Omicron Analyzer
The PV200 final project

Martin Vejnár

February 8, 2011

Chapter 1

Introduction

Omicron analyzer is a logic analyzer, modelled after the Sigma analyzer sold by Asix. It allows the state of a digital data bus to be sampled and transferred to a host computer. Two version of the analyzer are available—one built using Altera's DE2 development board which features a Cyclone II FPGA, and one built on a custom board with Altera's Cyclone IV E (see Figure 3.1 for a photograph).

Omicron analyzer can handle buses up to 16 bits wide at 25 million samples per second. The sampling rate is merely limited by the speed of the embedded SDRAM memory. For narrow buses, the sampling rate can therefore be higher: 8-bit buses can be sampled at a frequency up to 50MHz, 4-bit buses up to 100MHz and 2-bit buses up to 200MHz. (Note that currently the firmware can only sample at 50MHz.) The DE2 version can run the SDRAM chip at 100MHz, the sampling rates are therefore twice as high (on par with the commercial solution from Asix).

The analyzer stores compressed samples in a SDRAM memory, from which the samples can be read out. The interpretation of the samples is left to the host application (running on a host computer) and is out of the scope of this report.

Figure 1.1 shows the high-level structure of the analyzer. The core components of the analyzer are an FPGA chip and a SDRAM memory chip. The FPGA chip runs an embedded Nios II processor, which controls the sampling process. The system clock runs at the frequency of 50MHz (100MHz for the DE2 version). The sampling clock may potentially run faster (but currently doesn't).

All the board designs, the source code of the firmware for the FPGA and the software for the Nios CPU can be found in my repository on Bitbucket[1]. The repository also contains instructions on how to build and upload the firmware and software.

[1] `http://bitbucket.org/avakar/omicron_analyzer`

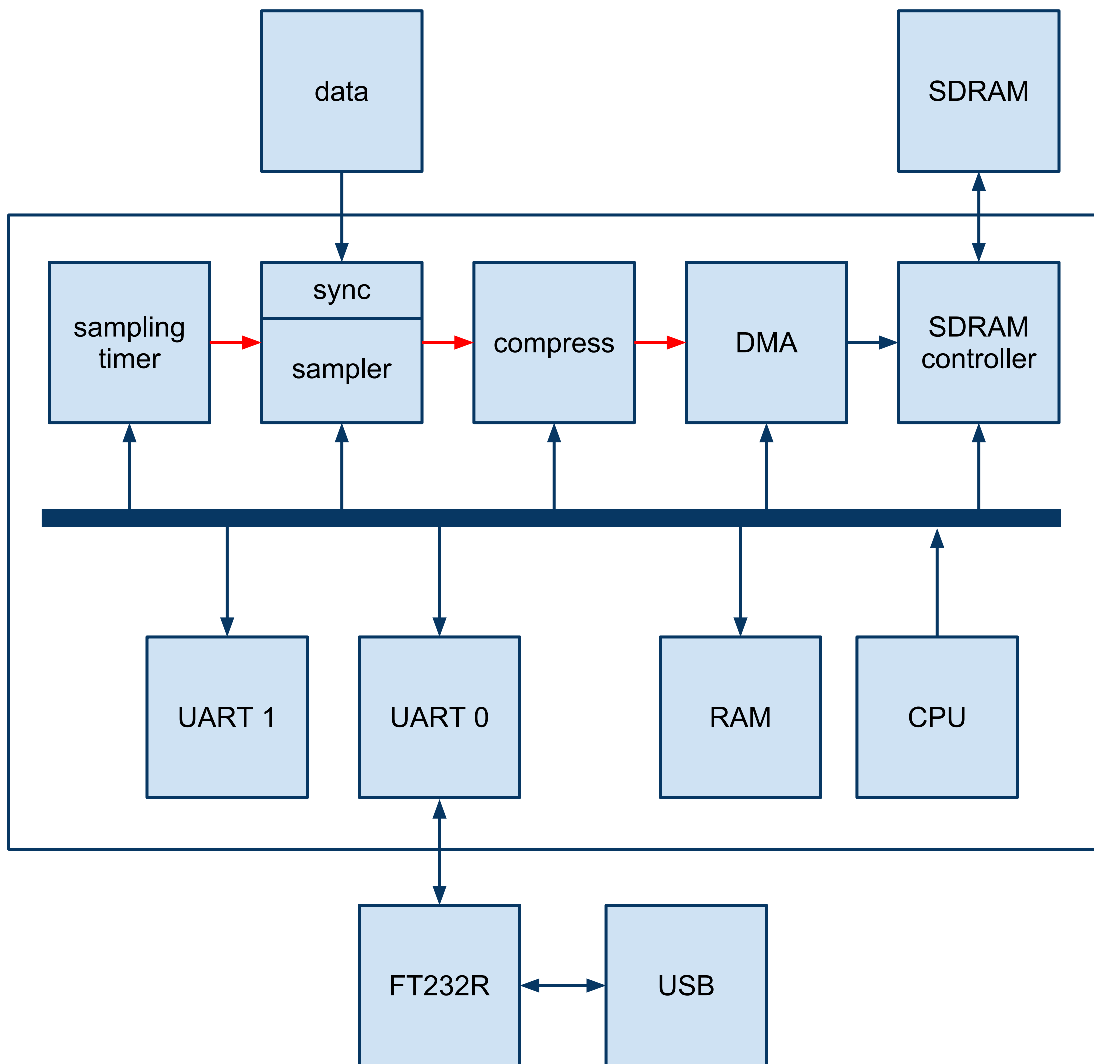


Figure 1.1: The overall architecture of the Omicron analyzer. The red arrows indicate Avalon streaming bus, the dark blue connection the Avali memory-mapped bus, with the arrows pointing from the master to the slave.

Chapter 2

Software

2.1 System architecture

The firmware of the FPGA chip was built using Altera's SOPC builder. The individual cores, which include the Nios II processor, the SDRAM controller, two UART controllers, the sampling timer, the sampler, the sample compressor and the DMA channer are all interconnected by the automatically generated Avalon bus.

All the cores, save for the Nios II processor, are available as memory-mapped slaves and can be controlled by the Nios software through a set of registers. The sampler, the compressor and the DMA channel are additionally connected by an Avalon streaming bus, through which the uncompressed and compressed samples are transported.

The DMA channel has a master port which is used to push data into the SDRAM controller. The default round-robin Avalon arbiter is currently used to distribute access to SDRAM between the DMA channel and the CPU. Recall that samples can arrive through the DMA channel as often as half the frequency of the SDRAM. Therefore, a custom arbiter should be built that would give priority to the DMA channel when its internal buffer becomes almost full, and allot access to the CPU otherwise. This strategy would allow burst transfers to be performed from the DMA channel, ensuring timely delivery of the data before the sample stream becomes saturated.

2.2 Sampling process

During sampling, the data values are read from FPGA input pins. The data is sampled in at a positive edge of the sampling clock. The data is then transported through a two-stage synchronizer to limit the probability of a metastate failure.

The sampler core analyzes the incoming data and produces samples when either the sampling timer overflows, or a sampling edge is detected on the input data. For each pin, a rising edge, a falling edge or a combination of both can be configured to force a sample to be created. Therefore, the sampler can either produce samples periodically (if the sampling is based on the internal timer) or based on an external clock signal

(on a single or both edges thereof). (A combination of the two sampling events can be configured as well—we have yet to determine a valid use-case for such a setting.)

The sampler can be configured to produce samples that are 16-bit, 8-bit, 4-bit, 2-bit or 1-bit wide. The pins that produce data in a narrow sampling mode (i.e. any mode but 16-bit mode) cannot be chosen at the moment—an n-bit sample is always produced from pins 0 to $n - 1$. The least significant bit of the samples corresponds to pin 0.

For samples shorter than 16-bits, an appropriate amount of consecutive samples are packed together into a single super-sample. The samples are packed so that the most significant bits of the super-sample contain the earliest sample. The super-samples are then streamed out of the sampler; the rest of the system works on units of 16-bits.

As can be seed in Figure 1.1, the sample stream from the sampler is fed directly into the sample compressor, which applies a basic RLE compression technique to the samples and produces a compressed stream (again with units of 16-bits). The compressor is guaranteed to produce a stream at most 1.5 times longer than the input stream. In a usual setting, where the state of the bus rarely changes, save for a few fast bursts of data, the compression ration can be as high as 1:65536.

The compressed stream is then fed into a DMA channel which buffers the data and transports them into the SDRAM controller.

2.3 Sample numbering

During sampling, each sample is assigned a number, the so-called sample index. The first sample is assigned a sample index zero. For n-bit sampling, the second sample is assigned the sample index of n. This way, each sample is assigned a number corresponding to the physical position in bits of the sample in the (uncompressed) sample stream.

Sample indexes are used to mark important events in the sample stream. Currently, sample indexes are used to mark the positions of a trigger event and the length of the stream.

2.4 Stopping the sampling process

The sampling process can be stopped either by the CPU (initiated by the host computer) or by the sampler by encountering a trigger.

A trigger is caused by encoutering an edge on the data bus. The set of edges that cause a trigger can be configured. When a trigger is hit, the current sample index is captured and is sent to the host when after the sampling is stopped.

The sampler additionally holds a counter which starts down-counting when a trigger is hit. The initial value of the trigger counter can be configured. The trigger counter is decremented whenever a sample is produced. When the trigger counter reaches zero, the sampling is stopped.

In either case, the sampler is the first component that is stopped. When narrow-sampling, if a partial super-sample is stored in the sampler, dummy samples are gen-

erated to pad the super-sample to 16-bit. The super-sample is then flushed into the compressor and the sampler indicates to the CPU that it stopped.

The CPU then stoppes the compressor (which again flushes any counters it currently stores), and then waits for the DMA channel to empty. Note that the three components must be flushed in this order to ensure that no samples are lost in the pipeline.

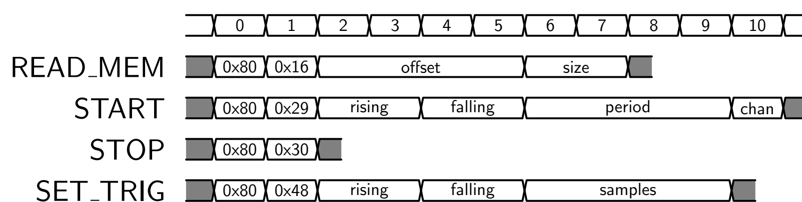
2.5 Host computer interface

The analyzer communicates with the computer through an UART. Two UARTs can be found on the custom board. UART 0 connects through the FT232R converter to the USB bus, the UART 1 can be connected to any 3.3V LVTTL-compatible device and can be used for debugging. The UART 0 can potentially communicate as fast as 3 megabauds—the RTS/CTS handshaking can be used to throttle the line. Note that the Nios processor will likely not be able to communicate this fast—a hardware buffers should added between the processor and the UART.[1]

The communication stream is divided into packets, which always have the following structure.

The 0x80 byte is used as a synchronization marker. The second byte contains the command identifier and the size of the rest of the packet. Both fields are 4-bits wide, the former contained in the high-order nibble. The maximum size of a packet is therefore 17 bytes (including the 0x80 synchronization). Besides packets, single-byte commands (other than 0x80) can also be sent through the UART (e.g. manually from a client like PuTTy). These may be useful for debugging.

The software ignores all packets with an unknown command/size combination. While a command is in progress, no other command will be processed. Note that until a hardware buffer is implemented for the UART interface, you cannot buffer multiple commands—the subsequent command will be lost. The following commands are defined.

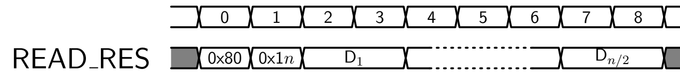


Note that all numbers are sent little-endian, i.e. the least significant byte first. The only exception to this rule is the transport of samples themselves (which occurs as the response to the READ_MEM command). The samples are transported the most significant byte first—this allows 8-bit packed bytes to arrive in order.

[1]Please see the documentation for the version of the firmware you're actually using. At the moment of this writing, the USB port is not yet used.

The READ_MEM command is used to retrieve data from the SDRAM. Note that the command can be used to read the data even when during sampling, allowing real-time transport of the samples. The offset and length parameters specify the range to read from the SDRAM. Note that SDRAM is 16-bits wide—both parameters are in 16-bit words.

As a response to READ_MEM command, the analyzer responds with the READ_RES command.



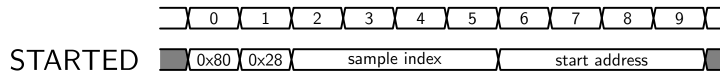
The analyzer will respond with as many READ_RES packets as is necessary to satisfy the requested length. The words are sent in big-endian format, i.e. the most significant byte first.

The START command begins the sampling process. The set of edges that cause a sample to be generated can be specified. The rising and falling masks specify for each pin the edges that that the pin reacts to. For example, to sample a bus on a rising edge of a clock connected to pin 4, you would specify the rising mask as 0x0010 and the falling mask as 0x0000 (note that the clock need not be a part of the samples; 4-bit or narrower sampling is very much allowed).

The period parameter specified the nmber of ticks the the sampling timer performs before it overflows and causes a sample. The sampling timer is clocked by the sampling clock (currently 50MHz for the custom board and 100MHz for the DE2 version; the frequency may be increased in the future). The timer can be disabled by setting the period to 0xFFFFFFFF.

The last byte specifies the binary logarithm of the number of channels to sample. For example, to generate 1-bit samples, specify zero. For full 16-bit samples, specify 4. Values higher than 4 are reserved.

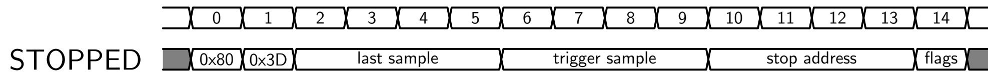
The analyzer start the sampling process and responds with the following packet.



The response specified the index of the first sample that will be produced; the start address specified where in the SDRAM will the sampling stream be stored.

The SET_TRIG command configures the trigger. The rising and falling masks are used in the same manner as the masks in the START command. The last arguments specifies the number of samples that are to be produced after the trigger is hit and before the sampling is automatically stopped. Passing the value of zero causes the sampling to be stopped immediately after the trigger is hit.

The STOP command causes the analyzer to stop sampling. After the sampling stops (and all samples are committed to SDRAM), either due to trigger or due to the STOP command, the analyzer responds with the STOPPED notification.



The STOPPED notification specifies the sample index that would be assigned to the next sample (i.e. index of the last sample plus one). If the trigger was hit during sampling, the second argument indicates the index of the sample during which this occured; if the trigger event did not occur, this field can be ignored. Stop address is a pointer to the SDRAM memory where the next word of the compressed sample stream would be stored. The least significant bit of the flags field indicated whether a trigger got hit during sampling.

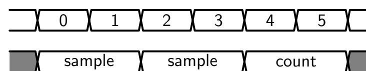
A part of this work is a client script written in python, which can be used to perform a sampling run, and can also serve as an inspiration for future host applications.

2.6 Sample compression

The compression technique used to decrease the length of the sample stream is a simple run-length encoding. The compressor works as follows.

- The first sample is copied to the output; the compressor transits into the IDLE state.
- When in IDLE state, each subsequent sample is produced into the output and compared to the previous one. If the samples are the same, the compressor transits into the COUNTING state, with the counter initialized to zero.
- In the counting state, no samples are output; they are merely compared to the previous sample. If the samples match, the counter is increased by one. If the counter reaches the maximum value of 0xFFFF, the value 0xFFFF is produced to the output and the compressor continues with the counte value of 0. If the samples fail to match, the current counter value is produced to the output, followed by the new sample. The compressor then transits back into the IDLE state.

This way, all runs of length at least two of samples are replaced by the three-word sequence



The count indicates the length of the run minus two. For counts greater than 0xFFFE, the word 0xFFFF is prepended and the count decreased by 0xFFFF is produced. (This algorithm can be applied iteratively, until count drops below 0xFFFF.)

It can be seen that the technique can have compressoon ratios as high as approximately 1:65536 (for a long stream with only a single sample value). Conversly, the compression will increase the size of the stream by at most 50% in the worst case (many runs of length two), thus allowing the sample pipeline to be designed to handle the stream.

Chapter 3

Hardware

While the firmware can be uploaded to the DE2 board, a much smaller and less expensive custom board was developed for the Omicron analyzer. The schematics and the board design can be found in the code repository in the `board` folder. Figure 3.1 shows the photograph of the board.

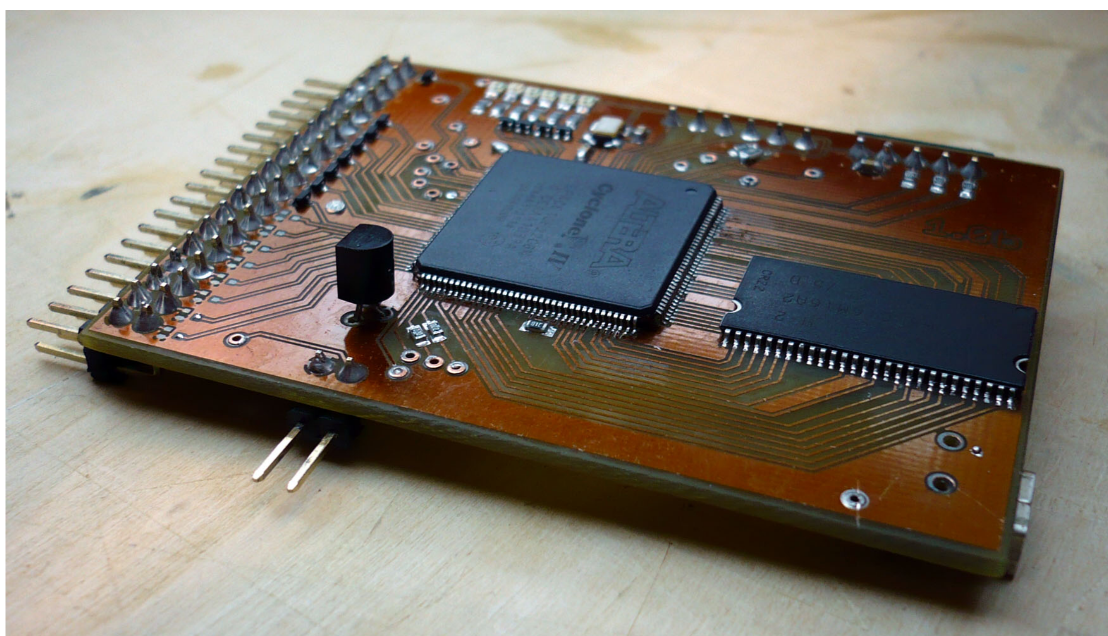


Figure 3.1: The custom board.

The core chips on the board include the 144-pin EP4CE10E22 FPGA chip from Altera (part of the Cyclone IV family), a 4 megabit EPCS4 configuration memory for the FPGA, a 32 megabyte SDRAM, and the FT232R USB-to-UART converter.

3.1 Interface

A mini-USB connector can be found on the board. The USB signal pair is connected to the FT232R converter, which provides the FPGA with LVTTL asynchronous serial data. The CTS/RTS handshake signals are also connected to the FPGA, allowing for hardware control flow. The hardware handshaking is necessary at higher speeds (the converter can communicate at speeds as high as 3 megabauds), as the internal FT232R buffers may overflow due to a rather large latency of the USB bus.

Additionally, a second 3.3V LVTTL-level UART is available on the board for debug purposes. For programming, the JTAG header is included. See the schematics for the pinouts.

3.2 Supply network

The USB port also supplies the necessary supply voltage of 5 volts. Three voltage regulators are present, generating 3.3 volts, 2.5 vols and 1.2 volts. The 1.2V supply is used to power the FPGA core; the 2.5V supplies the chip's analog PLL supply. The PLL supply pins are all bypassed with inductors to reduce possible oscillations. The 3.3V branch supplies everything else—the FT232R converter, the EPCS4 configuration memory, the SDRAM chip and the LEDs. It also supplies the I/O banks of the FPGA.

The 2.5V supply is also used as a supply voltage for the JTAG programming header—the 2.5V interface is compatible with the 3.3V of the FPGA I/O banks, yet ensuring that the inductive overshoot does not damage the JTAG pins (which for some reason lack the PCI clamp diode).

3.3 Pins

The data enter into the analyzer through an array of digital (pre-biased) bipolar transistors, which protect the FPGA chip from overvoltage and force the signals to a specified logic level. The transistors are biased with two 47 kiloohm resistors, making the input resistance sufficiently high (for a data bus) and setting the threshold voltage to approximately 1.4 volts.

Three of the pins are bidirectional—three transistors in a half-bridge configuration are used to drive them. Note that three bidirectional pins are sufficient for most types of serial communication. In particular, I2C, SPI, JTAG, UART, even PDI communication can be performed with these pins. (This makes the Omicron analyzer a replacement for Asix Presto as well as Asix Sigma, allowing in addition the programming of Atmel's XMEGA processors.)

While there are 16 pins (3 of which are bidirectional), there are 19 pins total. If the board is oriented with the FPGA on the top and the pins on the left-hand side, the first pin from the upper edge is channel 1, followed by an unconnected pin, a VDD supply for the bidirectional pins, the ground, followed by the rest of the channels (channels 2

through 16). This particual configuration allows the same connector as the one supplied with Asix Presto to be connected to the Omicron analyzer.

An mentioned before, the bidirectional pins, when they're set to output are driven either to ground or to the VDD pin. For convenience, the board can also connect its 5V supply to the VDD pin, allowing for TTL-level communication.

3.4 Other

The board has several LEDs—one at the bottom (green) indicating a 5V supply, and 6 LEDs at the top. The leftmost indicates voltage on the VDD pin (red), the one next to it the voltage on the internal 3.3V supply lanes (green). The remaining four red LEDs are controlled from the FPGA.

A 25MHz oscillator supplies the FPGA with clocks. The oscillator has a shutdown feature, but it is currently unused.

The EPCS4 configuration memory is used to boot the FPGA after power-on event. The memory can be programmed by first configuring the FPGA with a design containing Altera's SFL (Serial Flash Loader) IP core. Through this core, the configuration memory is accessible through JTAG. See Altera's documentation for details.