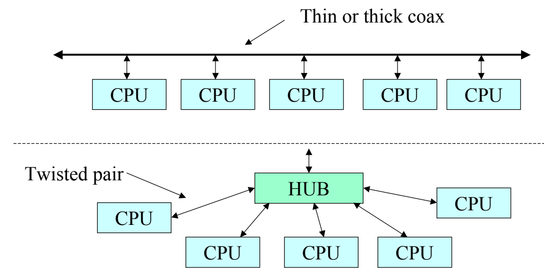
CSMA/CD
Carrier Sense Multiple Access/Collision Detect

- CSMA/CD is the bus protocol used by Ethernet
- Requires no central arbitration
- Single wire transmission medium
- Basic Protocol
  - CPU listens to the ‘wire’ and waits until it is quiet
  - Begins transmitting packet, listening while transmitting
  - If collision detected (garbage on wire), then stops transmitting
  - Wait some random time between 0 and backoff interval
  - Try again – if another collision, double size of backoff interval (up to some maximum) and try again

Ethernet Topology



Regardless of the physical topology, the logical model of a single wire holds.

Simulation Goal

- Our simulation goal will be to write a model that captures the basics of CSMA/CD arbitration
- Interesting from a VHDL viewpoint because
  - Need to use a resolved type to model the ‘wire’
  - No global clock signal – CPU model will respond to asynchronous events (generated both from within the model and externally)
- I will provide the framework of the simulation
- You will need to provide two things
  - CPU Model
  - Resolution function for the resolved type used to model the ‘wire’
- Our simulation will leave out some of the subtleties of the complete CSMD/CD arbitration model as defined in the ethernet standard, but will capture its essence

Modeling the ‘wire’

- Will model the single ‘wire’ as a resolved type called *epacket* - will be a record type with two fields
  - *drive_value* (std_logic) – resolved value of wire, used to check for collision
  - *sender* (integer) – CPU_ID of current driver of wire – only used for debugging
- Could have created other fields for packet contents, length etc. if trying do a more complete simulation
- When a CPU drives the wire, will set the drive_value field to a ‘0’
  - Resolution function should return a resolved *epacket* value that has drive_value = ‘0’ only if there is one driver with drive_value = ‘0’
  - Drive_value should be set to ‘X’ if more than one driver has drive_value = ‘0’
- CPU should set drive_value = ‘Z’ upon wire release

Packet Time, Slot Time

- Will use two constants that define time periods
  - SLOTtime – the length of time at the beginning of a transmission used to check for collisions
  - PKTtime – the time it takes to transmit a packet
- For 100 Mb ethernet, 1 SLOTtime = 512 bit times, will approximate as 5 ms
  - A SLOTtime is the minimum amount of time based on maximum cable length, and round trip wire delay that a collision can be reliably detected
- For 100 Mb ethernet, maximum packet size is 1500 bytes, so 1 PKTtime = 120 ms
- Will use CPU generic named *wait_interval* to control bus utilization
  - Will wait a random time between 0 and wait_interval*PKTtime seconds between packet sends

Transmitting a Packet

- Wait for a random amount of time between 0 and wait_interval * PKTtime (LOCAL state)
  - This represents the time that the CPU is not attempting a packet transmission
- (TRANSMIT_CHECK state) Wait for the wire to become free (drive.value = ‘Z’)
- Drive the wire (drive.value = ‘0’)
- Wait for SLOTtime, then check the wire for a collision
  - If a collision (drive.value = ‘X’) then go to collision state
  - If no collision (drive.value = ‘0’), then wait for PKTtime-SLOTtime (remainder of packet time), release wire (drive.value = ‘Z’), reset the *backoff* to 1, and go back to LOCAL state

Collision State

- If a collision has been detected, wait for a random time between 0 and *backoff* * SLOTtime
  - Initial value of *backoff* is 1
  - Note that wait time is based on SLOTtime, not PKTtime
- Double the backoff (backoff = backoff * 2)
  - Do not increase the maximum size of backoff past 1024
  - This is the maximum value in the ethernet standard.
  - The standard also specifies that if 16 consecutive collisions occurs, an error should be signaled from the ethernet interface and retransmission attempts ended (our model will not do this)
- After waiting, try transmitting again (go back to TRANSMIT_CHECK state)

cpu.vhd

- I have defined the entity for *cpu*, you will have write the architecture
- CPU ports are
  - ***active****: in std_logic* - should be a ‘0’ for entire period that CPU is active. Set to ‘Z’ when CPU is finished sending all packets
  - ***io*** *: inout epacket* - CPU’s connection to the wire
- CPU generics are
  - *wait_interval* - function discussed in previous slides
  - *cpu_id* - ID of this cpu, set by configuration
  - *packets_to_send* - this is the number of packets the CPU should send. After sending this number of packets, the CPU should set its *active* output ‘Z’, and suspend

etherpkg_.vhd, etherpkg.vhd

- These files contain the header and body declarations of the *etherpkg*.
  - Defines the *epacket* type
  - Defines various shared variables for statistics keeping
  - Defines the *do_report* procedure for statistics printing
- You only have to modify the *etherpkg.vhd* file and fill in the body of the *epacket* resolution function
  - The *drive_value* of epacket is the primary concern of the resolution function
  - Default value of *drive_value* should be ‘Z’
  - If more than one driver has a ‘0’ value, then *drive_value* should return ‘X’
  - If only one driver has a ‘0’ value, then *drive_value* should return ‘0’
  - Do what you want with the *sender* value of *epacket* - it is only useful for debugging.

Statistics

- *etherpkg* has several shared variable arrays for statistics
- Your *cpu* architecture should update the following shared variable arrays based on *cpu_id* :
  - *collisions* (integer array) – number of collisions seen by a CPU
  - *finish_time* (time array) – time when CPU suspends after last packet send
  - *latency_time* (time array) – total latency for a CPU. Latency is the amount of time before a successful packet transmission (the amount of time the CPU spends waiting because of collisions).
  - *packets_sent* (integer array) – total packets sent by a CPU
  - *backoff* (integer array) – maximum backoff reached by a CPU
- The *do_report* procedure is called by the *monitor* entity when the *active* signal becomes a ‘H’ (all CPUs are idle)
- The monitor entity also tracks bus utilization

cpu Modeling Approach

- Should use a FSM to model the CPU behavior, but transitions between states controlled by state changes and io events.
- Can use a single process with or without a sensitivity list.
- If using a sensitivity list, trigger the process based on *state* or *io* events.
- If not using a sensitivity list, control state transitions with wait statements.
- No inherent advantage to either method – use whatever method you understand the best.

State Transitions without sensitivity list

```
when S2 =>
        state <= S3 after some_delay;
        wait on state;
```

Specify a delay between state changes

or

```
when S2 =>
        state <= S3;
        wait on state;
```

No delay between state changes (a delta delay)

or

```
when S2 =>
        state <= S3;
        wait on io;
```

Change to another state based on an event from another signal

Typical Results

All times normalized to Packet Times!!
CPU #0 Finish Time: 1659, Packets: 100, Collisions: 28, Max Backoff Reached: 32, Total Latency: 1, LatencyPerPacket: 1.000000e-02
CPU #1 Finish Time: 1745, Packets: 100, Collisions: 35, Max Backoff Reached: 32, Total Latency: 1, LatencyPerPacket: 1.000000e-02
CPU #2 Finish Time: 1739, Packets: 100, Collisions: 36, Max Backoff Reached: 64, Total Latency: 2, LatencyPerPacket: 2.000000e-02
CPU #3 Finish Time: 1739, Packets: 100, Collisions: 37, Max Backoff Reached: 32, Total Latency: 2, LatencyPerPacket: 2.000000e-02
CPU #4 Finish Time: 1637, Packets: 100, Collisions: 40, Max Backoff Reached: 128, Total Latency: 5, LatencyPerPacket: 5.000000e-02
CPU #5 Finish Time: 1672, Packets: 100, Collisions: 34, Max Backoff Reached: 32, Total Latency: 1, LatencyPerPacket: 1.000000e-02
CPU #6 Finish Time: 1685, Packets: 100, Collisions: 23, Max Backoff Reached: 8, Total Latency: 0, LatencyPerPacket: 0.000000e+00
CPU #7 Finish Time: 1715, Packets: 100, Collisions: 49, Max Backoff Reached: 128, Total Latency: 3, LatencyPerPacket: 3.000000e-02
Wait Interval: 30, Bus Utilization: 44%, Total Packets: 800, Total Collisions: 282, AvgLatencyPerPacket: 1.875000e-02

Sanity Checking Report Output

- The *do_report* procedure normalizes all times to *PKTtime*
- CSMA is inherently fair – CPU finish times should be reasonably close to each other
  - If not, may be problem with the way you are calling the random number generator
- For high values of *wait_interval*, will not have many collisions
  - Finish times will be approximately wait_interval/2 * # of packets
- Latency values are small because backoff time based on SLOTtime, which is small compared to PKTtime
- Once backoff times saturate to 1024, bus utilization will fluctuate, results very dependent on random number generation

Simulation Requirements

- I have provide configurations for 1, 2 and 8 CPUs
  - Cfg_tb8.vhd, cfg_tb2.vhd, cfg_tb1.vhd
  - Use 1 CPU and 2 CPU configurations for debugging
  - The simulator resolution must be in 'ms' – edit the modelsim.ini file.
- The zip archive contains a file called *reese.ether.rawsol_cpu8* which is the output of my 8 CPU simulation for different values of *wait_interval*
  - The perl script *ether_sol.pl* can be used to run your simulation for these values.
  - You do not have to match my numbers exactly, simply have reasonable agreement
- I do not require any plots for this simulation or any answers
  - Obviously could use this simulation to answer questions similar to that in the 8 CPU + Arbiter simulation
  - I will simply run your model using the ether_sol.pl and look at the output

ZIP archive

- The zip archive unpacks a directory named 'ether.student'
- You should rename this directory to simply 'ether', and create a VHDL library named 'ether'
  - This is similar to the previous simulation
- The file *makefile.ether* in the *ether.student* directory is the makefile for the *ether* library
  - No modifications to this file should be needed