Flex Logix eFPGA as Silicon Debugger

Application Note v1.1

Contents

1. Introduction 2
2. About Flex Logix eFPGA 2
2.1. Introduction to Embedded FPGAs 2
2.2 EFLX Embedded FPGAs 2
3. Prior Art 3
4. Debug, Configuration and Management module in eFPGA 4
5. Simple Example 6
5.1 Traditional SoC module 6
5.2 Limited Visibility and Controllability 7
5.3 Protecting your design with eFLEX embedded FPGA 7
5.3.1. Improve Observability 8
5.3.2. Add Event Triggers 9
5.3.3. See Hidden Data with Embedded Logic Analyzer 10
5.3.4. Future-proof Your Design 12
6. Conclusions 14

1. Introduction

Silicon debugging is very challenging as visibility inside the device is limited to a limited set of viewpoints that are available in hardware. Those viewpoints may include:

- Device IOs
- Device registers
- Predefined signals routed to a debug port

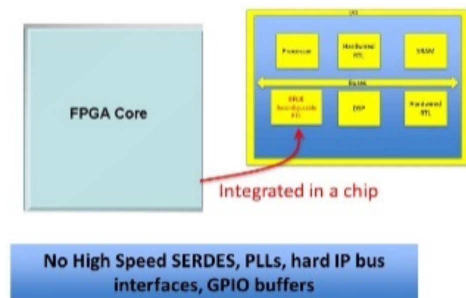
The first two resources are considered free as they are part of the product definition and don't require additional effort or resources to be implemented.

The debug port is purposely added to help debug the device and is usually composed of a big MUX that can select a variety of signals at different predefined points inside the device. At any given time, only a predefined set of those signals can be available at the debug port for observation.

This application note describes a new and improved way to add debug capabilities to a silicon device, significantly improving the visibility and control at various points inside the device.

2. About Flex Logix eFPGA

2.1. Introduction to Embedded FPGAs



An FPGA combines an array of programmable/ reconfigurable logic blocks in a programmable interconnect fabric. In an FPGA chip, the outer rim of the chip consists of a combination of GPIO, SERDES and specialized PHYs such as DDR3/4. In advanced FPGAs, the I/O ring is roughly 1/4 of the chip and the "fabric" is roughly 3/4 of the chip. The "fabric" itself is mostly interconnect in today's FPGA chips where 20-25% of the fabric area is programmable logic and 75-80% is programmable interconnect (not including configuration bits). Programmable logic including configuration bits and interconnect including configuration bits are closer to 50-50. An embedded FPGA is an FPGA fabric without the surrounding ring of GPIO, SERDES, and PHYs. Instead, an embedded FPGA connects to the rest of the chip using standard digital signaling, enabling very wide, fast on-chip interconnects.

2.2 EFLX Embedded FPGAs

Flex Logix provides high-density, high-performance, energy-efficient embedded FPGA hard IP. EFLX is a scalable architecture of silicon-proven IP that enables embedded FPGA's from 120 LUTs to >100K LUTs. The smallest core (EFLX-100) has 120 LUTs with 152 inputs and 152 outputs. The larger core (EFLX-2.5K) has 2,520 LUTs with 632 inputs and 632 outputs. The cores can be "tiled" to form arrays. The EFLX-100 cores can be tiled to build arrays up to 5x5 or 3,000 LUTs of reconfigurable logic with 760 inputs and 760 outputs. The EFLX-2.5K cores can be tiled to build arrays up to 7x7 or 123K LUTs of reconfigurable logic with 4,424 inputs and 4,424 outputs. The EFLX cores have two versions that can be mixed together in arrays. One version is all logic and the other has 22-bit MACs for DSP functions. The small core can have 2 MACs and the large cores can have 40 MACs. The cores also support memory structures that can be part of the array for small sizes or outside of the array for larger sizes.

The embedded FPGA is programmed using Verilog or VHDL which are input to a synthesis tool such as Synopsys' Synplify. The EFLX Compiler takes the synthesized output and packs/places/routes the array and generates a bit stream which programs the EFLX array to emulate the RTL. The bit stream for a single EFLX-100 is ~50K bits and can be stored in the same flash memory that stores the embedded processor code. The EFLX array can be reprogrammed at any time, just like with an embedded processor. EFLX arrays can be integrated into an SoC or MCU in three common ways as depicted in Figure 1.

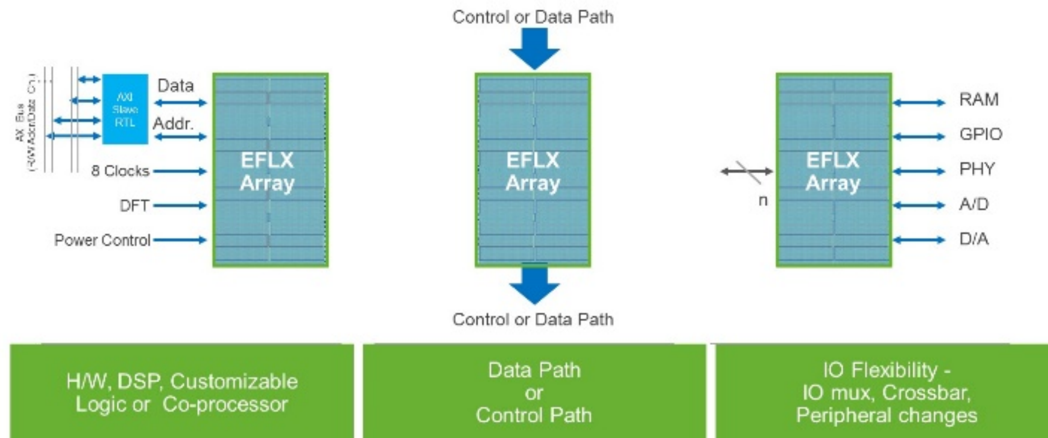


Figure 1 EFLEX use examples

3. Prior Art

There are many SoCs, custom built for various application like CPU, networking, storage, GPU, accelerators and many more. Their architecture is customized for their purpose but all of them include some basic modules that allow them to:

- Receive data
- Store the data
- Process the data
- Output results
- Manage the device
- Test and debug the device

The block diagram in Figure 2 shows one such design. Note how the Configuration & Management module has similar system connections as the Debug module. However, the Debug module usually connects to a larger variety of signals, including wide data busses, and typically has a large multiplexer that allows real-time visibility of those signals through the Debug Port.

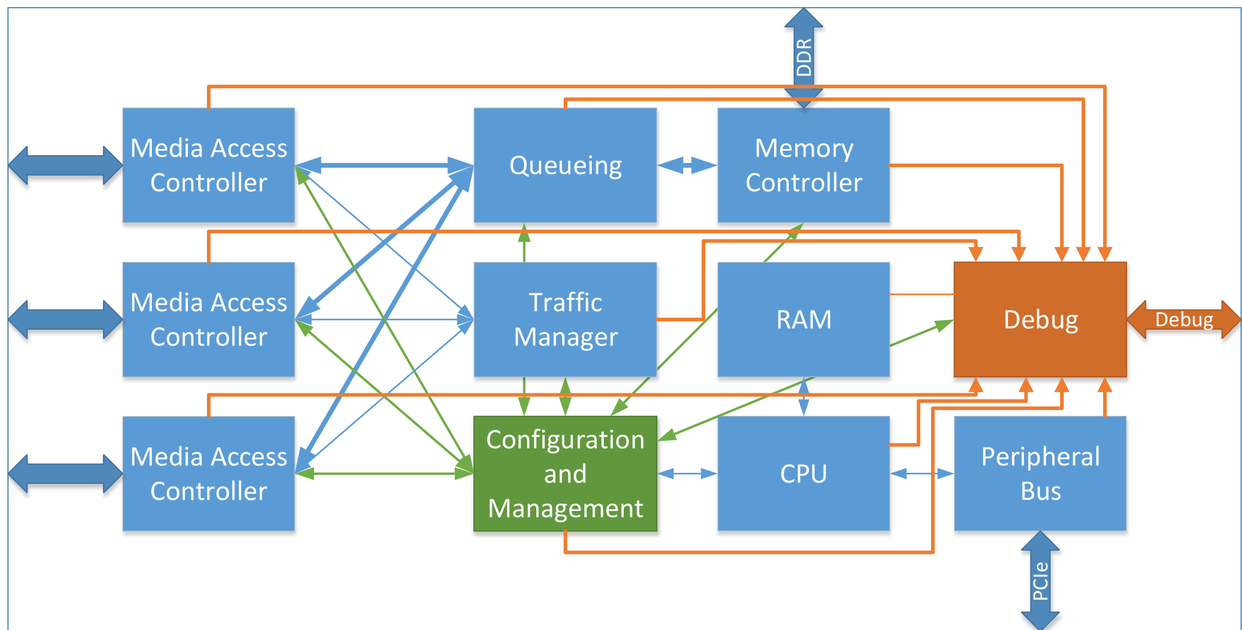


Figure 2 Typical SoC

This architecture creates routing congestion around the Debug module and is limited in visibility by the design of the multiplexer inside the Debug module. Furthermore, the signals can only be observed by physically connecting into the Debug Port making it extremely difficult to debug devices that are operational in the field.

4. Debug, Configuration and Management module in eFPGA

The Flex Logix eFPGA adds a high degree of flexibility to the SoC. The Configuration and Management module together with the Debug module are excellent candidates for implementation in eFPGA as shown in Figure 3.

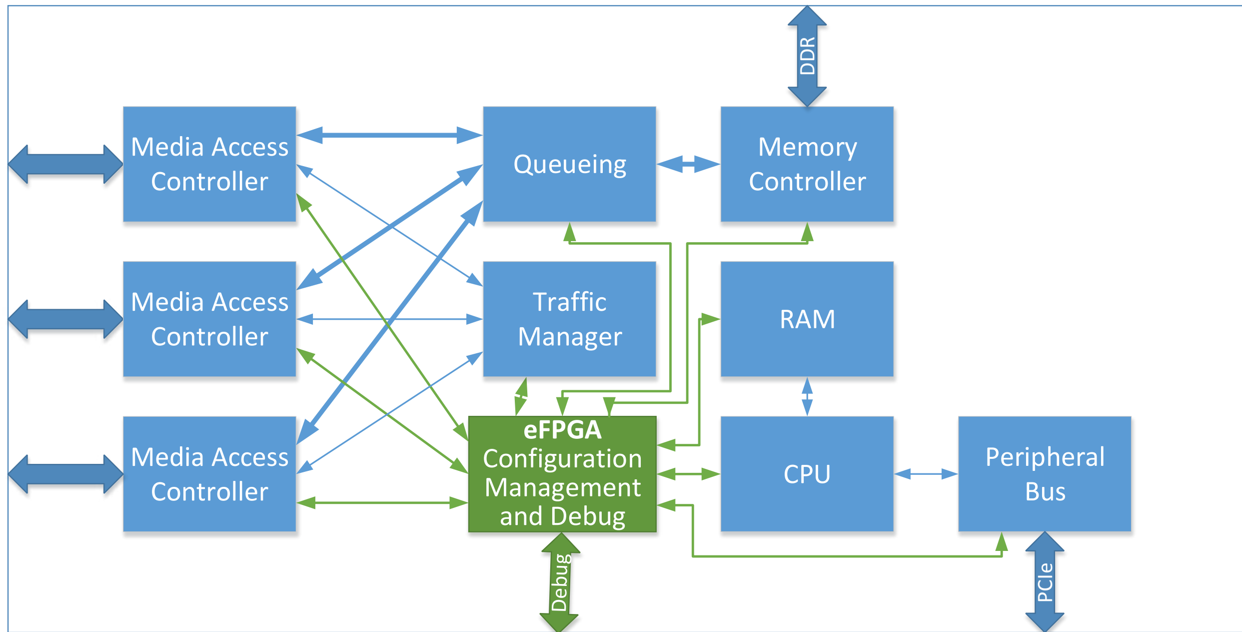


Figure 3 eFPGA adds flexibility and simplifies the architecture and routing

The factors that led to this suggestion are:

- Visibility and controllability are highly desirable for both management and debug functionality. One EFLX2.5K eFPGA core has 632 inputs and 632 outputs, and one EFLX100 core has 152 inputs and 152 outputs, providing expanded access to chip management and debug.
- Connectivity of both Configuration and Management module and the Debug module are very similar and largely redundant. Integration of Configuration Management and Debug into a single module and pruning of redundant connectivity reduces routing congestion.
- Debugging the SoC usually requires a combination of signal monitoring through the management and configuration as well as the debug port. Adding real-time visibility to all signals, as well as control, in a single point (eFPGA) highly simplifies the debug process.
- The logic implemented in the eFPGA core allows for real-time intervention at pre-defined points in the device allowing to add aftermarket functionality or bug workarounds to the device, reducing the risk of a device re-spin.
- The flexible logic inside the eFPGA allows for very complex event triggers that can help capture and filter relevant events without the use of external logic analyzers.
- Data can be captured and stored in the eFPGA or RAM for remote access.
- Additional statistics can be easily gathered based on the connected signals. Unneeded statistics can be eliminated to reduce silicon area.
- Static Configuration registers can be reduced to constants implemented in the eFPGA configuration file. Their values can be changed by modifying the eFPGA image and don't occupy flip-flops.
- Having those capabilities in all SoCs, without the need of expensive and complex lab equipment, allow for low-cost highly parallelized diagnostic effort in the lab and in the field over long periods of time.

5. Simple Example

5.1 Traditional SoC module

For this example, we'll use a sort engine as the DUT (Design Under Test). The sort engine receives packets of data on one side and outputs the same packets with their data sorted in ascending order on the other side.

The diagram in Figure 4 shows the architecture of this sort engine.

The operation mode of the sort engine is as following:

- Receive the data packets (*in_stage*)
- Store the data packets in the SRAM
- The *sort* module sorts the data in the SRAM
- Transmit the sorted data packets (*out_stage*)
- CPU interfaces to *config_and_management* to program various configuration parameters and monitor the activity
- The design can be debugged by observing various signals in the *debug_monitor* module

The input protocol is a simple *data_in* and *data_v_in* protocol, where the data packet starts with the assertion of the *data_v_in* signal and ends with its de-assertion. No pauses are permitted in a middle of a data packet, only between the packets. The module *in_stage* also has a *hold_in* output signal that, when TRUE, is asking the data sender to hold after the current packet, until *hold_in* is FALSE again.

In a similar way, the *out_stage* module has *data_out, data_v_out* outputs and *hold_out* input. The *data_out* and *data_v_out* are outputting sorted packets and the *hold_out* input signal can stop the *out_stage* module from outputting packets, also on packet boundaries.

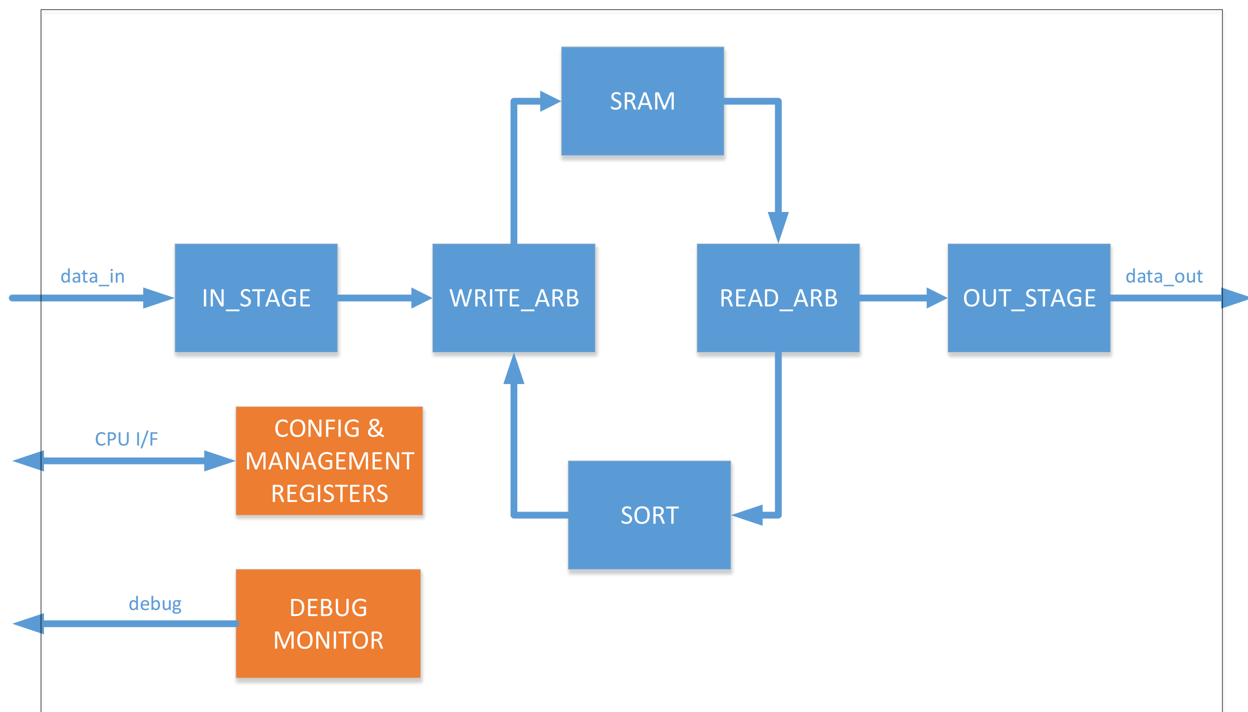


Figure 4 Sort engine in traditional SoC

The sort module will read data from the SRAM and sort the data, maintaining the packet structure. The sort algorithm used Quick-Sort.

5.2 Limited Visibility and Controllability

The Debug Monitor and Config & Management blocks (orange in Figure 4), require a lot of flexibility and occupy a lot of area in silicon.

Most of the registers in the Config & Management are rarely changed by the user and only contain static configuration.

The Debug module should allow visibility in all design signals but it is practically limited by the routing complexity of multiplexing many signals into a limited-size debug bus while allowing any possible configuration of signals. In practice, this module will limit visibility to a small set of predefined signal configurations.

5.3 Protecting your design with eFLEX embedded FPGA

The Flex Logix eFPGA allows a new degree of flexibility to be added to your design. Here are various features of the eFPGA cores that can help improve visibility and controllability of your design:

Feature	Benefit
Hundreds of inputs and outputs	• Greatest possible observability, tap into hundreds of places
Configurable routing	• Route any combination of signals to the debug port, triggers or storage for greatest visibility
Storage elements	• Sample data at the tap points, allows for backtracking and for monitoring high-speed signals • Saves the cost of logic analyzers and allows debug in the field
Configurable logic cells	• Build custom complex event triggers to detect events of interest
eLogicAnalyzer	• Store meaningful data around trigger events and access it remotely • Save the cost of logic analyzers and allow debug in the field
All of the above	• Augment or replace entire designs to save the ASIC from costly re-spins

The same design of Figure 4 can be improved by adding an eFPGA core as shown in Figure 5.

The eFPGA core connects to all major busses and signals in the original design. In addition, two interface multiplexers allow the eFPGA to select the normal operation of the design or completely replace the design.

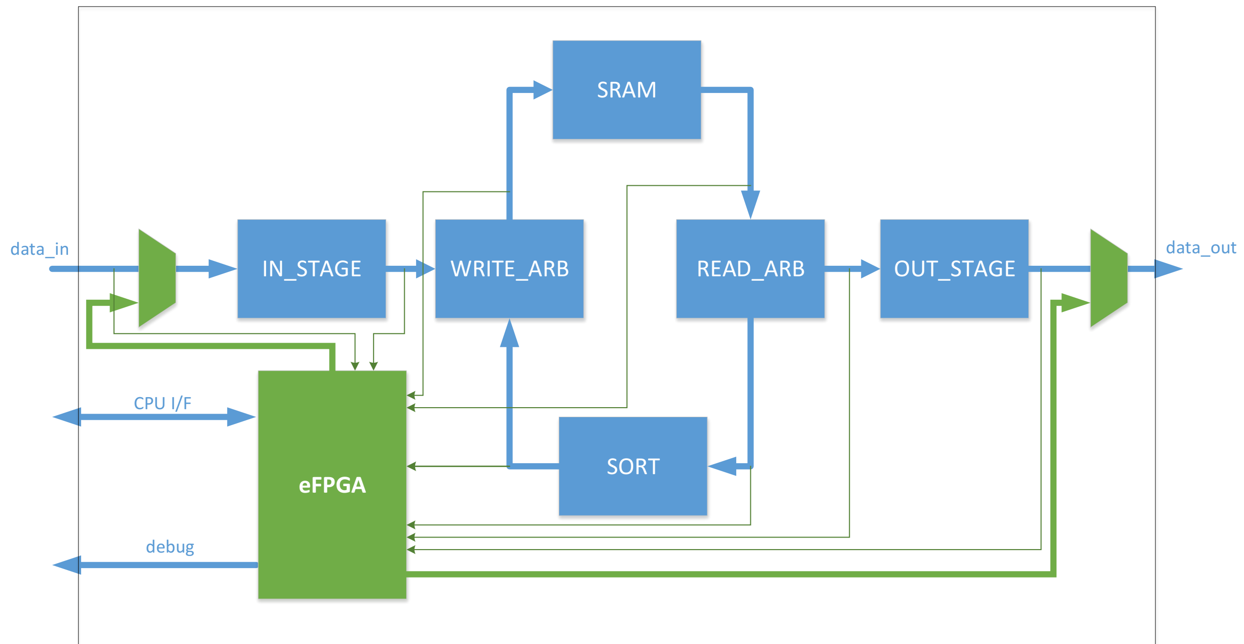


Figure 5 Sort engine design with eFPGA

5.3.1. Improve Observability

The original design had a limited number of signals connected to the debug port. Since the debug port is limited in size, only a small set of the signals can be observed at any time and the designer must predefine what signal combination will be made available. Below is a design example. Note: some signals are replicated in several configurations of the multiplexer and only 32 signals can be viewed at any given time.

```
// example of how a debug port is usually implemented in ASIC.
   // debug port outputs
   output [31:0] debug; // to debug port
   wire [2:0]    debug_mux; // CPU Write Register
   reg [31:0]    debug;
   wire [31:0]   debug1, debug2, debug3, debug4, debug5, debug6, debug7, debug8;

   assign debug1 = {data_v_in, data_v_out, hold_in, hold_out, req1, req2, WE1, WE2,
                    write_ptr, write_ptr_to_out_stage, read_ptr}; // mix of major signals
   assign debug2 = {hold_in, read_ptr[6:0], data_v_in,
                    write_ptr[6:0], data_in}; // cutting MSB of pointers to save wires
   assign debug3 = {ack_w2, WE2, WE1, WE, 3'h0, data_w, adrs_w}; // write_arb
   assign debug4 = {ack_r2, req2, req1, 4'h0, data_r, adrs_r}; // read_arb
   assign debug5 = {WE2, ack_w2, adrs_w2[1:0], req2, ack_r2, data_r2, adrs_r2}; // sort read
   assign debug6 = {WE2, ack_w2, adrs_r2[1:0], req2, ack_r2, data_w2, adrs_w2}; // sort write
   assign debug7 = {adrs_w1[1:0], adrs_w2[1:0], WE1, WE2, WE, data_w, adrs_w};  // SRAM read
   assign debug8 = {req1, req2, ack_r2, adrs_r1[1:0], adrs_r2[1:0],
                    data_r, adrs_r}; // SRAM read

   always @(debug1 or debug2 or debug3 or debug4 or
            debug5 or debug6 or debug7 or debug8 or debug_mux)
     begin
        case (debug_mux)
          0: debug = debug1;
          1: debug = debug2;
          2: debug = debug3;
          3: debug = debug4;
```

```
          4: debug = debug5;
          5: debug = debug6;
          6: debug = debug7;
          7: debug = debug8;
        endcase // case (debug_mux)
      end // always @ (...
```

Using the eFPGA you can achieve better observability as follows:

- Choose one or multiple EFLEX eFPGA cores based on design needs and implement them on your SoC
  - EFLEX-2.5K cores have 632 inputs and 632 outputs per core
  - EFLEX-100 cores have 152 inputs and 152 outputs per core
- Map the eFPGA inputs to the signals you want to monitor at the SoC design stage; those will typically be the signals on the right side of the **assign** statements above
- After ASIC production, assign any signals to the debug port (32-bit in this example)
- Compile and load the design in the eFPGA

```
// Example of how to route signals in the eFPGA
// No need to implement a MUX as the design can be changed and reloaded on demand

    output [31:0] debug; // to debug port

    assign debug = {data_v_in, data_v_out, hold_in, hold_out, req1, req2, WE1, WE2,
                    write_ptr, write_ptr_to_out_stage, read_ptr}; // mix of major signals
```

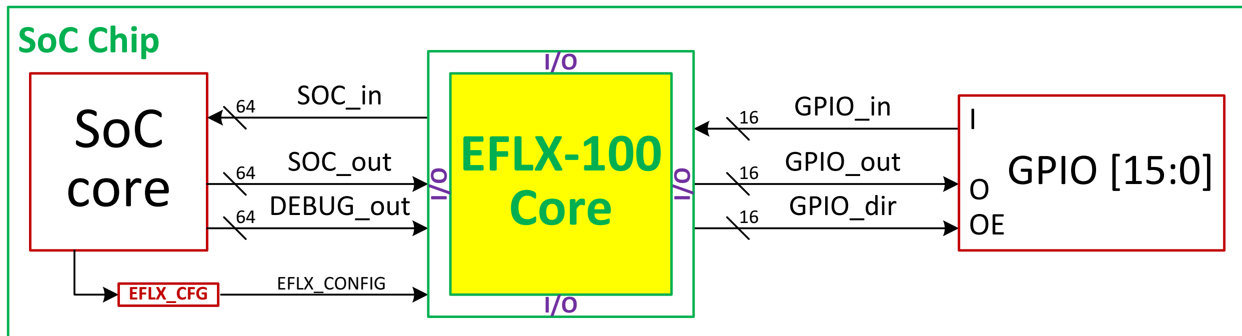


Figure 6 Example of debug signal multiplexing using a FLEX-100 core

5.3.2. Add Event Triggers

Looking for a specific event is like looking for a needle in a haystack unless you have a trigger that can identify the event. For example, in the sort design, we might encounter a situation where the data is corrupted and observing the fill level of the SRAM (when it fills up) will help find the bug. However, the debug multiplexer in this design only allows to see the read and write pointers without actually seeing what address is written to or read and by whom at the same time. A simple trigger will help detect the event while observing the relevant data, address and control busses as explained in the previous section.

Using the eFPGA you can build and trigger:

- Choose one or multiple EFLEX eFPGA cores based on design needs and implement them on your SoC
  - EFLEX-2.5K cores have 2,520 LUTs per core (or 1,860 LUTs and 40 DSPs)
  - EFLEX 100 cores have 96 LUTs per core

- Map the eFPGA inputs to the signals you want to monitor at the SoC design stage
- After the ASIC production, build the trigger logic in the eFPGA core
- Assign the triggers and any other signals to the debug port (32-bit in this example)
- Compile and load the design in the eFPGA

```
// Example of trigger logic to detect buffer overfill
wire buffer_overfill;
// makes sure there is always room for one more packet
assign buffer_overfill = (write_ptr - read_ptr) > ((1<<AddressSize) - max_pkt_size));
assign debug = {buffer_overfill, WE1, WE2, WE, 1'b0, adrs_w1, adrs_w2, adrs_w};
```

5.3.3. See Hidden Data with Embedded Logic Analyzer

A Debug port that's hardwired to a bunch of signals is not always sufficient to see all the desired signals (32 GPIO only in this example). In addition, the speed of the debug port is not always sufficient to see high-speed signals.

This limitation can be overcome by instancing the eLA (embedded Logic Analyzer) inside a EFLEX core. The eLA is fully implemented in the eFPGA core and can connect directly to the SoC debug port or an embedded CPU. The eLA can use the RAM elements of the EFLEX 2.5K or use external SRAM for larger storage.

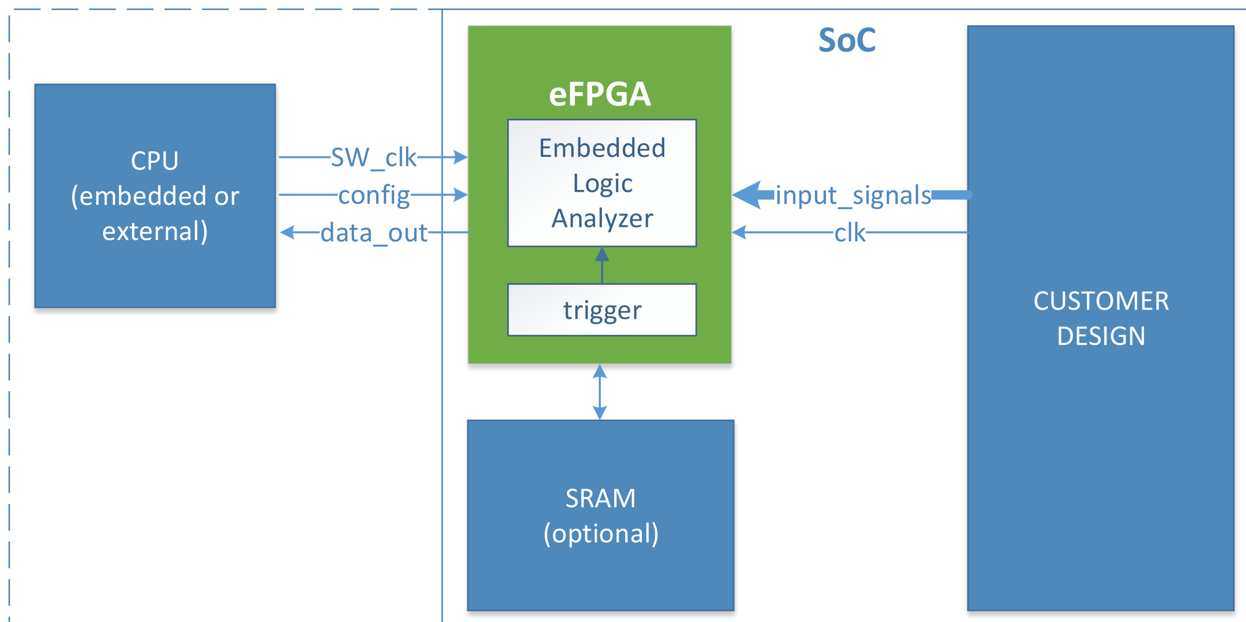


Figure 7 Embedded Logic Analyzer (eLA) and trigger on eFPGA

The Flex Logix eLA (Embedded Logic Analyzer) has the functionality of a high-end logic analyzer that is available in any design with a Flex Logix eFPGA. The eLA allows logging of internal signals at very high speeds based on complex triggers or continuous sampling. The trigger event location can be anywhere within the range of the logging buffer.

The triggers implemented as described in the previous section are used to trigger eLA. The eLA will store data around the trigger event, depending on its configuration. The eLA can be configured to work with an external SRAM (higher storage capacity) or use internal storage elements.

After the trigger is detected and data is ready for download, the CPU can access the data, through a very simple serial interface, process and display it. A GUI snapshot is shown in Figure 8, showing how the inputs and outputs on a pre-sortes stage described further on.

When the CPU is embedded, this operation can be performed completely remotely, even on chips deployed at a customer location, allowing for a very flexible debug capability for the SoC itself and even for the entire system where the SoC is deployed.

```
module eLogiAnalyser( // stub instance of Embedded Logic Analyzer
   // Outputs
   eLA_data_out,
   // Inputs
   clk, eLA_input_signals, eLA_trigger, eLA_SW_clk,
   eLA_config);

   parameter eLA_SIGNAL_COUNT = 128; // EFLEX-2.5K has 632 inputs, EFLEX-100 has 224 max inputs
   // Global inputs
   input clk; // high speed clock that samples eLA_input_signals
   // design inputs
   input [eLA_SIGNAL_COUNT:0] eLA_input_signals;
   input                      eLA_trigger;  // trigger condition
   // debug interface
   input                      eLA_SW_clk;   // software clock for eLA_data_out & eLA_config
   output                     eLA_data_out; // serial data out
   input                      eLA_config;   // configuration input for trigger location

endmodule // eLogiAnalyzer
```

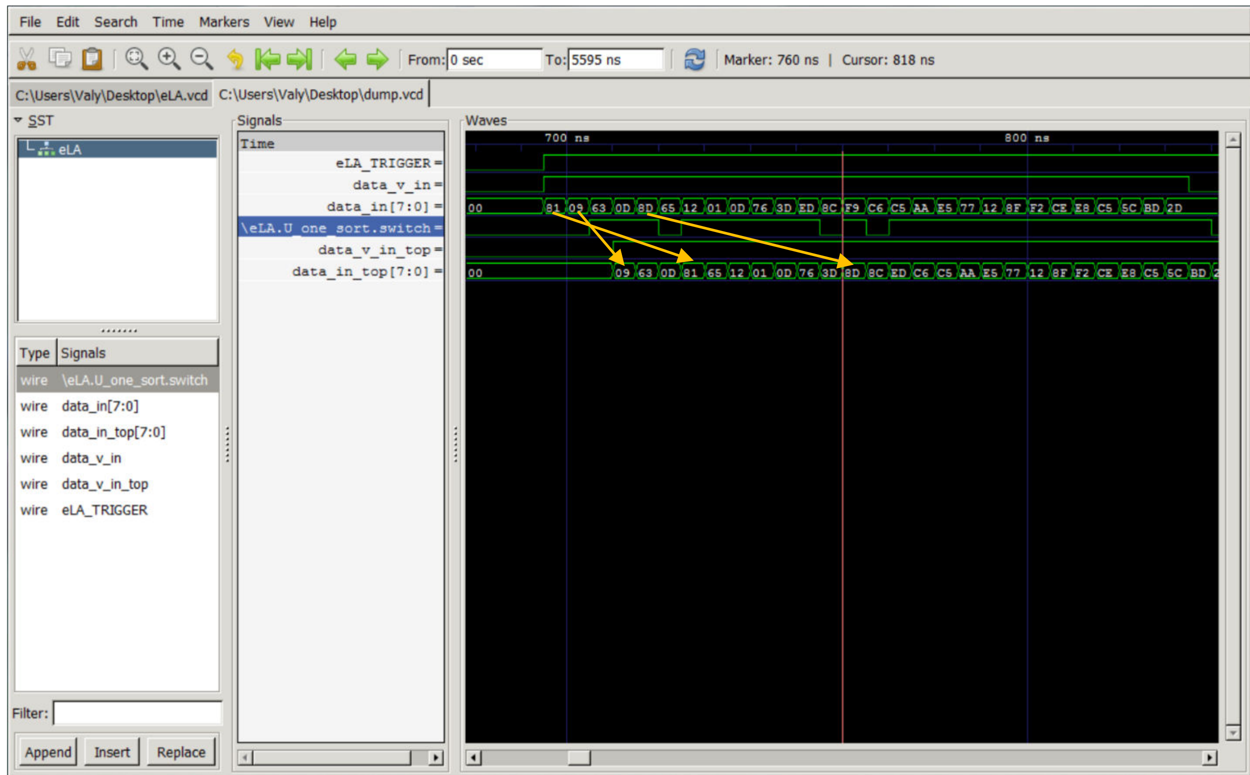


Figure 8 Embedded Logic Analyzer output display

5.3.4. Future-proof Your Design

New customer requirements and bug fixes are a high-risk factor to any SoC design. The use of silicon proven that eFPGA cores from Flex Logix can reduce this risk by allowing new functionality to be added to your existing design, by leveraging eFPGA to augment your existing SoC design.

For example, Figure 5 shows how eFPGA can be added to the Sort design in a way that it can inject or intercept data on the data-path.

The designer may determine that in some scenario, the sort machine is not fast enough and it must be somehow accelerated.

A simple pre-sort of the input data can speed-up the Quick-Sort algorithm, reducing a full set of reads and writes of each packet. Here is an example of such a design.

Figure 9 Pre-Sort

The input to the Pre-Sort is connected to the wire-tap of the *data_in* signal in Figure 5 and the outputs are connected to the bypass MUX (left-side Mux, lower input). The MUX is configured to select the Pre-Sort input and the design increases its performance by reducing the overall sort processing time.

```
// This module does one stage of ascending sort on a stream of data
// new reset signal as the eFPGA has its own reset
`timescale 100ps/100ps

module one_sort (
   // Outputs
   data_out, data_v_out,
   // Inputs
   clk, data_in, data_v_in
   );
   parameter WordSize = 16;

   input                  clk;
   input [WordSize-1:0]   data_in;
   input                  data_v_in;
   output [WordSize-1:0]  data_out;
   output                 data_v_out;

   reg [WordSize-1:0]     data_out, data_s1, data_s2;
   reg                    data_v_out, data_v_s1, data_v_s2;
   wire                   switch;

   assign      switch  = data_v_s1 & data_v_s2 & // don't switch on first and last data
               (data_s2 > data_s1); // ascending order

   // replacing the reset condition with initial for simulation purposes only
   initial begin
```

```
     data_out   = {WordSize{1'b0}};
     data_s1    = {WordSize{1'b0}};
     data_s2    = {WordSize{1'b0}};
     data_v_out = 1'b0;
     data_v_s1  = 1'b0;
     data_v_s2  = 1'b0;
    end

    always @(posedge clk)
      begin
        data_s1     <= #1 data_in;
        data_s2     <= #1 switch ? data_s2 : data_s1;
        data_out    <= #1 switch ? data_s1 : data_s2;

        data_v_out <= #1 data_v_s2;
        data_v_s1  <= #1 data_v_in;
        data_v_s2  <= #1 data_v_s1;
      end

endmodule // one_sort_stage
```

The design suggested here fits within one eFLEX-100 core. On TSMC 16FF+ process, this design runs at 454MHz.

Many other functions and modifications can be performed in similar ways.

6. Conclusions

Debugging capabilities are always welcomed, for initial product bring-up or for observing and debugging functionality later on.

A variety of debugging features are available by implementing the Flex Logix embedded FPGA cores. This includes:

- High observability – the many eFPGA IOs allow for thousands of signals to be monitored
- Event detection – the eFPGA logic can be configured to detect very complex signal patterns to help identify meaningful events
- High visibility – data can be logged by the eLA inside the eFPGA memory or in external SRAMs and then studied offline
- High controllability – the DUT can be augmented by logic modules implemented in the eFPGA, saving costly SoC re-spins and reducing time-to-market of new and improved functionality