CrashTest: Fast and Accurate Fault Analysis Platform

http://www.eecs.umich.edu/crashtest
Contributors: Andrea Pellegrini, Valeria Bertacco, Todd Austin, Kypros Constantinides, Junhao Jiang, Shobana Sudhakar, and Dan Zhang

Andrea Pellegrini *<apellegr@umich.edu>*
Valeria Bertacco *<valeria@umich.edu>*
Todd Austin *<austin@umich.edu>*
University of Michigan,
Electrical Engineering and Computer Science Dept.
2260 Hayward St.
Ann Arbor, MI 48109-2121

CrashTest

CrashTest is a fast, high-fidelity and flexible resiliency analysis system. Starting from a hardware description model of a design under analysis, CrashTest is capable of orchestrating and performing a comprehensive resiliency study by examining the design's reaction to a wide range of faults occurring during its operation. CrashTest injects faults directly in the gate-level netlist of the design and it leverages an FPGA hardware emulation platform to significantly accelerate the analysis process.

For two designs, the OpenSPARC and the Leon3, the suite comes with a number of bitstreams already setup for fault analysis. The CrashTest-ready designs provided at the website include a large number of fault locations (8,064 for the OpenSPARC and 640 for the Leon3) that are ready to be activated. For each CrashTest-ready design, we provide both the original structural Verilog design and the fault-ready Verilog description used to generate the bitstreams to be uploaded on the FPGA board. The basic set of software components needed to run CrashTest includes:

- A set of bitstreams ready to be used in the XUPV5 board
- The firmware to run on the on-board **fault injection manager** to activate the faults at runtime

A detailed presentation of CrashTest, including this document and published articles is available at: http://www.eecs.umich.edu/crashtest

The remainder of this document describes how to setup CrashTest with a proprietary design on an FPGA platform. In addition, the website indicated above provides a number of designs ready for analysis by CrashTest.

Andrea Pellegrini *<apellegr@umich.edu>*
Valeria Bertacco *<valeria@umich.edu>*
Todd Austin *<austin@umich.edu>*
University of Michigan, EECS Dept., 2260 Hayward St. Ann Arbor, MI 48109-2121

Contents

Andrea Pellegrini *<apellegr@umich.edu>*
Valeria Bertacco *<valeria@umich.edu>*
Todd Austin *<austin@umich.edu>*
University of Michigan, EECS Dept., 2260 Hayward St. Ann Arbor, MI 48109-2121

HW and SW Requirements and Setup

Hardware Requirements

The CrashTest suite is designed to work with the XUPV5 Development System from the Xilinx University Program. To use CrashTest you will need the following hardware components:

- Xilinx University Program XUPV5-LX110T Development System

- Xilinx Platform Cable USB or Xilinx Platform Cable USB II

- Serial Cable 9-pin serial to 10-pin header low profile slot plate

- Two null model serial cables to connect the board to a host computer

Andrea Pellegrini *<apellegr@umich.edu>*
Valeria Bertacco *<valeria@umich.edu>*
Todd Austin *<austin@umich.edu>*
University of Michigan, EECS Dept., 2260 Hayward St. Ann Arbor, MI 48109-2121

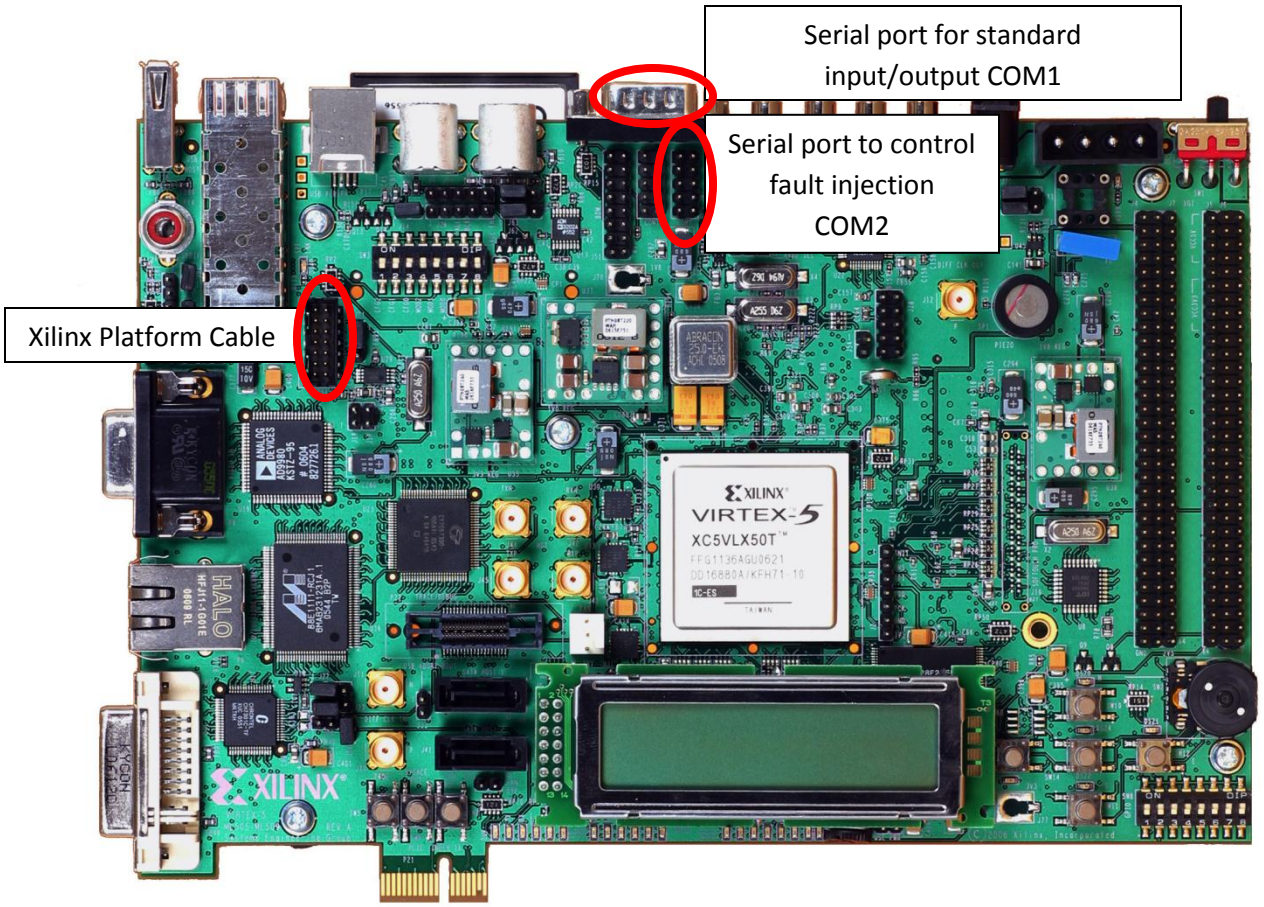


FIGURE 5 – XUP VIRTEX5 BOARD CONNECTIONS

Hardware Setup

Five steps are necessary to setup your board to use CrashTest:

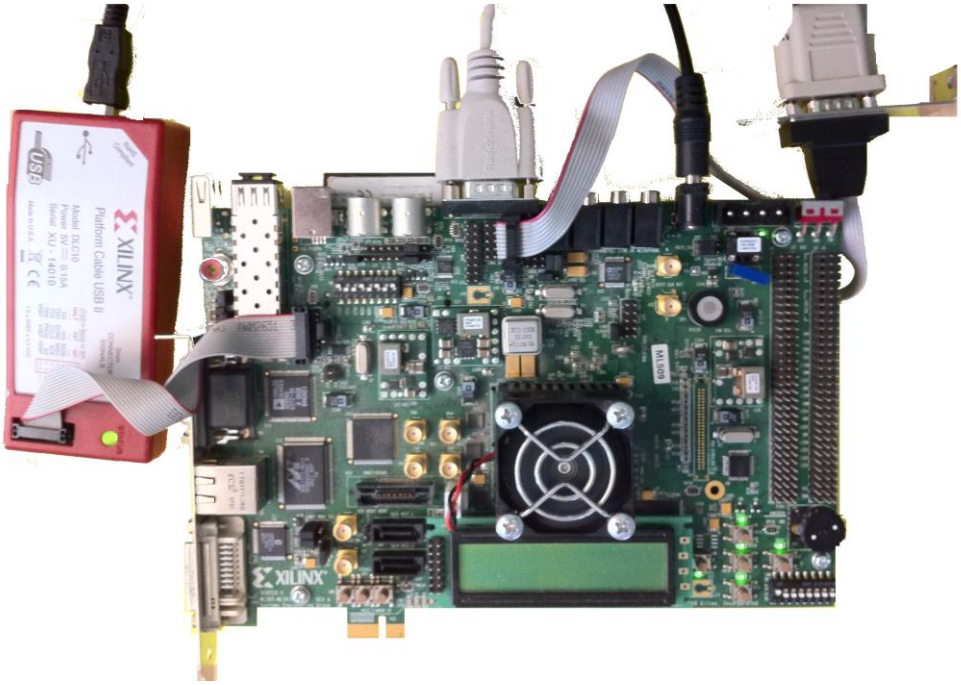
1. Connect the power cable to the board
2. Plug the Xilinx Platform Cable to the JTAG connector in the board. The JTAG port is located on the left part of the board shown above. This connection is used to configure the on-board FPGA device and to
3. Connect a first null model serial cable the COM1 port on the board. Such port is located on the top of the board in Figure5 and it consists of a 9-pin connector with a rounded-corner trapezoid shape. Connect the other end of the serial cable to your host. This communication channel will be used for the standard I/O of the design under test.

Andrea Pellegrini *<apellegr@umich.edu>*
Valeria Bertacco *<valeria@umich.edu>*
Todd Austin *<austin@umich.edu>*
University of Michigan, EECS Dept., 2260 Hayward St. Ann Arbor, MI 48109-2121

4. Connect the “Serial Cable 9-pin serial to 10-pin header low profile slot plate” to the “COM2” connector on the board. Plug the cable such that the low profile slot is pushed all the way in. Make sure the red wire on the bus is towards the outside of the board, as shown here:

5. Connect the second null modem serial cable to the other end of the “Serial Cable 9-pin serial to 10-pin header low profile slot plate”. This end should be a 9-pin connector with a rounded-corner trapezoid shape. This second communication channel will be used to inject faults in the design.

Once all cables are connected your board should look like this one:



Andrea Pellegrini *<apellegr@umich.edu>*
Valeria Bertacco *<valeria@umich.edu>*
Todd Austin *<austin@umich.edu>*
University of Michigan, EECS Dept., 2260 Hayward St. Ann Arbor, MI 48109-2121

Software Requirements

The CrashTest suite is designed to work with the Xilinx tool chain and has been developed and tested with Xilinx EDK System Version 11.5 and Synplicity Synplify Pro 9.1. Additional software (such as Microsoft Hyperterminal) is necessary to establish the communication between your host machine and the design running on the FPGA board.

The suite comes with a number of designs already setup for fault analysis. If the only goal is to run these design, then the complete EDK system is not necessary, and the only software applications required are:

- Xilinx Impact: necessary to download the configuration bitstream on the FPGA device
- Xilinx Microprocessor Debugger: necessary to connect to on-board processor running the CrashTest routines

Andrea Pellegrini *<apellegr@umich.edu>*
Valeria Bertacco *<valeria@umich.edu>*
Todd Austin *<austin@umich.edu>*
University of Michigan, EECS Dept., 2260 Hayward St. Ann Arbor, MI 48109-2121

The CrashTest Framework

CrashTest allows researchers to test the behavior of a design affected by faults. For this purpose, we publish the scripts required to generate fault-ready designs starting with any Verilog netlists. Furthermore, we provide full access to our project files for Synplify and EDK to generate the bitstreams for the FPGA. The current version of CrashTest has been evaluated using the Synopsys GTECH technology-independent cell library to create fault-ready netlists and it supports the following fault models: ***bridge***, ***path-delay***, ***stuck-at***, ***stuck-open***, **and** ***transient***.

In this chapter we will explain how to fault-enable a design. We first detail the fault models available with our tool. We then explain how to use the script to inject faults in a series of location in a Verilog netlist. Finally, we describe how the fault-enabled design is connected with the rest of the CrashTest system, and how the fault locations are controlled at runtime from the on-board fault injection manager.

Andrea Pellegrini *<apellegr@umich.edu>*
Valeria Bertacco *<valeria@umich.edu>*
Todd Austin *<austin@umich.edu>*
University of Michigan, EECS Dept., 2260 Hayward St. Ann Arbor, MI 48109-2121

Fault Models

The fault models injected with CrashTest are the following:

- **Bridge -** faults are inserted at the input ports of gates with 2 or more inputs.
- **Path delay -** faults are inserted at the input of Flip Flop components.
- **Transient -** these faults modify the value of the input that is stored in Flip Flop Components.
- **Stuck-at -** faults can be inserted in any net in the module.
- **Stuck-open -** faults can be inserted in any net in the module.

Bridge and **stuck-at** faults support two different fault values: 0 and 1. The fault value set for the bridge fault determines the value of the faulty nets when the values of the two faulty nets are different. The fault value for the stuck-at fault sets the value of the net subject to the fault.

In CrashTest any fault-enabled design is enhanced with inputs to control the fault injection. The logic that models the faults in the design is connected through scan chains and the faulty behavior can be activated at runtime through software. A hardware wrapper is provided to allow the communication between the fault injection manager and the fault locations in the design.

For each fault-enabled design the following inputs are added:

```
input          crashtest_fe_clk;      // CrashTest: Fault Enable Clock from wrapper
input [31:0]   crashtest_fe_data;     // CrashTest: Fault Enable Data from wrapper
input          crashtest_fv_clk;      // CrashTest: Fault Value Clock from wrapper
input [31:0]   crashtest_fv_data;     // CrashTest: Fault Value Data from wrapper
```

Two different scan chains are inserted in the design: a first one, crashtest_fe – or fault enable, is used to activate the faulty behavior. The second one, crashtest_fv – or fault value, is only used for the faults models that require a fault bias (stuck-at and bridge).

For instance, a stuck-at fault is modeled as follows:

```
// Fault Number: 0; Faulty Net: n584; Scan Chain: 0; Depth: 0
always@(posedge crashtest_fe_clk)
begin
        crashtest_fe_sc_0[0] <= crashtest_fe_data[0];
end
always@(posedge crashtest_fv_clk)
begin
        crashtest_fv_sc_0[0] <= crashtest_fv_data[0];
end
assign n584 = crashtest_fe_sc_0[0] ? crashtest_fv_sc_0[0] : crashtest_stuck_at_0[0];
```

In this case, the fault modeled is a stuck-at, and the value of the fault is taken from the data in the fv scan chain. Note that this fault is the first one in the scan chain, so it is fed directly by the CrashTest inputs of the design.

Andrea Pellegrini *<apellegr@umich.edu>*
Valeria Bertacco *<valeria@umich.edu>*
Todd Austin *<austin@umich.edu>*
University of Michigan, EECS Dept., 2260 Hayward St. Ann Arbor, MI 48109-2121

Note that this fault models a fault in the network called "n584". The gate that was driving that network in the original design is changed to drive a new wire, crashtest_stuck_at_0[0]:

```
GTECH_BUF U652 (.A(n855), .Z(crashtest_stuck_at_0[0]));
```

For more information about the fault models we refer to the paper "CrashTest: A fast High-Fidelity FPGA-based Resiliency Analysis Framework" (www.eecs.umich.edu/~valeria/research/publications/ICCD08CrashTest.pdf).

Generating Fault-Enabled Netlists

In this section we will illustrate how a netlist can be instrumented with the logic necessary to perform a fault campaign. Faults can be inserted into a module that is first synthesized against the GTECH technology independent library.

To insert faults in a design, you will need to supply the Perl script **crashtest_injector.pl** with the following arguments:

--input_netlist=<file_name>

Specifies the URL of the Verilog files that compose the design. This option can be repeated as many times as needed to include all the Verilog files used in the design.

--include_directory= <directory_path>

Contains the URL of the directory that stores the header files needed by the Verilog design.

-- temporary_directory = <directory_path>

Directory where the script will store temporary data. Maintaining the same temporary directory for all fault injections will speed up script runtime.

--output_directory = <directory_path>

Directory where the fault-ready netlist will be generated

--output_filename= <name_output_file>

Name of file containing the fault-ready netlist

--library =<path_to_the _library>

File containing the behavioral models of the library components used to synthesize the netlist.

--clock_net=<name_clock_net>

Name of the clock network in the design. Multiple assignments for this option are possible if the design contains multiple clock nets.

Andrea Pellegrini *<apellegr@umich.edu>*
Valeria Bertacco *<valeria@umich.edu>*
Todd Austin *<austin@umich.edu>*
University of Michigan, EECS Dept., 2260 Hayward St. Ann Arbor, MI 48109-2121

-- reset_net= <name_reset_net>

Name of the reset network in the design, if any is present.

--top_module= <top_module_name>

Name of the Verilog module where the faults will be inserted. To allow the script to work properly, the Verilog file containing this module should be a structural Verilog file.

--scan_chains = <num_scan_chains>

Width of the scan chains in the design. If not specified, its default value is 32.

--random_seed=<number>

This option allows you to change the random seed used to randomly select the fault locations.

--bridge=<num_faults>

This option defines the number of bridge faults to insert.

--path_delay=<num_faults>

This option defines the number of path-delay faults to insert.

--stuck_at=<num_faults>

This option defines the number of stuck-at faults to insert.

--stuck_open=<num_faults>

This option defines the number of stuck-open faults to insert.

--transient=<num_faults>

This option defines the number of transient faults to insert.

Note that only one type of faults can be inserted in the design at every run.
The Perl script **crashtest_injector.pl**, requires the following modules to function properly:

- `Data::Dumper;`
- `Verilog::Netlist;`
- `Verilog::Getopt;`

The Verilog module for Perl can be downloaded at: http://www.veripool.org/wiki/verilog-perl

Andrea Pellegrini *<apellegr@umich.edu>*
Valeria Bertacco *<valeria@umich.edu>*
Todd Austin *<austin@umich.edu>*
University of Michigan, EECS Dept., 2260 Hayward St. Ann Arbor, MI 48109-2121

Usage example

Assume that you would like to inject 128 faults in a module of the OpenSPARC T1 called TLU. Also, imagine that the design uses as clock an input signal called "`rclk`". Finally, assume that the OpenSPARCT1 directory is located at /opt/OpenSPARCT1.1.7/, and the netlist of the tlu synthesized against the GTECH library is in the file /opt/OpenSPARCT1.1.7/synthesized_netlists/tlu.nl.v

The following command will create a new file, called tlu.stuck_at.v, containing the logic required to model 128 faults. Since the width of the scan chains is not defined, the script will insert 32 scan chains for the "fault enable" signals and 32 scan chains for the "fault value" signals. Since 128 fault will be injected in the module the depth of the scan chains will be 4.

```
perl crashtest_injector.pl \

        --stuck_at=128 \

        --clock_net=rclk \

        --input_netlist=/opt/OpenSPARCT1.1.7/synthesized_netlists/tlu.nl.v \

        --top_module=tlu \

        --output_filename=tlu.stuck_at.v \

        --headers_directory=/opt/OpenSPARCT1.1.7/design/sys/iop/include/ \

        --verilog_directory=/opt/OpenSPARCT1.1.7/design/sys/iop/srams/rtl/ \

        --library=/opt/synopsys-synth-2006.06-sp4/packages/gtech/src_ver/gtech_lib.v
```

Using other technological libraries

The script relies on parsing the Verilog files to understand as much as possible of the design, but few parameters need to be defined to adapt it to use different technologic libraries. In this section we list the list of changes needed to adapt the script to function on different libraries:

A variable is set to match the name of the input that is used as clock for the sequential elements in the design. CP is used for GTECH, for other libraries you need to change it to match your library. You can have a list of port names that are used as clocks:

```
my @clock_ports = ("CP");
```

A variable contains the substring common to all components in the library to extract information about them and separate them from other components. Typically every component in a determined library is characterized by a determined string. For the GTECH library, the string is "GTECH".

```
my @gates_name_filter = ("GTECH");
```

This variable contains the full name of the flip flops components defined in the netlist. You can alter this variable with the list of names of the flip-flop cells in your design:

```
my @flip_flops_name_filter = ("GTECH_FD1");
```


Andrea Pellegrini *<apellegr@umich.edu>*
Valeria Bertacco *<valeria@umich.edu>*
Todd Austin *<austin@umich.edu>*
University of Michigan, EECS Dept., 2260 Hayward St. Ann Arbor, MI 48109-2121

For each supported flip flop, indicate which input should be used to insert path-delay and transient faults:

```
my %flip_flops_input_port = ();
$flip_flops_input_port{"GTECH_FD1"} = "D";
```

For each supported Flip flop, also indicate which input is used as clock signal:

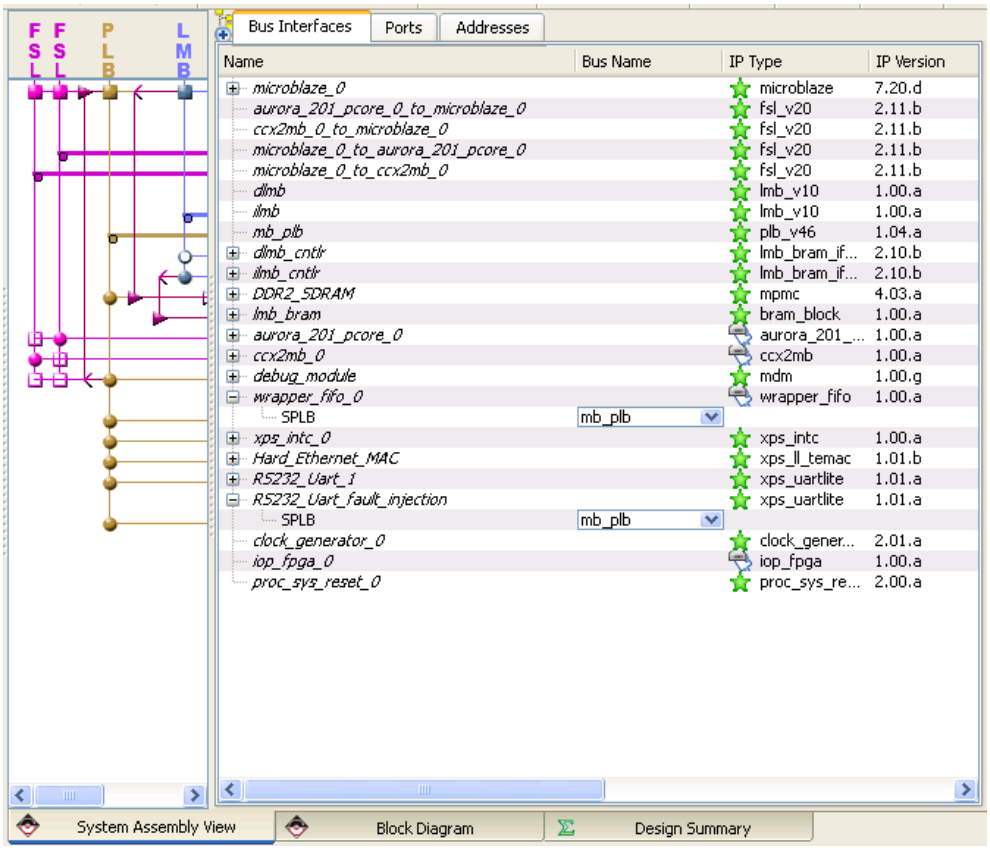
```
my %flip_flops_clock_port = ();
$flip_flops_clock_port{"GTECH_FD1"} = "CP";
```

If the design contains a clock tree, the fault injection script needs to reconstruct it to understand which nets compose it. Faults in the clock tree are currently not supported since they require the insertion of special FPGA components that allow clock-gating. A variable is set to allow the system know which components can be part of the clock tree (usually only buffers and inverters):

```
my @cells_allowed_in_clock_tree = ("GTECH_NOT", "GTECH_BUF");
```

CrashTest Xilinx EDK Project

A wrapper has been developed to allow the communication between the fault injection manager and the fault-enabled design. This module, called “wrapper_fifo”, is connected to the PLB bus of the system and allows the software running on the fault injection manager to control the faults in the design under test:

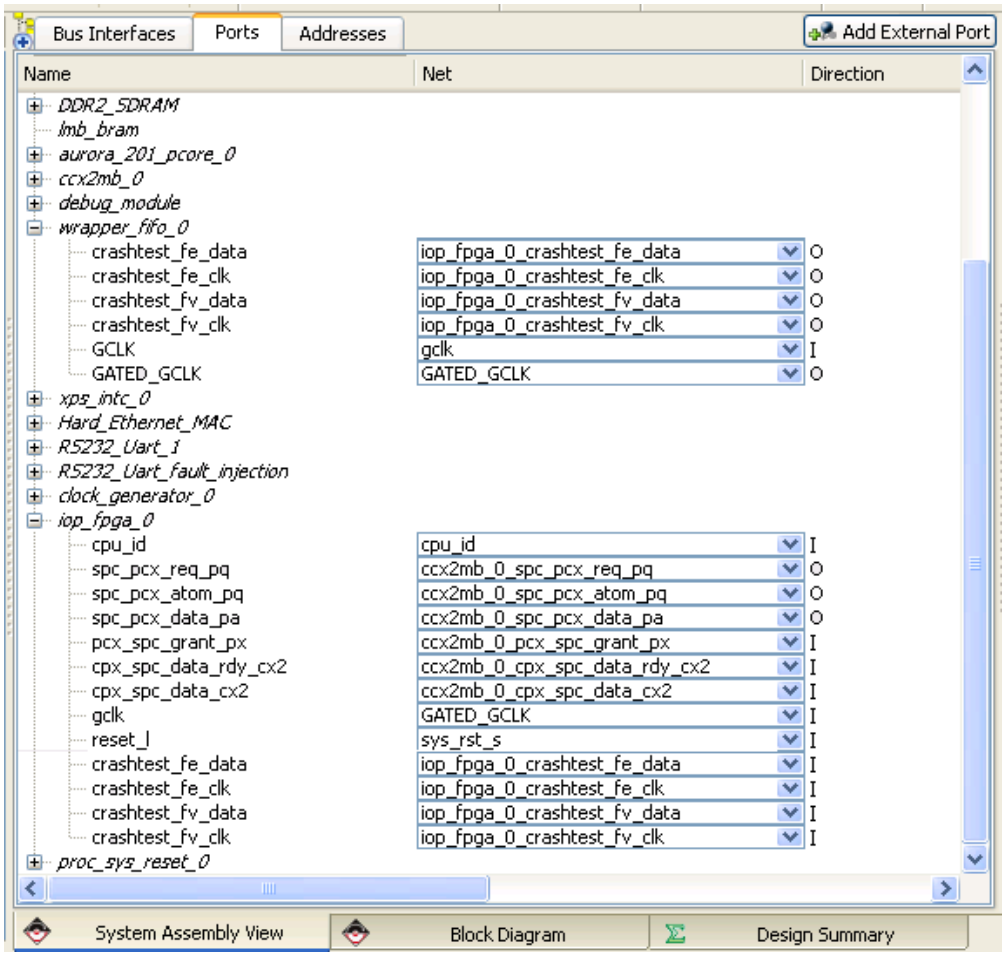


Andrea Pellegrini *<apellegr@umich.edu>*
Valeria Bertacco *<valeria@umich.edu>*
Todd Austin *<austin@umich.edu>*
University of Michigan, EECS Dept., 2260 Hayward St. Ann Arbor, MI 48109-2121

The wrapper_fifo is interfaced with the CrashTested design through 5 ports:

1. `crashtest_fe_clk`
2. `crashtest_fe_data`
3. `crashtest_fv_clk`
4. `crashtest_fv_data`
5. `GATED_GCLK`

The first four ports are needed to enable the faults at runtime and are connected to the relative scan chains. `crashtest_fe_clk` and `crashtest_fv_clk` are used to clock the values in the scan chains, while `crashtest_fe_data` and `crashtest_fv_data` will carry the data that will be loaded at the beginning of the relative scan chains. The signal GATED_GCLK is the clock provided to the design. To avoid spurious behaviors in the design due to the fault injection procedure, the clock provided to the design is paused while the scan chains that control the fault locations are loaded.



Andrea Pellegrini *<apellegr@umich.edu>*
Valeria Bertacco *<valeria@umich.edu>*
Todd Austin *<austin@umich.edu>*
University of Michigan, EECS Dept., 2260 Hayward St. Ann Arbor, MI 48109-2121

The wrapper_fifo exposes three registers needed to control the fault injection and the FIFO. The three registers accessible from software are:

1. FE Counter - register used to let the wrapper know how much data will be sent to the fault enable scan chains
2. FV Counter - register used to let the wrapper know how much data will be sent to the fault value scan chains
3. Clk Counter - counts the number of cycles in the design since the last fault injection

The following functions are provided to write and read the registers:

```
WRAPPER_FIFO_Write_FE_Counter((Xuint32*) baseaddr, (Xuint32*)data);
WRAPPER_FIFO_Write_FV_Counter((Xuint32*) baseaddr, (Xuint32*)data);
```

These functions take as inputs the base address of the wrapper_fifo instantiated in the design and the pointer to an integer value that contains the value to write in the register

```
WRAPPER_FIFO_Read_FE_Counter((Xuint32*) baseaddr, (Xuint32*)data);
WRAPPER_FIFO_Read_FV_Counter((Xuint32*) baseaddr, (Xuint32*)data);
WRAPPER_FIFO_Read_Clk_Counter((Xuint32*) baseaddr, (Xuint32*)data);
```

These functions take as inputs the base address of the wrapper_fifo instantiated in the design and the pointer to the variable that will be written with the value read from the register.

Data to the scan chains is sent by software with the following procedure:

```
WRAPPER_FIFO_WriteToFIFO((Xuint32*) baseaddr, (Xuint32*)data);
```

Note that the FIFO is shared among the fault-enable and the fault-value logic.

Fault injection procedure

Activating faults at runtime in the fault-enabled design is a delicate procedure that requires a precise sequence of operations. The two scan chains (fault-value and fault-enable) need to be loaded in a strict order, where the fault-value scan chain is always loaded first. This is important to avoid your fault injection procedure to cause system's spurious behavior.

The FIFO is shared among the two scan chains, and the counters associated to each scan chain determine which scan chain will be clocked and thus will load the data. For instance, if the the fault-value scan chain has to be loaded, the user will proceed with the following steps:

1. Write the number of times the fault-value scan chain will be clocked in FV_Counter
2. Write in the FIFO as many values as the depth of the scan-chain

The wrapper will load one value at the time in the selected scan chain (in this case the fault-enable), consuming the values in the FIFO and decrementing the respective counter. At the end of the procedure, if the scan chains were properly loaded, the value of the counter associated with the scan chain should be

Andrea Pellegrini *<apellegr@umich.edu>*
Valeria Bertacco *<valeria@umich.edu>*
Todd Austin *<austin@umich.edu>*
University of Michigan, EECS Dept., 2260 Hayward St. Ann Arbor, MI 48109-2121

zero. While the scan chains are loaded the clock to the design is paused and the `Clk_Counter` register is reset.

The fault injection procedure to inject the fault number `fault_index` works as follows:

```
#define         SCAN_CHAINS_WIDTH       32
#define         SCAN_CHAINS_DEPTH       FAULT_NUM / SCAN_CHAINS_WIDTH
#define         FV_COUNT                SCAN_CHAINS_DEPTH
#define         FE_COUNT                SCAN_CHAINS_DEPTH
void inject_faults(int fault_index) {
        int scan_chain_iter = fault_index / SCAN_CHAINS_WIDTH;
        int FE_Counter = 0;
        WRAPPER_FIFO_Write_FE_Counter(baseaddr, &FE_Counter);
        int FV_Counter = FV_COUNT;
        WRAPPER_FIFO_Write_FV_Counter(baseaddr, &FV_Counter);
        for(FV_index = 0; FV_index < FV_COUNT; FV_index++) {
                if(FV_index == scan_chain_iter) {
                        data_fifo = stuck_at << (fault_index % SCAN_CHAINS_WIDTH);
                }
                else {
                        data_fifo = 0x00000000;
                }
                WRAPPER_FIFO_WriteToFIFO(baseaddr, &data_fifo);
        }
        FE_Counter = FE_COUNT;
        WRAPPER_FIFO_Write_FE_Counter(baseaddr ,&FE_Counter);
        FV_Counter = 0;
        WRAPPER_FIFO_Write_FV_Counter(baseaddr, &FV_Counter);
        for(FE_index = 0; FE_index < FE_COUNT; FE_index++) {
                if(FE_index == scan_chain_iter) {
                        data_fifo = 1 << (fault_index % SCAN_CHAINS_WIDTH);
                }
                else {
                        data_fifo = 0x00000000;
                }
                WRAPPER_FIFO_WriteToFIFO(baseaddr, &data_fifo);
        }
}
```

This procedure is executed by the fault injection manager and `baseaddr` is assumed to store the base address of the wrapper_fifo instantiated in the design.

Andrea Pellegrini *<apellegr@umich.edu>*
Valeria Bertacco *<valeria@umich.edu>*
Todd Austin *<austin@umich.edu>*
University of Michigan, EECS Dept., 2260 Hayward St. Ann Arbor, MI 48109-2121

OpenSPARC T1

CrashTest for OpenSPARC T1 is based on the original implementation provided by Sun, and some experience with the original tool is recommended before using our design.

Due to the fact that injecting faults at the gate-level increases the resource needed by the design on FPGA, the OpenSPARC T1 has been partitioned in 13 modules and faults where injected on the different modules separately. No faults have been injected in memory arrays: caches, register files, TLBs, Load/Store queues, etc., are thus fault free. Note, however, that the logic that connects to such memory elements can be subject to faults. The OpenSPARC has been split in the following logic modules:

1. lsu - Load store unit
2. sparc_exu_alu - Arithmetic logic unit
3. sparc_exu_byp - Muxes for the bypass logic
4. sparc_exu_div - Divider
5. sparc_exu_ecc - Ecc check and correction
6. sparc_exu_ecl - Execution Unit Control Logic
7. sparc_exu_rml - Register management logic
8. sparc_exu_shft - Right and left shifter
9. sparc_ffu - Floating point frontend unit
10. sparc_ifu - Instruction fetch unit
11. sparc_mul_top - Multiplier
12. spu - Stream processing unit
13. tlu - Trap logic and memory management unit

For each module, 5 different bitstreams are provided. Each bitstream contains a hardware module enhanced with 128 randomly selected fault locations for the different models: **bridge, path delay, stuck-at, stuck-open, and transient**. The faults have been inserted in the GTECH netlist of the modules, and synthesized with Synopsys Design Compiler.

For the OpenSPARC, every bitstream is named with the following naming convention:

<module_name>.<fault_model>.bit

For instance, lsu.stuck_at.bit is a bitstream where 128 stuck-at faults were inserted in the lsu module.

Path delay and transient faults have not been inserted in the module “sparc_exu_shft” since this module does not contain any flip-flop. Thus the bitstreams provided for those two cases are fault-free.

To enable the activation of faults at runtime, a customized firmware for the Microblaze is provided. As long as no faults are activated in the design, the CrashTest bitstreams work exactly as the originals provided by Sun, and for more information about how to run programs or boot an operating system on the platform we refer the reader to Chapter 6 of the “OpenSPARC T1 Processor Design and Verification User’s Guide” (www.opensparc.net).

Andrea Pellegrini *<apellegr@umich.edu>*
Valeria Bertacco *<valeria@umich.edu>*
Todd Austin *<austin@umich.edu>*
University of Michigan, EECS Dept., 2260 Hayward St. Ann Arbor, MI 48109-2121

Injecting Faults

This section details how to proceed to activate the fault locations on OpenSPARC runtime. We will walk through an example where we will enable a stuck-at fault in the lsu of the OpenSPARC T1.

1. **Download the bitstream to configure the FPGA device**
   To download the bitstream, invoke impact and download the following bitstream to configure the FPGA device:
   `lsu.stuck_at.bit`

2. **Start any terminal window, such as Hyperterminal, and connect to the serial port connected to the standard IO of the FPGA board (COM1 on the board in Figure 5).**
   The communication settings should be set to 9600 baud, data 8 bits, parity none, stop bits 1, and flow control none.

3. **Start any terminal window, such as Hyperterminal, and connect to the serial port that is connected to the secondary serial port of the FPGA board (COM2 on the board in Figure 5).**
   The communication settings should be set to 9600 baud, data 8 bits, parity none, stop bits 1, and flow control none.

4. **Download the fault_manager and the benchmark**
   Update the memory on the board with the fault_manager (modified OpenSPARC firmware) and the program to run, invoke xmd and type the following commands:

```
XMD% connect mb mdm -cable type xilinx_platformusb port auto
XMD% dow fault_manager/executable.elf
XMD% dow -data proms/1c1t_prom.bin 0x8ff00000
XMD% dow -data hello_world.mem.image.gz 0x8af00000
XMD% run
```

5. **Running the system.**
   After inputting the last command, the system will start running and outputting on the Hyperterminal connected to the standard IO of the board.

   The output from the test will appear in the terminal window. If a firmware that supports CrashTest has been loaded the string "`CRASHTEST defined`" will be output to the console.

```
CRASHTEST defined

MBFW_INFO: Uncompressing ram_disk .....
MBFW_INFO: Uncompressed ram_disk
MBFW_INFO: Initializing OpenSPARC T1 DRAM from 0x80100000 to 0x8AF00000
MBFW_INFO: Initialized OpenSPARC T1 DRAM
CRASHTEST:  - Initializing interrupt handler
MBFW_INFO: XIntc interrupt controller initialized.
```


Andrea Pellegrini *<apellegr@umich.edu>*
Valeria Bertacco *<valeria@umich.edu>*
Todd Austin *<austin@umich.edu>*
University of Michigan, EECS Dept., 2260 Hayward St. Ann Arbor, MI 48109-2121

```
CRASHTEST:  - Enabling interrupts
MBFW_INFO: Setting Temac operating speed to 100 Mbit/sec
MBFW_INFO: Ethernet controller initialization completed.
MBFW_INFO: Network controller initialized.
MBFW_INFO: Microblaze firmware initialization completed.

MBFW_INFO: Powering on OpenSPARC T1
``Alive and well ...
Strand start set = 0x1
Total physical mem = 0xac00000
Scrubbing the rest of memory
Number of strands = 0x1
membase = 0x0
memsize = 0x1000000
physmem = 0xac00000
 done
returned status 0x0
setup everything else
Setting remaining details
Start heart beat for control domain
```

6. **Enable a fault.**
   To inject a fault, type a character to the terminal connected to the COM2 serial port on the board (see Figure 5). This will cause the firmware to stop the execution of the benchmark and wait for user's inputs about the fault to inject. In this example the bitstream with stuck-at faults was used. The firmware will ask which fault to inject and, if the fault model supports it, which value to inject in the fault location. CrashTest will report the number of fault that will be enabled in hexadecimal. For instance, in the following log the fault location enabled was the number 78.

```
CRASHTEST: ##########################################
CRASHTEST: Starting the Fault Injection
CRASHTEST: Which fault should be enabled(between 1 and 1024):
CRASHTEST: StuckAt0 or StuckAt1:
CRASHTEST: Injecting fault number:
CRASHTEST: 0000004e
CRASHTEST: Fault Injection Ended Successfully
CRASHTEST: ##########################################

MBFW_ERROR: translate_addr(): Couldn't find T1 to Microblaze mapping for
physical address 0x9702000280

MBFW_INFO: pcx_pkt: addr_hi_ctrl  0x48201B97
MBFW_INFO: pcx_pkt: addr_lo       0x02000280
MBFW_INFO: pcx_pkt: data1         0x00000000
MBFW_INFO: pcx_pkt: data0         0x00000030
```


Andrea Pellegrini *<apellegr@umich.edu>*
Valeria Bertacco *<valeria@umich.edu>*
Todd Austin *<austin@umich.edu>*
University of Michigan, EECS Dept., 2260 Hayward St. Ann Arbor, MI 48109-2121

Leon3

CrashTest for Leon3 builds on the original design provided by Gaisler research, enhancing the design with a microprocessor (Microblaze) that acts as fault manager. In this configuration, faults are injected in the hardware of the Leon3 microprocessor, but not in its memory arrays (RF, TLB, caches). All other components of the SoC are original.

Bitstreams with 5 fault types are provided: **bridge, path delay, stuck-at, stuck-open, and transient**. Each one of them contains 128 faults in random locations across the processor. As for the OpenSPARC the faults have been modeled in the GTECH netlist of the Leon3s module, synthesized with Synopsys Design Compiler.

For the Leon3, every bitstream is named in the following fashion:

Leon3.<fault_model>.bit

To dynamically enable fault each locations at runtime, a software program to be run on the Microblaze, the fault manager, is provided. As long as no faults are activated in the design, the CrashTest bitstreams work exactly as the originals provided by Gaisler, so for more information about how to run programs or boot an operating system on the platform we refer the reader to the documents available at www.gaisler.com.

Injecting Faults

This section will detail how faults can be injected in the system at runtime. We will walk through an example where we will enable a stuck-at fault in the lsu of the OpenSPARC T1.

1. **Download the bitstream to configure the FPGA device**
   To download the bitstream, invoke impact and download the following bitstream to configure the FPGA device:
   `Leon3.stuck_at.bit`

2. **Start any terminal window, such as Hyperterminal, and connect to the serial port connected to the standard IO of the FPGA board (COM1 on Figure 5).**
   The communication settings should be set to 38400 baud, data 8 bits, parity none, stop bits 1, and flow control none.

3. **Download the fault_manager and the benchmark**
   To update the memory on the board with the fault_manager and the program to run, invoke xmd and type the following commands:

```
XMD% connect mb mdm -cable type xilinx_platformusb port auto
XMD% dow fault_manager_leon3/executable.elf
XMD% run
```


Andrea Pellegrini *<apellegr@umich.edu>*
Valeria Bertacco *<valeria@umich.edu>*
Todd Austin *<austin@umich.edu>*
University of Michigan, EECS Dept., 2260 Hayward St. Ann Arbor, MI 48109-2121

```
XMD% terminal -jtag_uart_server 4321
```

Then open hyperterminal and set it to connect to: TCP/IP localhost:4321

4. **Running the system.**
   The system will start running and outputting on the Hyperterminal connected to the standard IO of the board.

   The output from the test will appear in the terminal window. If a firmware that supports CrashTest has been loaded the string "`CRASHTEST defined`" will be output to the console.

```
CRASHTEST defined
...
Booting Linux
Booting Linux...
PROMLIB: Sun Boot Prom Version 0 Revision 0
Linux version 2.6.21.1 (root@chomper) (gcc version 3.4.4) #1 Thu Apr 3
20:29:42
EDT 2008
ARCH: LEON
...
```

5. **Enable a fault.**
   To inject a fault, type a character to the terminal connected to the microblaze. This will cause the microblaze to stop the execution of the benchmark and wait for user's inputs about the fault to inject. In this example the bitstream with stuck-at faults was used. The firmware will ask which fault to inject and, if the fault model supports it, which value to inject in the fault location. CrashTest will report the number of fault that will be enabled in hexadecimal. For instance, in the following log the fault location enabled was the number 78.

```
CRASHTEST: ##########################################
CRASHTEST: Starting the Fault Injection
CRASHTEST: Which fault should be enabled(between 1 and 1024):
CRASHTEST: StuckAt0 or StuckAt1:
CRASHTEST: Injecting fault number:
CRASHTEST: 0000004e
CRASHTEST: Fault Injection Ended Successfully
CRASHTEST: ##########################################
```


Andrea Pellegrini *<apellegr@umich.edu>*
Valeria Bertacco *<valeria@umich.edu>*
Todd Austin *<austin@umich.edu>*
University of Michigan, EECS Dept., 2260 Hayward St. Ann Arbor, MI 48109-2121

Once the fault is injected, the system will keep running the benchmark:

```
Mount-cache hash table entries: 512
Unable to handle kernel paging request at virtual address 00100000
tsk->{mm,active_mm}->context = ffffffff
tsk->{mm,active_mm}->pgd = fc000000
              \|/ ____ \|/
              "@'/ ,. \`@"
              /_| \__/ |_\
                 \__U_/
swapper(0): Oops [#1]
PSR: f3900fc7 PC: f006626c NPC: f0066270 Y: 5a000000    Not tainted
%G: 0000003b 00100100  f093fca0 f093fca0  ffffffff 00000000  f000e000 00000007
%O: 00000000 00000000  f0102a44 00000044  00000001 f0135038  f000fc50 f00bce98
%L: f02edbe0 f00ff0d0  f093fca0 0000003c  f0939800 f093fca0  00000000 00200200
%I: f0938f80 000000d0  00000001 f000fd18  f093fca8 00100100  f000fcb8 f00660d0
Caller[f00660d0]Caller[f00b19e8]Caller[f009b534]Caller[f009b7d4]Caller[f0120cc0
]
Caller[f0120b20]Caller[f010cffc]Caller[f010c7a0]Caller[00000000]Instruction
DUMP
: ee24a004  c2054000  c2248000 <e4206004> e4254000  ea24a004  80a4e000
34bfffc9
  e4054000
Kernel panic - not syncing: Attempted to kill the idle task!
Press Stop-A (L1-A) to return to the boot prom
IU in error mode (tt = 0x03)
f0012028  81cc8000   rett  %l2
```


Andrea Pellegrini *<apellegr@umich.edu>*
Valeria Bertacco *<valeria@umich.edu>*
Todd Austin *<austin@umich.edu>*
University of Michigan, EECS Dept., 2260 Hayward St. Ann Arbor, MI 48109-2121