Functional Programming for Hardware Design: The Good, The Bad, The Ugly.

Carl Seger

CSE, Chalmers

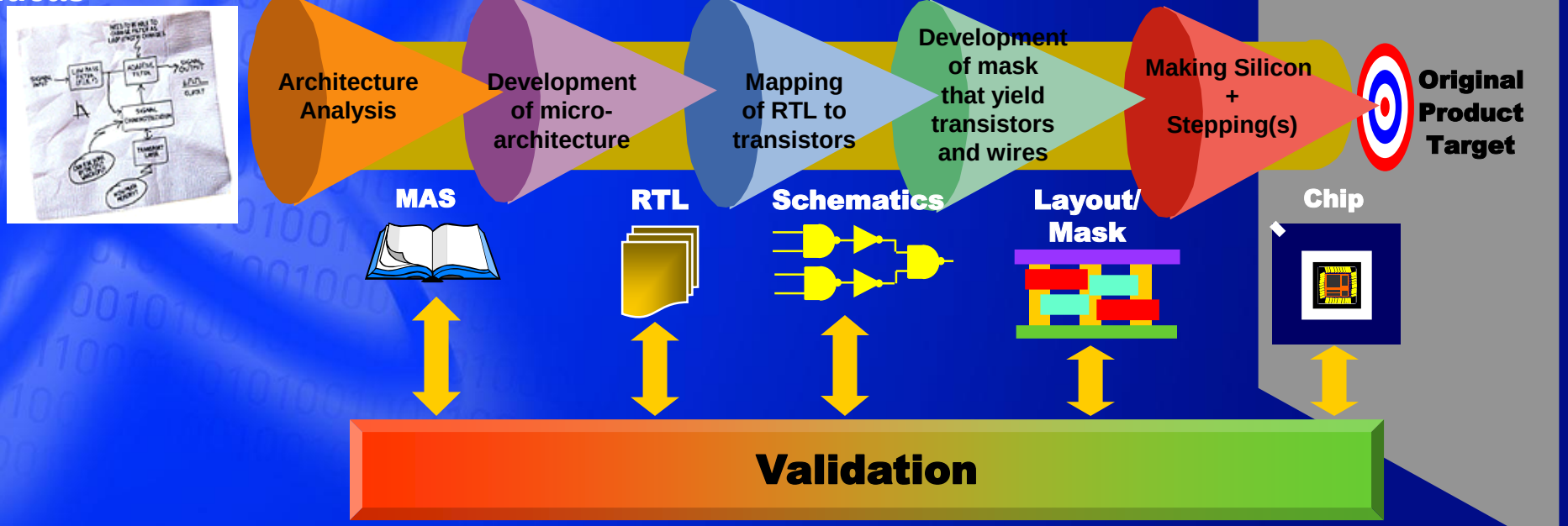
February 13, 2020

Outline

- Context of talk
- History of Voss/fl
- Using fl for HW design
- Conclusions
  - The Good
  - The Bad
  - The Ugly

Context

The Design Process at 10,000 ft

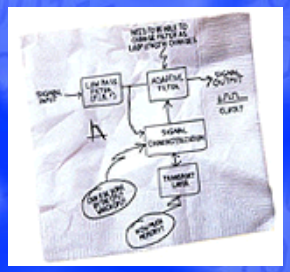


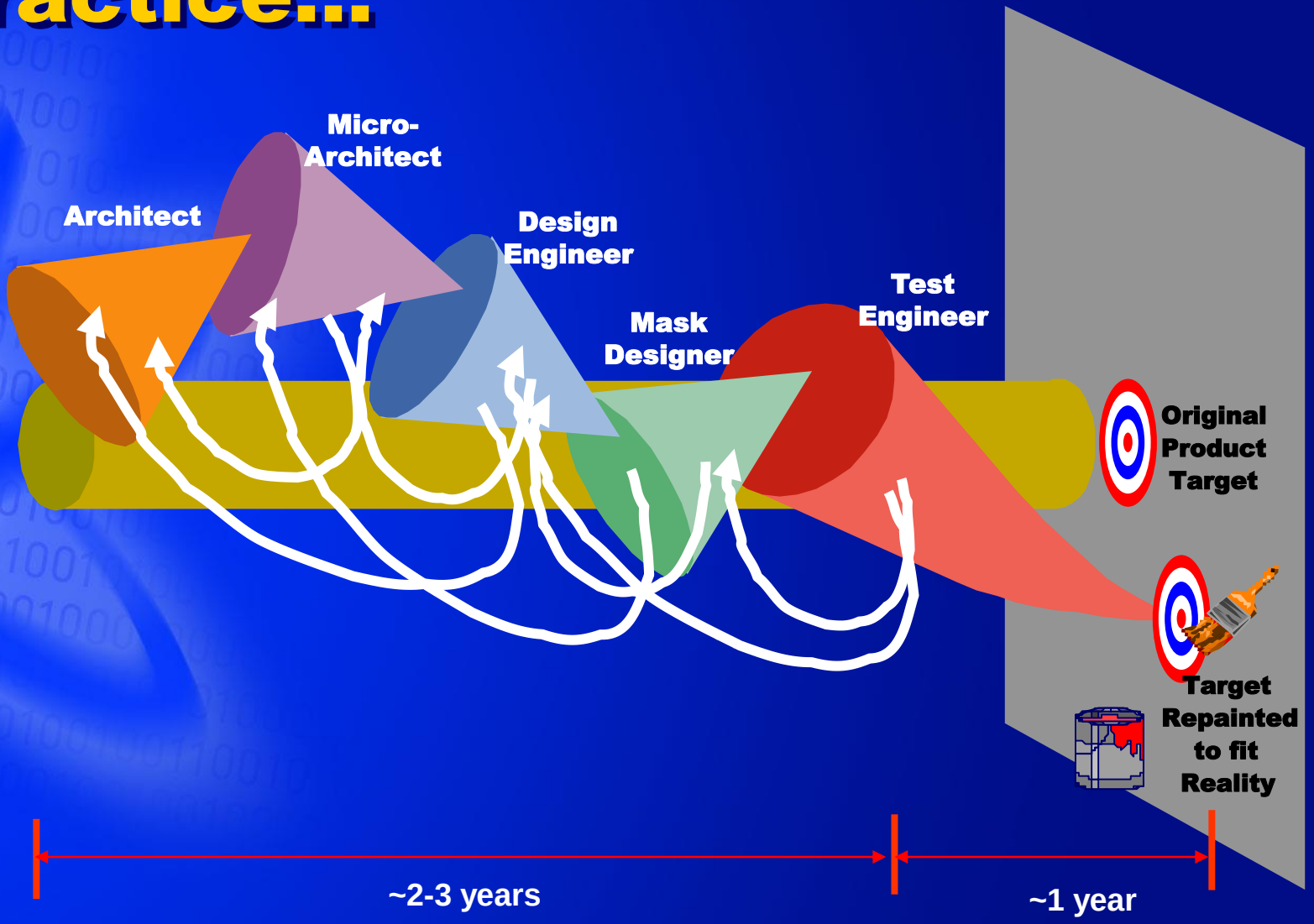
MAS: Micro-Architecture Specification

RTL: Register-Transfer Language

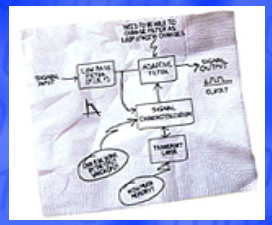
This is the theory...

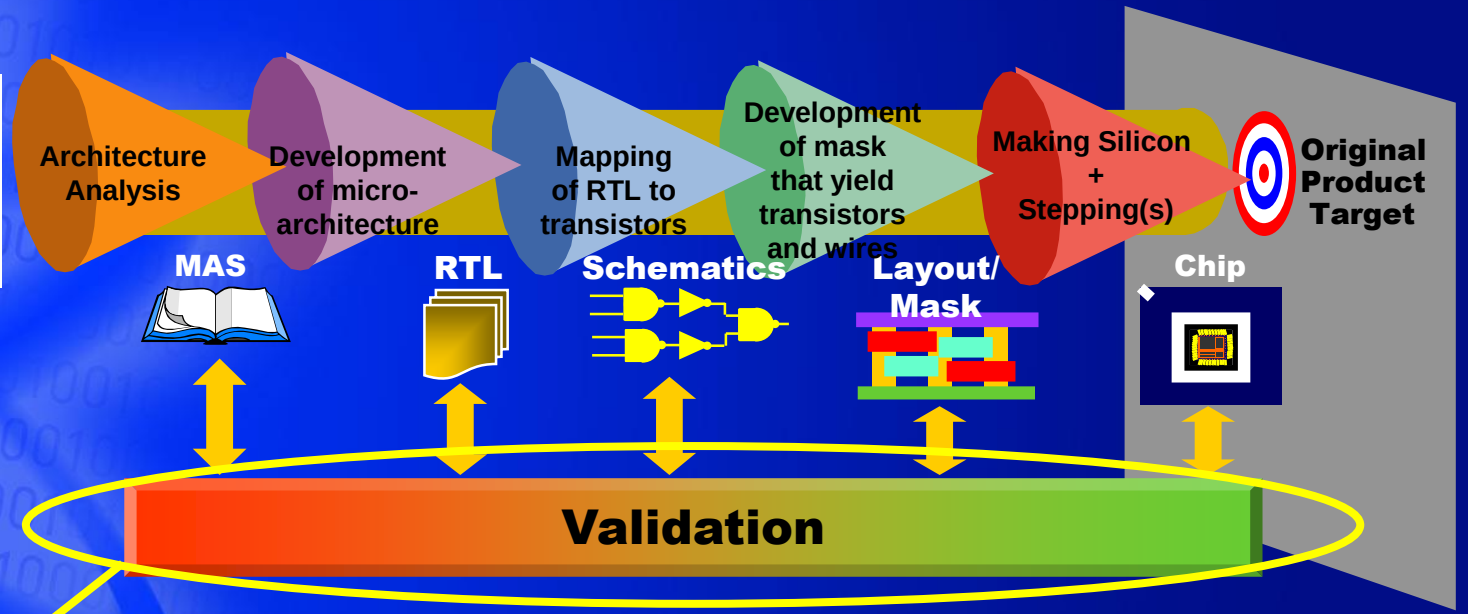
In Practice...




Micro-
Architect
Architect
Design
Engineer
Mask
Designer
Test
Engineer
Original
Product
Target
Target
Repainted
to fit
Reality
~2-3 years
~1 year

Validation





How to: 1) check we captured what we wanted
2) check that we did not make a mistake along the way

History of Voss/fl

Brief History

- Once upon a time (1990) at University of British Columbia there were:
  - a generalized symbolic simulator (symbolic trajectory evaluator - STE) written in C and highly optimized,
  - a general purpose theorem prover (HOL), and
  - two experts eyeing each other suspiciously.
- Heureka moment:
  - Use STE as decision procedure for HOL
    - Needed to evaluate expressions
      - With Boolean expressions + quantification
- Birth of fl….

Brief History cont.

- Reality hits:
  - Very cumbersome to use.
    - Interacting through HOL, but all debugging happened down in fl
  - Started using fl by itself
    - Simple typos crashed the system → type system → full language
  - fl embedded in the Voss system became very powerful for formal verification work.
- Intel lost $470 million in 1995 due to FDIV bug.
  - "Invite" Carl to spend the summer at Intel…
- 25 years later…

The Four Phases of fl Usage

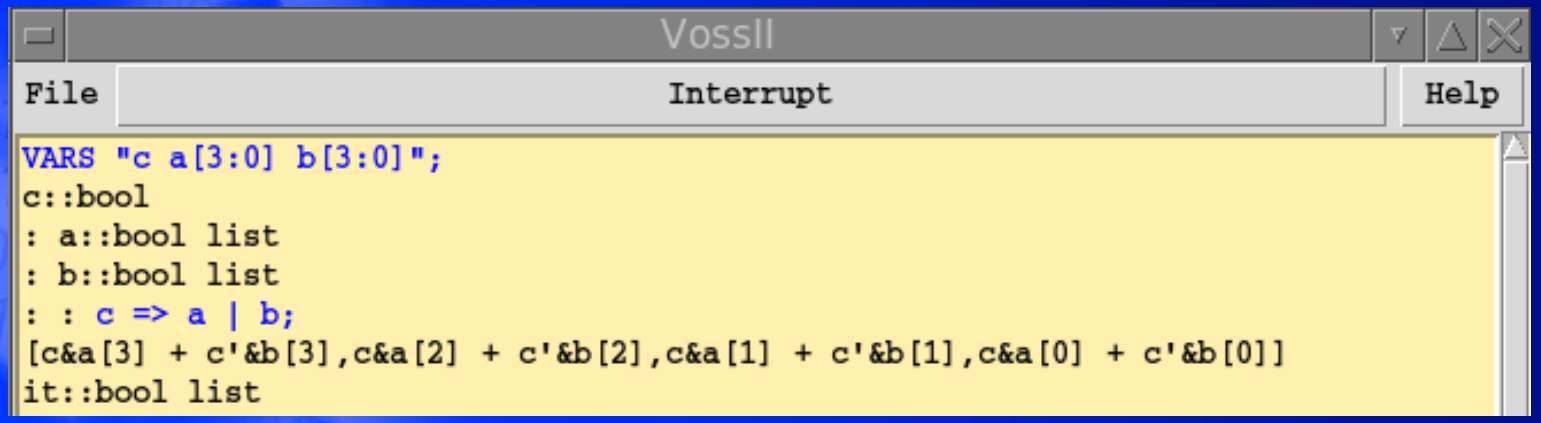
Phase 1: As Scripting & Implementation Language

- The STE engine needs to be controlled.
  - Usually differently for each hardware/proof effort.
- Integrating the Binary Decision Diagrams & SAT solver tightly into language makes
  - Creating custom decision procedures very easy
  - Makes debugging highly productive.
- An interpreted language is very helpful here
- Have complete control of the language allows rapid extensions and enhancements.

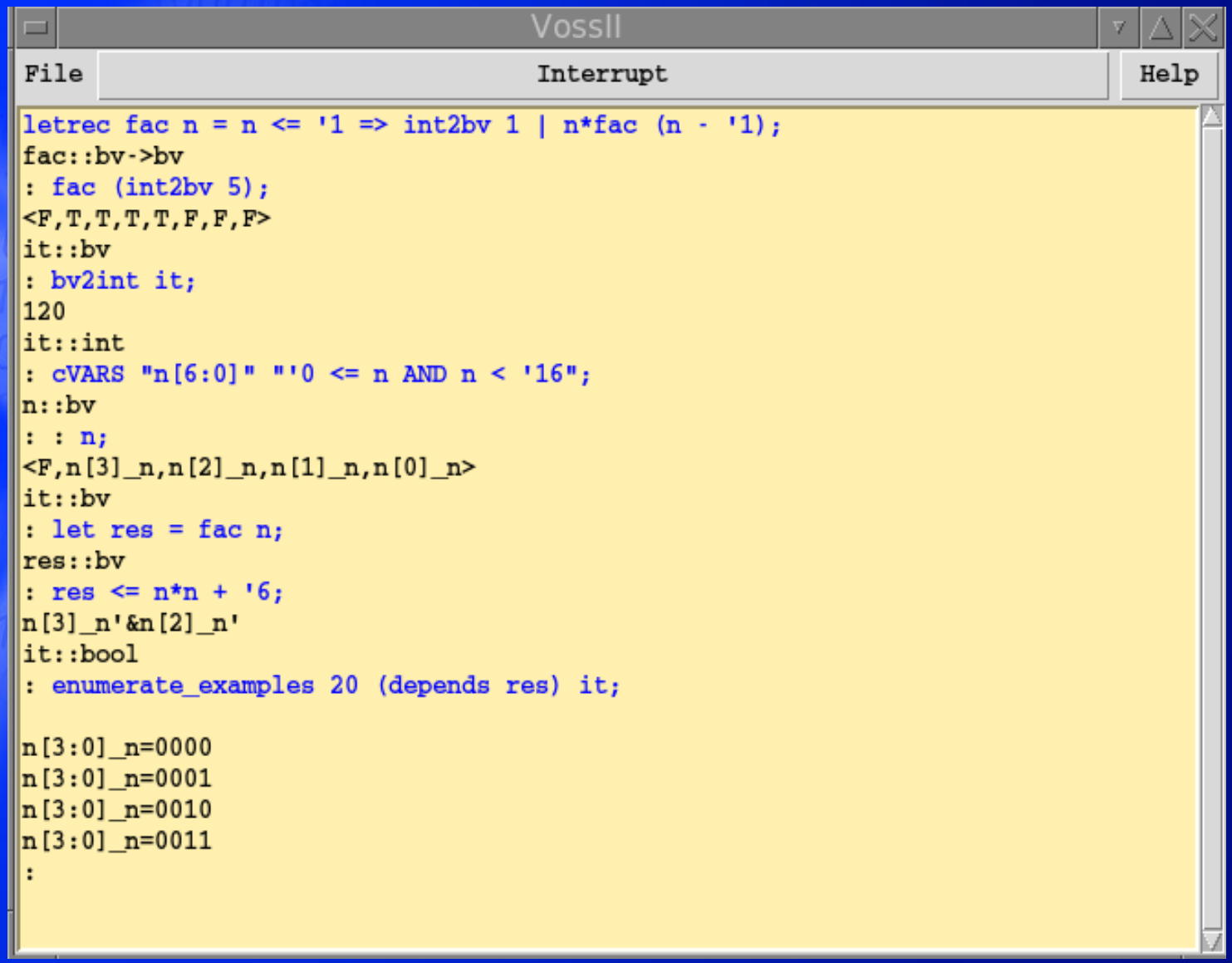
Phase 2: As Property Specification Language

- fl with BDDs started to look like a quite useful specification language.
- To make this even better, the language was extended to allow conditionals to be symbolic.
- Mechanisms for making it easy to create simple Domain Specific Languages for specification were also added

Example of Symbolic Conditional



Even in Control Structures...



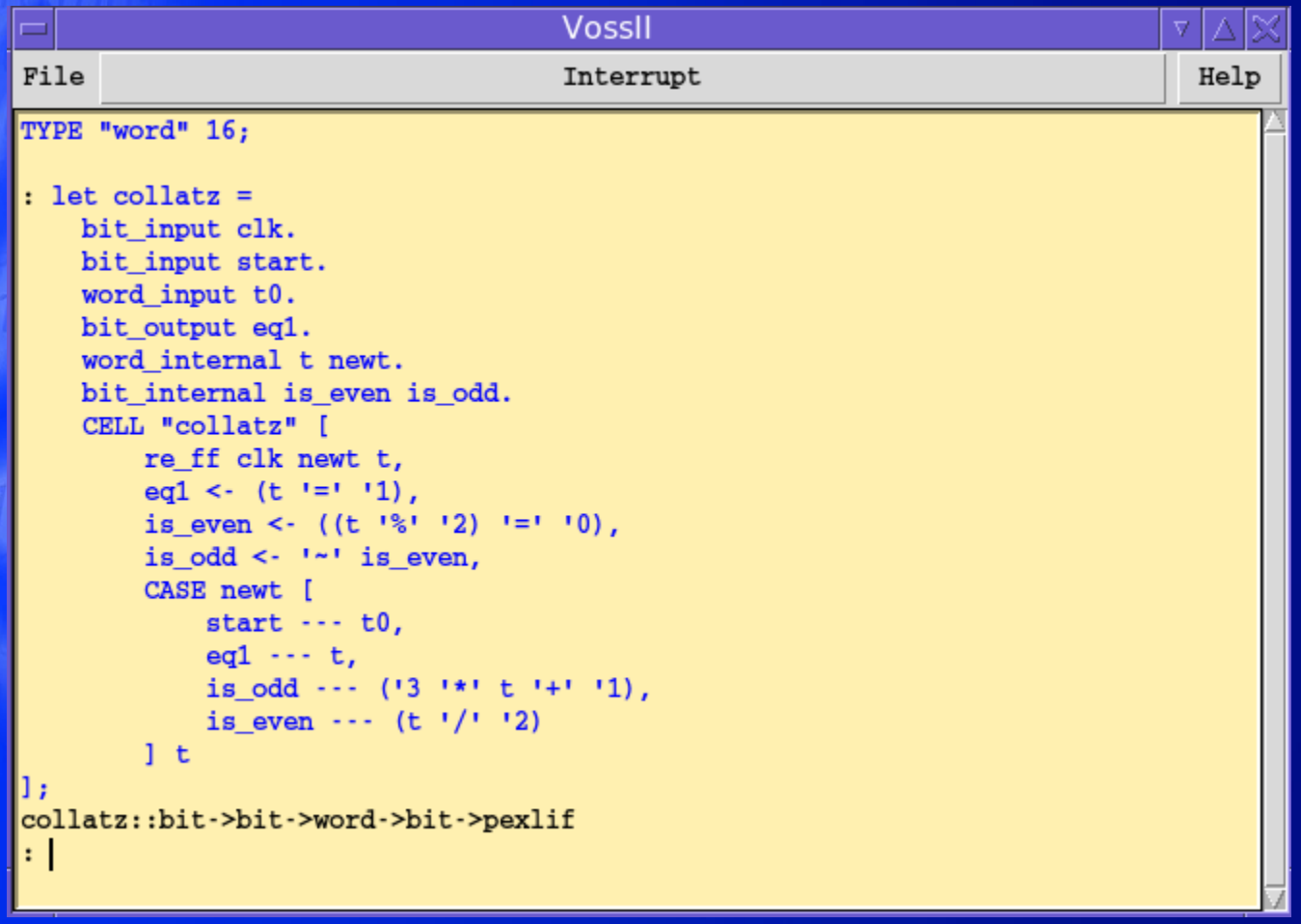
Phase 3: As Term Language in a Theorem Prover

- HOL-Voss (separate theorem prover and model checker):
  - HOL provided TP, fl provided model checking capabilities
  - Very difficult to use, common case slow, overkill
- VossProver (deep embedding of logic in fl)
  - Easier to use, but still extra layer of interpretation
  - Very cumbersome to extend
- Reflection
  - Introduced reflection in fl so that fl programs can manipulate other fl programs.
  - No overhead for end user, trivial to extend, some “noise” in the theorem proving from fl (e.g., print statements etc.)

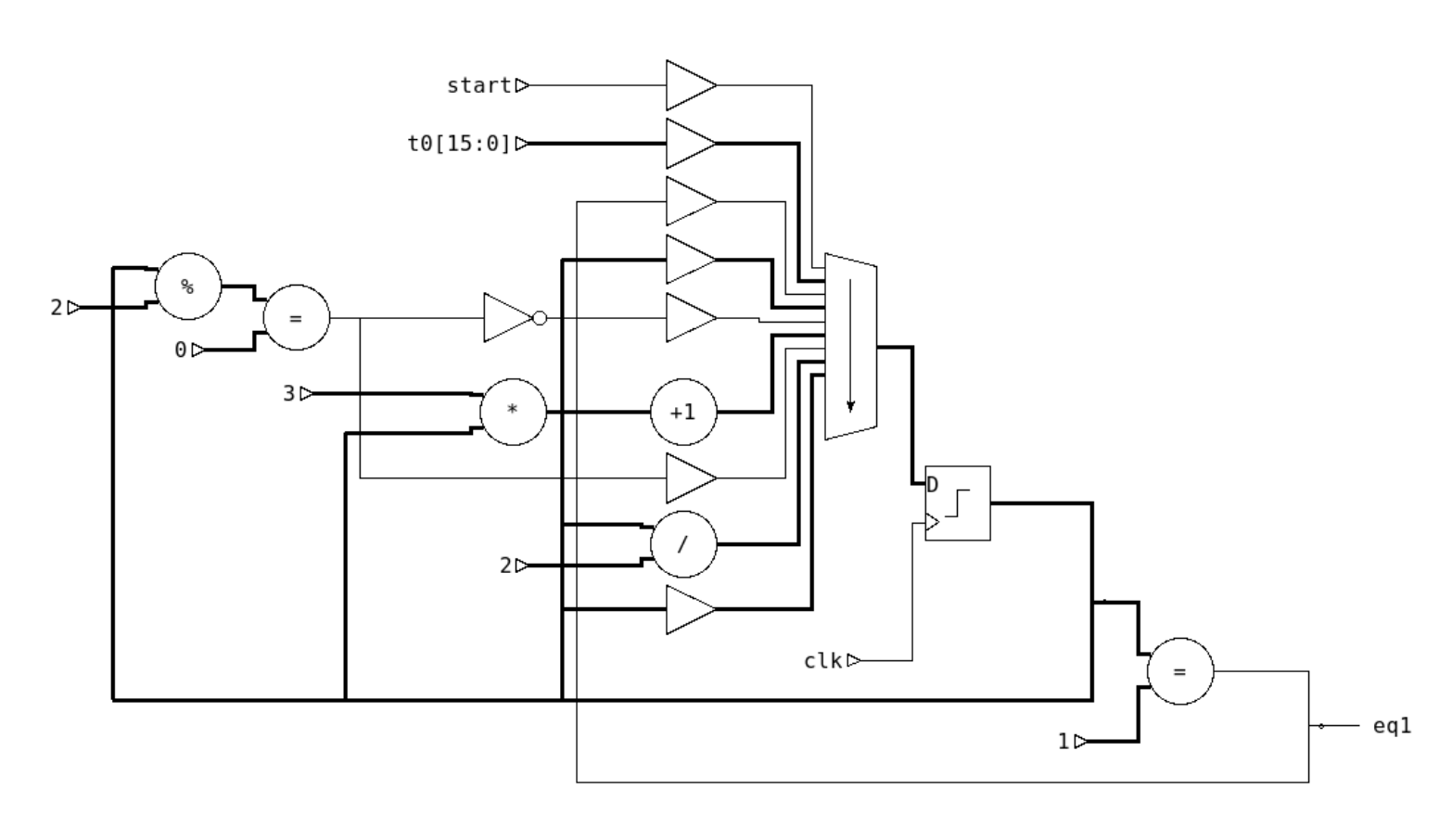
Phase 4: As Hardware Modeling Language (HFL)

- Concise and very general.
- Strong type checking without much overhead
- Easily extensible
- Tightly integrated with the formal verification engine
  - Allows: “Integrated Design and Verification”

Example of HFL



Example of HFL cont.



Circuit to evaluate the Collatz conjecture.

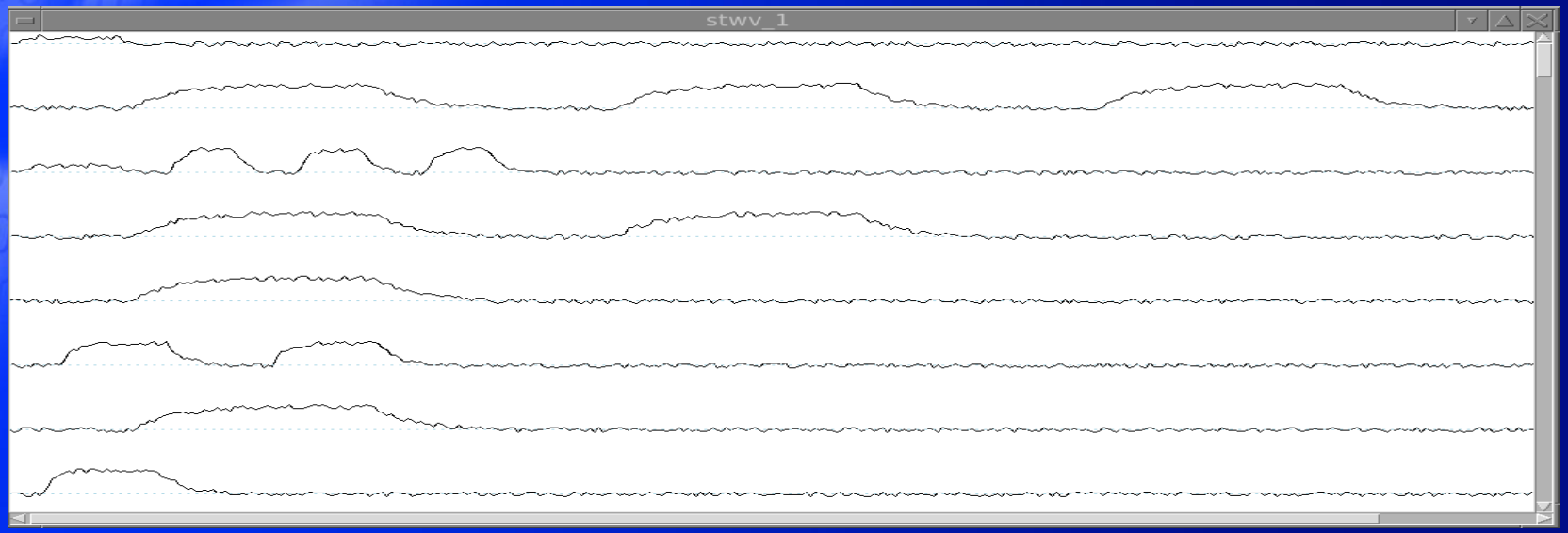
Deep Dive in Using fl

Example 1:

Symbolic Time Series Specification

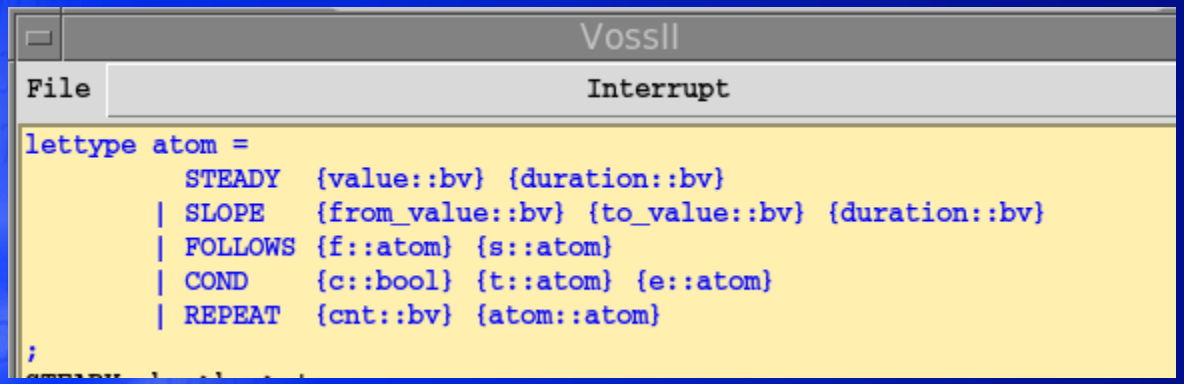
Challenge

- Suppose you have been tasked to design a circuit that watches a (noisy) input signal and that needs to recognize certain patterns.
  - For example, if the input could look like:



- How do you write a spec?

DSL For Time Series: Datatype



- Note use of bv (bitvectors) rather than ints.
  - A bv is a dynamically growing list of bools used to represent a 2's complement number.
  - Symbolic if-then-else automatically adjust the size of a bv so that the "then" and "else" bvs are structurally equal.

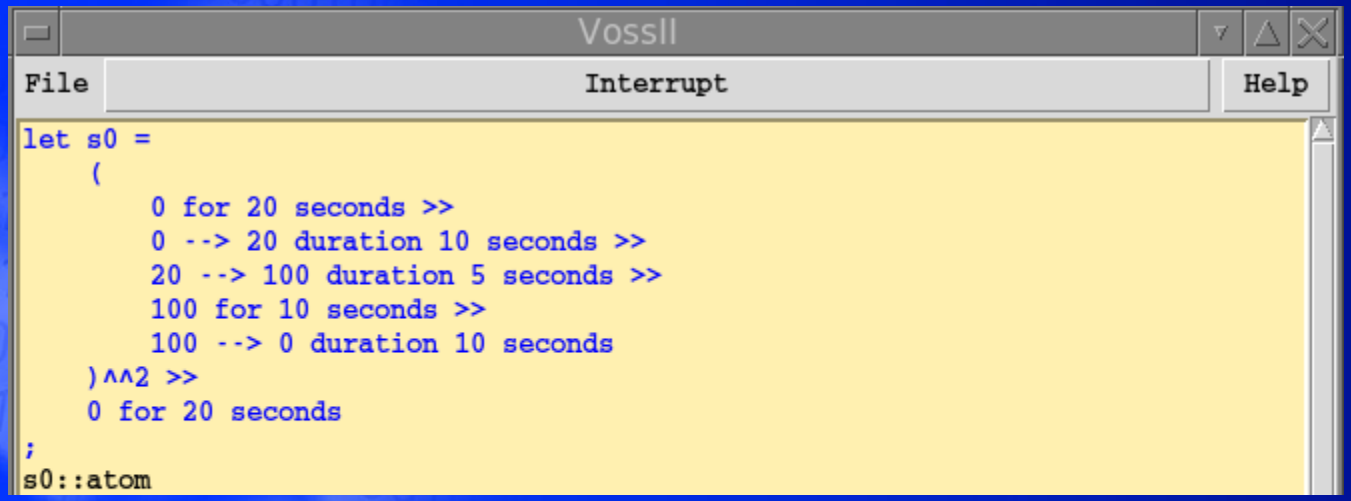
DSL Functions

```
let to_ii from_v to_v = (int2bv from_v, int2bv to_v);
let to_ib from_v to_v = (int2bv from_v, {to_v::bv});
let to_bi from_v to_v = ({from_v::bv}, int2bv to_v);
let to_bb from_v to_v = ({from_v::bv}, {to_v::bv});
overload --> to_ii to_ib to_bi to_bb;
infix 5 -->;


let steady_ii value time = STEADY (int2bv value) (int2bv time);
let steady_bi value time = STEADY value (int2bv time);
let steady_ib value time = STEADY (int2bv value) time;
let steady_bb value time = STEADY value time;
overload for steady_ii steady_bi steady_ib steady_bb;
infix 4 for;

let slope_i (fv,tv) time = SLOPE fv tv (int2bv time);
let slope_b (fv,tv) time = SLOPE fv tv time;
overload duration slope_i slope_b;
infix 4 duration;

let >> l1 l2 = FOLLOWS l1 l2;
infix 3 >>;


let repeat_i al cnt = REPEAT (int2bv cnt) al;
let repeat_b al cnt = REPEAT cnt al;
overload ^^ repeat_i repeat_b;
infix 6 ^^;

let second t = t;
postfix second;

let seconds t = t;
postfix seconds;

let If c = c;
let Else t e = (t,e);
let Then c (t,e) = COND c t e;
if_then_else_binder Then Else;
```

Extensive use of overloading

Extensive use of fixity declarations

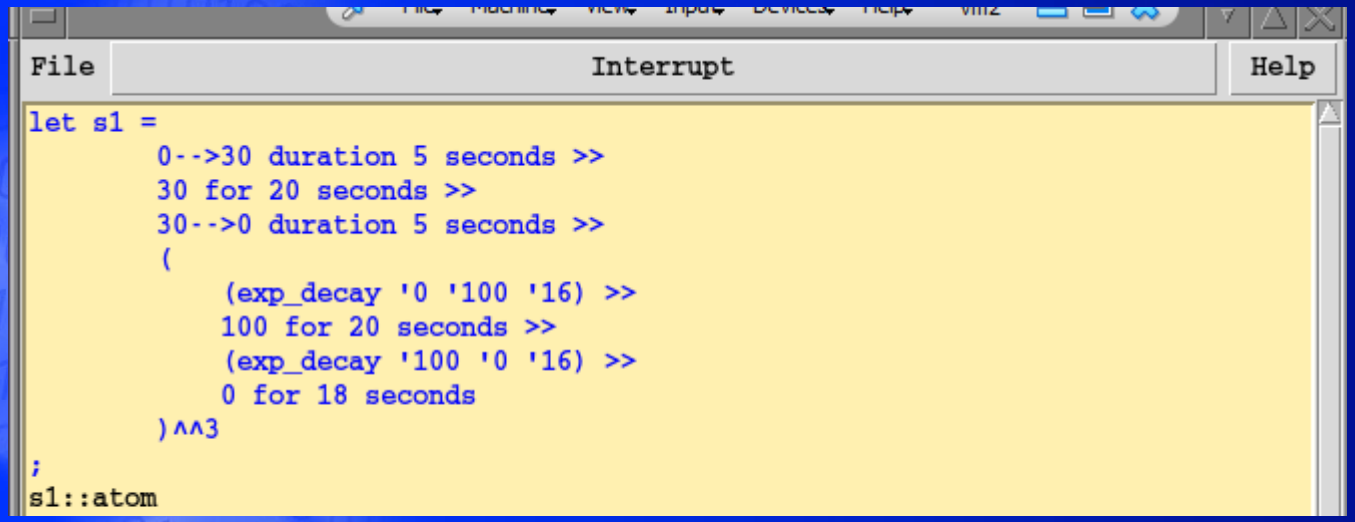
Simple Example of Time Series



User Extensible Language....

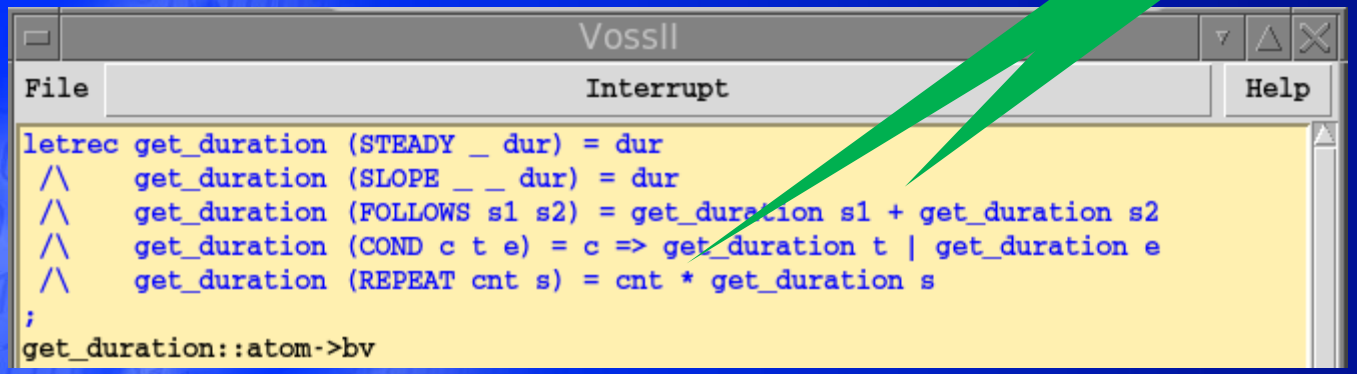
```
let decay_fn n d from_v to_v =
    let ratio = (int2float n)/(int2float d) in
    let percent = int2bv (round (100.0 * (1.0 - exp (-3.0*ratio)))) then
    from_v + ((to_v - from_v)*percent)/int2bv 100
;

let exp_decay {from_v::bv} {to_v::bv} {time::bv} =
    If(time <= '1) Then (SLOPE from_v to_v time) Else (
    If(time <= '2) Then (
            let mid_v = decay_fn 1 2 from_v to_v in
            (SLOPE from_v mid_v '1) >>
            (SLOPE mid_v to_v '1)
    ) Else (
    If(time <= '3) Then (
            let m13 = decay_fn 1 3 from_v to_v in
            let m23 = decay_fn 2 3 from_v to_v in
            (SLOPE from_v m13 '1) >>
            (SLOPE m13 m23 '1) >>
            (SLOPE m23 to_v '1)
    ) Else (
     ...
    ) Else (
            let m18 = decay_fn 1 8 from_v to_v in
            let m28 = decay_fn 2 8 from_v to_v in
            let m38 = decay_fn 3 8 from_v to_v in
            let m48 = decay_fn 4 8 from_v to_v in
            let m58 = decay_fn 5 8 from_v to_v in
            let m68 = decay_fn 6 8 from_v to_v in
            let m78 = decay_fn 7 8 from_v to_v in
            let dur i =
                (int2bv i)*time/int2bv 8 -
                (int2bv (i-1))*time/int2bv 8
            in
            (SLOPE from_v m18 (dur 1)) >>
            (SLOPE m18 m28  (dur 2)) >>
            (SLOPE m28 m38  (dur 3)) >>
            (SLOPE m38 m48  (dur 4)) >>
            (SLOPE m48 m58  (dur 5)) >>
            (SLOPE m58 m68  (dur 6)) >>
            (SLOPE m68 m78  (dur 7)) >>
            (SLOPE m78 to_v (dur 8))
    )))))))
;
```

More Complex Example of TS



DSL to Sequence

Extensive use of overloading

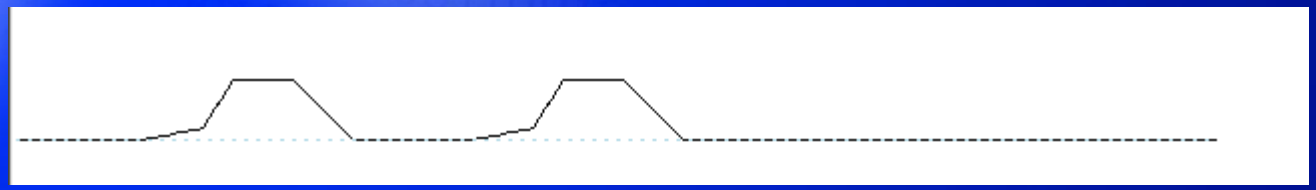


Simple Example of Time Series

VossII

File | Interrupt | Help

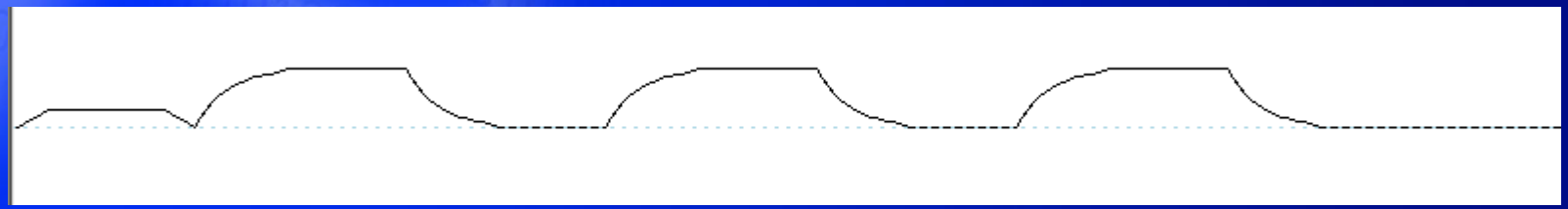
```
let s0 =
    (
        0 for 20 seconds >>
        0 --> 20 duration 10 seconds >>
        20 --> 100 duration 5 seconds >>
        100 for 10 seconds >>
        100 --> 0 duration 10 seconds
    )^^2 >>
    0 for 20 seconds
;
s0::atom
: let s0_flat = (atom2sequence 200 s0);
s0_flat::bv list
: let v0 = visualize_sequences (get_sequence_examples 40 s0_flat);
v0::void
: v0;
:
```



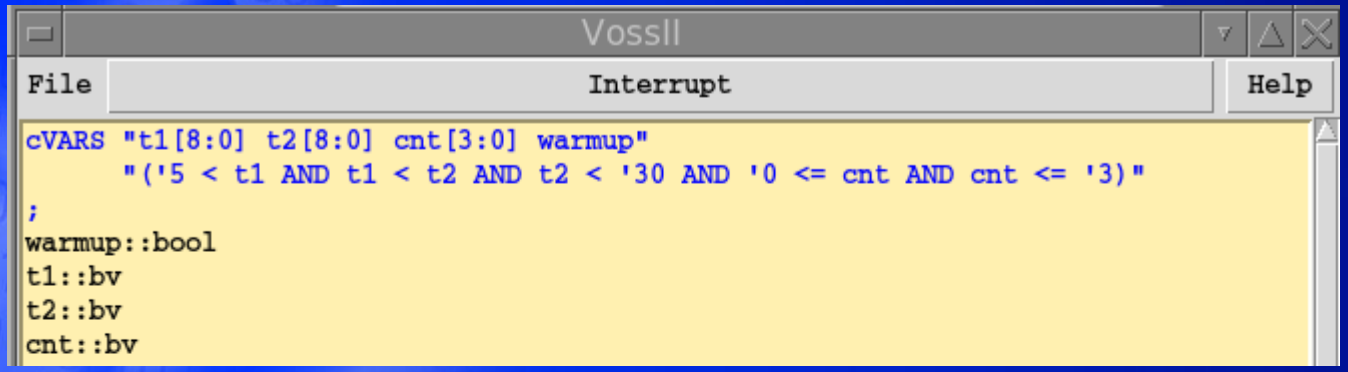
More Complex Example of TS

File | Interrupt | Help

```
let s1 =
        0-->30 duration 5 seconds >>
        30 for 20 seconds >>
        30-->0 duration 5 seconds >>
        (
            (exp_decay '0 '100 '16) >>
            100 for 20 seconds >>
            (exp_decay '100 '0 '16) >>
            0 for 18 seconds
        )^^3
;
s1::atom
: let s1_flat = (atom2sequence 400 s1);
s1_flat::bv list
: visualize_sequences (get_sequence_examples 40 s1_flat);
:
```



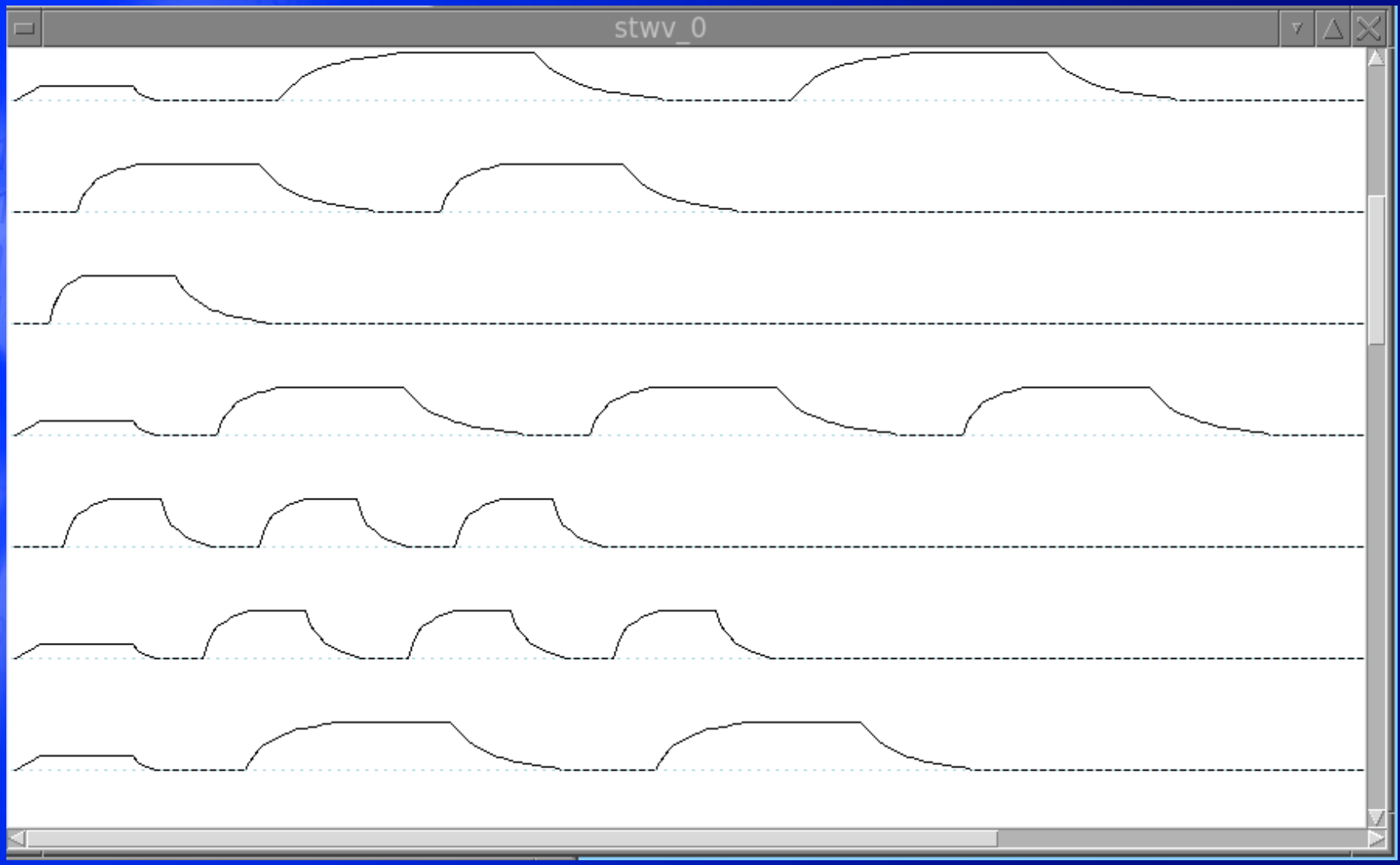
Example of Symbolic Spec.



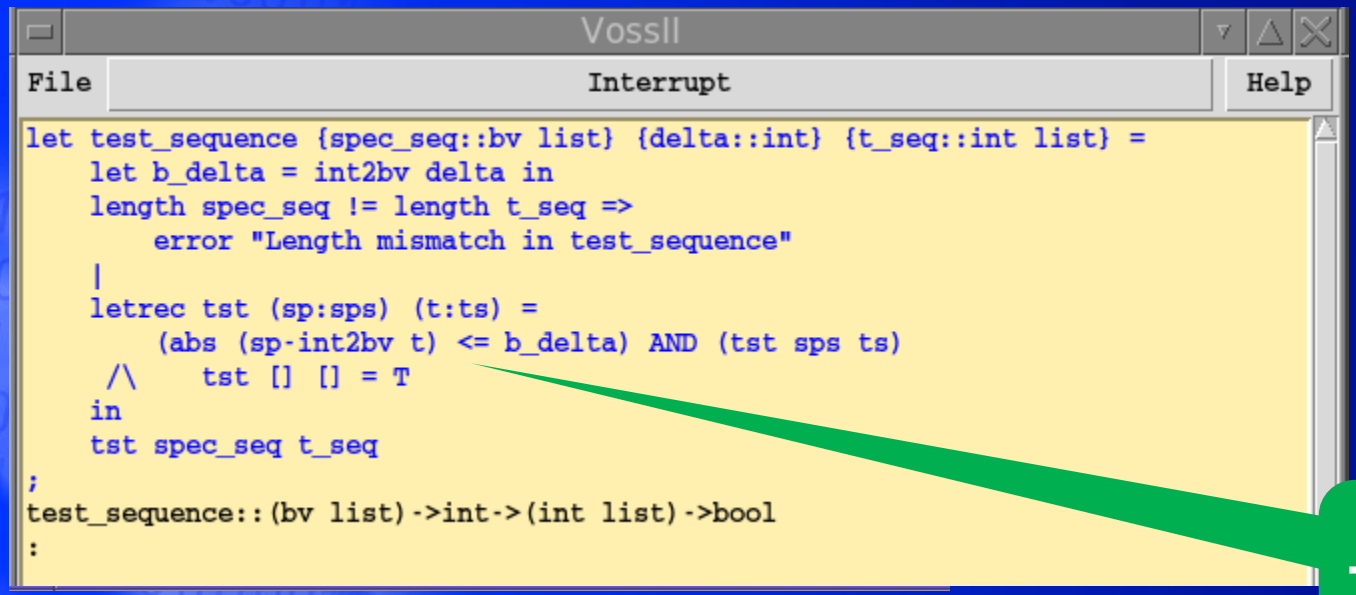
Symbolic Spec. Characteristics

```
: let s2_flat = (atom2sequence 400 s2);
s2_flat::bv list
: time (length s2_flat);
(400, "134.8")
it::int#string
: bdd_size s2_flat;
36579
it::int
: 2**(length (depends s2_flat));
8192
it::int
```

Enumerated Examples from Spec.



Test Within Specification?



Tolerance

Characterize a Collection of TSs.

```
: let covered = itlist (\ts. \r. r OR (test_sequence s2_flat 10 ts)) tests F;
covered::bool
: bdd_size covered;
149
it::int
: forcing covered;
[("cnt[1]_n", F)]
it::string#bool list
: cnt;
<F,cnt[1]_n,cnt[0]_n>
it::bv
```

- In other words: There are no time series in our tests that has the pattern repeated 2 or 3 times.

Why a Symbolic Spec?

- Use it as spec. to verify an implementation.
- Check if explicit test satisfies the spec.
- Characterize a collection of tests.
- Generate “interesting” tests for corner cases.
- Use it to create labels for supervised machine learning.
- Augment existing training data for supervised machine learning.
- …

Summary

“The Good”

- FI has ended up serving as:
  - Property specification language
  - Implementation language for FPV & FEV tools
  - Scripting language for the end-user
  - Term language for theorem proving
  - Modeling language for hardware
  - Environment for developing symbolic algorithms
- And it is quite good at all of them!

“The Bad”

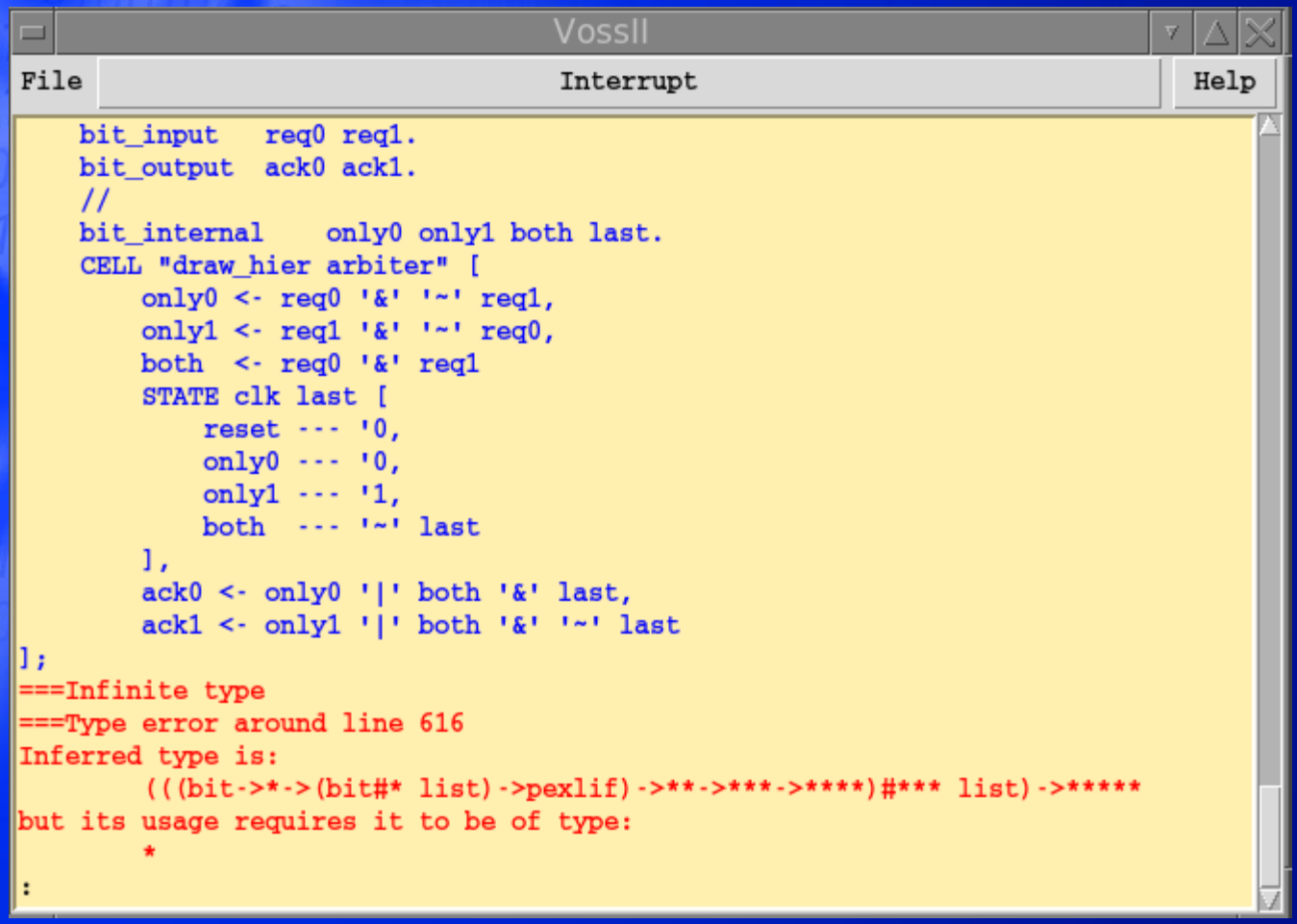
- Execution speed and memory footprint is a serious issue when dealing with large (LARGE) designs.
- Many functions have migrated into C inside VossII to provide sufficient performance
  - Correctness issue
  - Flexibility issue
- The mixture of fl and tcl/tk code for GUI is a source of much headache.
  - Visualization is a requirement, but difficult to do cleanly in functional language.

The “Ugly”

- Embedded Domain Specific Languages always end up with some quirky syntax to enable embedding in host language.
  - FI provides a number of novel fixity and binders to minimize this, but not removing it entirely
- Type errors can be intimidating….

“The Ugly”

- A missing comma yields:



“The Ugly”

- Unused (not explicitly typed) declaration makes type checker fail with very little explanation….

VossII

File | Interrupt | Help

```
),'(3)] bit->bit->bit->bit->bit->bit->pexlif
: let arbiter =
    bit_input   clk reset.
    bit_input   req0 req1.
    bit_output  ack0 ack1.
    //
    bit_internal    only0 only1 both last.
    internal        foo.
    CELL "draw_hier arbiter" [
        only0 <- req0 '&' '~' req1,
        only1 <- req1 '&' '~' req0,
        both  <- req0 '&' req1,
        STATE clk last [
            reset --- '0,
            only0 --- '0,
            only1 --- '1,
            both  --- '~' last
        ],
        ack0 <- only0 '|' both '&' last,
        ack1 <- only1 '|' both '&' '~' last
];
arbiter::[hw_values(12),hw_constr(34),hw_mk_var(17),hw_size(15),hw_destr(77),+(5
),'(3)] bit->bit->bit->bit->bit->bit->pexlif
```

Lessons Learned

Why is it Successful?

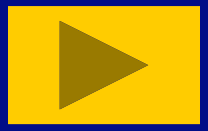
- Voss/fl provides a unified environment that makes it easy to build, extend, and use formal verification methods.
- There is a natural fit in the semantic model for specifications (functional)
- The performance of the interpreter is not on the critical path for most applications
- The system is easily and safely extensible by the (experienced) user.
- Voss/fl provides a major new capability!
  - The cost of “swallowing” fl is paid back by the new capabilities.

Announcement:

- VossII is (as of a week) open source (Apache 2.0)!
- See: https://github.com/TeamVoss/VossII
- There are prebuilt binaries for Linux if you just want to try it out. See the README on github repository for more details.
- If this presentation has made you curious, download it and do something wonderful with it
  - But please tell me what you did ☺

Questions

Backup Slides



Demo of Using fl for Hardware Design

Larger Example

```
: ENUM "four_phase" ["IDLE", "REQ", "ACK", "DONE"];
let rd_protocol_ifc =
    bit_input            clk.
    bit_input            start.
    bit_input            ack.
    input                din.
    bit_output           req.
    bit_output           done.
    output               dout.
    four_phase_internal  state.
    bit_internal         reset.
    CELL "draw_hier rd_ifc_fsm" [
        reset <- '~' start,
        req <- is_REQ state,
        Moore_FSM "ifc" clk state (reset --- IDLE) [
                IDLE    --- start    --- REQ,
                REQ     --- ack      --- ACK,
                ACK     --- '~' ack --- DONE
        ],
        re_ff_en clk (is_REQ state '&' ack) din dout,
        done <- is_DONE state
];
```

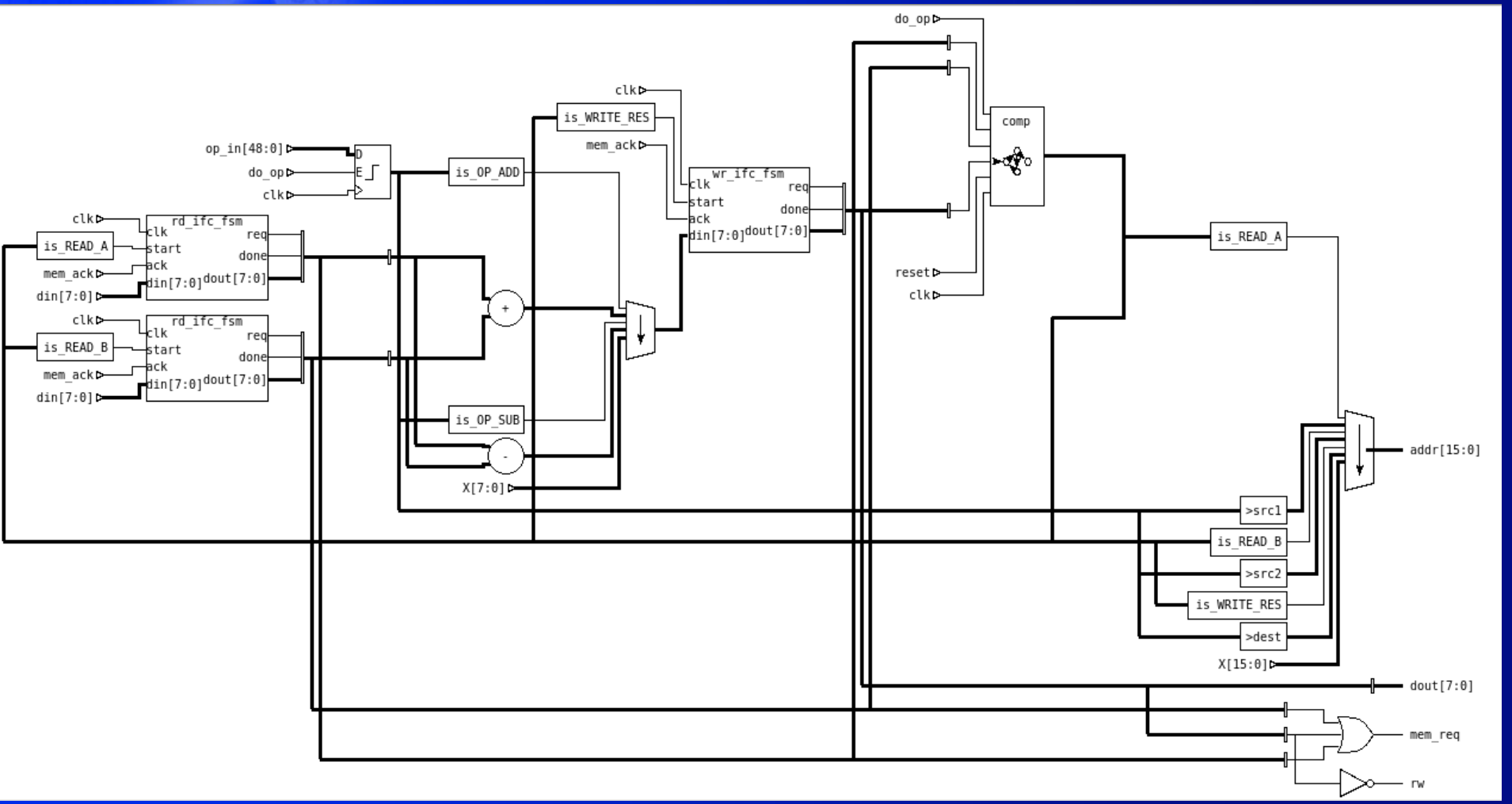
Larger Example

```
ENUM "process" ["IDLE", "READ_A", "READ_B", "COMP", "WRITE_RES"];
TYPE "byte" 8;
TYPE "data" 8;
TYPE "addr" 16;
ENUM "opcode" ["OP_ADD", "OP_SUB"];
STRUCT "op" [
        ("opcode",  "opcode"),
        ("dest",    "addr"),
        ("src1",    "addr"),
        ("src2",    "addr")
];
```

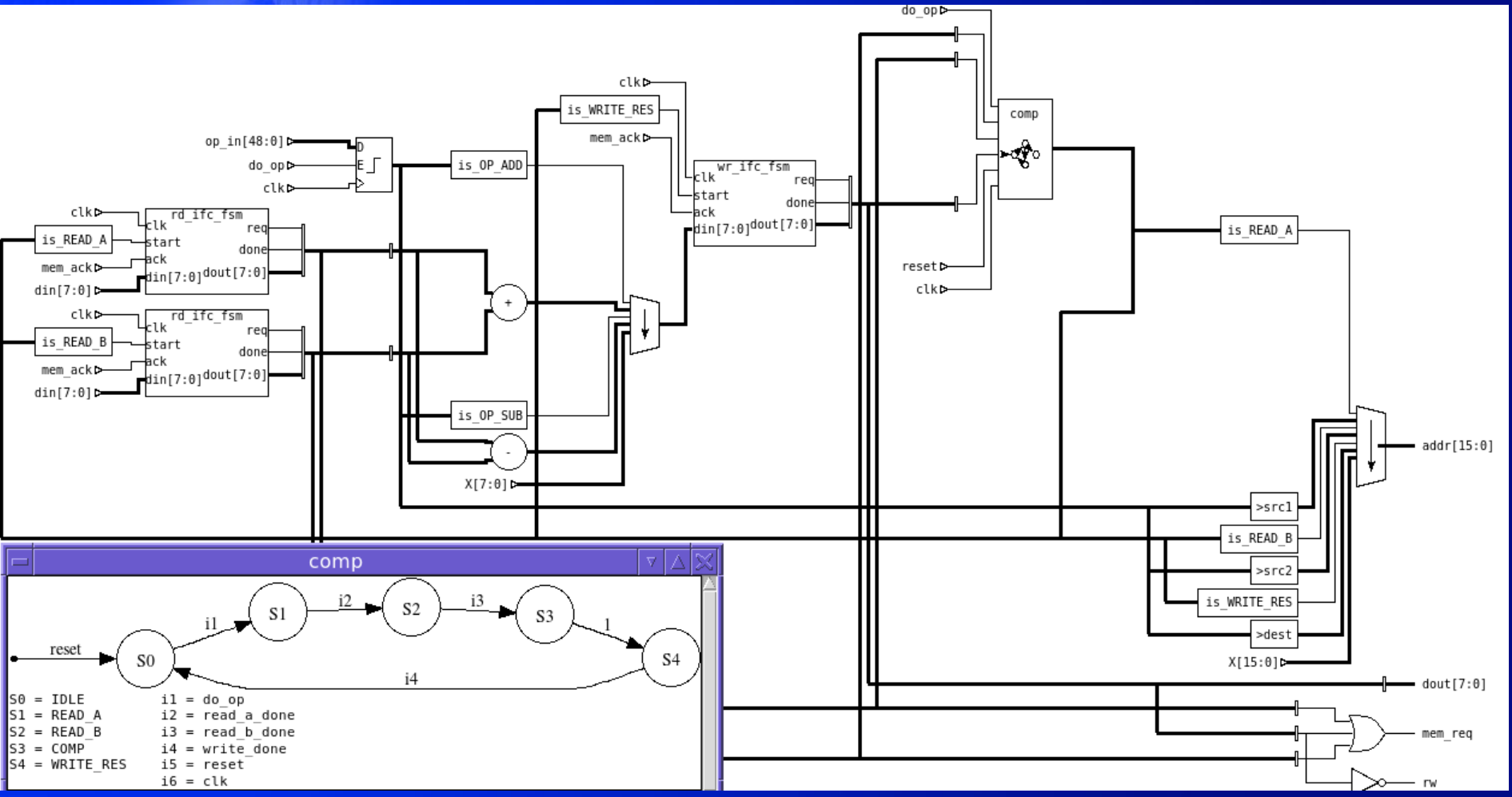
Larger Example

```
let test =
    bit_input           clk reset do_op.
    op_input            op_in.
    addr_output         addr.
    data_input          din.
    data_output         dout.
    bit_output          mem_req rw.
    bit_input           mem_ack.
    //
    internal    r1 r2 r top_fsm op.
    internal    read_a_done read_b_done write_done rdA_req rdB_req wr_req.
    CELL "test" [
    re_ff_en clk do_op op_in op,
    Moore_FSM "comp" clk top_fsm (reset --- IDLE) [
        IDLE    --- do_op ---           READ_A,
        READ_A  --- read_a_done ---     READ_B,
        READ_B  --- read_b_done ---     COMP,
        COMP    --- default ---         WRITE_RES,
        WRITE_RES --- write_done ---    IDLE
    ],
    rd_protocol_ifc clk (is_READ_A top_fsm) mem_ack din rdA_req read_a_done r1,
    rd_protocol_ifc clk (is_READ_B top_fsm) mem_ack din rdB_req read_b_done r2,
    wr_protocol_ifc clk (is_WRITE_RES top_fsm) mem_ack r wr_req write_done dout,
    mem_req <- rdA_req '|' rdB_req '|' wr_req,
    CASE addr [
        is_READ_A top_fsm --- op-->src1,
        is_READ_B top_fsm --- op-->src2,
        is_WRITE_RES top_fsm --- op-->dest
        ] 'X,
    rw <- '~' wr_req,
    CASE r [
        is_OP_ADD (op-->opcode) --- (r1 '+' r2),
        is_OP_SUB (op-->opcode) --- (r1 '-' r2)
        ] 'X
];
```

Larger Example



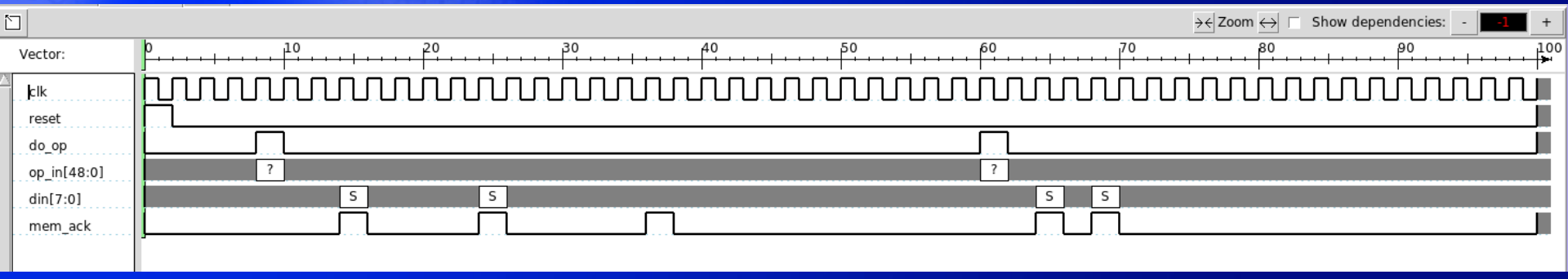
Larger Example



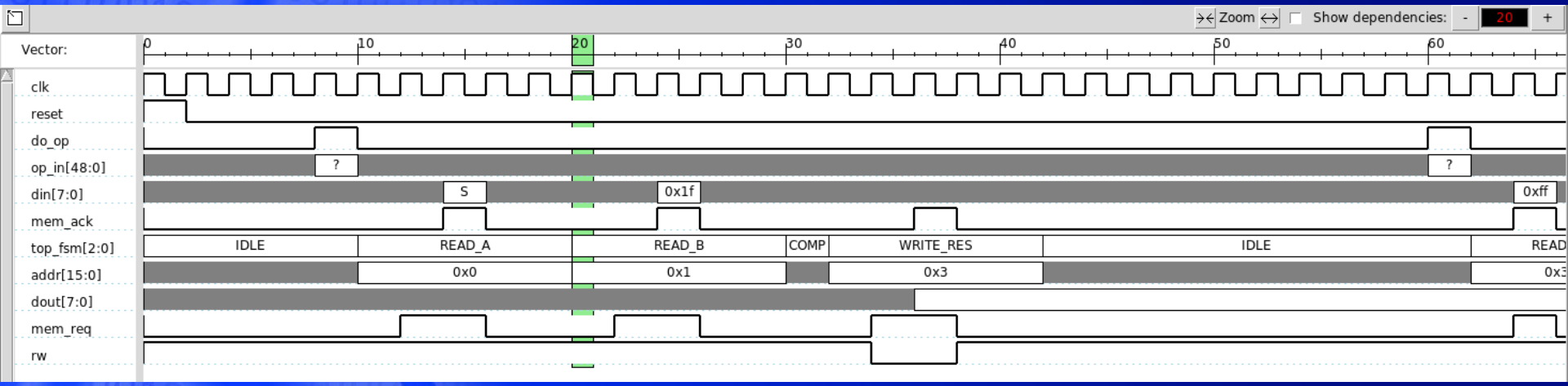
Simulation Stimuli Creation

```
: let ant =
    "clk" is_clock 50
  and
    "reset" is "1" in_cycle 0 followed_by "0" for 49 cycles
  and
    "do_op" is "1" in_cycle 4 followed_by
               "1" in_cycle 30
              otherwise "0" until 50 cycles
  and
    "op_in[48:0]" is (mk_op OP_ADD '3 '0 '1) in_cycle 4 followed_by
                     (mk_op OP_SUB '0 '3 '1) in_cycle 30
  and
    "din[7:0]" is "a[7:0]" in_cycle 7 followed_by
                  "b[7:0]" in_cycle 12 followed_by
                  "A[7:0]" in_cycle 32 followed_by
                  "B[7:0]" in_cycle 34
  and
    "mem_ack" is "1" in_cycle 7 followed_by
                 "1" in_cycle 12 followed_by
                 "1" in_cycle 18 followed_by
                 "1" in_cycle 32 followed_by
                 "1" in_cycle 34
              otherwise "0" until 50 cycles
;
```

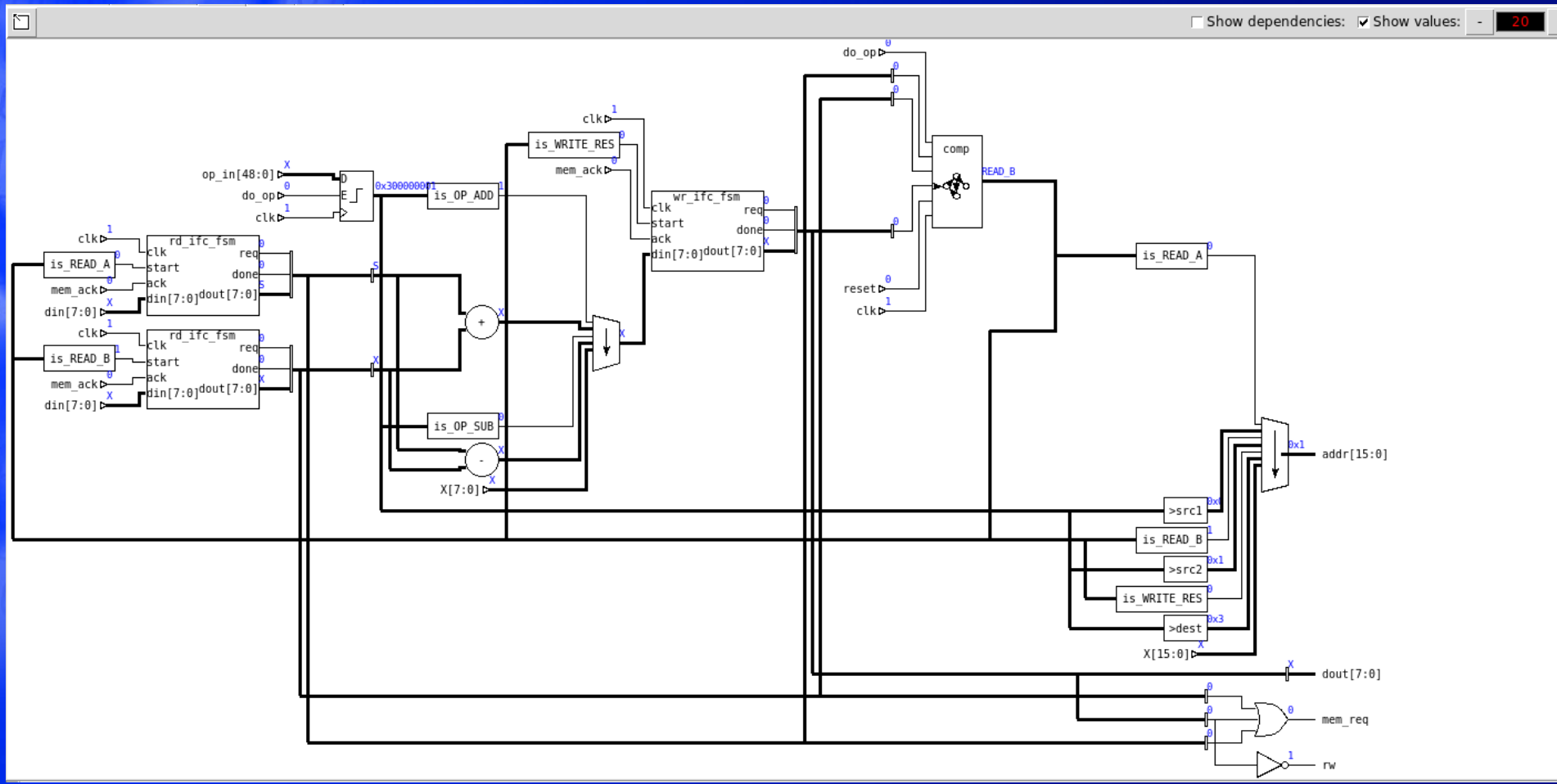
Yet another DSL!



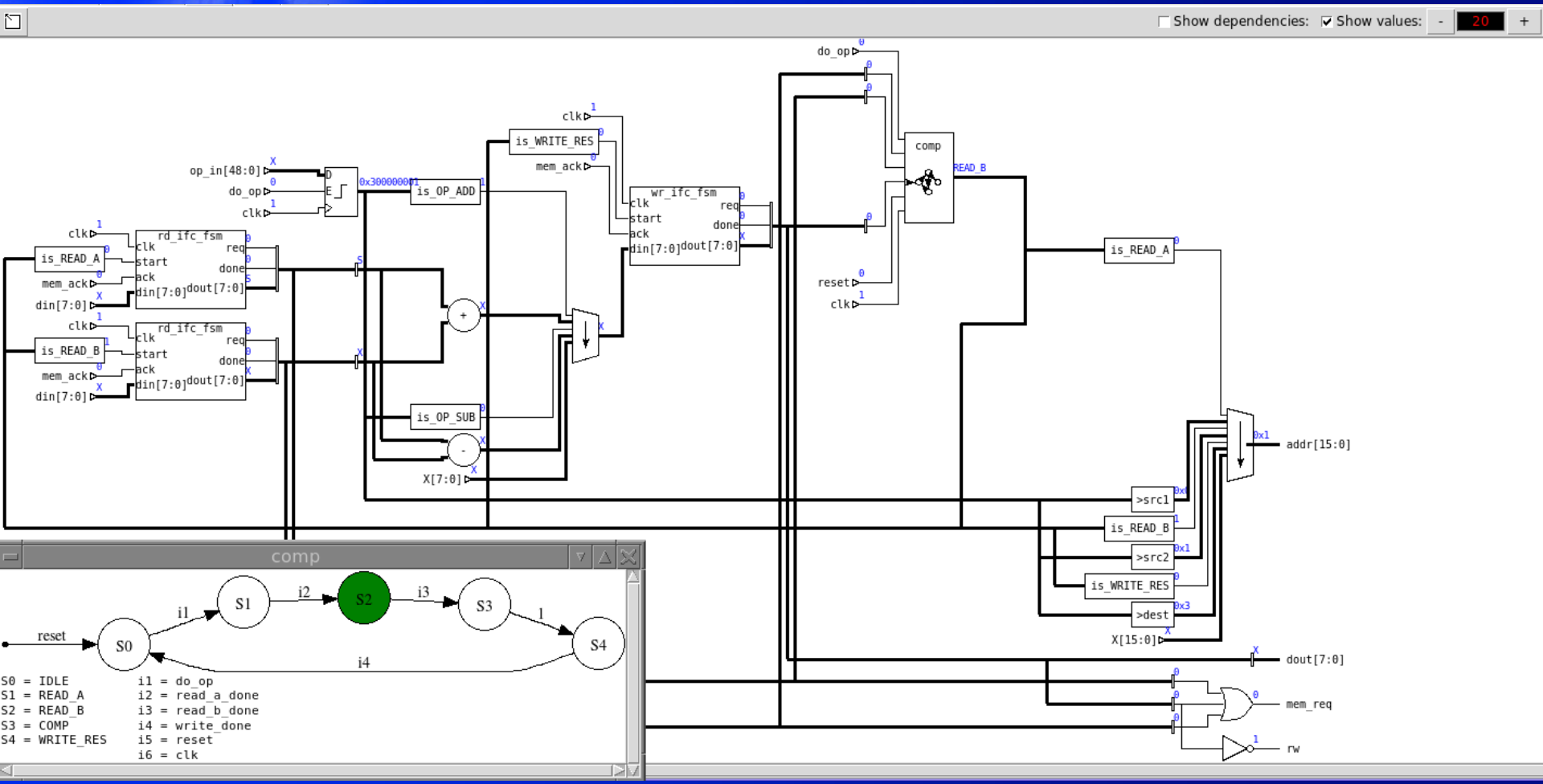
Waveform View



Circuit View



FSM Behavior View



Track Information Flow