A self-scrubber for FPGA-based systems

G. Alonzo Vera
alonzo.vera@micro-rdc.com

Xiaoyin (Mark) Yao
mark.yao@micro-rdc.com

Keith Avery
Air Force Research Laboratory,
Space Vehicles Directorate, Kirtland Air Force Base, N.M.

This material is based upon work supported by the United States Air Force under Contract No. FA9453-08-M-0096. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force.
Agenda

- Background and other reported solutions
- Self-scrubbing basics
- Architecture
- Algorithms
- Roadmap
Agenda

- Background and other reported solutions
- Self-scrubbing basics
- Architecture
- Algorithms
- Roadmap
Scrubbing is the process of removing errors from a memory's content by re-writing it periodically with correct values. Usually used to address the issue of error accumulation that could defeat TMR.

First approach was blind scrubbing (open-loop scrubbing)
- This can be done in the fly (without stopping the system) or by “rebooting” the device every so often if the system can tolerate down time.
- Susceptibility to SEFIs

Current more sophisticated approaches are readback – detect – scrub (closed-loop scrubbing)
- Frame based scrubbing
- Design dependent scrubbing strategies.

Scrubber can be external or internal.

Not all resources can be scrubbed.
- SRL16s, LUT RAM
- BRAMs
Summary of reported solutions

- Commonly implemented: Blind scrubbing
  - NASA/GSFC Radiation effects analysis group V4 scrubber (06/2007)

- Some examples of read, detect, scrub
  - Sandia-Xilinx Virtex FPGA SEU experiment on the International Space Station. Cross scrubbing between V4 and V5.
  - LANL flight experiment for Virtex I and its derivations
  - BYU ICAP-based scrubber. Uses picoblaze
  - Radix4 configuration scrubber
  - Aeroflex Scrubber, an implementation of 989
  - XAPP 714: self scrubber, not longer supported
  - XAPP 779: V2 scrubber
  - XAPP 988: V4 scrubber
  - XAPP 989: lastest supported solution from Xilinx (V2/V4)
Agenda

- Background and other reported solutions
- **Self-scrubbing basics**
- Architecture
- Algorithms
- Roadmap
Self Scrubbing

Requirements:

- Internal access to configuration memory (ICAP in Xilinx devices).
- Address self-susceptibility to SEU.
- Error detection (and correction?) capability.
- Custom floorplan to separate system being scrubbed from the scrubber itself.
- Reduced logical resources consumption
- External (?) safe storage for golden copies of the bistream.
- Flexibility to implement selective, frame-based scrubbing.
Agenda

- Background and other reported solutions
- Self-scrubbing basics
- Architecture
- Algorithms
- Roadmap
Architecture

- Top level block diagram where scrubber and payload share the logical resources of the FPGA.

  - small footprint ..
  - flexibility ..
  - reporting ..
  - external storage ..
  - read/write access ..
  - error detection ..
Architecture

- XCV5LX50 Layout example.
- Constrained implementation to separate the scrubber from device's payload.
- TMR version ~x3.2

Device Utilization Summary (estimated values)

<table>
<thead>
<tr>
<th>Logic Utilization</th>
<th>Used</th>
<th>Available</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Slice Registers</td>
<td>750</td>
<td>26800</td>
<td>2%</td>
</tr>
<tr>
<td>Number of Slice LUTs</td>
<td>785</td>
<td>26800</td>
<td>2%</td>
</tr>
<tr>
<td>Number of fully used Bit Slices</td>
<td>243</td>
<td>1292</td>
<td>18%</td>
</tr>
<tr>
<td>Number of bonded IOBs</td>
<td>48</td>
<td>220</td>
<td>21%</td>
</tr>
<tr>
<td>Number of Block RAM/FIFO</td>
<td>1</td>
<td>48</td>
<td>2%</td>
</tr>
<tr>
<td>Number of BUFG/BUFGCTRLs</td>
<td>1</td>
<td>32</td>
<td>3%</td>
</tr>
</tbody>
</table>
Simple dual address bus architecture for transferring data between peripherals.
### FemtoCntrl: instructions

<table>
<thead>
<tr>
<th>BIT</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st form -&gt;</td>
<td>CODE</td>
<td>DATA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2nd form -&gt;</td>
<td>CODE</td>
<td>TARGET ADDRESS</td>
<td>SOURCE ADDRESS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3rd form -&gt;</td>
<td>CODE</td>
<td>TARGET ADDRESS</td>
<td>DATA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Function

- **NOOP**: No operation
- **AJMB**: Absolute Jump if STATUS_REG’s bit specified by DATA field is true
- **AJMP**: Absolute Jump always. DATA field represents LSB of absolute address
- **RJMB**: Relative Jump if STATUS_REG’s bit specified by DATA field is true
- **RJMP**: Relative Jump always. DATA field represents LSB of absolute address
- **CALL**: Call sub-routine
- **RETN**: Return from subroutine
- **UNUSED**: Free instruction space
- **MOVE**: Move data from source to target
- **UNUSED**: Free instruction space
- **LOAD**: Load data to target
PROM organization for Femto

- **segment 2** (scrub block1)
- **segment 1** (readback block1)
- **segment 0** (original bitstream)
Femto Flow

1. femtoasm.pl
   - Generates a machine code text file (.txt) and an extended prom data text file (.prom.txt)

2. exe2coe.pl
   - Generates a COE file for memory initialization and a MEM file for bitstream initialization and simulation

3. data2mem
   - Incorporates new BRAM data into the bitstream without compilation being necessary

4. gen_prom.pl
   - Generates a text file with data to be included in the PROM's programming file. Uses utilities to extract frames info from bitstream

5. create_cmd.pl
   - Creates BATCH file with commands for impact

6. pc.pl
   - Incorporates data generated in previous step with MCS file

_outputs:
- .asm
- .txt
- prom.txt
- .cmd
- .coe
- .mem
- _2.mem
- bd.bmm
- _new.bit
- .mcs
- _mcs.txt
- _new.mcs
Agenda

- Background and other reported solutions
- Self-scrubbing basics
- Architecture
- Algorithms
- Roadmap
Read-detect-scrub and SEFI detection are two representative tasks the scrubber must perform.

Different SEFI tests proposed by Xilinx. FAR test is the most representative.
START:
#Send command sequence (see [13])
for “j” = 0 to N
  for “i” = 0 to 3
    load byte “i” of word “j” into p_icap reg;
  end for;
end for;

#Load 41 into a register to count 41 words of a frame. Set accumulator register to 0 to start the count.
load 41 into reg1;
load 0 into accu_reg;

#Prepare to readback frame
load ENABLE_READ into p_icap control register

#wait until busy signal is low
READ_STAT:
move status register to reg1;
conditional jump to READ_FR if reg1=BUSY_LOW;
always jump to READ_STAT;

#Read in frame and calculate its CRC
READ_FR:
for “i” = 0 to 41
  move p_icap register into p_crc data reg;
  jump conditional to CRC_CAL if accu_reg=reg1
end for;

#Compare calculated CRC to old CRC. By given
#the address of the locally stored CRC
CRC_CALC:
move crc_address into p_crc address register

#Read in comparison result
move status to reg1;
jump conditional to SCRUB if reg1=CRC_ERROR;
jump always to START;
Agenda

- Background and other reported solutions
- Self-scrubbing basics
- Architecture
- Algorithms
- Roadmap
Roadmap

- Supporting components: Formal verification plan for software components (flow) and RTL (Q2-10)
- TMR implementation
- Static testing: Fault injection / scrubbing automatic test (Q3-10)
- Dynamic testing
  - Synthetic: Fault injection / detect / scrub / error monitoring (Q4-10)
  - Proton testing: radiate / detect / scrub / error monitoring (Q4-10)
  - “Real application” test?
Questions?

alonzo.vera@micro-rdc.com