header.write(6'h18, 4'hF, 32'hC0000000);
// Base_Address_Register[3]
device_header.write(6'h1C, 4'hF, 32'h00000001);
// Base_Address_Register[4]
device_header.write(6'h20, 4'hF, 32'h00000000);
// Base_Address_Register[5]
device_header.write(6'h24, 4'hF, 32'h00000002);

// Capability Command Register set the UnitID to 2
capability_register.write(6'h00, 4'hF, 32'h00020000);
};
};
};
```

Built-In Scenario Definitions

The HT model includes 9 different kinds of pre-defined scenarios:

1. **INTERMIX**: Scenario that performs random read/write/flush/fence/broadcast traffic to a targeted address range. The transactions are unique in that they target no overlapping regions of memory.

2. **INTERMIX_OVERLAY:** Scenario that performs random read/write/flush/fence/broadcast traffic to a targeted address range. The transactions are non unique in that they target overlapping regions of memory.

3. **WRITE_IN_ORDER**: Scenario that performs in order write request of random size to a targeted_address range.

4. **READ_IN_ORDER**: Scenario that performs in order read requeste to a targeted_address range.

5. **READ_WRITE_IN_ORDER:** Scenario that performs in order and alternating write and read request of random size to a targeted_address range

6. **INTERRUPT:** Scenario that performs in interrupts targeted towards the Host Bridge.

7. **READ_POR_REGS:** Scenario that reads and tests POR values of all registers for a HT device in the chain from the Host Bridge.

8. **WALK_ONES_REGS:** Scenario that writes a '1' to every bit in all the registers for a HT device in the chain from the Host Bridge. Only bits that are not masked off in the ht_basic_reg func_write_mask will be written. Write will be followed by a read to the register as a check.

9. **INTERMIX_REGS:** Scenario that intermixes reads and writes to all the registers for a HT device in the chain from the Host Bridge. Write data is randomly generated and then ended with the func_write_mask.

Further detailed information can be found in the “ht_scenerio_definitions.e” file in the installation directory.

Included Files

File Name	Description
ht_channel_interface.e	The channel represents the heart of the HT protocol. It manages packets and queuing. It schedules packets for transmission and reassembles the packets received by the link interface. Also, it allows for packet reordering.
ht_common.e	Contains random structs that are used throughout the model.
ht_core.e	This file imports the core files. It provides a good separation of the model allowing the core to be compiled while maintaining the flexibility of easy user customization.
ht_coverage.e	User defined coverage file
ht_declarations.e	This file contains all defines and enumerated type definitions.
ht_device.e	This file creates a foundation for an HT device
ht_device_control.e	This file contains all of the control information needed for a device.
ht_host_bridge.e	This file contains the definition for a Host bridge device.
ht_io_device.e	The file defines the behavior of the IO, tunnel and EOC devices.
ht_link.e	This file contains the necessary set up to run the model in Standalone Mode
ht_link_interface.e	This file contains the description of the most primitive nature of the device. The link interface handles the HT protocol at the bit time level. It also maintains and checks the TX/RX CRCs.
ht_packet_definition.e	This file contains the description of a Hyper Transport packet.
ht_sample_test.e	This file contains the example definitions for a test case. It will determine what traffic to be generated and the set all of the runtime flags in the 'e' model. This file will import ht_sample_bench.e and ht_top_control.e
ht_sample_bench.e	This file contains the example of a test bench. It will define the different instantiations of the 'e' model and provide the wires/structures used to connect the devices. It can be viewed as defining the system topology.
ht_scenario_definitions.e	This file contains the definition of the traffic generation of the model ht_host_bridge.e and the behavior of the host bridge device. It also provides the definition of a basic chain initialization sequence.
ht_top.e	This file imports the core definition files.
HT_CHANGE_LOG	This file contains the code revision log for the Hyper Transport model.