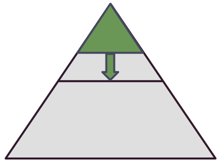
Verification with TL-Verilog

Steve Hoover
Founder, Redwood EDA
steve.hoover@redwoodeda.com

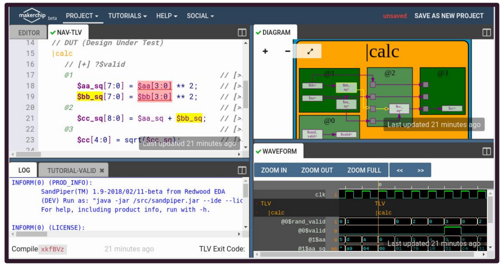
Feb. 2020

Agenda

- Methodology



- Makerchip TL-Verilog online IDE



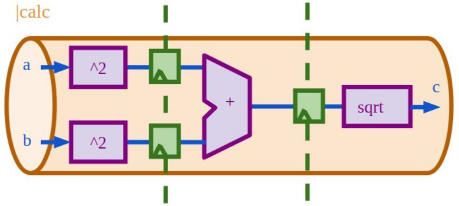
```
   // DUT (Design Under Test)
   |calc
      // [+] ?$valid
      @1
         $aa_sq[7:0] = $aa[3:0] ** 2;        // [>
         $bb_sq[7:0] = $bb[3:0] ** 2;        // [>
      @2                                      // [>
         $cc_sq[8:0] = $aa_sq + $bb_sq;      // [>
      @3                                      // [>
         $cc[4:0] = sqrt($cc_sq);            // [>
```

```
INFORM(0) (PROD_INFO):
      SandPiper(TM) 1.9-2018/02/11-beta from Redwood EDA
      (DEV) Run as: "java -jar /src/sandpiper.jar --ide --lic
      For help, including product info, run with -h.

INFORM(0) (LICENSE):
```

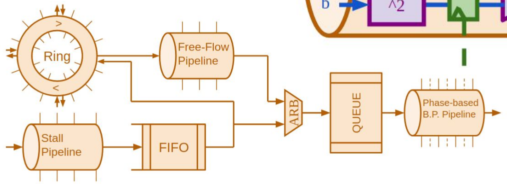
Compile xkfBVz 21 minutes ago TLV Exit Code:

```
clk
TLV
  |calc
      @0$rand_valid
          @0$valid
             @1$aa
          @1$aa_sq
```

- Pipelines (Timing Abstraction)

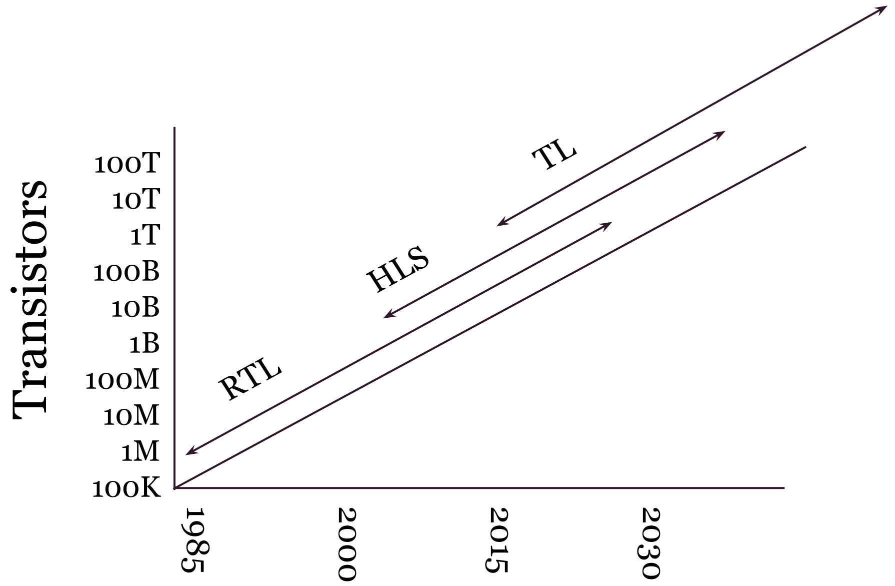


- Transactions

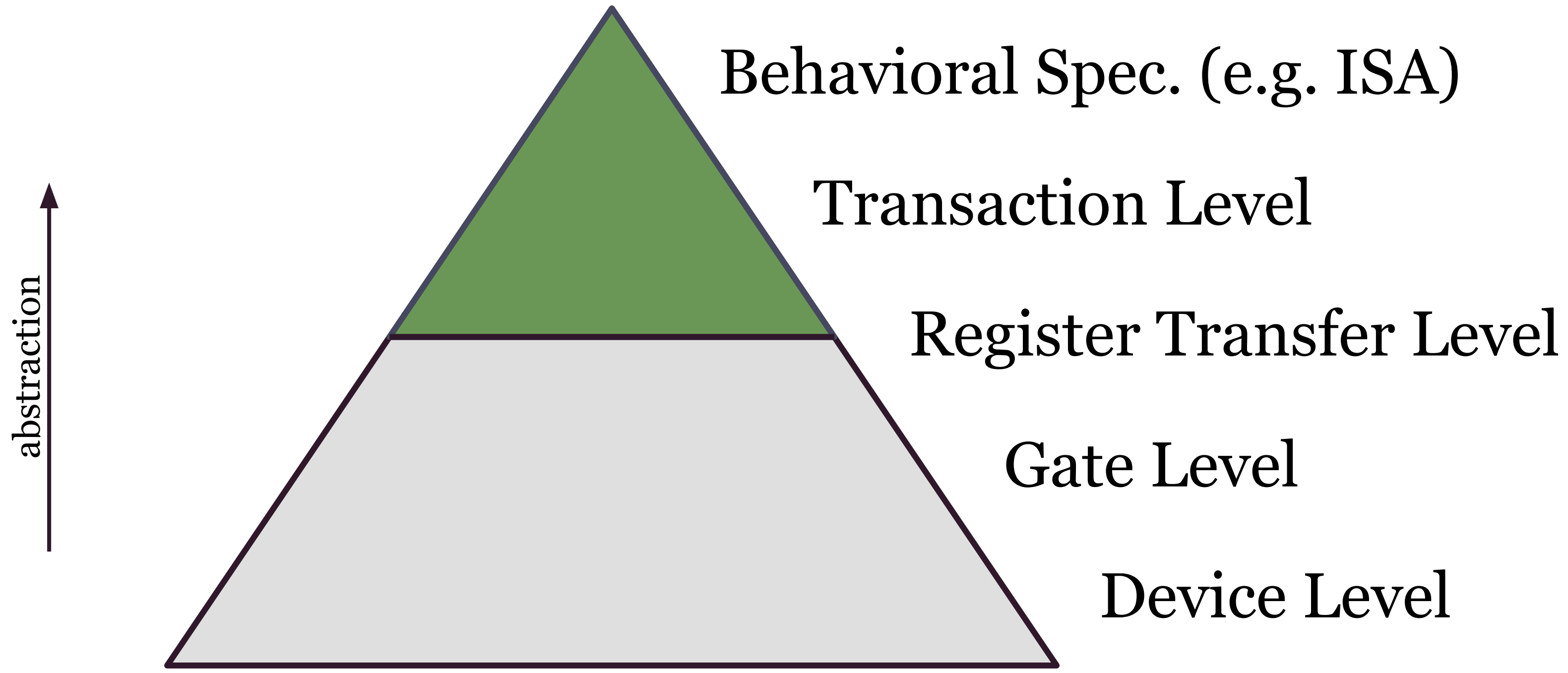


- A case study: Formally verifying RISC-V in 1 page of code

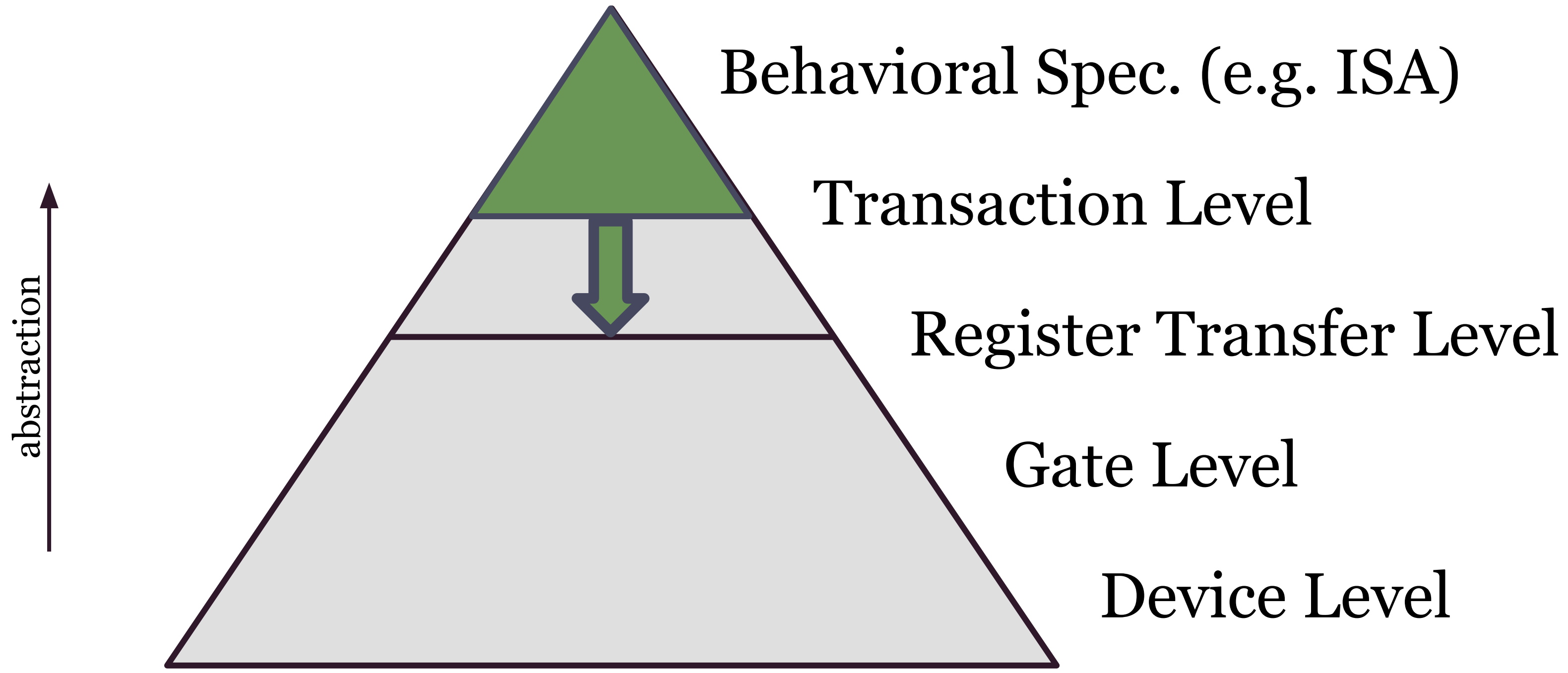
Where Transaction-Level Design Fits In



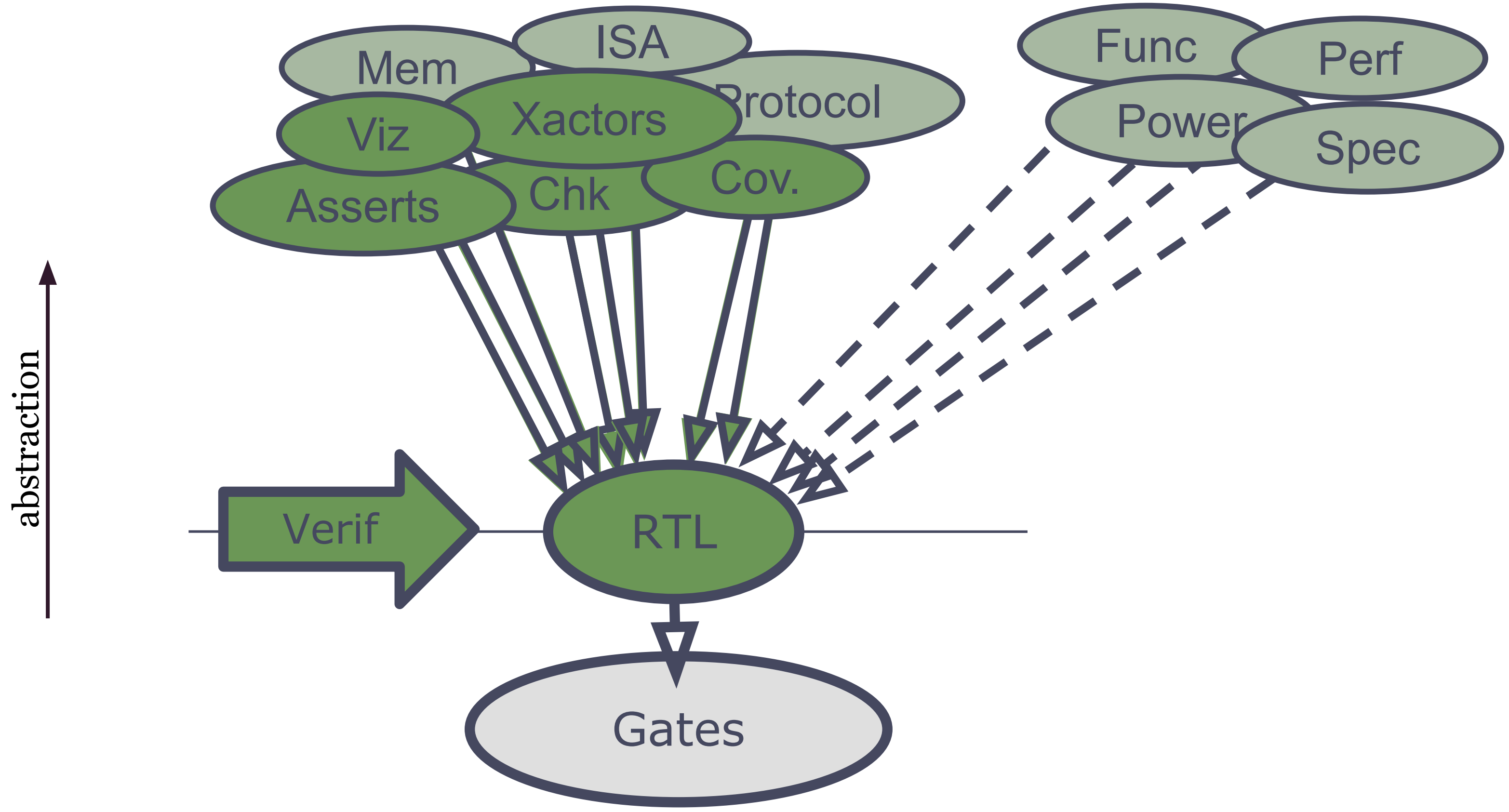
Abstraction Levels



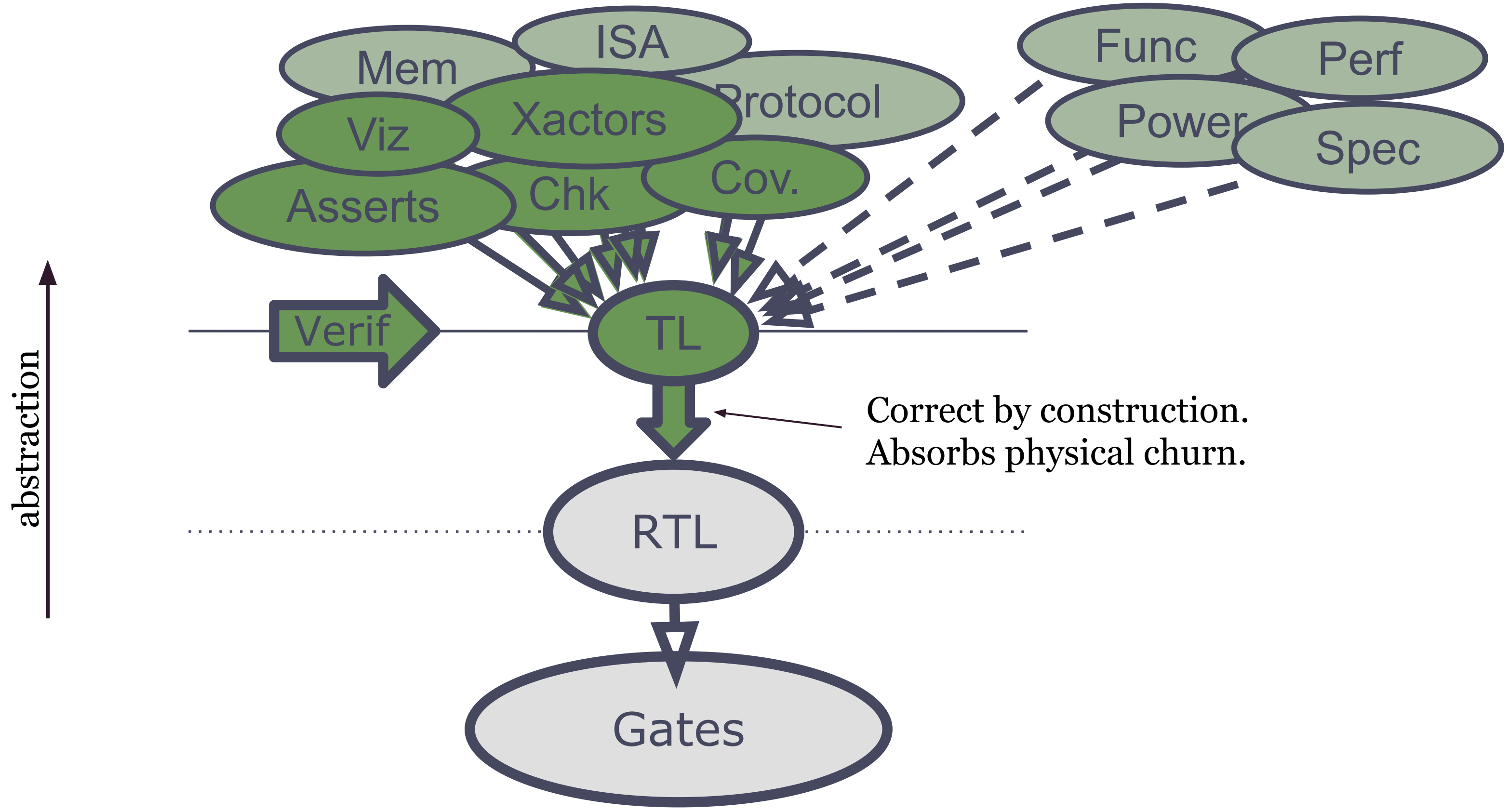
Abstraction Levels



RTL Design Methodology

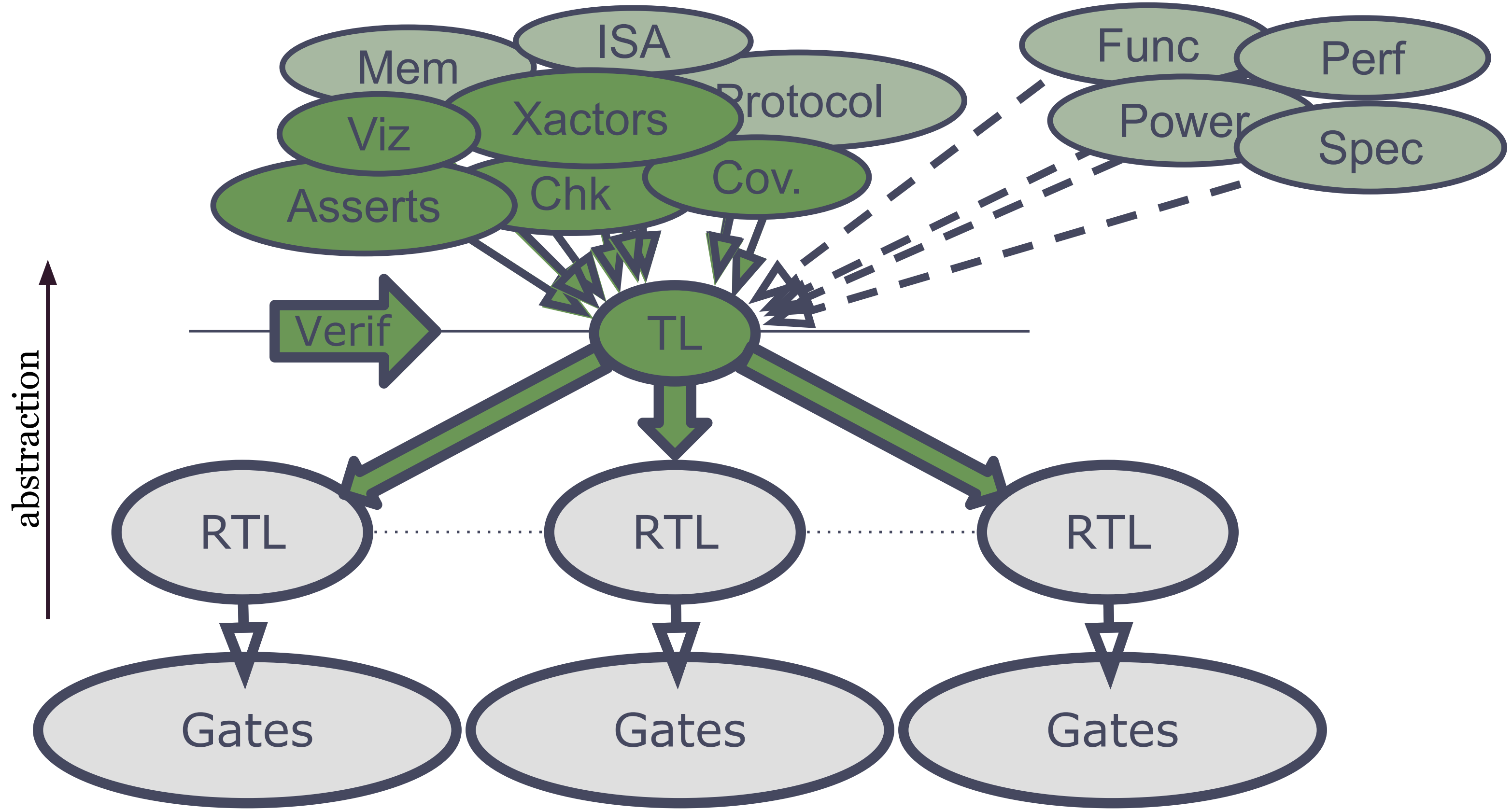


Transaction-Level Design Methodology



Correct by construction.
Absorbs physical churn.

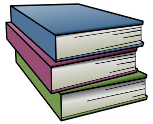
Transaction-Level Design Methodology



Post-RTL Directions

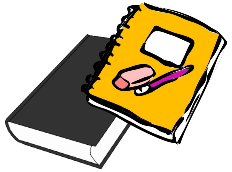
EDA Industry: C++-Based HLS

- Integrate w/ C++-based verification
- Synthesize algorithms to gate-level RTL
- Target multiple platforms (s/w, GPU, FPGA, etc.)



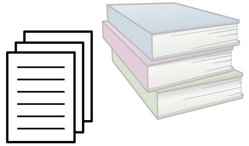
Academia: DSLs (Chisel, CλaSH, ...)

- Better RTL
- Leverage s/w techniques to construct h/w



Designers: TL-X (TL-Verilog)

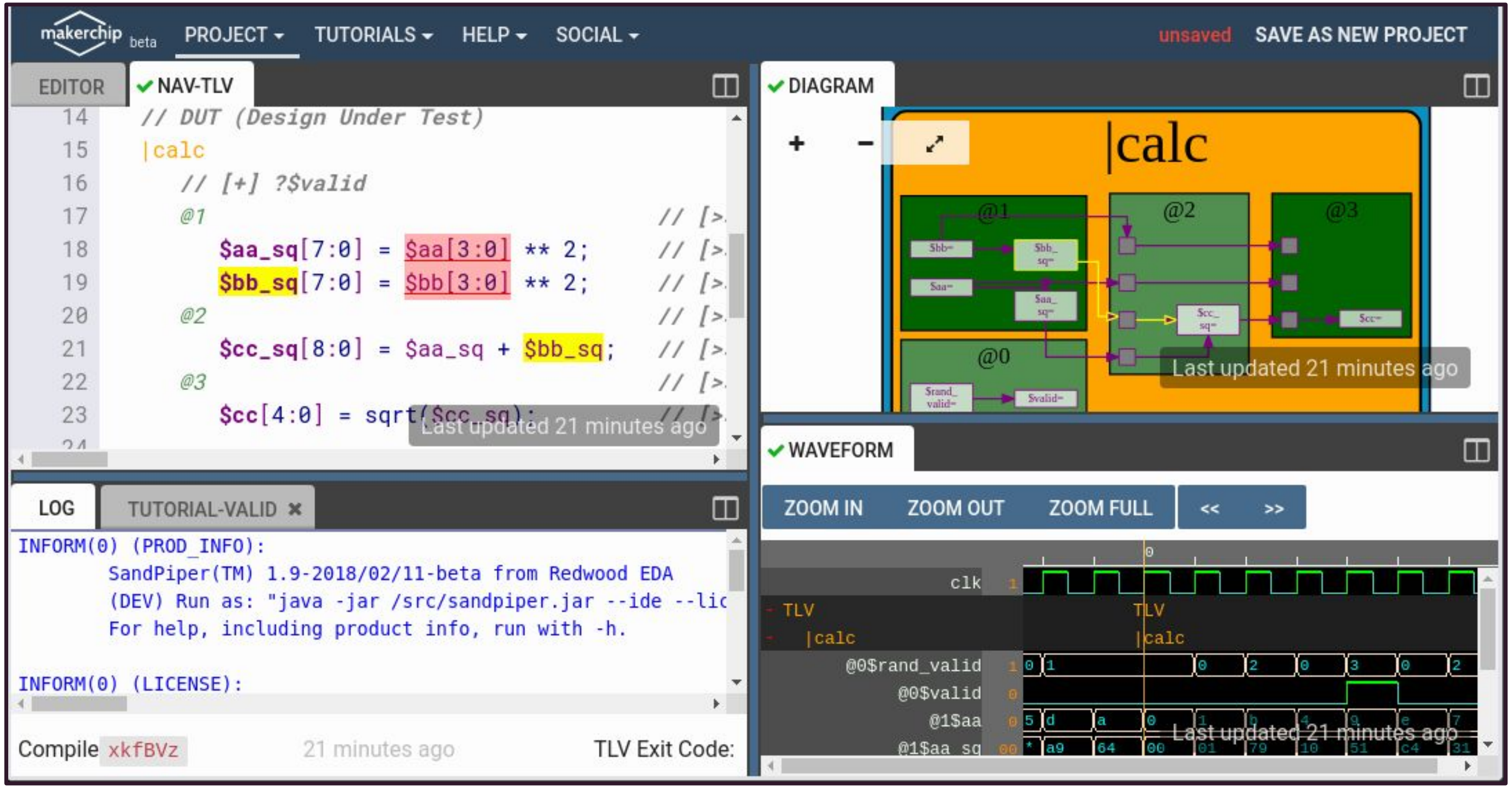
- H/w modeling (w/ HLS) deserves its own language
- Abstraction as context for details (if details are needed)



Transaction-Level Verilog...

- Extends SystemVerilog environment
- Eliminates baggage
  (generate, blocking vs. non-blocking, packed vs. unpacked, always blocks, sensitivity lists, reg vs. wire vs. logic vs. bit, etc.)
- *Initially* focused on design (vs. verification)
- Enables fully-synthesizable HLM

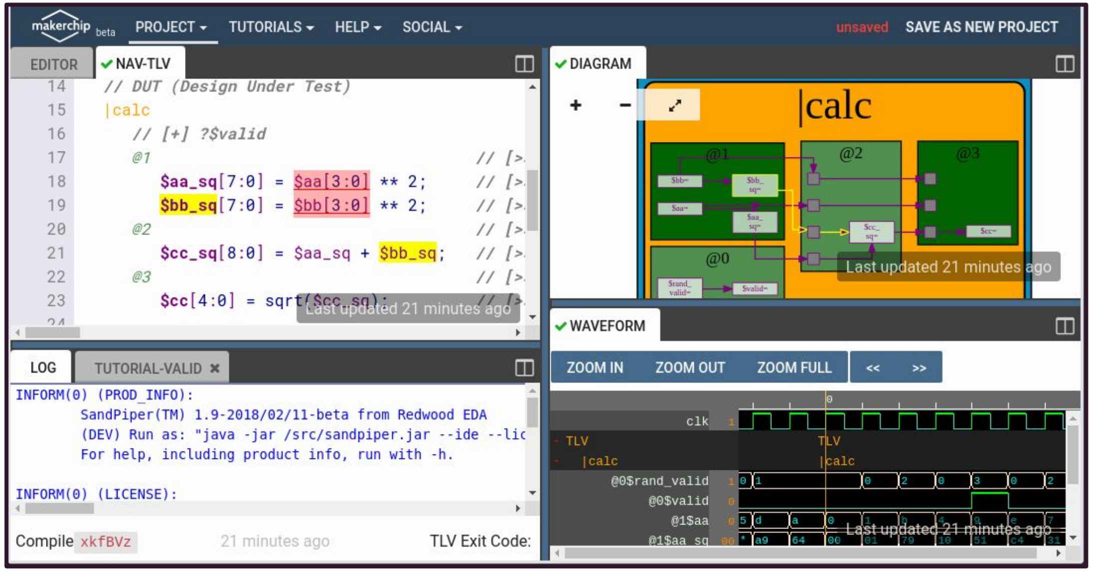
Makerchip.com



```
// DUT (Design Under Test)
|calc
   // [+] ?$valid
   @1                                                // [>
      $aa_sq[7:0] = $aa[3:0] ** 2;                   // [>
      $bb_sq[7:0] = $bb[3:0] ** 2;                   // [>
   @2                                                // [>
      $cc_sq[8:0] = $aa_sq + $bb_sq;                 // [>
   @3                                                // [>
      $cc[4:0] = sqrt($cc_sq);                       // [>
```

```
INFORM(0) (PROD_INFO):
        SandPiper(TM) 1.9-2018/02/11-beta from Redwood EDA
        (DEV) Run as: "java -jar /src/sandpiper.jar --ide --lic
        For help, including product info, run with -h.

INFORM(0) (LICENSE):
```

Lab: Makerchip Platform



```
   // DUT (Design Under Test)
   |calc
      // [+] ?$valid
      @1                                            // [>
         $aa_sq[7:0] = $aa[3:0] ** 2;              // [>
         $bb_sq[7:0] = $bb[3:0] ** 2;              // [>
      @2                                            // [>
         $cc_sq[8:0] = $aa_sq + $bb_sq;            // [>
      @3                                            // [>
         $cc[4:0] = sqrt($cc_sq);                  // [>
```

```
INFORM(0) (PROD_INFO):
      SandPiper(TM) 1.9-2018/02/11-beta from Redwood EDA
      (DEV) Run as: "java -jar /src/sandpiper.jar --ide --lic
      For help, including product info, run with -h.

INFORM(0) (LICENSE):
```

Compile xkfBVz 21 minutes ago TLV Exit Code:

1. On desktop machine, in modern web browser (not IE), go to: makerchip.com
2. Click “IDE”.

Reproduce this screenshot:

1. Open “Tutorials” “Validity Tutorial”.
2. In tutorial, click **Load Pythagorean Example**
3. Split panes and move tabs.
4. Zoom/pan in Diagram w/ mouse wheel and drag.
5. Zoom Waveform w/ “Zoom In” button.
6. Click `$bb_sq` to highlight.

Lab: Combinational Logic

A) Inverter

1. Open "Examples" (under "Tutorials").
2. Load "Default Template".
3. Make an inverter.
   In place of:

```
   //...
```

   type:

```
   $out = ! $in1;
```

   (Preserve 3-space indentation)
4. Compile ("E" menu) & Explore

Note:

1. There was no need to declare `$out` and `$in1` (unlike Verilog).
2. There was no need to assign `$in1`. Random stimulus is provided, and a warning is produced.

B) Other logic

1. Make a 2-input gate.
   (Boolean operators: (`&&`, `||`, `^`))

Lab: Vectors

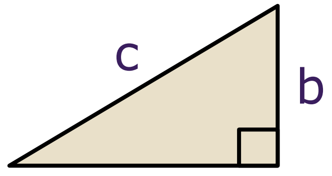
`$out[4:0]` creates a “vector” of 5 bits.

Arithmetic operators operate on vectors as binary numbers.

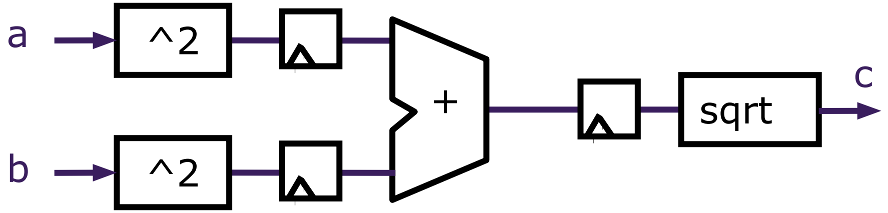
1. Try:
   ```
   $out[4:0] = $in1[3:0] + $in2[3:0];
   ```
   (Cut-n-paste)
2. View Waveform (values are in hexadecimal and addition can overflow)

A Simple Pipeline

- Let's compute Pythagoras's Theorem in hardware.
- We distribute the calculation over three cycles.

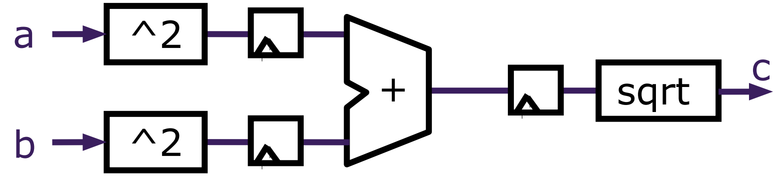


c = sqrt(a^2 + b^2)

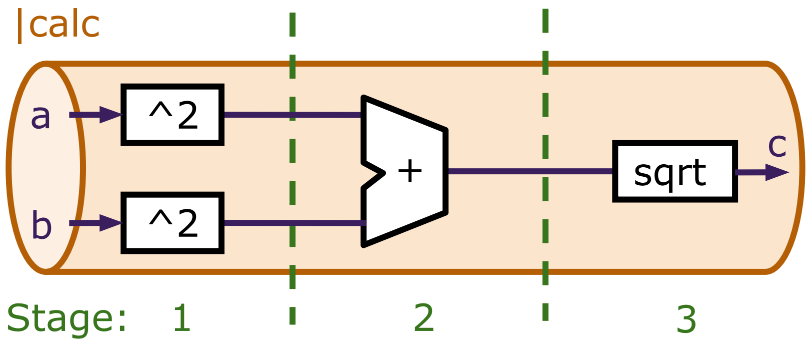


A Simple Pipeline - Timing-Abstract

RTL:

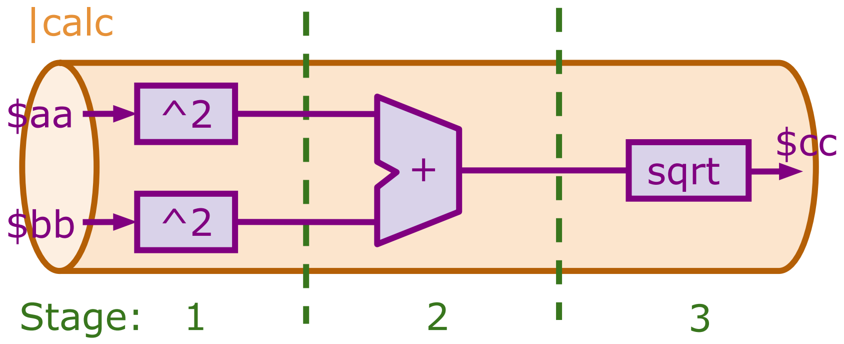


Timing-abstract:



➔ Flip-flops and staged signals are implied from context.

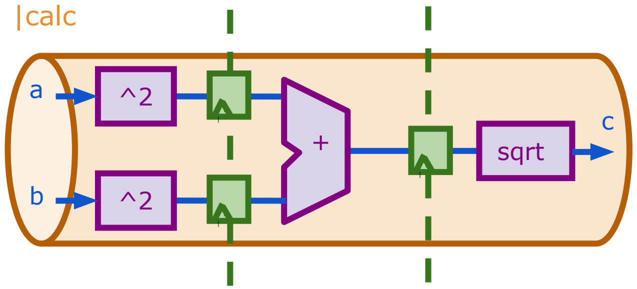
A Simple Pipeline - TL-Verilog



TL-Verilog

```
|calc
   @1
      $aa_sq[31:0] = $aa * $aa;
      $bb_sq[31:0] = $bb * $bb;
   @2
      $cc_sq[31:0] = $aa_sq + $bb_sq;
   @3
      $cc[31:0] = sqrt($cc_sq);
```

SystemVerilog vs. TL-Verilog



System Verilog

```
// Calc Pipeline
logic [31:0] a_C1;
logic [31:0] b_C1;
logic [31:0] a_sq_C1,
             a_sq_C2;
logic [31:0] b_sq_C1,
             b_sq_C2;
logic [31:0] c_sq_C2,
             c_sq_C3;
logic [31:0] c_C3;
always_ff @(posedge clk) a_sq_C2 <= a_sq_C1;
always_ff @(posedge clk) b_sq_C2 <= b_sq_C1;
always_ff @(posedge clk) c_sq_C3 <= c_sq_C2;
// Stage 1
assign a_sq_C1 = a_C1 * a_C1;
assign b_sq_C1 = b_C1 * b_C1;
// Stage 2
assign c_sq_C2 = a_sq_C2 + b_sq_C2;
// Stage 3
assign c_C3 = sqrt(c_sq_C3);
```

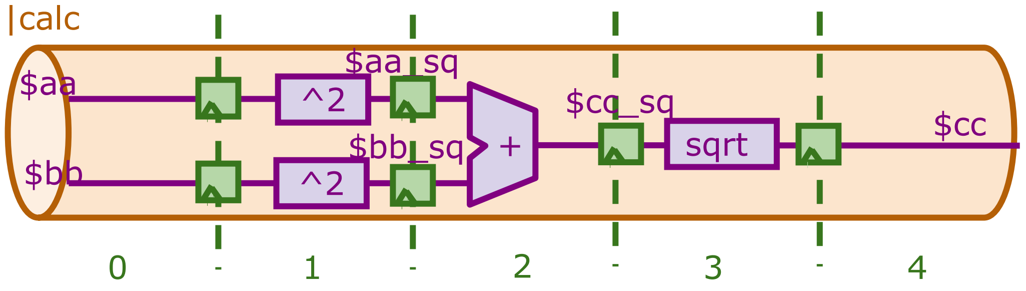
~3.5x

TL-Verilog

```
|calc
   @1
      $aa_sq[31:0] = $aa * $aa;
      $bb_sq[31:0] = $bb * $bb;
   @2
      $cc_sq[31:0] = $aa_sq + $bb_sq;
   @3
      $cc[31:0] = sqrt($cc_sq);
```

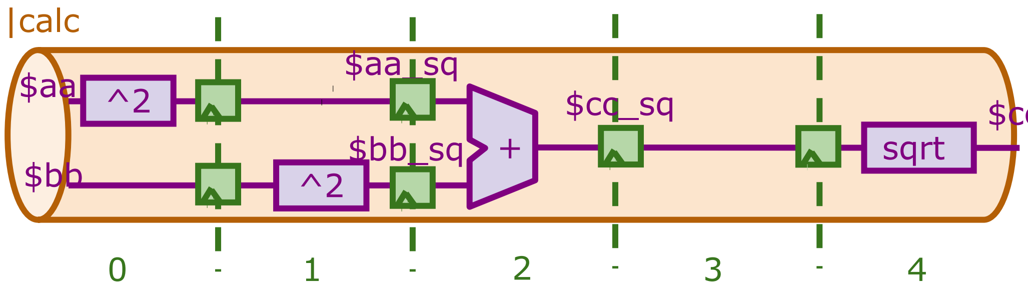
Retiming -- Easy and Safe

```
|calc
   @1
      $aa_sq[31:0] = $aa * $aa;
      $bb_sq[31:0] = $bb * $bb;
   @2
      $cc_sq[31:0] = $aa_sq + $bb_sq;
   @3
      $cc[31:0]  = sqrt($cc_sq);
```



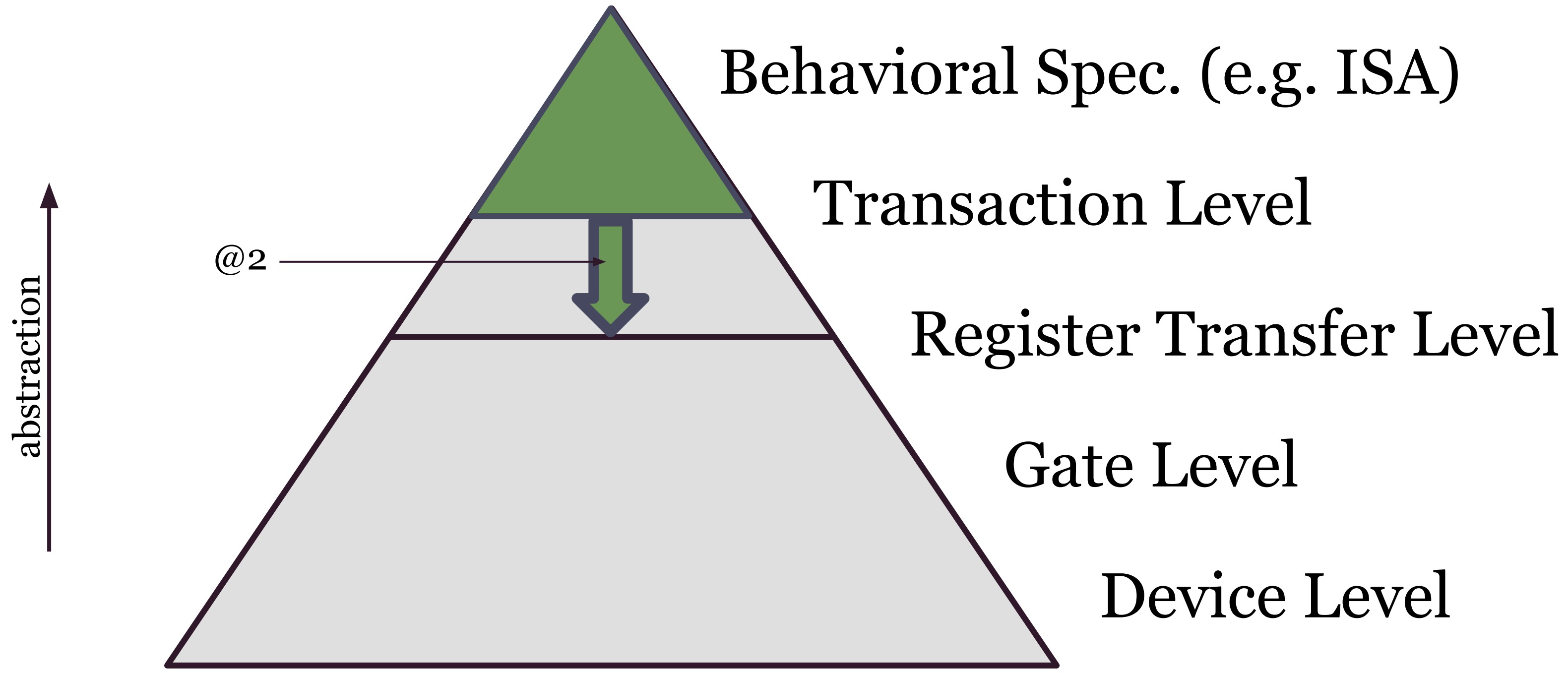
==

```
|calc
   @0
      $aa_sq[31:0] = $aa * $aa;
   @1
      $bb_sq[31:0] = $bb * $bb;
   @2
      $cc_sq[31:0] = $aa_sq + $bb_sq;
   @4
      $cc[31:0]  = sqrt($cc_sq);
```



Staging is a physical attribute. No impact to behavior.

Abstraction Levels



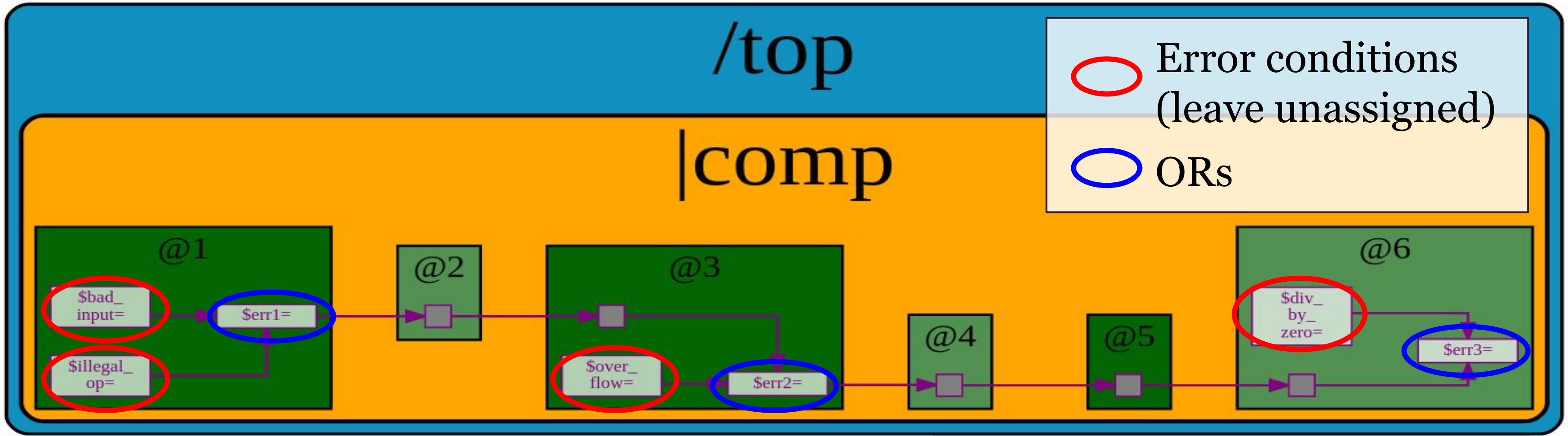
Retiming in SystemVerilog

```
// Calc Pipeline
logic [31:0] a_C1;
logic [31:0] b_C1;
logic [31:0] a_sq_C0,
             a_sq_C1,
             a_sq_C2;
logic [31:0] b_sq_C1,
             b_sq_C2;
logic [31:0] c_sq_C2,
             c_sq_C3,
             c_sq_C4;
logic [31:0] c_C3;
always_ff @(posedge clk) a_sq_C2 <= a_sq_C1;
always_ff @(posedge clk) b_sq_C2 <= b_sq_C1;
always_ff @(posedge clk) c_sq_C3 <= c_sq_C2;
always_ff @(posedge clk) c_sq_C4 <= c_sq_C3;
// Stage 1
assign a_sq_C1 = a_C1 * a_C1;
assign b_sq_C1 = b_C1 * b_C1;
// Stage 2
assign c_sq_C2 = a_sq_C2 + b_sq_C2;
// Stage 3
assign c_C3 = sqrt(c_sq_C3);
```

VERY BUG-PRONE!

Lab: Pipeline

1) See if you can produce this:



which ORs together (| |) various error conditions
that can occur w[illegible]ipeline.

3 spaces

Open in Makerchip

(makerchip.com/sandbox/0/0xGhJP)

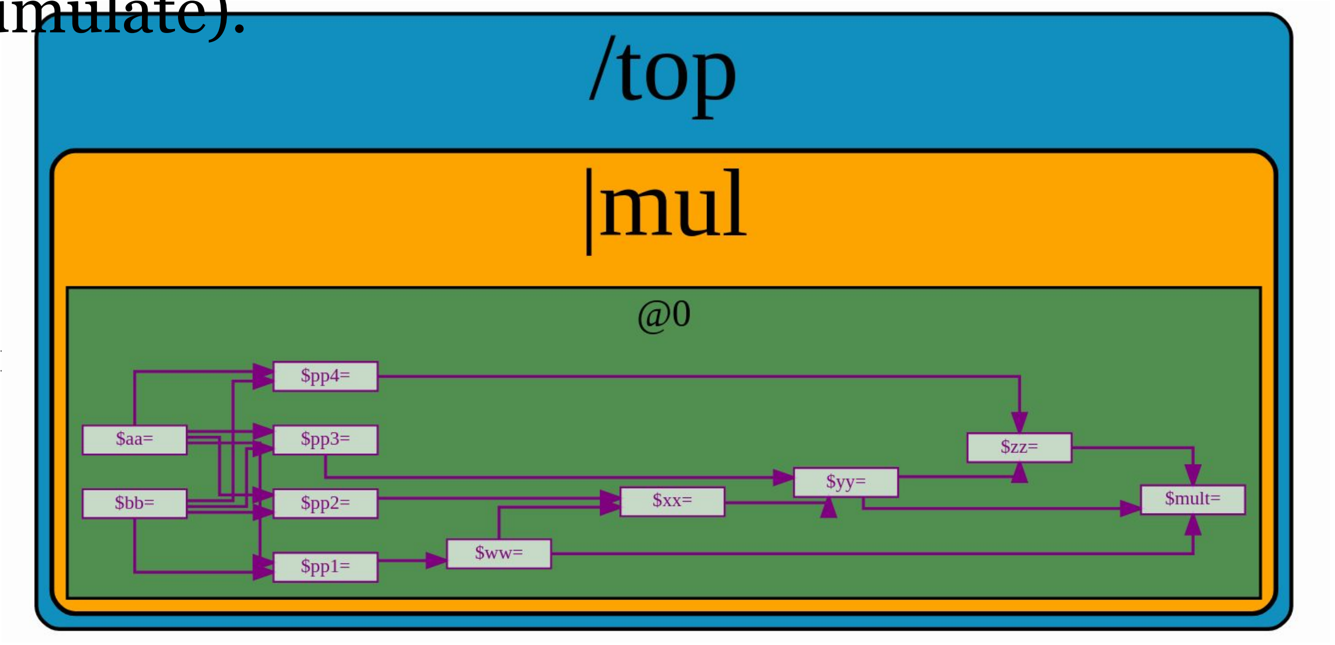
(for reference)

```
|calc
   @1
      $aa_sq[31:0] = $aa * $aa;
      $bb_sq[31:0] = $bb * $bb;
   @2
      $cc_sq[31:0] = $aa_sq + $bb_sq;
   @3
      $cc[31:0]  = sqrt($cc_sq);
```

Partial-Products Multiplication

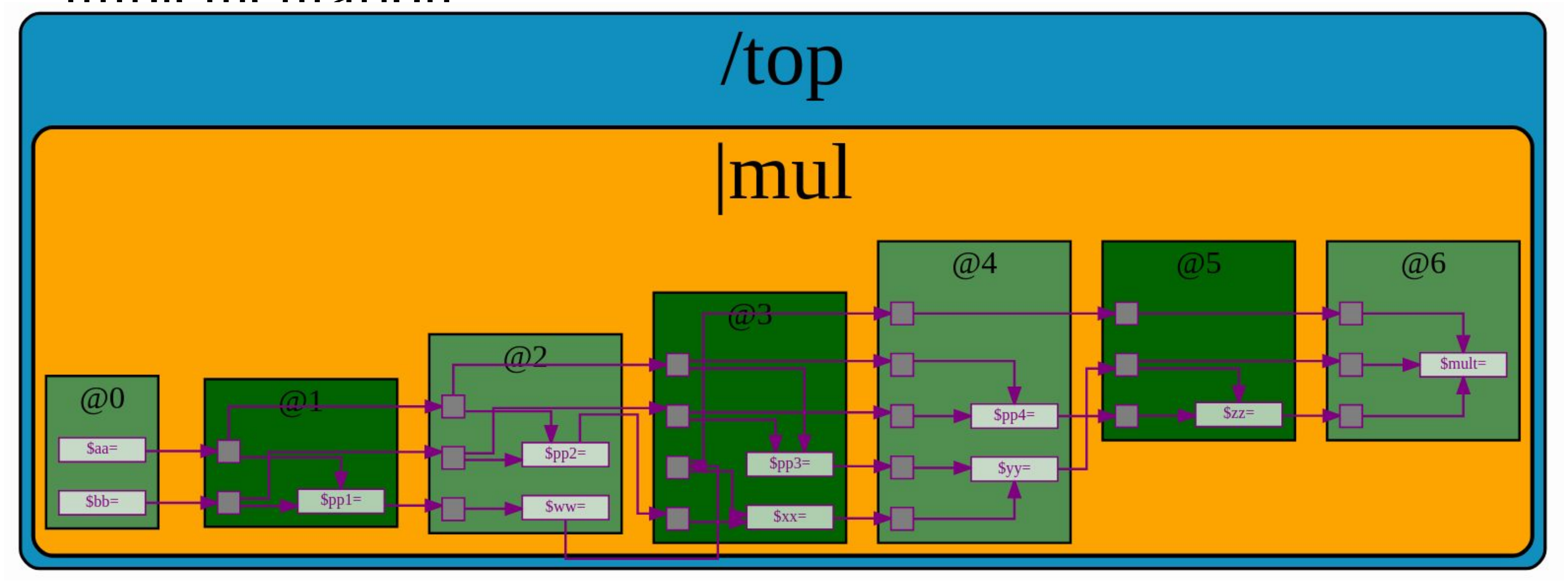
Some FPGAs natively support 18-bit MAC (multiply-accumulate).

```
 ($aa)       A a
 ($bb)     x B b
            ----
($pp1)       (ab)
($pp2)   + (Ab)
        ---------
          xxxxww
     ($pp3)   + (a
        ---------
          yyyyww
       ($pp4) + (A
      ----------
        zzzzyyww   |
```



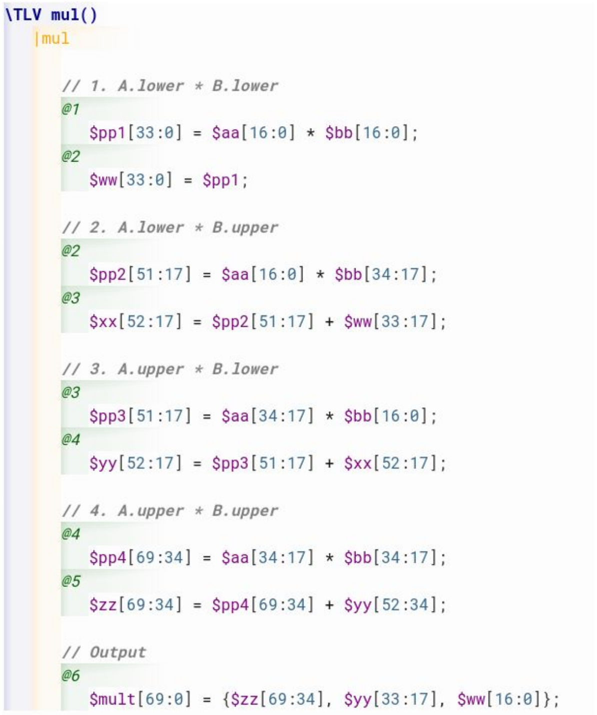
Wide Pipelined Multiply in an FPGA

surf-vhdl.com suggests the following pipelined implementation:



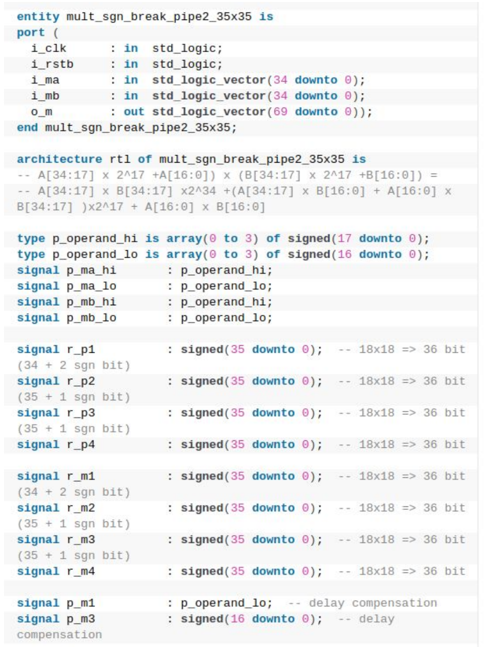
Logic Comparison

TL-Verilog

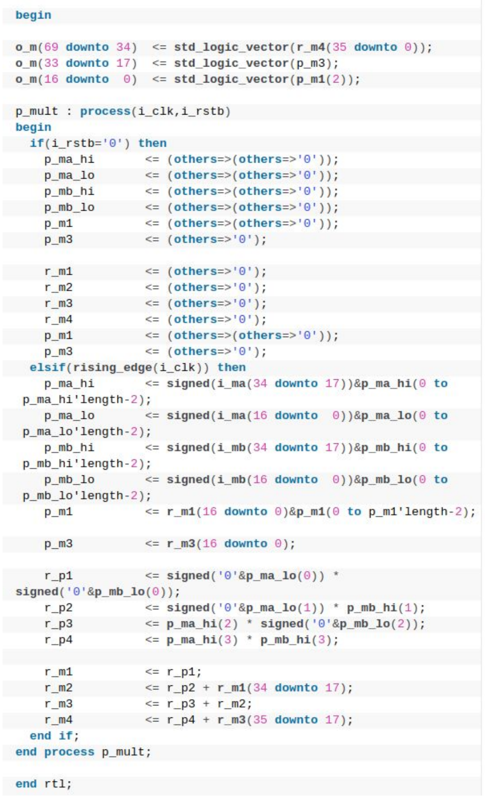


```
\TLV mul()
   |mul

      // 1. A.lower * B.lower
      @1
         $pp1[33:0] = $aa[16:0] * $bb[16:0];
      @2
         $ww[33:0] = $pp1;

      // 2. A.lower * B.upper
      @2
         $pp2[51:17] = $aa[16:0] * $bb[34:17];
      @3
         $xx[52:17] = $pp2[51:17] + $ww[33:17];

      // 3. A.upper * B.lower
      @3
         $pp3[51:17] = $aa[34:17] * $bb[16:0];
      @4
         $yy[52:17] = $pp3[51:17] + $xx[52:17];

      // 4. A.upper * B.upper
      @4
         $pp4[69:34] = $aa[34:17] * $bb[34:17];
      @5
         $zz[69:34] = $pp4[69:34] + $yy[52:34];

      // Output
      @6
         $mult[69:0] = {$zz[69:34], $yy[33:17], $ww[16:0]};
```

VHDL

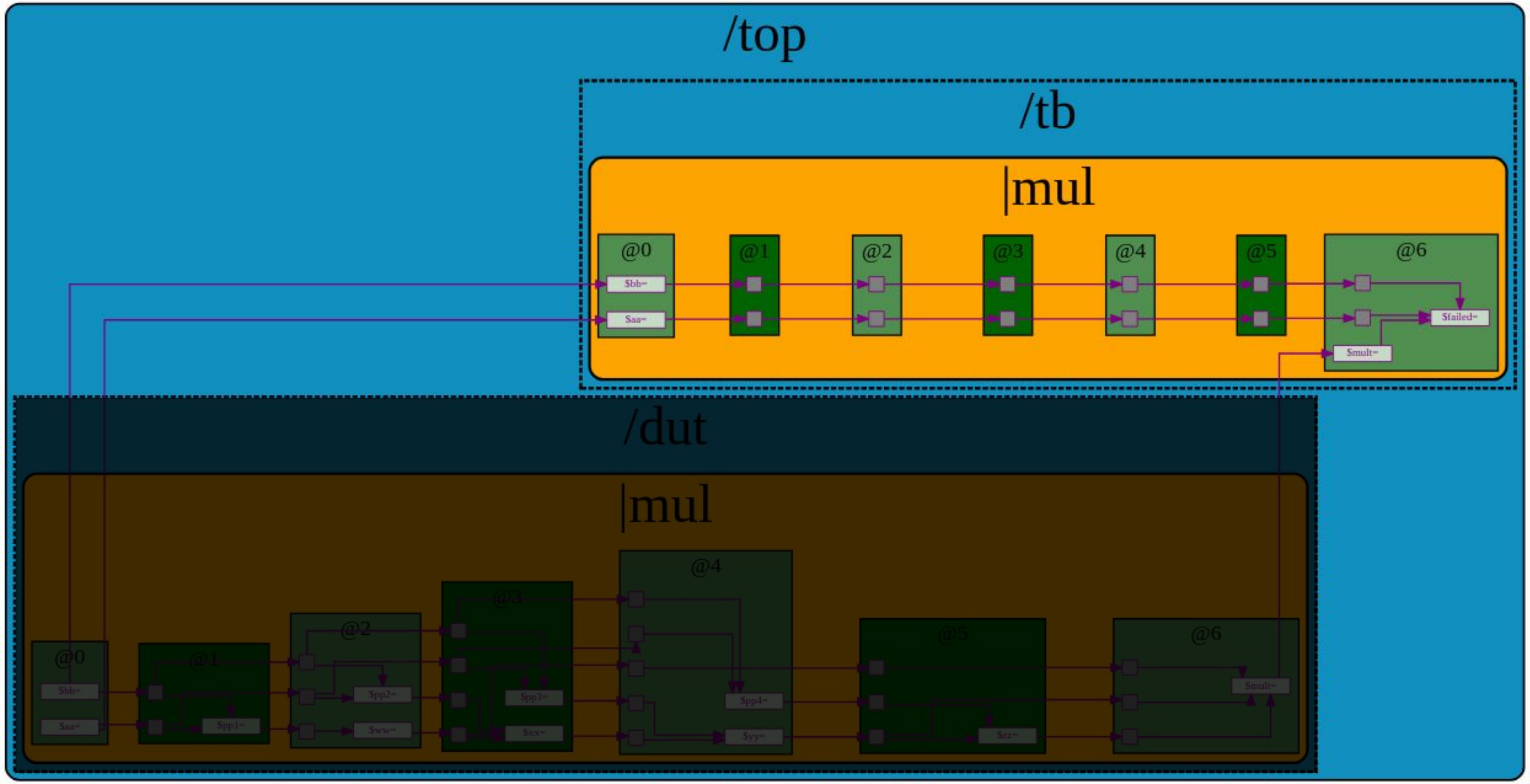


```
entity mult_sgn_break_pipe2_35x35 is
port (
  i_clk       : in  std_logic;
  i_rstb      : in  std_logic;
  i_ma        : in  std_logic_vector(34 downto 0);
  i_mb        : in  std_logic_vector(34 downto 0);
  o_m         : out std_logic_vector(69 downto 0));
end mult_sgn_break_pipe2_35x35;

architecture rtl of mult_sgn_break_pipe2_35x35 is
-- A[34:17] x 2^17 +A[16:0]) x (B[34:17] x 2^17 +B[16:0]) =
-- A[34:17] x B[34:17] x2^34 +(A[34:17] x B[16:0] + A[16:0] x
B[34:17] )x2^17 + A[16:0] x B[16:0]

type p_operand_hi is array(0 to 3) of signed(17 downto 0);
type p_operand_lo is array(0 to 3) of signed(16 downto 0);
signal p_ma_hi        : p_operand_hi;
signal p_ma_lo        : p_operand_lo;
signal p_mb_hi        : p_operand_hi;
signal p_mb_lo        : p_operand_lo;

signal r_p1           : signed(35 downto 0);  -- 18x18 => 36 bit
(34 + 2 sgn bit)
signal r_p2           : signed(35 downto 0);  -- 18x18 => 36 bit
(35 + 1 sgn bit)
signal r_p3           : signed(35 downto 0);  -- 18x18 => 36 bit
(35 + 1 sgn bit)
signal r_p4           : signed(35 downto 0);  -- 18x18 => 36 bit

signal r_m1           : signed(35 downto 0);  -- 18x18 => 36 bit
(34 + 2 sgn bit)
signal r_m2           : signed(35 downto 0);  -- 18x18 => 36 bit
(35 + 1 sgn bit)
signal r_m3           : signed(35 downto 0);  -- 18x18 => 36 bit
(35 + 1 sgn bit)
signal r_m4           : signed(35 downto 0);  -- 18x18 => 36 bit

signal p_m1           : p_operand_lo;  -- delay compensation
signal p_m3           : signed(16 downto 0);  -- delay
compensation
```

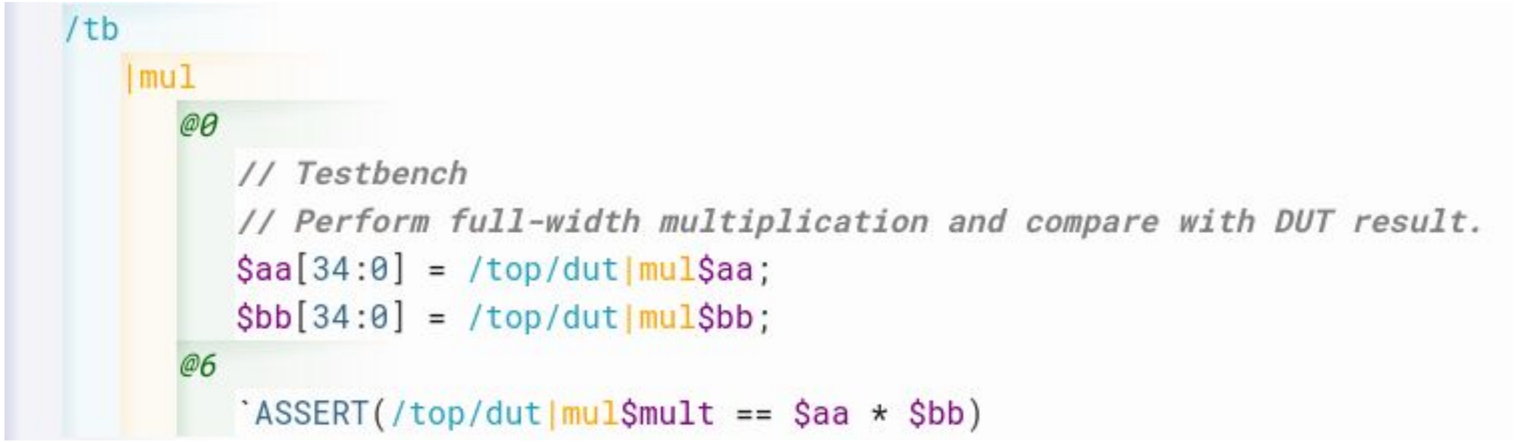


```
begin

o_m(69 downto 34)  <= std_logic_vector(r_m4(35 downto 0));
o_m(33 downto 17)  <= std_logic_vector(p_m3);
o_m(16 downto  0)  <= std_logic_vector(p_m1(2));

p_mult : process(i_clk,i_rstb)
begin
  if(i_rstb='0') then
    p_ma_hi        <= (others=>(others=>'0'));
    p_ma_lo        <= (others=>(others=>'0'));
    p_mb_hi        <= (others=>(others=>'0'));
    p_mb_lo        <= (others=>(others=>'0'));
    p_m1           <= (others=>(others=>'0'));
    p_m3           <= (others=>'0');

    r_m1           <= (others=>'0');
    r_m2           <= (others=>'0');
    r_m3           <= (others=>'0');
    r_m4           <= (others=>'0');
    p_m1           <= (others=>(others=>'0'));
    p_m3           <= (others=>'0');
  elsif(rising_edge(i_clk)) then
    p_ma_hi        <= signed(i_ma(34 downto 17))&p_ma_hi(0 to
 p_ma_hi'length-2);
    p_ma_lo        <= signed(i_ma(16 downto  0))&p_ma_lo(0 to
 p_ma_lo'length-2);
    p_mb_hi        <= signed(i_mb(34 downto 17))&p_mb_hi(0 to
 p_mb_hi'length-2);
    p_mb_lo        <= signed(i_mb(16 downto  0))&p_mb_lo(0 to
 p_mb_lo'length-2);
    p_m1           <= r_m1(16 downto 0)&p_m1(0 to p_m1'length-2);

    p_m3           <= r_m3(16 downto 0);

    r_p1           <= signed('0'&p_ma_lo(0)) *
signed('0'&p_mb_lo(0));
    r_p2           <= signed('0'&p_ma_lo(1)) * p_mb_hi(1);
    r_p3           <= p_ma_hi(2) * signed('0'&p_mb_lo(2));
    r_p4           <= p_ma_hi(3) * p_mb_hi(3);

    r_m1           <= r_p1;
    r_m2           <= r_p2 + r_m1(34 downto 17);
    r_m3           <= r_p3 + r_m2;
    r_m4           <= r_p4 + r_m3(35 downto 17);
  end if;
end process p_mult;

end rtl;
```

Testbench Diagram

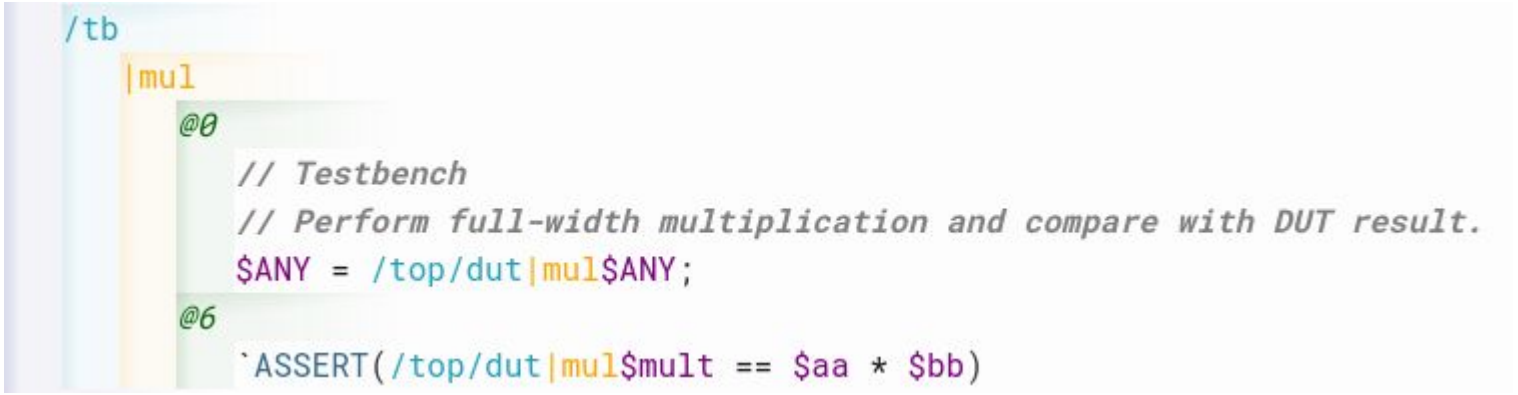


Testbench Code



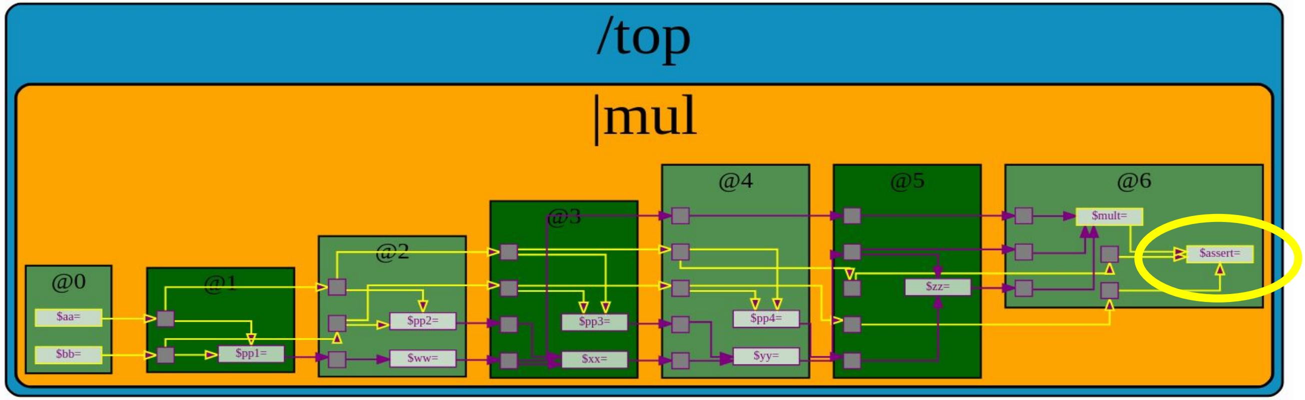
```
/tb
   |mul
      @0
         // Testbench
         // Perform full-width multiplication and compare with DUT result.
         $aa[34:0] = /top/dut|mul$aa;
         $bb[34:0] = /top/dut|mul$bb;
      @6
         `ASSERT(/top/dut|mul$mult == $aa * $bb)
```

Testbench Code



```
/tb
   |mul
      @0
         // Testbench
         // Perform full-width multiplication and compare with DUT result.
         $ANY = /top/dut|mul$ANY;
      @6
         `ASSERT(/top/dut|mul$mult == $aa * $bb)
```

Inline Assertion



SystemVerilog (SVA):

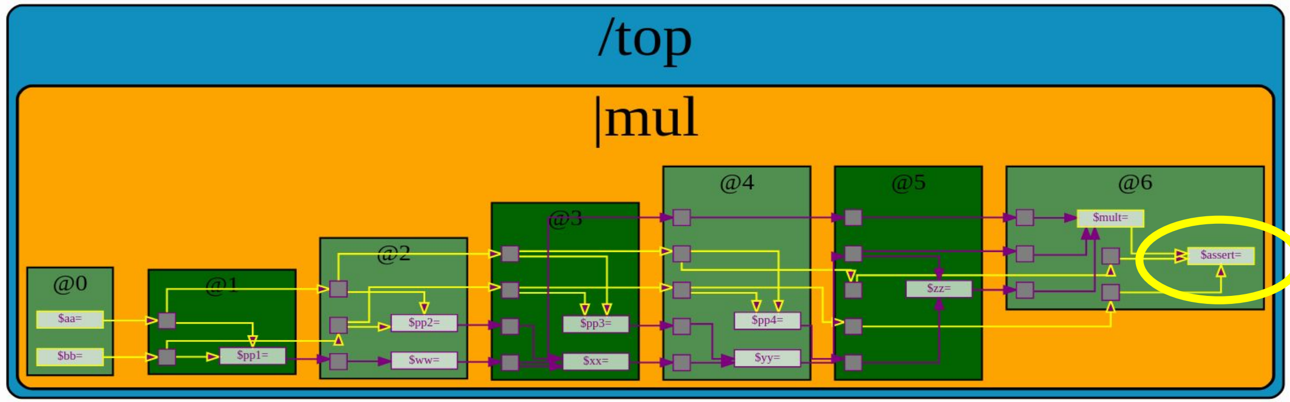
```
`ASSERT(mult_st6 == $past(a_st0, 6) * $past(b_st0, 6))
```

TL-Verilog:

```
`ASSERT($mult == $aa * $bb);
```

Still valid if pipeline depth changes.

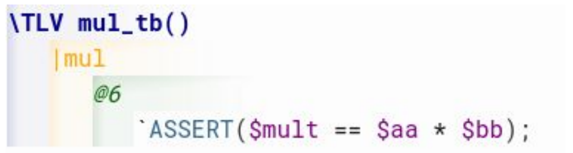
Isolated Testbench



SystemVerilog:

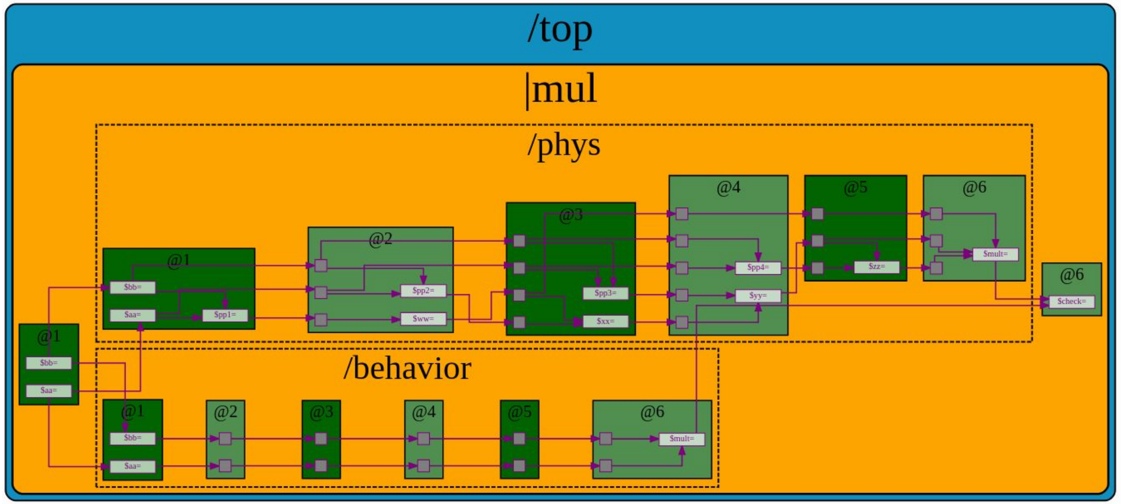
```
bind mul mul_asserts p1 (.*);
module fifo1_asserts (
     clk, reset,
     input logic [34:0] a_st0,
     input logic [34:0] b_st0,
     input logic [69:0] mult_st6);
   `ASSERT(mult_st6 == $past(a_st0, 6) * $past(b_st0, 6))
endmodule
```

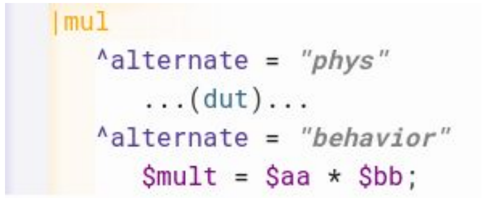
TL-Verilog:



```
\TLV mul_tb()
   |mul
      @6
         `ASSERT($mult == $aa * $bb);
```

Ideal (not currently supported)



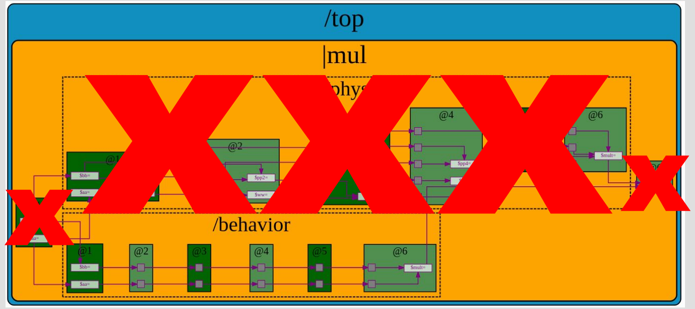


```
|mul
  ^alternate = "phys"
    ...(dut)...
  ^alternate = "behavior"
    $mult = $aa * $bb;
```

(or something to this affect)

Simulation/Emulation

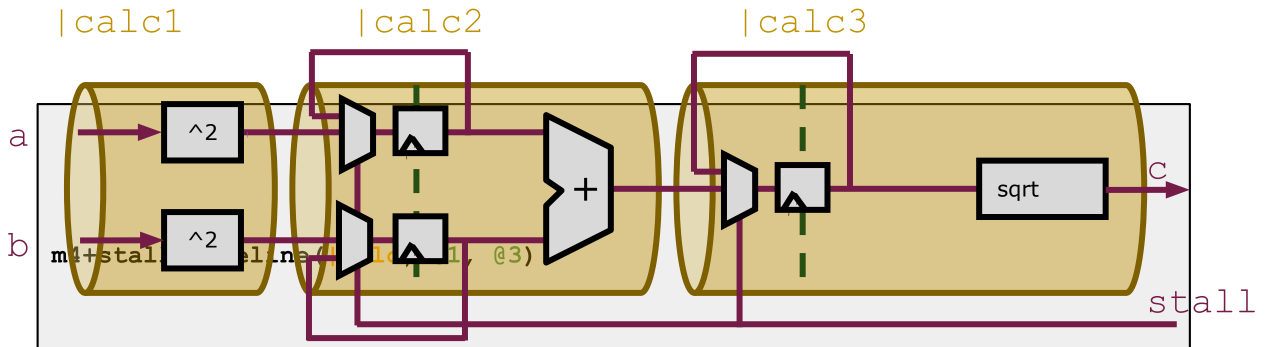
Full-chip



Stuff We’ll Skip

- State
- Validity
- Pipeline interactions
- Hierarchy (replication)

Adding Stall Network

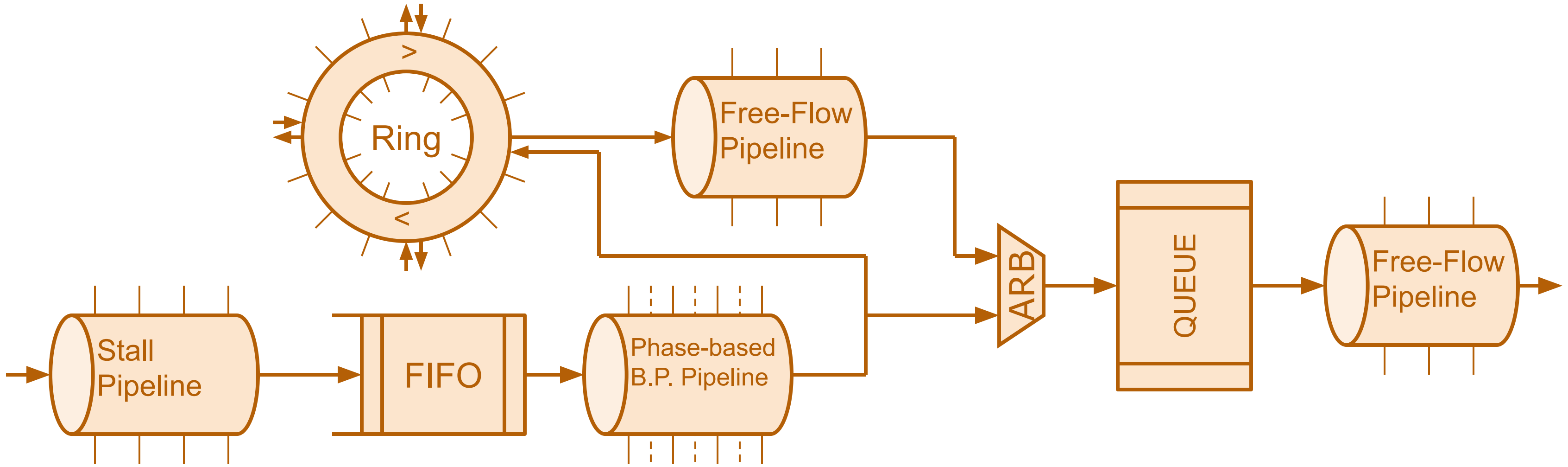


```
m4+stall_pipeline(|calc, @1, @3)

|calc1
   @1
```

```
m4+stall_pipeline(|calc, @1, @3)

|calc1
   @1
      $aa_sq[31:0] = $aa * $aa;
      $bb_sq[31:0] = $bb * $bb;
|calc2
   @1
      $cc_sq[31:0] = $aa_sq + $bb_sq;
|calc3
   @1
      $cc[31:0] = sqrt($cc_sq);
```

```
|calc
   @1
      $aa_sq[31:0] = $aa * $aa;
      $bb_sq[31:0] = $bb * $bb;
   @2
      $cc_sq[31:0] = $aa_sq + $bb_sq;
   @3
      $cc[31:0] = sqrt($cc_sq);
```

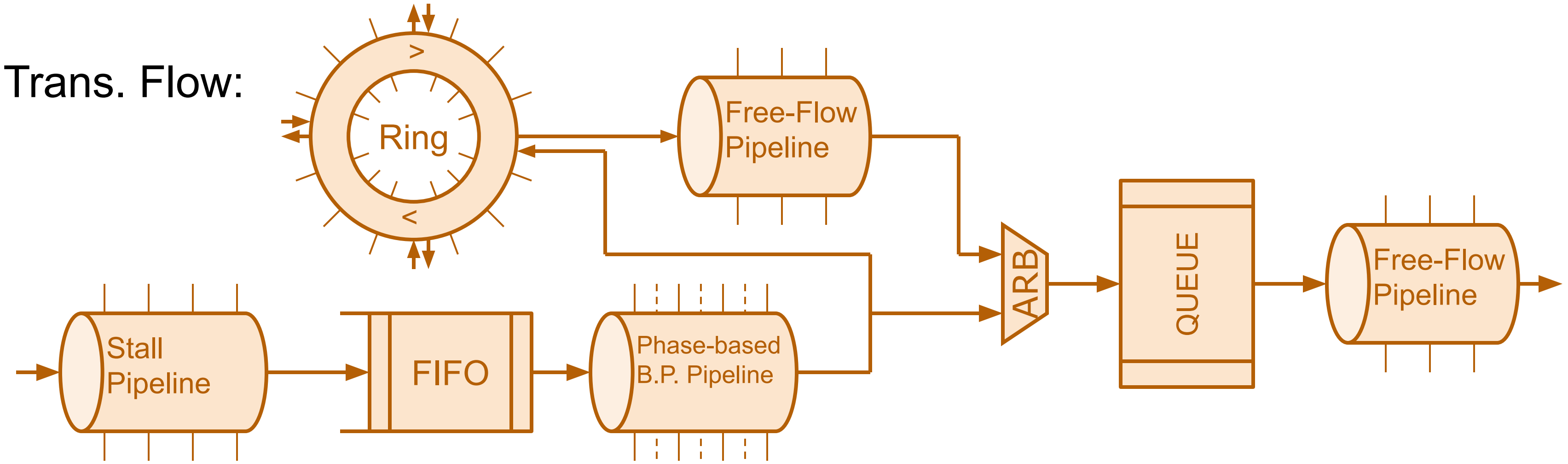
Transaction Flow



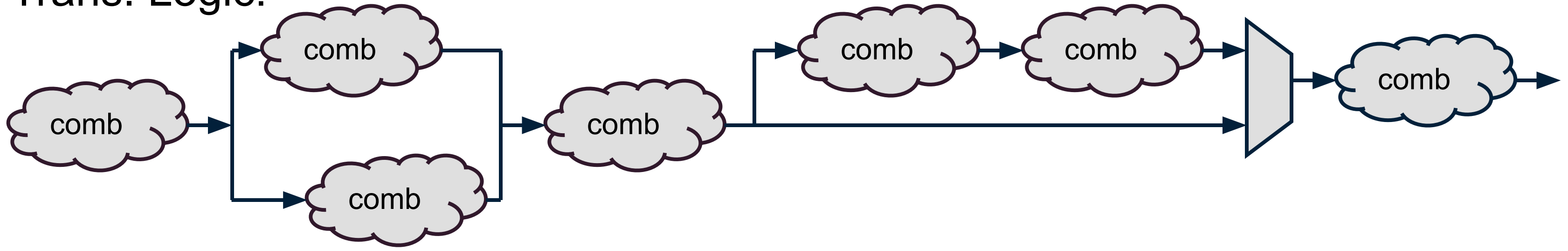
- Flow constructed from pre-verified library components.
- Transaction logic added into this context.

Transaction Flow

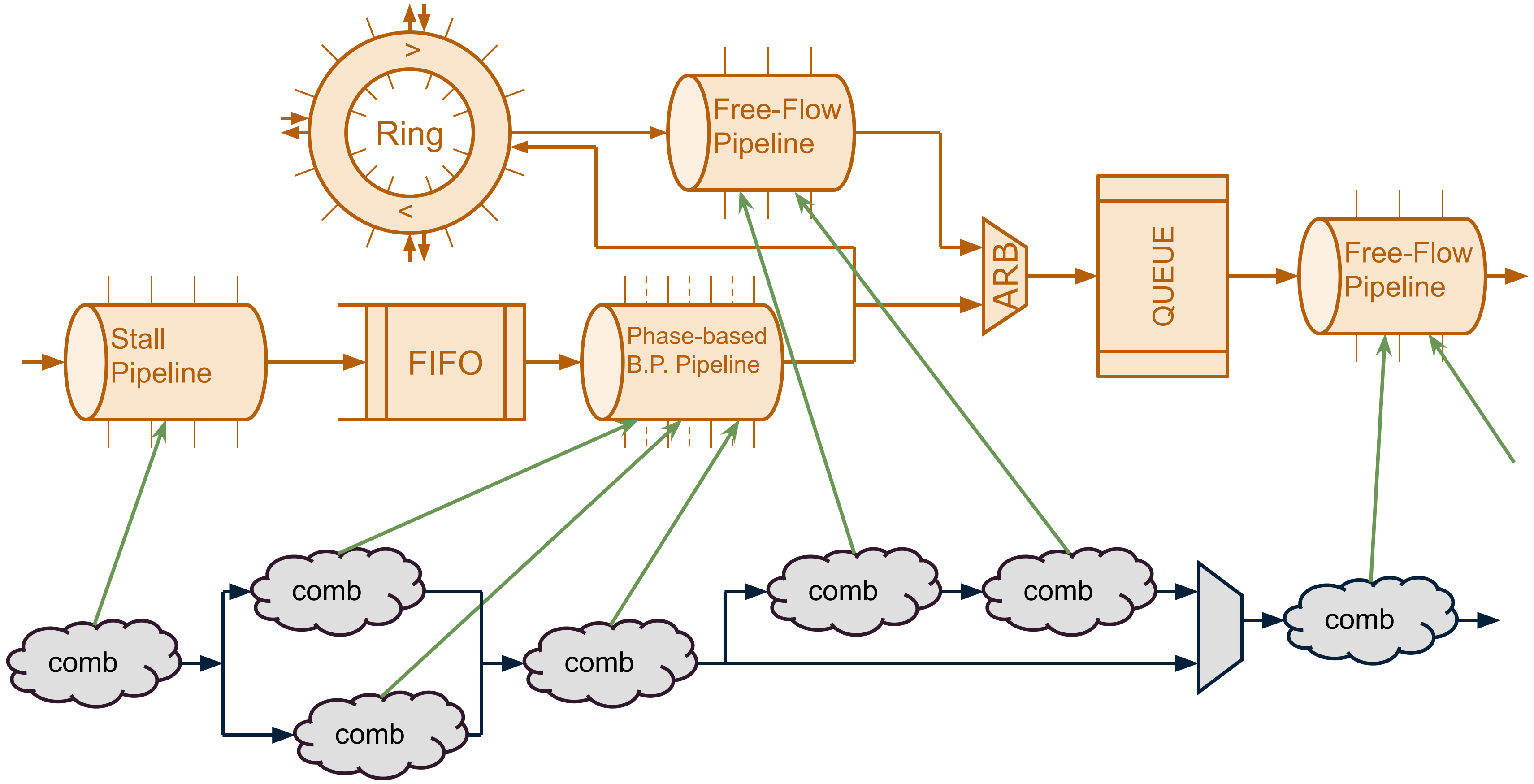
Trans. Flow:



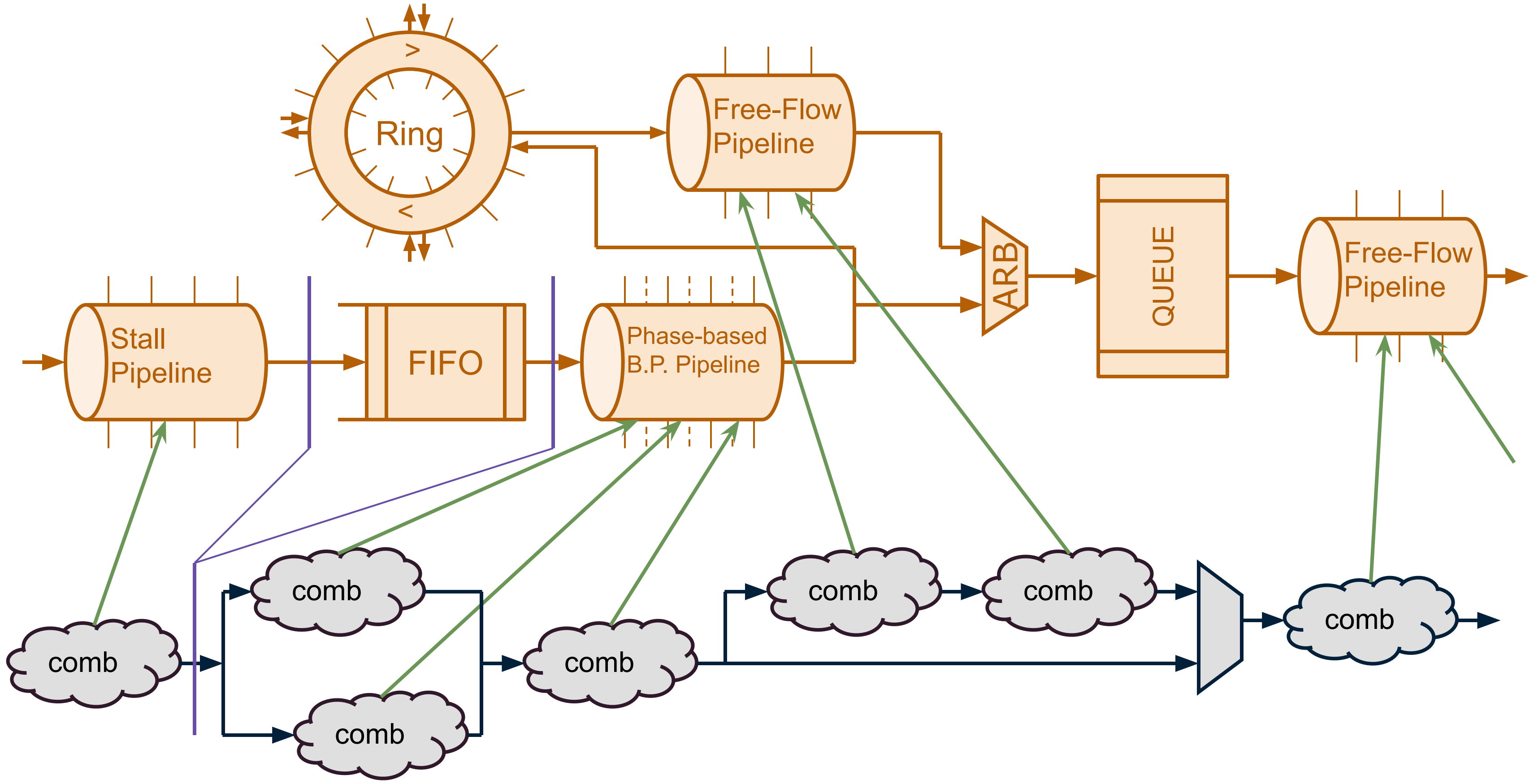
Trans. Logic:



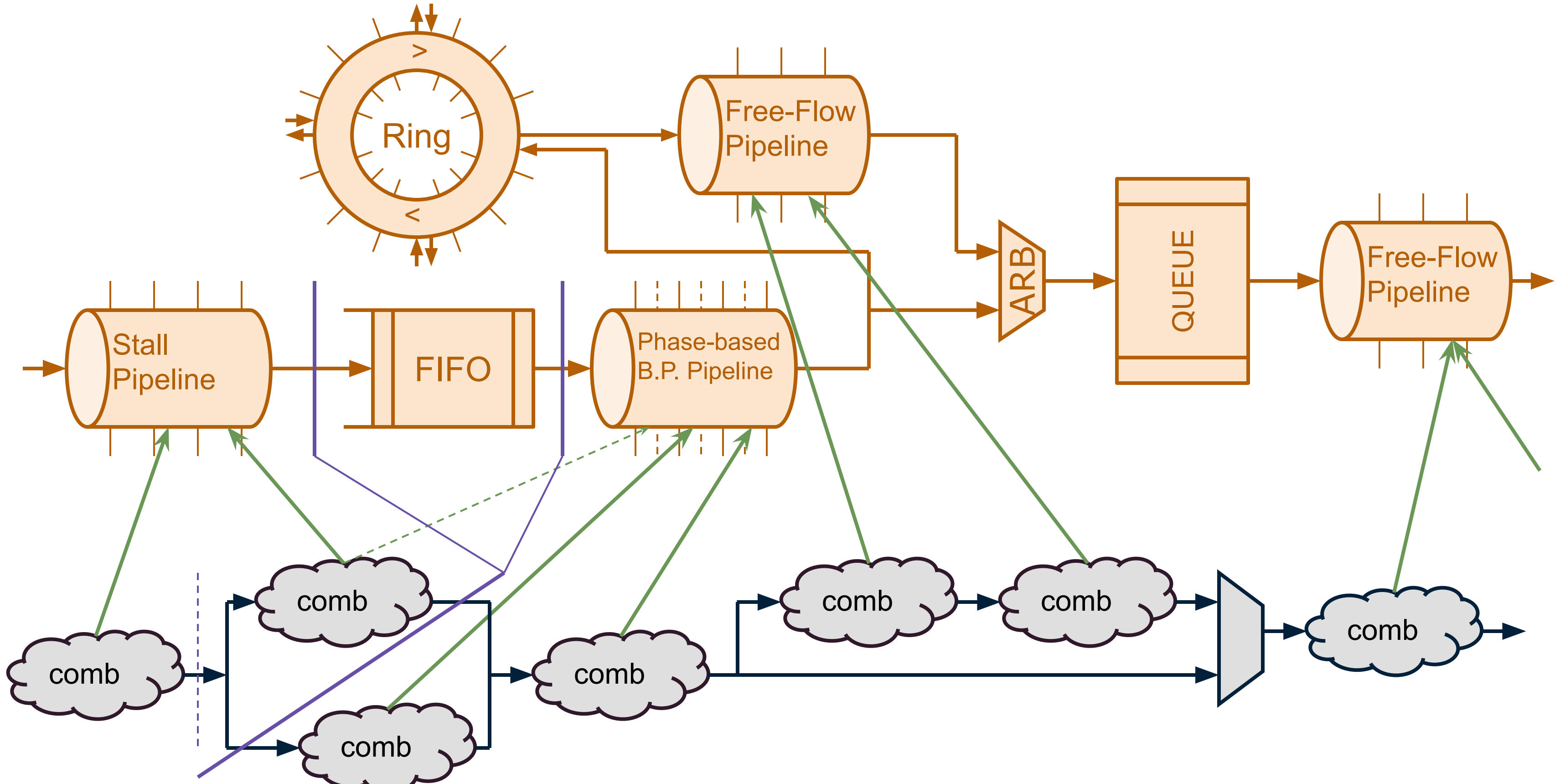
Transaction Flow



Transaction Flow: Retiming

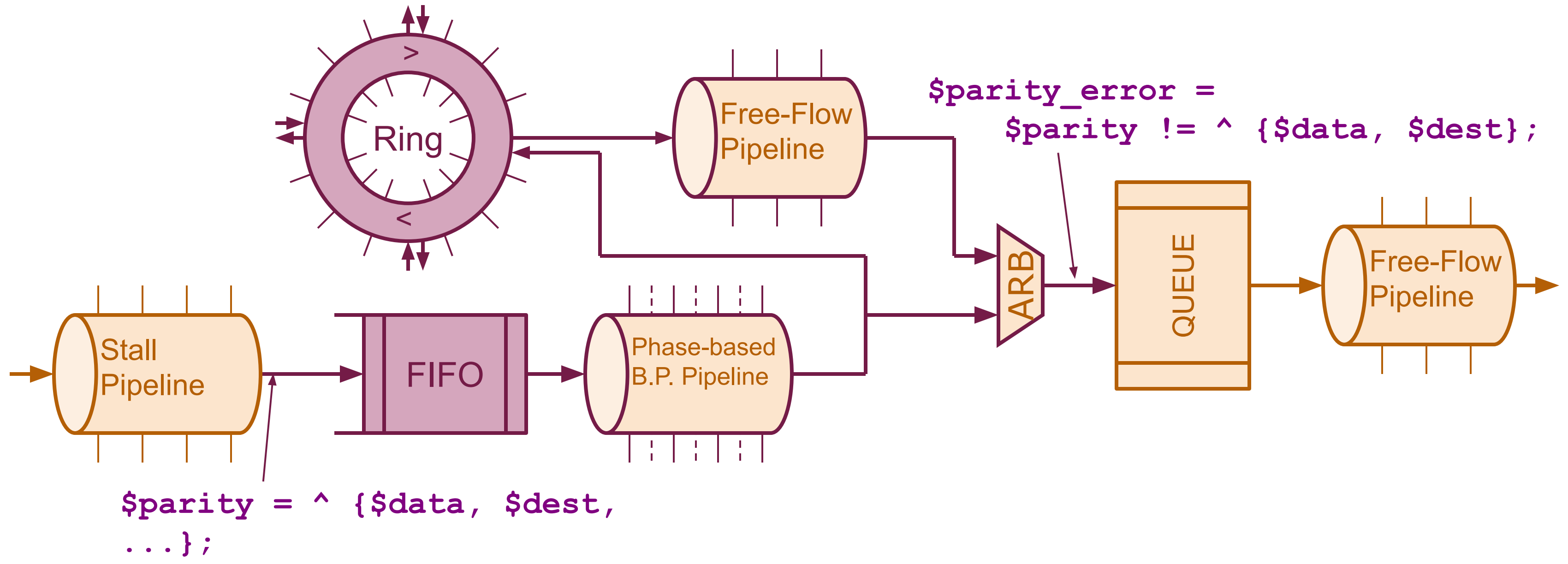


Transaction Flow: Retiming



Transaction Flow: Scenario

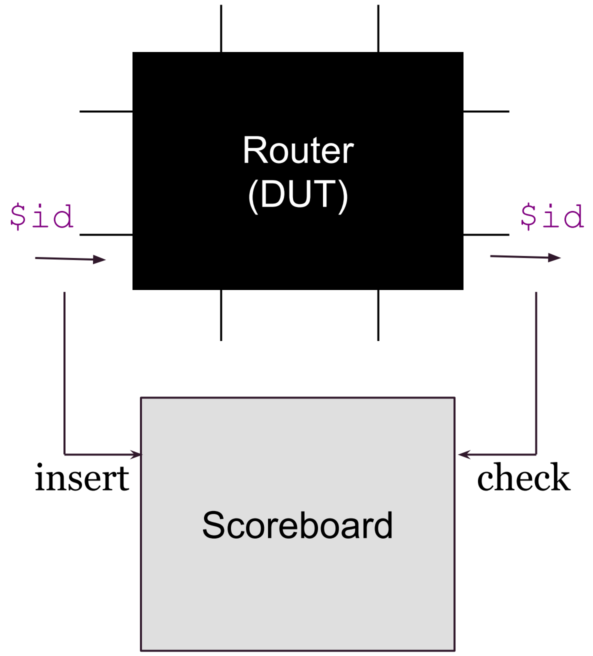
Error rate too high. Require parity protection on FIFO and Ring.



Ring
>
<
Free-Flow Pipeline
$parity_error = $parity != ^ {$data, $dest};
ARB
QUEUE
Free-Flow Pipeline
Stall Pipeline
FIFO
Phase-based B.P. Pipeline
$parity = ^ {$data, $dest, ...};

2 lines of TL-Verilog vs. 100s of lines of RTL change (across files).

Router Verification



Goal:

Verify each out-going packet (black-box).

Approach:

Keep a Scoreboard to store and retrieve packet info for checking.

Problem:

How can we match inbound w/ outbound?

Packets have no ID#.

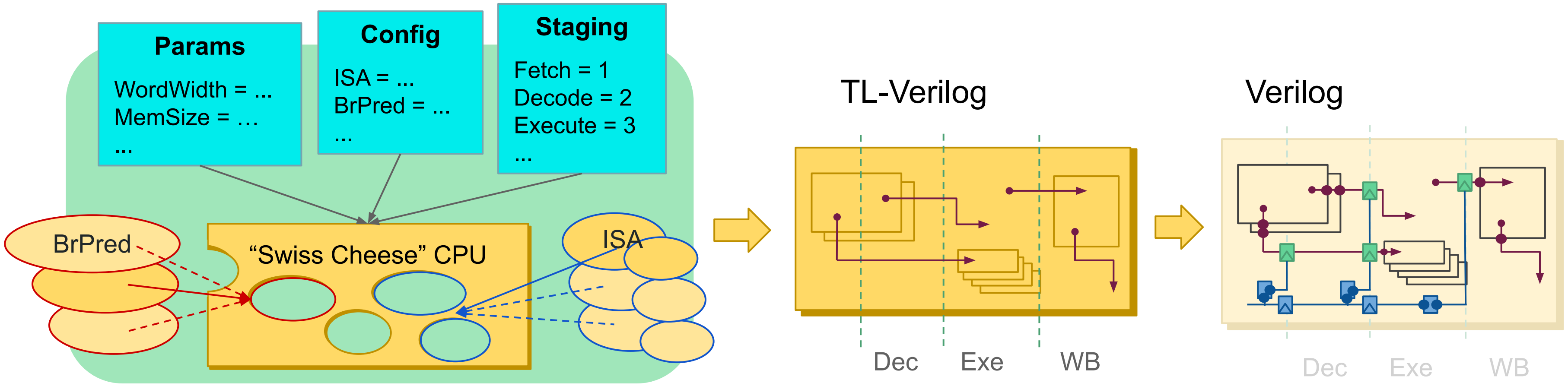
Solution:

Give them an ID# (Scoreboard index). Duh.
No need to touch DUT code.

Lab: (Modified from Video) Flow Tutorial

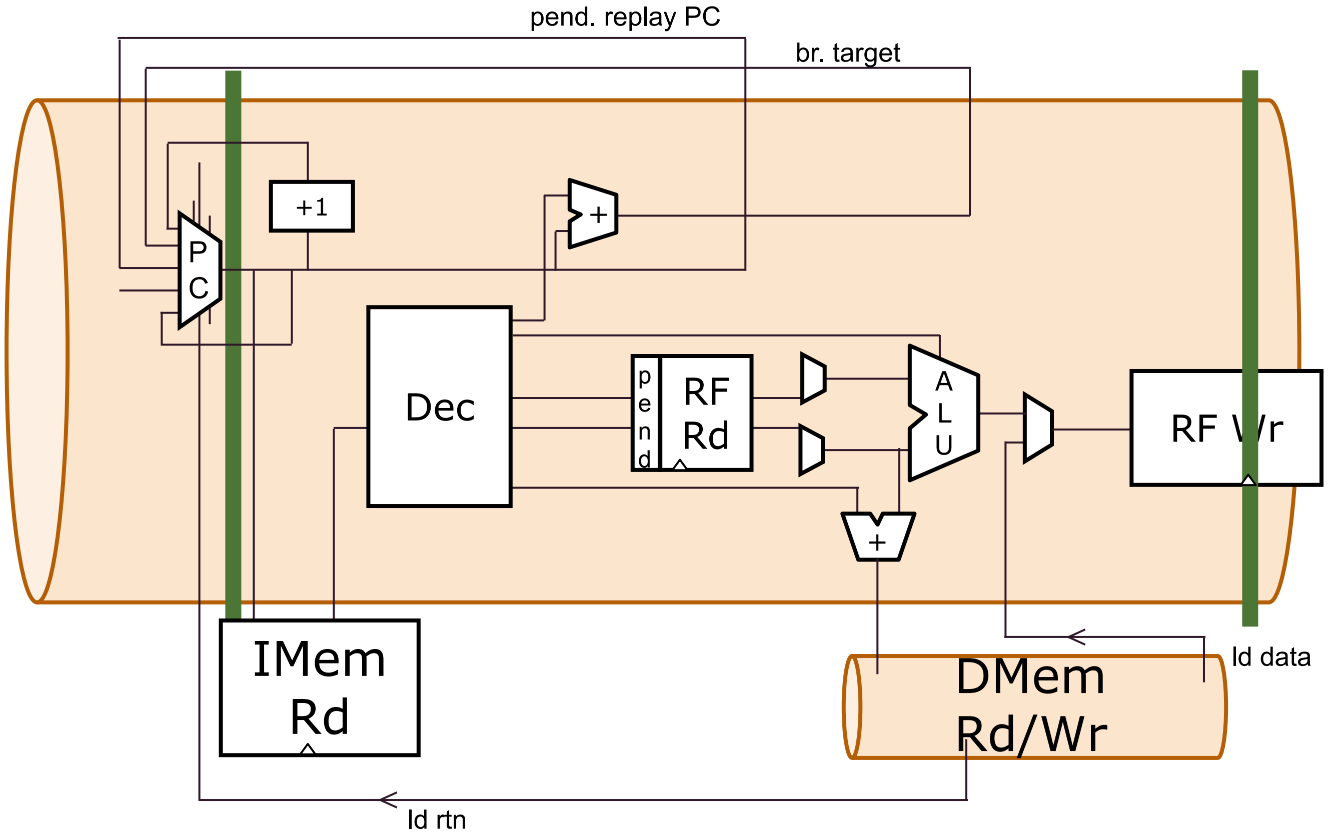
Do the “Flow Tutorial” in the Makerchip IDE. This tutorial is similar to the `$parity` example from slide 40, but using a simpler ring architecture.

WARP-V

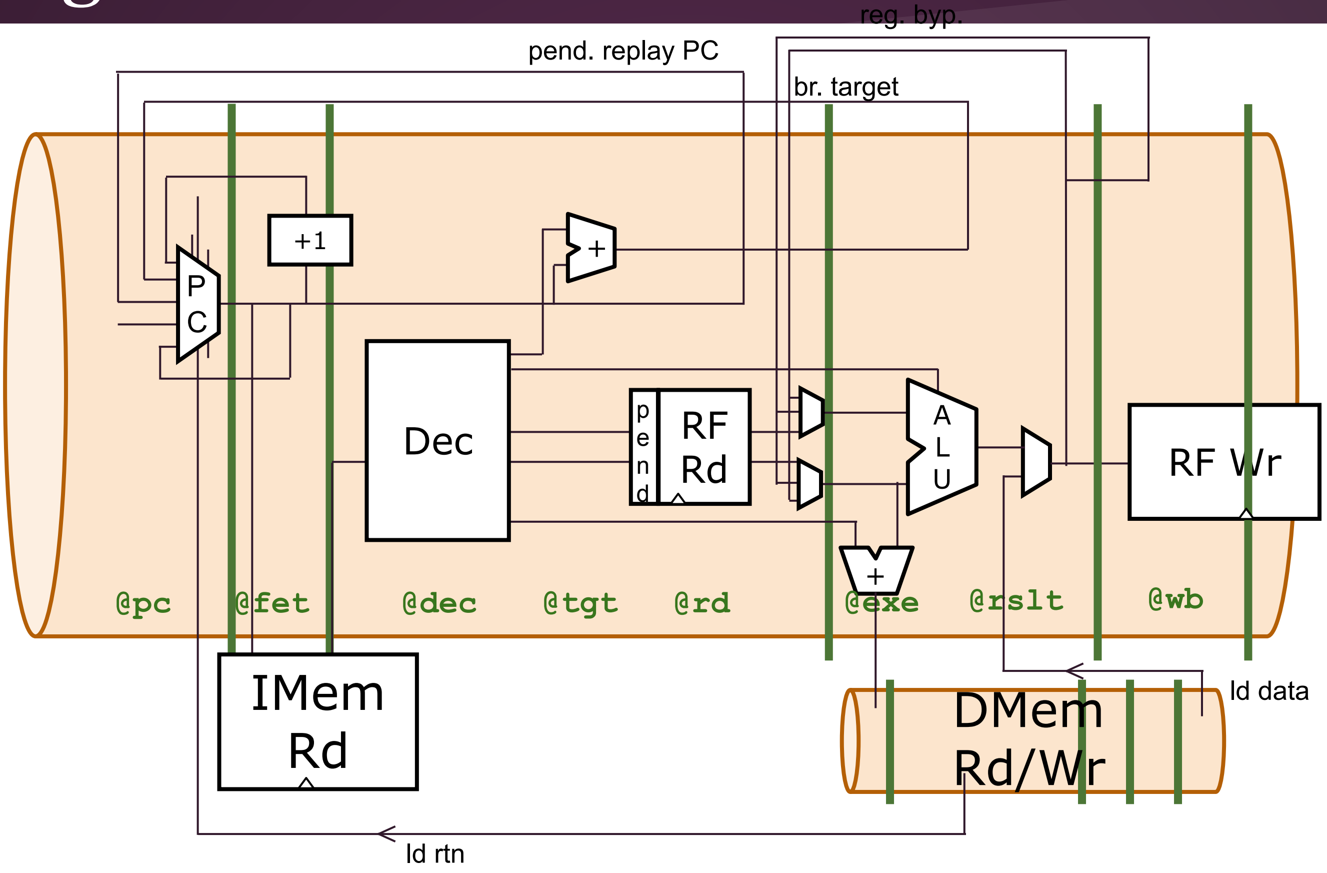


Params
WordWidth = ...
MemSize = …
...
Config
ISA = ...
BrPred = ...
...
Staging
Fetch = 1
Decode = 2
Execute = 3
...
BrPred
"Swiss Cheese" CPU
ISA
TL-Verilog
Dec
Exe
WB
Verilog
Dec
Exe
WB

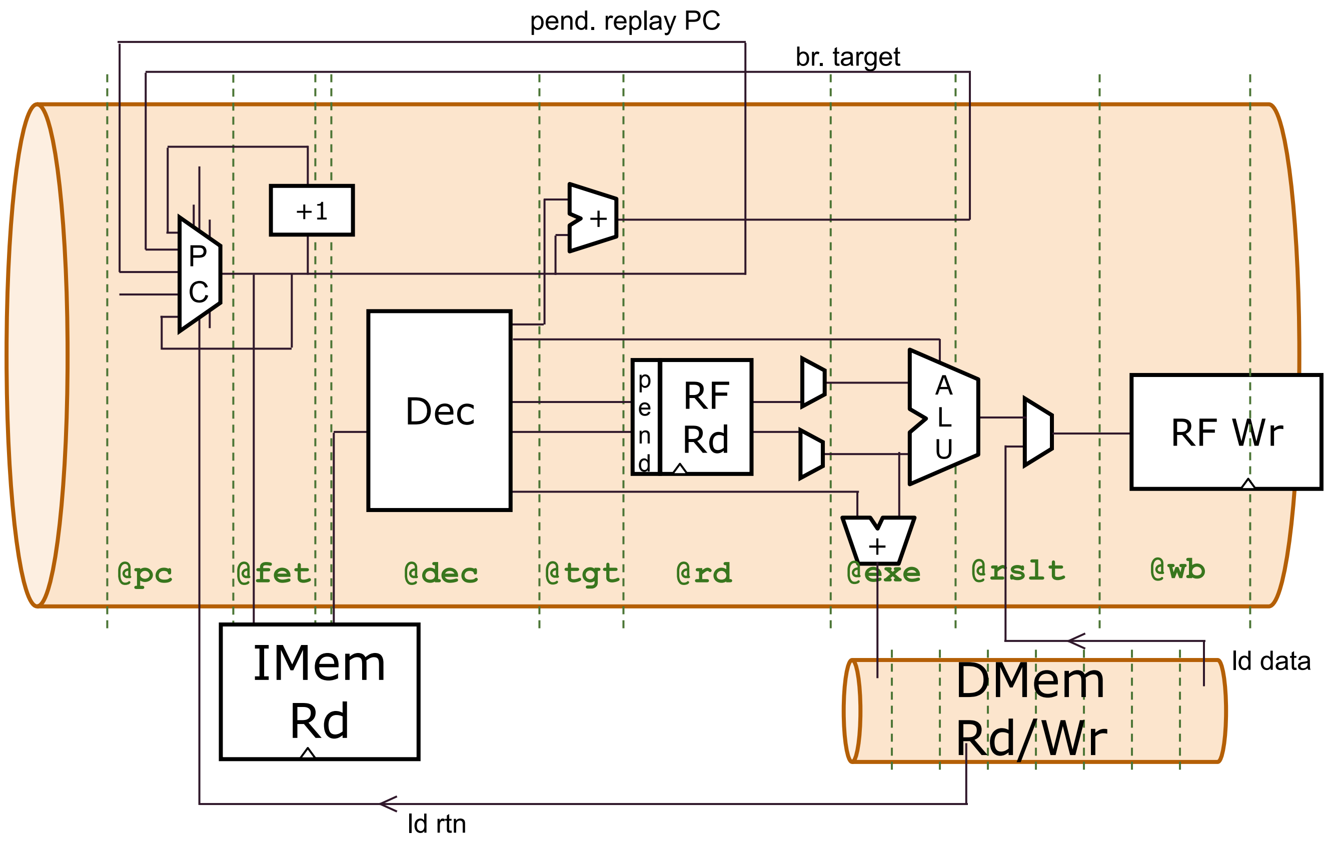
1-Stage WARP-V



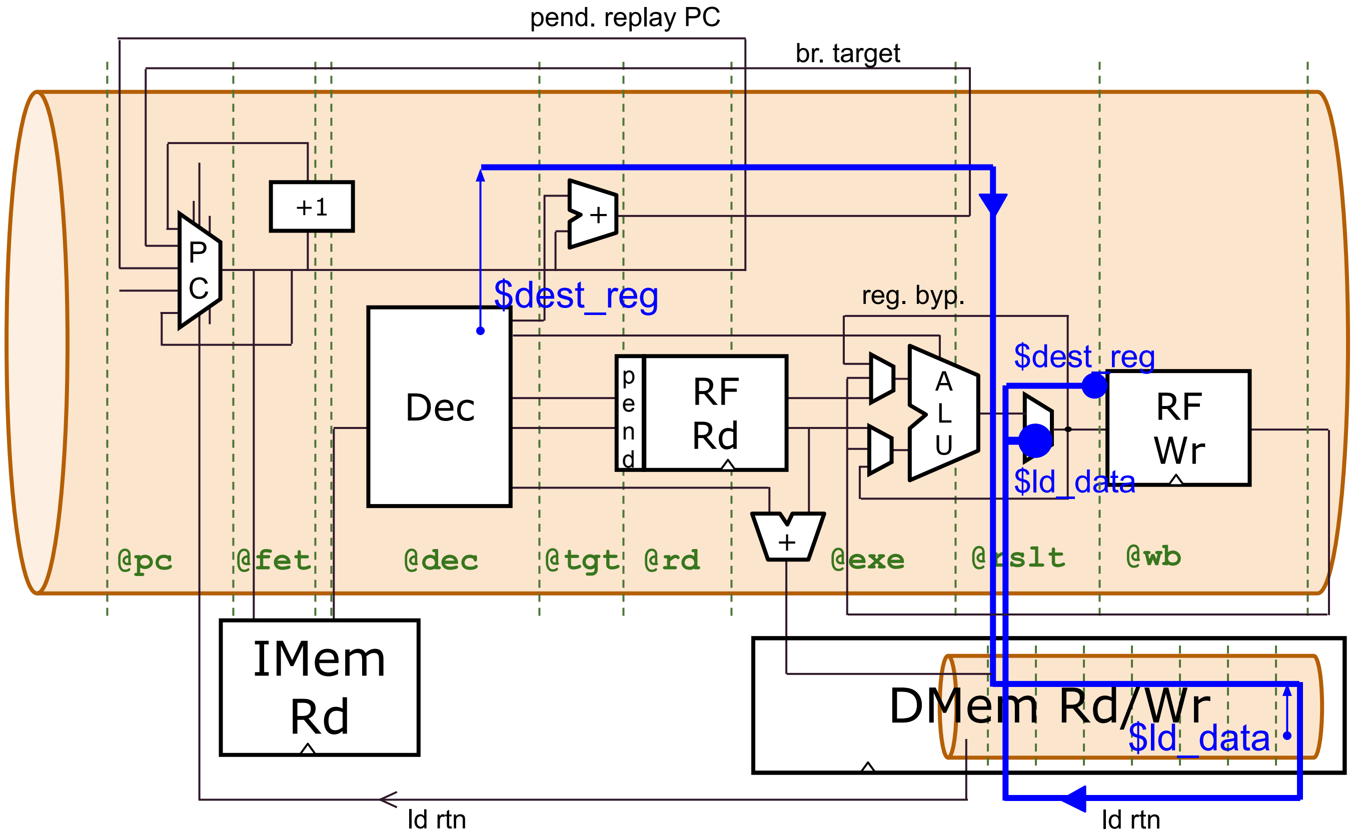
4-Stage WARP-V



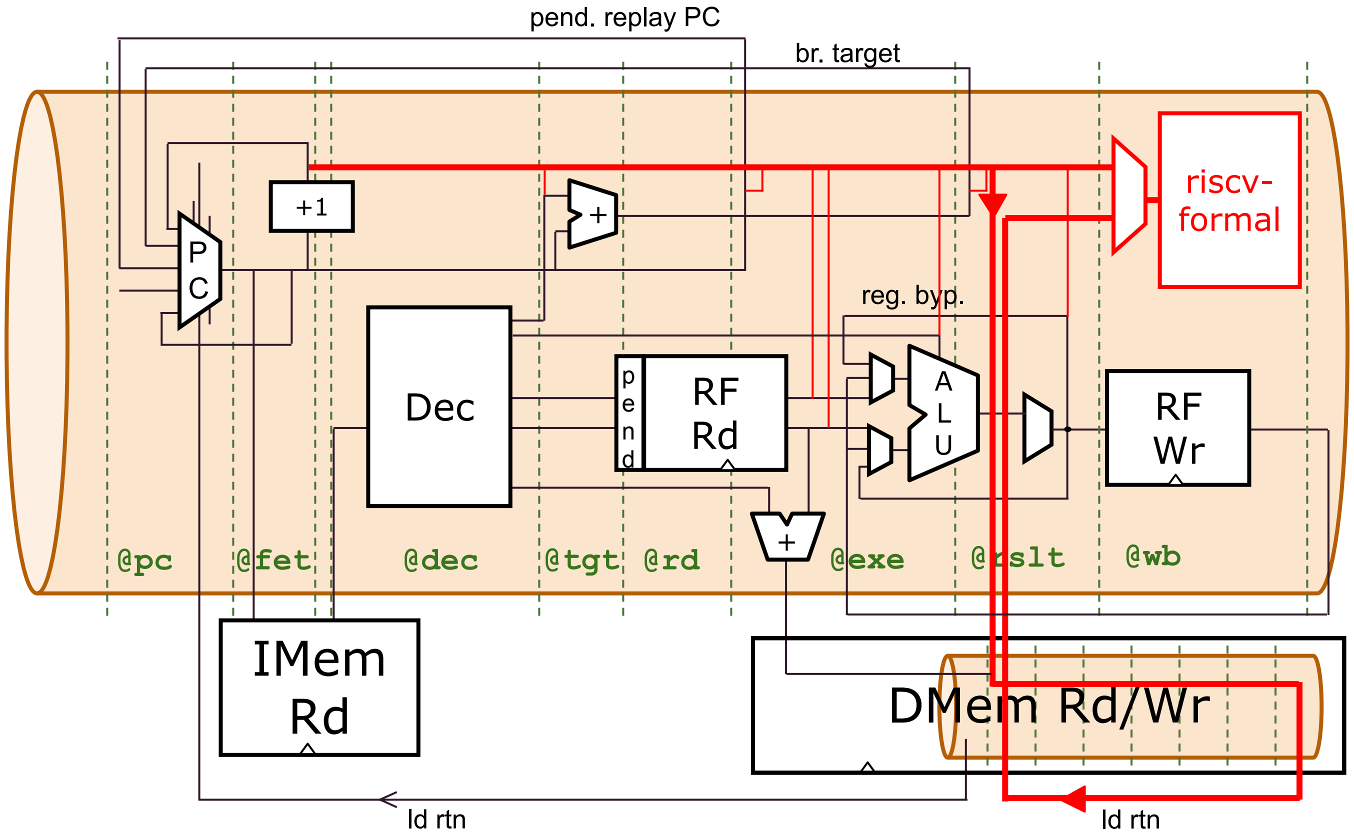
N-Stage WARP-V



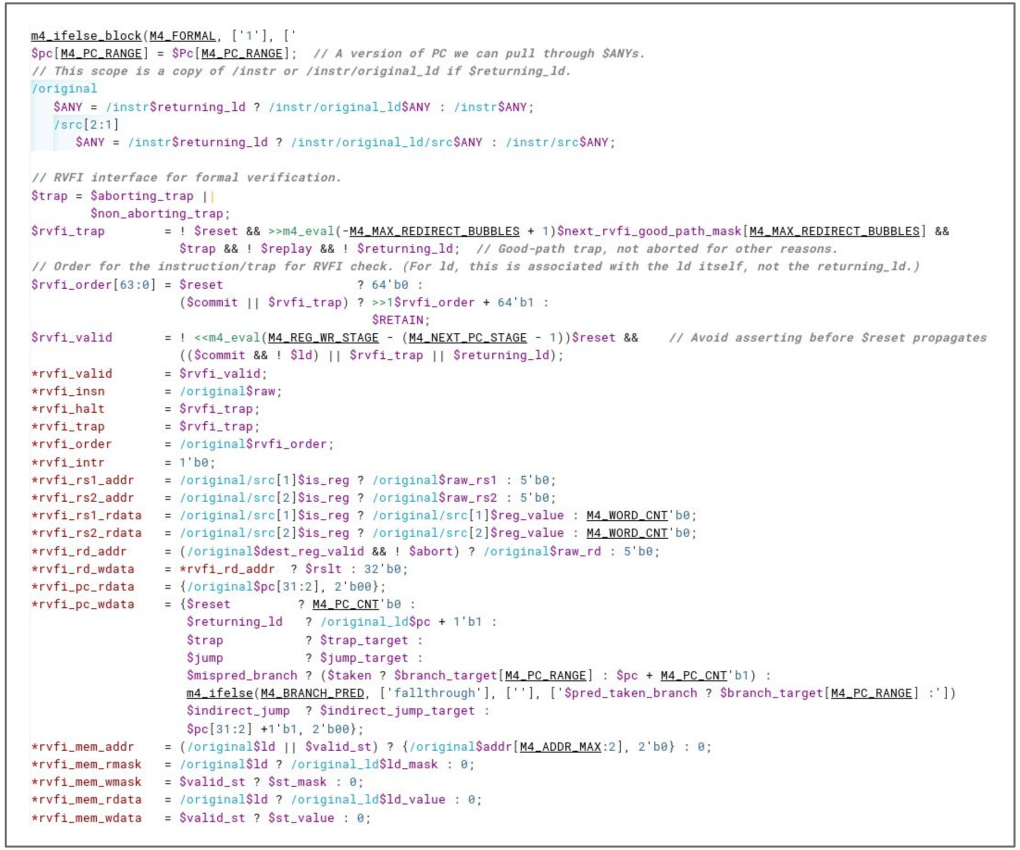
WARP-V Load Recirculation



WARP-V Formal Verification

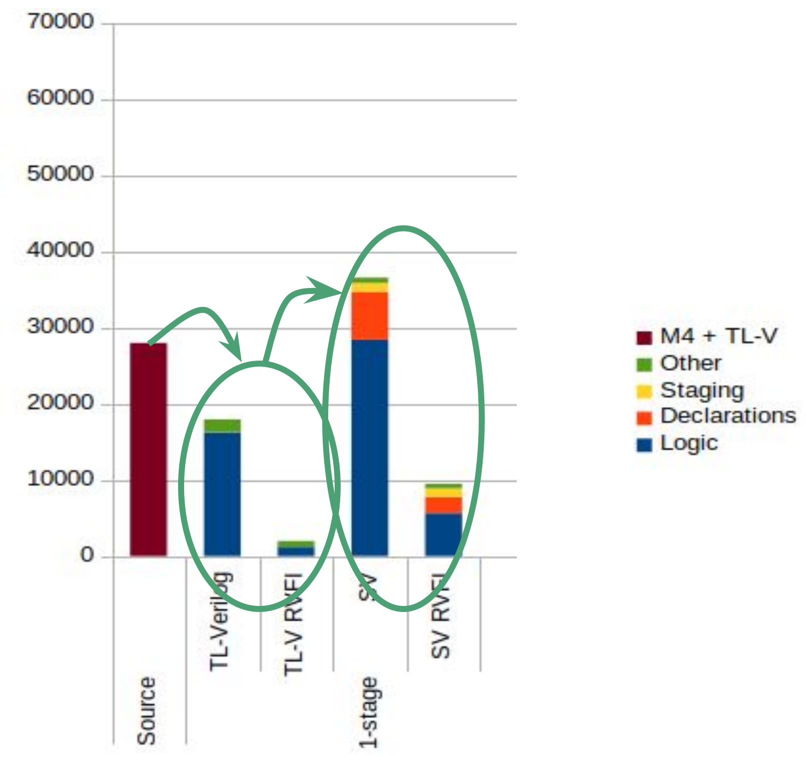


1 Page

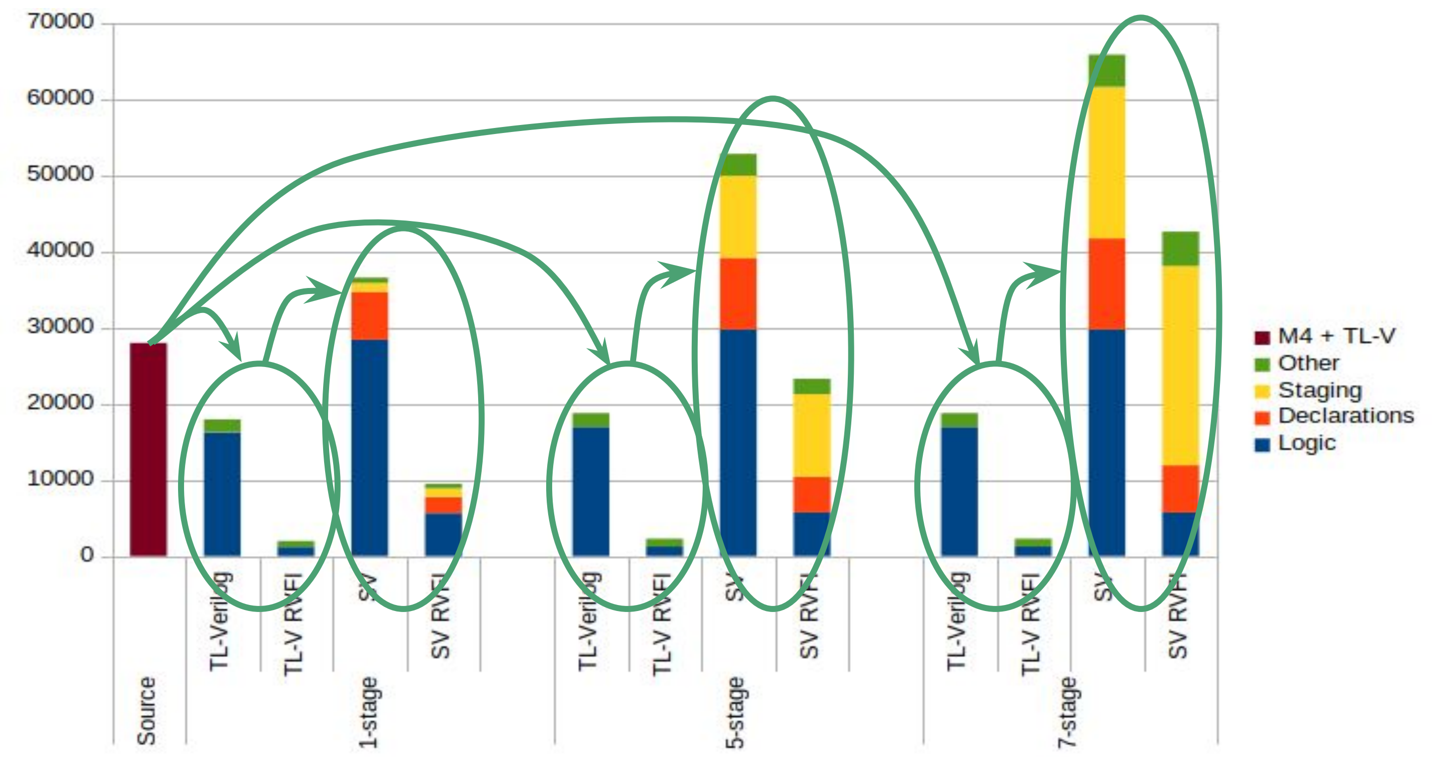


```
m4_ifelse_block(M4_FORMAL, ['1'], ['
$pc[M4_PC_RANGE] = $Pc[M4_PC_RANGE];  // A version of PC we can pull through $ANYs.
// This scope is a copy of /instr or /instr/original_ld if $returning_ld.
/original
   $ANY = /instr$returning_ld ? /instr/original_ld$ANY : /instr$ANY;
   /src[2:1]
      $ANY = /instr$returning_ld ? /instr/original_ld/src$ANY : /instr/src$ANY;

// RVFI interface for formal verification.
$trap = $aborting_trap ||
        $non_aborting_trap;
$rvfi_trap        = ! $reset && >>m4_eval(-M4_MAX_REDIRECT_BUBBLES + 1)$next_rvfi_good_path_mask[M4_MAX_REDIRECT_BUBBLES] &&
                    $trap && ! $replay && ! $returning_ld;  // Good-path trap, not aborted for other reasons.
// Order for the instruction/trap for RVFI check. (For ld, this is associated with the ld itself, not the returning_ld.)
$rvfi_order[63:0] = $reset                    ? 64'b0 :
                    ($commit || $rvfi_trap) ? >>1$rvfi_order + 64'b1 :
                                              $RETAIN;
$rvfi_valid       = ! <<m4_eval(M4_REG_WR_STAGE - (M4_NEXT_PC_STAGE - 1))$reset &&    // Avoid asserting before $reset propagates
                    (($commit && ! $ld) || $rvfi_trap || $returning_ld);
*rvfi_valid       = $rvfi_valid;
*rvfi_insn        = /original$raw;
*rvfi_halt        = $rvfi_trap;
*rvfi_trap        = $rvfi_trap;
*rvfi_order       = /original$rvfi_order;
*rvfi_intr        = 1'b0;
*rvfi_rs1_addr    = /original/src[1]$is_reg ? /original$raw_rs1 : 5'b0;
*rvfi_rs2_addr    = /original/src[2]$is_reg ? /original$raw_rs2 : 5'b0;
*rvfi_rs1_rdata   = /original/src[1]$is_reg ? /original/src[1]$reg_value : M4_WORD_CNT'b0;
*rvfi_rs2_rdata   = /original/src[2]$is_reg ? /original/src[2]$reg_value : M4_WORD_CNT'b0;
*rvfi_rd_addr     = (/original$dest_reg_valid && ! $abort) ? /original$raw_rd : 5'b0;
*rvfi_rd_wdata    = *rvfi_rd_addr  ? $rslt : 32'b0;
*rvfi_pc_rdata    = {/original$pc[31:2], 2'b00};
*rvfi_pc_wdata    = {$reset          ? M4_PC_CNT'b0 :
                     $returning_ld   ? /original_ld$pc + 1'b1 :
                     $trap           ? $trap_target :
                     $jump           ? $jump_target :
                     $mispred_branch ? ($taken ? $branch_target[M4_PC_RANGE] : $pc + M4_PC_CNT'b1) :
                     m4_ifelse(M4_BRANCH_PRED, ['fallthrough'], [''], ['$pred_taken_branch ? $branch_target[M4_PC_RANGE] :'])
                     $indirect_jump  ? $indirect_jump_target :
                     $pc[31:2] +1'b1, 2'b00};
*rvfi_mem_addr    = (/original$ld || $valid_st) ? {/original$addr[M4_ADDR_MAX:2], 2'b0} : 0;
*rvfi_mem_rmask   = /original$ld ? /original_ld$ld_mask : 0;
*rvfi_mem_wmask   = $valid_st ? $st_mask : 0;
*rvfi_mem_rdata   = /original$ld ? /original_ld$ld_value : 0;
*rvfi_mem_wdata   = $valid_st ? $st_value : 0;
```

Code Comparison



Code Comparison



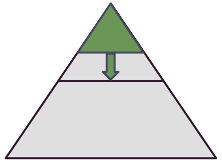
Summary

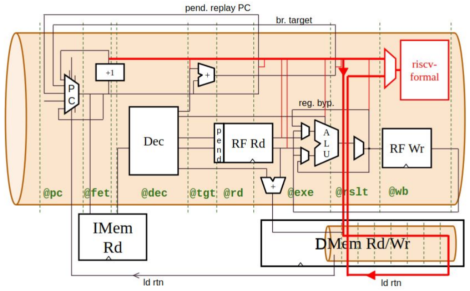
TL-Verilog models are easier to verify because...

- Designs have fewer bugs.
  - Less code == fewer bugs.
  - The complex parts are from pre-verified libraries.
- Test harness development can be automated.
- Verification models remain valid
  - through design changes
  - for multiple implementations

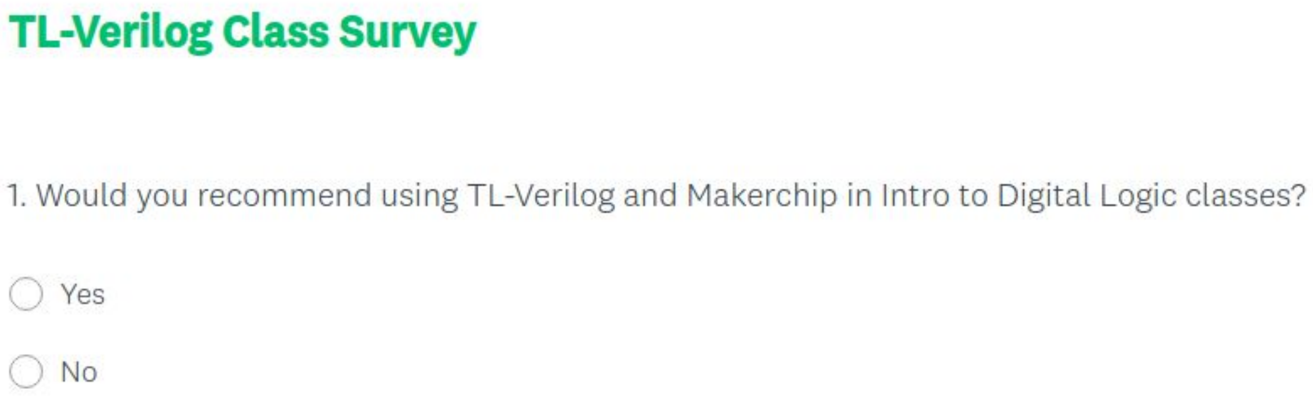
TL-Verilog is good for verification models because...

- It is 100% synthesizable.
- Same language and tools for logic and verif.





Survey



TL-Verilog Class Survey

1. Would you recommend using TL-Verilog and Makerchip in Intro to Digital Logic classes?

○ Yes

○ No