USB/FireWire Simulation

- The goal of this simulation is to write a VHDL model for a system that is somewhat similar in nature to the new serial standards USB (Universal Serial Bus) and IEEE Firewire
- Simulation aspects
  - Totally event driven – no global clock
  - Communication is bidirectional, half-duplex over a single wire
  - Data type is a resolved data type using a record structure
  - Network structure is a tree structure that consists of a root node, hubs, and endpoints

Universal Serial Bus

- Universal Serial Bus is a new synchronous serial protocol for low to medium speed data transmission
- Full speed signaling 12 Mbs
- Low Speed signaling 1.5 Mbs
- Intended devices are keyboards, mice, joysticks, speakers; other low to medium speed IO devices

<table>
<thead>
<tr>
<th>PERFORMANCE</th>
<th>APPLICATIONS</th>
<th>ATTRIBUTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOW SPEED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interactive Devices</td>
<td>Keyboard, Mouse</td>
<td>Lower cost, Hot plug-unplug, Ease of use</td>
</tr>
<tr>
<td></td>
<td>Stylus</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Game peripherals</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Virtual Reality peripherals</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Monitor Configuration</td>
<td></td>
</tr>
<tr>
<td>MEDIUM SPEED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Phone, Audio, Compressed Video</td>
<td>P3</td>
<td>Low cost, Ease of use, Guaranteed Bandwidth, Dynamic Attach-Detach</td>
</tr>
<tr>
<td>500Kbs - 1Mbps</td>
<td>PDS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Audio</td>
<td></td>
</tr>
<tr>
<td>HIGH SPEED</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Video, Disk</td>
<td>Disk</td>
<td>High Bandwidth, Guaranteed latency, Ease of use</td>
</tr>
<tr>
<td>0.5-500 Mbps</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3-1. Application Space Taxonomy
Physical Topology is point-to-point tree.

Root: primary controller
Hub: allows the connection of multiple USB devices
Endpoint: Source or sink of information within a USB device

A USB device that contains an ENDPOINT (source/sink of data) is called a 'function'.

A USB device can be just a function, just a hub, or both a hub and a function.

Physical connection is point to point.
Physical Interface

Differential Signaling, Half duplex

Full Duplex: data transmission can occur in both directions at the same time

Half Duplex: data transmission can go in only one direction at a time

Table 7-1. Signaling Levels

<table>
<thead>
<tr>
<th>Bus State</th>
<th>Signaling Levels</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>From Originating Driver</td>
</tr>
<tr>
<td>Differential +1</td>
<td></td>
</tr>
<tr>
<td>Differential -1</td>
<td></td>
</tr>
<tr>
<td>Differential +0</td>
<td></td>
</tr>
<tr>
<td>Differential -0</td>
<td></td>
</tr>
</tbody>
</table>

Input Levels:

- Differential Input Sensitivity: 
  - | 0.2 V
  - Inclusive VIL range: 0.8 V
  - Single-Ended Receiver Threshold: 0.3 V

Output Levels:

- Static Output Low: 
  - RL of 1.5 kΩ to 3.3 V
  - 0.3 V
- Static Output High: 
  - RL of 15 kΩ to GND
  - 2.5 V

Vse = Voltage Single Ended threshold
On disconnect, D+, D- become same voltage value (Vss). Condition is known as a Single-Ended 0.

On connection of a high-speed device, D+ > D-. Idle state is D+ > D-, so idle state is a differential ‘1’.

On connection of a low-speed device, D- > D+. Idle state is D- > D+, so idle state is a differential ‘0’.
Idle State: D+, D- outside of range of VSE, at either a differential '1' (high speed) or '0' (low speed).
Active State: D+, D- transition to signal a 1 or 0

Why differential signaling??

Differential signaling very good at rejecting common-mode noise. If noise is coupled into a cable, then usually it is coupled into all wires in the cable. This 'common-mode' noise (Vcm) can be rejected by input amplifier.

\[ V_o = (D+) - (D-) \]

\[ V_o = (V_{cm} + D+) - (V_{cm} + D-) = (D+) - (D-) \]
Non-return to zero (NRZ) - normal data transitions.

NRZ – Inverted (not a good description, is not inverse of NRZ). A transition for every zero bit.

Strings of zeros means lots of transitions. Strings of ‘1’s means steady line.

Bit Stuffing – a ‘0’ is inserted after every six consecutive ‘1’s in order to ensure a signal transition so that receiver clock can remain synchronized to the bit stream.

Data Encoding Sequence:

Bit stuffing done automatically by sending logic. Sync pattern starts data transmission and is seven ‘0’s followed by a ‘1’.

Receiver/Xmitter logic uses a 48 Mhz internal local clock.

48Mhz/12Mbs = 4 clocks per bit time for high speed signaling.

48Mhz/1.5 Mbs = 32 clocks per bit time for low speed signaling.

A guaranteed transition every 7 bit times allows local clock synchronization to the serial data stream. Sync pattern allows clock sync at beginning of packet.
Data Formatting

- Data sent in packets
- Packets will have:
  - Start of Packet Sync Pattern (8 bits, 7 zeros + 1 one)
  - Packet ID (PID) – identifies type of packet. 8 bits total, but only 4 unique bits
  - Address field - 11 bits. 7 bits for USB device (so 128 possible USB devices on bus, host is always address 0), 4 bits for internal use by USB device
  - Frame number field (11 bits) – incremented by host
  - Data Payload (up to 1023 bytes for high-speed connection)
  - CRC bits - 5 bits for address field, and 16 bits for data field
  - EOP strobe – single ended 0 (160ns-175 ns for high-speed, 1.25 us to 1.75 us for high-speed)
- Not all packets sent over USB bus have all of these fields (always have SOP, EOP and PID). Packet without data field is a token packet.

Packet Types

Table 8-1. PID Types

<table>
<thead>
<tr>
<th>PID Type</th>
<th>PID Name</th>
<th>PID(16)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Token</td>
<td>OUT</td>
<td>00001</td>
<td>Address + endpoint number in host = function transaction</td>
</tr>
<tr>
<td>IN</td>
<td>01001</td>
<td></td>
<td>Address + endpoint number in function = host transaction</td>
</tr>
<tr>
<td>SETUP</td>
<td>01101</td>
<td></td>
<td>Start of frame marker and frame number</td>
</tr>
<tr>
<td>DATA1</td>
<td>00011</td>
<td></td>
<td>Data packet PID even</td>
</tr>
<tr>
<td>DATA2</td>
<td>01011</td>
<td></td>
<td>Data packet PID odd</td>
</tr>
<tr>
<td>Handshake</td>
<td>ACK</td>
<td>00010</td>
<td>Receiver accepts error free data packet</td>
</tr>
<tr>
<td>NAK</td>
<td>01010</td>
<td></td>
<td>Rx device cannot accept data or Tx device cannot send bits</td>
</tr>
<tr>
<td>STALL</td>
<td>01110</td>
<td></td>
<td>Endpoint is stalled</td>
</tr>
<tr>
<td>Special</td>
<td>PRE</td>
<td>01100</td>
<td>Host requests priority. Enables downstream bus traffic to low speed devices.</td>
</tr>
</tbody>
</table>

Figure 8-9: Bulk Transaction Format
VHDL Model

- Our VHDL Model will present a VERY abstract view of a USB network
- Models:
  - root.vhd -- models the root node
  - hub.vhd -- models a hub, has an upstream port and 2 downstream ports
  - endpoint.vhd -- models an endpoint
- Data packet will contain a packet ID, an address (destination), and a payload
  - Payload is an 80 character string
  - Packet types of POUT, PIN, PACK, ERR, NONE

---

root.vhd

```vhdl
library ieee;
use ieee.std_logic_1164.all;
library work;
use work.usbpkg.all;

entity root is
  generic ( ADDR : natural := 0 );
  port ( signal dport : inout pkt );
end root;
```

address of root is always ‘0’.

Only one port on root.

---

hub.vhd

```vhdl
library ieee;
use ieee.std_logic_1164.all;
library work;
use work.usbpkg.all;

entity hub is
  generic ( HUBDELAY : time := 5 ns );
  port ( signal downstrm_a,downstrm_b : inout pkt ;
         signal upstrm   : inout pkt );
end hub;
```

One upstream port, two downstream ports
endpoint.vhd

library ieee;
use ieee.std_logic_1164.all;
library work;
use work.usbpkg.all;

entity endpoint is
  generic (MANUF : string := "FooBar Enterprises";
           ADDR : natural := 0);
  port (dport : inout pkt);
end endpoint;

Data stored at endpoint

Address of endpoint, cannot be 0.

usbpkg.vhd

PACKAGE usbpkg IS
  constant PTIME: time :=  1 us;  -- packet time
  constant RTIME: time :=  70 ns; -- turn around time
  constant MAXENDPT: natural := 32;  -- maximum # of endpoints
  type ptype is (NONE, POUT, PIN, PACK, ERR);
  type upkt is    RECORD
    id: ptype;
    dest : integer ;
    data: string(1 to 80);
  END RECORD;
  type upkt_vector is array (natural range <>) of upkt;
  function resolve_upkt (s : upkt_vector) return upkt;
  subtype pkt is resolve_upkt upkt;
END usbpkg;

Packet type
Packet destination
Packet payload
Resolved data type

Protocol
- Root initiates all transactions
- Root will either send a PIN or POUT packet with an destination (address) field set
  - At endpoint, if destination field matches endpoint address, process packet else ignore packet
  - if POUT packet, endpoint responds with ACK packet and places local data (initially set to MANUF string) in ACK packet
  - if PIN packet, endpoint copies packet payload ('data' field) into local data, and responds with ACK packet – the data field of this ACK packet is a don’t care
Releasing the Line

- To simulate ‘releasing’ the line, after either a POUT, PIN, or PACK packet is sent, send a packet of type NONE
- A signal between a hub and an endpoint/root will only have 2 drivers
  - To resolve the two drivers, look at the packet type
  - A packet type of NONE resolved with POUT/PIN/PACK will return POUT/PIN/PACK
  - A packet type of NONE resolved with NONE will return NONE
  - A packet type of POUT/PIN/PACK resolved with POUT/PIN/PACK will return a packet of type ERR (this should not happen – if it does, then you have a packet collision which should never happen).
- root_a.vhd illustrates how to send/receive packets

HUB operation

- On downstream ports, any packets of type PACK or NONE should be echoed to upstream port
  - PACK packet can only come from an endpoint
- On upstream port, any packet that is not a PACK packet should be echoed to both downstream ports
- Use HUBDELAY generic for delay time through hub

What do you have to do?

- Complete the resolution function for pkt data type
- Complete architectures for ENDPOINT and HUB
- Test your design with ‘tb_a.vhd’ and ‘tb_b’.vhd
  - I may test your code with other configurations!!!!
- The root_a.vhd code does the following:
  - Loops sending POUT packets to addresses 1 to 32. If a PACK is received, know that there is an ENDPOINT at that address
  - Prints out data from ACK packet to screen
  - Sends a PIN packet to the endpoint with the data from the PACK packet modified
  - Send a POUT packet to the endpoint to verify that the endpoint stored the new data - wait for the PACK response and print returned data to console
Sample Run with `tb_b.vhd`
```
> qhsim -lib usb tb_b -c -do "run 150 us;quit"
# Found Endpoint 3, data is Cypress Mouse
# Modifying Endpoint...
# Finished Scan for Endpoints
```