Source-Level Debugging and Beyond

Stanford AHA Agile Hardware Project

Stanford University

Status quo of hardware generators

- Different frameworks actively developed by different research groups
  - Chisel – UC Berkeley
  - Magma – Stanford University
  - PyMTL – Cornell University

- Embedded in High-level languages
  - Chisel – Scala
  - Magma – Python
  - PyMTL – Python

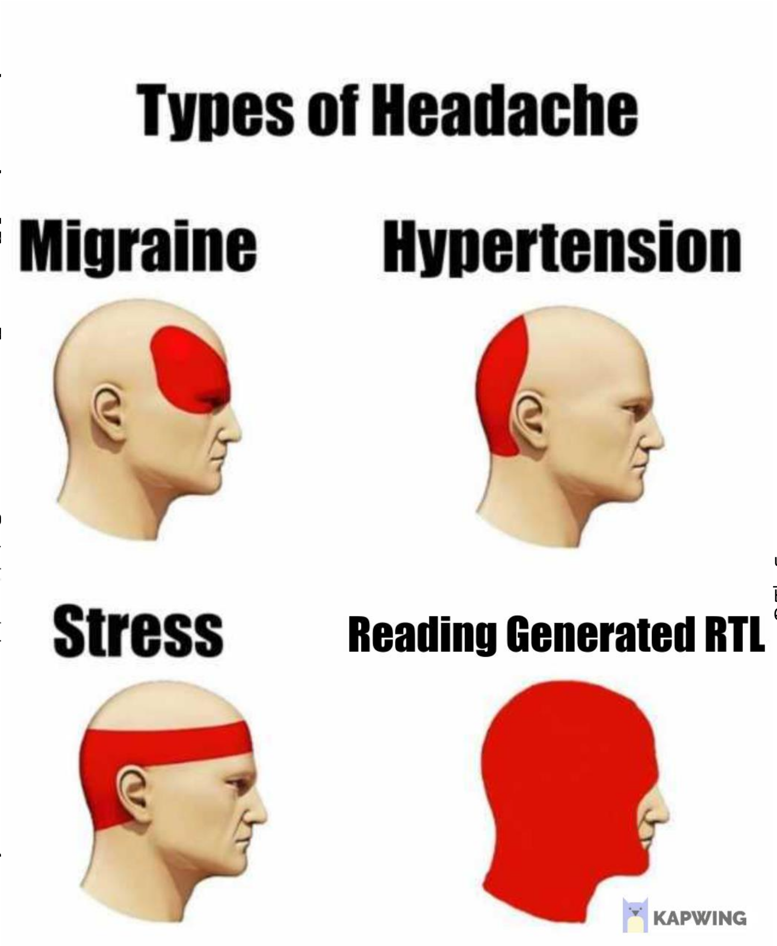
- Aggressive Compiler optimization
  - Unreadable RTL code

Example of generated RTL code

```
assign _T_2 = data[1];
assign _T_4 = data[2];
assign _T_5 = _T_4 ? 2'h2 : 2'h
assign _T_6 = data[3];
assign _T_7 = _T_6 ? 2'h3 : 2'h
assign _GEN_0 = {{1'd0}, _T_2};
assign _T_10 = _GEN_0 > _T_5;
assign _T_11 = _T_10 ? {{1'd0},
assign _T_12 = _T_11 > _T_7;
assign h_bit = _T_12 ? _T_11 :
```

Chisel

```
wire [31:0] Mux2xOutUInt32_inst0$coreir_commonlib_mux2x32_inst0
assign Mux2xOutUInt32_inst0$coreir_commonlib_mux2x32_inst0$_joi
         {bit_const_0_None_out,bit_const_0_None_out,bit_const_0
         onst_0_None_out,bit_const_0_None_out,bit_const_0_None_
         t_const_0_None_out,bit_const_0_None_out,bit_const_0_No
         ,bit_const_0_None_out,bit_const_0_None_out,bit_const_0
         [0],Mux2xOutUInt1_inst0$coreir_commonlib_mux2x1_inst0$
coreir_mux #(
    .width(32)
) Mux2xOutUInt32_inst0$coreir_commonlib_mux2x32_inst0$_join (
    .in0(Mux2xOutUInt32_inst0_I0_in),
    .in1(Mux2xOutUInt32_inst0$coreir_commonlib_mux2x32_inst0$_j
    .sel(magma_Bits_3_eq_inst1_out),
    .out(Mux2xOutUInt32_inst0$coreir_commonlib_mux2x32_inst0$_j
);
wire [31:0] Mux2xOutUInt32_inst0_I0_out;
assign Mux2xOutUInt32_inst0_I0_out = {self_regb_O1[15:0],self_r
mantle_wire__typeBitIn32 Mux2xOutUInt32_inst0_I0 (
    .in(Mux2xOutUInt32_inst0_I0_in),
    .out(Mux2xOutUInt32_inst0_I0_out)
);
```



```
ut,bit_const_0_None_out,             bit_const_0_None_out,bit_c
_const_0_None_out,bit_const_0_None_out,bit_const_0_None_out,bi
bit_const_0_None_out,bit_const_0_None_out,bit_const_0_None_out
0],Mux2xOutUInt1_inst1$coreir_commonlib_mux2x1_inst0$_join_out
```

Magma

Stanford University

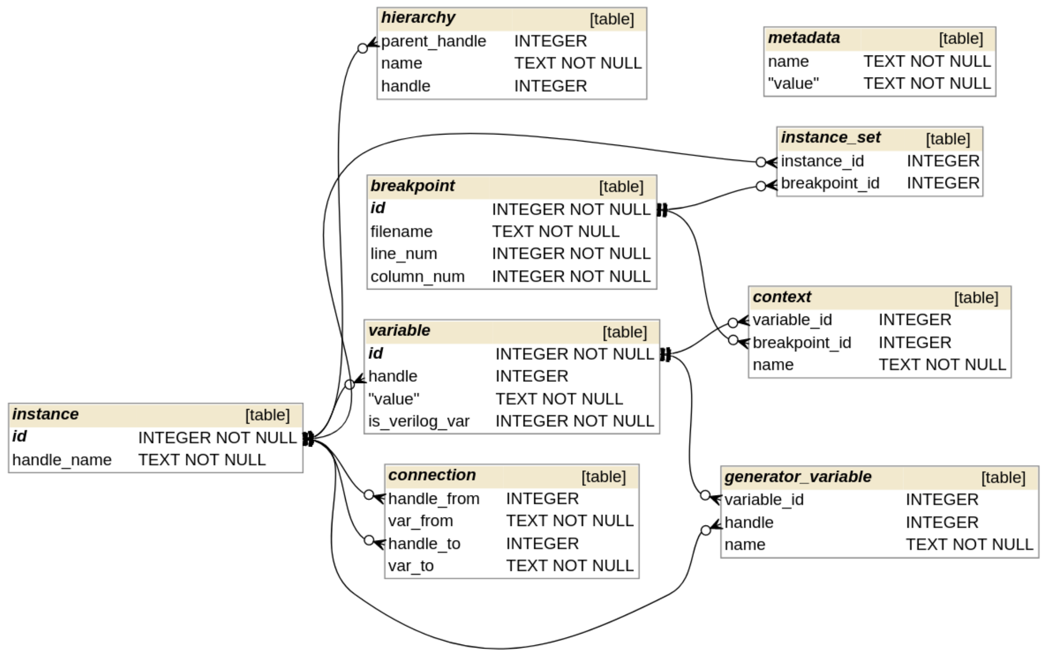
Bring source-level debugging to hardware generators

Main idea:

- Create and maintain a symbol table to map high-level constructs to their correspondence in RTL.
- Support breakpoint at source-level and frame/context reconstruction.
- Debugging utilities, e.g. REPL (read-evaluate-print-loop) in high-level languages.

What does the symbol table look like?

- Implemented in SQLite3, easy to verify and supported by virtually every programming language.
- Fast query at runtime
- Easy to prototype



hierarchy	[table]
parent_handle	INTEGER
name	TEXT NOT NULL
handle	INTEGER

metadata	[table]
name	TEXT NOT NULL
"value"	TEXT NOT NULL

instance_set	[table]
instance_id	INTEGER
breakpoint_id	INTEGER

breakpoint	[table]
id	INTEGER NOT NULL
filename	TEXT NOT NULL
line_num	INTEGER NOT NULL
column_num	INTEGER NOT NULL

context	[table]
variable_id	INTEGER
breakpoint_id	INTEGER
name	TEXT NOT NULL

variable	[table]
id	INTEGER NOT NULL
handle	INTEGER
"value"	TEXT NOT NULL
is_verilog_var	INTEGER NOT NULL

instance	[table]
id	INTEGER NOT NULL
handle_name	TEXT NOT NULL

connection	[table]
handle_from	INTEGER
var_from	TEXT NOT NULL
handle_to	INTEGER
var_to	TEXT NOT NULL

generator_variable	[table]
variable_id	INTEGER
handle	INTEGER
name	TEXT NOT NULL

Schema is undergoing rapid changes to account for different usage. This is just for illustration.

Stanford University

Breakpoint emulation: the old way

Basic idea:

- DPI function call to take over the execution of the simulation
- If the breakpoint is not inserted, return the DPI call immediately
- If inserted, halt until user wants to continue simulation

Advantages:

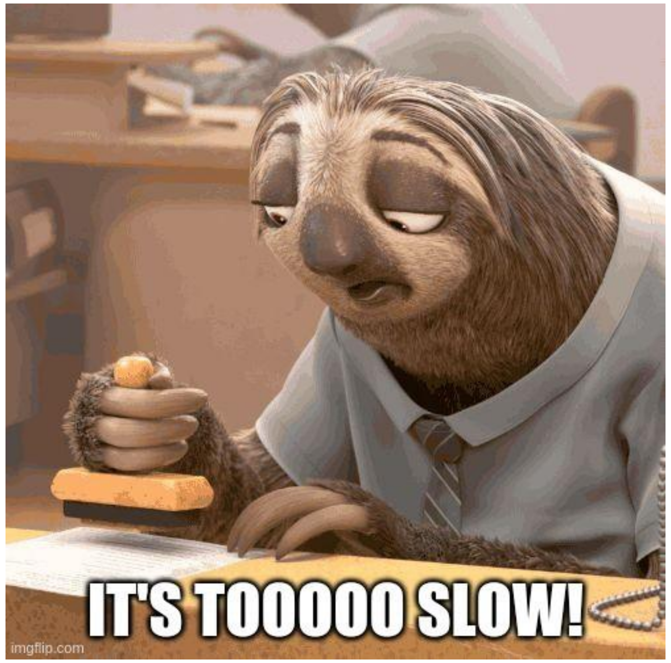
- It works with the simulators we want to support: Verilator + commercial simulators
- Easy to protype

```
always_comb begin
    breakpoint_trace (INSTANCE_ID, 32'h0);
    done = 1'h0;
    breakpoint_trace (INSTANCE_ID, 32'h1);
    h_bit = 2'h0;
    breakpoint_trace (INSTANCE_ID, 32'h2);
    if (!done) begin
        breakpoint_trace (INSTANCE_ID, 32'h6);
        if (data[0]) begin
            breakpoint_trace (INSTANCE_ID, 32'h7);
            done = 1'h1;
            breakpoint_trace (INSTANCE_ID, 32'h8);
            h_bit = 2'h0;
        end
    end
end
```

Examples and Problems with DPI-Emulation

Problems:

- Extremely inefficient
- Requires RTL code generation changes for each supported hardware generator
- Uncontrollable "break" semantics due to the difference in RTL simulation and source language



Breakpoint emulation: the new way

Key insights:

- For a synchronous design, the simulation state must stabilize @posedge of the clock.
- Static single assignment (SSA) is widely used in hardware generator frameworks and has some nice properties we can use.
- No need to keep track of line-to-line correspondence since in most cases the mapping is unidirectional.

Solution:

SSA-based breakpoint emulation

Advantages:

- More efficient. Ideally zero overhead if no breakpoints inserted
- Compiler friendly. No changes required for RTL generation
- Flexible breakpoint semantics for different source languages

Static single assignment (SSA) 101

Commonly used SSA in hardware generators example:

```
a := 0
if (b) {
    a := 2
} else {
    a := 3
}
a := 4
```

```
a_0 := 0
a_1 := 2
a_2 := 3
a_3 := b? a_1: a_2
a_4 := 4
a   := a_4
```

Outcome:

- If/switch control logic can be generated via continuous assignment (ternary) or structural mux
- Lose original code structure

Benefits:

- Dead code elimination
- Constant propagation
- Many more

Hardware "virtual" breakpoint

Each breakpoint defined in the symbol table needs

- Condition
- Trigger values

Condition is obtained through the dominance frontier set during SSA transformation.

Trigger values are used to emulate always_comb semantics if needed

Virtual breakpoint in action

Source code:

```
a := some_value
b := some_value
if (a) {         // line 3
    b := 0;      // line 4
} else {
    b := 1;      // line 6
}
c := b;          // line 7
```

RTL Code:

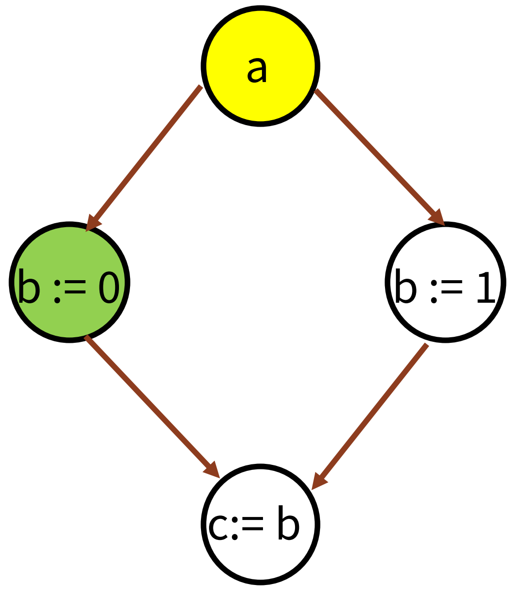
```
logic a, b, b_0, b_1, b_2, c

assign b_0 = 0;
assign b_1 = 1;
assign b_2 = a? b_0: b_1;
assign b = b_2;
assign c = b_2;
```

Compute the enable condition

Source code:

```
a := some_value
b := some_value
if (a) {          // line 3
    b := 0;       // line 4  <- target line
} else {
    b := 1;       // line 6
}
c := b;           // line 7
```



Condition for line 4: **a**

What about sequential logic?

- SSA is not applicable to sequential logic due to the design convention and non-blocking assignments.
- No worries! The enable condition is the AND of conditional logic stack!

Walk up the stack

```
if (a) {
  b <= 0                  <- a
} else {
    if (c) {              <- !a
        b <= 1            <- !a && c
    } else {
        b <= 2            <- !a && !c
    }
}
```

Stanford University

Frame/context reconstruction

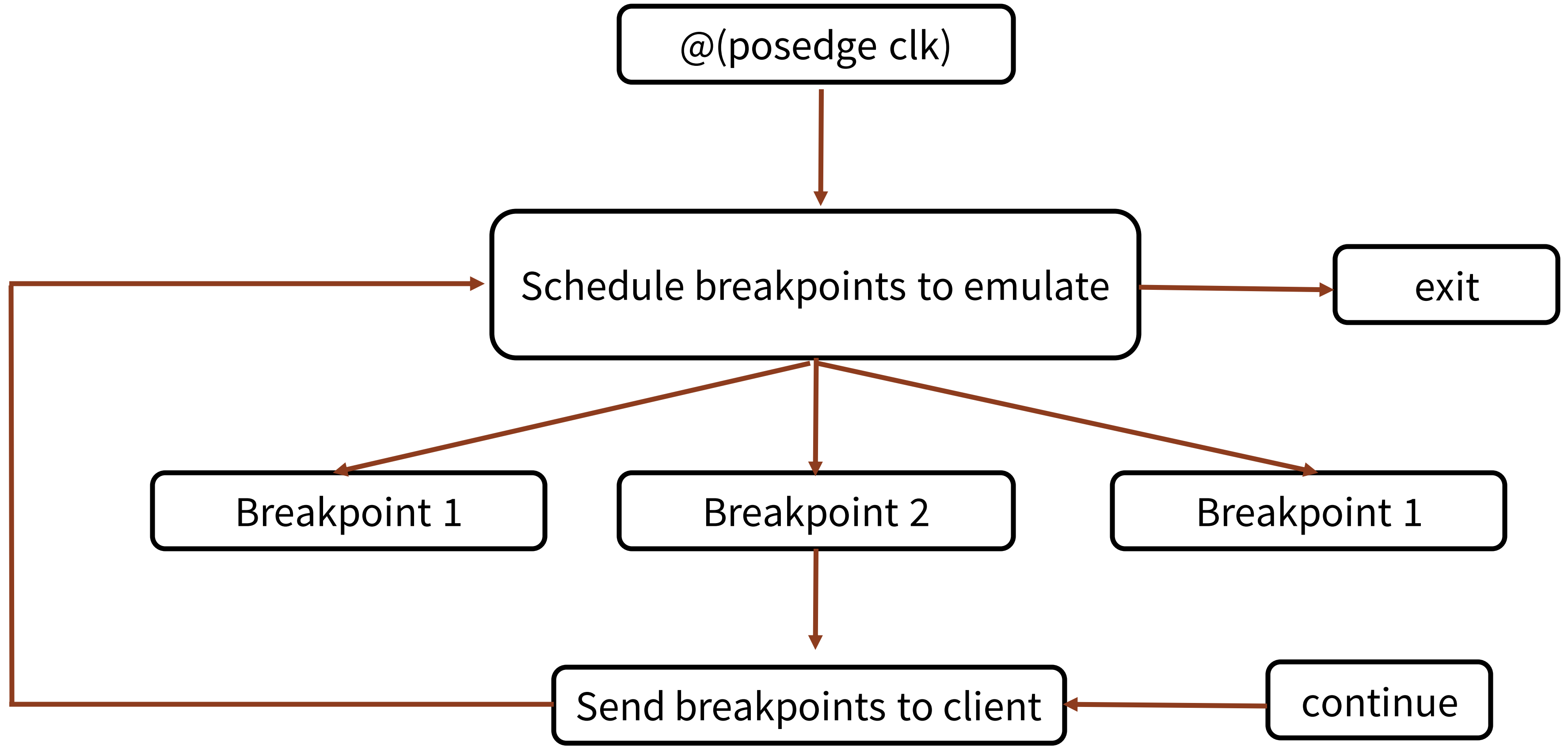
- Heavy lifting is done at the compiler side to keep track of symbols
- Query frame information once a breakpoint is hit and reconstruct the frame

```
if (a) {                // line 3
    b := 0;             // line 4
} else {
    b := 1;             // line 6
}
c := b;                 // line 7 <- frame
```

```
assign b_0 = 0;
assign b_1 = 1;
assign b_2 = a? b_0: b_1;
assign b = b_2;
assign c = b_2;
```

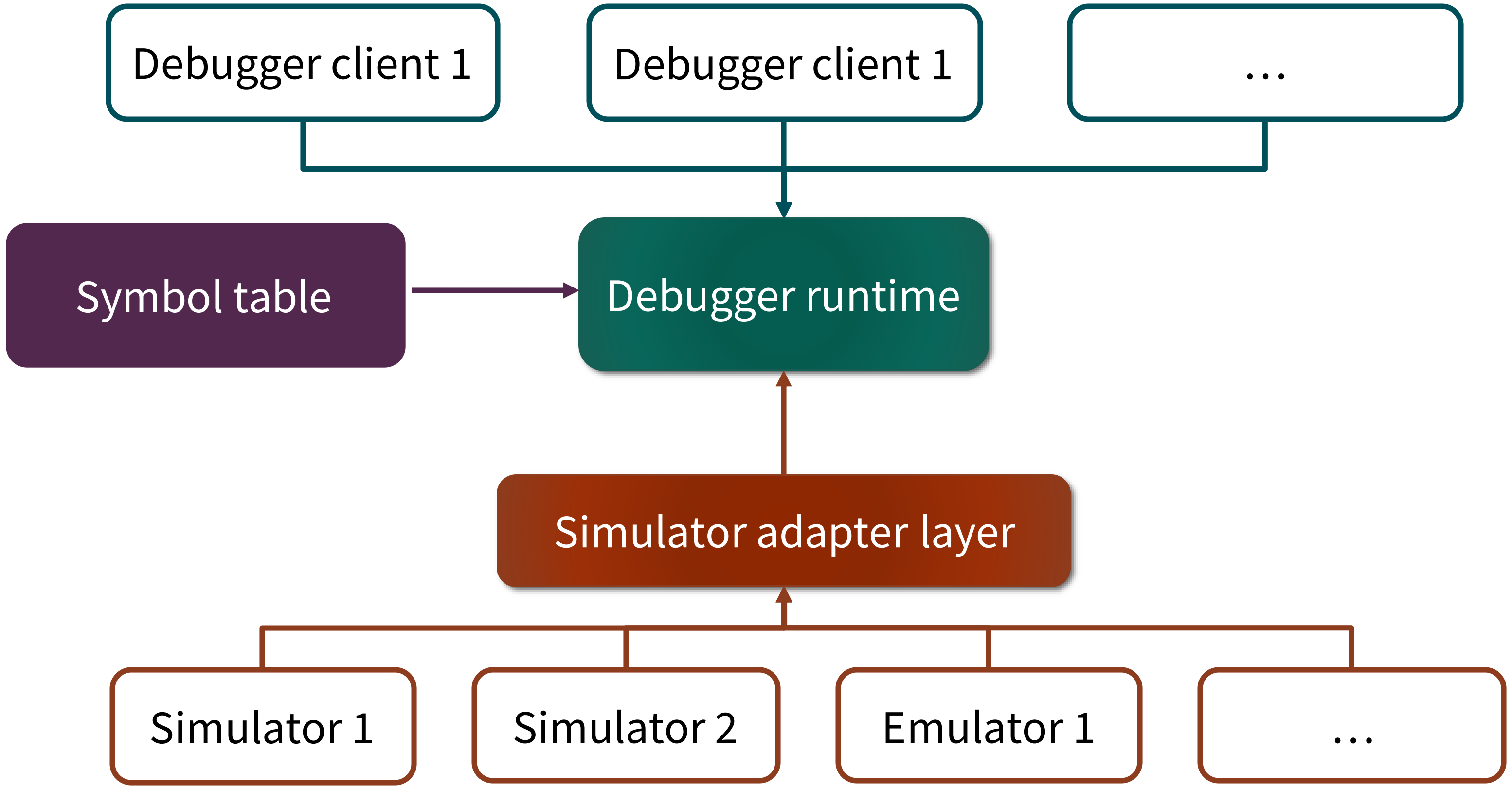
```
a -> a
b -> b_2
```
<- remap from SSA renaming!

Breakpoint emulation loop



Framework design

Tentative name: hgdb – hardware generator debugger



Waveform + Reversed Debugging

- Reversed debugging is challenging in software, e.g. rr
- We have waveform dumps in hardware!

- Without waveforms:
  - We can only reverse debug within the same timestamp
- With waveforms:
  - Rewind to any timestamp - unlimited power!

Emulation via trace

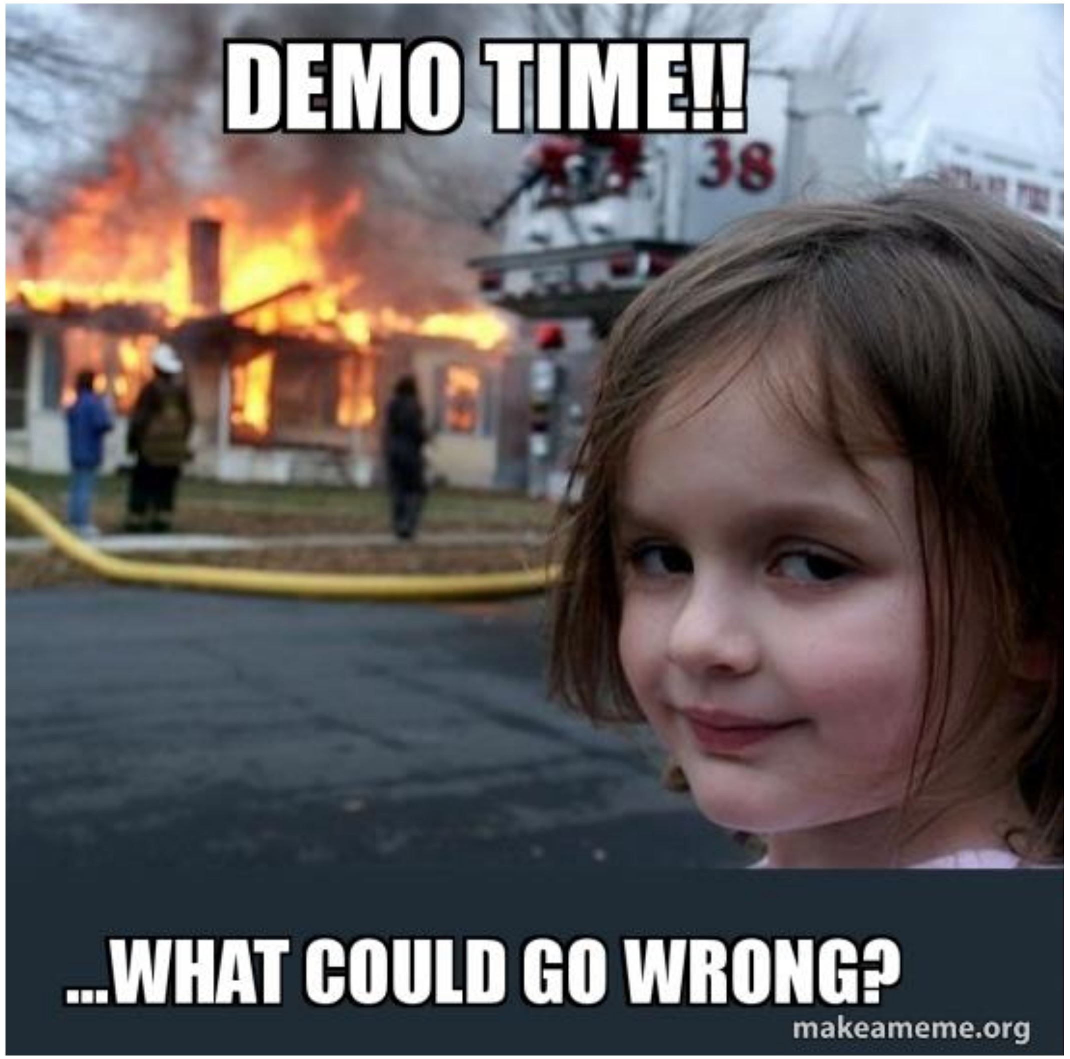
- Emulates a simulator by digesting dumped VCD info
- Implements the interface required by the adapter layer

But better:

- Since all values go through the runtime, we can reuse all the debugger tools at source-level!

Demonstration

- Simple VendingMachine in chisel (gdb-like interface)
  - Reverse debugging in VS Code
- Mini-riscv in Magma (VS Code IDE)



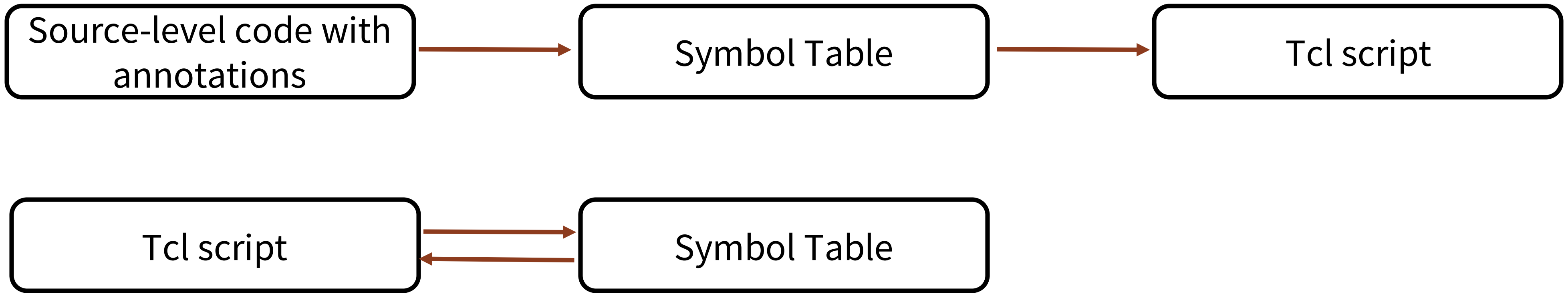
Physical design: a tale of two cities

Now that we can debug at source level, can we do similar things to physical design?

Two approaches:

- Setting constraints directly in the source code language, which later gets translated into tcl script.
- Dynamically query design information in tcl script and get required constraints.

Either way a symbol table is required!



Tcl dynamic query in action

Before:

```
create_power_domain AON -elements { PowerDomainOR DECODE_FEATURE_12
    coreir_eq_16_inst0 and_inst1 FEATURE_AND_12 PowerDomainConfigReg_inst0
    const_511_9 const_0_8}
```

After:

```
# hgdb is the tcl binding package
package require hgdb
# open the symbol table
set db [open_symbol_table $filename]
# query instances with annotation of "powerdomain"
set result [get_design_instances_with_anno $db "power_domain"]
# setting up power domain AON instances
create_power_domain AON -elements $result
```

Work in progress/future work

- Integration into magma and possibly Chisel
- Adding more functionality to tcl bindings
- Enhanced waveform viewer (with proper symbol mapping)
- Squash more bugs and adding features to the core runtime library
- More IDE-based debugger!
- Collaborating with Synopsys