

# Verilog Synthesis and Formal Verification with Yosys

Clifford Wolf  
Easterhegg 2016

# Overview

A) Quick introduction to HDLs, digital design flows, ...

B) Verilog HDL Synthesis with Yosys

1. OSS iCE40 FPGA Synthesis flow
2. Xilinx Verilog-to-Netlist Synthesis with Yosys
3. OSS Silego GreenPAK4 Synthesis flow
4. Synthesis to simple Verilog or BLIF files
5. ASIC Synthesis and custom flows

C) Formal Verification Flows with Yosys

1. Property checking with build-in SAT solver
2. Property checking with ABC using miter circuits
3. Property checking with yosys-smtbmc and SMT solvers
4. Formal and/or structural equivalence checking

# Quick Introduction

- What is Verilog? What are HDLs?
- What are HDL synthesis flows?
- What are verification, simulation, and formal verification?
- What FOSS tools exist for working with Verilog designs?
- How to use Yosys? Where is the documentation?

# What is Verilog? What are HDLs?

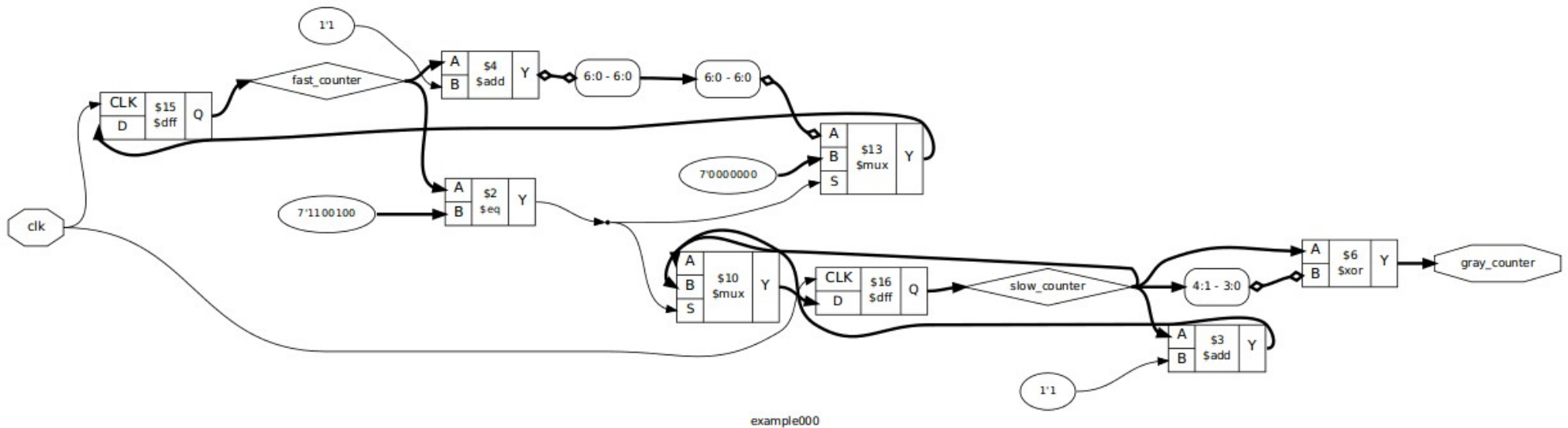
- Hardware Description Languages (**HDLs**) are **computer languages** that describe **digital circuits**.
- The two most important HDLs are **VHDL** and **Verilog / SystemVerilog**.  
(SystemVerilog is Verilog with a lot of additional features added to the language.)
- Originally HDLs were only used for **testing and documentation**. But nowadays HDLs are also used as design entry (instead of e.g. drawing schematics).
- Converting HDL code to a circuit is called **HDL Synthesis**.

# Simple Verilog Example

```
module example000 (  
    input clk,  
    output [4:0] gray_counter  
);  
    localparam PRESCALER = 100;  
  
    reg [$clog2(PRESCALER)-1:0] fast_counter = 0;  
    reg [4:0] slow_counter = 0;  
  
    always @(posedge clk) begin  
        if (fast_counter == PRESCALER) begin  
            fast_counter <= 0;  
            slow_counter <= slow_counter + 1;  
        end else begin  
            fast_counter <= fast_counter + 1;  
        end  
    end  
  
    assign gray_counter = slow_counter ^ (slow_counter >> 1);  
endmodule
```

# Simple Verilog Example

.. this corresponds to the following circuit:



(Diagram generated by Yosys using "prep; show -stretch".)

# What are HDL synthesis flows?

- A few tools must be combined to convert HDL code for example into an ASIC mask layout.
  - **Synthesis**: Conversion of HDL design to netlist
  - **Placer**: Physical placement of cells from netlist, minimizing total wire length and routing congestions
  - **Router**: Create physical wiring for placed design
  - **Timer**: Analyze timing of placed and routed design
  - And finally a tool for **creating production files**
- Such a collection of tools is called an **HDL synthesis flow**.

# What are verification, simulation, and formal verification?

- **Verification** is the process of checking if a design is correct.
- Usually this is done by **simulating** the design.
  - A **testbench** is written that controls the input wires going into the design, and checks the outputs coming out of the design.
  - Usually testbenches are **written in the same HDL** as the design. HDLs usually have a **non-synthesizable** portion of the language just for writing testbenches.
  - However, **simulation can only check a small fraction** of all possible input patterns. Remember the Pentium FDIV bug?
- **Formal verification** uses modern techniques (SAT/SMT solvers, BDDs, etc.) to **prove correctness** by essentially doing an exhaustive search through the **entire possible input space**.



# What (production-ready) FOSS tools exist for working with Verilog designs?

- Icarus Verilog
  - **Simulator for Verilog** and some SystemVerilog
  - Uses interpreter model. **Compiles quickly** but executes slowly.
- Verilator
  - **Simulator for Verilog** and some SystemVerilog
  - Converts HDL design to C++ code (supports different APIs). **Uses C/C++ compiler** to create executable.
  - Compiles slowly but **executes quickly**.
  - **Testbenches** must be written **in C/C++**.
- GTKWave
  - (Verilog) Simulators can write **simulation waveforms** to **VCD files**.
  - GTKWave is a **viewer for VCD files** (and a few other waveform file formats).
- Yosys
  - **Synthesis and some formal verification for Verilog** designs
  - Modular architecture makes Yosys **easy to re-target**

# How to use Yosys?

- Yosys is controlled by **scripts** that call **Yosys commands** in sequence.

Yosys commands can

- **Read designs** in different input formats (“front-end” commands)
  - **Write designs** in different output formats (“back-end” commands)
  - **Perform operations** on the design in memory (“passes”)
  - **Some commands just call** a sequence of **other commands** (for example “synth”)
- A simple example script:

```
read_verilog mydesign.v
read_verilog mycomponent_1.v
read_verilog mycomponent_2.v
synth -top mydesign
dfflibmap -liberty mylibrary.lib
abc -liberty mylibrary.lib
write_edif synthesis_out.edif
```

- Manual, command help messages, appnotes, publications, useful links at Yosys website:  
<http://www.clifford.at/yosys/documentation.html>

# Output of “help synth” (excerpt)

The following commands are executed by this synthesis command:

begin:

hierarchy -check [-top <top>]

...

coarse:

proc

opt\_const

opt\_clean

check

opt

wreduce

alumacc

share

opt

fsm

opt -fast

memory -nomap

opt\_clean

fine:

opt -fast -full

memory\_map

opt -full

techmap

opt -fast

abc -fast

opt -fast

check:

hierarchy -check

stat

check

...

# Yosys Synthesis Flows

ICE40 FPGAs

Xilinx 7-Series FPGAs

Silego GreenPAK4 FPGAs

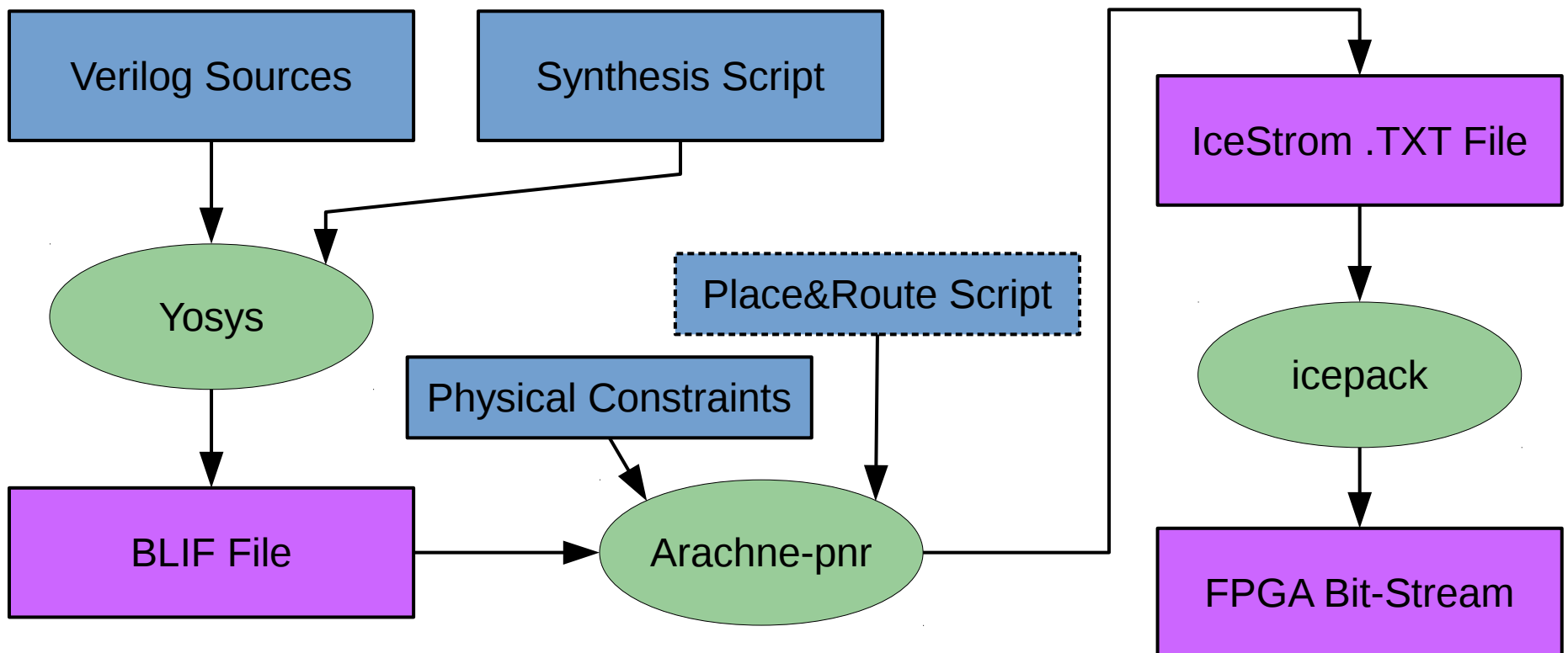
Verilog to BLIF (or simple Verilog)

ASIC Synthesis and custom flows

# Project IceStorm and the IceStorm Flow

Project IceStorm is a **reverse engineering** effort to **document the bit-stream format** of **Lattice iCE40 FPGAs**. Currently iCE40 LP/HX 1K/4K/8K are completely documented. (<http://www.clifford.at/icestorm/>)

The IceStorm flow consists of **Yosys** (synthesis), **Arachne-pnr** (place & route), and some low-level tools directly from Project IceStorm (such as icepack, icetime, or iceprog). This is – as far as I know – the **first completely open source Verilog-to-bitstream FPGA flow**. (Details: See my 32C3 and FOSDEM 2016 presentations.)



# Yosys Synthesis for iCE40

- Simply read Verilog design and call `synth_ice40` to synthesize and write BLIF netlist. For example:

```
read_verilog mydesign.v
read_verilog mycomponent_1.v
read_verilog mycomponent_2.v
synth_ice40 -top mydesign -blif synthesis_out.blif
```

- Cell library is compatible with Lattice [iCE40 Technology Library](#)

```
SB_IO, SB_GB_IO, SB_GB, SB_LUT4, SB_DFF*,
SB_RAM40_4K*, SB_PLL40*, SB_WARMBOOT
```

- Using [BLIF](#) as easy-to-parse [netlist format](#) between Yosys and Arachne-pnr

- Some non-standard extensions for parameters and attributes
- Simple example:

```
.model top
.inputs a b c d
.outputs y
.gate SB_LUT4 I0=b I1=c I2=d I3=a O=y
.param LUT_INIT 0000011111111000
.end
```

# Complete flow for iCE40 with Yosys / Arachne-pnr / IceStorm

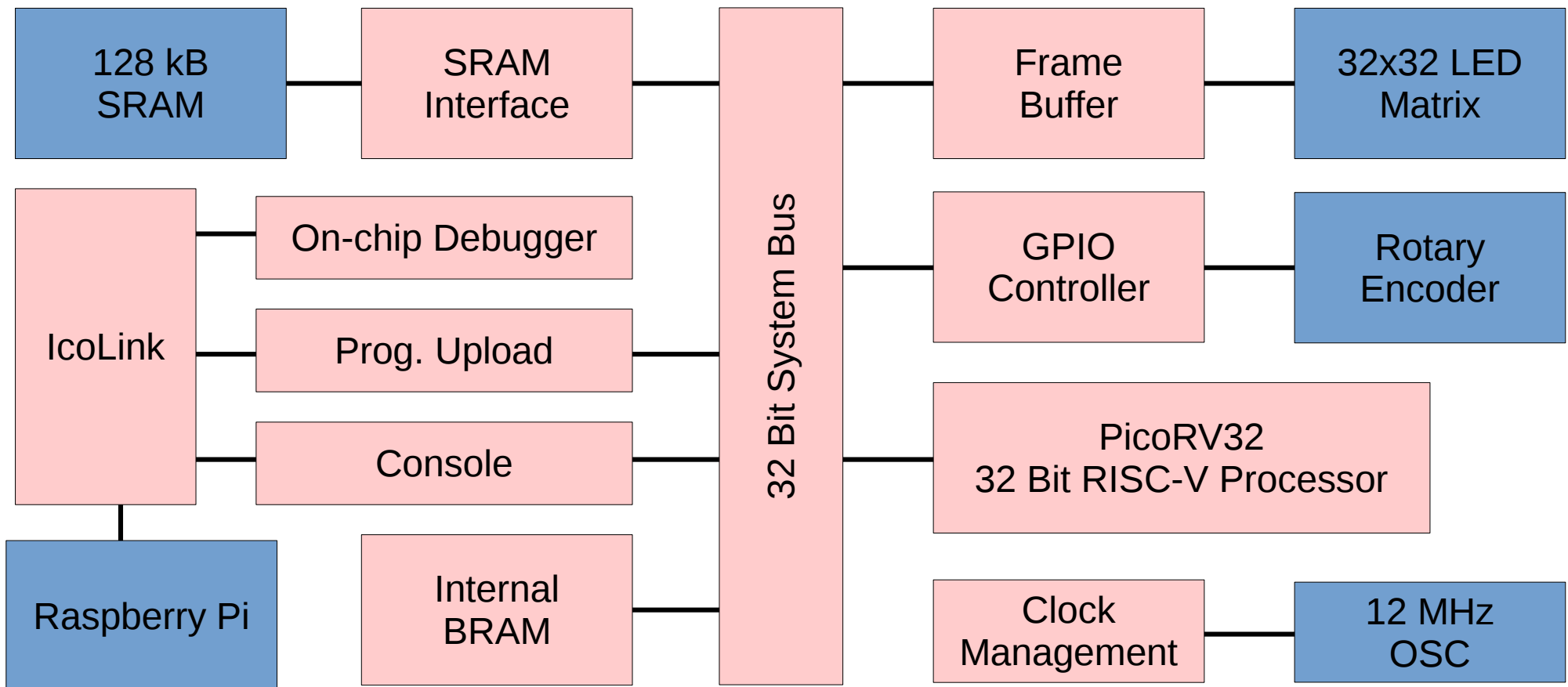
```
# Read Verilog design files
# Write BLIF netlist (mydesign.blif)
yosys -p "
    read_verilog mydesign.v
    read_verilog mycomponent_1.v
    read_verilog mycomponent_2.v
    synth_ice40 -top mydesign -blif mydesign.blif
"

# Read BLIF netlist (mydesign.blif)
# Read physical constraints file (mydesign.pcf)
# Write ASCII representation of bitstream (mydesign.asc)
arachne-pnr -s 1 -d 8k -p mydesign.pcf -o mydesign.asc mydesign.blif

# Read ASCII representation of bitstream (mydesign.asc)
# Write bitstream binary file (mydesign.bin)
icepack mydesign.asc mydesign.bin

# Program FPGA board
iceprog mydesign.bin
```

- Our flow and the “small” iCE40 FPGAs are powerful enough for “real-world” applications, not just academic experiments and proof-of-concepts.
- At 32C3 we presented a SoC featuring a RISC-V 32 Bit processor and a small framebuffer, running code generated by GCC.
- Synthesis of this SoC with Yosys completes in under a minute on average PC hardware and the SoC only fills about half of the FPGA, leaving space for additional peripherals.





# Comparison with Lattice iCEcube2 Flow

	Yosys Arachne-pnr	Synplify Pro SBT Backend	Lattice LSE SBT Backend
Packed LCs	2996	2647	2533
LUT4	2417	2147	2342
DFF	1005	1072	945
CARRY	497	372	372
RAM4K	8	7	8
Synthesis Time	30 seconds	30 seconds	21 seconds
Implementation Time	81 seconds	405 seconds	415 seconds

## Notes:

- 1) Timings for Intel Core2 Duo 6300 at 1860 MHz running Ubuntu 15.04.
- 2) Using iCEcube2.2014.12 because I had troubles activating the license on newer versions.
- 3) SoC from 32C3 presentation without internal boot memory and frame buffer because Synplify Pro and LSE both could not infer implementations using iCE40 block RAM resources from the behavioral Verilog code.

# Timing Analysis Comparison

Design	Timing Tool	Yosys Arachne-pnr (unconstrained)	Lattice LSE SBT Backend (constrained to 100 MHz)
PicoRV32_AXI (w/ reduced pin count)	sbtime	N/A	41.74 MHz
	icetime -i	54.33 MHz	41.75 MHz
	icetime -im	53.02 MHz	41.40 MHz
Navre AVR Clone (from Milkymist SoC)	sbtime	N/A	45.82 MHz
	icetime -i	29.89 MHz	45.59 MHz
	icetime -im	27.61 MHz	44.90 MHz
Whishbone SPI Core (from OpenCores)	sbtime	N/A	62.13 MHz
	icetime -i	42.62 MHz	62.23 MHz
	icetime -im	38.89 MHz	61.14 MHz

Current Limitations of IceTime: No STA (purely topological), No multiple clock domains, Pessimistic (-m) or estimated interconnect model, No modeling of launch/capture clock paths

# Yosys Synthesis for Xilinx 7-Series FPGAs (1/2)

- A Yosys script for Xilinx 7-Series FPGA synthesis looks very similar to an iCE40 script:

```
read_verilog example.v ...
synth_xilinx -top example -edif example.edif
```

- A Xilinx Vivado TCL script can be used to place and route the EDIF netlist generated by Yosys:

```
read_xdc example.xdc
read_edif example.edif
link_design -part xc7a35tcp236-1 -top example
opt_design; place_design; route_design
report_utilization; report_timing
write_bitstream -force example.bit
```

# Yosys Synthesis for Xilinx 7-Series FPGAs (2/2)

- See “`help synth_xilinx`” for the sequence of commands used for Xilinx synthesis.
  - [http://www.clifford.at/yosys/cmd\\_synth\\_xilinx.html](http://www.clifford.at/yosys/cmd_synth_xilinx.html)
- It is not very different from e.g. `synth_ice40`!
  - Adding support for new FPGA architectures to Yosys is not very hard!

# OSS Silego GreenPAK4 Flow

- Silego GreenPAK4 are small mixed-signal programmable logic devices (PLDs) with up to 25 (!) LUTs, a few FFs, Counters, DAC/ADC, Slave SPI, etc.
  - Andrew Zonenberg is writing OpenFPGA, a place and route tool targeting Silego GreenPAK4 (and maybe other small PLDs in the future):
    - <https://github.com/azonenberg/openfpga>
  - Yosys synthesis for GreenPAK4 with

```
synth_greenpak4 -top <top-module> -json <json-file>
```
- 
- See “`help write_json`” for help on the JSON output format created by Yosys.
  - This is the recommended file format for using Yosys as front-end for other tools, if there are no good arguments for using any of the other supported output file formats.
  - This format is also used for example in `hardcaml-yosys` (path from Verilog to HardCaml, a HDL embedded in Ocaml):
    - <https://github.com/ujamjar/hardcaml-yosys>

# Verilog to BLIF with Yosys

- Many (older FOSS and/or academic) tools in the digital design domain use the Berkeley Logic Interchange Format (BLIF) as input format.
  - Examples: ABC, SIS/MVSIS, CUDD, ...
- Yosys can be used to convert Verilog files to BLIF files, thus adding a Verilog front-end to those tools.
- This is of course sometimes as simple as:

```
read_verilog mydesign.v ...
synth -top mydesign
write_blif mydesign.blif
```

# Verilog to Simple Verilog

- Many tools support a (usually ill defined) simple subset of Verilog. In most cases this subset is not large enough to support any pre-existing real-world designs.
- **Yosys can** be used to **read fully-featured Verilog**, only synthesize the non-trivial high-level constructs, and then **write a much simpler Verilog** file.

```
read_verilog mydesign.v
prep -top mydesign
write_verilog -noattr mydesign_simple.v
```

(The “prep” command is like “synth”, but only does some conservative coarse-grain transformations.)

# ASIC Synthesis and Custom Flows

- ASIC flows use custom scripts instead of something like a generic “synth\_asic” command. This allows users to better control their flow and customize it to their needs.
- Usually those script have a generic part that is using a command like “synth”, followed by the target-specific custom part.
- The same general layout applies to most non-ASIC custom flows as well.

Generic part →

```
# read design  
read_verilog mydesign.v
```

```
# generic synthesis  
synth -top mytop
```

Target-specific part →

```
# mapping to mycells.lib  
dfflibmap -liberty mycells.lib  
abc -liberty mycells.lib  
opt_clean
```

```
# write synthesized design  
write_edif synth.edif
```



## Yosys Formal Verification Flows

Property checking with built-in SAT solver

Property checking with ABC using miter circuits

Property checking with yosys-smtbmc

Formal and structural equivalence checking

# Verilog Language Features for Formal Verification (1/2)

- **SystemVerilog** adds some features that are key for formal verification, such as the **assert** and **assume** keywords. For example:

```
module example010(input [7:0] A, B, output Y);  
    assign Y = A[0] ^ B[0];  
    assume property (A+1 == B);  
    assert property (Y);  
endmodule
```

- E.g. the Yosys “sat” command can be used to prove that the asserts hold under the assumptions:

```
read_verilog -formal example010.v  
prep -top example010  
sat -set-assumes -prove-asserts
```

# Verilog Language Features for Formal Verification (2/2)

- In addition to “assert property” and “assume property” statements, SystemVerilog supports `assert()` and `assume()` in **behavioral code** as well. For example:

```
module example011(input enable, input [7:0] A, B, output Y);
  assign Y = A[0] ^ B[0];
  always @* begin
    if (enable) begin
      assume (A+1 == B);
      assert (Y);
    end
  end
endmodule
```



# More “sat” command features

- Bounded Model Checking (BMC):
  - `sat -seq 50 -prove-asserts -set-assumes`
- Temporal Induction Proofs:
  - `sat -tempinduct -prove never_one 0`
- Writing traces as VCD files:
  - `sat ... -dump_vcd <vcd_filename> ...`
- Writing SAT problem in DIMACS format:
  - `sat ... -dump_cnf <dimacs_filename> ...`
- Interactive Troubleshooting:
  - `sat -seq 15 -set foo 23 -set-at 10 never_one 1 -show bar`

# Pros for and cons against using Yosys “sat” command

- Pros for “sat”
  - Integrated with Yosys, ease of use
  - Good reporting of counter-examples (e.g. VCD)
  - Command line options for setting up ad-hoc proofs
- Cons against “sat”
  - Using eager SMT solver based on MiniSAT (i.e. not very fast compared to state-of-the-art SMT solvers)
  - No theories of arrays (i.e. no native support for memories)
  - No advanced proof strategies (such as IC3)

# Miter circuits

- Miter circuits are circuits that are used to prove properties of circuits.
  - It is the circuit under test, **augmented with additional checker logic**, and an **output that goes high** when the **properties are violated**.
  - Usually the **regular outputs** of the circuit are **thrown away**.
  - Miter circuits can be used to **run formal proofs in external tools** that do not understand concepts such as asserts and assumptions.
  - Yosys ships with commands to create miter circuits from designs that use assert and assume statements.

# A simple sequential example (1/2)

```
module example030(input clk, output [63:0] state);
    reg [63:0] state = 123456789;

    function [63:0] xorshift64star;
        input [63:0] current_state;
        begin
            xorshift64star = current_state;
            xorshift64star = xorshift64star ^ (xorshift64star >> 12);
            xorshift64star = xorshift64star ^ (xorshift64star << 25);
            xorshift64star = xorshift64star ^ (xorshift64star >> 27);
            xorshift64star = xorshift64star * 64'd 2685821657736338717;
        end
    endfunction

    always @(posedge clk)
        state <= xorshift64star(state);

    assert property (state != 0);
endmodule
```



# A simple sequential example (2/2)

- Proving example using Yosys “sat” command and simple temporal induction:

```
read_verilog -formal example030.v
prep -top example030
sat -prove-asserts -tempinduct
```

- Proving using a miter circuit and IC3 implementation in ABC (via ABC command “pdr”):

```
read_verilog -formal example030.v
prep -top example030
miter -assert example030
techmap; opt -fast
write_blif example030_miter.blif
```

```
! yosys-abc -c 'read_blif example030_miter.blif; strash; pdr'
```

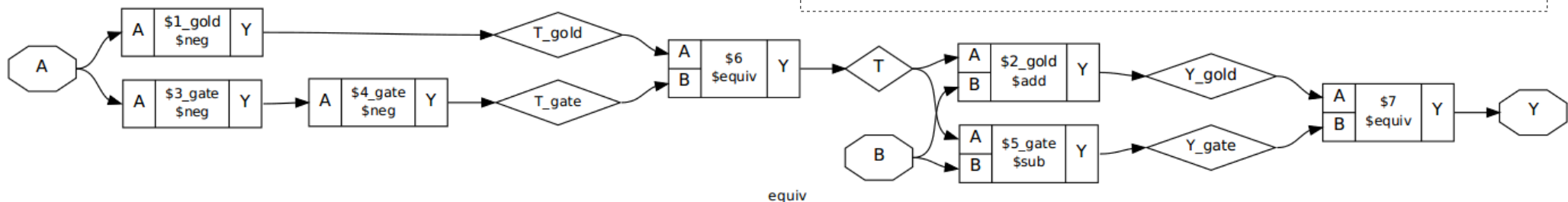
# Equivalence Checking

- The `equiv_*` commands in Yosys are for equivalence checking.
- This equivalence checker uses hints like net names to partition the circuit into to-be-proved-equivalent smaller circuits.
- This is extremely helpful for troubleshooting Yosys passes and/or perform pre-vs-post synthesis verification.
- The prover is capable of considering multiple time-steps (`equiv_simple -seq N`) and even perform temporal induction (`equiv_induct`).
- The command `equiv_struct` can be used for structural equivalence checking of mapped netlists.

```
module gold(input A, B, output Y);  
  wire T = -A;  
  assign Y = T + B;  
endmodule  
  
module gate(input A, B, output Y);  
  wire T = --A;  
  assign Y = T - B;  
endmodule
```

```
read_verilog example040.v  
opt_clean -purge  
equiv_make gold gate equiv  
hierarchy -top equiv  
clean -purge; show  
equiv_simple  
equiv_status -assert
```

Found 2 \$equiv cells in equiv:  
Of those cells 2 are proven and 0 are unproven.  
Equivalence successfully proven!



# Formal Verification with yosys-smtbmc

- **SMT-LIB2 is a language** for describing SMT problems
- Practically **all SMT solvers support it**, because SMT competition is using it as language for test problems
- Yosys can **convert Verilog designs to SMT-LIB2 code snippets**
- **Yosys-smtbmc** is a Python script that can **run various formal methods** on those code snippets.
- Comes with a Python library (`smtio.py`) for writing custom proofs.

# Simple SMT-LIB2 Example Code

```
(set-option :produce-models true)
(set-logic QF_LIA)


(declare-fun s () Int)
(declare-fun e () Int)
(declare-fun n () Int)
(declare-fun d () Int)
(declare-fun m () Int)
(declare-fun o () Int)
(declare-fun r () Int)
(declare-fun y () Int)

(assert (distinct s e n d m o r y))
(assert (distinct s m 0))

<...>
```

\* **This example uses the QF\_LIA logic**  
(Quantifier Free Linear Integer Arithmetic)

**Yosys-smtbmc uses QF\_AUFBV**  
(Quantifier Free, Arrays, Uninterpreted  
Functions, and Bit Vectors)

send		9567
+ more		+ 1085
-----		-----
money		10652

```
<...>

(define-fun is_digit ((i Int)) Bool (and (<= 0 i) (>= 9 i)))
(assert (and (is_digit s) (is_digit e) (is_digit n) (is_digit d)
             (is_digit m) (is_digit o) (is_digit r) (is_digit y)))

(define-fun four_digits ((d1 Int) (d2 Int) (d3 Int) (d4 Int)) Int
  (+ (* 1000 d1) (* 100 d2) (* 10 d3) d4))
(define-fun five_digits ((d1 Int) (d2 Int) (d3 Int) (d4 Int) (d5 Int)) Int
  (+ (* 10000 d1) (four_digits d2 d3 d4 d5)))

(assert (= (+ (four_digits s e n d) (four_digits m o r e))
           (five_digits m o n e y)))

(check-sat)
(get-value ((four_digits s e n d)
           (four_digits m o r e)
           (five_digits m o n e y)))
```

```
$ z3 -smt2 example050.smt2
sat
(((four_digits s e n d) 9567)
 ((four_digits m o r e) 1085)
 ((five_digits m o n e y) 10652))
```

# Yosys-smtbmc example

```
module example060(input clk);
  reg [3:0] cnt_a = 1, cnt_b = 0;
  reg [7:0] counter = 150;

  always @(posedge clk) begin
    if (cnt_a) begin
      if (cnt_a == 10) begin
        cnt_a <= 0;
        cnt_b <= 1;
      end else
        cnt_a <= cnt_a + 1;
      counter <= counter + 1;
    end else begin
      if (cnt_b == 10) begin
        cnt_b <= 0;
        cnt_a <= 1;
      end else
        cnt_b <= cnt_b + 1;
      counter <= counter - 1;
    end
  end
end

<...>
```

<...>

```
assert property (100 < counter && counter < 200);
assert property ((counter == cnt_a + 149) ||
                 (counter == 161 - cnt_b));
assert property ((cnt_a == 0) != (cnt_b == 0));
assert property (cnt_a <= 10);
assert property (cnt_b <= 10);

always @* begin
  if (cnt_a)
    assert (counter == cnt_a + 149);
  else
    assert (counter == 161 - cnt_b);
end
endmodule
```

```
yosys -q - <<EOT
read_verilog -formal example060.v
prep -top example060
write_smt2 -mem -bv -wires example060.smt2
EOT
```

```
yosys-smtbmc example060.smt2
yosys-smtbmc -i example060.smt2
```

# Thank You!

## Questions?

I have ~20 bonus slides after this slide, in case there is interest in ...

- SAT and SMT solver basics
- SMT-LIB2 Syntax / Introduction
- Details to Yosys SMT2 back-end
- Custom proofs with `smtio.py`



<http://www.clifford.at/papers/2016/yosys-synth-formal/>

↑↑ Link to slides and example codes ↑↑

# The SAT Problem

- Given a boolean expression in any number of variables, is there an assignment for the variables so that the expression evaluates to true?
- Usually the expression is specified as CNF (Commutative Normal Form, in this Context also Clause Normal Form), for example:  
$$(a + b + c) * (a + \text{NOT}(e)) * (e + \text{NOT}(c)) * \dots$$
- Conversion of any boolean expression to an equisatisfiable CNF has linear time complexity. (Conversion to an equivalent CNF would be exponential!)
- The 3-SAT problem (max 3 literals per clause) is equivalent to the general SAT problem.
- Nomenclature:
  - Clause – A disjunction, such as “(a + b + c)”
  - Positive Literal – Non-inverted variable, such as “a”
  - Negative Literal – Inverted variable, such as “NOT(e)”

# SAT Solvers

- SAT solvers solve a SAT problem
  - Input in CNF (common format: DIMACS)
  - When sat: solver provides a model: The assignments that satisfies the formula
  - Some solvers can provide kernels for unsat
- Many solvers support incremental solving:
  - Pushing/popping of clauses (via different mechanisms)
  - Solving second problem using learned clauses from first version of the problem
- Can't compare solvers using worst-case performance
  - Annual competitions using industrial and synthetic DIMACS files



# SMT Solvers

- SMT = SAT Modulo Theories
  - SAT Solver + Additional higher-level functions (“Theories”)
- Similar in functionality to SAT solvers, but higher-level problem descriptions
  - Structure of original problem is preserved much better
  - Some SMT solvers support quantifiers, making them QBF solvers
  - Some theories add infinite state, further extending the domain of SMT solvers
- Common input language for SMT solvers:
  - The SMT-LIB Language (currently in version 2.5)
  - Annual competitions with industrial and artificial problem sets
- Different “Theories” like Integers, Bit-Vectors, Arrays, Reals, etc..
  - A set of theories that might be used in conjunction is called a “Logic”
  - Logic used most often for BMC using Yosys:
    - QF\_AUFBV (Quantifier-free, Arrays, Uninterpreted functions, Bit-Vectors)
  - Yosys benchmarks will be included in 2016 SMT competition

# Formal Verification of digital Circuits: BMC and Temporal Induction

- Bounded Model Checking (BMC):

- Model of the circuit for N time steps
- Constraints for initial conditions
- Constraints for violated asserts
- When sat:
  - assert violated within N time steps
  - model contains counter-example

← Yosys: sat -seq N

- Temporal Induction

- Proving properties hold forever: Try induction!
- Model of the circuit for N+1 time steps
- Constraints for met asserts in first N time steps
- Constraints for violated asserts in last time step
- When unsat: Unbounded proof of properties

← Yosys: sat -tempinduct

- More advanced algorithms find additional invariants to prove with a smaller N

- For example IC3 (implemented in command pdr in ABC)

# SMT-LIB: General Syntax

- Lisp S-expression -like syntax
- First (non-option) statement must declare logic:  
`(set-logic QF_UF)`
- Since SMT-LIB 2.5 the solver may also auto-detect the logic:  
`(set-logic ALL)`
- A “variable” is declared as uninterpreted function without arguments:  
`(declare-fun myvar () Bool)`
- Constraints can be declared as asserts:  
`(assert (or myvar (not myvar)))`
- Use `(check-sat)` to run the solver

(The information on this slide is sufficient to encode any CNF as SMT-LIB file.)

# SMT-LIB: Options

- Disable printing of ( success ) after each statement:  
    `(set-option :print-success false)`  
    (this is the default for many solvers)
- Enable generation of modules:  
    `(set-option :produce-models true)`
- Setting of random seed (optional):  
    `(set-option :random-seed 42)`

# SMT-LIB: Inspecting the Model

- Using `(get-value)`
- Requires option `:produce-models`
- Only when `(check-sat)` returned `sat`
- Works with every quantifier-free closed expression

```
$ cat smtex01.smt2
(set-option :produce-models true)
(set-logic QF_UF)
(declare-fun a () Bool)
(declare-fun b () Bool)
(declare-fun c () Bool)
(assert (or a b))
(assert (and b (not c)))
(assert (or c (not a)))
(check-sat)
(get-value (a b c))
(get-value ((xor a b)))
```

```
$ z3 -smt2 smtex01.smt2
sat
((a false)
 (b true)
 (c false))
(((xor a b) true))
```

# SMT-LIB: Incremental Solving

- Create new context for assertions:  
(push 1)
- Destroy innermost context for assertions:  
(pop 1)
- Use a larger integer to push/pop multiple contexts at once.

```
$ cat smtex02.smt2
(set-option :produce-models true)
(set-logic QF_UF)
(declare-fun a () Bool)
(declare-fun b () Bool)
(assert (xor a b))
(check-sat)
```

```
(push 1)
(assert (and a b))
(check-sat)
```

```
(pop 1)
(assert (or a b))
(check-sat)
```

```
$ z3 -smt2 smtex02.smt2
sat
unsat
sat
```

# SMT-LIB: Functions, Constants

- Bool constants: `true` and `false`
  - Some theories define additional constants

- Defining a function:

```
(define-fun
  my_and                ; function name
  ((a Bool) (b Bool)) ; arguments
  Bool                 ; return type
  (and a b))          ; expression
```

# SMT-LIB: Custom Sorts

- Types are called sorts in SMT-LIB nomenclature
- Use `(declare-sort my_sort_name 0)` to declare custom sorts, for example:

```
$ cat smtex03.smt2
(set-option :produce-models true)

(set-logic QF_UF)
(declare-sort color 0)
(declare-fun r () color) ; red
(declare-fun g () color) ; green
(declare-fun b () color) ; blue
(assert (distinct r g b))

(declare-fun at () color) ; austria
(declare-fun de () color) ; germany
(declare-fun li () color) ; liechtenstein
(declare-fun it () color) ; italy
(declare-fun ch () color) ; switzerland

(assert (or (= at r) (= at g) (= at b)))
(assert (or (= de r) (= de g) (= de b)))
(assert (or (= li r) (= li g) (= li b)))
(assert (or (= it r) (= it g) (= it b)))
(assert (or (= ch r) (= ch g) (= ch b)))

(assert (distinct at de))
(assert (distinct at li))
(assert (distinct at it))
(assert (distinct at ch))
(assert (distinct ch de))
(assert (distinct ch li))
(assert (distinct ch it))

(check-sat)
(get-value (r g b at de li it ch))

$ z3 -smt2 smtex03.smt2
sat
((r color!val!0)
 (g color!val!1)
 (b color!val!2)
 (at color!val!1)
 (de color!val!2)
 (li color!val!2)
 (it color!val!2)
 (ch color!val!0))
```



# SMT-LIB: Theory of Bit-Vectors

- Declaring a Bit-Vector:  
`(declare-fun my_bv_var () (_ BitVec 16))`
- Concatenation and extraction:  
`(concat a b)`  
`((_ extract 11 8) c)`
- Operations on Bit-Vectors, for example:  
`(bvadd a b)`
- Bit-Vector constants:  
`#b1101001`
- Conversion from/to Bool:  
`(ite bool_expr #b1 #b0)`  
`(= bv_expr #b1)`

# SMT-LIB: Theory of Arrays

- Declaring an Array (16 elements, 8 bits each):

```
(declare-fun my_array () (Array (_ BitVec 4) (_ BitVec 8)))
```

- Accessing an Array element:

```
(select my_array #b0011)
```

- Creating a modified version of an Array:

```
(store my_array #b1010 #b00110011)
```

- Can be used to model memory efficiently!

# SMT Solvers for QF\_AUFBV

- The logic QF\_AUFBV is required for our verification flow. The following solvers support this logic (listed from fastest to slowest, results may vary).
- Yices2
  - “This EULA permits use of the software **only for projects that do not receive external funding** other than government research grants and contracts. Any **other use requires a commercial license.**”
- Z3
  - “Z3 is licensed under the **MIT license.**” This is the default solver for yosys - smt bmc.
- CVC4
  - “The source code of CVC4 is open and available to students, researchers, software companies, and everyone else to study, to modify, and to redistribute original or modified versions; distribution is under the terms of the **modified BSD** license. However, CVC4 can be configured (and is, by default) to link against some **GPLed libraries**, [...]”
- MathSAT5
  - “MathSAT5 is available for **research and evaluation purposes only**. It can not be used in a commercial environment, particularly as part of a commercial product, without written permission. [...]”

# Yosys SMT2 Back-End (1/2)

- The Yosys SMT2 back-end converts digital designs into SMT2 code
  - Use option `-bv` to enable support for theory of bit-vectors (for RTL cells)
  - Use option `-mem` to enable support for theory of arrays (for memories)
- The SMT2 back-end defines one sort for each module that represents the state of the module at one point in time
  - Example: `module main` → `main_s`
  - Usage: `(declare-fun s01 () main_s)`
- It also defines a function for each net
  - By default only for some wires, use for example `-wires` or `-regs` for better coverage
  - Example: `main.foobar` in `s01` → `(|main_n foobar| s01)`
- And it defines a function for state transitions
  - Example: `(assert (main_t s01 s02))`

# Yosys SMT2 Back-End (2/2)

- Additional functions for module states:
  - Initial state: `(main_i s01)`
  - Assertions met: `(main_a s01)`
  - Assumptions met: `(main_u 01)`
- Access to memories:
  - Get array for memory `mymem`: `(|main_m mymem| s01)`
- More information: `help write_smt2`

# Yosys-SMTBMC Tool

- Extending the SMT2 code created by the SMT2 back-end to useful proofs is not hard, but sometimes inconvenient.
- The tool `yosys-smtbmc` can read the SMT2 files created by `write_smt2` and to perform
  - Bounded Model Checking and
  - Temporal Induction Proofs
- It can use various different SMT solvers (default: z3)
- Can dump traces as SMT2 files for performance comparisons
- Can write counter-examples as VCD files

# Custom proofs with `smtio.py`

- `smtio.py` is a python library for remote controlling SMT solvers and building proofs around the SMT2 files written by Yosys
  - `yosys-smtbmc` is just a small script using `smtio.py`
- PicoRV32 CPU
  - `sync.sh` / `async.sh` bounded equivalence checker for different configurations of the CPU core (`async.sh` for cores with different cycles per instructions)
- PonyLink Slave Reset Verification
  - PonyLink is a chip-to-chip master-slave communications core using a single half-duplex communications channel
  - A custom `smtio.py` proof is used to prove that
    - The slave will always stop sending after a short period of time, regardless of its initial state
    - Resetting the slave over the link will always succeed, regardless of the slaves initial state