Transaction-Level Verilog and its Ecosystem

Steve Hoover
Founder, Redwood EDA
Dec. 15, 2020

Agenda

- Makerchip demo
- The role of abstraction in ASIC/FPGA design
- Hands-on learning
  - Combinational logic
  - Sequential logic
  - Pipelines
- Wrap-up

Makerchip.com

makerchip beta PROJECT TUTORIALS HELP SOCIAL unsaved SAVE AS NEW PROJECT

COURSE SLIDES | TUTORIAL-VALID

Lab 1 - Makerchip Platform

Reproduce this screensho

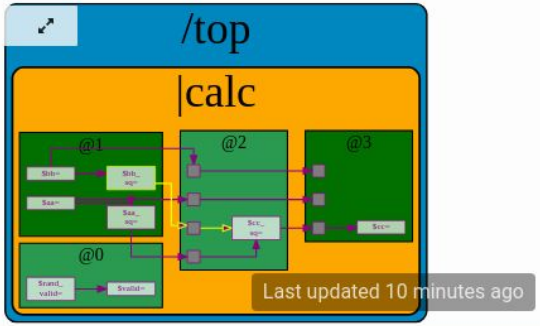
1. Open "Tutorials" "Validity Tutorial".
2. In tutorial, click Load Pythagorean Example
3. Split panes ( ) and move tabs.
4. Zoom/pan in Diagrar w/ mouse wheel and drag.

EDITOR | NAV-TLV

```
15  |calc
16     // [+] ?$valid
17     @1                                  // [>>>]
18        $aa_sq[7:0] = $aa[3:0] ** 2;     // [>>>]
19        $bb_sq[7:0] = $bb[3:0] ** 2;     // [>>>]
20     @2                                  // [>>>]
21        $cc_sq[8:0] = $aa_sq + $bb_sq;   // [>>>]
22     @3                                  // [>>>]
```

Last updated 10 minutes ago

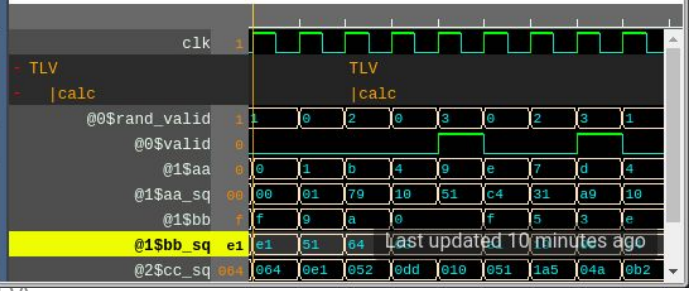
DIAGRAM



Last updated 10 minutes ago

LOG | WAVEFORM

ZOOM IN ZOOM OUT ZOOM FULL << >>



Last updated 10 minutes ago

RTL

Year	Processor	Clock	Transistors	HDL
1985	i386	?	?	Verilog (to verify)
2017	32-core AMD Epyc	?	?	Verilog

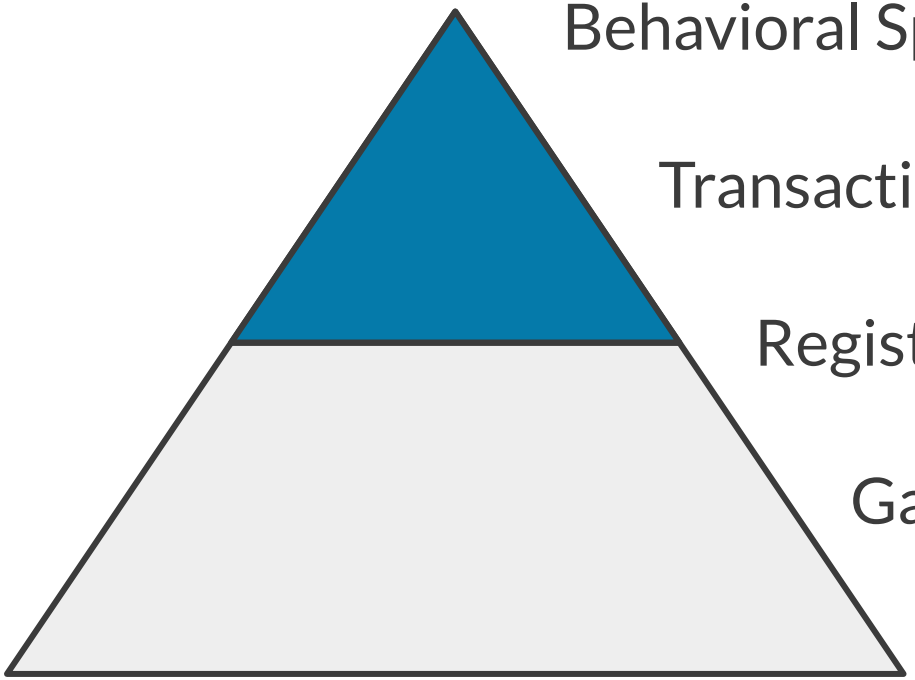
RTL

Year	Processor	Clock	Transistors	HDL
1985	i386	33MHz	?	Verilog (to verify)
2017	32-core AMD Epyc	3.2GHz (**~100x**)	?	Verilog

RTL

Year	Processor	Clock	Transistors	HDL
1985	i386	33MHz	275K	Verilog (to verify)
2017	32-core AMD Epyc	3.2GHz (~100x)	19.2B (**>70,000x**)	Verilog

Abstraction Levels



Behavioral Spec. (e.g. ISA)

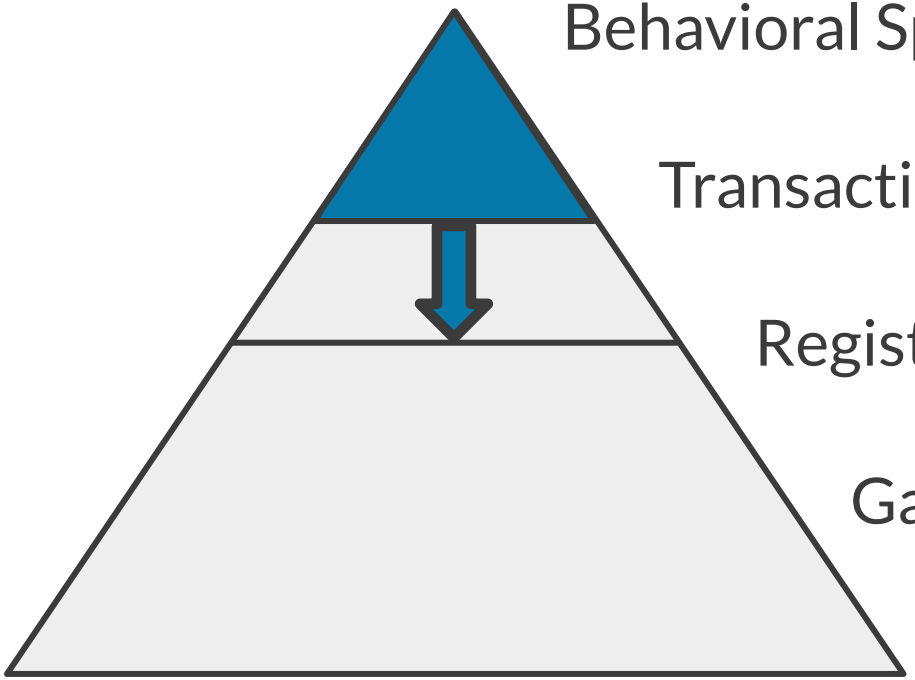
Transaction Level

Register Transfer Level

Gate Level

Device Level

Abstraction Levels



Behavioral Spec. (e.g. ISA)

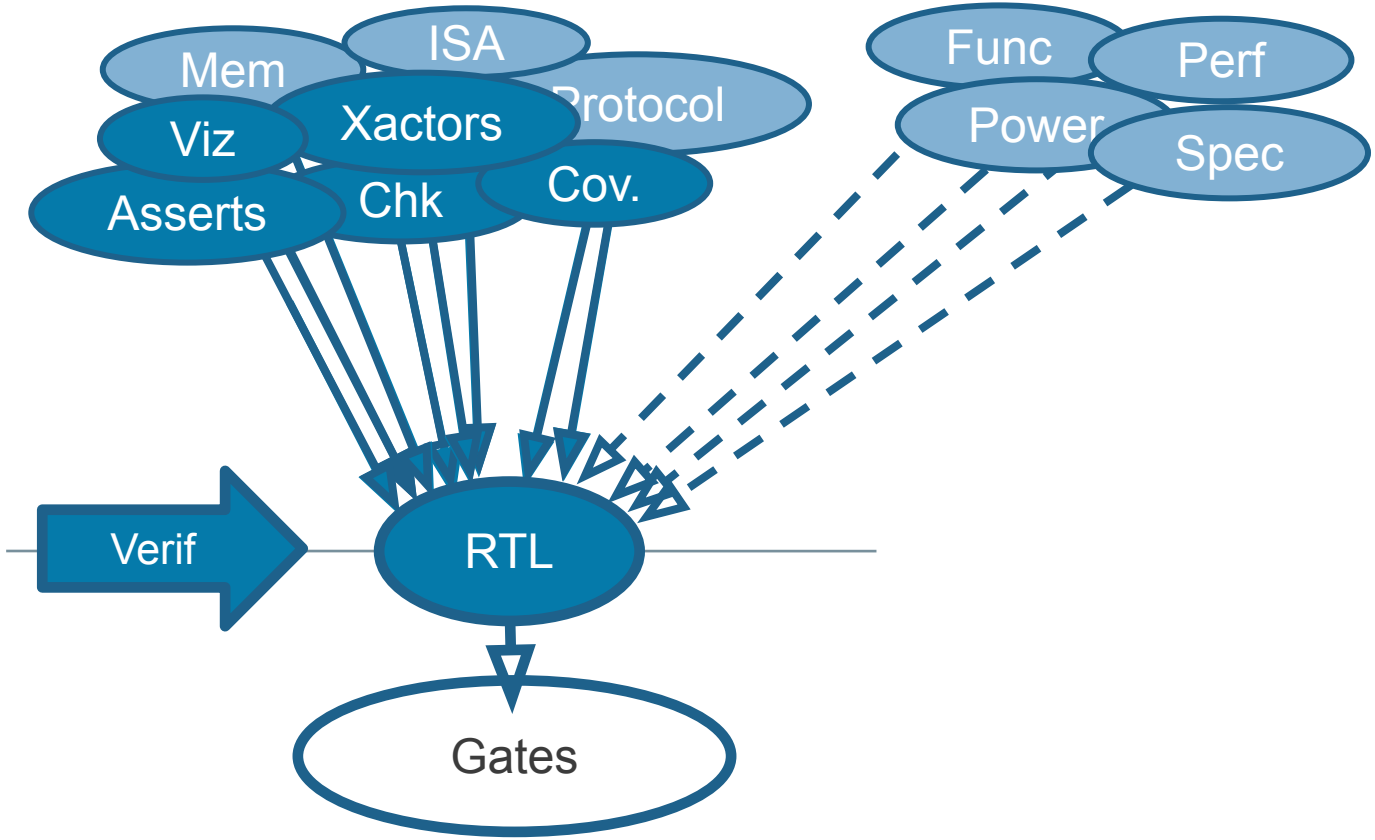
Transaction Level

Register Transfer Level

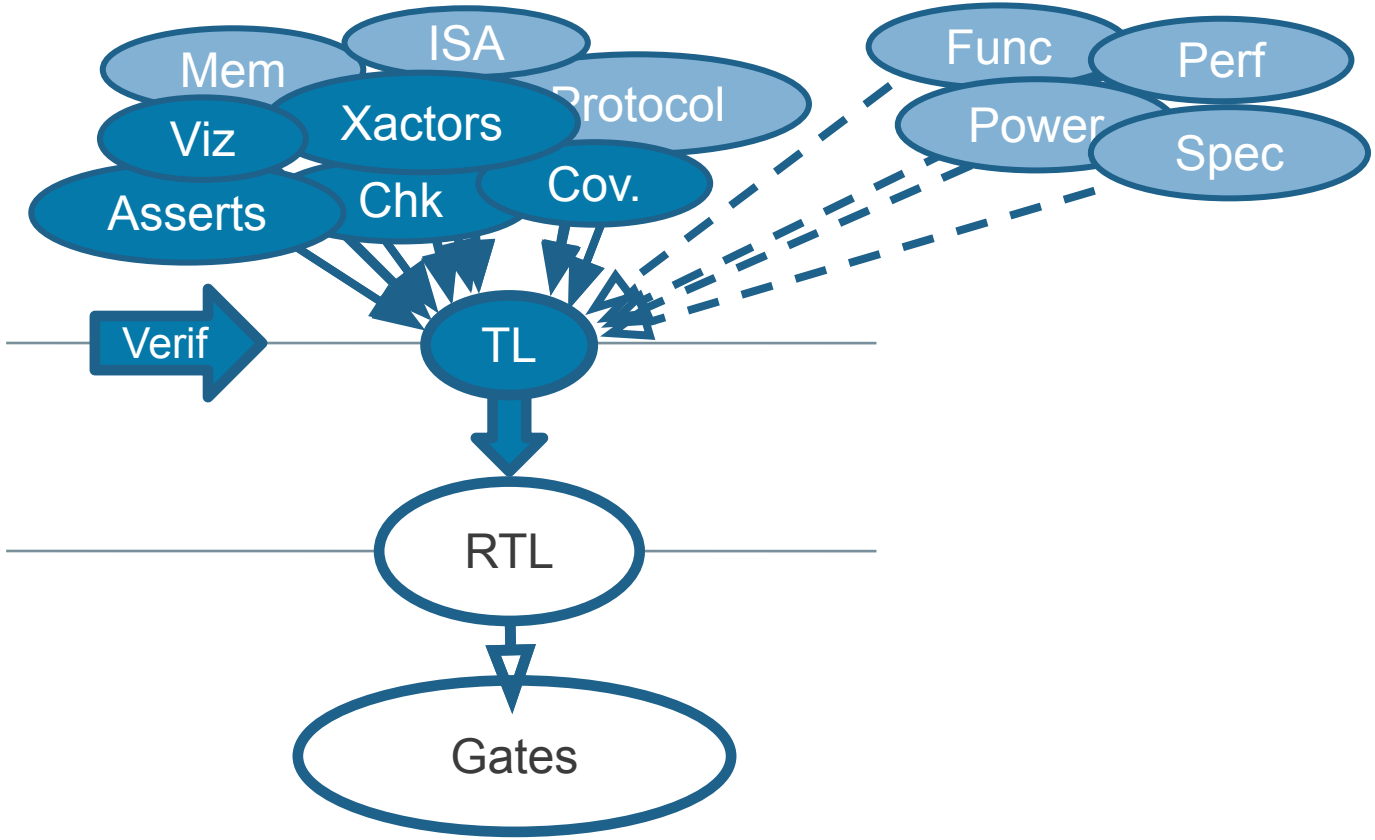
Gate Level

Device Level

RTL Design Methodology



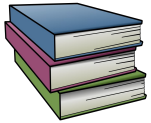
Transaction-Level Design Methodology



Alternate 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.)

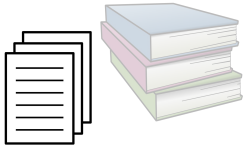


Academia: DSLs (Chisel, CλaSH, ...)

- 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)



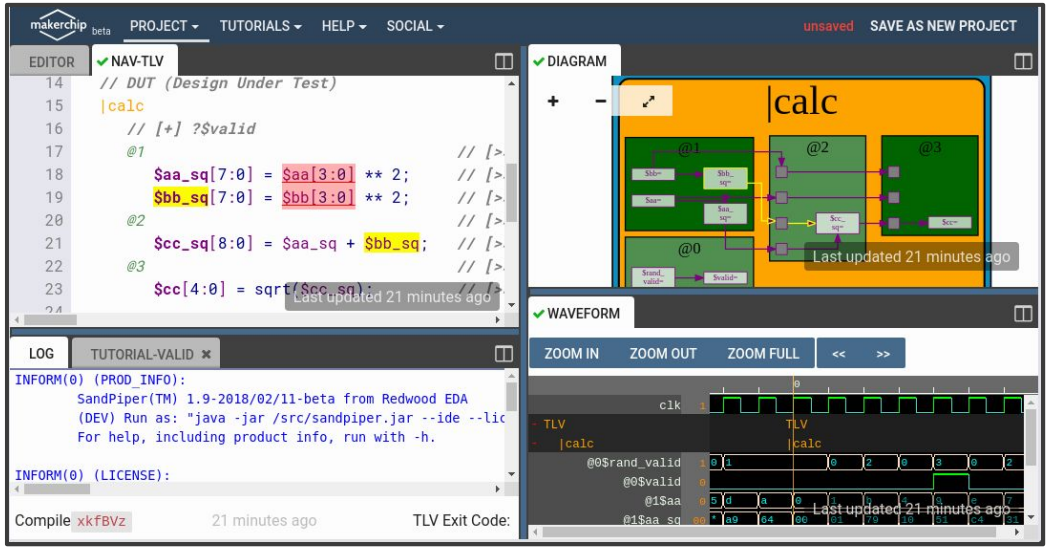
Lab: Makerchip Platform

Reproduce this screenshot:

1. Open: barc.makerchip.com.
2. Click “IDE”.
3. Open “Tutorials” “Validity Tutorial”.
4. In tutorial, click:

Load Pythagorean Example

5. Split panes and move tabs.



7. Zoom/pan in Diagram w/ mouse wheel and drag.
8. Zoom Waveform w/ “Zoom In” button.
9. 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

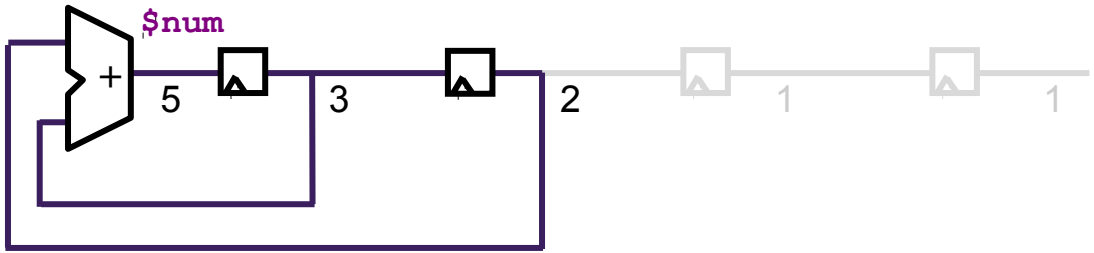
`$out[4:0]` creates a “vector” of 5 bits.

Arithmetic operators operate on vectors as binary numbers.

1. Try:
   ```
   $out[4:0] = $in1[3:0] + $in2[3:0];
   ```
2. View Waveform (values are in hexadecimal)

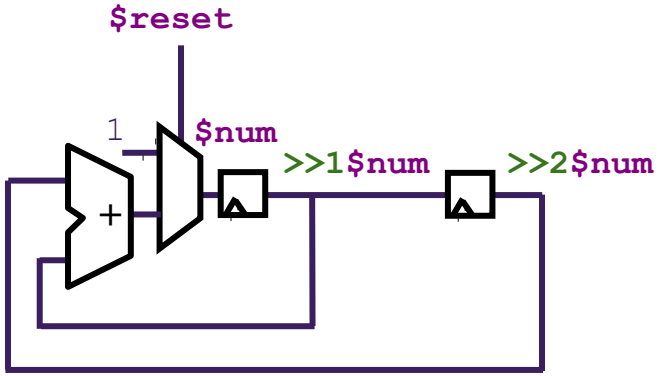
Sequential Logic - Fibonacci Series

Next value is sum of previous two: 1, 1, 2, 3, 5, 8, 13, ...



Fibonacci Series - Reset

Next value is sum of previous two: 1, 1, 2, 3, 5, 8, 13, ...



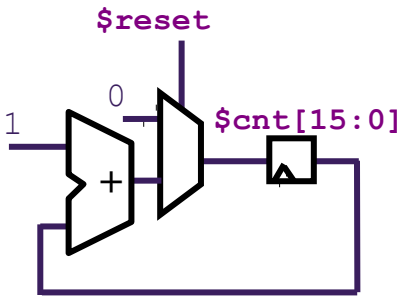

```
$num[31:0] = $reset ? 1 : (>>1$num + >>2$num);
```

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

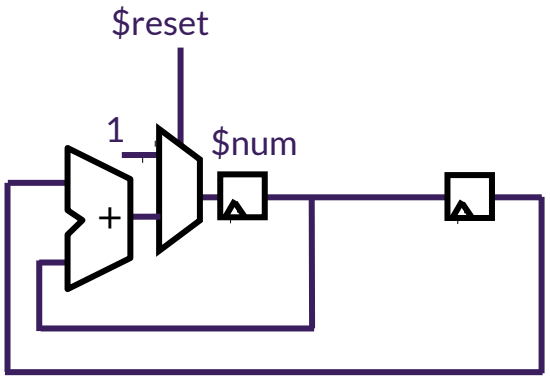
Lab: Counter

Lab:

1. Design a free-running counter:



Reference Example: Fibonacci Sequence (1, 1, 2, 3, 5, 8, ...)



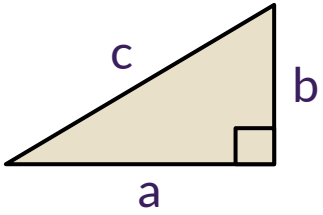

```
\TLV
   $num[31:0] = $reset ? 1 : (>>1$num + >>2$num);
```

3-space indentation
(no tabs)

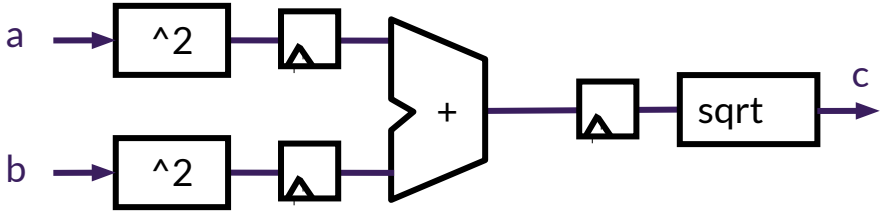
(makerchip.com/sandbox/0/0wjhLP)

A Simple Pipeline

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

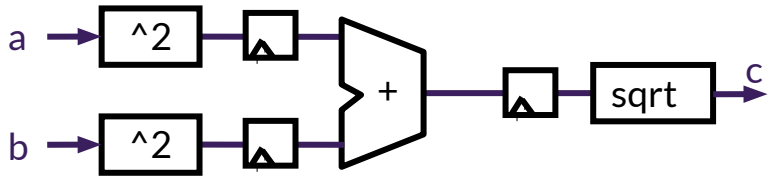


c = sqrt(a^2 + b^2)

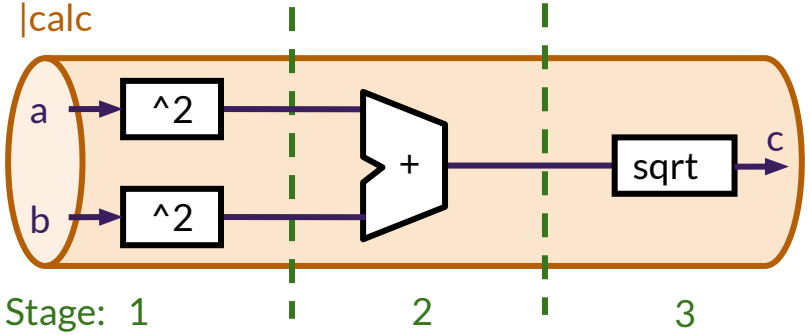


A Simple Pipeline - Timing-Abstract

RTL:

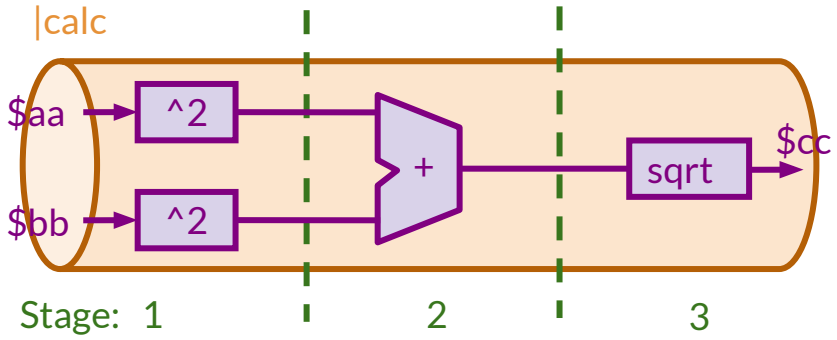


Timing-abstract:



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

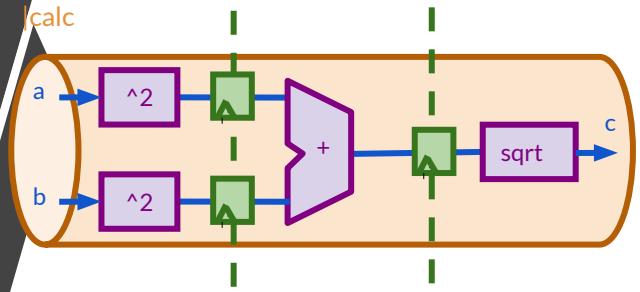
A Simple Pipeline - TL-Verilog



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

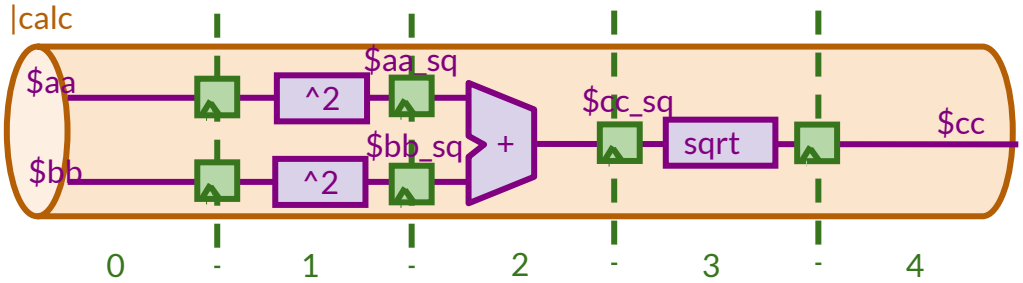
TL-Verilog

```
|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);
```

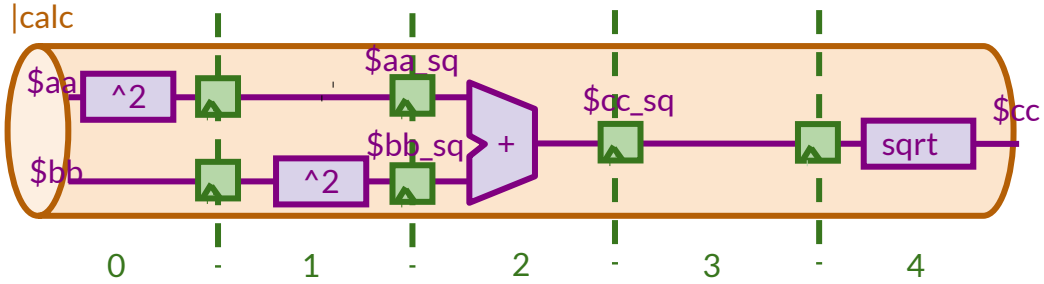
Retiming -- Easy and Safe

```
|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.

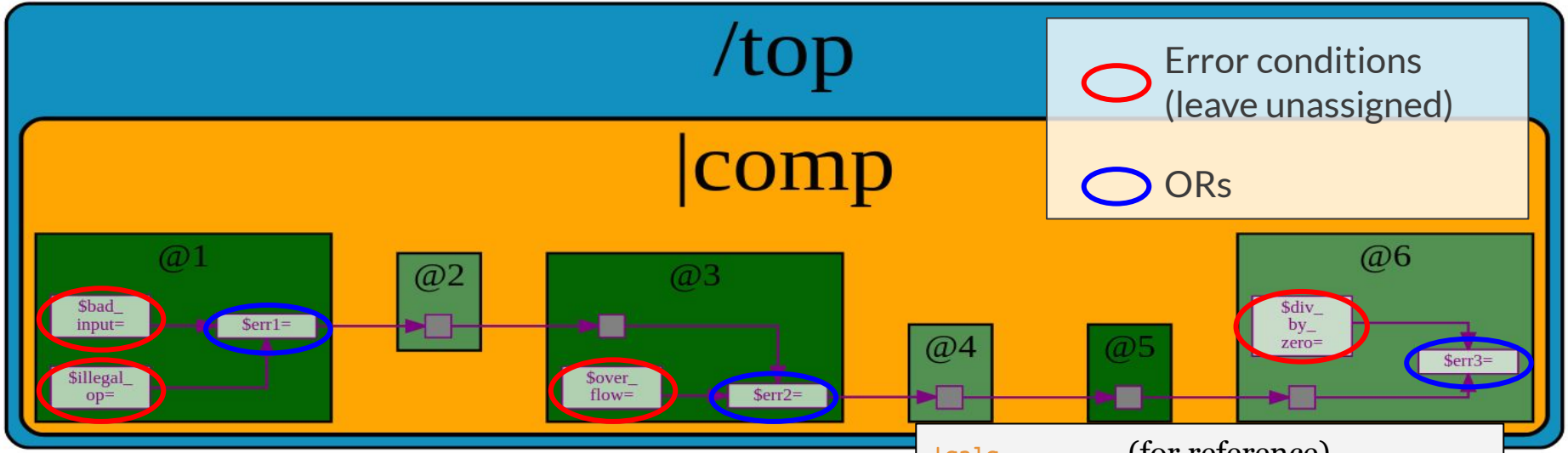
Retiming in SystemVerilog

```
// 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

See if you can produce this:



which ORs together (| |) various error conditions that can occur within a computation pipeline.
(And keep this open.)

(for reference)

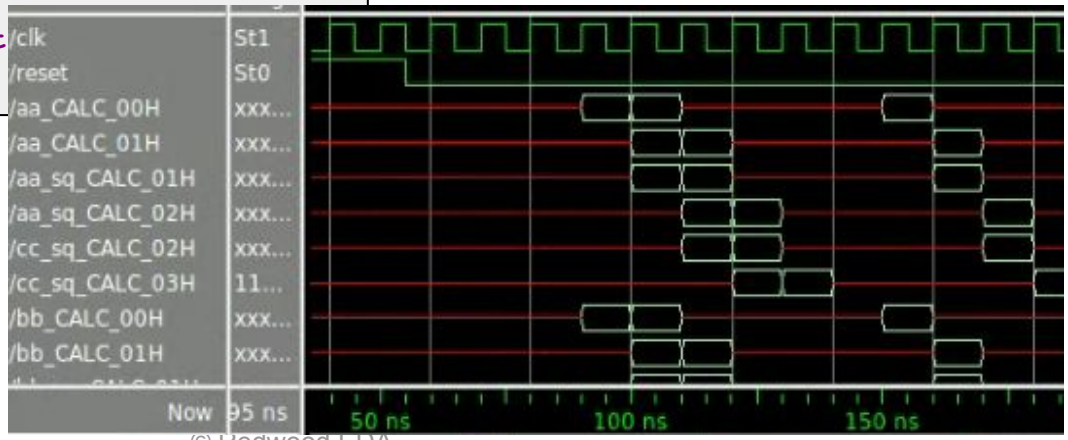
```
|calc
   @1
      $aa_sq[31:0] = $aa * $aa;
      $bb_sq[31:0] = $bb * $bb;
   @2
      $cc_sq[31:0] = $aa_sq + $bb_sq;
   @3
      $cc[31:0] = sqrt($cc_sq);
```

Validity

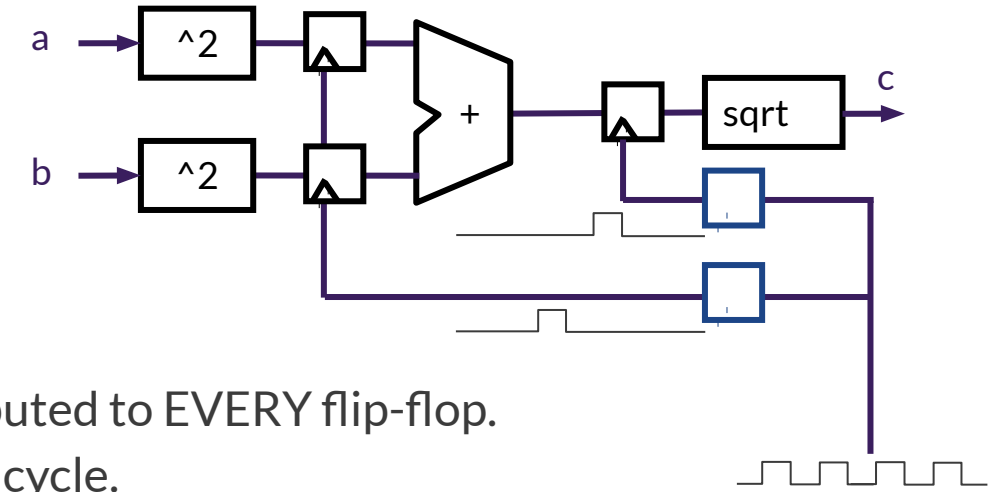
```
|calc
   @1
      $valid = ...;
   ?$valid
      @1
         $aa_sq[31:0] = $aa * $aa;
         $bb_sq[31:0] = $bb * $bb;
      @2
         $cc_sq[31:0] = $aa_sq + $bb_sq;
      @3
         $cc[31:0] = sqrt
```

Validity provides:

- Easier debug
- Cleaner design
- Better error checking
- Automated clock gating



Clock Gating



- Motivation:
  - Clock signals are distributed to EVERY flip-flop.
  - Clocks toggle twice per cycle.
  - This consumes power.
- Clock gating avoids toggling clock signals.
- FPGAs generally use very coarse clock gating + clock enables.
- TL-Verilog can produce fine-grained gating or enables.

Validity

Try this on your error logic.

```
\TLV
   |comp
      ?$valid
         @1
            $err1 = $bad_input | $illegal_op;
         @3
            $err2 = $err1 | $over_flow;
         @6
            $err3 = $err2 | $div_by_zero;
```

(Use Ctrl-] to indent a block of selected code.)

Observe the diagram and waveform.

Awesomeness We Don't Have Time For

- Pipeline interactions
- State
- Hierarchy
- **Transaction flows**
- Modularity and reuse
- Hardware construction

Training

- TL-V RISC-V Workshops in March:
  - Hobbyists: RISC-V International/Linux Foundation
  - Students: MYTH Workshop
  - Professionals: IEEE
- Makerchip tutorials
- Other videos, slides, articles, papers, etc: redwoodeda.com/publications

Open-Source TL-Verilog Projects

Flexible RISC-V CPU

Google
Summer of Code

Cloud FPGAs

Hardware-Accelerated
Web Applications

Visual Debug