An Ethernet (802.3) MAC Receiver In Esterel

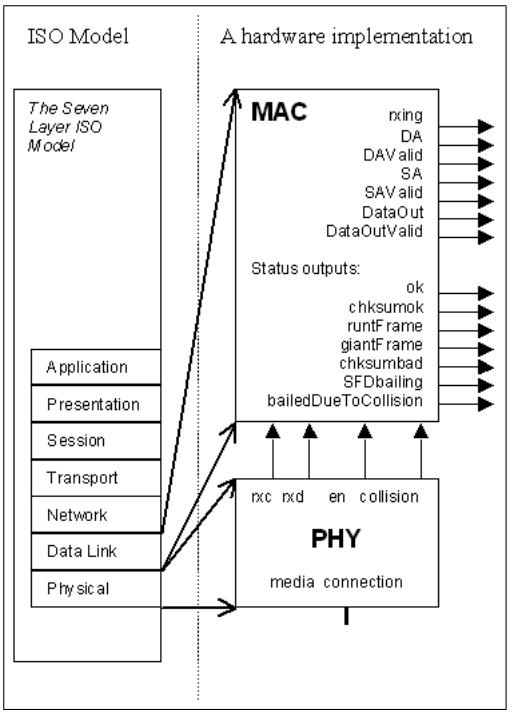
Esterel EDA Technologies
679 av. Dr. J. Lefebvre
06270 Villeneuve-Loubet, France

www.esterel-eda.com

February 13, 2008

This is a companion document to the Esterel Studio project `EthernetMACreceiver` available on the Esterel EDA Technologies web site. It describes particulars of the Esterel Studio implementation of the design and provides descriptions of each Esterel Studio configurations and the "self documenting" stimulus files which accompany the project.

Below is a reference diagram showing how the MAC receiver fits into a system. For those not already familiar with the Ethernet protocol at the MAC layer there is an appendix with a brief description it.



The diagram shows the ISO seven layer reference model on the left. On the right is a block diagram of the hardware for the two layers which are of concern for this design.

Note that the specific inputs and outputs are not part of the ISO specification. For this design project we are assuming a PHY device of a particular kind and we are assuming that certain types of outputs are required from the MAC ... your design needs may differ.

Briefly, at the top of the ISO model is the Applications layer; an email client for example. At that layer it is not important for the application to know nor care about how its message gets sent; whether or not it is sent in pieces, how error checking is done, if it sent by wire or wireless, etc. In this layered model messages are sent and received successfully provided that each layer communicates to its neighbors according to protocol.

So the PHY's responsibility is mainly turning activity on the media into well behaved digital signals. And the MAC monitors these signals and reports up to the next level if it found a packet amongst the activity, and reports some status about the packet. Just as the MAC level sits above the PHY and continuously monitors the PHY outputs, so too must the module which instantiates this MAC receiver continuously monitor the receiver's outputs.

The specifications for this particular receiver are:

1. provide and indication via the signal `rxing` that the receiver is acting on the PHY's outputs.
2. extract the Destination Address and present it in a 48 bit register,
3. extract the Source Address and present it in a 48 bit register,
4. extract all data and present it as it arrives on an 8-bit port, OutByte,
5. provide the following status/result outputs:
   - runtFrame : if, after the start of the frame, the `en` signal is deasserted before the minimum frame size arrives.
   - giantFrame : if, after the start of the frame, the `en` signal is NOT deasserted at or before the maximum frame size arrives.
   - chksumok : Frame Check Sum (FCS) == computed CRC on `en` deassertion.
   - chksumbad : the frame is within the right length, but shame about the CRC.
   - ok : the frame is within the right length and the CRC checks out ok.
   - SFDbailing : asserted every time the receiver is scanning for the SFD and receives an unexpected value or early end of transmission (`en` deassert).
   - bailedDueToCollision : reception was in progress but a late collision occurred.

1 Implementation Details

This section describes the main features of the design and details of their implementation.

1.1 module EthernetMACreceiver, the top level

Here is a fragment of the top level of the design:

```
module EthernetMACreceiver :
...
loop
    weak abort
        sustain rxing if en
        ||
        run MAC_rx_SFD   // emits  sfddetected, SDFbailing
        ;
        {
        run MAC_rx_DISS  // Awaits sfddetected, then extracts
                         // the DA, SA, and payload.
                         // emits giant, runtFrame
        ||
        run MAC_rx_CRC   // Awaits sfddetected, then starts CRC.
                         // emits chksumok, chksumbad
        }
    when
      case (giant)      do await immediate not en ;
                        emit giantFrame
      case (collision)  do await immediate not en ;
                        emit bailedDueToCollision
      case (chksumbad or runtFrame or SFDbailing) do nothing
      case (chksumok)   do emit ok
    end abort
    ;
    // After each transmission all devices must wait the IFS time
    emit assert END_OF_EN = not en;
    await IFStimeInTicks tick
end loop
```

The top level of the design places three sub modules in parallel, although the MAC_rx_DISS and MAC_rx_CRC modules wait on a signal from the MAC_rx_SFD before commencing operation. This signal is `sfddetected`, and the conditions for its assertion are simple: the `en` signal is asserted with no collisions the correct Start of Frame Delimiter pattern appears on `rxd`. Thereafter the receiver collects, and checks, data.

While there is a single simple entry condition, there are a number of exit conditions, and different designs may call for different priorities and different status indications from the receiver. In this design, the receiver presents the status outputs for only one cycle following the deassertion of `en`. But the signals could just as well be held until cleared by a controlling device, or held until the beginning of the next frame.

All three of the sub modules terminate on the same conditions shown in the case statements. The case statements are processed in order, highest to lowest priority. Here is a line by line explanation:

`case (giant) do await immediate not en ; emit giantFrame`

`giant` is emitted by `MAC_rx_DISS` the cycle that the number of bits received exceeds the maximum frame length. Being a giant implies that the `en` input has not been deasserted. In order to provide a consistent interface to the controlling logic which expects status to be available the cycle following the deassertion of `en`, the `emit giantFrame` statement is held off until `en` does finally deassert. It giant frames are to be allowed, perhaps if this design was used in a network protocol analyzer, a more promiscuous and error tolerant behaviour might be more appropriate. In that case the upper level should not abort on `giant`.

```
case (collision) do await immediate not en ; emit bailedDueToCollision
```

Similar to the `giantFrame` situation, a collision can come at any time, and therefore the receiver must save the collision notification until `en` deasserts.

```
case (chksumbad or runtFrame or SFDbailing) do nothing
```

The signals above are all primary outputs. Their very assertion is enough and there is `nothing` more to be done.

```
case (chksumok) do emit ok
```

Finally, if there were no problems, `emit ok`. Note that all these error conditions take priority over `ok`. This is important since, for example, it is possible to receive a runt frame that has a correct checksum. Simultaneous `runtFrame` and `chksumok` should not allow `ok` to be asserted.

Note that it is ultimately the deassertion of `en` that transfers control to the statement `await IFStimeInTicks tick`. As described in the appendix, the clock that this MAC receiver design is running off is derived from the PHY and we do not assume that the received clock `rxc` is continuous clock OR that it may come and go with the presence or absence of the bitstream. By assuming only a single `rxc` following the deassertion of `en` and using that deassertion to restart the loop, the receiver is prepared for either situation.

A note on the signal `rxing`: An indication that the receiver is in operation is primarily of interest to a companion transmitter. A transmitter can itself monitor the PHY outputs and determine when the line is open to attempt transmission, or it can utilize circuitry in the receiver. In this design, `rxing` is merely an echo of `en`, but it could be changed to include, for example, the receiver's wait on the IFStime.

1.2 module MAC_rx_SFD, Detecting the Start of Frame

Below is the body of the MAC_rx_SFD module.

```
await en and not pre(en) ;
weak abort
    loop
      weak abort // weak because 'x' must assert to exit the block
        await immediate rxd ; pause ;                // Got first 1
        emit preamblerestart if     rxd ; pause ;    // expecting 0
        emit preamblerestart if not rxd ; pause ;    // expecting 1
        emit preamblerestart if     rxd ; pause ;    // expecting 0
        emit preamblerestart if not rxd ; pause ;    // expecting 1
        emit preamblerestart if     rxd ; pause ;    // expecting 0
        emit preamblerestart if not rxd ; pause ;    // expecting 1
        loop
          emit sfddetected   if     rxd ; pause ;    // 1 for SFD or
                                                     // 0 to continue
                                                     // the preamble.
          emit preamblerestart  if not rxd ; pause ;// expecting 1
        end loop
      when preamblerestart
    end loop
when
    case (not en)      do emit SFDbailing
    case (collision)   do nothing
    case (sfddetected) do nothing // This is the normal out
end abort ;
```

The purpose of this module is to indicate to the other modules that the Start of Frame Delimiter has been detected. It must then "turn itself off" since if it continued to monitor the incoming bitstream it might continue to report `sfddetected` every time the 10101011 SFD pattern happened to passed by in the data stream. The `when ... case (x)` provides the exit condition for the `weak abort` statement.

The `case (collision) do nothing` does nothing because the `EthernetMACreceiver` module above causes this sub module to abort on collision. So the collision `case` here is a placeholder. If this design was used as a simple receiver then it should have such conservative behaviour. But if this design was used in a network protocol analyzer, a more promiscuous and error tolerant behaviour might be more appropriate. In that case the upper level abort on `collision` could be removed. There are in fact a number of `do nothing` statements in this design and they are simply placeholders for more elaborate behaviour.

1.3 module MAC_rx_DISS, Dissassembling the Frame

The code fragment below shows only the main control path of the MAC_rx_DISS module. The section replaced with the comment "//data collection" is uncomplicated and has been removed to highlight the control flow. However the entire module is in the appendix.

The control is: begin on the cycle following `sfddetected` and each clock thereafter do "data collection" and shift in `rxd` until `en` deassertion. If six bytes have been received, assert and hold `DAValid`, if six more are received, assert and hold `SAValid`, if too many bits have been received, assert `giant`, and if too few, assert `runtFrame`. Note that this module will abort if `en` deasserts as well if `giant` is asserted (See the `module EthernetMACreceiver, the top level` sub section for details).

```
await sfddetected ;
// The very NEXT tick after sfddetected is asserted will be
// bit 0 of the first byte of Destination Address.
abort
{
    loop
      emit next ?rcvdbyte[assert<8>(?bitcount)] <= rxd ;
      pause ;
    // data collection
    end loop
    ||
    sustain {
        next DAValid if DAValid or ?bytecount = 6,
        next SAValid if SAValid or ?bytecount = 12,

        next ?allbitcount <= assert<GIANT_BITS +1>(?allbitcount+1),
        giant if ?allbitcount > GIANT_BITS
    }
}  when not en
;
emit runtFrame if ?allbitcount < RUNT_BITS ;
```

Note that this module emits `giant`, causing an abort in the higher module. Also note that the receiver cannot determine if the frame is a runt until after `en` deasserts; hence the final line in the code above.

1.4 module MAC_rx_CRC, Checking the Checksum

This module consists of two sections operating in parallel; computing the CRC, and aligning the incoming FCS for comparison the computed CRC. The entire module can be found in the appendix. The control is trivial, wait for sfddetected and run continuously until the runmodule is terminated from above (Again, see the `module EthernetMACreceiver, the top level` sub section for details).

2 The "Self Documenting" Stimulus Files

The Esterel Studio project comes with twelve stimulus files. The first ten stimulus files:

```
helloworld.esi
misaligned_7bit.esi  misaligned_1bit.esi   misaligned.esi
onebitcorrupt.esi    short1bit.esi         toofewbytes.esi
onebittoobig.esi     giant.esi             collision.esi
```

are partly self explanatory by name, but the expected result is also coded in ASCII in the frame itself. If these stimulus files are run and the format ASCII is chosen in the waveform viewer for the OutByte bus then the expected behaviour can be read directly. Briefly,

- `helloworld.esi` chksumok, and is the only one which asserts "ok".
- `toofewbytes.esi` has a correct checksum, but is a runt.
- `giant.esi` has a correct checksum, but is a giant.
- `short1bit.esi` will be a misaligned (chksumbad), runt not reported.
- `collision.esi` late collision, assert bailedDueToCollision on en-¿ 0.
- `onebitcorrupt.esi` correct in size and alignment, but chksumbad.
- `onebittoobig.esi` giant, will be a misaligned (chksumbad) not reported.
- `misaligned_7bit.esi` chksumbad due to having 1 bit short of a byte.
- `misaligned_1bit.esi` chksumbad due to having 7 bits short of a byte.
- `misaligned.esi` chksumbad due to having 4 bits short of a byte.

The remaining two:

```
soferror.esi          en_deassertonpreamble.esi
```

have errors early in the transmission. `en_deassertonpreamble.esi`, has a corrupt preamble, which in itself doesn't stop the `MAC_rx_SFD` module from searching for a real SFD. But `en` is deasserted before that time which causes the assertion of `SFDbailing`.

The file `soferror.esi` begins with the bit pattern 1010101, but then receives a 0, and so fails to produce a correct SFD. The receiver continues to search for the 10101011 SFD pattern and happens to find one further on in the data stream. As expected, the computed CRC and FCS do not match. So a chksumbad is asserted. Also, because the "false" SFD was detected late, this frame also is reported as a runt.

3 Configurations

3.1 Formal Verification

For formal verification a new hierarchical layer is added in order to include the `MAC_rx_CHECKERS` module. That module contains three assertions:

```
assert CRC_OUTPUTS       = chksumbad # chksumok,
assert SIZE_OUTPUTS      = runtFrame # giantFrame,
assert NO_MIXED_MSSG     = ok => not (chksumbad or giantFrame or runtFrame)
```

which are derived from the specification in the appendix. `MAC_rx_CHECKERS` also includes six constraints on `en` and `collision` to prohibit the formal verification engine from creating impossible counter examples. There is also an `esi` file which can be used to push the design towards the end of a frame and better check termination behaviour with giant frames.

3.2 HDL Code Generation

The `Sim` configuration can also be used for code generation. The design as is can be compiled as "Monolithic monoclock" or as "Modular monoclock" provided that the three sub modules `MAC_rx_SFD`, `MAC_rx_DISS`, and `MAC_rx_CRC` have "No modular generation" selected as the "Modular HDL Generation Kind". Selecting "As top level module" as the "Modular HDL Generation Kind" will produce an error message such as the one shown below:

```
Error-ESTEREL(stn)-1: Module MAC_rx_SFD cannot be used as a toplevel
                                    submodule since it can terminate
```

As described in the `Implementation Details` section, the three run modules can be terminated from the top level. In order to manage this control, the compiler adds control signals. Because these signals are not part of the specified input and outputs of the submodule, they cannot generate them as independent "top level" modules.

A Reduced 802.3 MAC Receiver Specification

Ethernet traffic is transmitted one Frame at a time as a serial bitstream. The physical connection to the transmission medium, be it wired or wireless RF or IR, is at what is called the Physical layer (the PHY). Practically speaking, the PHY receiver device converts this bitstream into well behaved digital signals. These signals are provided to the next layer, the Media Access Control (MAC) layer, where this design module comes in.

The exact nature of those signals and their timing are dependent on the particular PHY chip used, but for the purpose of this design we assume that the PHY presents to this MAC receiver module the four single bit signals:

```
rxc,
rxd,
en,
collision.
```

`rxc` is the received clock.

`rxd` is the received data. It is sampled on all `rxc` rising edges while `en` is asserted.

`en` is the enable and indicates that the `rxd` signal is valid.

`collision`, which indicates that although there is a signal on the line the received data is not valid, which is probably due to multiple transmitters active simultaneously. If `collision` is asserted the MAC receiver must wait for the deassertion of `en` and return to listening for the next data frame (next assertion of `en`).

The primary job of the MAC receiver is to extract from those signals the data portion of the Frame. (The "data portion" is everything from the Destination address to the end of the Payload. See below.) Because it cannot be guaranteed that the data received is without error, the MAC receiver must also provide some error checking.

So in the main, the MAC receiver is a serial to parallel converter, with error checking.

The Frame has the following fields with the sizes shown:

```
+-------------------------------------------------------------------------+
|7 bytes |1 byte|   6 bytes      | 6 bytes   |2 bytes|46-1500 bytes|4 bytes|
+--------+------+----------------+-----------+-------+-------------+-------+
|Preamble| SFD  |Destination Addr|Source Addr| Type  |   Payload   |  FCS  |
+-------------------------------------------------------------------------+
 ^                                                                       ^
 |                                                                       |
 +-------- First bit sent.                          Last bit sent. -------+
```

SFD is Start of Frame Delimiter. It is the bit pattern 10101011. The very next bit after the SFD is the first bit of the Destination Address (DA).

All data following the final bit of the SFD up to the first bit of the FCS are transmitted BYTE wise, LSB to MSB. So if one is sending a payload of "Hello, World!", that will appear on OutByte as:

```
| 48 | 65 | 6C | 6C | 6F | 2C | 57 | 6F | 72 | 6C | 64 | 21 |
but it will appear on the bitstream as:
| 12 | A6 | 36 | 36 | F6 | 34 | EA | F6 | 4E | 36 | 26 | 84 |
```

FCS is Frame CheckSum. The MAC receiver module will calculate a CRC from the first bit of the Destination Address to the last bit of the payload. It is the same CRC algorithm

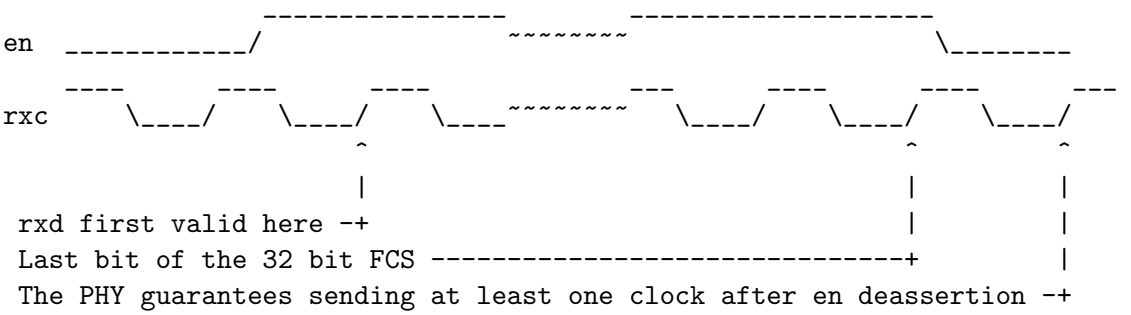
that was used to compute the FCS, so the last 4 bytes which contain the transmitted FCS must match the computed CRC. The FCS is transmitted bit 31 down to bit 0, not byte by reversed byte as with the rest of the frame.

Following is a list of the assumptions about the behaviour of the PHY "device" that this MAC receiver module receives input from.

The clock that this MAC receiver design is running off is derived from the PHY and that the PHY has synchronized `rxd` to the receiver's clock. But importantly, we assume that the received clock, `rxc`, may be a continuous clock from the PHY chip, OR it may come and go with the presence or absence of the bitstream.

For the purpose of this design, this does not matter. `rxc` in the input clock and is always well behaved (no short pulses, no changing frequency, etc) with the exception that, while `en` is inactive, `rxc` may be held low for long periods of time.

The PHY's relationship between `rxc` and `en` are as follows:



In summary, at a minimum this MAC layer must continuously monitor the four signals from the PHY and extract data and perform the following checks that should be performed:

1. the frame must be at least as long as the specified minimum frame length,
2. the frame must be no longer than the specified maximum frame length,
3. the frame must contain an integer number of bytes,
4. the receiver should generate a Cyclic Redundancy Check (CRC) and compare it to the incoming frame's Frame CheckSum (FCS).
5. The receiver should also detect collisions and failures in the SFD and subsequently wait for the `en` signal to be deasserted then await the reassertion of `en`.

NOTE: If the output OutByte is displayed in ASCII format in a waveform viewer, the received frame will be human readable (If it was sent as human readable that is).

NOTE: To make testing easier, the maximum and minimum size frames are greatly restricted to a range between 31 and 51 bytes inclusive. The InterFrame Spacing (IFS) which is specified to be 9.6 usec at 10 M bits/sec is reduced to 10 clock ticks.

A module MAC_rx_DISS, Dissassembling the Frame

```
await sfddetected ;
// The very NEXT tick after sfddetected is asserted will be
// bit 0 of the first byte of Destination Address.
abort
{
    loop
      emit next ?rcvdbyte[assert<8>(?bitcount)] <= rxd ;
      pause ;
      switch ?bitcount
        case 7 do
          emit next {
            ?bitcount <= 0,
            DataOutValid,   // Capture data on byte boundaries.
            ?DataOut <= ?rcvdbyte,
            ?bytecount <= assert<(GIANT_BYTES)+1>(?bytecount + 1)
          }
        case 0 do
          emit next {
            ?bitcount <= assert<8>(?bitcount + 1),

            ?DA[40..47] <= ?rcvdbyte if ?bytecount = 1,
            ?DA[32..39] <= ?rcvdbyte if ?bytecount = 2,
            ?DA[24..31] <= ?rcvdbyte if ?bytecount = 3,
            ?DA[16..23] <= ?rcvdbyte if ?bytecount = 4,
            ?DA[8..15]  <= ?rcvdbyte if ?bytecount = 5,
            ?DA[0..7]   <= ?rcvdbyte if ?bytecount = 6,

            ?SA[40..47] <= ?rcvdbyte if ?bytecount = 7,
            ?SA[32..39] <= ?rcvdbyte if ?bytecount = 8,
            ?SA[24..31] <= ?rcvdbyte if ?bytecount = 9,
            ?SA[16..23] <= ?rcvdbyte if ?bytecount = 10,
            ?SA[8..15]  <= ?rcvdbyte if ?bytecount = 11,
            ?SA[0..7]   <= ?rcvdbyte if ?bytecount = 12
          }
        default do
            emit next ?bitcount <= assert<8>(?bitcount + 1)
      end switch
    end loop
    ||
    sustain {
        next DAValid if DAValid or ?bytecount = 6,
        next SAValid if SAValid or ?bytecount = 12,

        next ?allbitcount <= assert<GIANT_BITS +1>(?allbitcount+1),
        giant if ?allbitcount > GIANT_BITS
    }
}  when not en
;
emit runtFrame if ?allbitcount < RUNT_BITS ;
```

A module MAC_rx_CRC, Checking the Checksum

```
await sfddetected ;
{ // sfddetected
loop     // aligning the FCS data BEGIN
    emit  next ?LastByte[ assert<8>(?bitcount)] <= rxd ;
    pause ;
    switch ?bitcount
      case 7 do
            emit next ?bitcount <= 0

      case 0 do
            emit next {
              ?bitcount <= assert<8>(?bitcount + 1),

              ?Last4Bytes[8..31]  <= ?Last4Bytes[0..23],
              ?Last4Bytes[0..7]   <= reverse(?LastByte),

    /* To end the CRC, since the frame length is unknown, the CRC
     * cannot be computed as the bits arrive or the Computed CRC
     * will include the incoming FCS in the computation. The
     * solution here is to delay the start of the CRC computation
     * by the FCS length such that the received CRC and computed
     * CRC align at the final bit.
     */
              ?LastCRC5   <= ?LastCRC4,
              ?LastCRC4   <= ?LastCRC3,
              ?LastCRC3   <= ?LastCRC2,
              ?LastCRC2   <= ?LastCRC1,
              ?LastCRC1   <= ?ComputedCRC
            }
      default do
            emit next ?bitcount <= assert<8>(?bitcount + 1)
    end switch
end loop // aligning the FCS data END
||
  {
  // computing the CRC BEGIN
    pause ;
    /* This delay is here because the SFD statement terminates
     * immediately, meaning that this process is entered while the
     * final start of frame delimiter bit is present. The very NEXT
     * tick will be bit 0 of the first byte of Destination Address.
     * The CRC needs to start on that tick.
     */
    abort
      sustain x32 <= ?ComputedCRC[31] xor rxd
      ||
      loop
/*
 * CRC32 is:
 * 2^32 + 2^26 + 2^23 + 2^22 + 2^16 + 2^12 + 2^11 + 2^10 + 2^8 +
 *  2^7 + 2^5 + 2^4 + 2^2 + 2^1 + 1
 */
```

```
        emit next {
            ?ComputedCRC[27..31] <= ?ComputedCRC[26..30],
            ?ComputedCRC[26]     <= ?ComputedCRC[25]      xor x32,
            ?ComputedCRC[24..25] <= ?ComputedCRC[23..24],
            ?ComputedCRC[23]     <= ?ComputedCRC[22]      xor x32,
            ?ComputedCRC[22]     <= ?ComputedCRC[21]      xor x32,
            ?ComputedCRC[17..21] <= ?ComputedCRC[16..20],
            ?ComputedCRC[16]     <= ?ComputedCRC[15]      xor x32,
            ?ComputedCRC[13..15] <= ?ComputedCRC[12..14],
            ?ComputedCRC[12]     <= ?ComputedCRC[11]      xor x32,
            ?ComputedCRC[11]     <= ?ComputedCRC[10]      xor x32,
            ?ComputedCRC[10]     <= ?ComputedCRC[9]       xor x32,
            ?ComputedCRC[9]      <= ?ComputedCRC[8],
            ?ComputedCRC[8]      <= ?ComputedCRC[7]       xor x32,
            ?ComputedCRC[7]      <= ?ComputedCRC[6]       xor x32,
            ?ComputedCRC[6]      <= ?ComputedCRC[5],
            ?ComputedCRC[5]      <= ?ComputedCRC[4]       xor x32,
            ?ComputedCRC[4]      <= ?ComputedCRC[3]       xor x32,
            ?ComputedCRC[3]      <= ?ComputedCRC[2],
            ?ComputedCRC[2]      <= ?ComputedCRC[1]       xor x32,
            ?ComputedCRC[1]      <= ?ComputedCRC[0]       xor x32,
            ?ComputedCRC[0]      <=  rxd    xor ?ComputedCRC[31]
        } ;
        pause ;
      end loop ;
    when not en ;

    emit { // Always emit one or the other chksum signals
      if ((?LastCRC5=?Last4Bytes)and(?bitcount=1))then chksumok
      else                                            chksumbad
      end if
    }
  // computing the CRC END
  }
} // sfddetected
```