Detecting Non-Functional Circuit Activity in SoC Designs

Asia & South Pacific Design Automation Conference 2018
Agenda

1. Motivation: What is the benefit of determining non-functional activity?

2. Our Methodology
   - Design Analysis
   - Activity Simulation

3. Evaluation Results

4. Summary & Future Work
What is the benefit of determining non-functional activity?

1 MOTIVATION
The benefit of determining non-functional activity…

- When designing SoCs, a bunch of optimizations at RTL for reducing the toggle activity in a design, like Clock Gating, Operand Isolation, …

- **Basic idea of each method**: Identify under which conditions several signals or signal groups are not needed to assure correct circuit function.

As long as $S = T$, all activity at $A$ is cut off by the multiplexer!

If division result is not needed, divider activity is redundant!

We refer to toggle activity in a design, that is **not needed for a correct function**, as being non-functional!
The benefit of determining non-functional activity...

- Commercial tools like Synopsys PrimeTime or ANSYS PowerArtist provide activity metrics such as the toggle activity to identify design issues.
  - But: Non-functional (redundant) activity needs to be identified manually!

ANSYS PowerArtist: The yellow boxes and arrows are designer knowledge and are obtained manually!

Today: Is there a chance to obtain non-functional activity automatically?
2 Our Methodology
Methodology

(A) Design Analysis. Needs to be done only once per design!

RTL Design → Elaboration → Elaborated RTL Netlist → Design Analysis → Simulation Graph

Done by Synopys Design Compiler.
GTECH-based RTL netlist preserving the original design structure including components, ports and registers.

Offline Design Analysis leads to a graph that is used for activity simulation.

(B) Activity Simulation. Needs to be done per testbench!

VCD Files → Activity Simulator → Activity Traces
• **Starting Point:** Synthesizable RTL Design
  - Elaboration with Synopsys Design Compiler.
  - Generation of a Verilog RTL Netlist.
    - Maps statements like if, case, … to generic boolean logic.
    - Retains RTL design structure (components, registers, ports).
    - Used for a formal design analysis.

```
module my_module(CLK, A, S, X);
  input CLK, A, S;
  output X;
  reg R;

  assign X = S ? R : A;
  always @(posedge CLK) R <= A;
endmodule;
```

```
module my_module(CLK, A, S, X);
  input CLK, A, S;
  output X;
  wire N0;
  GTECH_MUX2(.A(A), .B(NO),
              .S(S) .Z(X));
  SEQGEN R_reg(.clocked(CLK),
              .next_state(A), .Q(N0))
endmodule;
```
Design Analysis: Formalizing Design

- Deriving a graph representation of the elaborated RTL netlist!

```
module my_module(CLK, A, S, X);
  input CLK, A, S;
  output X; wire N0;
  GTECH_MUX2(.A(A), .B(NO), .S(S), .Z(X));
  SEQGEN R_reg(.clocked(CLK), .next_state(A), .Q(N0))
endmodule;
```

Nodes represent circuit elements like component ports or registers

Edges represent functional dependencies between nodes

<table>
<thead>
<tr>
<th>NODE</th>
<th>COMPONENT</th>
<th>TYPE</th>
<th>DIRECTION</th>
<th>DATA</th>
<th>TRIGGER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>/top</td>
<td>net</td>
<td>input</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CLK</td>
<td>/top</td>
<td>net</td>
<td>input</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>S</td>
<td>/top</td>
<td>net</td>
<td>input</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>R</td>
<td>/top</td>
<td>flip-flop</td>
<td>-</td>
<td>A</td>
<td>CLK</td>
</tr>
<tr>
<td>X</td>
<td>/top</td>
<td>net</td>
<td>output</td>
<td>A \land \neg S \lor R \land S</td>
<td>-</td>
</tr>
</tbody>
</table>

Simulation Graph

EDGE | SOURCE | SINK
--- | ------ | ------
{A, R} | A | R
{CLK, R} | CLK | R
{A, X} | A | X
{R, X} | R | X
{S, X} | S | X
Boolean Expression Diagrams (BEDs)

- BEDs\(^1\) are a generalization of Binary Decision Diagrams (BDDs).
- Nodes can be either shannon nodes (similar to nodes in BDDs, each with low and high outgoing high edge) or terminal nodes, but also operation nodes!
- Efficient methods for converting BEDs into full or partial BDDs are available!

For each node \( X \) and each incoming edge \( V \rightarrow X \), a \textit{don’t care function} will be derived \( D(F_X, V) = F_{X,V=\top} \bigotimes F_{X,V=\bot} \): Under which condition is \( V \rightarrow X \) inactive?

- Control signals have higher priority!
- Rounding eliminates conflicts between don’t care conditions (e.g. if \( A \) is inactive because \( S \) is true, \( S \) cannot be inactive because \( A==B \), at the same time!)

\[
D(X,A) \quad \text{Don't care (A)} \quad \text{up (S) round (S)} \quad \text{Rounded } D(X,A) \\
D(X,S) \quad \text{Don't care (S)} \quad \text{up (\emptyset) round (\emptyset)} \quad \text{Rounded } D(X,S)
\]
Design Analysis: Don’t Care Analysis

- Finally the simulation graph gets a boolean function INACTIVE for each edge that determines under which condition a specific edge is not used.
  - Something similar is done for a register: KEEPS_VALUE

<table>
<thead>
<tr>
<th>NODE</th>
<th>COMPONENT</th>
<th>TYPE</th>
<th>DIRECTION</th>
<th>DATA</th>
<th>TRIGGER</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>/top</td>
<td>net</td>
<td>input</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CLK</td>
<td>/top</td>
<td>net</td>
<td>input</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>S</td>
<td>/top</td>
<td>net</td>
<td>input</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>R</td>
<td>/top</td>
<td>flip-flop</td>
<td>-</td>
<td>A</td>
<td>CLK</td>
</tr>
<tr>
<td>X</td>
<td>/top</td>
<td>net</td>
<td>output</td>
<td>A ∧ ¬S ∨ R ∧ S</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EDGE</th>
<th>SOURCE</th>
<th>SINK</th>
</tr>
</thead>
<tbody>
<tr>
<td>{A, R}</td>
<td>A</td>
<td>R</td>
</tr>
<tr>
<td>{CLK, R}</td>
<td>CLK</td>
<td>R</td>
</tr>
<tr>
<td>{A, X}</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>{R, X}</td>
<td>R</td>
<td>X</td>
</tr>
<tr>
<td>{S, X}</td>
<td>S</td>
<td>X</td>
</tr>
</tbody>
</table>

**KEEPES_VALUE**

<table>
<thead>
<tr>
<th>Added during don’t care analysis.</th>
</tr>
</thead>
<tbody>
<tr>
<td>INACTIVE</td>
</tr>
<tr>
<td>⊥</td>
</tr>
</tbody>
</table>
• Simulation Graph is now used with a VCD file to determine functional and non-functional activity per cycle.

• Two-Phase Simulation Model:
  a) Forward Simulation
  b) Backward Propagation
• During **forward simulation** all don't care functions are solved by taking the exact values of each signal from the VCD trace.

• For each cycle, we determine:
  - Is a **register written** this cycle or does it retain its current value?
  - Is an **edge actively read** or is it inactive?

---

**Activity Simulation: Forward Simulation**

<table>
<thead>
<tr>
<th>NODE</th>
<th>COMPONENT</th>
<th>TYPE</th>
<th>DIRECTION</th>
<th>DATA</th>
<th>TRIGGER</th>
<th>KEEPS VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>/top</td>
<td>net</td>
<td>input</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CLK</td>
<td>/top</td>
<td>net</td>
<td>input</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>S</td>
<td>/top</td>
<td>net</td>
<td>input</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>R</td>
<td>/top</td>
<td>flip-flop</td>
<td>-</td>
<td>A</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>X</td>
<td>/top</td>
<td>net</td>
<td>output</td>
<td>A &amp; ¬S &amp; V &amp; S</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**VCD File**

<table>
<thead>
<tr>
<th>NODE</th>
<th>0ns</th>
<th>10ns</th>
<th>20ns</th>
<th>30ns</th>
<th>40ns</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLK</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>S</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Forward Simulation**

<table>
<thead>
<tr>
<th>EDGE</th>
<th>SOURCE</th>
<th>SINK</th>
<th>INACTIVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>{A, R}</td>
<td>A</td>
<td>R</td>
<td>-</td>
</tr>
<tr>
<td>{CLK, R}</td>
<td>CLK</td>
<td>R</td>
<td>-</td>
</tr>
<tr>
<td>{A, X}</td>
<td>A</td>
<td>X</td>
<td>S</td>
</tr>
<tr>
<td>{R, X}</td>
<td>R</td>
<td>X</td>
<td>¬S</td>
</tr>
<tr>
<td>{S, X}</td>
<td>S</td>
<td>X</td>
<td>-</td>
</tr>
</tbody>
</table>

---

**Simulation Graph**
Activity Simulation: Backward Propagation

- During **backward propagation** all information is propagated backwards in time and space.
  - **Time Propagation:** Remove register writes without any future read!
  - **Space Propagation:** If a node has only non-functional outgoing edges, mark all incoming edges non-functional!

We end up with a cycle-by-cycle trace, which shows which registers and which edges are functional or non-functional in a particular cycle!
3 Evaluation Results
Implementation

- Methodology implemented as a Scala library and integrated as a plugin into Synopsys Design Compiler.

  ```
  # Include the Activity Simulator Plugin for Design Compiler
  source syn_dc_plugin.tcl
  # Read in Design and elaborate it
  analyze -library ${library} -format verilog {rtl/zet.v rtl/zet_addsub.v ...}
  elaborate zet
  # Add Clock Constraints
  source zet.sdc

  # Generate Simulation Graph
  compile activity simulator zet.simulator

  # Simulate Activity for a bunch of VCD files
  report_activity zet.simulator -output activity -vcd "./sim/01_jmpmov.vcd"
  -vcd "./sim/18_div.vcd" -vcd_path "test_zet/zet" -activity -smt -clock_gating
  ```

- Evaluation done based on a variety of open source + commercial designs:
  - x86-compatible open source processor (http://zet.aluzina.org)
  - RISC-V based Murax SoC (https://github.com/SpinalHDL/VexRiscv)
  - Commercial ASIP architecture
Example Design: x86 Zet

- x86-compatible open source processor design (opencores.org)
- Activity report for Zet running 19_segpr (1 division)

This is the activity trace that we can get from recent tools (actual activity).

This is the trace of functional activity (our tool).

Plugin runtime for this design\(^1\):
- Design Analysis: ~25 seconds
- Simulation: ~350 microseconds per simulated cycle

\(^1\) Differences between both curves indicate non-functional activity.
Example Design: x86 Zet
- x86-compatible open source processor design (opencores.org)
- Sleep Mode Trace for Zet running 18_div (30 divisions)

Green: Component is active!
White: Component is idle!
Reports on RISC-V-based Murax SoC

- RISC-V-based Murax SoC (https://github.com/SpinalHDL/VexRiscv)
  - RISC-V attached to an AXI bus with on-chip RAM
  - 2 timers attached to an APB bus using an AXI ↔ APB bridge

Software running on the RISC-V.

Sleep Mode Traces derived by our tool.

- Plugin runtime for this design\(^1\):
  - Design Analysis: \( \sim 30\) min
  - Simulation: \( \sim 1.5\) milliseconds per simulated cycle

\(^1\) Intel Core i5-3470 3.2GHz, Scientific Linux 7.4, no multi-threading implemented
Commercial ASIP architecture in two different flavours:
- A) Implementation **without** functional unit clock gating
- B) Implementation **with** functional unit clock gating

Evaluation of Clock Gating Efficiency of both flavours using the Dhrystone benchmark:

- **Plugin runtime for this design**: 
  - Design Analysis: ~15 min
  - Simulation: ~3 to 5 milliseconds per simulated cycle

1 Intel Core i5-3470 3.2GHz, Scientific Linux 7.4, no multi-threading implemented
4 SUMMARY
Summary & Future Work

- Developed, implemented and evaluated a methodology for detecting functional and non-functional activity in RTL simulations.

- Evaluated a variety of designs: an open source processor, a commercial ASIP and a RISC-V based SoC

- **Future Work**
  - Speed up simulation speed by either lossless graph compressions or by losing some accuracy, for example merging N 1-bit registers into one simulation graph node.
  - Analysis of the PULPino SoC design.
  - Using Sleep Mode Traces for pattern-based clustering of a design into power domains like in\(^1\).

---

Thank you!

Dustin Peterson

Eberhard Karls Universität Tübingen
Lehrstuhl für Eingebettete Systeme
Fon: +49 7071 - 29 – 75458
Fax: +49 7071 - 29 – 5062
E-Mail: dustin.peterson@uni-tuebingen.de