

Formal Verification with SymbiYosys and Yosys-SMTBMC

Clifford Wolf

Availability of various EDA tools for students, hobbyists, enthusiasts

- FPGA Synthesis

- Free to use:
 - Xilinx Vivado WebPack, etc.
- Free and Open Source:
 - Yosys + Project IceStorm
 - VTR (Odin II + VPR)

- HDL Simulation

- Free to use:
 - Xilinx XSIM, etc.
- Free and Open Source:
 - Icarus Verilog, Verilator, etc.

- Formal Verification

- Free to use:
 - ???
- Free and Open Source:
 - ???

.. and people in the industry are complaining they can't find any verification experts to hire!

Yosys, Yosys-SMTBMC, SymbiYosys

- Yosys
 - FOSS Verilog Synthesis tool and more
 - highly flexible, customizable using scripts
 - Formal Verification (Safety Properties, Liveness Properties, Equivalence, Coverage)
 - FPGA Synthesis for iCE40 (Project IceStorm), Xilinx 7-series (Vivado for P&R), GreenPAK4 (OpenFPGA), Gowin Semi FPGAs, MAX10, ...
 - ASIC Synthesis (full FOSS flows: Qflow, Coriolis2)
- Yosys-SMTBMC
 - A flow with focus on verification of safety properties using BMC and k-induction, using SMT2 circuit descriptions generated by Yosys
- SymbiYosys
 - A unified front-end for many Yosys-based formal verification flows

SymbiYosys Features

- Bounded verification of safety properties
- Unbounded verification of safety properties
- Generation of test benches from cover statements
- Verification of liveness properties
- Formal equivalence checking [TBD]
- Reactive Synthesis [TBD]

Solvers:

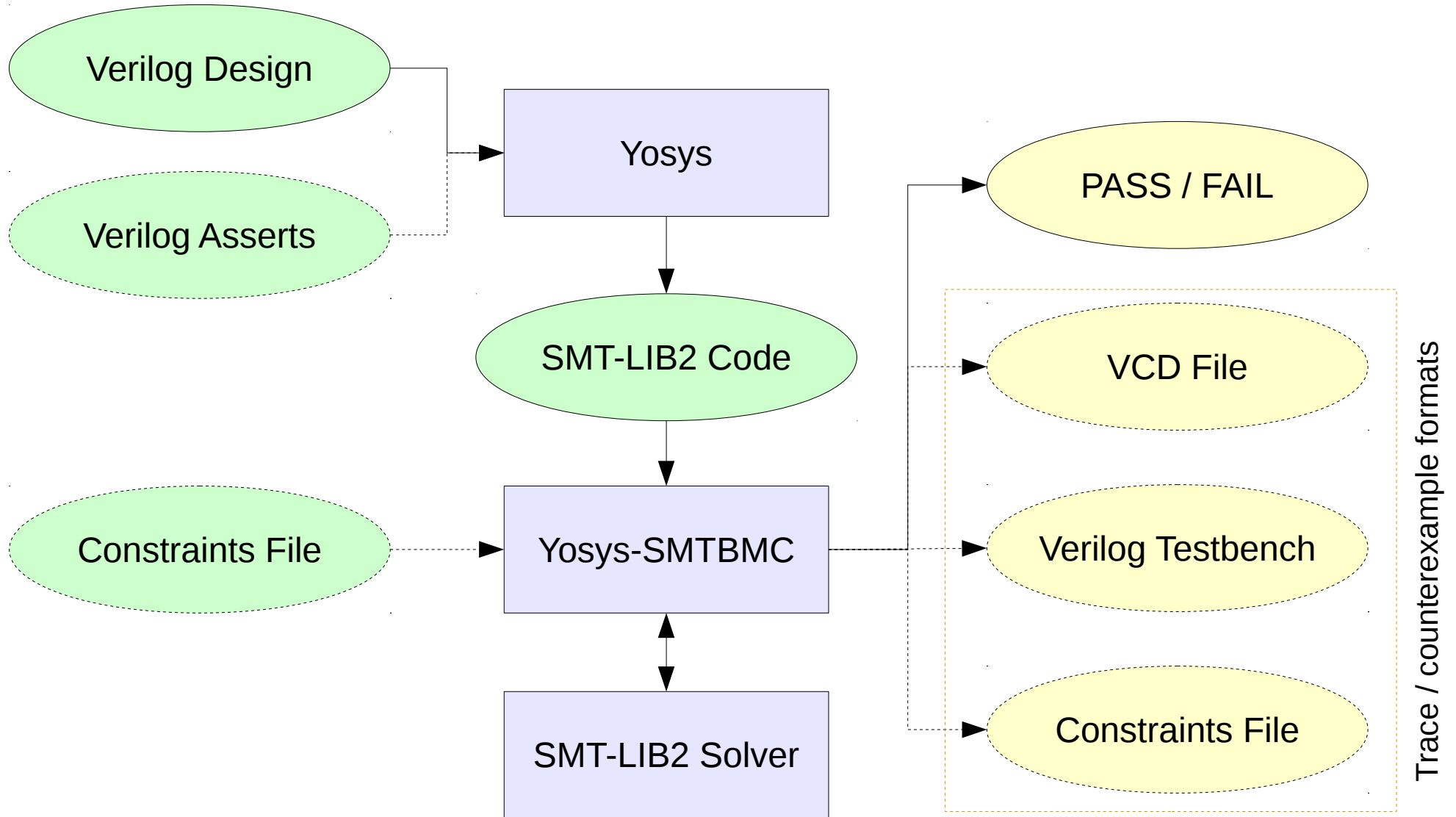
– SMT2:

- Yices 2, Z3, Boolector, CVC4, Mathsat
- easy to extend to any SMT2 solver with QF_AUFBV, QF_ABV, QF_BV, or QF_UFBV support

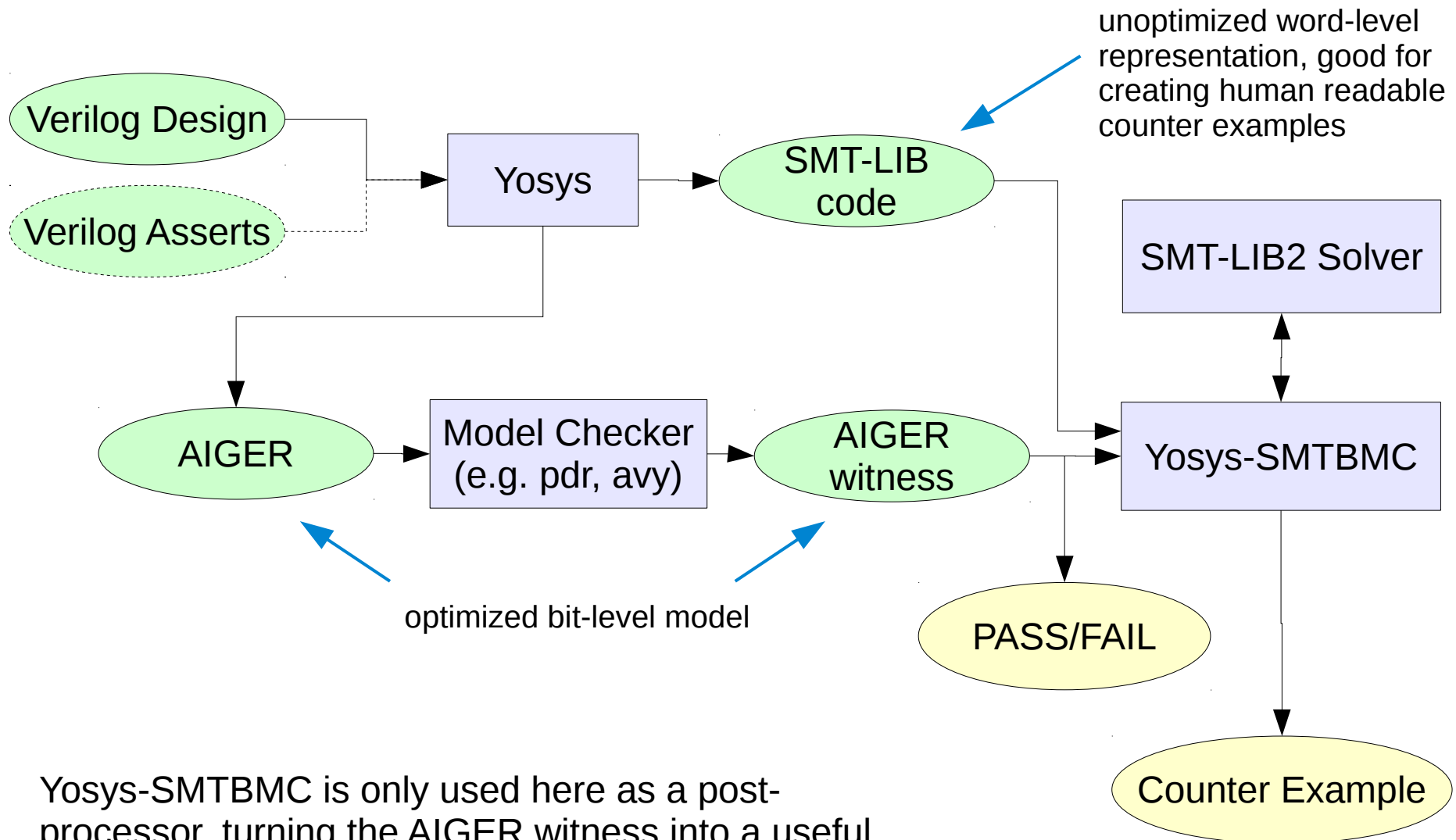
– AIGER:

- super_prove, Avy, everything in ABC (including pdr)
- easy to extend to any AIGER solver for safety and/or liveness properties

SymbiYosys flow with Yosys-SMTBMC



SymbiYosys flow with AIGER model checker



Yosys-SMTBMC is only used here as a post-processor, turning the AIGER witness into a useful human readable counter example (e.g. VCD).

Hello World

hello.v

```
module hello (  
    input clk, rst,  
    output [3:0] cnt  
);  
    reg [3:0] cnt = 0;  
  
    always @(posedge clk) begin  
        if (rst)  
            cnt <= 0;  
        else  
            cnt <= cnt + 1;  
    end  
  
    `ifdef FORMAL  
        assume property (cnt != 10);  
        assert property (cnt != 15);  
    `endif  
endmodule
```

hello.sby

```
[options]  
mode prove  
depth 10  
  
[engines]  
smtbmc z3  
  
[script]  
read_verilog -formal hello.v  
prep -top hello  
  
[files]  
hello.v
```

Hello World

```
$ sby -f hello.sby
SBY [hello] Removing direcory 'hello'.
SBY [hello] Copy 'hello.v' to 'hello/src/hello.v'.
SBY [hello] engine_0: smtbmc z3
...
...
...
SBY [hello] engine_0.basecase: finished (returncode=0)
SBY [hello] engine_0: Status returned by engine for basecase: PASS
SBY [hello] engine_0.induction: finished (returncode=0)
SBY [hello] engine_0: Status returned by engine for induction: PASS
SBY [hello] summary: Elapsed clock time [H:MM:SS (secs)]: 0:00:00 (0)
SBY [hello] summary: Elapsed process time [H:MM:SS (secs)]: 0:00:00 (0)
SBY [hello] summary: engine_0 (smtbmc z3) returned PASS for basecase
SBY [hello] summary: engine_0 (smtbmc z3) returned PASS for induction
SBY [hello] summary: successful proof by k-induction.
SBY [hello] DONE (PASS, rc=0)
```

- The sby option `-f` causes sby to remove the output directory if it already exists.
- The output directory contains all relevant information, including copies of the HDL design files.

Yosys Formal Verilog Specs

`assert()`, `assume()`, `restrict()`

- Yosys does not support SVA properties! Only immediate assertions, plus some convenient non-standard Verilog features.
- `assert(expression);`
 - Error if the expression evaluates to false
- `assume(expression);`
 - Simulation: Error if expression evaluates to false
 - Verification: Only consider traces where expression is true
- `restrict(expression);`
 - Simulation: Ignored.
 - Verification: Only consider traces where expression is true
- When to use `assume()`, when `restrict()`?
 - Use `assume()` if your asserts depend on it, use `restrict()` when it's just there to help with the proof, but the asserts would hold without it.

Fairness and Liveness

- `if (req) assume(s_eventually resp);`
 - Assume the LTL spec “ $G (req \rightarrow F resp)$ ”
- `if (req) assert(s_eventually resp);`
 - Assert the LTL spec “ $G (req \rightarrow F resp)$ ”
- Fairness and Liveness is only supported in AIGER-based flows at the moment.

Free Variables

- `wire [7:0] cmd = $anyseq;`
 - Behaves like an additional primary input
- `wire [7:0] cmd = $anyconst;`
 - Behaves like an additional primary input that is latched in the first cycle.
- `rand reg [7:0] cmd;`
- `rand const reg [7:0] cmd;`
 - For improved SV compatibility (only valid SV in checker .. endchecker block, Yosys supports it everywhere)

```

module fib (
    input clk, pause, start,
    input [3:0] n,
    output reg busy, done,
    output reg [9:0] f
);
    reg [3:0] count;
    reg [9:0] q;

    initial begin
        done = 0;
        busy = 0;
    end

    always @(posedge clk) begin
        done <= 0;
        if (!pause) begin
            if (!busy) begin
                if (start)
                    busy <= 1;
                count <= 0;
                q <= 1;
                f <= 0;
            end else begin
                q <= f;
                f <= f + q;
                count <= count + 1;
                if (count == n) begin
                    busy <= 0;
                    done <= 1;
                end
            end
        end
    end
end

```

fib.v

```

`ifndef FORMAL
    always @(posedge clk) begin
        if (busy) begin
            assume (!start);
            assume ($stable(n));
        end

        if (done) begin
            case ($past(n))
                0: assert (f == 1);
                1: assert (f == 1);
                2: assert (f == 2);
                3: assert (f == 3);
                4: assert (f == 5);
                5: assert (f == 8);
            endcase
            cover (f == 13);
            cover (f == 144);
            cover ($past(n) == 15);
        end

        assume (s_eventually !pause);

        if (start && !pause)
            assert (s_eventually done);
    end
`endif
endmodule

```

fib_{prove, live, cover}.sby

fib_prove.sby

```
[options]
mode prove

[engines]
abc pdr

[script]
read_verilog -formal fib.v
prep -top fib

[files]
fib.v
```

Prove safety properties in `fib.v` using IC3 (pdr).

fib_live.sby

```
[options]
mode live

[engines]
aiger suprove

[script]
read_verilog -formal fib.v
prep -top fib

[files]
fib.v
```

Prove liveness properties in `fib.v`. This assumes that safety properties are already proven.

fib_cover.sby

```
[options]
mode cover
append 10

[engines]
smtbmc z3

[script]
read_verilog -formal fib.v
prep -top fib

[files]
fib.v
```

Create a trace for each cover statement in the design (and check asserts for that trace). Add 10 additional time steps after the cover statement has been reached.

parcase.v

```
module parcase (input clk, A, B, C, D, E, BUG, output reg Y);
  always @(posedge clk) begin
    Y <= 0;
    if (A != B || BUG) begin
      (* parallel_case *)
      case (C)
        A: Y <= D;
        B: Y <= E;
      endcase
    end
  end
endmodule
```

```
[script]
read_verilog -formal parcase.v
prep -top parcase
assertpmux

$ sby -f parcase.sby
...
... Assert failed in parcase: parcase.v:6
...
SBY [parcase] DONE (FAIL)
```

memcmp.v

```
module memory1 (  
    input clk,  
    input [3:0] wstrb,  
    input [15:0] waddr,  
    input [15:0] raddr,  
    input [31:0] wdata,  
    output [31:0] rdata  
);  
    reg [31:0] mem [0:2**16-1];  
    reg [15:0] buffered_raddr;  
  
    // "transparent" read  
    assign rdata = mem[buffered_raddr];  
  
    always @(posedge clk) begin  
        if (wstrb[3]) mem[waddr][31:24] <= wdata[31:24];  
        if (wstrb[2]) mem[waddr][23:16] <= wdata[23:16];  
        if (wstrb[1]) mem[waddr][15: 8] <= wdata[15: 8];  
        if (wstrb[0]) mem[waddr][ 7: 0] <= wdata[ 7: 0];  
        buffered_raddr <= raddr;  
    end  
endmodule
```

memcmp.v

```
module memory2 (
    input clk,
    input [3:0] wstrb,
    input [15:0] waddr,
    input [15:0] raddr,
    input [31:0] wdata,
    output [31:0] rdata
);
    reg [31:0] mem [0:2**16-1];
    reg [31:0] buffered_wdata;
    reg [31:0] buffered_rdata;
    reg [3:0] buffered_wstrb;
    reg waddr_is_not_raddr;

    wire [31:0] expanded_wstrb = {{8{wstrb[3]}}, {8{wstrb[2]}}, {8{wstrb[1]}}, {8{wstrb[0]}}};
    wire [31:0] expanded_buffered_wstrb = {{8{buffered_wstrb[3]}}, {8{buffered_wstrb[2]}},
                                           {8{buffered_wstrb[1]}}, {8{buffered_wstrb[0]}}};

    assign rdata = waddr_is_not_raddr ? buffered_rdata :
        (buffered_wdata & expanded_buffered_wstrb) |
        (buffered_rdata & ~expanded_buffered_wstrb);

    always @(posedge clk) begin
        mem[waddr] <= (wdata & expanded_wstrb) | (mem[waddr] & ~expanded_wstrb);
        buffered_wstrb <= wstrb;
        buffered_wdata <= wdata;
        buffered_rdata <= mem[raddr];
        waddr_is_not_raddr <= waddr != raddr;
    end
endmodule
```


memcmp.v

```
module memcmp (
    input clk,
    input [3:0] wstrb,
    input [15:0] waddr,
    input [15:0] raddr,
    input [31:0] wdata,
    output [31:0] rdata1,
    output [31:0] rdata2
);
    memory1 mem1 (
        .clk (clk ), .wstrb(wstrb ),
        .waddr(waddr ), .raddr(raddr ),
        .wdata(wdata ), .rdata(rdata1)
    );

    memory2 mem2 (
        .clk (clk ), .wstrb(wstrb ),
        .waddr(waddr ), .raddr(raddr ),
        .wdata(wdata ), .rdata(rdata2)
    );
endmodule
```

memcmp.smtc

```
initial
assume (= [mem1.mem] [mem2.mem])

always 1
assert (= [mem1.mem] [mem2.mem])
assert (= [rdata1] [rdata2])
```

memcmp.sby

```
[options]
mode prove
smtc memcmp.smtc
depth 10

[script]
read_verilog -formal memcmp.v
prep -nordff -top memcmp

...
```

memcheck.v

```
module memory (  
    input clk, we,  
    input [31:0] addr,  
    input [7:0] wdata,  
    output reg [7:0] rdata  
);  
  
    reg [7:0] bank_0 [0:2**30-1];  
    reg [7:0] bank_1 [0:2**30-1];  
    reg [7:0] bank_2 [0:2**30-1];  
    reg [7:0] bank_3 [0:2**30-1];  
  
    always @(posedge clk) begin  
        case (addr[1:0])  
            2'b 00: begin  
                rdata <= bank_0[addr >> 2];  
                if (we) bank_0[addr >> 2] <= wdata;  
            end  
            2'b 01: begin  
                rdata <= bank_1[addr >> 2];  
                if (we) bank_1[addr >> 2] <= wdata;  
            end  
            2'b 10: begin  
                rdata <= bank_2[addr >> 1]; // <- BUG  
                if (we) bank_2[addr >> 2] <= wdata;  
            end  
            2'b 11: begin  
                rdata <= bank_3[addr >> 2];  
                if (we) bank_3[addr >> 2] <= wdata;  
            end  
        endcase  
    end  
endmodule
```

memcheck.v

```
module memcheck (
    input clk, we,
    input [31:0] addr,
    input [7:0] wdata,
    output [7:0] rdata
);

memory uut (
    .clk (clk ),
    .we (we ),
    .addr (addr ),
    .wdata(wdata),
    .rdata(rdata)
);

reg monitor_valid = 0;
wire [31:0] monitor_addr = $anyconst;
reg [7:0] monitor_data;

always @(posedge clk) begin
    if ((addr == monitor_addr) && we) begin
        monitor_valid <= 1;
        monitor_data <= wdata;
    end
    if (($past(addr) == monitor_addr) && monitor_valid &&
        $past(monitor_valid)) begin
        assert (rdata == $past(monitor_data));
    end
end

end
endmodule
```

memcheck.sby

```
[options]
mode bmc
expect fail
depth 10
```

...

multiclk.v

```
module multiclk(input clk, output [3:0] counter_a, counter_b);
    reg [3:0] counter_a = 0;
    reg [3:0] counter_b = 0;

    always @(posedge clk)
        counter_a <= counter_a + 1;

    always @(posedge clk)
        counter_b[0] <= !counter_b[0];

    always @(negedge counter_b[0])
        counter_b[1] <= !counter_b[1];

    always @(negedge counter_b[1])
        counter_b[2] <= !counter_b[2];

    always @(negedge counter_b[2])
        counter_b[3] <= !counter_b[3];

    assert property (counter_a == counter_b);
endmodule
```

multiclk.sby

```
...

[script]
read ...
prep ...
clk2fflogic

...
```

setreset.v

```
module setreset(input clk, input set, rst, d, output q1, q2);
    reg q1 = 0;

    always @(posedge clk, posedge set, posedge rst)
        if (rst)          q1 <= 0;
        else if (set)     q1 <= 1;
        else              q1 <= d;

    reg q2_s = 0, q2_r = 0, q2_l;
    wire q2 = q2_l ? q2_s : q2_r;

    always @(posedge clk, posedge set)
        if (set) q2_s <= 1;
        else    q2_s <= d;

    always @(posedge clk, posedge rst)
        if (rst) q2_r <= 0;
        else    q2_r <= d;

    always @* begin
        if (rst)          q2_l <= 0;
        else if (set)     q2_l <= 1;

        assert property (q1 == q2);
    end
endmodule
```

setreset.sby

```
...

[script]
read ...
prep ...
clk2fflogic

...
```

End-to-end Formal Verification of RISC-V Cores with `riscv-formal`

- `riscv-formal` is a framework for formal verification of RISC-V Processor Cores against ISA spec using SymbiYosys.
- Processors must implement RVFI (RISC-V Formal Interface) to be verifiable using `riscv-formal`.
- A separate verification task for each instruction
- And a few additional verification tasks to verify consistent state between instructions and correct implementation of memory I/O
- `riscv-formal` spec is formally verified for equivalence against `riscv-isa-sim` (aka Spike), the official reference simulator (C++).
- Currently supported ISAs: RV32I, RV32IC
- Currently supported cores: PicoRV32 and Rocket.
- Public GitHub repository is up to 6 months behind development branch!
 - i.e. what is visible in the public GitHub repo does not reflect the current status of the project.

Future Work

- Very limited support for SVA properties
 - Using AST transformations to clocked always blocks with immediate assertions.
 - However: I highly recommend sticking to immediate assertions in new code. It does not look like FOSS simulators are going to support SVA properties anytime soon.
- Improved support for Verilog x-propagation
 - Currently only available with Yosys “sat” flow
 - Adding a Yosys pass that transforms the design into a circuit problem with explicit *__x nets
- Better integration with commercial Verific library
 - Already supported: Synthesizable VHDL and SystemVerilog
 - Under construction: Full SVA and PSL support

Commercial Offers by Symbiotic EDA

- SymbiYosys with commercial add ons
 - Parsers for SystemVerilog, VHDL, PSL
 - Additional commercial solvers
 - Available as
 - on-premise solution or
 - cloud instances rented by the hour
- Formal Verification Consulting Services by the day
- SymbiYosys support contract by the month

Thanks!

Full slide deck, relevant links and examples:

<http://www.clifford.at/papers/2017/smtbmc-sby/>

Questions?

Keywords:

- Yosys, Yosys-SMTBMC
- SymbiYosys
- BMC, k-Induction
- Safety Properties
- `assert()`, `assume()`, `restrict()`
- Liveness Properties
- `assert(s_eventually ...)`
- `$anyseq`, `$anyconst`
- `assertpmux`, `clk2fflogic`
- SMT2 Encodings, AIGER
- RISC-V Formal



References

- Bounded Model Checking, Armin Biere, Handbook of Satisfiability. Armin Biere, Marijn Heule, Hans von Maaren and Toby Walsh (Eds.), pages 457-481
- Satisfiability Modulo Theories, Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia and Cesare Tinelli. Handbook of Satisfiability. Armin Biere, Marijn Heule, Hans von Maaren and Toby Walsh (Eds.), pages 852-885
- Temporal Induction by Incremental SAT Solving. Niklas Een, Niklas Sörensson, BMC 2003.
- The SMT-LIB Standard: Version 2.5, by Clark Barrett, Pascal Fontaine, and Cesare Tinelli.
- Boolector 2.0. Aina Niemetz, Mathias Preiner, Armin Biere. Journal of Satisfiability, Boolean Modeling and Computation (JSAT), vol. 9, 2015, pages 53-58.
- Yices 2.2. Bruno Dutertre. CAV'2014.