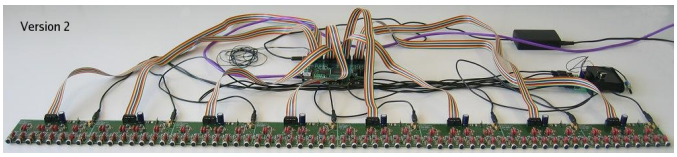
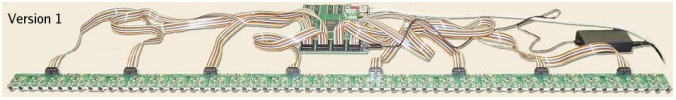
Documentation of the synchronization board for markIIIs

ROCHET Cedrick





Information Access Division
National Institute of Standards and Technology *

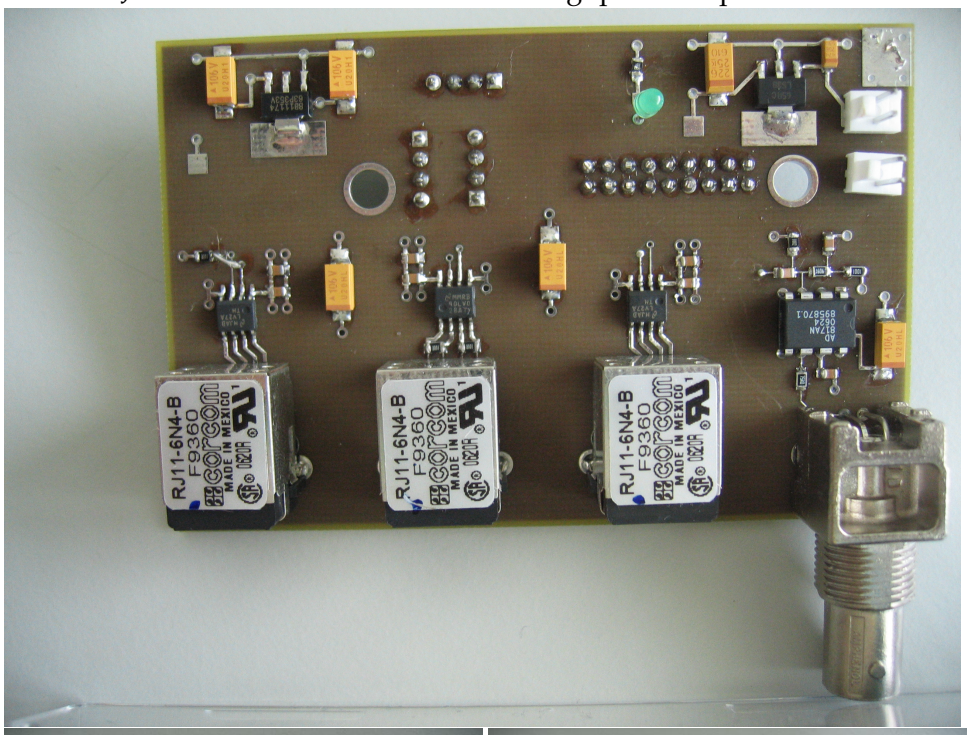
November 23, 2006

*NIST, 100 Bureau Drive, Stop 3460, Gaithersburg, MD 20899-3460., www.nist.gov

Contents

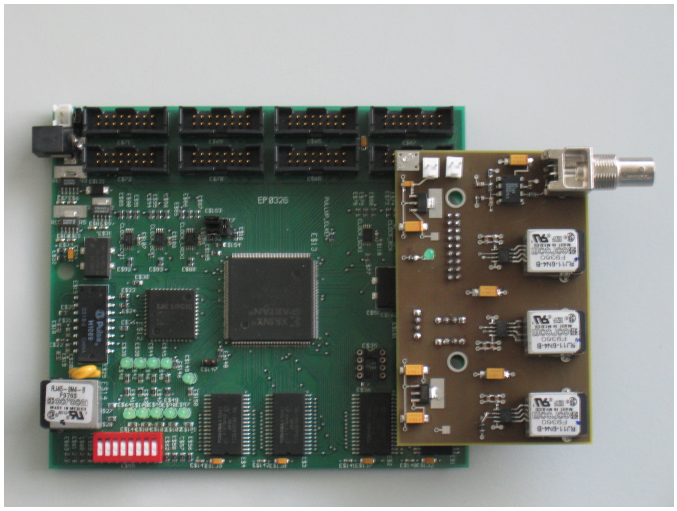
1 Setup

The kit is composed of the synchronization board, a power cable and a RJ11 CAT5e cable. The following pictures present each one.

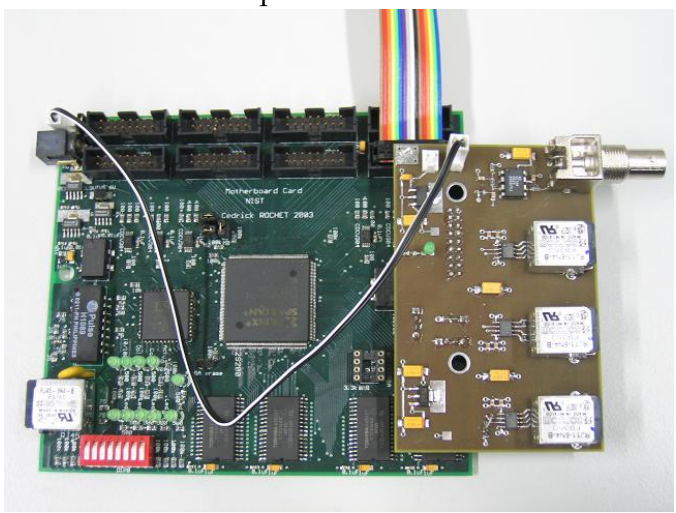


The setup of the system is quite simple. Proceed as follow:

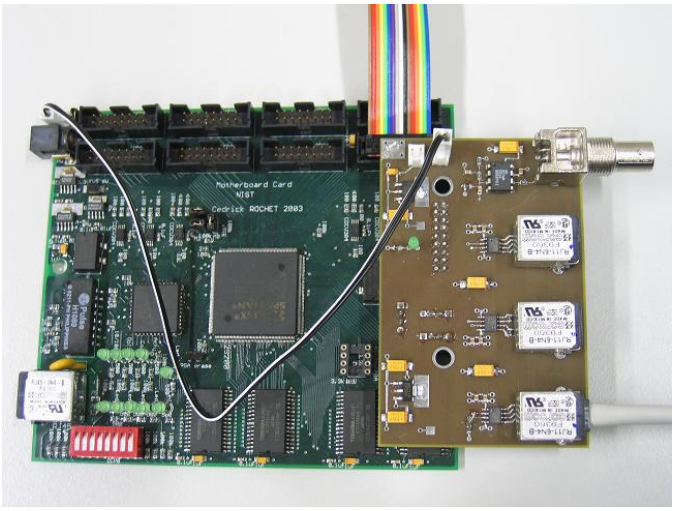
1. Plug the synchronization board onto the motherboard.



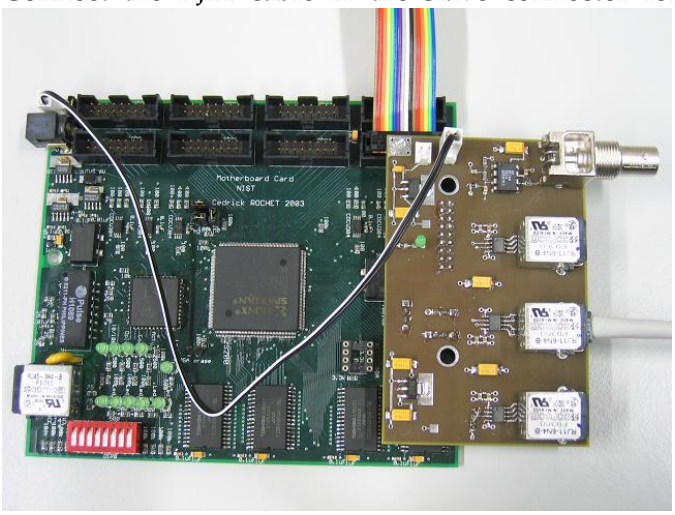
2. Connect the power cable between both boards.



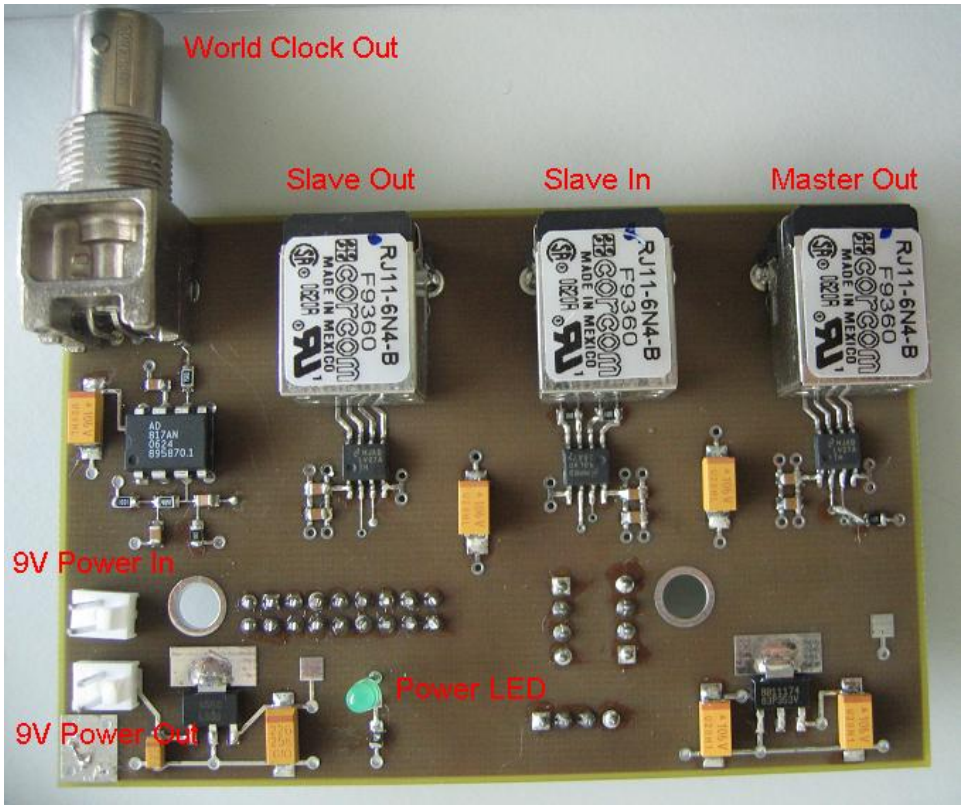
3. Connect the RJ11 cable in the master connector for the master card



4. Connect the RJ11 cable in the slave connector for the slave card



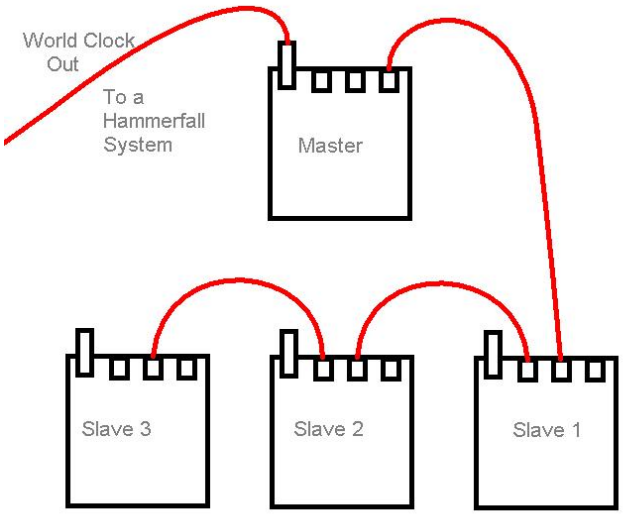
The card has the following inputs-outputs:



1. Master Out is the connector for the output synchronization signal of the master array.

2. Slave In is the connector for the input synchronization signal from the master array.

3. Slave Out is the connector for the ouptut synchronization signal from the master array. The system is built following a chain of arrays with one slave array forwarding the signal to the next one as shown on the following schematic.



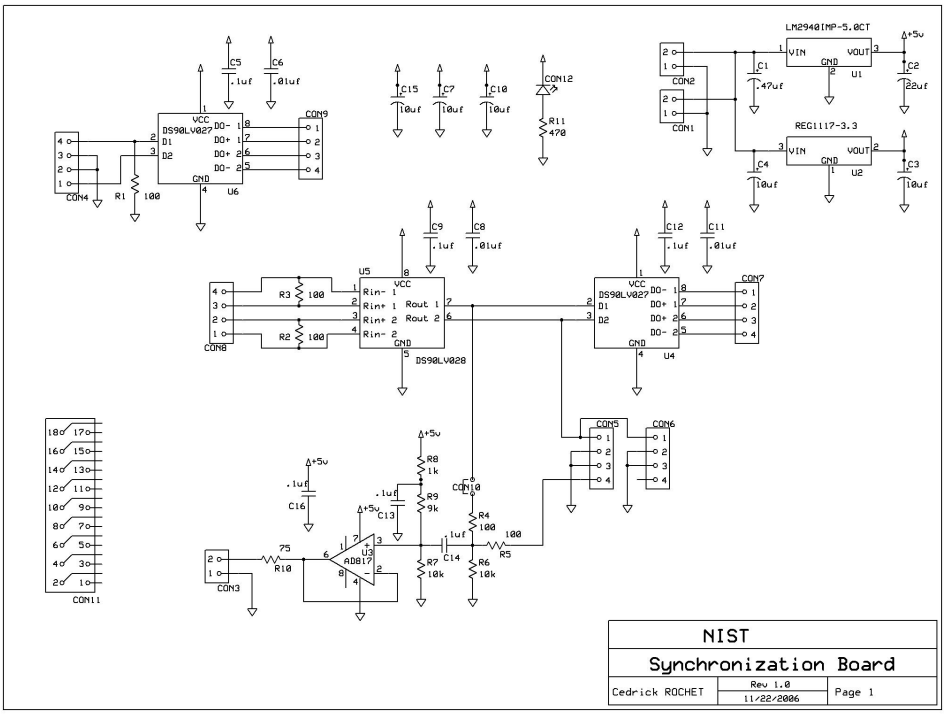
4. World clock Out is the connector to send a world clock signal to another system. This signal is just the sampling frequency of the array.

5. 9V power In is the connector for recieving the power from the motherboard.

6. 9V power Out is the connector to provide the 9V of the power supply to any additional board. It can also be used to connect a recommneded metal box to the ground.

7. Power Led indicates if the synchronization board recieves power.

2 schematics



3 VHDL code on the motherboard

In VHDL, there is one main module who contains all the other ones. The figure 1 represents an overview of the eleven submodules interactions.

In order to go deeper in the explanations, we are going to take each of the twelve modules :

- synchronization module,
- capture module,
- sram interface module,
- capture_udp_frame module,
- bootp module,
- arp module,

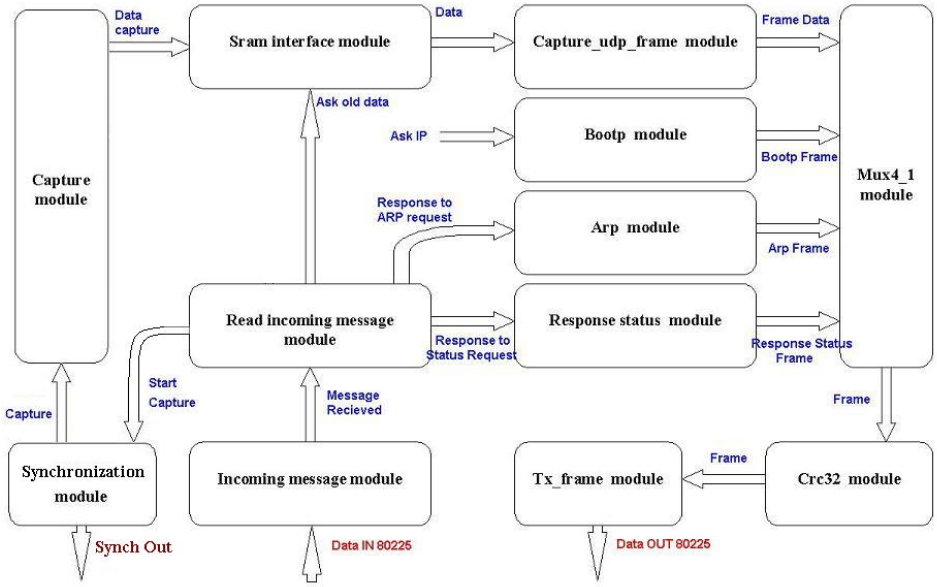


Figure 1: Gathering schematic of all the modules.

- response status module,
- mux4_1 module,
- CRC32 module,
- tx_frame module,
- incoming message module,
- read incoming message module.

In the main module, there is two sub-programs:

- Comptetemps: process to count time in seconds,
- state_machine: process used at startup to ask IP address as a starting point to be completely operational.

The rest of this module is just the connection between the different sub-modules like on figure 1.

3.1 The synchronization module

This module is used to synchronize multiple arrays and generate the right clock for the capture module.

Figure 2: The capture module.

The *master_clk* is provided by the oscillator on the Motherboard. The *slave_clk* is provided by an other Motherboard through the synchronization board. The signal *low_reset* is used to initialize the state machines in the module. The *SCKI*(System Clock) for the converters is generated from *master_clk* or *slave_clk* depending on the mode of the array. It is at 11.289MHz $(22.050 * 512)$.

First in master mode meaning when *sync_slave* is low, *master_clk* is forwarded to the capture module through *cap_clk*. The *start_capture_in* is a signal used to start the capture module by forwarding it through *start_capture_out* but with a fixed delay read on *start_offset(7:0)*. *lrck* signal which is the sampling frequency of the system is forwarded to *sync_cap_clk_slave*.

The signal *sync_cap_clk_master* is the signal distributed to the slave Motherboards and received as *sync_cap_clk_slave*. It is a succession of 3 words:

- "0101101" to reset the clock divider in the FPGA,
- "0101001" to start capture,
- "0100101" to stop capture.

In slave mode meaning when *sync_slave* is high, *slave_clk* is forwarded to the capture module through *cap_clk*. The system is waiting for its

command words though the input *sync_cap_clk_slave* upon reception of the words it takes the same action as the master. Upon reception of the start word, the *start_capture_out* is started but with a fixed delay read on *start_offset(7:0)*.

3.2 The capture module

It is part of the design where the data coming from the converters are standardized in packets.

The figure 3 can be decomposed it in two parts:

- the interface with the converters and
- the interface with the rest of the design which is a memory.

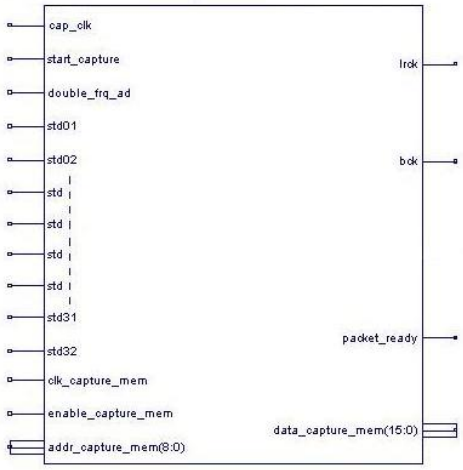


Figure 3: The capture module.

First, the signal *start_capture* is used at the beginning to tell when the capture shall start. The *cap_clk* is the clock on which everything is synchronized.

The interface with the converters is composed of 3 clocks (*BCK, SCKI* and *LRCK*) and 32 inputs(*std01-std32*). *cap_clk* (33.8688MHz) is the main clock that builds the 3 others (for the hardware see the clock stage). In fact these clocks depends of the sampling rate. For example for a sampling rate of 22.050kHz in 24 bits: *LRCK* (sampling clock) value is 22.050kHz, *BCK*

(Bit Clock) value is 1.058MHz $(22.050 * 24 * 2)$ and *SCKI*(System Clock) is 11.289MHz $(22.050 * 512)$. The signal *double_frq_ad* forces this module to double the sampling frequency and in doing so doubles the volume of data.

The return of the converters, *stdxx*, is serial so this module take care of placing the different streams like describes in the figure 4.

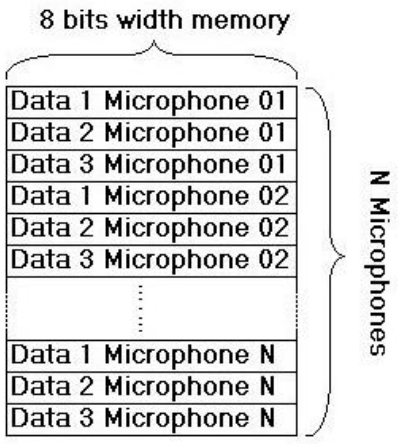


Figure 4: Data organization in the memory.

The advantage of using a memory is in the fact that it is using a dual port ram memory. So the process of capturing the data clocked on *cap_clk* is independent from the rest of the design like the Ethernet or storage part. So the sram interface module is interacting with a simple memory: *data_capture_mem(15:0)*, *addr_capture_mem(8:0)*, *enable_capture_mem*, *clk_capture_mem*. The signal *packet_ready* tells the rest of the design when the memory is filled and can be read.

3.3 The sram interface module

In this module, we are storing the data from the capture module into 4 physical memory chips on a bus of 32 bits. The storage is used as a buffer of about 0.5 seconds in the case that the packet is incorrect or dropped by the computer.

First, *start_capture* is a signal used at the beginning to tell when the capture shall start. The signal *reset_sram_interface* is used at startup to initialize the signals inside the module. In this module, there is 3 input clocks:

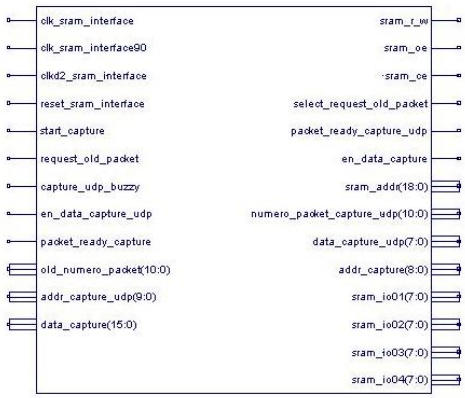


Figure 5: The sram interface module.

- *clk_sram_interface* is the main clock through the module,
- *clkd2_sram_interface* is a just used to synchronize the division by 2 of the main clock in this module and in the capture_udp_frame module,
- *clk_sram_interface90* is used by the inner memory to compensate for the delay in the data bus.

Basically, this module reads the data in the memory of the capture module, transfers it to the external memory and then reads in the external memory the packet asked and put it in a memory that can be read by the capture_udp_frame module.

So the signals *data_capture(15:0)*, *addr_capture(8:0)*, *enable_data_capture*, *clk_sram_interface* and *packet_ready_capture* are respectively connected to *data_capture_mem(15:0)*, *addr_capture_mem(8:0)*, *enable_capture_mem*, *clk_capture_mem* and *packet_ready* of the capture module.

As we have seen in the hardware, the signals *sram_r_w* (read/write control), *sram_oe* (output enable), *sram_ce* (chip enable) are controlling the external memory. The four 8 bits bus *sram_io01*, *sram_io02*, *sram_io03*, *sram_io04* are the data bus with the memory. The bus *sram_addr(18:0)* is controlling the address of the external memory.

The capture_udp_frame module, connected to this module to get the data out, is interacting with a simple memory: *data_capture_udp(7:0)*, *addr_capture_udp(9:0)*, *en_data_capture_udp*, *clkd2_sram_interface*.

The signal *packet_ready_capture_udp* tells to the capture_udp_frame module when the memory is filled and can be read. The bus *numero_packet_capture_udp(10:0)* gives the packet identifying number.

The signal *capture_udp_buzzy* tells when the memory is actually used by the capture_udp_frame module.

To ask a missed packet, the mem_read_incoming_msg module sends the information on the bus *old_numero_packet(10:0)* and put *req_old_packet* to 1 until the information is used.

3.4 The capture_udp_frame module

This module is getting the data form the memory of the sram interface module and put it in a UDP frame.

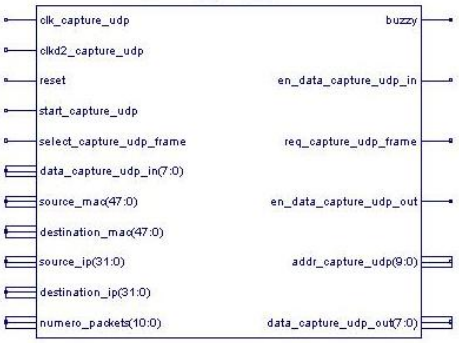


Figure 6: The capture_udp_frame module.

First, the *clk_capture_udp* is the main clock through this module. The clock *clkd2_capture_udp* is a just used to synchronize the division by 2 of the main clock in this module and in the sram interface module. the signal *reset* is used at startup to initialize the state machine inside the module. The signal *buzzy* tells to other modules that it's making a frame while it is high.

The module builds the different UDP protocol fields from the inputs *destination_mac* (MAC address of the destination provided by the main module), *source_mac* (MAC address of the source provided by the main module), *destination_ip* (IP address of the destination provided by the main module), *source_ip* (IP address of the source provided by the main module).

The different protocols need more information than the few provided by the main module but as they don't really change from one message to

another, they are directly fixed in the software: the UDP port has been fixed to 32767, the size of the packet to 964.

Figure 7: The data frame.

The data frame is constituted of different fields (cf figure 7) filled as follow:

- type packet is on 1 byte: 86 (8 as response and 6 as response of type 6),
- numero packet is on 2 bytes in a range from 0 to 2048 (it's corresponding to the 11 highest bits of the sram memory address),
- reserved is on 1 byte,
- data is on 960 bytes: 64 channels * 3 bytes precision * 5 data frames.

This module is taking the data in the memory of the sram interface module with the signals : *data_capture_udp_in(7:0)*, *addr_capture_udp(9:0)*, *en_data_capture_udp_in*, *clk_capture_udp* (clock of the module).

And finally the module sends the complete message to the mux4_1 module with *req_capture_udp_frame*, *select_capture_udp_frame*, *data_capture_udp_out(7:0)* and *en_data_capture_udp_out*. For more information, see the mux4_1 module.

3.5 The bootp module

This module is activated at startup in order to make a request IP address.

First, the *clk_bootp* is the main clock through this module. The signal *reset* is used at startup to initialize the signals inside the module. The signal *buzzy* tells to other modules that it's making a frame while it is high. The signal *start_bootp* activates the module.

Since you can fix the *source_mac* (MAC address of the source provided by the main module) by hand with the DIP switch, this value couldn't be implemented in advance.

The signal *seconds(7:0)* gives the elapsing time since startup.

And finally the module sends the complete message to the mux4_1 module with *req_bootp_frame*, *select_bootp_frame*, *data_bootp_out(7:0)* and *en_data_bootp_out*. For more information, see the mux4_1 module.

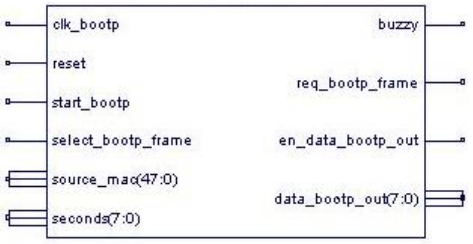


Figure 8: The bootp module.

3.6 The arp module

This module is activated to reply to an arp request to any computer placed on the network.

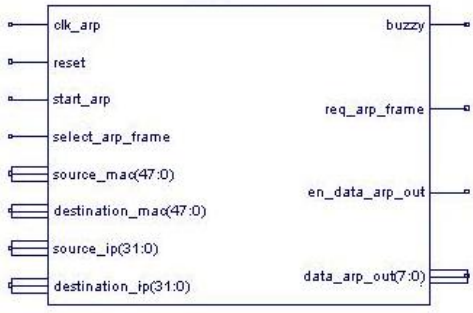


Figure 9: The arp module.

First, the *clk_arp* is the main clock through this module. The signal *reset* is used at startup to initialize the signals inside the module. The signal *buzzy* tells to other modules that it's making a frame while it is high. The signal *start_arp* activates the module.

The module builds the different ARP protocol fields from the inputs *destination_mac* (MAC address of the destination provided by the read incoming message module), *source_mac* (MAC address of the source provided by the read incoming message module), *destination_ip* (IP address of the destination provided by the read incoming message module), *source_ip* (IP address of the source provided by the read incoming message module).

And finally the module sends the complete message to the

mux4_1 module with *req_arp_frame*, *select_arp_frame*, *data_arp_out(7:0)* and *en_data_arp_out*. For more information, see the mux4_1 module.

3.7 The response status module

This module is activated to reply to any request made to the microphone array and received and understood by the module read message.

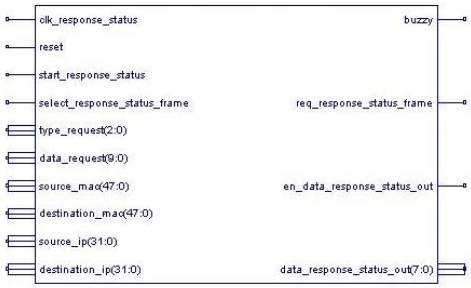


Figure 10: The response status module.

First, the *clk_response_status* is the main clock through this module. The signal *reset* is used at startup to initialize the signals inside the module. The signal *buzzy* tells to other modules that it's making a frame while it is high. The signal *start_arp* activates the module.

The module builds the different UDP protocol fields from the inputs *destination_mac* (MAC address of the destination provided by the read message module), *source_mac* (MAC address of the microphone array), *destination_ip* (IP address of the destination provided by the read message module), *source_ip* (IP address of the the microphone array), *type_request(2:0)* (number given to identify the response) and *data_request(9:0)* (data of the response).

Here is the different types of response implemented:

- request 02: status of slave/master mode,
- request 03: ID of the microphone array,
- request 05: status of the capture.
- request 08: status of the sampling frequency multiplier.

- request 11: value of the delay to start the capture. It used to compansate the propagation of the synchronization signal from one array to the other.

And finally the module sends the complete message to the mux4_1 module with *req_response_status_frame*, *select_response_status_frame*, *data_response_status_out(7:0)* and *en_data_response_status_out*. For more information, see the mux4_1 module.

3.8 The mux4_1 module

This module is a multiplexer of the bootp module output, arp module output, response status module output and capture_udp_frame module that sends the frame in the CRC32 module.

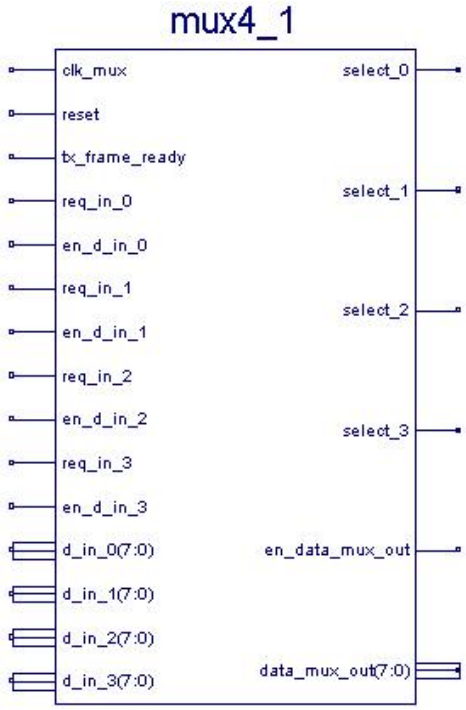


Figure 11: The mux4_1 module.

First, the *clk_mux* is the main clock through this module. The signal *reset* is used at startup to initialize the signals inside the module.

One of the four previous module makes a request with *req_in_x*. Then when the multiplexer is ready to give him the right of passage to the CRC32 module, the mux4_1 module put the signal *select_x* to high. At the same time the data entering made by the signals *en_d_in_x* and *d_in_x(7:0)* is directly connected to *en_data_mux_out* and *data_mux_out(7:0)* which is connected to the entry of the CRC32 module.

The signal *tx_frame_ready* tells when the previous frame has been sent completely to the 80225.

3.9 The CRC32 module

This module is adding the CRC32 value of the frame at the end of it.

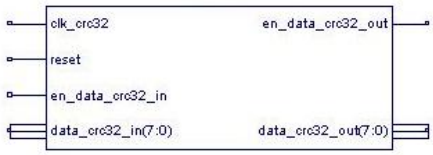


Figure 12: The CRC32 module.

This module is quite simple: it's taking the data in *data_crc32_in(7:0)* when *en_data_crc32_in* is high, adding the value of the computation of the CRC32 at the end and sending the data through *data_crc32_out(7:0)* and *en_data_crc32_out* to the tx_frame module.

3.10 The tx_frame module

This module is adding the preambule to the frame and changing from 8bit wild to 4 bits wild to enter the 80225.

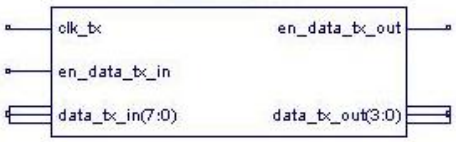


Figure 13: The tx_frame module.

Like the previous one, this module is quite simple: it's taking the data in *data_tx_in(7:0)* when *en_data_tx_in* is high, adding the preamble in front and sending the whole message to the 80225 through *data_tx_out(3:0)* and *en_data_tx_out*.

3.11 The incoming message module

This module takes any message coming form the Ethernet physical layer device, filters the messages with the right MAC address , verify the CRC32 of it and put it in the memory of the FPGA to put read.

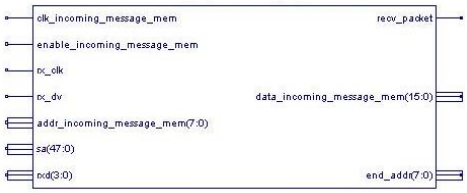


Figure 14: The incoming message module.

The data from the 80225 are coming from the signals *rx_clk*, *rx_dv* and *rxd(3:0)*. The message MAC address destination is compared to the signal *sa(47:0)* and if everything is right *recv_packet* goes high. Then the module read_incoming_message reads it in the memory with *clk_incoming_message_mem*, *addr_incoming_message_mem(7:0)*, *data_incoming_message_mem(15:0)* and *enable_incoming_message_mem* until *end_addr(7:0)*.

3.12 The read incoming message module

This module is one of the most complex of the design because it's reading the memory in the incoming message module and takes action based on the message.

The data read from the incoming message module is passing by *clk_mem_read*, *addr_incoming_msg_mem(7:0)*, *data_incoming_msg_mem(15:0)* and *enable_incoming_msg_mem* until *end_addr_msg(7:0)* when the signal *recv_packet_msg* went high.

In any case it stores the MAC and IP address of the sender *mac_sender(31:0)*, *ip_sender(47:0))* and tests if the IP address of destination is correct. In our case the one of the microphone array, *my_ip(31:0)*.

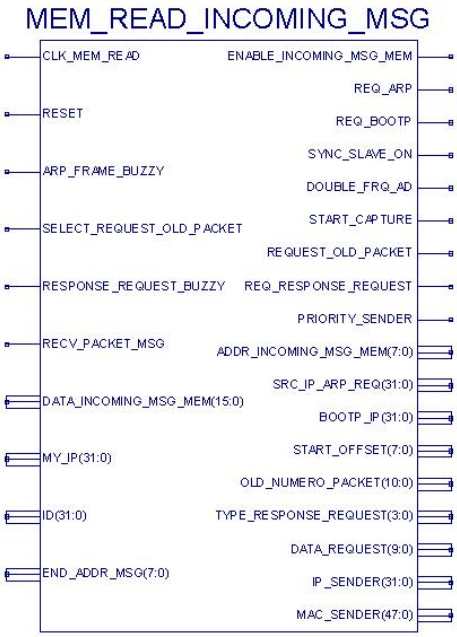


Figure 15: The read incoming message module.

As the message is read along, the module determines the kind of message depending of the protocol used:

- ARP: *req_arp* goes high and the IP source is *src_ip_arp_req(31:0)*,
- BOOTP: *req_bootp* goes high and the IP is given with *bootp_ip(31:0)* after verification of the "random" number *ID(31:0)*,
- request 01: slave/master mode. the output is *sync_slave_on*,
- request 02: ask the status of the slave/master mode. The output are *req_response_request*=1, *type_response_request(3:0)*= "0010", *data_request(9:0)*="0x00000000x" (x=1 means slave mode activated),
- request 03: ask the ID of the microphone array. The output are *req_response_request*=1, *type_response_request(2:0)*= "0011", *data_request(9:0)*=MAC address microphone array(9:0),
- request 04: capture on/off. the output is *start_capture*,

- request 05: ask the status of the capture. The output are *req_response_request*=1, *type_response_request(3:0)*= "0101", *data_request(9:0)*="0000000000x" (x=1 means capture mode activated),

- request 06: ask the packet in memory with the number *old_numero_packet(10:0)* and *request_oldpacket* goes high.

  If the microphone is not in capture mode, it will send back an error with the output *req_response_request*=1, *type_response_request(3:0)*= "0110", *data_request(9:0)*="0001101110" (binary for 'n').

- request 07: sampling frequency is doubled or not. The output is *double_frq_ad* and up when doubled.

- request 08: ask the status of the sampling frequency multiplier. The output are *req_response_request*=1, *type_response_request(3:0)*= "0111", *data_request(9:0)*="0000000000x" (x=1 means sampling frequency doubled),

- request 09: ask a range of packets in memory through the number *old_numero_packet(10:0)* and *request_old_packet* goes high. *old_numero_packet(10:0)* is increasing by one at each *request_old_packet* until it meets the end value of the range. Be careful in using this function because it won't stop the normal data packet traffic but will insert the packets asked in the middle of the data traffic... The activation of this option on range more than 5 packets can overload your network card under to much traffic... Remember that at normal operation the traffic on the line is about 4.4MBytes per second.

  If the microphone is not in capture mode, it will send back an error with the output *req_response_request*=1, *type_response_request(3:0)*= "0110", *data_request(9:0)*="0001101110" (binary for 'n').

- request 10: set the value of the delay in the start of the capture the output is *start_offset(7:0)*

- request 11: ask the value of the delay in the start of the capture. The output are *req_response_request*=1, *type_response_request(3:0)*= "1011", *data_request(9:0)*=value of the offset.

If one of the signals *arp_frame_buzzy* or *response_request_buzzy* is high then if an other frame come along the current one is ignored.

The signal *priority_sender* permits to differentiate the IP and MAC address of the computer receiving the data of the capture from another one making a request.

3.13 The MI interface for configuration and status

The MI(Media Interface) is the interface with the 80225 registers.

The 80225 has a MI serial port to access the device's configuration inputs and read out the status outputs.the MI serial port consists of 8 lines: MDC, MDIO, MDINT, and MDA[3:0]. However, only 2 lines, MDC and MDIO, are needed to shift data in and out. So this permits the engine of the design to configure the 80225. But since everything is in auto-negotiation there is no real use of it.