Open Source Implementation 2.5: CSMA/CD

CSMA/CD is part of the Ethernet MAC and most of the Ethernet MAC is implemented in hardware. An open source Ethernet example is available from OPENCORE (www.opencores.org), which consists of a synthesizable Verilog code. By synthesizable we mean the Verilog code is complete enough to be compiled, through a series of tools, into a circuit. It provides the implementation of the layer-2 protocol according to the IEEE specifications for the 10 Mbps and 100 Mbps Ethernet. Note that this open source implementation is the only hardware example in this text. All others are software.

Hardware Block Diagram

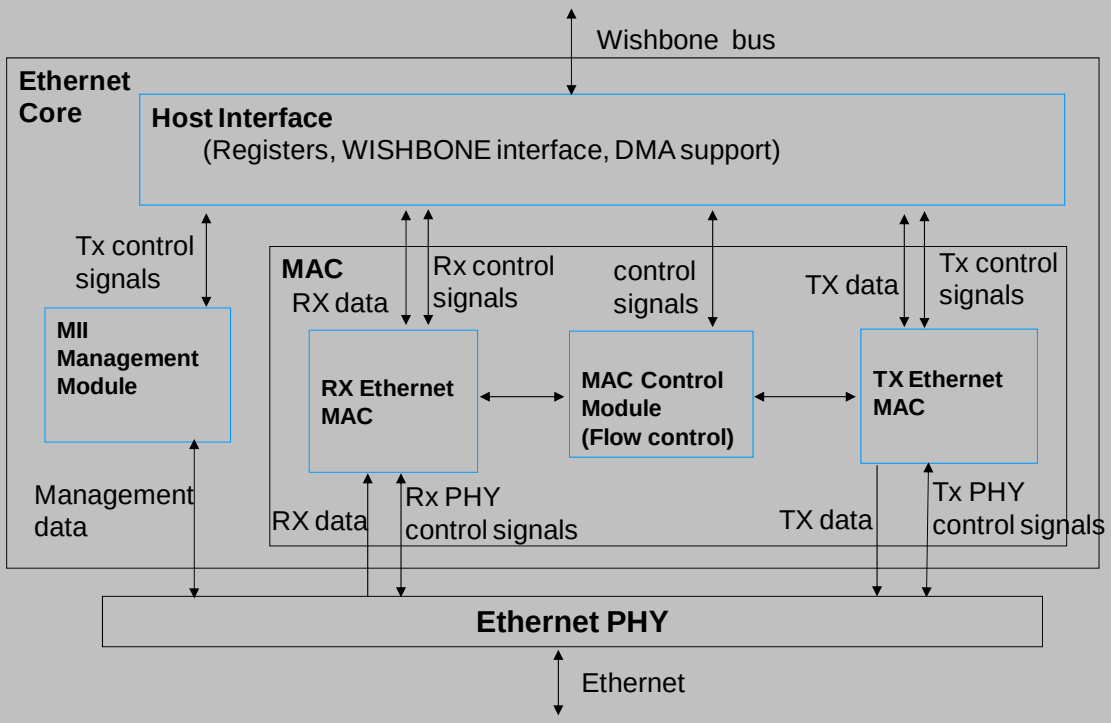


Figure 2.16 Architecture of Ethernet MAC core.

Figure 2.16 illustrates the architecture of OPENCORE Ethernet Core, which mainly consists of host interface, transmit (TX) module, receive (RX) module, MAC control module, and Media Independent Interface (MII) management module. They are described below.

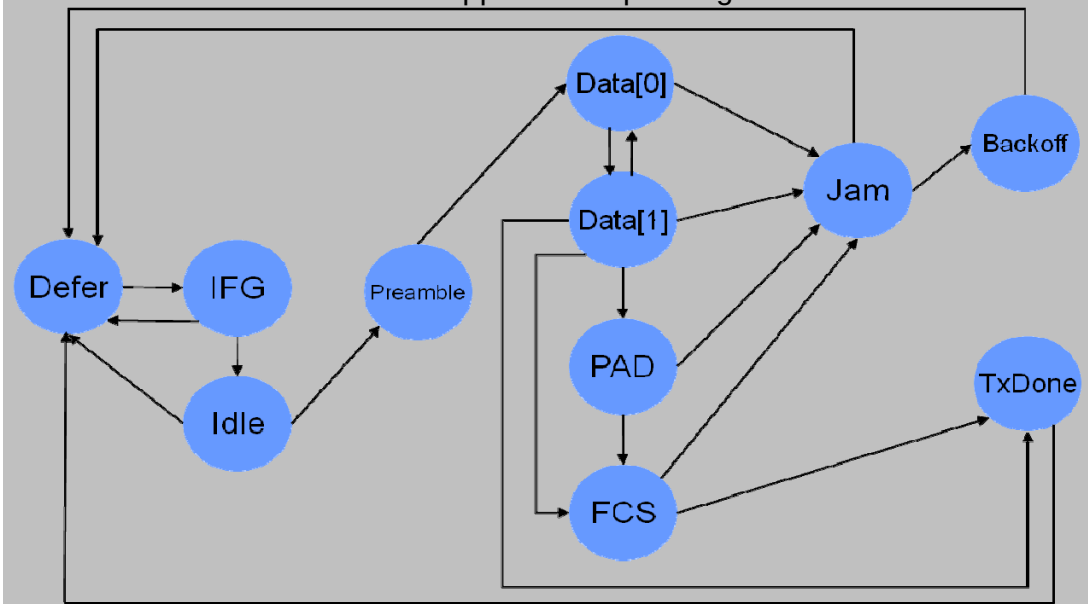
1. The TX and RX modules enable all transmit and receive functionalities. These modules handle preamble generation and removal. Both modules incorporate the CRC generators for error detection. In addition, the TX module has a random time generation used in the back-off process, and monitors the *CarrierSense* and *Collision* signals to exercise the main body of CSMA/CD.
2. The MAC control module provides full duplex flow control, which transfers the PAUSE control frames between the communicating stations. Therefore, the MAC Control Module has control frame detection and generation, interfaces to TX and RX MAC, PAUSE timer and Slot timer.
3. The MII management module implements the standard of IEEE 802.3 MII, which provides the interconnections between the Ethernet PHY and MAC layers. Through the MII interface, the processor can force Ethernet PHY to run at 10 Mbps or 100 Mbps, and configure it to perform at full or half duplex mode. The MII management module has the sub-modules for operation controller, shift registers, output control module and clock generator.
4. The host interface is a WISHBONE (WB) bus connecting the Ethernet MAC to the processor

and external memory. The WB is an interconnection specification of OPENCORE projects. Only DMA transfers are supported so far for data transferring. The host interface also has status and register modules. The status module records the statuses written to the related buffer descriptors. The register module is used for Ethernet MAC operations, and it includes configuration registers, DMA operation, and transmit and receive status.

State Machines: TX and RX

In the TX and RX modules, TX and RX state machines control their behaviors, respectively. Figure 2.17 presents both state machines. We only describe the behaviors of the TX state machine here, since the RX state machine works similarly. The TX state machine starts from the *Defer* state, which waits until the carrier is absent (i.e., the `CarrierSense` signal is false) and then enters the *IFG* state. After the Inter-frame Gap (IFG), the TX state machine enters the *Idle* state, waiting for a transmission request from the WB Interface. If there is still no carrier present, the state machine can go to the *Preamble* state to start a transmission; otherwise, it goes back to the *Defer* state and waits until the carrier is absent again. In the *Preamble* state, the preamble 0x5555555 and Start Frame Delimiter 0xd are sent, and the TX state machine goes to the *Data[0]* and *Data[1]* states to transmit from the Least Significant Byte (LSB) nibbles of the data byte, and then informs the Wishbone Interface to provide next data byte until the end of the packet.

- If a collision occurs during transmission, the TX state machine goes to the *Jam* state to send jam signal, waits for a period of backoff time in the *Backoff* state, and then goes back to the *Defer* state for the next transmission attempt.
- When only one byte is left to be sent (no collision during transmission),
  (1) if the total frame length is greater or equal to the minimum frame length, then TX state machine enters the *FCS* state to calculate the 32-bit CRC value from the data if CRC is enabled, appends the value to the end of the frame, and then goes to the *TxDone* state; otherwise the TX state machine directly goes to the *TxDone* state.
  (2) If the frame length is shorter than the minimum frame length and padding is enabled, then the TX state machine goes to the *PAD* state and the data is padded with zeros until the minimum frame length is achieved. The remaining states are the same as those in (1). However the *PAD* state is skipped when padding is disabled.



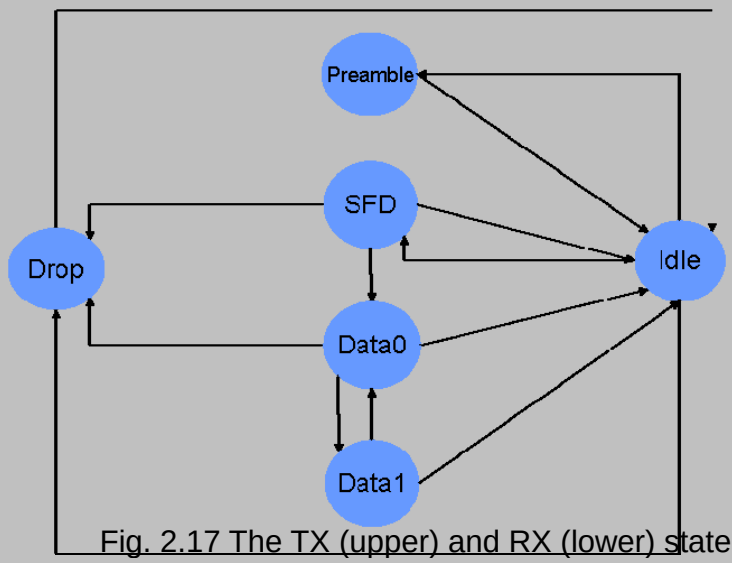


Fig. 2.17 The TX (upper) and RX (lower) state machines.

Programming CSMA/CD Signals and Nibble Transmission

Figure 2.18 is a segment of Verilog code that *programs* the key CSMA/CD signals and nibble transmission. An *output signal* is an arithmetic combination of various *input signals* and updated once in every *clock cycle*. All output signals are updated in *parallel*, which is the key difference with the *sequentially* executed software code. The symbols "~", "&", "|", "^" and "=" denote the operations of "not", "and", "or", "xor" and "assign", respectively. The conditional expression "exp ? exp : exp", has exactly the same meaning as that in the C language.

A station observes the activity on the PHY media in the half-duplex mode. Besides the carrier due to the transmission of a frame, a collision resulting from simultaneous transmission of two or more stations (denoted by the `Collision` variable) is also observed. If a collision occurs, all stations stop transmitting, set `StartJam` (entering the *`Jam`* state) and back off for a random time (`StartBackOff` is set) in the *`Backoff`* state. The state machine may go back to the *`Defer`* state if the carrier is present in the *`Jam`* state or the *`Backoff`* state.

The code in nibble transmission selects the nibble (4 bits) to be transmitted based on which state the TX machine is in. The TX state machine switches between the *`Data[0]`* and *`Data[1]`* states during little-endian transmission, so *`MTxD_d`*, i.e., the transmit data nibble, is loaded with *`TxData[3:0]`* and *`TxData[7:4]`* alternatively. In the *`FCS`* state, the CRC value is loaded nibble by nibble, as the CRC calculation is implemented with the *`Crc`* shift register. In the *`Jam`* state, the hex value of 1001 (i.e., 4'h9) is arbitrarily loaded as the jam signal, although the jam content is unspecified in the 802.3 standard. In the *`Preamble`* state, the preamble 0x5555555 and Start Frame Delimiter 0xd are loaded in turn.

```
CSMA/CD Signals
assign StartDefer = StateIFG & ~Rule1 & CarrierSense & NibCnt[6:0] <= IPGR1
& NibCnt[6:0] != IPGR2
| StateIdle & CarrierSense
| StateJam & NibCntEq7 & (NoBckof | RandomEq0 | ~ColWindow | RetryMax)
| StateBackOff & (TxUnderRun | RandomEqByteCnt)
| StartTxDone | TooBig;
assign StartDefer = StateIdle & ~TxStartFrm & CarrierSense
             | StateBackOff & (TxUnderRun | RandomEqByteCnt);
assign StartData[1] = ~Collision & StateData[0] & ~TxUnderRun &
               ~MaxFrame;
assign StartJam = (Collision | UnderRun) & ((StatePreamble & NibCntEq15)
             |(StateData[1:0]) | StatePAD | StateFCS);
assign StartBackoff = StateJam & ~RandomEq0 & ColWindow & ~RetryMax
    & NibCntEq7 & ~NoBckof;
Nibble transmission
always @ (StatePreamble or StateData or StateData or StateFCS or StateJam
or  StateSFD or TxData or  Crc or NibCnt or NibCntEq15)
begin
 if(StateData[0])   MTxD_d[3:0] = TxData[3:0];   // Lower nibble
 else  if(StateData[1])   MTxD_d[3:0] = TxData[7:4]; // Higher nibble
 else  if(StateFCS) MTxD_d[3:0]={~Crc[28],~Crc[29],~Crc[30],~Crc[31]};
```

```
    // Crc
else  if(StateJam)   MTxD_d[3:0] = 4'h9;        // Jam pattern
else  if(StatePreamble)
             if(NibCntEq15)     MTxD_d[3:0] = 4'hd;  // SFD
          else     MTxD_d[3:0] = 4'h5;           // Preamble
      else   MTxD_d[3:0] = 4'h0;
end
```

Figure 2.18 CSMA/CD Signals and nibble transmission

Since the TX module starts the back-off process after a collision has been detected, it waits for some duration derived from a pseudo random as shown in Figure 2.19. It applies the “binary exponential” algorithm to generate a random back-off time within the predefined restriction. An element `x[i]` in the array `x` is a random bit of either 0 or 1, and the Random array can be viewed as the binary representation of the random value (totally 10 bits, as the range of the random number is from 0 to 2^k-1, where $k = min(n, 10)$ and n is the number of re-trials.) According to each statement in Figure 2.19, when `RetryCnt` is larger than `i`, `Random[i]` may be set to 1 if `x[i] = 1`; otherwise `Random[i]` is set to 0 by assigning bit 0 (denoted by 1’b0) to it. In other

words, one more high-bit in the random values is likely to be set to 1, which means the range of the random values exponentially grows. After the random value is derived, it will be latched into the RandomLatched variable if the transmission channel is jammed (judged from the StateJam and StateJam_q variables), e.g., due to collision. If the random value happens to be 0 (i.e., backoff time is 0), the RandomEq0 variable is set and the backoff procedure will not be started (StartBackoff is false in the last assign statement of Figure 2.18).

```
assign Random [0] = x[0];
assign Random [1] = (RetryCnt > 1) ? x[1] : 1'b0;
assign Random [2] = (RetryCnt > 2) ? x[2] : 1'b0;
assign Random [3] = (RetryCnt > 3) ? x[3] : 1'b0;
assign Random [4] = (RetryCnt > 4) ? x[4] : 1'b0;
assign Random [5] = (RetryCnt > 5) ? x[5] : 1'b0;
assign Random [6] = (RetryCnt > 6) ? x[6] : 1'b0;
assign Random [7] = (RetryCnt > 7) ? x[7] : 1'b0;
assign Random [8] = (RetryCnt > 8) ? x[8] : 1'b0;
assign Random [9] = (RetryCnt > 9) ? x[9] : 1'b0;
always @ (posedge MTxClk or posedge Reset)
begin
  if(Reset)
    RandomLatched <= 10'h000;
  else
    begin
      if(StateJam & StateJam_q)
        RandomLatched <= Random;
    end
end
assign RandomEq0 = RandomLatched == 10'h0;
```

Figure 2.19 Back-off Random Generator