Timing-Abstract Circuit Design in Transaction-Level Verilog

Steve Hoover

Founder, Redwood EDA

steve.hoover@redwoodeda.com

ICCD Nov. 8, 2017

Agenda

- Motivation
  - The complexity crisis
  - How we manage complexity today
  - What’s not working
- Timing-Abstract Design in TL-Verilog
- Results

Verilog

Verilog was born of a different era…

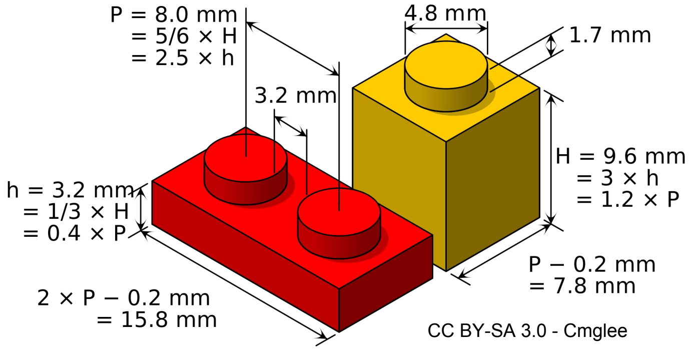
Year	Processor	Clock	Transistors	HDL	IDE
1985	i386	33MHz	275K	Verilog (to verify)	Emacs/ vi
2017	AMD Epyc	3.0GHz (~100x)	19.2B (>70,000x)	Verilog	XEmacs/ Vim

We can’t continue designing this way!

SoC Methodology

Manage complexity through modularity and reuse of IP building blocks.

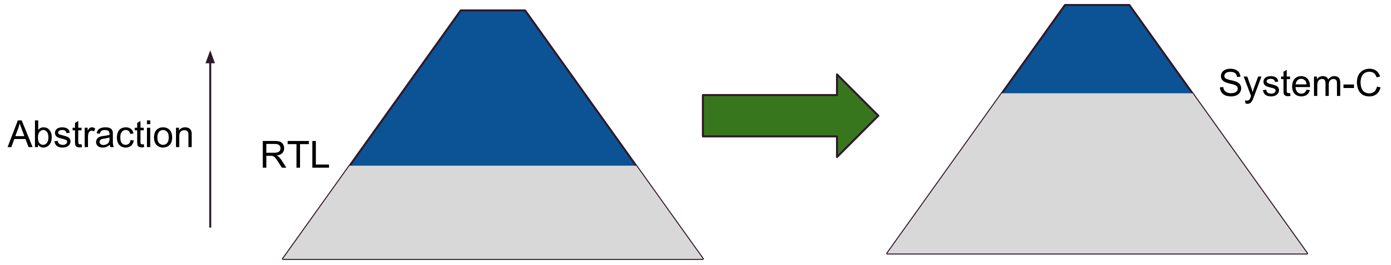
- IP utilized in different contexts, with different constraints for:
    - area
    - power
    - performance
    - test/debug infrastructure
    - **clock frequency**
- RTL expresses *an* implementation, with particular constraints



RTL is <u>not</u> good for IP!

High-Level Synthesis

Design algorithm-level and let tools generate RTL, under given physical constraints.



- **Fantastic** for <u>some</u> designs.
- For others, SystemC becomes RTL.

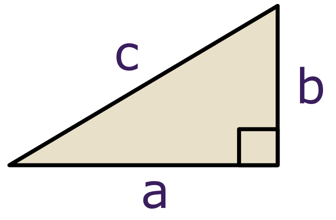
Many designs require RTL details!

The Need

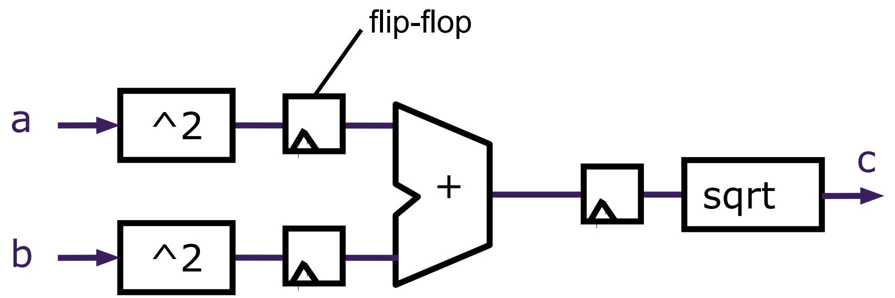
We need to be able to model cycle-level interactions in a way that is easier to manage.

A Simple Pipeline

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

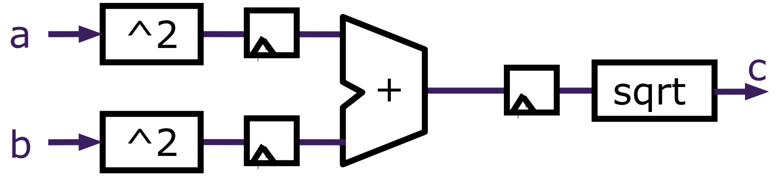


c = sqrt(a^2 + b^2)

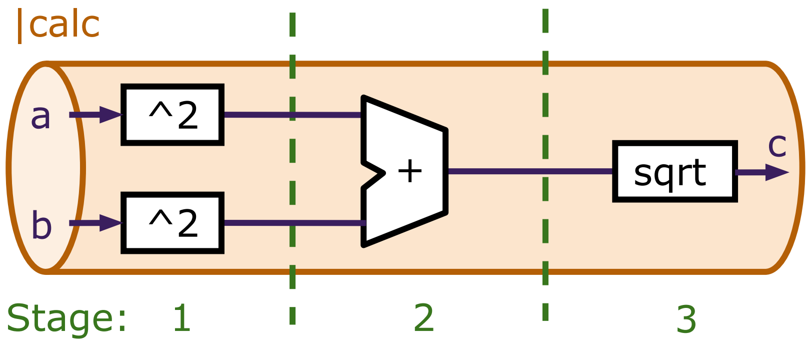


A Simple Pipeline - Timing-Abstract

RTL:

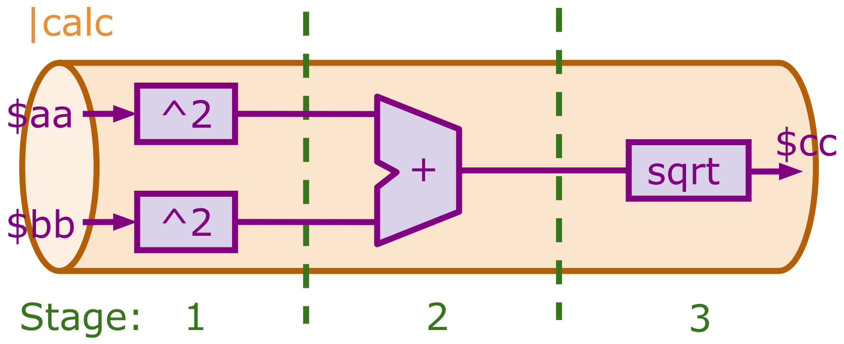


Timing-abstract:



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

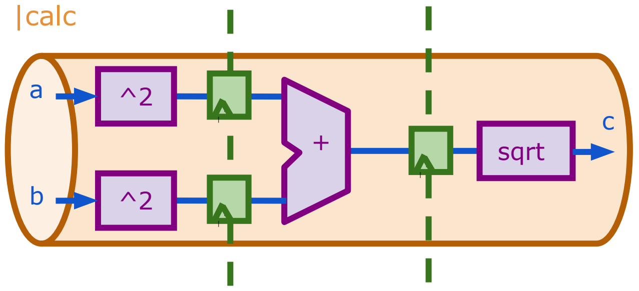
A Simple Pipeline - TL-Verilog



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);
```

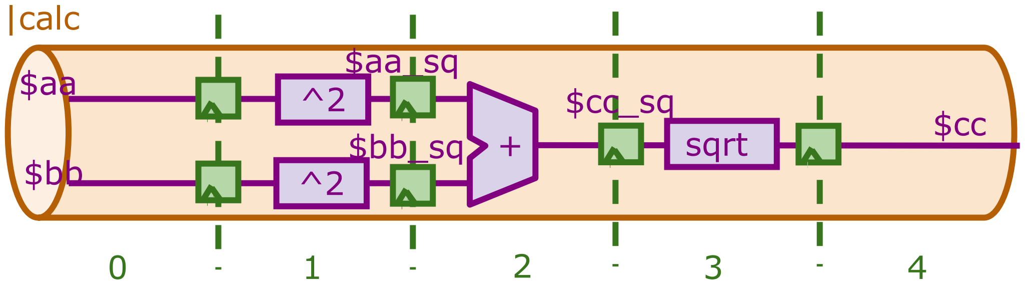
~3.5x

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);
```

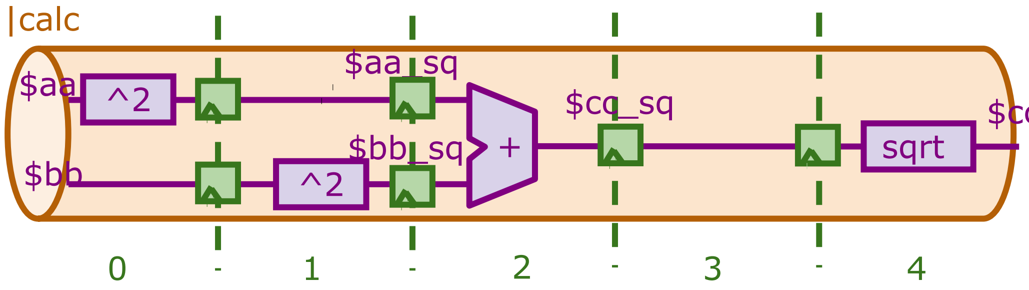
Retiming -- Easy and Safe

```
|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);
```



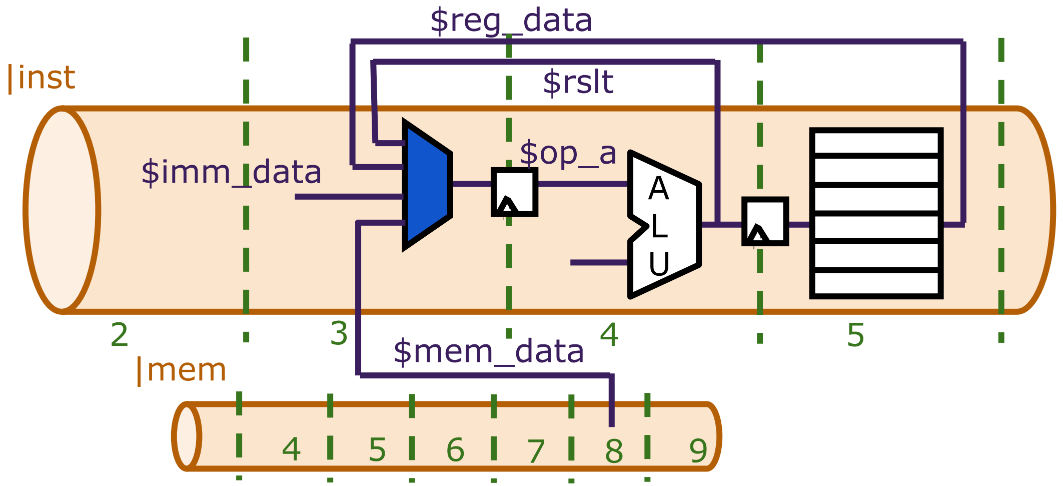
Staging is a <u>physical</u> attribute. No impact to behavior.

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);
```

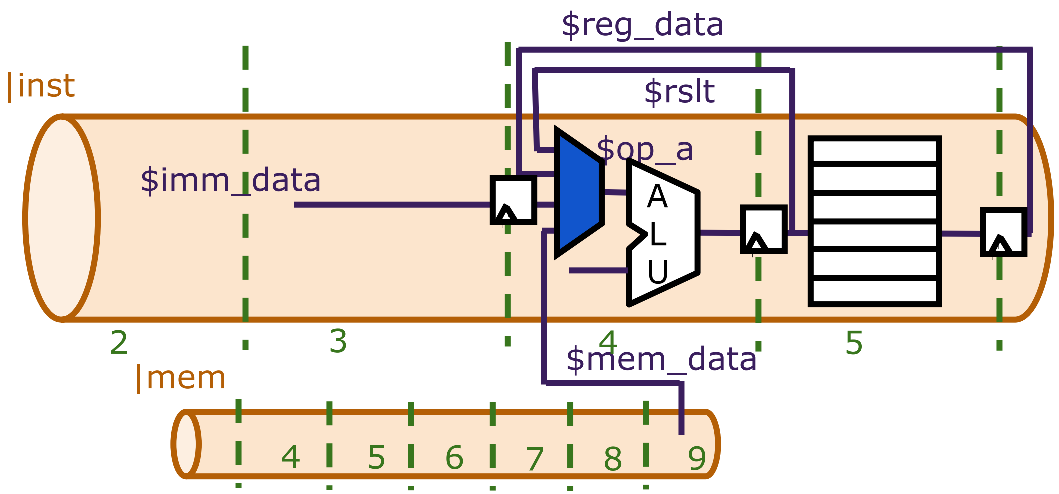
VERY BUG-PRONE!

Operand Mux



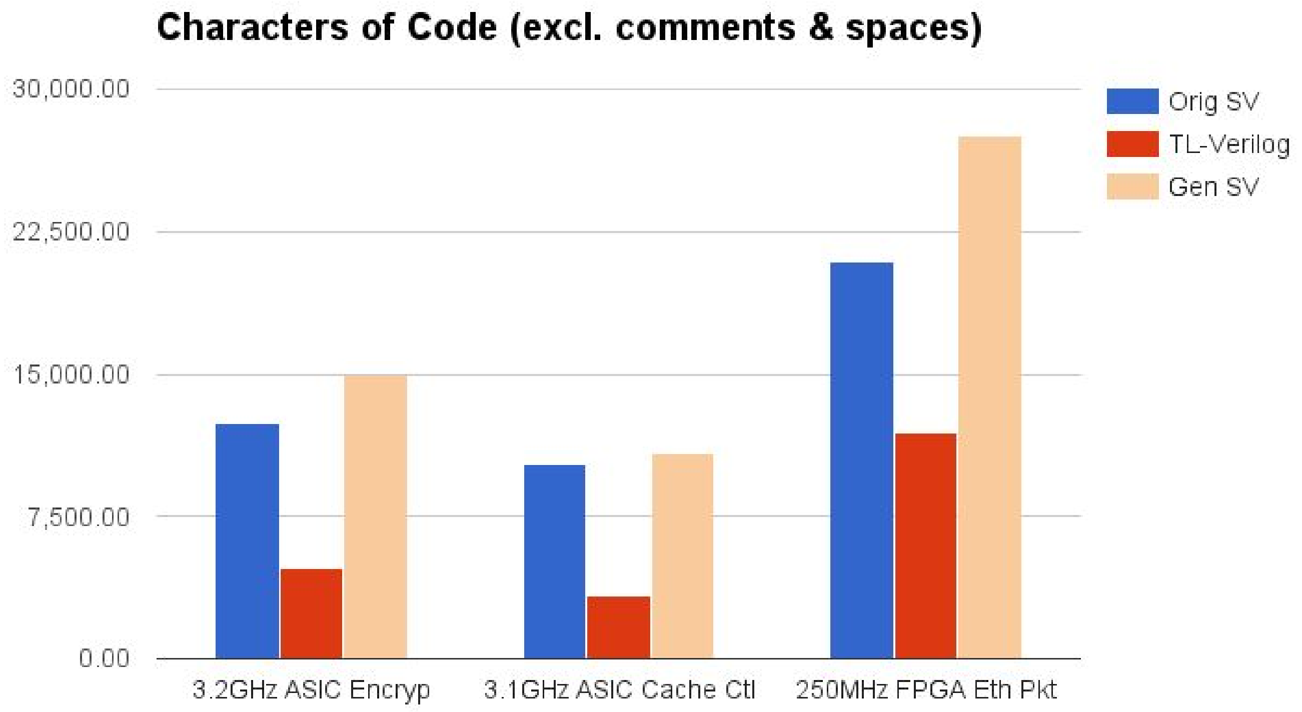
```
|inst
   @3
      $op_a[63:0] =
         ($op_a_src == REG) ? >>2$reg_data :
         ($op_a_src == BYP) ? >>1$rslt     :
         ($op_a_src == IMM) ? $imm_data    :
         ($op_a_src == MEM) ? /top|mem>>5$mem_data :
                              64'b0;
```

Operand Mux Retimed



```
|inst
   @4
      $op_a[63:0] =
         ($op_a_src == REG) ? >>2$reg_data :
         ($op_a_src == BYP) ? >>1$rslt     :
         ($op_a_src == IMM) ? $imm_data    :
         ($op_a_src == MEM) ? /top|mem>>5$mem_data :
                              64'b0;
```

Code Size Results from Industry Examples



Benefits of TL-Verilog

Less code, fewer bugs!

Less code *change*, fewer bugs!

Typically:

- ½ the code
- ~¼ the change for reuse
- ~⅙ the code for HLM

In certain real-world cases:

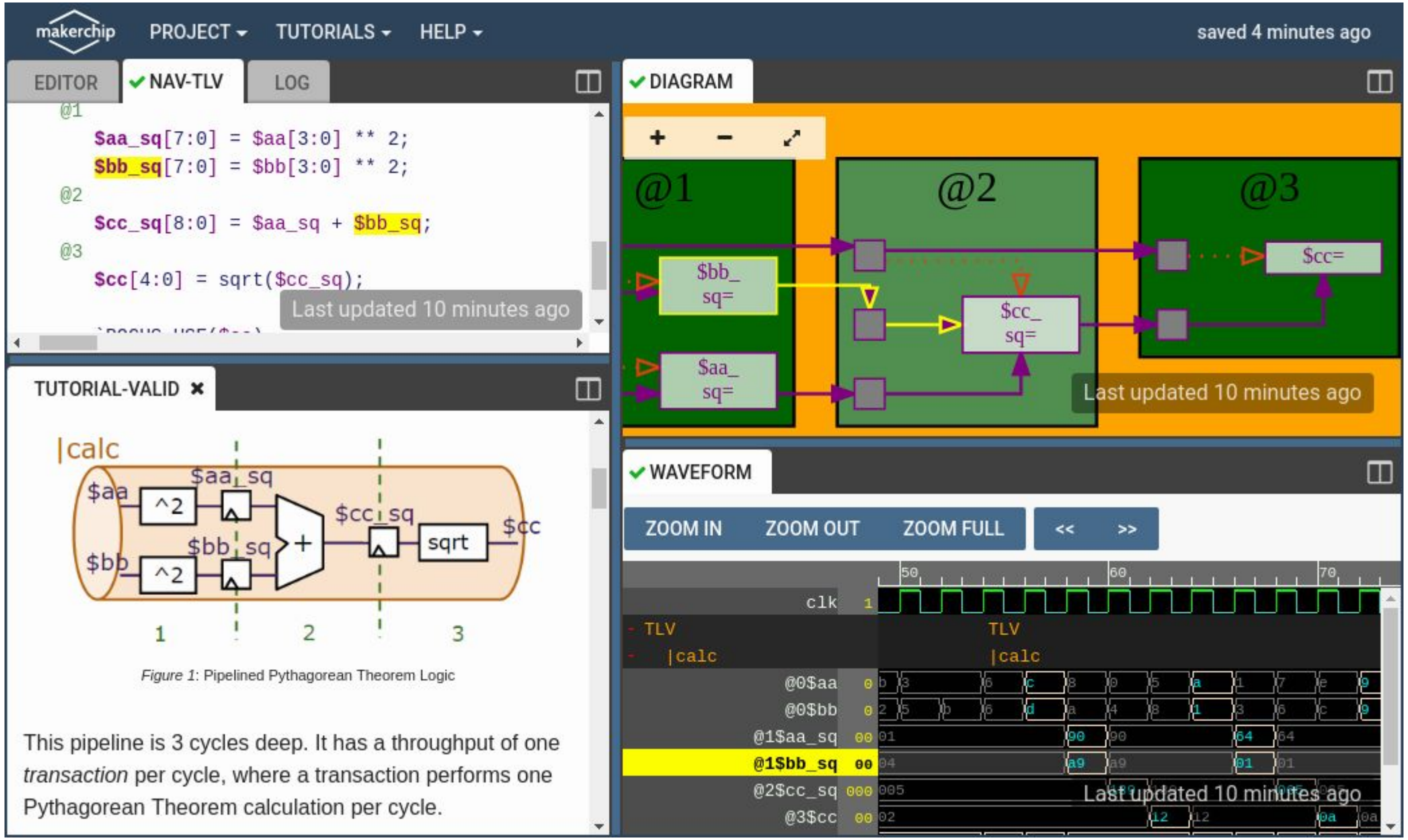
- 1/200 the code change!

More to TL-Verilog

- Hierarchy
- State
- Validity
- Clock gating
- Transactions!

(and more in proposal phase)

makerchip.com



```
@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);
```

Figure 1: Pipelined Pythagorean Theorem Logic

This pipeline is 3 cycles deep. It has a throughput of one *transaction* per cycle, where a transaction performs one Pythagorean Theorem calculation per cycle.

Be a part of it!

Reach out to me at:

```
steve.hoover@redwoodeda.com
```

Learn more at:

```
makerchip.com
```