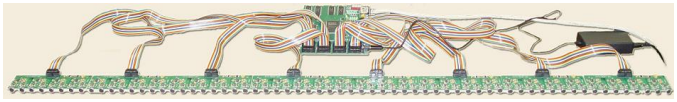
Documentation of the Microphone Array Mark III

ROCHET Cedrick



Information Access Division
National Institute of Standards and Technology *

September 25, 2003

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

Contents

1 Historic.

In this part we are going to see the previous models of microphone arrays that were developed at NIST.

1.1 The First Generation

The first generation was very primitive. Each Ns microphones were put in parallel. So the output signal was the sum of each microphone. The figure 1 illustrates this.

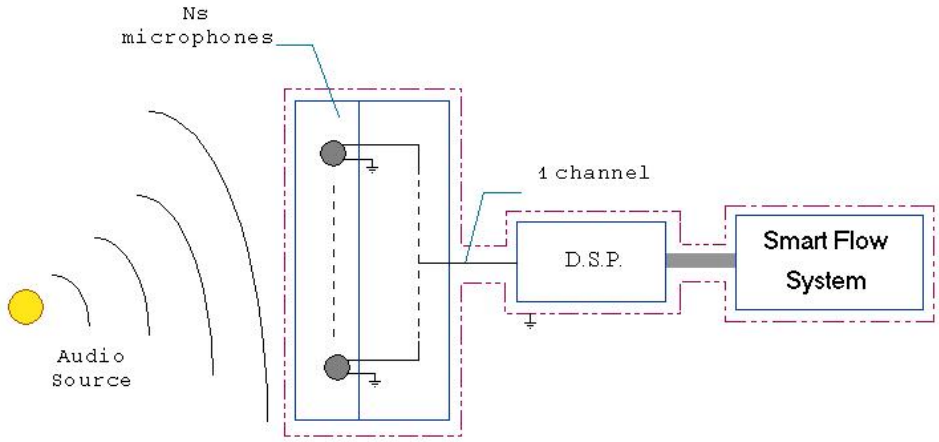


Figure 1: Microphone Array Mark I schematic.

In this case the beamforming was done analogically and at $90\,\mathrm{deg}$ of the center. The result is a single output analogic signal as seen on the overview of the microphone array first generation in figure 2.

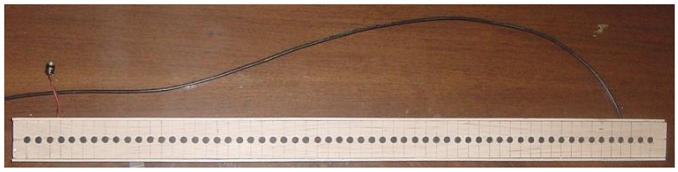


Figure 2: Microphone Array Mark I.

1.2 The Second Generation

The second generation is actually the one currently in use. At the beginning of my internship, I had to build three second generation array to support the construction of an advanced meeting room data acquisition facility . In the same time I learned how the Smart Flow System works. As you can see, on figure 3 , the system is relatively simple but efficient.

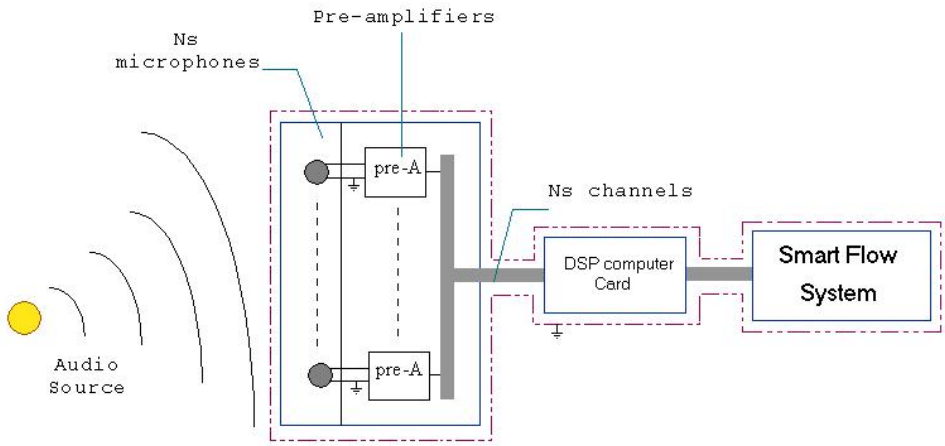


Figure 3: Microphone Array Mark II schematic.

Fast and efficient, DSP is used in signal processing to acquire and process data. It consists of an A/D converter with a processor, and interfaces. The DSP hardware that will be used in this project has only 64 inputs. Thus, we are restricted to use no more than 64 channels (i.e., 64 sensors).

Figure 4: Microphone Array Mark II.

As you can see on the figure 4, the second generation array has a discrete port and a preamplifier for each microphone. The second generation microphone array used the analogical preamplifier shown on figure5.

Figure 5: Microphone Array Mark II Preamplifier.

While the second generation array allowed for digital phased array processing, on this model there were many problems:

- This simple preamplifier is very sensitive to electromagnetic interference because of the technology employed and the size of this card.
- In order to make the connection between the microphone array and the DSP computer's card we need a one hundred conductor cable and connector set. The connections on the both sides of the cable are very fragile and it can break the contact easily. On the DSP computer's card the 64 channels are multiplexed through eight analog to digital converters, which introduce a fractional sample phase delay in groups of eight channels. So, in order to clean the signal of each microphone we designed digital codec band-pass filters.
- Also, intermittent loss of contact between some array channels and the card occur
- On the cable all the channels are analogical fall prey to many electromagnetic interference(cf figure 4).
- Finally, the manufacturer,who made this DSP computer's card, no longer support it.

As a result, we had the necessity of building a new generation of microphone array.

2 The Idea.

The objective of the third generation array is to place the preamplifier and analog to digital converter close to the microphones in order to reduce any noise due to the environment. So this new generation uses a completely different design as shown in figure 6

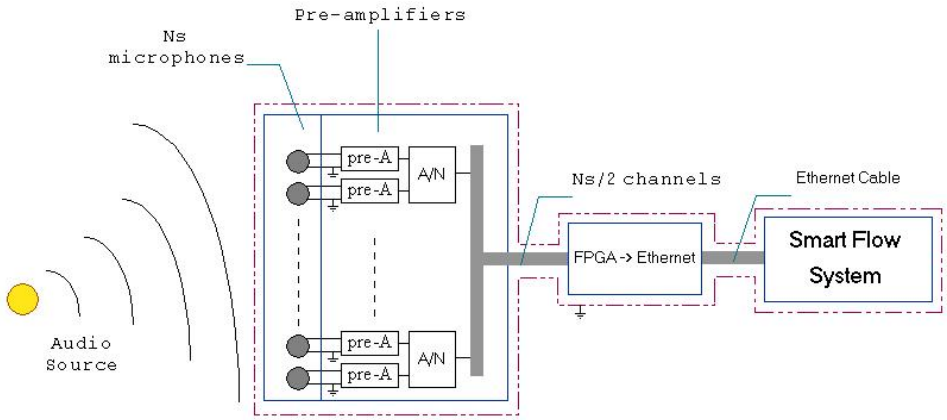


Figure 6: Microphone Array Mark III schematic.

In this new design, the DSP card on the computer becomes obsolete. In order to reduce the complexity of design, I decided to separate the problem in two different boards. The first one, the Microboard is in fact, a sound capture device. The second part is a Motherboard which sends, via an FPGA, data from the Microboard to the Ethernet (cf figure 7).

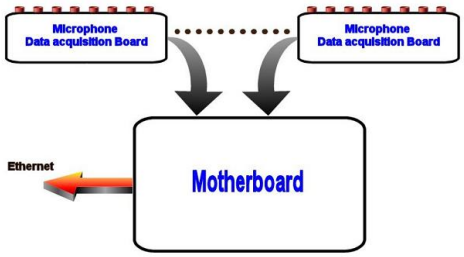


Figure 7: Microphone Array Mark III schematic.

Now we are going to see in more details these two boards.

3 The Hardware.

3.1 Microboard

3.1.1 An overview

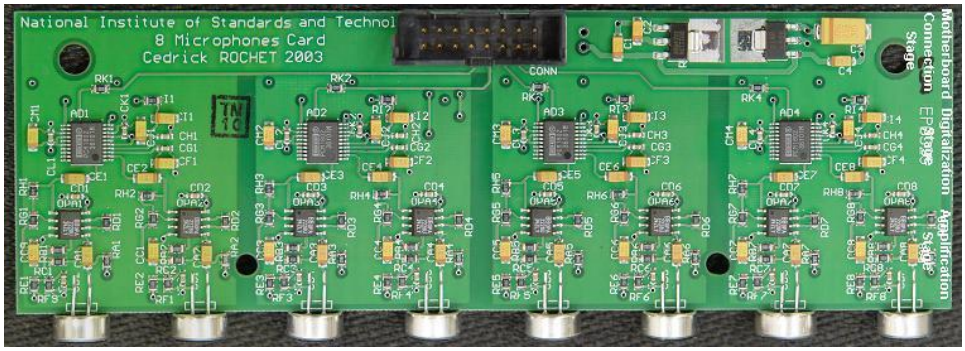


Figure 8: Microboard photo with presentation of the different stages.

On the figure 8 you can see clearly the disposition of the different stages on the board.

In order to go deeper in the explanations, we are going to divide this board into three parts:

- the microphone amplification stage,
- the digitalization stage,
- and the motherboard connection stage.

Our major objective is to reduce the noise factor and have the best performances at the lowest cost for the converter.

We made the choice of working in SMT in order to have very small traces. This will eliminate much electromagnetic interference particularly since the microphones are high impedance and long conductor runs incur noise penalty.

An other choice was to make the Microboard in four layers. This kind of technology is far more better to get less interferences since the GND and VCC planes are close to the signal planes. The difference of price between a 2 and a 4 layer board has been considered but the difference in big quantities wasn't sufficient to really reduce the overall price.

Another choice was to put 8 microphones on the same board. This is based on the fact that 64 is a multiple of 8, 4 digitizers can share the same clocks and power supply and the size of the board is reasonable.

Each microphone is spaced with 2cm. The value has been chosen by it's adequacy to human voice frequency range. So the width of the microphone array is about 130cm.

3.1.2 Microphone amplification stage

In this part we are going to have a closer look of this stage.

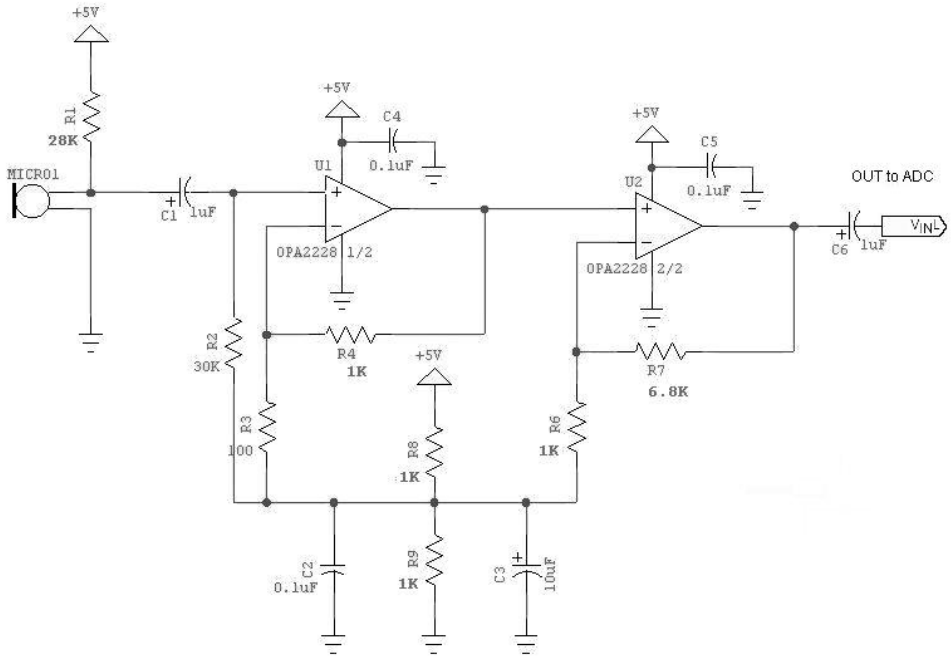


Figure 9: Schematic of the amplification stage for the left channel.

How it Works

As you can see from the figure 9, the microphone creates the sound signal which after is amplified.

The sensors used for the previous microphone array generations are Panasonic WM-52B electret microphones (`http://www.`

mci.panasonic.co.jp/english/prdct/ecm/pdf/wm-53bs_{w}m-52b_{5}4b.pdf). They offer a good frequency response for human voice frequency range.This microphone needs a phantom power of 1.5V made by the resistor R1.

The capacitor C1 is cutting the DC part of the signal entering the operational amplifier. I used an OPA2228 series op amps because it's the combination of low noise and wide bandwidth with high precision. You can find its specification here: http://www-s.ti.com/sc/psheets/sbos110/sbos110.pdf. The OPA2228 is decoupled with the capacitor C4 (C4 and C5 are the same capacitor). The R2 resistor is here to get the current out of the entry of the OPA2228. The gain is done in two stages: the first one has a gain of 100 done by R3 and R4 and the second one done by R6 and R7 has a gain of 6.81.

Since the OPA is powered by the GND and the +5V, a phantom GND at 2.5V was created: it's done by the resistor R9 and R8. The capacitor C2 and C3 decoupled it.

The last capacitor C6 removes again the DC component of the signal before entering the A/D.

This amplification stage is done in double for one stereo digitizer. So you have a left ($V_{IN}L$) and a right ($V_{IN}R$) channel entering the digitizer.

3.1.3 Digitalization stage

The second stage of the Microboard is made mainly by the PCM1802 (cf figure 10).

I have chosen the Analogic/Digital converter PCM1802 because of its 24-Bit Stereo capability, low-cost, single chip stereo Analog-to-Digital Converter (ADC) with single-ended analog voltage inputs. The PCM1802 uses a delta-sigma modulator with 64x or 128x oversampling, a digital decimation filter, and a serial inter-face which supports Slave mode operation and two data formats. The reason of a low cost is his range of work adapted to the sound. You can find its specification here: http://www-s.ti.com/sc/ds/pcm1802.pdf.

In that part, I just followed the indication given by the documentation in the worst case scenario. So for more information you can refer to the PDF file given previously except the 100Ω resistor for the signal integrity of DOUT because of the transmission line between the PCM1802 and the FPGA.

Three clocks LRCK, BCK and SCKI are given by the Motherboard:

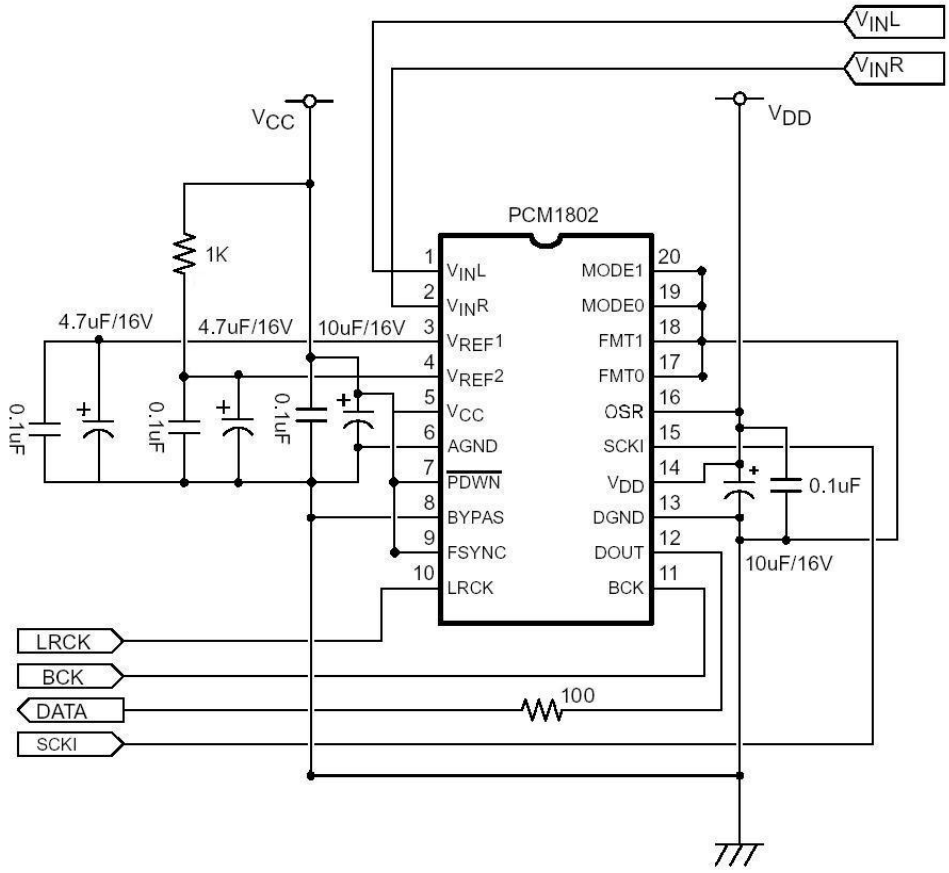


Figure 10: Schematic of the digitalization stage.

- LRCK is the sampling frequency,
- BCK is the bit clock,
- SCKI is the system clock of the PCM1802.
- DOUT is the data output of PCM1802 based on the three previous clocks.

The setup of the digitizer is done so that it's in a slave mode (MODE1=0, MODE0=0, FSYNC=1), left justified, 24 bit (FMT1=0 and FMT0=0). In this model, I am not using the power-down mode controlled by the FPGA ($\overline{PDWN} = 1$) but I am putting the low-cut filter mode (BYPASS=0) and

the oversampling ratio at *128 F_S (OSR=1).

3.1.4 Motherboard connection stage

The connection to the Motherboard is done by a connector which pinout is presented in the figure 11.

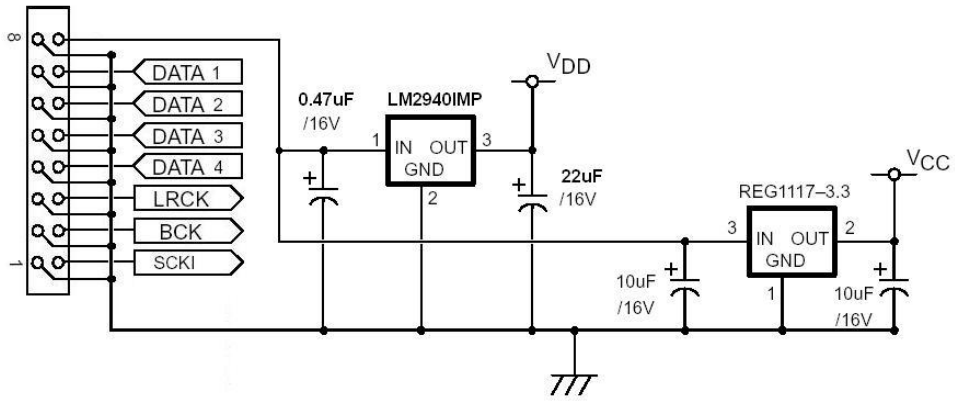


Figure 11: Schematic of the Motherboard connection stage.

The four DATA lines are the outputs of the four PCM1802 of the board.

The Microphone amplification stage needs a 5V power supply but the Digitalization stage needs a 5V power supply and a 3.3V power supply. So after test I chose these two regulators:

- the LM2940 for the 5V `http://www.national.com/ds/LM/LM2940.pdf`,
- the REG1117 for the 3.3V `http://www-s.ti.com/sc/psheets/sbvs001b/sbvs001b.pdf`.

The connector as been designed for a flat ribbon cable with a GND wire between each signal or power supply wire.

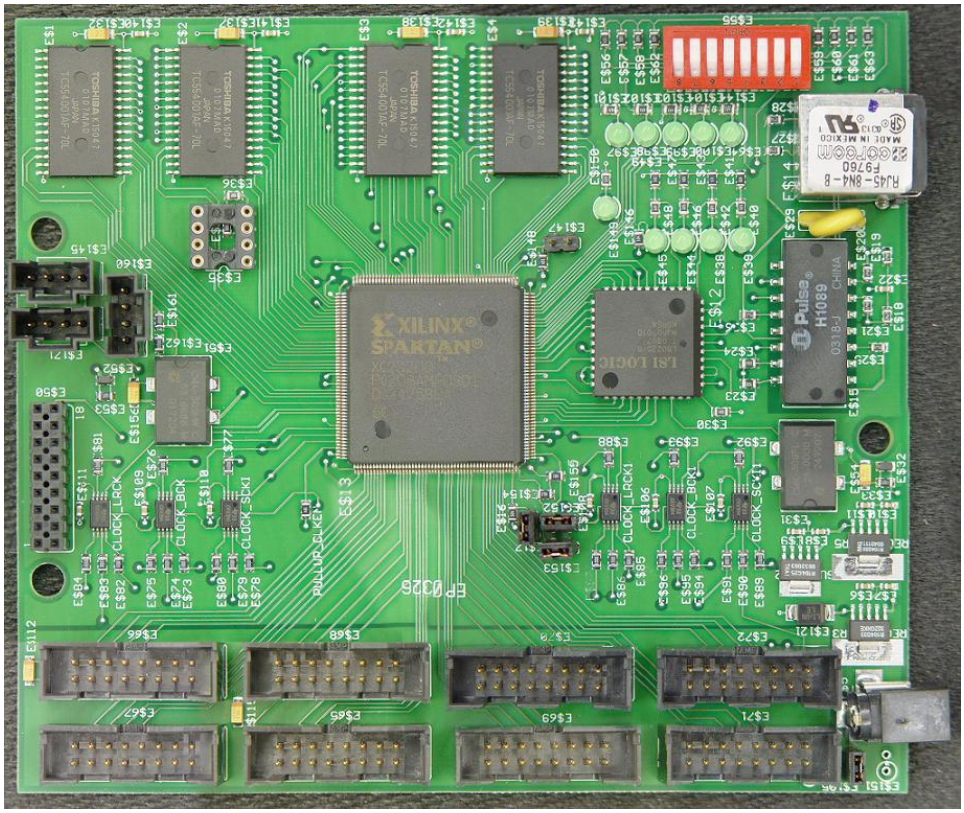


Figure 12: Photo of the Motherboard.

3.2 Motherboard

3.2.1 An overview

As you can see on the figure 12, the FPGA is the gathering part of the project.

The cables at the top of the figure 12 are the data collection cables connected to the 8 Microboards.

The red DIP is here to fix the MAC address of the microphone array.

The LEDs are here to give the status of the microphone array but we will see that deeper later.

Form the figure 13 we can divide this board into seven parts:

- the power stage (2.5V, 3V, 5V),

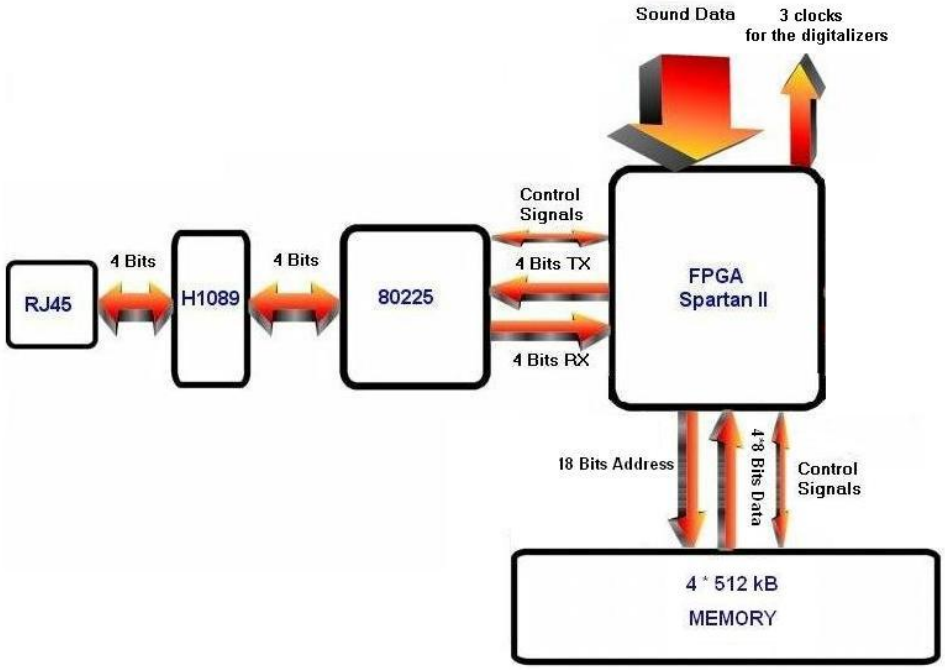


Figure 13: Motherboard schematic overview.

- the PROM stage,
- the clock stage (25MHz and 33.8688MHz),
- the data collection stage,
- the FPGA stage,
- the memory stage,
- the Ethernet stage (80225, H1089 and RJ45),

This board like the Microboard is done in 4 layers to reduce it's size and have a better signal integrity.

3.2.2 Power stage

At the level of the power supply, I didn't use the 110V as main power supply because the project is open-source and safety considerations are important. So if our users make the same one, the DC 9V from an external

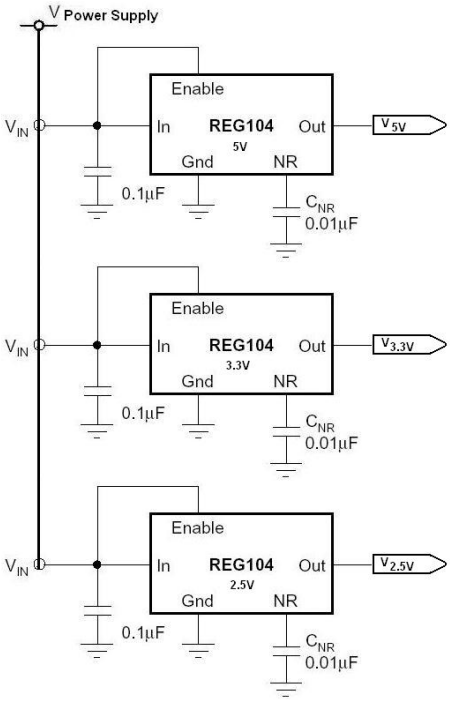


Figure 14: Schematic of the power stage.

transformer will be safer. The TOSHIBA memory used and the 2 oscillators require a voltage of 5V.

But the FPGA needs 2.5V and 3.3V(I/Os). So after some research, I found the Burr-Brown's REG104. The REG104 is a family of low-noise, low-dropout linear regulators with low ground pin current. The specification was better than everything else so I used it in 3 different versions for the 3 different voltage presented in the figure 14.You can find its specification here: `http://www-s.ti.com/sc/psheets/sbvs025b/sbvs025b.pdf`.

3.2.3 PROM stage

In order to have an automatic programming of the FPGA at startup, I had a PROM in which the VHDL program is burned. We will see later about this program.

As told in the documentation of the FPGA, the PROM chosen was the

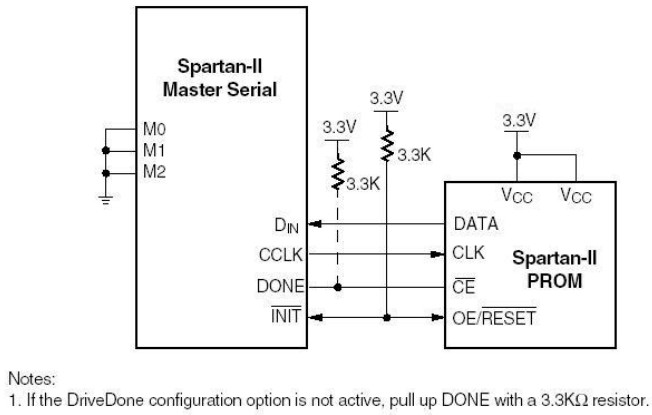


Figure 15: Schematic of the PROM stage.

XC17S200 because of its memory size of 2Megs. You can find its specification here: `http://www.xilinx.com/bvdocs/publications/ds078.pdf`.

3.2.4 Clock stage

This stage is used two times in the design as presented in figure 16:

- to give the main clock for the digitalization to the FPGA which creates three clocks form it (LRCK, BCK, SCKI),
- to give the 25MHz clock to the Ethernet as we will see later.

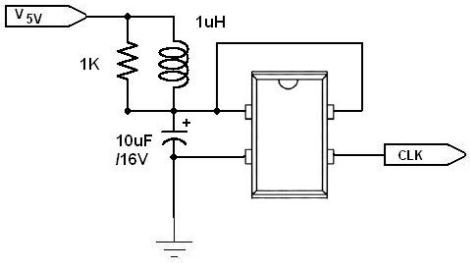


Figure 16: Schematic of the clock stage.

The RLC system is here to clean the bounce created by the oscillator.

3.2.5 Data collection stage

On the Motherboard there is 8 connectors and the clocks signals created by the FPGA are amplified 8 times for the 8 Microboards (cf figure 17).

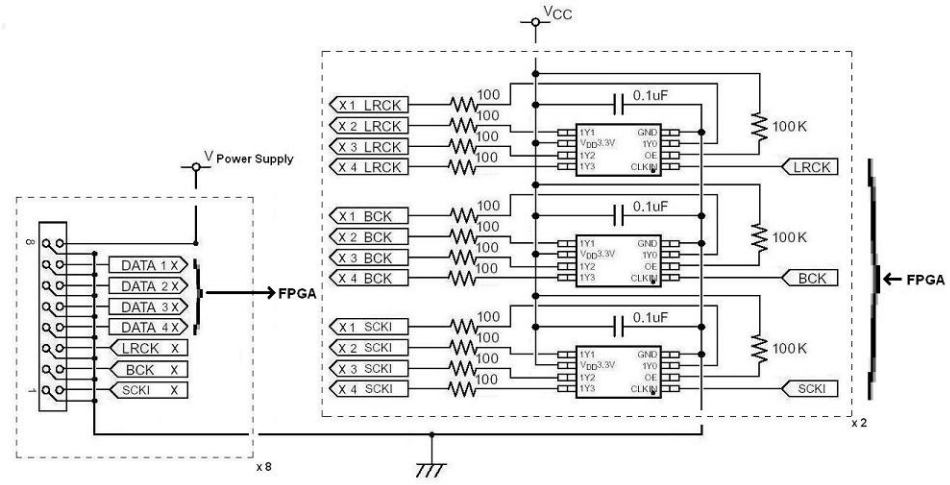


Figure 17: Schematic of the data collection stage.

The data lines are directly connected to the I/O pins of the FPGA.

The output of the FPGA is not appropriate to drive the clocks of 32 converters, so I used two clock drivers with a low-skew propagation delay: the CDCV304. You can find its specification here: `http://www-s.ti.com/sc/ds/cdcv304.pdf`.

The six CDCV304 are decoupled with six $0.1\mu F$ capacitors.

As previously the 100Ω resistors are for the signal integrity of the clocks because of the transmission line between the PCM1802 and the FPGA.

3.2.6 FPGA stage

My choice directly go to the Xilinx's FPGA because of my good engineering experience in the past with it. I oriented my research for the right FPGA to the Spartan-II 2.5V FPGA Family.You can find its specification here: `http://www.xilinx.com/partinfo/ds001.htm`.

The Spartan ® -II 2.5V Field-Programmable Gate Array family gives users high performance, abundant logic resources, and a rich feature set, all at an exceptionally low price. The six-member family offers densities rang-

ing from 15,000 to 200,000 system gates. System performance is supported up to 200 MHz. Spartan-II devices deliver more gates, I/Os, and features per dollar than other FPGAs by combining advanced process technology with a streamlined Virtex-based architecture. Features include block RAM (to 56K bits), distributed RAM (to 75,264 bits), 16 selectable I/O standards, and four DLLs.

For soldering reason I chose the PQ208 package: the only package I can solder without using a computer controlled machine.

3.2.7 Memory stage

The connection between the FPGA and the memory is quite simple (cf figure 18).

The four memories are acting like one with 32 bits data width and with a depth of 18 bits of address. This configuration gives us 2Mbytes of memory as buffer. Compare to our actual computers it doesn't seem a lot but it's quite enough for our real-time purpose.

The memory used is from TOSHIBA. It's speed of 70ns was exactly what we where looking for. Here is the data-sheet: `http://www.toshiba.com/taec/components/Datasheet/4001a.pdf`.

As you can see we are decoupling each memory with a $0.01\mu F$ and a $10\mu F$ capacitors. Even though this memory is working in 5V, the FPGA is 5V compliant and 3.3V output of the FPGA is more than the minimal accepted by the memory.

3.2.8 Ethernet stage

In order to interface to the Ethernet, different circuit for the Ethernet LAN Controller like the Quality Semiconductor QS6611, the National Semiconductor DP83840A MII, ICS1890 MII, the Mitel Semiconductor's NWK914, the TDK Semiconductor 78Q2120 and the the SEEQ Technology 80225 10/100 BASE-TX physical layer were available. Finally, the last one was chosen because of the price and quality of the documentation available.

The schematic on figure 19 comes from the documentation of the 80225:

The 80225 gets it's 25 MHz clock form the clock stage that we spoke about previously.

The 80225 is a highly integrated analog interface IC for twisted pair Ethernet applications. The 80225 can be configured for either (100Base-TX) or 10 Mbps (10Base-T) Ethernet operation. The 80225 consist of 4B5B/Manchester encoder/decoder, scrambler/descrambler, transmitter

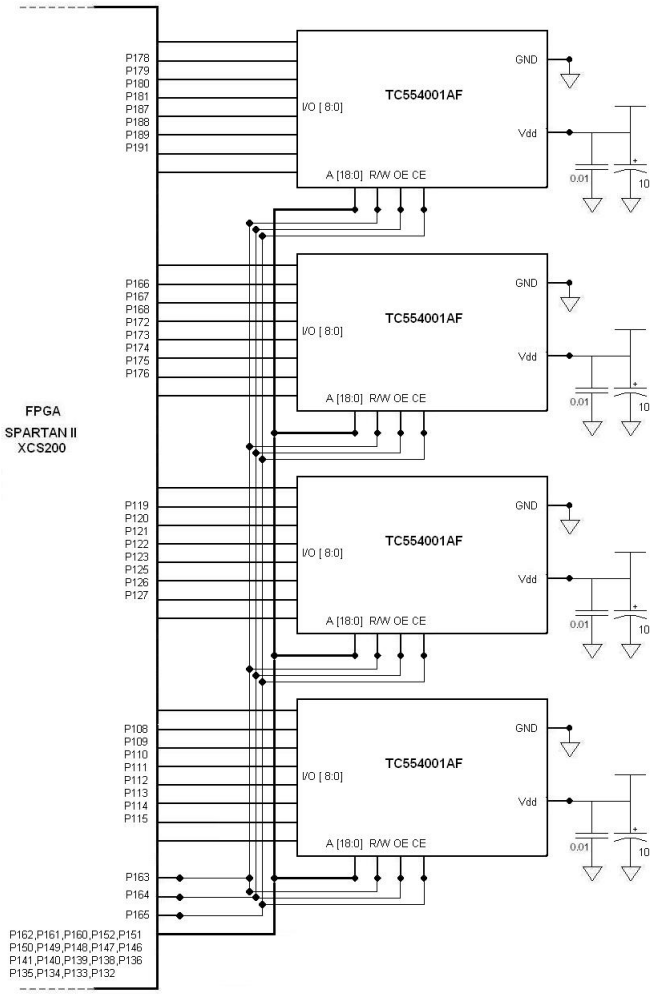


Figure 18: Schematic of the memory stage.

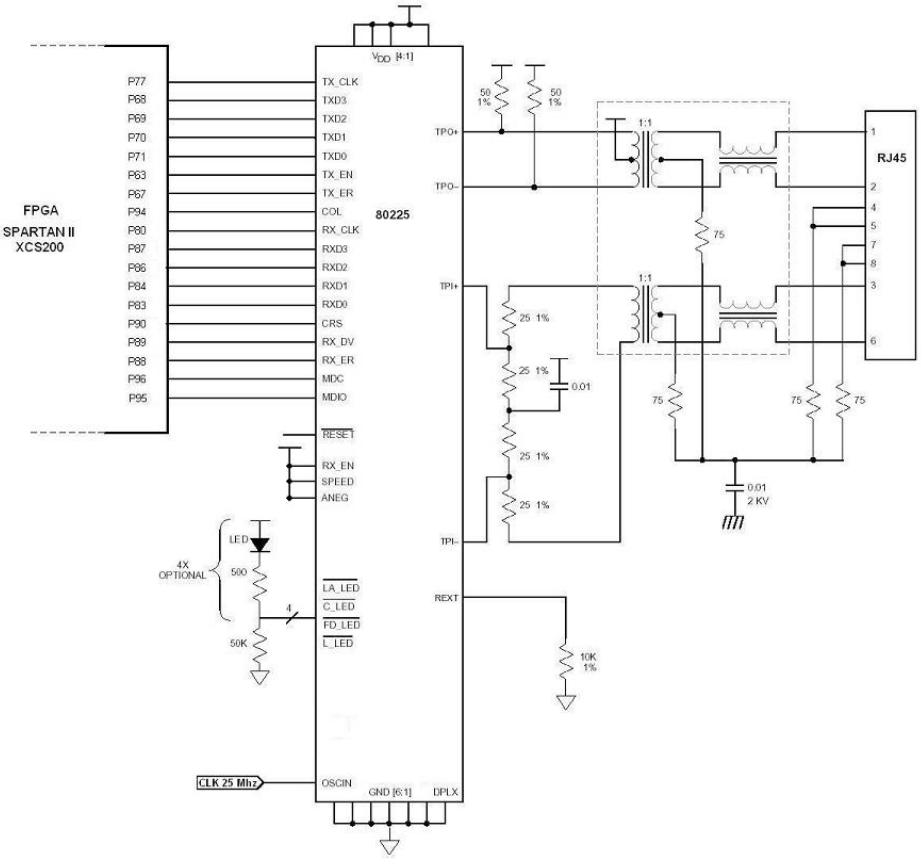


Figure 19: Schematic of the Ethernet stage.

with wave shaping and output driver, twisted pair receiver with on chip equalizer and baseline wander correction, clock and data recovery, Auto Negotiation, controller interface (MII), and serial port (MI).You can find its specification here: `http://www.lsilogic.com/techlib/techdocs/networking/80225.pdf`.

With the 80225, it's advised to use a pulse's H1089. You can find its specification here: `http://www.pulseeng.com/pdf/4303.pdf`.

4 The VHDL program for the FPGA.

In this section I am going to explain my software design in VHDL. But first of all here is a summary description of this language.

4.1 The VHDL

The name of VHDL is the result of VHSIC(Very High Speed Integrated Circuits) Hardware Description Language. So the VHDL is not a programming language but a hardware description language.It is a formal notation intended for use in all phases of the creation of electronic systems. Because it is both machine readable and human readable, it supports the development, verification, synthesis and testing of hardware designs, the communication of hardware design data, and the maintenance, modification, and procurement of hardware.
Historically, the VHDL is based on Ada(which is Pascal based) and was develop by TI, IBM, Intermetrics in 1983 as and became IEEE Std 1076-1987 and 1993.

So I'm going to describe the different modules used in my design and after we will have a gathering schematic of the modules. but first a description of UDP.

4.2 UDP

A frame on the Ethernet network is composed of different messages encapsulated each one in an other. So in order to send some data through the Ethernet to a computer I have to respect some protocol.

Figure 20: Different fields of an UDP message.

The preamble is 64 bits long (1 0 1 0 1 0 ..... 1 0 1 0 1 1) and corresponds to the period of synchronization between the two physical layer devices. This part of a message is taken care of in the tx_frame module.

The CRC32 is 64 bits long and has a value calculated form the rest of the message except the preamble. This part of a message is taken care of in the CRC32 module.

The main part of the message (MAC header, IP header, UDP header and DATA) is taken care of in the modules capture_udp_frame, bootp_frame and response_status_udp_frame. For more information about the different fields, you can refer to the RFC's or on this website:`http://www.networksorcery.com/enp/topic/ipsuite.htm`.

4.3 The Main VHDL program

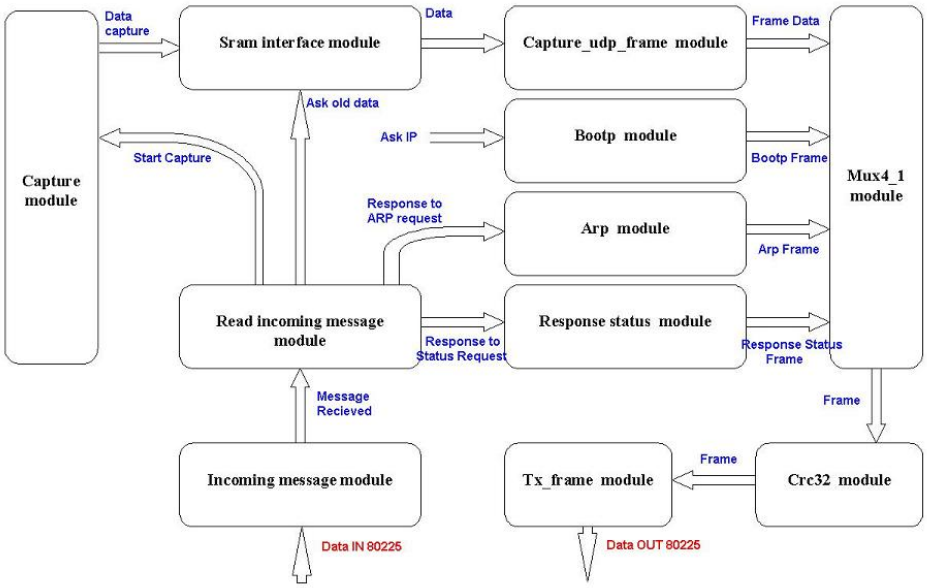


Figure 21: Gathering schematic of all the modules.

In VHDL, there is one main module who contains all the other ones. The figure 21 represents an overview of the eleven submodules interactions.

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

- capture module,
- sram interface module,
- capture_udp_frame module,

- bootp module,
- arp module,
- 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 21.

4.4 The capture module

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

The figure 22 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.

In this module, there is two main clocks: *cap_clk* and *cap_clk_slave*. The *cap_clk* is provided by the oscillator on the Motherboard. The *cap_clk_slave* is provided by an other Motherboard through the synchronization cable.

First in master mode, *start_capture* is a signal used at the beginning to tell when the capture shall start. The signal *low_reset* is used to initialize the state machines in the module. 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

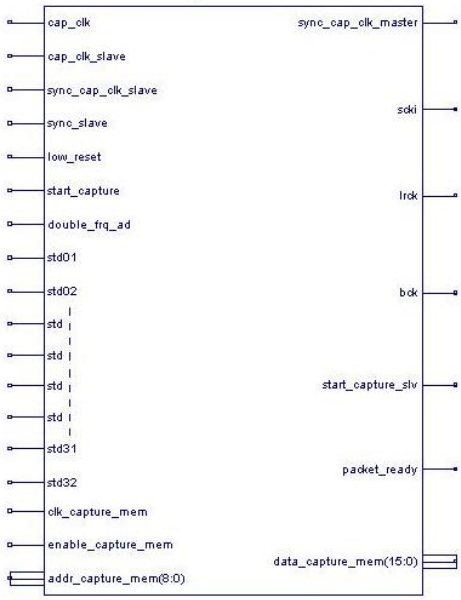


Figure 22: The capture module.

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 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 tops:

- top to reset the clock divider in the FPGA,
- top to start capture,
- top to end capture.

The return of the converters, *stdxx*, is serial so this module take care of placing the different streams like describes in the figure 23.

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*

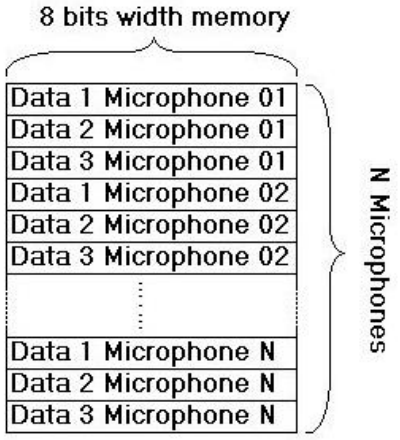


Figure 23: Data organization in the memory.

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.

In slave mode, the clock *cap_clk_slave* is used as the main clock. The slave mode is activated when *sync_slave* is up. The synchronization signal is received through *sync_cap_clk_slave* and do the same thing as in master mode except that it's based on an outside signal. The signal *start_capture_slv* goes up in slave mode to activate the sram interface module, the capture_udp_frame module.

4.5 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:

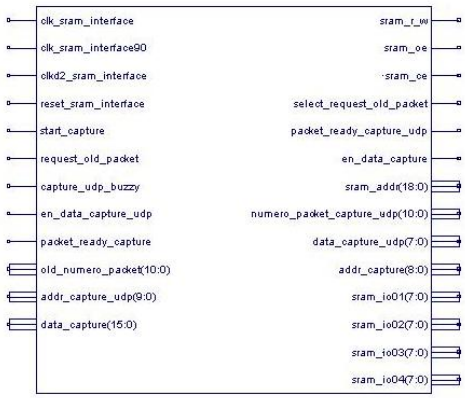


Figure 24: The sram interface module.

- *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.

4.6 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.

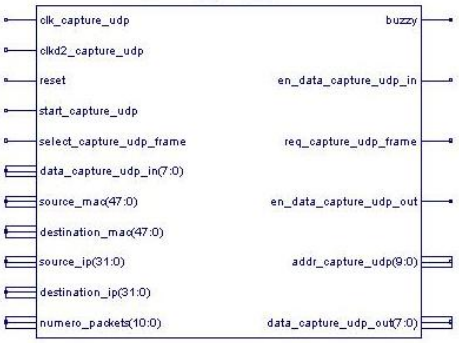


Figure 25: The capture_udp_frame module.

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 26: The data frame.

The data frame is constituted of different fields (cf figure 26) 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.

4.7 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.

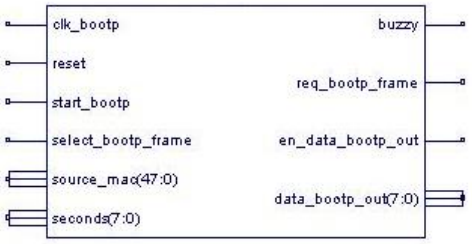


Figure 27: The bootp module.

4.8 The arp module

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

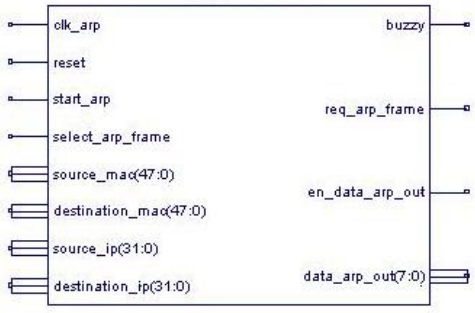


Figure 28: The arp module.

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.

4.9 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.

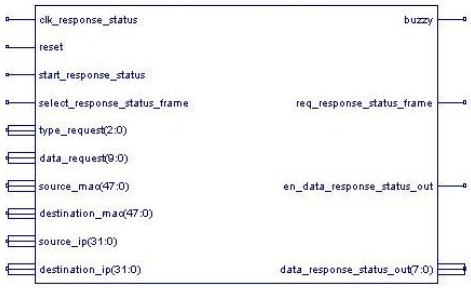


Figure 29: The response status module.

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.

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.

4.10 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.

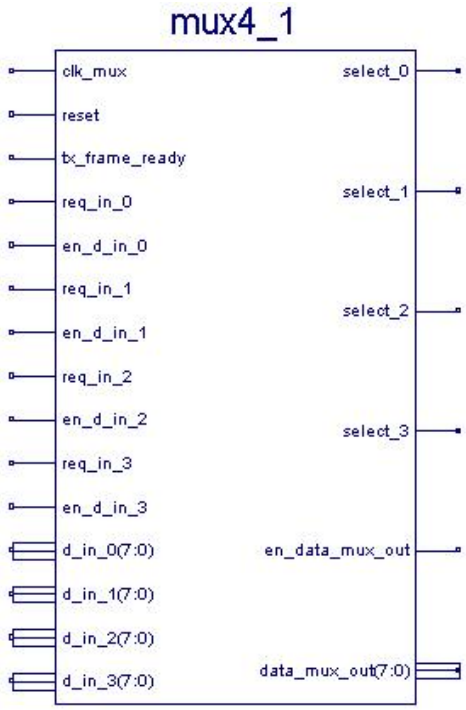


Figure 30: The mux4_1 module.

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.

4.11 The CRC32 module

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

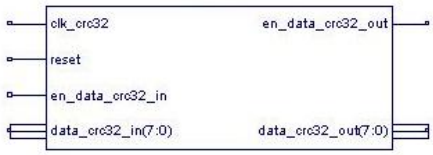


Figure 31: The CRC32 module.

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.

4.12 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.

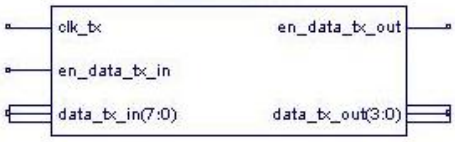


Figure 32: The tx_frame module.

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*.

4.13 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.

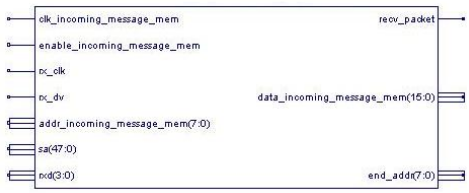


Figure 33: The incoming message module.

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)*.

4.14 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)*.

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)*,

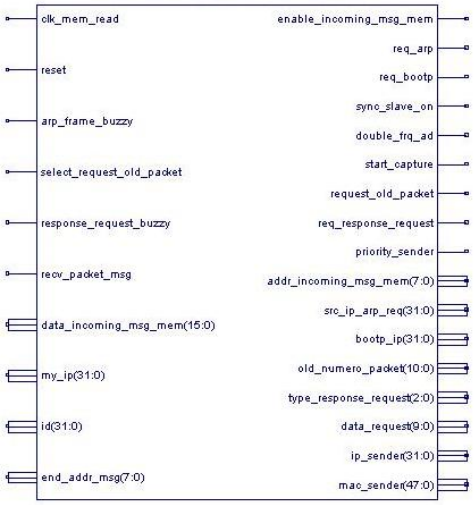


Figure 34: The read incoming message module.

- 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(2:0)*= "010", *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)*= "011", *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(2:0)*= "101", *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(2:0)*= "110", *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(2:0)*= "111", *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(2:0)*= "110", *data_request(9:0)*="0001101110" (binary for 'n').

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.

4.15 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.

5 The kit Microphone Array Mark III.

This section is dedicated to the kit industrialized by the company Bench-mark Electronics, Inc. (BEI) located in Manassas, Virginia.

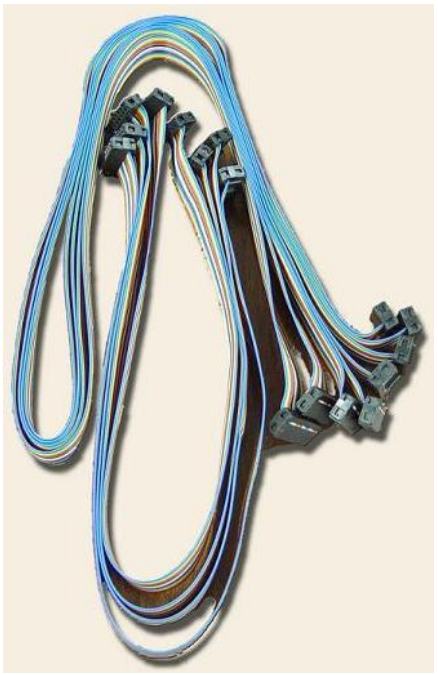
5.1 The Microphone Array mark III kit

The kid is composed of:

- one 9V power supply (a) figure 35,
- 75 feet of CAT5-RJ45 cable(b) figure 35,
- 8 connection cables between the Microboards and the Motherboard(a) figure 36,
- 1 synchronization cable(b) figure 36,
- 1 Motherboard(a) figure 37.
- 8 Microboards(b) figure 37,

(a) 9V power supply (b) 75 feet RJ45 cable

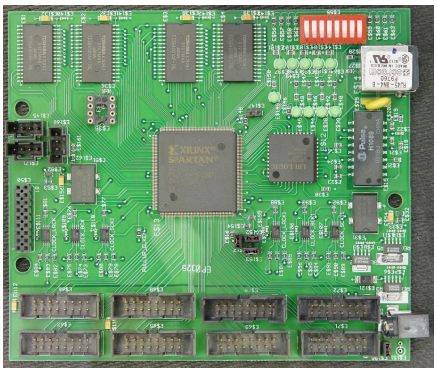
Figure 35: The inputs cables.



(a) 8 connection cables

(b) 1 synchronization cable

Figure 36: The inside cables.



(a) 1 Motherboard

(b) 8 Microboards

Figure 37: The boards.

5.2 Hardware setup

5.2.1 Motherboard Overview

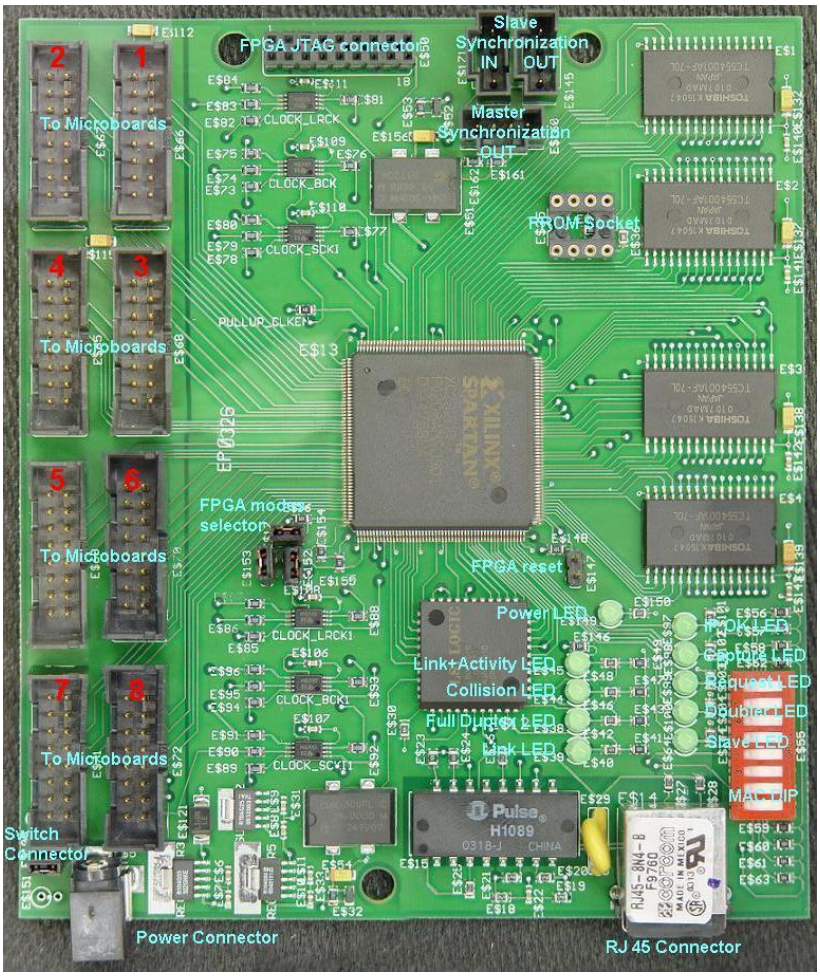


Figure 38: Motherboard detailed overview.

The following tables presents the description of each named object on the figure 38.

Name	Description
Power LED	Indicate if power is ON or OFF. 0 = OFF 1 = ON
Link+Activity LED	Indicate the occurrence of link or Activity on the Ethernet. 0 = Link Detect Blink = Link Detect and Activity 1 = No Link Detect
Collision LED	Indicate the occurrence of a collision on the Ethernet. 0 = Collision Detect 1 = No Collision
Full Duplex LED	Indicate if the Motherboard is in Full Duplex Mode on the Ethernet. 0 = Full Duplex Mode Detect with Link Pass 1 = Half Duplex
Full Duplex LED	Indicate if the Motherboard is in Full Duplex Mode on the Ethernet. 0 = Full Duplex Mode Detect with Link Pass 1 = Half Duplex
Link LED	Indicate a 10/100 Mbps Detect on the Ethernet. 0 = 100Mbit Mode Detected with Link Pass 1 = 10Mbit Mode Detected
IP OK LED	Indicate if the Microphone Array Mark III has an IP address Ethernet. 0 = No IP address and waiting an BOOTP reply 1 = Has an IP address
Capture LED	Indicate if the Microphone Array Mark III is capturing sound. 0 = No capture 1 = Capturing
Request LED	Indicate if the Microphone Array Mark III is sending old packets. 0 = Not sending old packets 1 = Sending old packets

Name	Description
Doubler LED	Indicate if the Microphone Array Mark III is capturing in 22050Hz or 44100Hz. 0 = 22050Hz mode 1 = 44100Hz mode
Slave LED	Indicate if the Microphone Array Mark III is in slave mode. 0 = Master mode 1 = Slave mode

Name	Description
Switch Connector	2 Pins for a power switch. By default, there is a jumper
Power Connector	Connector for the power supply.
RJ45 Connector	Connector for the Ethernet cable.
MAC DIP	Dip switch to setup the MAC address of the Microphone Array Mark III.

Name	Description
FPGA reset	2 Pins for a reset button. It's active only when the power is ON
FPGA modes selector	Select the different modes of the FPGA configuration: parallel, serial,JTAG. By default, the 3 2-pins connectors have jumpers, to allow PROM programing. For other modes see the FPGA documentation.
Prom Socket	Socket for the PROM. The PROM should be inserted has shown on the socket
FPGA JTAG connector	This connector is used for a JTAG programming of the FPGA from a computer through the XILINX'S cable: Parallel Cable III, Model DLC5.

Name	Description
Master Synchronization OUT	Connector for the synchronization cable. The other end of the synchronization cable should be connected to Slave Synchronization IN. The synchronization signals are going out of the Motherboard ,in Master Mode, through this connector
Slave Synchronization IN	Connector for the synchronization cable. The other end of the synchronization cable should be connected to Slave Synchronization OUT or Master Synchronization OUT. The synchronization signals are going in the Motherboard, in Slave Mode (to be detected), through this connector
Slave Synchronization OUT	Connector for the synchronization cable. The other end of the synchronization cable should be connected to Slave Synchronization IN. The synchronization signals are propagate from the Slave Synchronization IN to another Motherboard through this connector

Name	Description
To Microboards 1	Connector for first Microboard.
To Microboards 2	Connector for second Microboard.
To Microboards 3	Connector for third Microboard.
To Microboards 4	Connector for fourth Microboard.
To Microboards 5	Connector for fifth Microboard.
To Microboards 6	Connector for sixth Microboard.
To Microboards 7	Connector for seventh Microboard.
To Microboards 8	Connector for eighth Microboard.

5.2.2 PROM setup

The PROM provided is normally not programed. The program can be found at this URL: `http://www.nist.gov/smartspace/toolChest/cmaiii/`

To configure the PROM, the Xilinx Serial PROM programmer from Roman-Jones, Inc. is adequate. It can be bought at `http://www.digikey.com`.

The PROM used is referenced as XC17S200APD8C.

5.2.3 MAC address setup

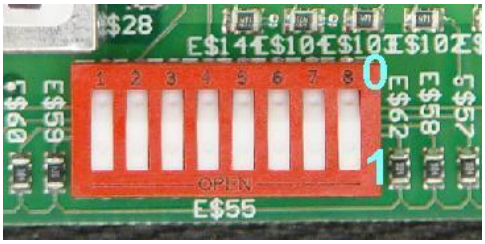


Figure 39: The MAC dip.

Fixed by the VHDL programm	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
0001 0000 0000 0000 0000 0000 0000 0000 0000 0011	x	x	x	x	x	x	x	x

The RED DIP switch (cf figure 39)setups the MAC address. A MAC address is made of 48 bits. The DIP switch fixed the last 8 ones and the VHDL program fixed the 40 remaining one. By default, as one the photo, the MAC address is : 10:00:00:00:03:00.

Examples:

- If the switch 1 on the DIP is changed, compared to the photo, the MAC address becomes : 10:00:00:00:03:80.
- If the switch 8 on the DIP is changed, compared to the photo, the MAC address becomes : 10:00:00:00:03:01.
- If all the switches on the DIP are changed, compared to the photo, the MAC address becomes : 10:00:00:00:03:FF.

Configuration Modes	Preconfiguration Pulls-ups	E$153	E$17	E$152
Master Serial mode	Yes (default)	0	0	0
	Yes	0	0	1
Slave Parallel mode	Yes	0	1	0
	No	0	1	1
Boundary-Scan mode	Yes	1	0	0
	No	1	0	1
Slave Serial mode	No	1	1	0
	Yes	1	1	1

5.2.4 2-Pin heads setup

5.2.4.1 FPGA mode The Spartan-II FPGA supports the following four configuration modes:

- Slave Serial mode
- Master Serial mode
- Slave Parallel mode
- Boundary-scan mode

The Configuration mode pins (E$153, E$17, E$152) select among these configuration modes with the option in each case of having the IOB pins either pulled up or left floating prior to configuration. The selection codes are listed in the previous table. Configuration through the boundary-scan port is always available, independent of the mode selection. Selecting the boundary-scan mode simply turns off the other modes. The three mode pins have internal pull-up resistors, and default to a logic High if left unconnected.

The default mode enables a FPGA configuration form the PROM or the JTAG connector but to do both will results in problems... If you do a JTAG configuration, you have to remove the PROM.

5.2.4.2 Reset The shortcut of the 2-Pin head E$147 will result in the RESET of the FPGA from any programs like a "Power off - Power on" cycle. This 2-Pin head E$147 is the option of a live reset on the box.

5.2.4.3 Power Switch The shortcut of the 2-Pin head E$105 will result in the POWER ON of the Microphone Array. This 2-Pin head E$105 is the option of a power switch on the box.

5.3 Cable Connections setup

5.3.1 Motherboard-Microboard connections

The cables presented on the figure 36 (a) are connecting the Motherboard to the 8 Microboards. These cables have a socket connector with central polarizing key because a mismatch connection would led to a shortcut.

The figure 38 have numbered sockets:

Name	Description
To Microboards 1	Connector for first Microboard.
To Microboards 2	Connector for second Microboard.
To Microboards 3	Connector for third Microboard.
To Microboards 4	Connector for fourth Microboard.
To Microboards 5	Connector for fifth Microboard.
To Microboards 6	Connector for sixth Microboard.
To Microboards 7	Connector for seventh Microboard.
To Microboards 8	Connector for eighth Microboard.

5.3.2 Synchronization connections

The Motherboard (cf figure 38) has 3 synchronization connections detailed in this table:

With the cable provided (cf figure 36 (b)) the synchronization is possible with a least 4 Motherboards in daisy chain.To do with even more Motherboards, you might have to upgrade the cable to an other one of better quality.

The delay introduce between two synchronized boards is about 3 ns ($3 * 10^{-9} s$). The delay is the time needed to propagate along the about 30cm of the cable. Compared to the frequency of 22050Hz of the converter the error is about 0.01%.

6 Linux Tuning.

6.1 BOOTP server

Just after being power up, the microphone array is doing a BOOTP request on the network. You need on a computer, in direct liaison or connected through a switch, a DHCP daemon. The Internet Software Consortium DHCP Server, dhcpd, implements the Dynamic Host Configuration Protocol (DHCP) and the Internet Bootstrap Protocol (BOOTP). DHCP allows hosts on a TCP/IP network to request and be assigned IP addresses, and also to discover information about the network to which they are attached. BOOTP provides similar functionality, with certain restrictions.

The DHCP protocol allows a host which is unknown to the network administrator to be automatically assigned a new IP address out of a pool of IP addresses for its network. In order for this to work, the network administrator allocates address pools in each subnet and enters them into the ***/etc/dhcpd.conf*** file. On startup, dhcpd reads the dhcpd.conf file and stores a list of avail- able addresses on each subnet in memory.

To be able to do that you have to be logged as administrator on the machine.

After, you need to create the file ***/etc/dhcpd.conf*** Here is a copy of the file ***dhcpd.conf***

```
option subnet-mask 255.255.255.0;
option broadcast-address 10.0.0.255;
option routers 10.0.0.1;
option domain-name-servers 10.0.0.1;
option domain-name "array.org";

ddns-update-style ad-hoc;

subnet 10.0.0.0 netmask 255.255.255.0 {

        host array2 {
                hardware ethernet 10:00:00:00:03:00;
                fixed-address 10.0.0.2;
        }

        host array3 {
                hardware ethernet 10:00:00:00:03:CA;
                fixed-address 10.0.0.3;
        }

        host array4 {
                hardware ethernet 10:00:00:00:03:FF;
                fixed-address 10.0.0.4;
        }
}
```

In order to activate the daemon you have to enter the following command:

dhcpd

You are know all set and the green LED closest to the SRAM memory should light...normally...

You can notice that all the hardware Ethernet (MAC) addresses are the same : 10:00:00:00:03:XX This is because XX is given by the red DIP switch: XX goes from 00 to FF. This enables us to put up to 256 microphone arrays on the same network...

6.2 Kernel tuning

The amount of data created by the microphone array system is about 4.4MBytes a second if the frequency doubler is not activated otherwise, it goes up to 10 MBytes a second. At some point you might experience on a normal Linux computer some loss of packets.... To resolve the problem, there is two ways:

- tune the Linux kernel,
- or upgrade the machine.

6.2.1 sysctl.conf

Sysctl.conf is a simple file containing sysctl values to be read in and set by sysctl. Sysctl is used to modify kernel parameters at runtime. The parameters available are those listed under /proc/sys/ Here is a copy of the file in:***/etc/sysctl.conf***

```
# Kernel sysctl configuration file for Red Hat Linux
#
# For binary values, 0 is disabled, 1 is enabled.  See sysctl(8) and
# sysctl.conf(5) for more details.

# Controls IP packet forwarding
net.ipv4.ip_forward = 0

# Controls source route verification
net.ipv4.conf.default.rp_filter = 1

# Controls the System Request debugging functionality of the kernel
kernel.sysrq = 1

# Controls whether core dumps will append the PID to the core filename.
# Useful for debugging multi-threaded applications.
kernel.core_uses_pid = 1
```

```
#increase linux TCP bufferlimits
net.core.rmem_max = 8388608
net.core.wmem_max = 8388608
net.core.rmem_default = 2097152
net.core.wmem_default = 2097152

#increase linux autotuning TCP buffer limits
net.ipv4.tcp_rmem = 4096 2097152 8388608
net.ipv4.tcp_wmem = 4096 2097152 8388608
net.ipv4.tcp_mem = 8388608 8388608 8388608
```

First you have to be logged as administrator on the machine. After changing your ***/etc/sysctl.conf*** you need to apply the settings. To do so, you have to enter the following command:

sysctl -p

Then relaunch the software you were using and everything should be fine except if you don't meet the minimum requirement.

6.2.2 Minimum requirement

If you don't change ***/etc/sysctl.conf***, you are going to lose packets even with a XP3000+...

So the minimum we used (with ***/etc/sysctl.conf*** changed) was a 1GHz CPU with 1 GBytes of memory and two network card:

- one in direct connection to the microphone array,
- one to send the data the the rest of the network.

The microphone array works perfectly fine on a switch. The problem, in that case, is that everybody on the network have access, through given software, to it and, for example, can listen what is happening in the room.

7 The softwares on the computer.

7.1 The command and control

The program called ***array_simple_controls.c*** is here to give you the basic commands and setup to do in order to give orders to the microphone array.

Here is the code of this file:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <signal.h>
#include <string.h>

#include <sys/poll.h>           //library for the timout on answer
#include <getopt.h>             //library option

static const char *default_dip = "10.0.0.2";
const char *dip;

const char *progname, *shortname;

unsigned char msg[14];
int fd;
struct sockaddr_in adr;
int len;

static int ID_array;
static char PROM_NB[8];

static void print_usage(FILE *stream)
{
  fprintf(stream, "Usage : %s [-h] [-d IP]\n"
          "  -h        --help           Prints this message\n"
          "  -d IP     --dest IP        Listen to this IP (default: %s)\n"
          "\n For more information contact: cedrick.rochet@nist.gov\n"
          "\n---------------------------- Official Notice -----------------------------\n"
          "This software was developed at the National Institute of Standards and      \n"
          "Technology by employees of the Federal Government in the course of their    \n"
          "official duties. Pursuant to Title 17 Section 105 of the United States Code \n"
          "this software is not subject to copyright protection and is in the public   \n"
          "domain.                                                                   \n\n"
          "array_simple_controls is an experimental system as is offered AS IS. NIST   \n"
          "assumes no responsibility whatsoever for its use by other parties, and      \n"
          "makes no guarantees and NO WARRANTIES, EXPRESS OR IMPLIED, about            \n"
          "its quality, reliability, fitness for any purpose, or any other             \n"
          "characteristic. We would appreciate acknowledgement if the software is used.\n\n"
          "This software can be redistributed and/or modified freely provided that any \n"
          "derivative works bear some notice that they are derived from it, and any    \n"
          "modified versions bear some notice that they have been modified from the    \n"
          "original.                                                                   \n"
```

```
          "-------------------------------------------------------------------------\n",
          progname, default_dip);
}

void options(int argc, char ***argv)
{
  static struct option optlist[] = {
    { "help",    0, 0, 'h'},
    { "dest",    1, 0, 'd'},
    { 0,         0, 0, 0  }
  };

  int usage = 0, finish = 0, error = 0;

  dip = default_dip;

  for(;;) {
    int opt = getopt_long(argc, *argv, "h:d:", optlist, 0);
    if(opt == EOF)
      break;
    switch(opt) {
    case 'h':
      usage = 1;
      finish = 1;
      error = 0;
      break;
    case 'd':
      dip = optarg;
      break;
    case '?':
    case ':':
      usage = 1;
      finish = 1;
      error = 1;
      break;
    default:
      abort();
    }
    if(finish)
      break;
  }

  if (usage)
    print_usage(error ? stderr : stdout);

  if (finish)
    exit(error);

  *argv += optind;
}

static void send_msg(void)
{
  if(sendto(fd, msg, len, 0, (const struct sockaddr *)&adr, sizeof(adr)) < 0) {
    perror("sendto");
    exit(1);
```

```
  }
}

static void recieve_msg(void)
{
  struct pollfd pfd;
  int res;
  pfd.fd = fd;
  pfd.events = POLLIN;
  res = poll(&pfd, 1, 100);
  if(!res) {
    fprintf(stderr, "Timeout on answer\n");
    len = 0;
    exit(0);
    return;
  }

  len = recv(fd, msg, sizeof(msg), 0);
}

static int ask_status_slave(void)
{
  int done = 0;
  do {
    msg[0] = 0;
    msg[1] = 2;           //request number

    msg[2] = 0;
    msg[3] = 0;

    len = 4;
    send_msg();

    recieve_msg();
    if(!len)
      continue;
    done = (msg[0] == 2);

  } while(!done);
  printf("Your Microphone Array slave is: %i\n",(int) msg[2]);
  return (int) msg[2];
}

static void slave_on(void)
{
  int done = 0;
  do {
    msg[0] = 0;
    msg[1] = 1;           //request number
    msg[2] = 0xff;
    msg[3] = 0xff;

    len = 4;
    send_msg();

    done = (ask_status_slave() == 1);
```

```
  } while(!done);
}

static void slave_off(void)
{
  int done = 0;
  do {
    msg[0] = 0;
    msg[1] = 1;           //request number
    msg[2] = 0;
    msg[3] = 0;

    len = 4;
    send_msg();

    done = (ask_status_slave() == 0);

  } while(!done);
}

static void ask_id(void)
{
  int done = 0;
  do {
    msg[0] = 0;
    msg[1] = 3;           //request number
    msg[2] = 0;
    msg[3] = 0;

    len = 4;
    send_msg();

    recieve_msg();
    if(!len)
      continue;
    done = (msg[0] == 3);

  } while(!done);
  ID_array = (msg[2])|(msg[3]<<8);
  memcpy(PROM_NB,msg+6,8);
  printf("Capture on %x with PROM %s\n", ID_array, PROM_NB);
}

static int ask_status_capture(void)
{
  int done = 0;
  do {
    msg[0] = 0;
    msg[1] = 5;           //request number
    msg[2] = 0;
    msg[3] = 0;

    len = 4;
    send_msg();

    recieve_msg();
```

```
    if(!len)
      continue;
    done = (msg[0] == 5);

  } while(!done);
  printf("Your Microphone Array capture is: %i\n",(int) msg[2]);
  return (int) msg[2];
}

static void capture_on(void)
{
  int done = 0;
  do {
    msg[0] = 0;
    msg[1] = 4;           //request number
    msg[2] = 0xff;
    msg[3] = 0xff;

    len = 4;
    send_msg();

    done = (ask_status_capture() == 1);

  } while(!done);
}

static void capture_off(void)
{
  int done = 0;
  do {
    msg[0] = 0;
    msg[1] = 4;           //request number
    msg[2] = 0;
    msg[3] = 0;

    len = 4;
    send_msg();

    done = (ask_status_capture() == 0);

  } while(!done);
}

static void ask_old_packet(short packet_number)
{
  // Be careful in using this fuction because it won't stop the normal data packet traffic
  // but will insert the packets asked in the middle of the data traffic...
  // If the microphone is not in capture mode, it will send back a back with msg[2]=n

  char *p;

  p = (char *) &packet_number;

  msg[0] = 0;
  msg[1] = 6;           //request number
  msg[2] = p[0];
```

```
  msg[3] = p[1];

  len = 4;
  send_msg();

  printf("Your packet number is: %i",packet_number);
}

static int ask_status_speed(void)
{
  int done = 0;
  do {
    msg[0] = 0;
    msg[1] = 8;           //request number
    msg[2] = 0;
    msg[3] = 0;

    len = 4;
    send_msg();

    recieve_msg();
    if(!len)
      continue;

    done = (msg[0] == 7);

  } while(!done);

  printf("Your Microphone Array speed is: %i\n",(int) msg[2]);
  return (int) msg[2];
}

static void speed_on(void)
{
  int done = 0;
  do {
    msg[0] = 0;
    msg[1] = 7;           //request number
    msg[2] = 0xff;
    msg[3] = 0xff;

    len = 4;
    send_msg();

    done = (ask_status_speed() == 1);

  } while(!done);
}

static void speed_off(void)
{
  int done = 0;
  do {
    msg[0] = 0;
    msg[1] = 7;           //request number
    msg[2] = 0;
```

```
    msg[3] = 0;

    len = 4;
    send_msg();

    done = (ask_status_speed() == 0);

  } while(!done);
}

static void ask_range_old_packet(short first_packet_number, short last_packet_number)
{
  // Be careful in using this fuction 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 a back with msg[2]=n

  char *pfirst;
  char *plast;

  // conversion of the shorts in chars to fill msg
  pfirst = (char *) &first_packet_number;
  plast  = (char *) &last_packet_number;

  msg[0]=0x00;
  msg[1]=0x09;           //request number
  msg[2] = pfirst[0];
  msg[3] = pfirst[1];
  msg[4] = plast[0];
  msg[5] = plast[1];

  len = 4;
  send_msg();

  printf("Your packet range is: %i - %i \n",first_packet_number,last_packet_number);
}

int main(int argc, char **argv)
{
  int d;
  int choix01, choix02, choix03;

  int number1;
  int number2;

  /* Option processing */
  progname = argv[0];
  shortname = strrchr(progname, '/');
  if(!shortname)
    shortname = progname;
  else
    shortname++;
```

```
options(argc, &argv);

/*socket connection*/
fd = socket(PF_INET, SOCK_DGRAM, 0);
if(fd<0) {
  perror("socket");
  return 0;
  exit(1);
}

memset(&adr, 0, sizeof(adr));
adr.sin_family = AF_INET;
adr.sin_port = htons(32767);
adr.sin_addr.s_addr = INADDR_ANY;
//printf("Bind to IP: %s\n", &adr.sin_addr.s_addr);

if(bind(fd, (struct sockaddr *)&adr, sizeof(adr)) < 0) {
  perror("bind");
  return 0;
  exit(1);
}

memset(&adr, 0, sizeof(adr));
adr.sin_family = AF_INET;
adr.sin_port = htons(32767);
inet_aton(dip, &adr.sin_addr);
printf("Listen on IP: %s\n", dip);

ask_id();

choix01 = ask_status_slave();
choix02 = ask_status_capture();
choix03 = ask_status_speed();

for(;;) {

  printf("\n1: SLAVE MODE ON/OFF                  - request01\n");
  printf("2: REQUEST STATUS SLAVE MODE          - request02\n");
  printf("3: REQUEST ID                         - request03\n");
  printf("4: CAPTURE ON/OFF                     - request04\n");
  printf("5: REQUEST STATUS CAPTURE             - request05\n");
  printf("6: REQUEST OLD PACKET                 - request06\n");
  printf("7: DOUBLE FREQUENCY AD ON/OFF         - request07\n");
  printf("8: REQUEST STATUS DOUBLE FREQUENCY AD - request08\n");
  printf("9: REQUEST RANGE OLD PACKETS          - request09\n");
  printf("0: QUIT\n");
  printf("Your choice is: ");

  scanf("%d",&d);

  switch(d){

    case 0 :
      printf("Quitting...\n");
      exit(0);
      break;
```

```
case 1:
  if (choix01 == 1) {
    slave_off();
    choix01=0;
  } else {
    slave_on();
    choix01=1;
  }
  break;

case 2:
  ask_status_speed();
  break;

case 3:
  ask_id();
  break;

case 4:
  if (choix02 == 1)  {
    capture_off();
    choix02=0;
  } else {
    capture_on();
    choix02=1;
  }
  break;

case 5:
  ask_status_capture();
  break;

case 6:
  printf("Your number packet is: ");
  scanf("%d", &number1);

  ask_old_packet((short)number1);
  break;

case 7:
  if (choix03 == 1) {
    speed_off();
    choix03=0;
  } else {
    speed_on();
    choix03=1;
  }
  break;

case 8:
  ask_status_speed();
  break;

case 9:
```

```
        printf("Your first number packet is: ");
        scanf("%d", &number1);

        printf("Your last number packet is: ");
        scanf("%d", &number2);

        ask_range_old_packet((short)number1, (short)number2);
        break;

      default :
        printf("Error: this input is not right: %i\n",d);
        exit(0);
        break;
    }
  }
  return 0;
}
```

7.2 The oscilloscope

The oscilloscope was made with the goal of viewing, listening and recording in a file the data received from one channel. It helped us to adjust the value of some resistors in the Microphone amplification stage like the gain.

At the top of the window, the ID of the microphone array is given and the PROM version. In our case it's 3ca.

The red numbers represents the highest and the lowest points of the channel at the time of picture.

The yellow numbers and letters represent the channel number/the key. For example in the top left corner is the channel 32 and if you type 1 you will listen it or the bottom right corner is the channel 47 and if you type 'v' you will listen to it.

The figure 41 represents the oscilloscope in record mode of the channel 21 at the UNIX time 1048627720 in the file /tmp/21-1048627720.ary.

The figure 41 represents also different channels (16 to 31) than the previous one (32 to 47) (cf figure 41). To pass from on quarter of 64 to another you have to use the keys F1, F2, F3, F4, F5:

- F1: first quarter with the channels from 0 to 15,
- F2: second quarter with the channels from 16 to 31,
- F3: third quarter with the channels from 32 to 47,
- F4: fourth quarter with the channels from 48 to 63,
- F5: all quarters.

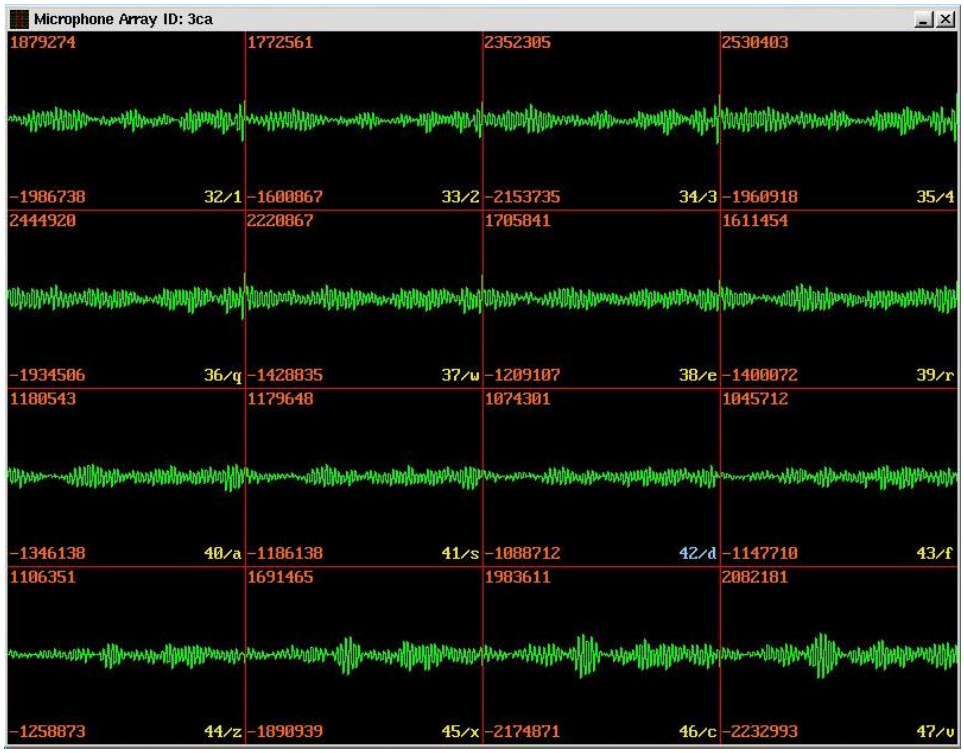


Figure 40: Oscilloscope in listening mode of the channel 42.

If you try F5 you might not be able to see of the channels because the window is in 1600*1200. So another key that might be interesting is the tab key. It toggles from window to full screen the oscilloscope.

To get a capture of a channel you have first to select it with the corresponding key in our case w and then press [space] to toggle the capture in the file on/off.

7.3 The library

Project....

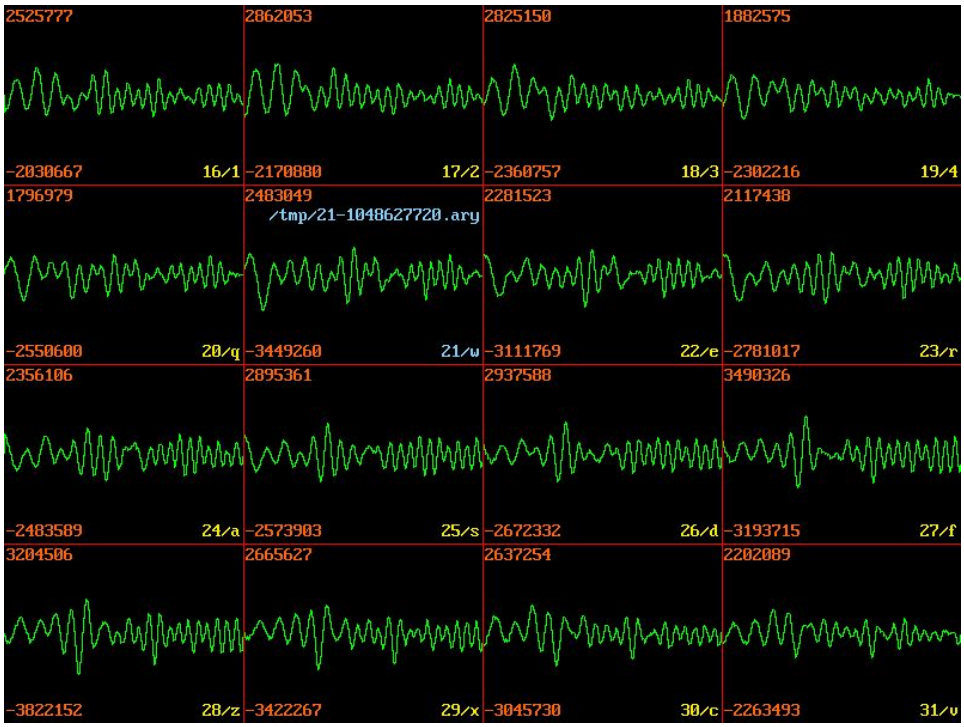


Figure 41: Oscilloscope in capturing mode of the channel 21.

Name	Description
Master Synchronization OUT	Connector for the synchronization cable. The other end of the synchronization cable should be connected to Slave Synchronization IN. The synchronization signals are going out of the Motherboard ,in Master Mode, through this connector
Slave Synchronization IN	Connector for the synchronization cable. The other end of the synchronization cable should be connected to Slave Synchronization OUT or Master Synchronization OUT. The synchronization signals are going in the Motherboard, in Slave Mode (to be detected), through this connector
Slave Synchronization OUT	Connector for the synchronization cable. The other end of the synchronization cable should be connected to Slave Synchronization IN. The synchronization signals are propagate from the Slave Synchronization IN to another Motherboard through this connector