High-Level Synthesis Techniques for In-Circuit Assertion-Based Verification

John Curreri
Ph.D. Candidate of ECE, University of Florida

Dr. Greg Stitt
Assistant Professor of ECE, University of Florida

Dr. Alan D. George
Professor of ECE, University of Florida

April 19, 2010
High-Level Synthesis

- **Ease of programming**
  - No HDL coding required for application acceleration
  - Abstraction of communication function

- **Provides built-in support**
  - Methods to gain speedup
    - Pipelining of loops
    - High-performance library functions
  - Mitigation of race conditions
    - Signals
    - Semaphores
  - Communication
    - Buffered streaming transfers
    - DMA transfers

- e.g. Impulse-C tool flow
High-Level Synthesis Verification

- **HLS simulation**
  - Executes source on CPU
  - Does not provide accurate verification

- **HDL simulation and analyzers**
  - Provide accurate verification
  - Productivity lowered by the need to understand machine-generated HDL

- **Verification framework needed**
  - Maintains high abstraction level and provides accurate verification
  - Programmers are not required to understand HDL or HDL tools
ANSI-C Assertion Debugging

- Error checking
  - Used to check if variables are in an acceptable range
- Example usage
  - ```c
    int num,i,x[10];
    while(num==0) {
      num=x[i];
      i++
      assert(i<10);
    }
  ```
- Failure actions
  - Failure information is printed to stderr
    - ```c
      assert_test.c:7: main: Assertion `i==0' failed.
    ```
    - W = file name; X = line number; Y = function name; Z= expression
  - Program terminated using abort()
- Assertion checking switch
  - `#define NDEBUG`
  - Disables assertion checking
Related Research

- Assertion languages and libraries
  - VHDL assertion statements
  - SystemVerilog Assertions (SVA)
  - Open Verification Library (OVL)
  - Property Specification Language (PSL)

- Commercial assertion tools
  - Temento’s DiaLite

- Academic debugging tools
  - Camera’s debugging environment
  - Sea Cucumber synthesizing compiler

- Logic analyzers
  - Xilinx’s ChipScope
  - Altera’s SignalTap
Verification Framework Overview

- **Assertion-based verification usage**
  - Document and check for conditions that should never occur during execution

- **In-circuit verification process**
  - Open application files in GUI
  - Single-click instrumentation
    - Converts assertions to if statements
    - Generates communication channels
    - Creates software function to display errors and program abort if failure detected
  - Use standard tool flow to compile/execute
  - Assertion failure output during execution

- **Seamlessly transfer assertions from simulation to runtime**

**HLS Assert**

**Assertion Code**

```c
assert (address >= 0);
```

**Assertion Output**

```
memtest_hw.c:14: Assertion 'address >= 0' failed.
```
Standard Assertion

- **Assertion conversion**
  - FPGA side
  - *Assert* statement changed to *if* statement

- **False evaluation**
  - FPGA side
  - Sends a message with a unique identifier

- **Assertion notification**
  - CPU side
  - Function to receive, decode, and display failed assertions
  - ANSI-C output format

```c
Source Code (hardware)
assert(a[0] != 1); // line 17

Conversion (hardware)
if(a[0] != 1){
int identifier = 17;
cc_stream_write(stream_name, &identifier, sizeof(int32));
}

Conversion (software)
cc_stream_read(stream_name, &identifier, sizeof(int32));
switch(identifier) {
  case 17
    fprintf(stderr,'memtest_hw c 17:"Assertion a[0] != 1 failed \
'failed"

```


### Assertion Optimizations

- **Parallelization**
  - Assertion checking can slow down the application
    - Move assertion checkers to a separate parallel process
  - Communication can slow down pipelined assertions
    - Move communication calls to a third process

<table>
<thead>
<tr>
<th>Clock cycles</th>
<th>Standard</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>App. line 1</td>
<td>App. line 1</td>
</tr>
<tr>
<td>2</td>
<td>Check assertion</td>
<td>App. line 2</td>
</tr>
<tr>
<td>3</td>
<td>Failure communication</td>
<td>App. line 3</td>
</tr>
<tr>
<td>4</td>
<td>App. line 2</td>
<td>App. line 4</td>
</tr>
</tbody>
</table>
State-machine Comparison

Original

Standard assertion

Optimized assertion

assert((j <= 0 || a[0] == i) && (b[0] == 2 || i > 0));

Array data retrieval requires an extra state

Boolean operators require many additional states
Assertion Optimizations

- Resource replication
  - Application and assertion are competing for data access
    - Replicate data structure (e.g., duplicated block RAM that is dedicated for assertion read access)

<table>
<thead>
<tr>
<th>Clock cycles</th>
<th>Standard</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>App. read a[0]</td>
<td>App. read a[0]</td>
</tr>
<tr>
<td>3</td>
<td>App. read a[1] Communication</td>
<td>Application Communication</td>
</tr>
<tr>
<td>4</td>
<td>Application</td>
<td>Application</td>
</tr>
</tbody>
</table>
Assertion Optimizations

- Resource sharing
  - Minimize FPGA resources usage of assertions
  - Reuse assertion checking and communication resources amongst all assertion calls

<table>
<thead>
<tr>
<th>P</th>
<th>Application process</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Assertion checker</td>
</tr>
<tr>
<td>C</td>
<td>Failure communication</td>
</tr>
</tbody>
</table>

![Diagram](image)

Impulse C wrapper

Standard

- P
- A
- C

11
- 32
- 32
- 32
- 32
- 32

Optimized

- P
- A

1
- 1
- 1
- 1
- 4

Impulse C wrapper

- 32
- 32
- 32
- 32
In-Circuit Verification Case Study

- Assertion in Line 6 shows Impulse-C translation mistake
  - **Simulation**
    - 64-bit comparison of 4294967286 > 4294967296 evaluates to false
  - **Execution on target platform**
    - 5-bit comparison of 22 > 0 evaluates to true

- Assertion in Line 8 shows user translation mistake
  - Impulse-C simulation requires C code for HDL function
  - Behaviors of C code and HDL may not be the same
  - Assertions can be used to check that behaviors match

```c
1   co_uint64 c2, c1;
2   co_int32 address, array[20], out;
3   c2 = 4294967286; c1 = 4294967296;
4   if (c2 > c1) address = c2 - c1;
5   else address = 0;
6   assert(address >= 0);
7   out = user_hdl(address);
8   assert((30 > out) && (out > 20));
9   array[address] = out;
```
Debugging Case Study

- `assert(0);`
  - Used to “trace” execution
  - To find when an application fails to complete (hangs)
  - Positive indicator rather than negative indicator

- **NABORT**
  - Stops application from aborting

- **Output comparison**
  - Line numbers of the failed assertions
    - Software simulation vs. platform execution
    - Hang occurred at a memory read at end of loop

- **Solution**
  - Memory read replaced with memory write
  - Correction allowed the process to complete execution

```c
263  for ( i = 1; i < status; i++ ) {
264     IF_SIM(printf("HW:DE:%i\n",i));
265     assert(0);
266     ...  
392     assert(0);
393     co_memory_readblock(...);
394     assert(0);
395 }  
396     assert(0);
297     co_signal_post(done, status);
```
Application Case Studies

- **Triple-DES**
  - Optimized assertions
    - No latency overhead
    - FPGA overhead to the right
  - Standard assertions
    - More ALUT (+0.125%)
    - Higher freq. (144.74MHz)

- **Edge detection**
  - Optimized assertions
    - No performance overhead
    - FPGA overhead to the right
  - Standard assertions
    - Less ALUTs (+0.03%)

### EP2S180

<table>
<thead>
<tr>
<th>Original</th>
<th>Assert</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Used (out of 143520)</td>
<td>13677 (9.53%)</td>
<td>13851 (9.65%)</td>
</tr>
<tr>
<td>Comb. ALUT (out of 143520)</td>
<td>7929 (5.52%)</td>
<td>8025 (5.59%)</td>
</tr>
<tr>
<td>Registers (out of 143520)</td>
<td>10019 (6.98%)</td>
<td>10055 (7.01%)</td>
</tr>
<tr>
<td>Block RAM (9383040 bits)</td>
<td>222912 (2.37%)</td>
<td>223488 (2.38%)</td>
</tr>
<tr>
<td>Frequency (MHz)</td>
<td>145.71</td>
<td>141.98</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Original</th>
<th>Assert</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Used (out of 143520)</td>
<td>12250 (8.54%)</td>
<td>12273 (8.56%)</td>
</tr>
<tr>
<td>Comb. ALUT (out of 143520)</td>
<td>6726 (4.69%)</td>
<td>6809 (4.75%)</td>
</tr>
<tr>
<td>Registers (out of 143520)</td>
<td>9371 (6.53%)</td>
<td>9417 (6.56%)</td>
</tr>
<tr>
<td>Block RAM (9383040 bits)</td>
<td>141120 (1.50%)</td>
<td>141696 (1.51%)</td>
</tr>
<tr>
<td>Frequency (MHz)</td>
<td>77.52</td>
<td>79.31</td>
</tr>
</tbody>
</table>

Impulse-C design, XD1000, Stratix-II (EP2S180)
Scalability Case Study

- **Resource overhead**
  - Optimized shown to right
  - 128 processes
    - 4.07% ALUTs standard
    - 1.34% of ALUTs optimized
      - Over a 3x improvement

- **Frequency overhead**
  - Shown in graph to right
  - 128 processes
    - 154MHz standard
      - 18.8% overhead
    - 189MHz optimized
      - 18.5% improvement

**Impulse-C design, XD1000, Stratix-II (EP2S180)**
Performance Overhead Case Study

- Single-comparison assertion
  - Lower bound on optimization improvements
- Scalar variable
  - Optimized overhead reduced to zero
- Array
  - Optimized overhead
    - Rate reduced to zero
    - Latency reduced

<table>
<thead>
<tr>
<th>Assertion data structure</th>
<th>Latency</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unoptimized</td>
<td>Optimized</td>
<td></td>
</tr>
<tr>
<td>Scalar variable</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Array (non-consecutive)</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Array (consecutive)</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

**TABLE 3**
Single-Comparison Assertion

<table>
<thead>
<tr>
<th>Overhead</th>
<th>Unoptimized</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency</td>
<td>Rate</td>
<td>Latency</td>
</tr>
<tr>
<td>Scalar variable</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Array</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

**TABLE 4**
Pipeline Single-Comparison Assertion

Impulse-C design, XD1000, Stratix-II (EP2S180)
Conclusions

- Created first framework/tool (to our knowledge) for HLS in-circuit assertion-based verification
  - Familiar and easy to use ANSI-C assertions
  - Automated conversion for Impulse C
- Application case studies performed
  - Low area and frequency overhead
  - Highly scalable
  - Minimal to no change of application’s state machine
- Future work
  - Fully automate generation of optimized assertions
  - Add capability to check timing via assertions
Questions
References


References


