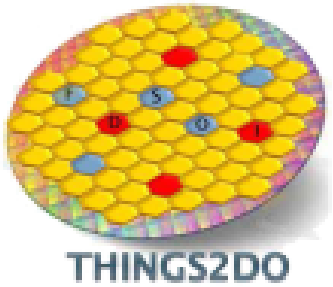
EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

MATHEMATISCH-
NATURWISSENSCHAFTLICHE FAKULTÄT
Eingebettete Systeme



GEFÖRDERT VOM
Bundesministerium
für Bildung
und Forschung

eniac
JOINT UNDERTAKING

Detecting Non-Functional Circuit Activity in SoC Designs

Asia & South Pacific Design Automation Conference 2018

15.02.2018, Dustin Peterson | University of Tuebingen | Germany

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

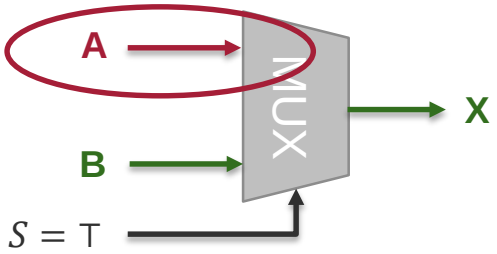
What is the benefit of determining non-functional activity?

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!



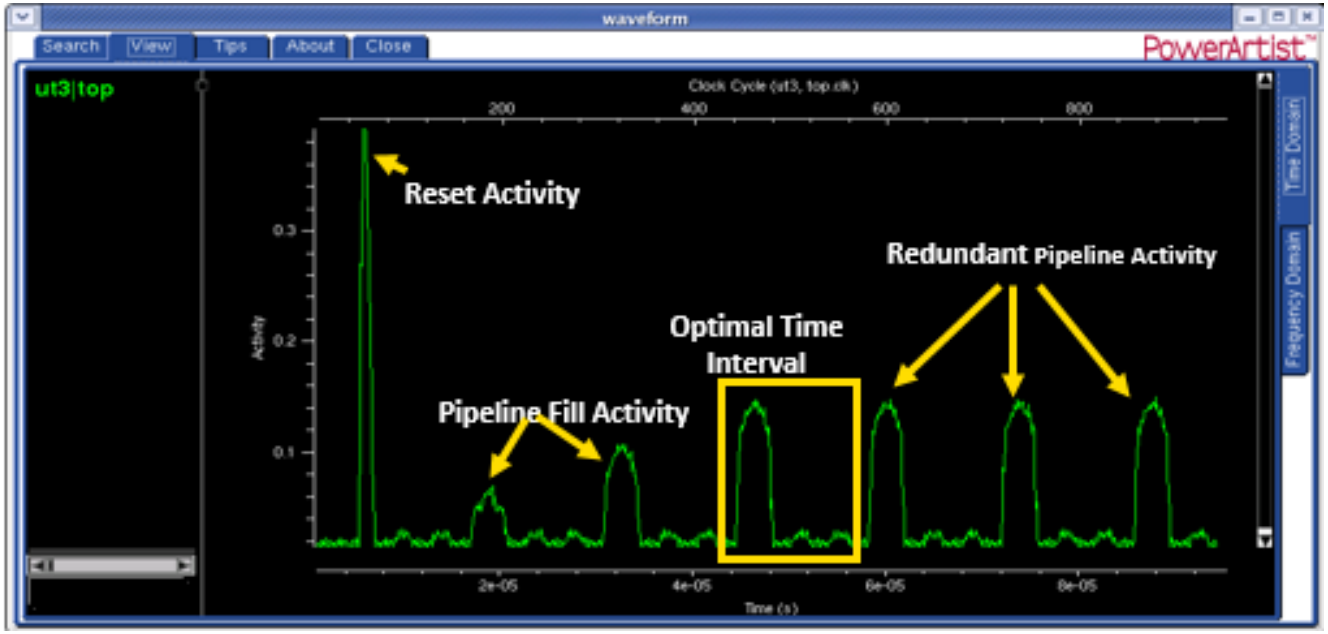
If division result is not needed, divider activity is redundant!

Pipelined Processor

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!



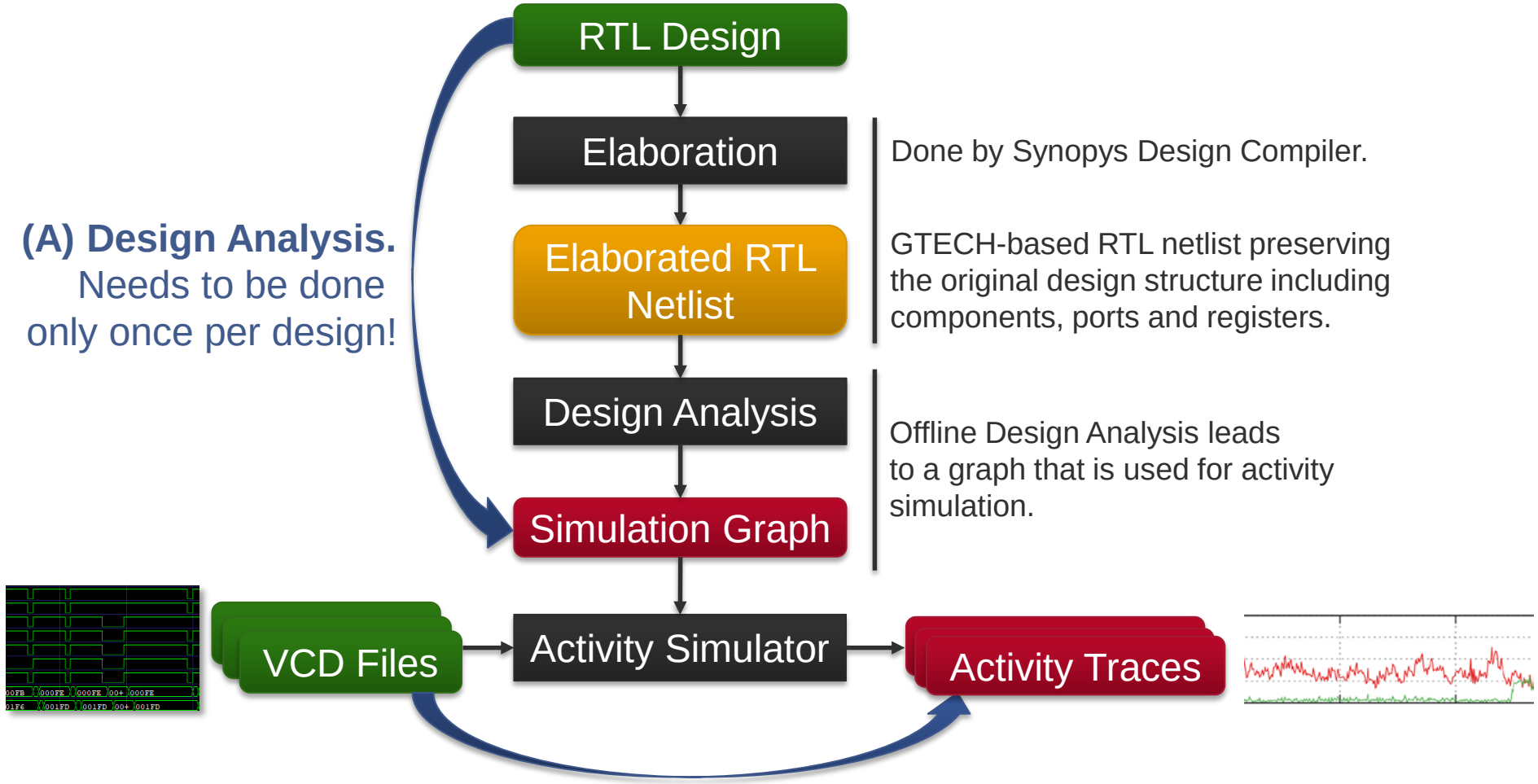
ANSYS PowerArtist: The yellow boxes and arrows are designer knowledge and are obtained manually!

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!



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

Elaboration

- **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;
```

RTL Design

```
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;
```

Elaborated RTL Netlist

Design Analysis: Formalizing Design

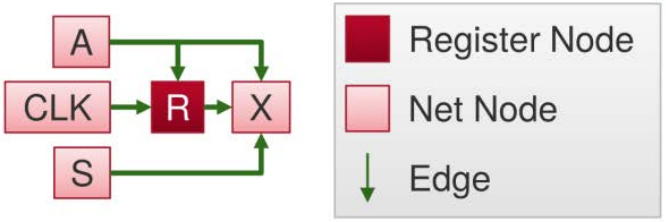
- Deriving a graph representation of the elaborated RTL netlist!

```
module my_module(CLK, A, S, X);
  input CLK, A, S;
  output X; wire N0;

  GTECH_MUX2(.A(A), .B(N0), .S(S), .Z(X));
  ¥**SEQGEN** R_reg(.clocked(CLK), .next_state(A), .Q(N0))

endmodule;
```

Nodes represent circuit elements like component ports or registers

NODE	COMPONENT	TYPE	DIRECTION	DATA	TRIGGER
A	/top	net	input	-	-
CLK	/top	net	input	-	-
S	/top	net	input	-	-
R	/top	flip-flop	-	A	CLK
X	/top	net	output	$A \wedge \neg S \vee R \wedge S$	-

Edges represent functional dependencies between nodes

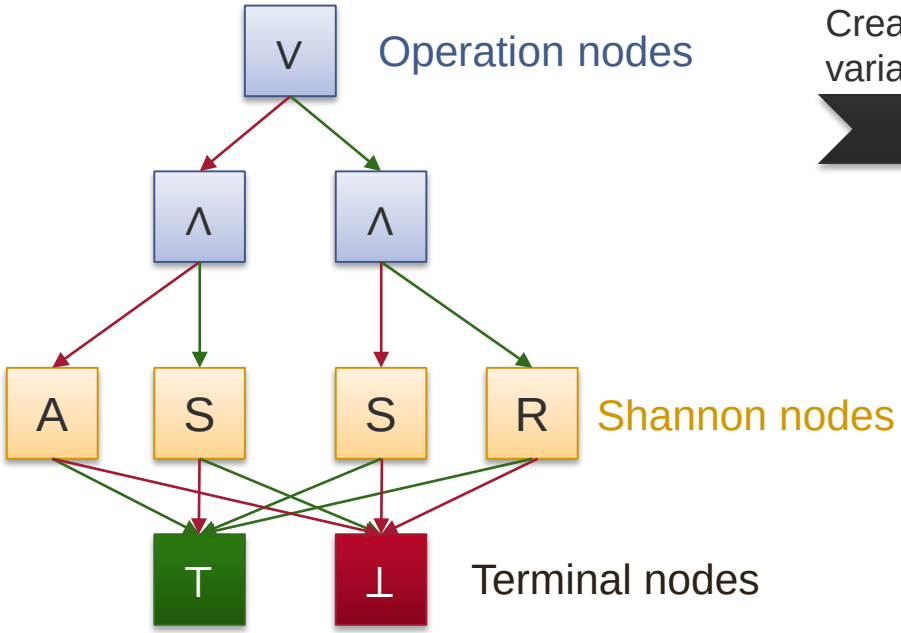


EDGE	SOURCE	SINK
{A, R}	A	R
{CLK, R}	CLK	R
{A, X}	A	X
{R, X}	R	X
{S, X}	S	X

Simulation Graph

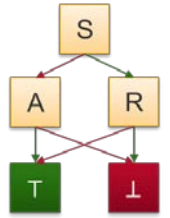
Boolean Expression Diagrams (BEDs)

- 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!



Creating a full BDD with the variable order S > A > R.

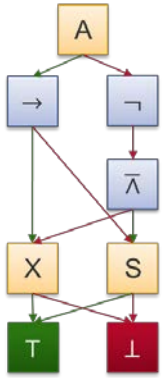
up_all ({S,A,R})



Full BDD with S > A > R.

Creating a partial BDD with only A.

up ({A})

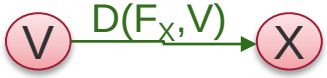


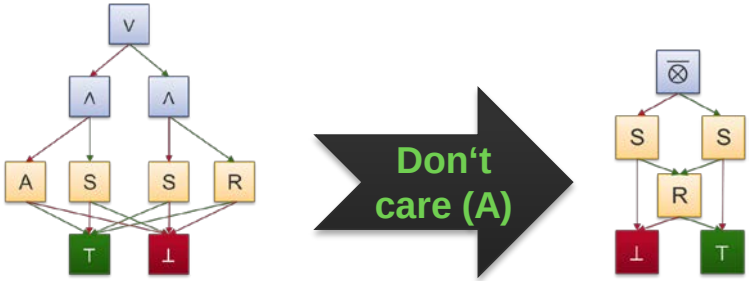
Partial BDD with A at root.

[1]H. Andersen and H. Hulgaard, “Boolean expression diagrams,” Proceedings of Twelfth Annual IEEE Symposium on Logic in Computer Science, 1997.

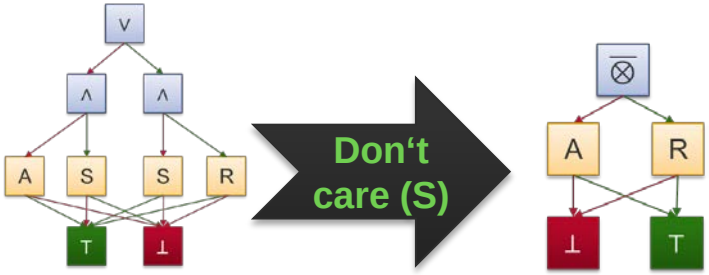
Design Analysis: Don‘t Care Analysis

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





Transfer function F_X of X. D(X,A)

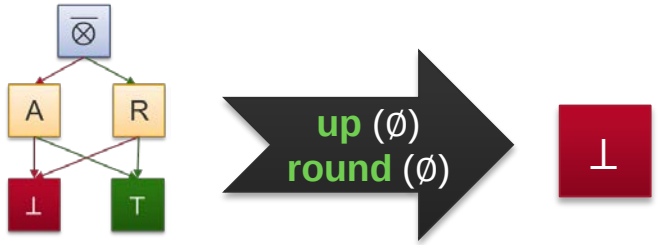


F_X D(X,S)

- The resulting function $D(F, v)$ is then converted to a partial BDD with a heuristic variable order (e.g. S > A > R) and **rounded down** to higher-order variables!
  - 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!)

$D(X, A)$

Rounded $D(X, A)$

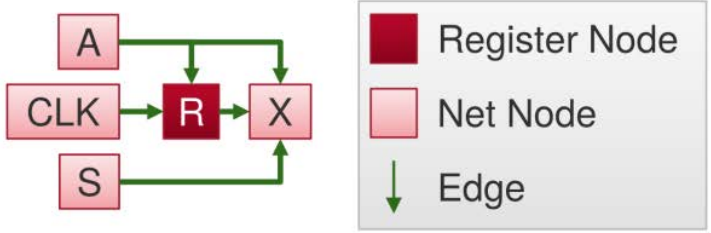


$D(X, S)$ Rounded $D(X, S)$

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

NODE	COMPONENT	TYPE	DIRECTION	DATA	TRIGGER
A	/top	net	input	-	-
CLK	/top	net	input	-	-
S	/top	net	input	-	-
R	/top	flip-flop	-	A	CLK
X	/top	net	output	$A \wedge \neg S \vee R \wedge S$	-



EDGE	SOURCE	SINK
{A, R}	A	R
{CLK, R}	CLK	R
{A, X}	A	X
{R, X}	R	X
{S, X}	S	X

Added during don‘t care analysis.

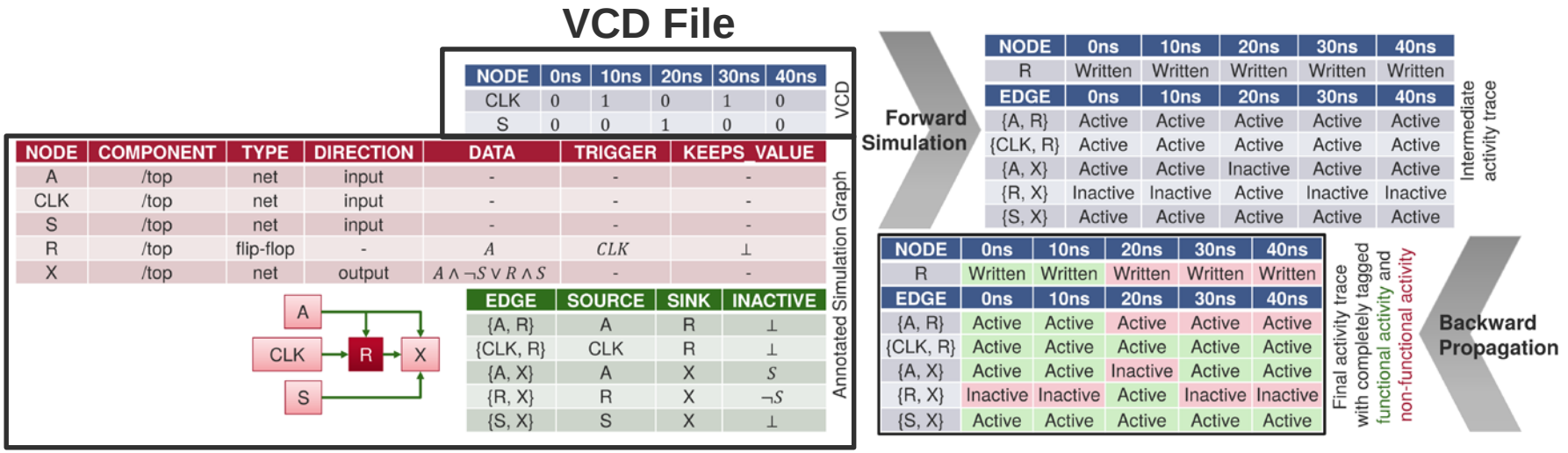
KEEPS_VALUE
-
-
-
$\perp$
-

INACTIVE
$\perp$
$\perp$
S
$\neg S$
$\perp$

Activity Simulation

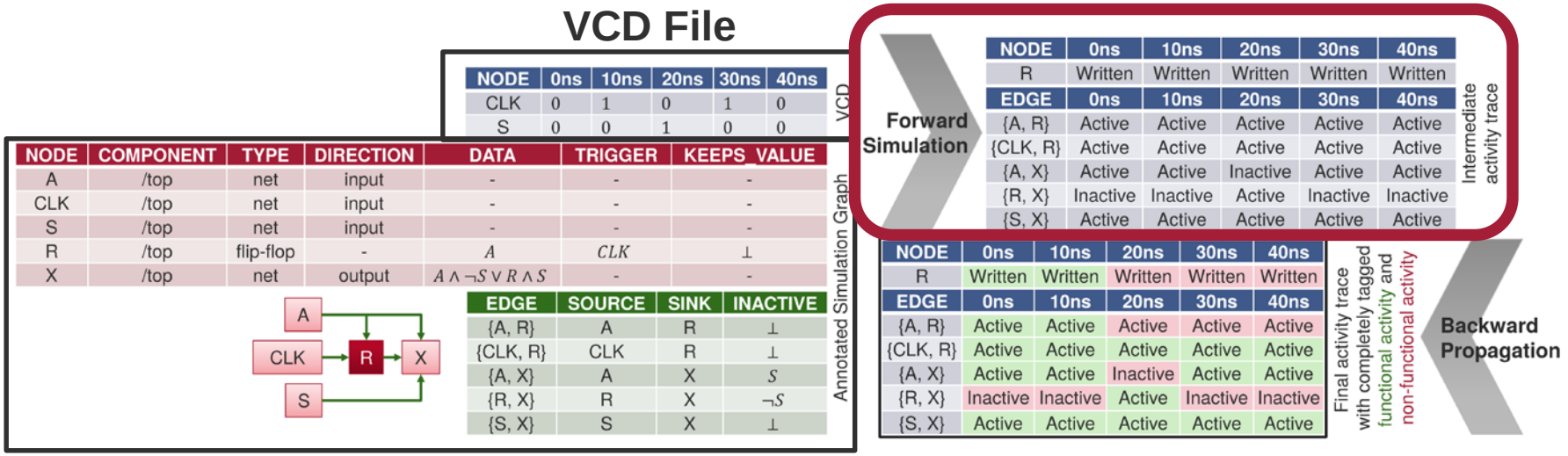
- Simulation Graph is now used with a VCD file to determine functional and non-functional activity per cycle.

- **Two-Phase Simulation Model:**
  a) Forward Simulation
  b) Backward Propagation



Activity Simulation: Forward Simulation

- 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?



NODE	0ns	10ns	20ns	30ns	40ns
CLK	0	1	0	1	0
S	0	0	1	0	0

NODE	COMPONENT	TYPE	DIRECTION	DATA	TRIGGER	KEEPS_VALUE
A	/top	net	input	-	-	-
CLK	/top	net	input	-	-	-
S	/top	net	input	-	-	-
R	/top	flip-flop	-	A	CLK	$\perp$
X	/top	net	output	$A \wedge \neg S \vee R \wedge S$	-	-

EDGE	SOURCE	SINK	INACTIVE
{A, R}	A	R	$\perp$
{CLK, R}	CLK	R	$\perp$
{A, X}	A	X	S
{R, X}	R	X	$\neg S$
{S, X}	S	X	$\perp$

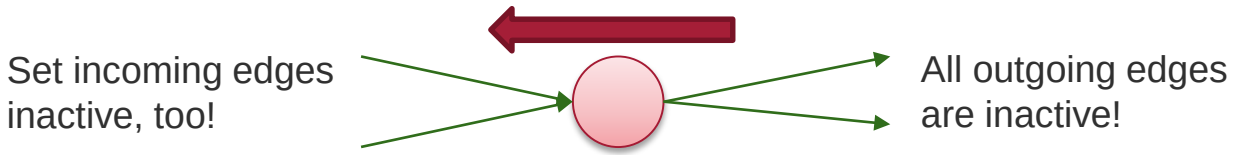
NODE	0ns	10ns	20ns	30ns	40ns
R	Written	Written	Written	Written	Written
EDGE	0ns	10ns	20ns	30ns	40ns
{A, R}	Active	Active	Active	Active	Active
{CLK, R}	Active	Active	Active	Active	Active
{A, X}	Active	Active	Inactive	Active	Active
{R, X}	Inactive	Inactive	Active	Inactive	Inactive
{S, X}	Active	Active	Active	Active	Active

NODE	0ns	10ns	20ns	30ns	40ns
R	Written	Written	Written	Written	Written
EDGE	0ns	10ns	20ns	30ns	40ns
{A, R}	Active	Active	Active	Active	Active
{CLK, R}	Active	Active	Active	Active	Active
{A, X}	Active	Active	Inactive	Active	Active
{R, X}	Inactive	Inactive	Active	Inactive	Inactive
{S, X}	Active	Active	Active	Active	Active

Simulation Graph

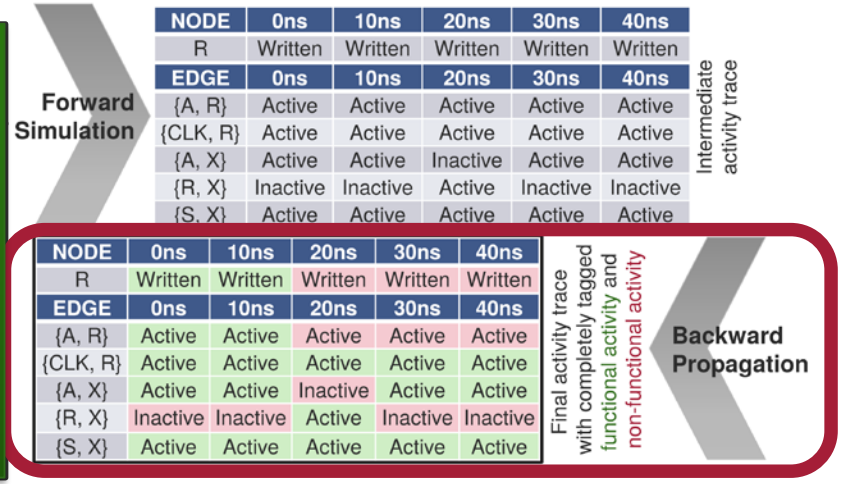
Activity Simulation: Backward Propagation

- 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!



VCD File

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!



NODE	0ns	10ns	20ns	30ns	40ns
R	Written	Written	Written	Written	Written
EDGE	0ns	10ns	20ns	30ns	40ns
{A, R}	Active	Active	Active	Active	Active
{CLK, R}	Active	Active	Active	Active	Active
{A, X}	Active	Active	Inactive	Active	Active
{R, X}	Inactive	Inactive	Active	Inactive	Inactive
{S, X}	Active	Active	Active	Active	Active

NODE	0ns	10ns	20ns	30ns	40ns
R	Written	Written	Written	Written	Written
EDGE	0ns	10ns	20ns	30ns	40ns
{A, R}	Active	Active	Active	Active	Active
{CLK, R}	Active	Active	Active	Active	Active
{A, X}	Active	Active	Inactive	Active	Active
{R, X}	Inactive	Inactive	Active	Inactive	Inactive
{S, X}	Active	Active	Active	Active	Active

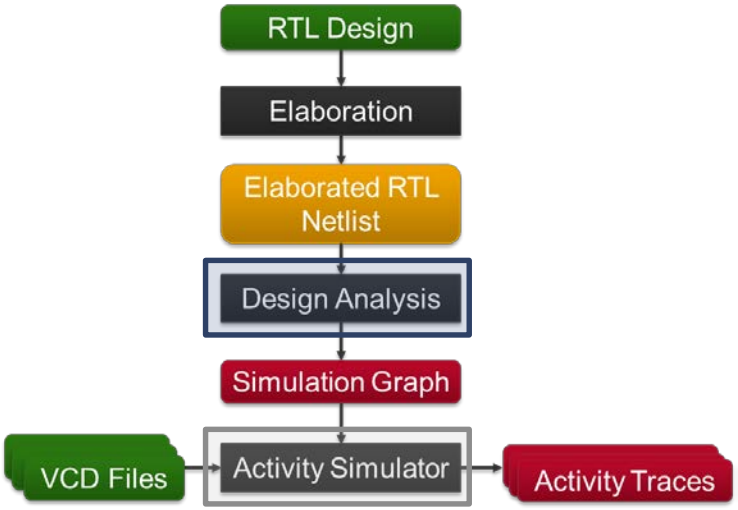
Simulation Graph

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
```

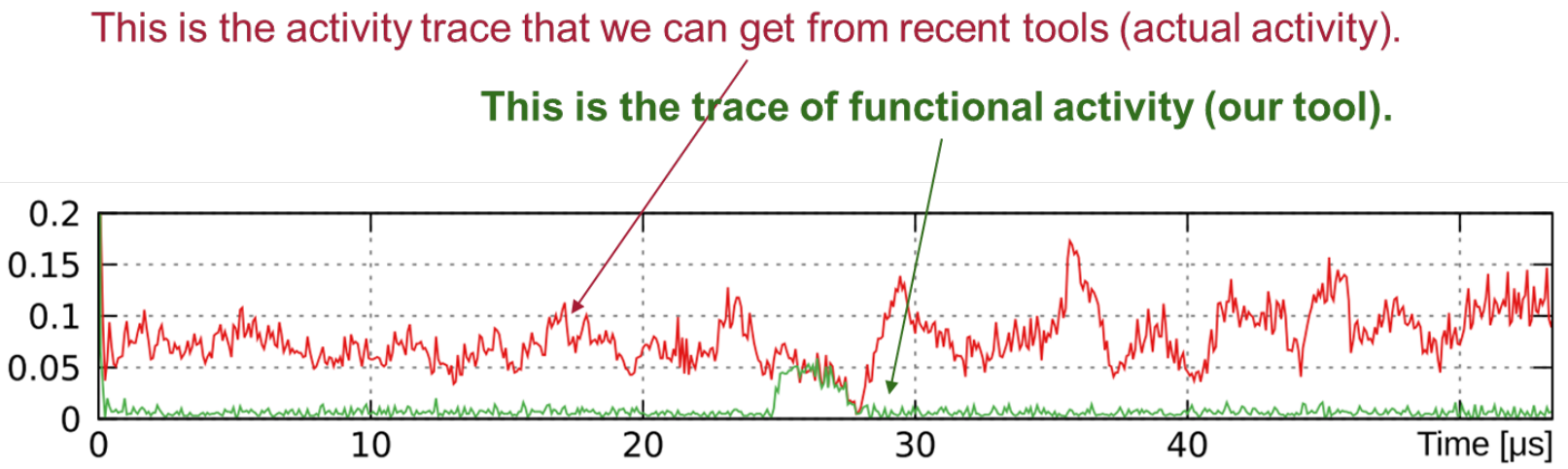


- **Evaluation done based on a variety of open source + commercial designs:**
  - x86-compatible open source processor (http://zet.aluzina.org)
  - RISC-V based Murax SoC (https://github.com/SpinalHDL/VexRiscv)
  - Commercial ASIP architecture

Reports on x86-compatible processor Zet v1.3.1

Example Design: x86 Zet

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



Difference between both curves is the non-functional activity!

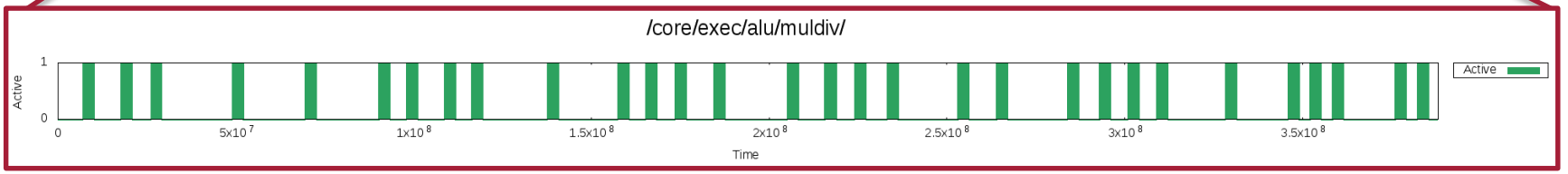
- **Plugin runtime for this design[1]:**
  - Design Analysis: ~25 seconds
  - Simulation: ~350 microseconds per simulated cycle

Reports on x86-compatible processor Zet v1.3.1

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!

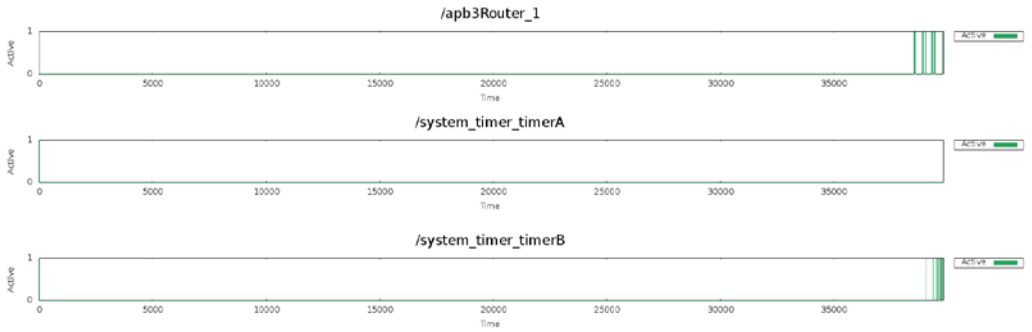


Reports on RISC-V-based Murax SoC

- **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

```c
#include <stdint.h>
#include <murax.h>

void main() {
  for(int i=0; i < 100; i++);

  prescaler_init(TIMER_PRESCALER);
  TIMER_PRESCALER->LIMIT = 1;
  timer_init(TIMER_B);
  TIMER_B->LIMIT = 8;
  TIMER_B->CLEARS_TICKS = 0x00010002;
  while(1);
}
```

Software running on the RISC-V.



Sleep Mode Traces derived by our tool.

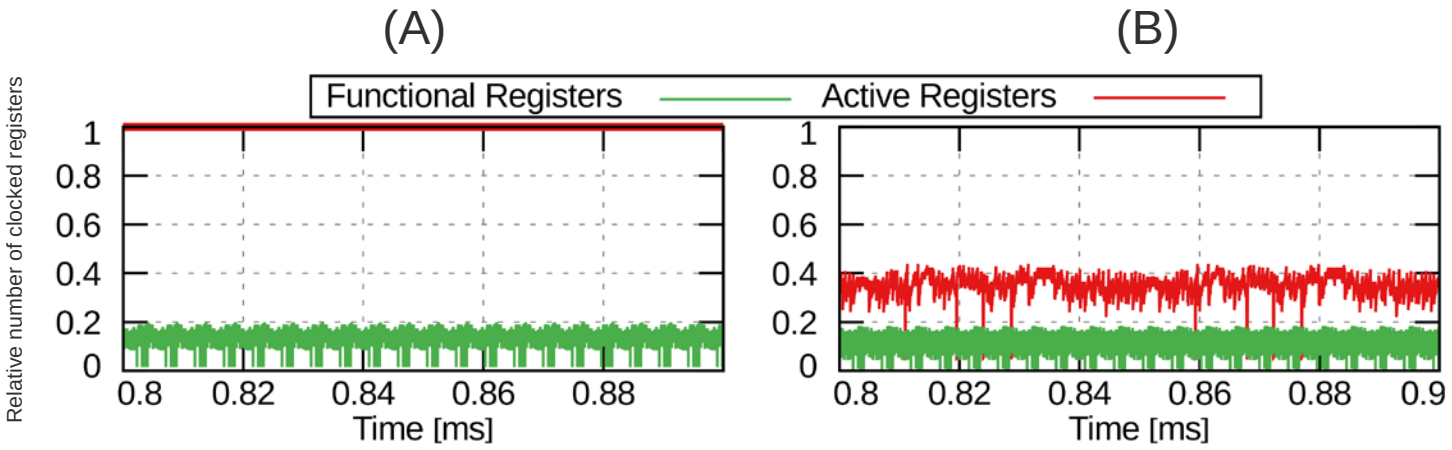
- **Plugin runtime for this design[1]:**
  - Design Analysis: ~30 min
  - Simulation: ~1.5 milliseconds per simulated cycle

[1] Intel Core i5-3470 3.2GHz, Scientific Linux 7.4, no multi-threading implemented

Reports on commercial ASIP

- **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:**



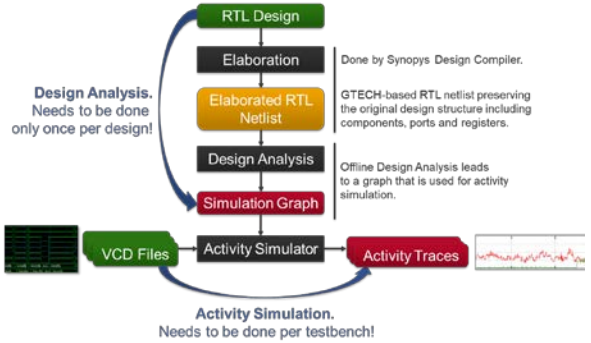
- **Plugin runtime for this design[1]:**
  - Design Analysis: ~15 min
  - Simulation: ~3 to 5 milliseconds per simulated cycle

[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].

[1] A. Dobriyal et al., “Workload Driven Power Domain Partitioning,” in Progress in VLSI Design and Test, 2012.

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