These slides accompany the webinar at:
https://www.udemy.com/course/1549918

Pipelining RISC-V with Transaction-Level Verilog

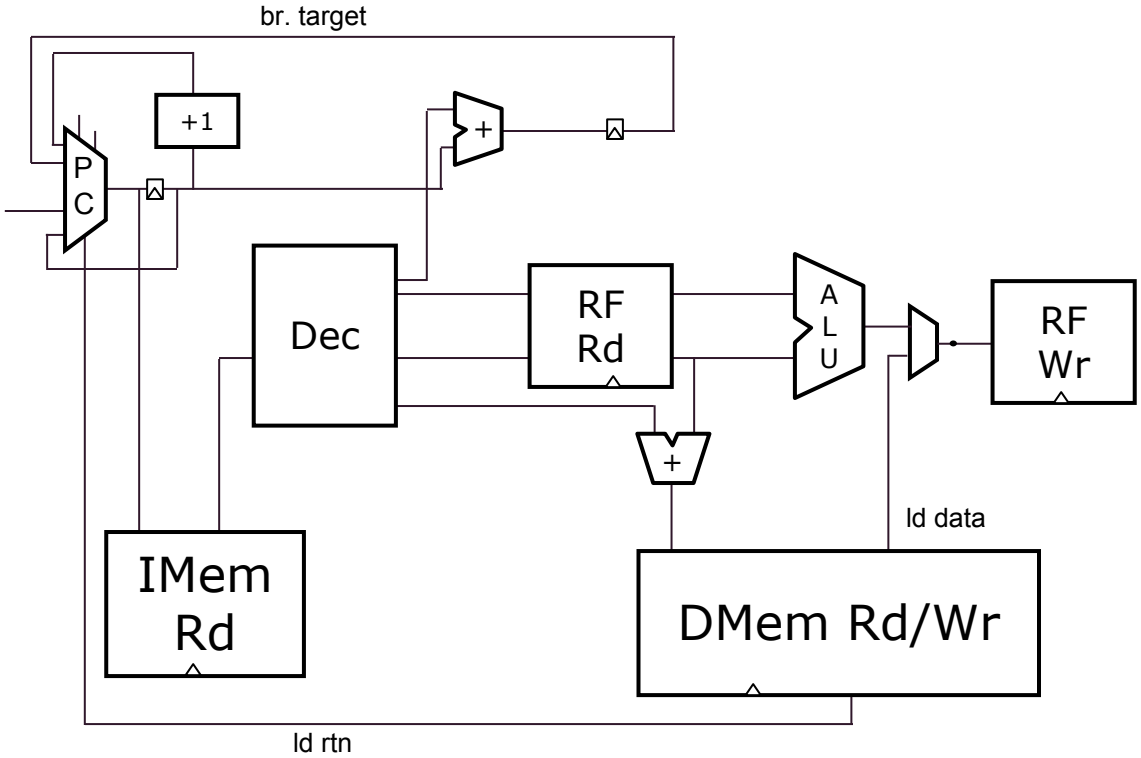
Two College Courses in Digital Logic in Three Hours

Steve Hoover
Founder, Redwood EDA
steve.hoover@redwoodeda.com
Feb. 10, 2018

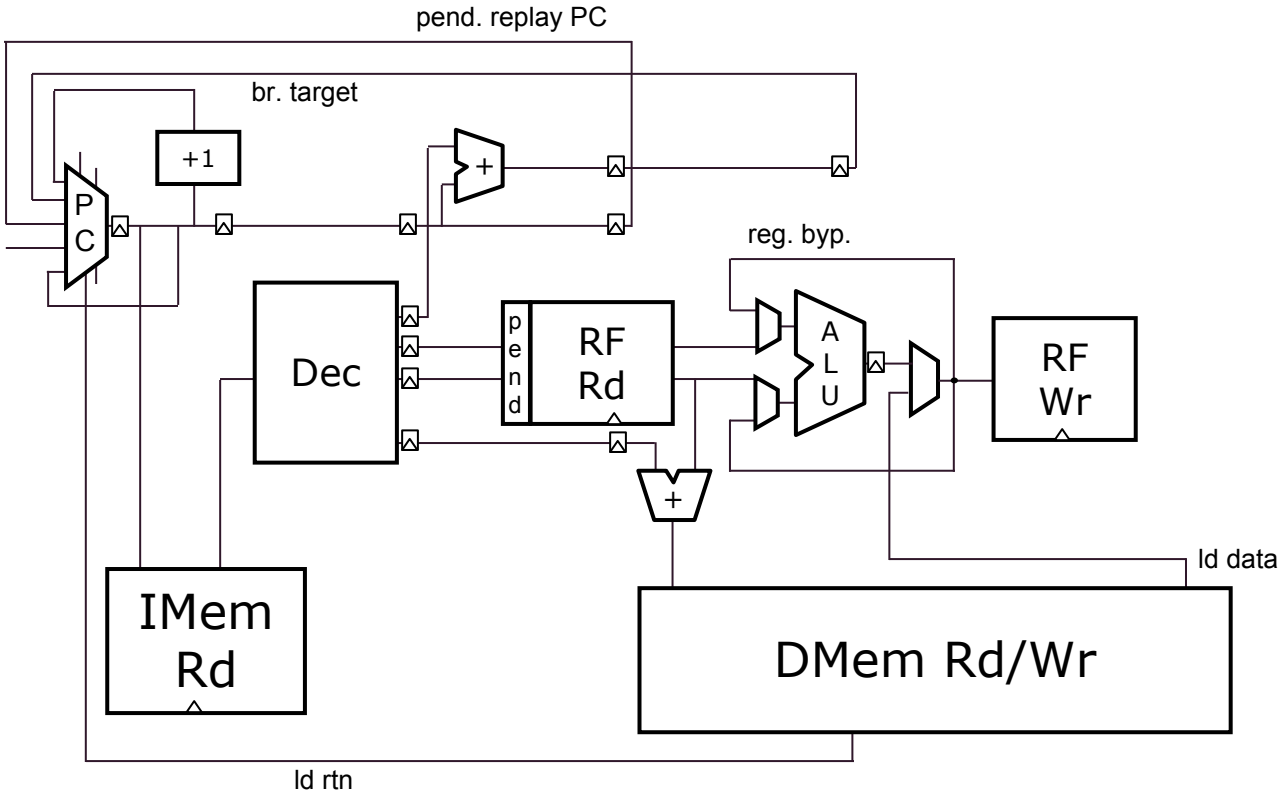
Agenda

- RISC-V Overview
- IP Design Methodology
- Design Concepts using TL-Verilog in Makerchip.com
  - Combinational Logic
  - Sequential Logic
  - Pipelines
  - Validity
  - Pipeline Interactions
  - Hierarchy
  - Elaboration
  - Interfacing with Verilog/SystemVerilog
  - State
  - Transactions
  - Verification
- Summary & Certification Challenge

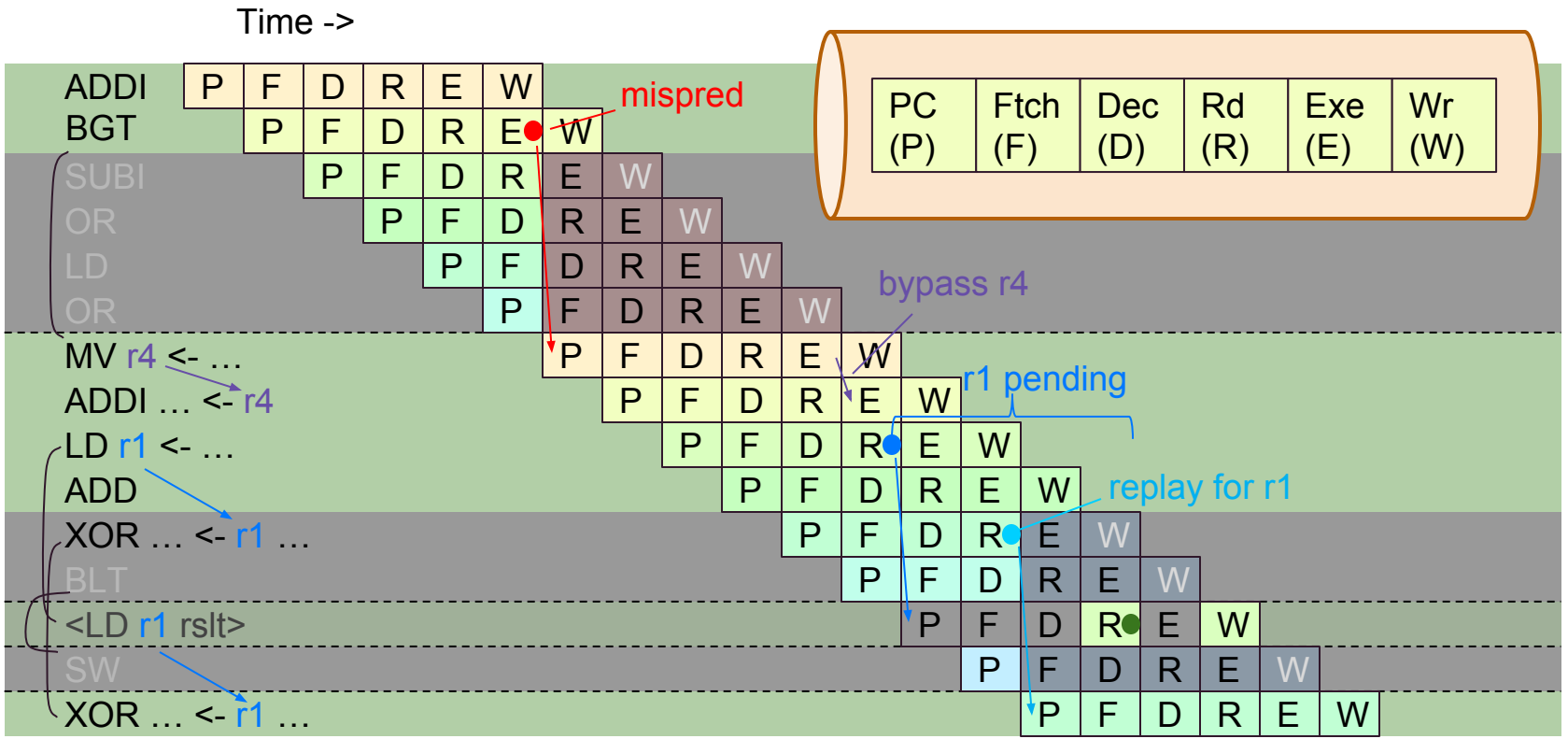
Example RISC-V Block Diagram



Pipelined RISC-V Block Diagram



RISC-V Waterfall Diagram & Hazards



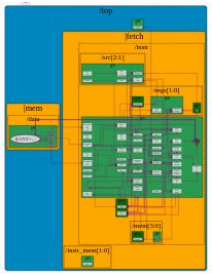
Agenda

- RISC-V Overview
- IP Design Methodology
- Design Concepts using TL-Verilog in Makerchip.com
  - Combinational Logic
  - Sequential Logic
  - Pipelines
  - Validity
  - Pipeline Interactions
  - Hierarchy
  - Elaboration
  - Interfacing with Verilog/SystemVerilog
  - State
  - Transactions
  - Verification
- Summary & Certification Challenge

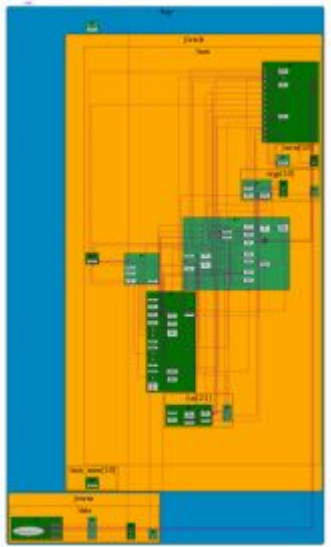
RISC-V IP Challenges

Every CPU core serves a different purpose

- General-purpose computing
- HPC workloads
- Hardware acceleration of X
- Microcontroller for X
- I/O processor
- Etc.



Low-Power
1-Stage
FPGA

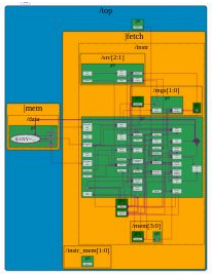


High-Perf
7-Stage
ASIC

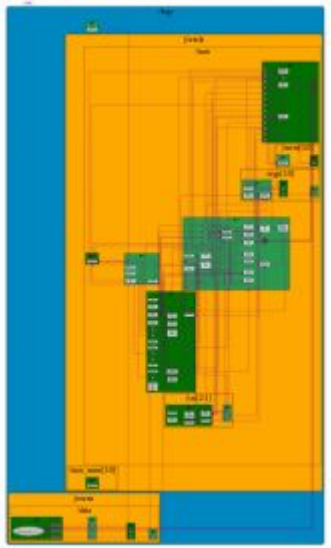
RISC-V IP Challenges

Every implementation is constrained differently

- area
- power
- performance
- test/debug infrastructure
- **clock frequency**



Low-Power
1-Stage
FPGA

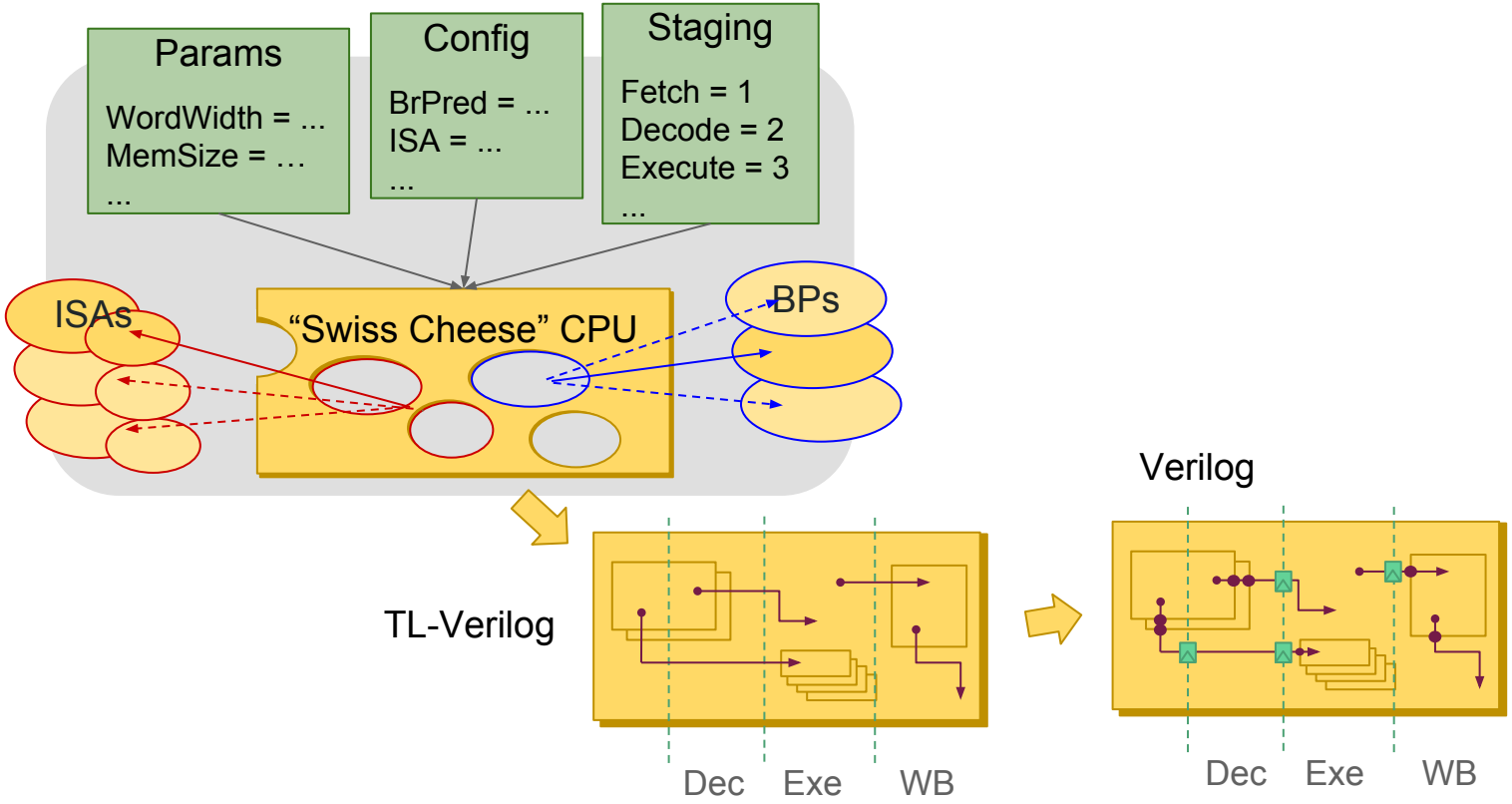


High-Perf
7-Stage
ASIC

RTL expresses *an* implementation.

RTL is not good for IP!

WARP-V: RISC-V CPU Core Generator



WARP-V Code (currently)

In a single 1500-line file (~1.5 wks of coding):

- The uArch model
- The RISC-V ISA logic
- A Mini-CPU ISA (for demonstration and academic use)
- A rudimentary RISC-V assembler
- A tiny sample program in RISC-V and Mini ISA.

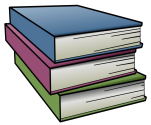
WIP: No caches, CSRs, etc.

Alternate Directions

SystemC + HLS

- Integrate w/ C++-based verification models
- Synthesize algorithms to gate-level RTL
- Tools optimize for constraints

Driven by: EDA Industry



Chisel, CλaSH, etc.

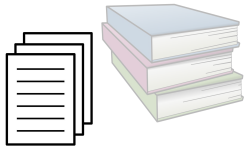
- Leverage s/w techniques to construct h/w

Driven by: Academia

TL-X (TL-Verilog)

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

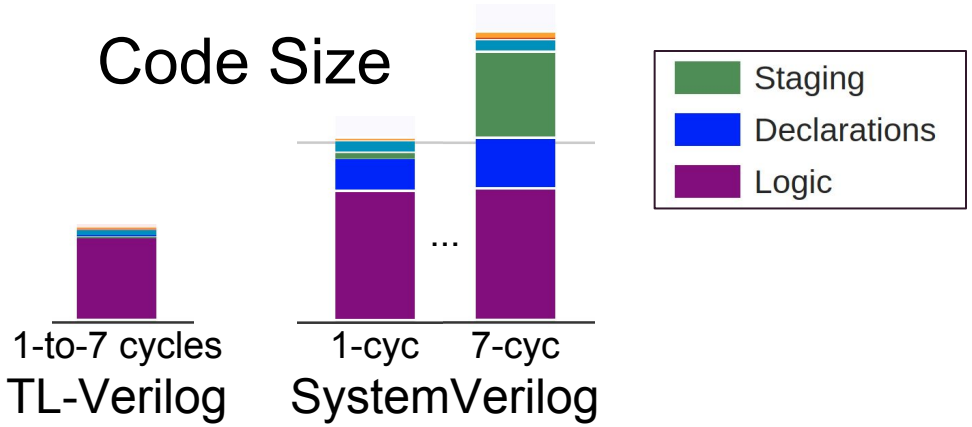
Driven by: Designers



TL-Verilog because…

For IP, like CPUs

- Concise
- Explicit
- But flexible



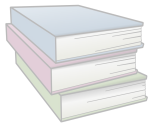
For learning

- TL-Verilog constructs ⇔ logic design concepts
- Simple to learn and code
- You can do more - two courses in 3 hrs.
- It has a free online IDE -- Makerchip (also edaplayground.com)
- To get you ahead of the curve

Simplifying

Stuff in Verilog you'll never need again:

- reg vs. wire vs. logic vs. bit
- blocking vs. non-blocking
- packed vs. unpacked
- generate blocks
- loops
- always blocks
- sensitivity lists



Stuff you need to learn

- pipelines
- hierarchy
- state
- transactions

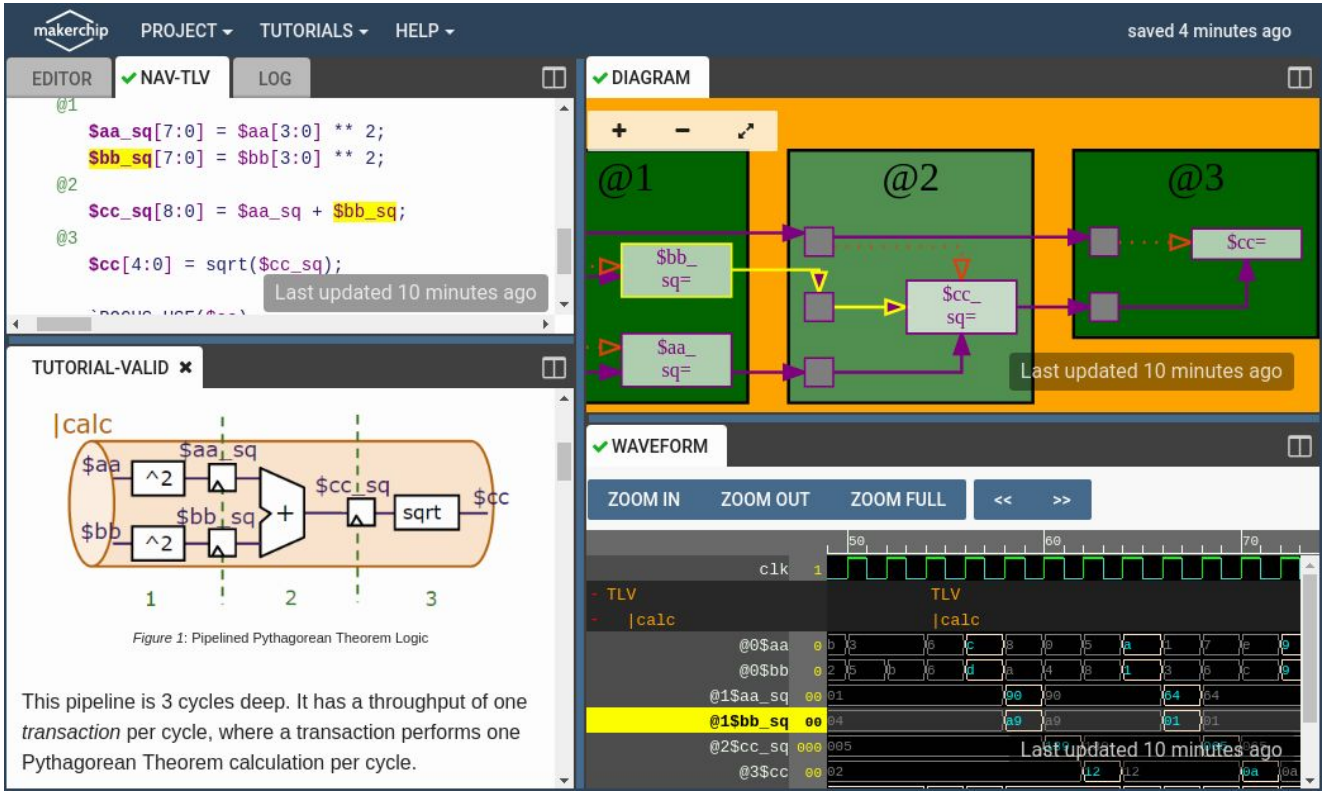
Verilog Spec TOC

Need
Not Crucial
Obsolete

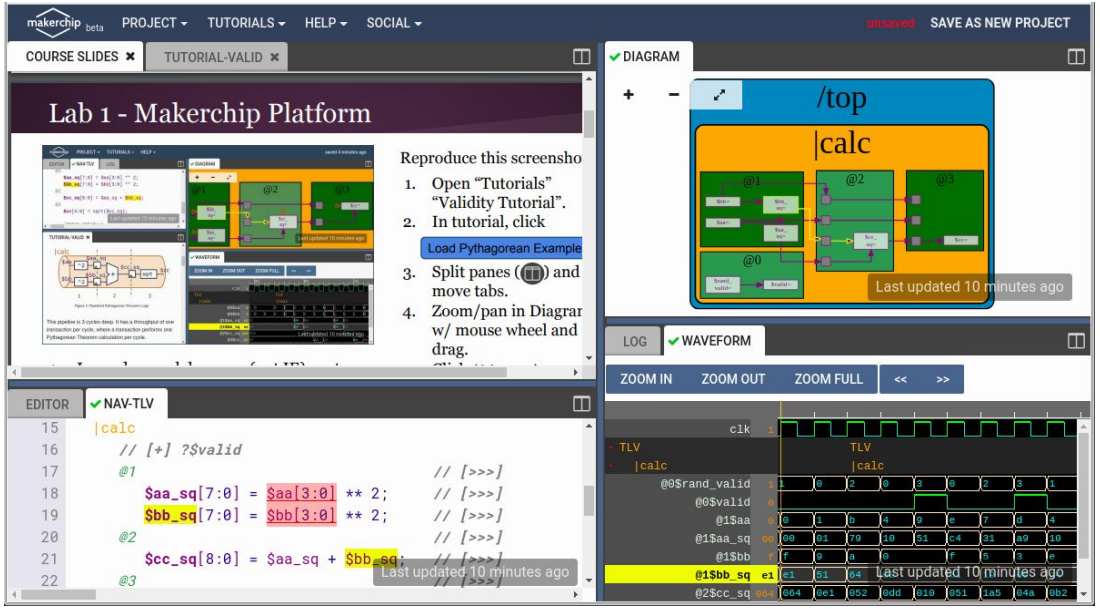
Agenda

- RISC-V Overview
- IP Design Methodology
- Design Concepts using TL-Verilog in Makerchip.com
  - Combinational Logic
  - Sequential Logic
  - Pipelines
  - Validity
  - Pipeline Interactions
  - Hierarchy
  - Elaboration
  - Interfacing with Verilog/SystemVerilog
  - State
  - Transactions
  - Verification
- Summary & Certification Challenge

Makerchip


makerchip
PROJECT
TUTORIALS
HELP
saved 4 minutes ago
EDITOR
NAV-TLV
LOG
@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);
Last updated 10 minutes ago
TUTORIAL-VALID
|calc
$aa
$bb
^2
^2
$aa_sq
$bb_sq
$cc_sq
+
sqrt
$cc
1
2
3
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.
DIAGRAM
@1
@2
@3
$bb_ sq=
$aa_ sq=
$cc_ sq=
$cc=
Last updated 10 minutes ago
WAVEFORM
ZOOM IN
ZOOM OUT
ZOOM FULL
<<
>>
clk
TLV
|calc
@0$aa
@0$bb
@1$aa_sq
@1$bb_sq
@2$cc_sq
@3$cc
Last updated 10 minutes ago

Lab: Makerchip Platform



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. 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.

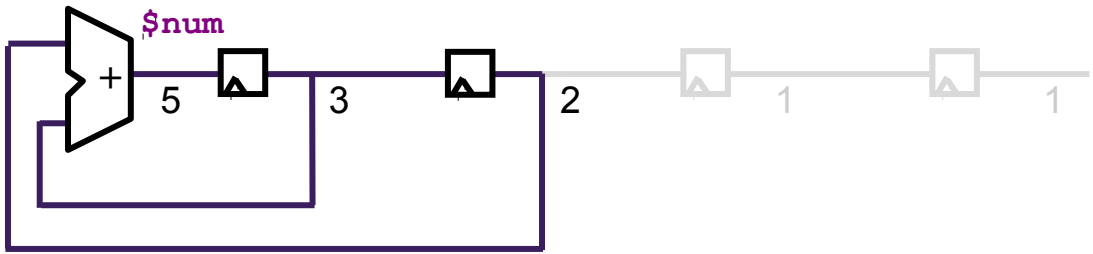
1. Try:
   ```
   $out[4:0] = $in1[3:0] + $in2[3:0];
   ```
   (Cut-n-paste)
2. View Waveform (values are in hexadecimal and addition can overflow)

Agenda

- RISC-V Overview
- IP Design Methodology
- Design Concepts using TL-Verilog in Makerchip.com
  - Combinational Logic
  - Sequential Logic
  - Pipelines
  - Validity
  - Pipeline Interactions
  - Hierarchy
  - Elaboration
  - Interfacing with Verilog/SystemVerilog
  - State
  - Transactions
  - Verification
- Summary & Certification Challenge

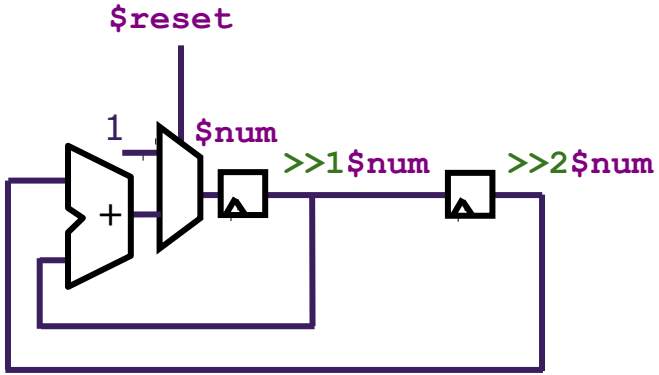
Sequential Logic - Fibonacci Series

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, …



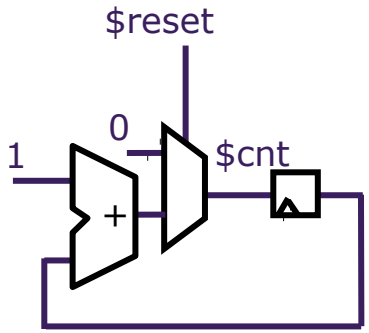

```
$num[31:0] = $reset ? 1 : (>>1$num + >>2$num);
```

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

Lab: Counter

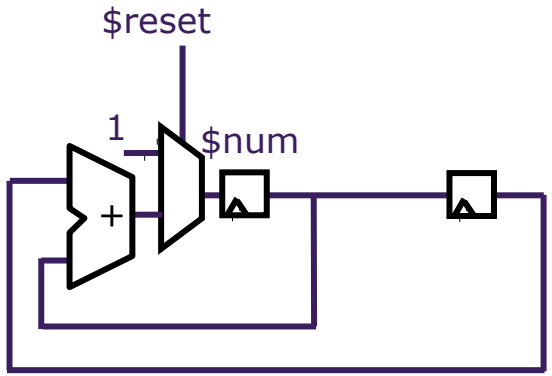
Lab:

1. Design a free-running counter:



2. Compile and explore.

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




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

3-space indentation
(no tabs)

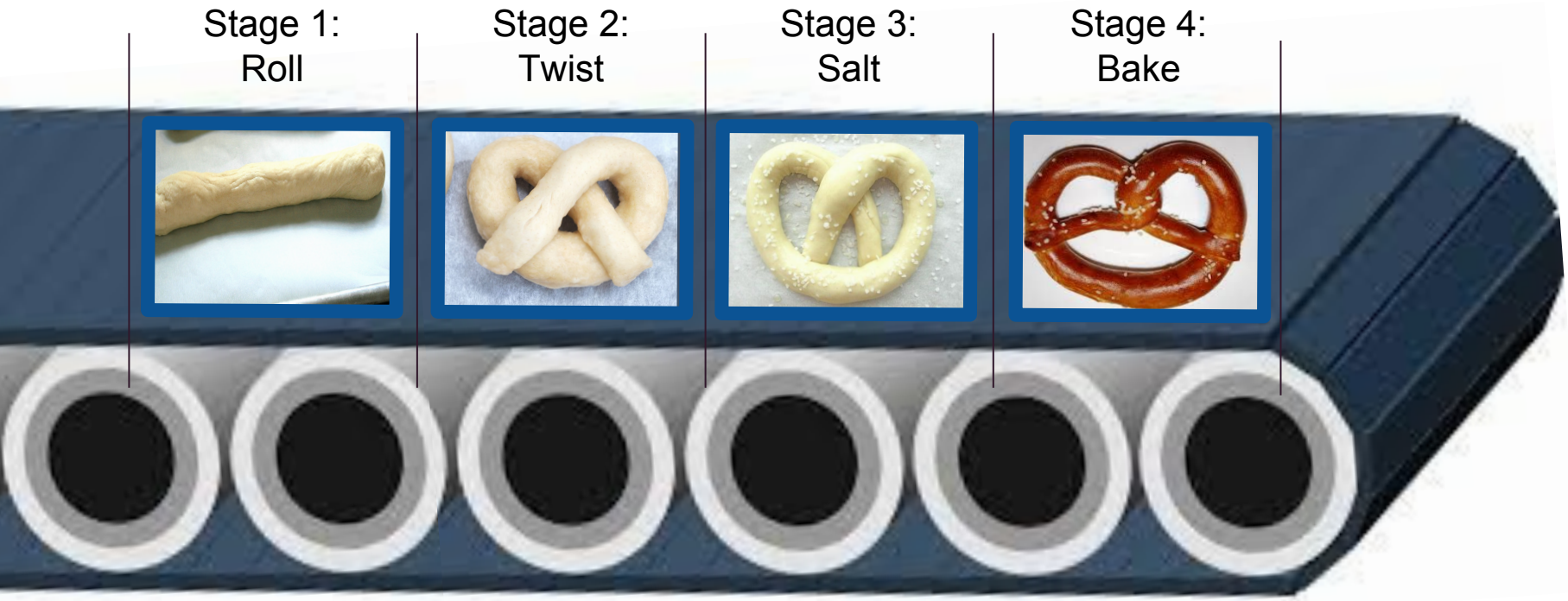
(makerchip.com/sandbox/0/0wjhLP)

Agenda

- RISC-V Overview
- IP Design Methodology
- Design Concepts using TL-Verilog in Makerchip.com
  - Combinational Logic
  - Sequential Logic
  - Pipelines
  - Validity
  - Pipeline Interactions
  - Hierarchy
  - Elaboration
  - Interfacing with Verilog/SystemVerilog
  - State
  - Transactions
  - Verification
- Summary & Certification Challenge

Pipeline

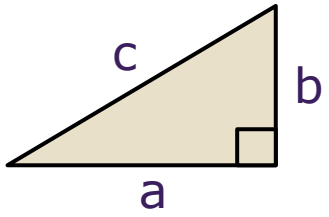
Pretzel = Transaction



A Simple Pipeline

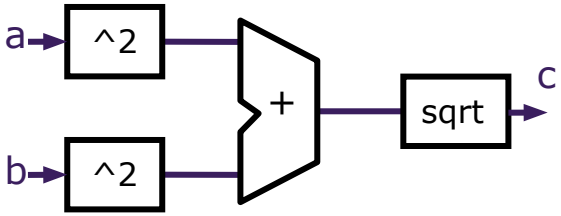
Pythagoras's Theorem circuit:

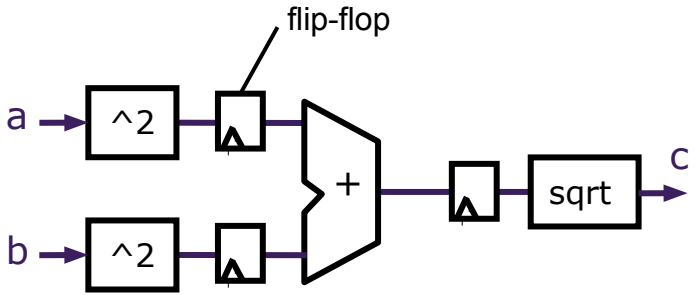
```
$aa_sq[31:0] = $aa * $aa;
$bb_sq[31:0] = $bb * $bb;
$cc_sq[31:0] = $aa_sq + $bb_sq;
$cc[31:0] = sqrt($cc_sq);
```



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

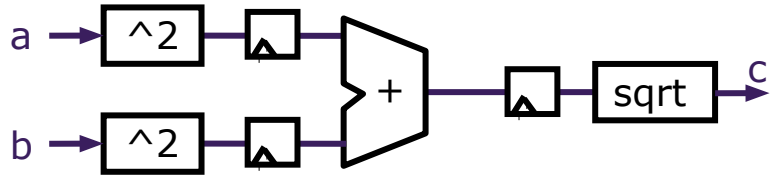
Too much for one cycle. Distribute over three cycles.



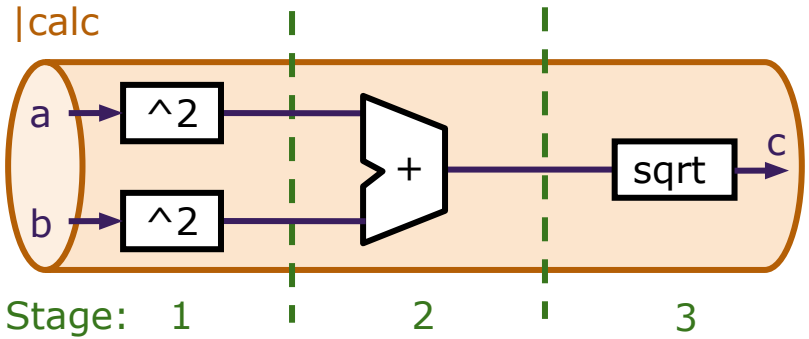


Timing-Abstraction

RTL:

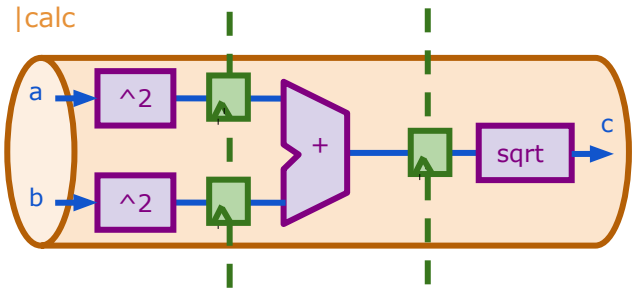


Timing-abstract:



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

TL-Verilog vs. SystemVerilog



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

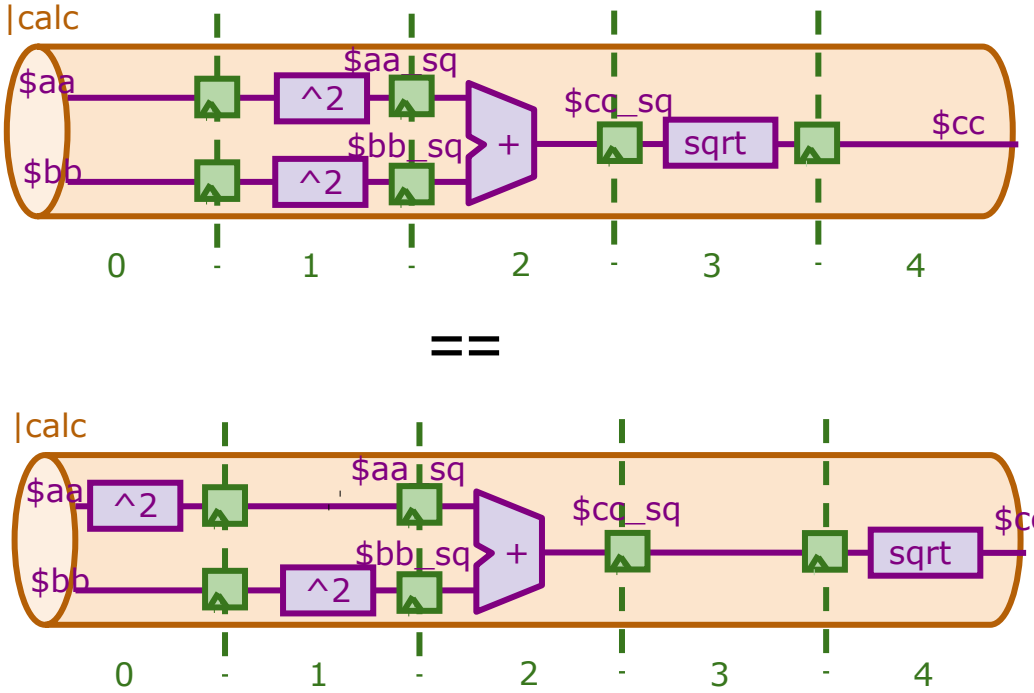
TL-Verilog

```
|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.

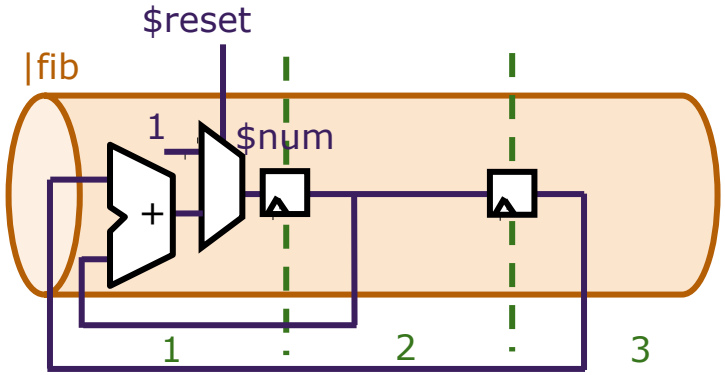
Retiming in SystemVerilog

```
// 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!

Fibonacci Series Pipeline

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

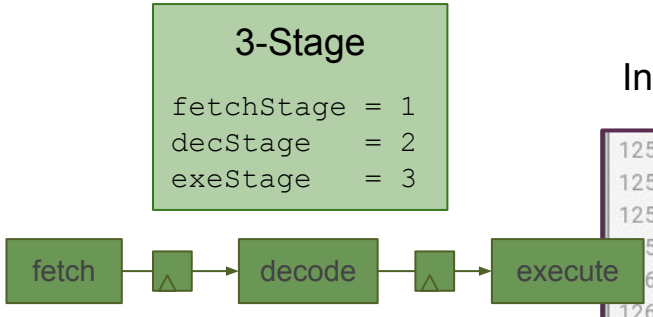



```
|fib
   @1
      $num[31:0] = $reset ? 1 : (>>1$num + >>2$num);
```

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

Parameterized Pipelines

```
@fetchStage
   <fetch-logic>
@decodeStage
   <decode-logic>
@executeStage
   <execute-logic>
```

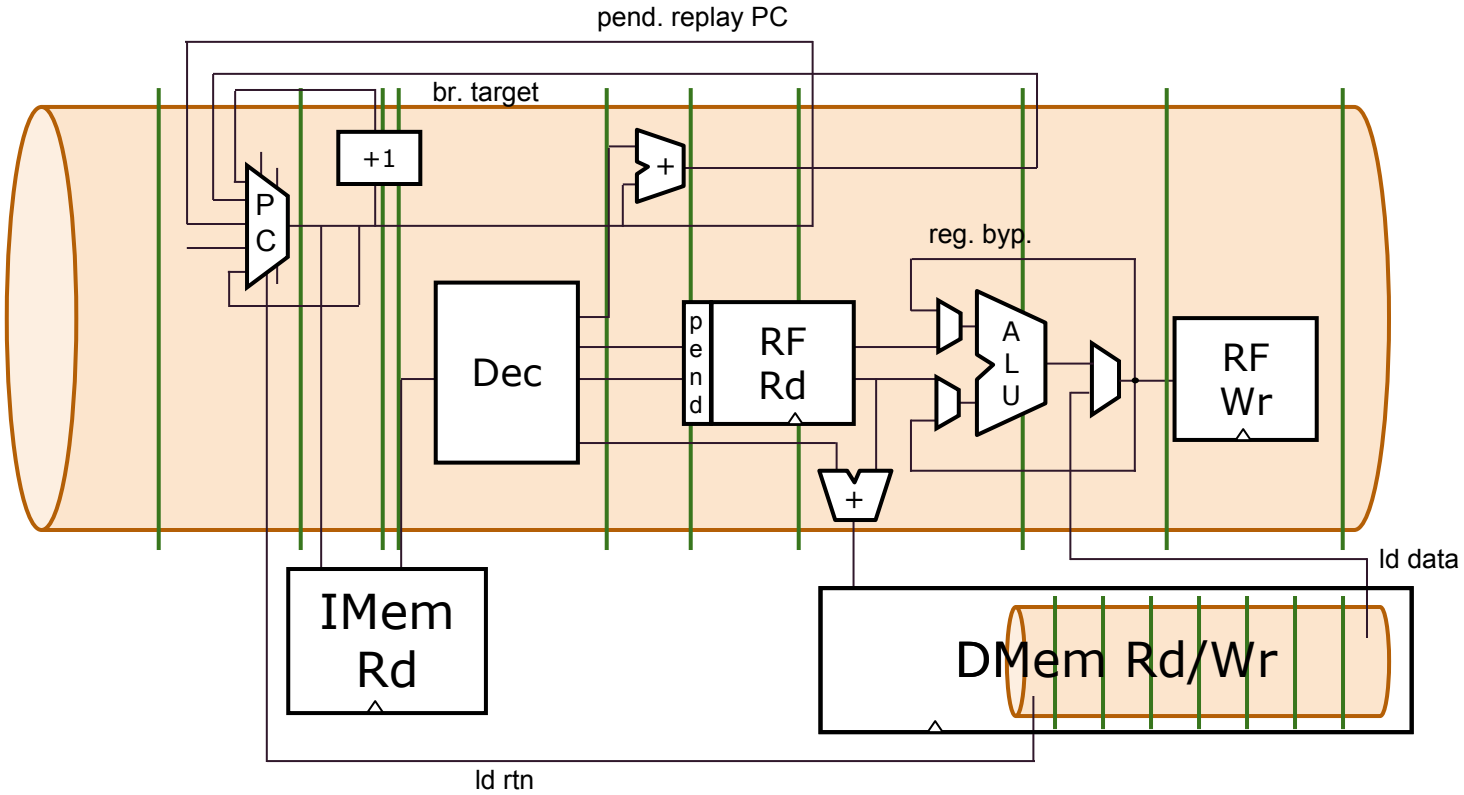


In WARP-V:

```
1256   |fetch
1257      /instr
1258         @M4_REG_WR_STAGE
1259            // =========
1260            // Reg Write
1261            // =========
1262
1263            $reg_write = $reset ? 1'b0
1264            \always_comb
1265               if ($reg_write)
1266                  /regs[$dest_reg]<<1$
1267
1268         // There's no bypass on pendi
1269         @M4_EXECUTE_STAGE
1270            /regs[*]
1271               $reg_match = /instr$des
1272               $next_pending =  // Sho
1273               /instr$reset ? 1'b0
1274               // set for loads
```

Not Practical with RTL!

WARP-V Parameterized Staging



Agenda

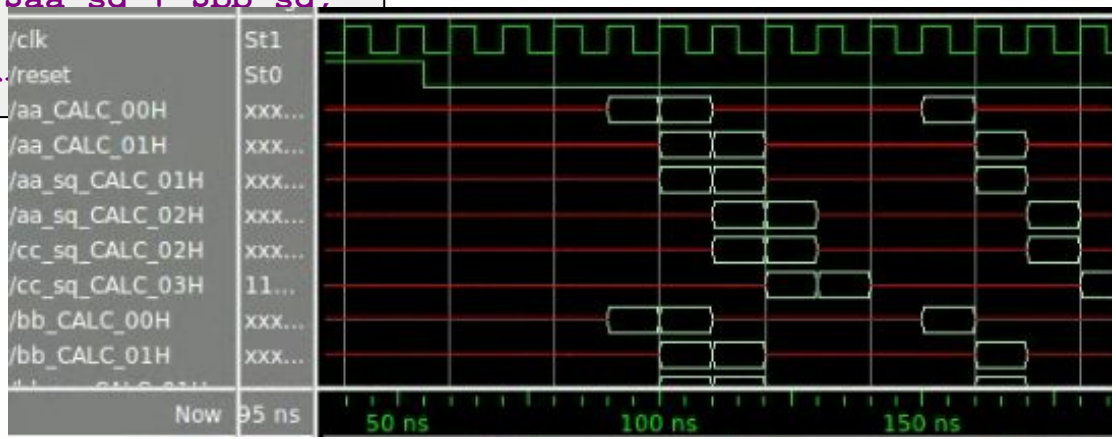
- RISC-V Overview
- IP Design Methodology
- Design Concepts using TL-Verilog in Makerchip.com
  - Combinational Logic
  - Sequential Logic
  - Pipelines
  - Validity
  - Pipeline Interactions
  - Hierarchy
  - Elaboration
  - Interfacing with Verilog/SystemVerilog
  - State
  - Transactions
  - Verification
- Summary & Certification Challenge

Validity

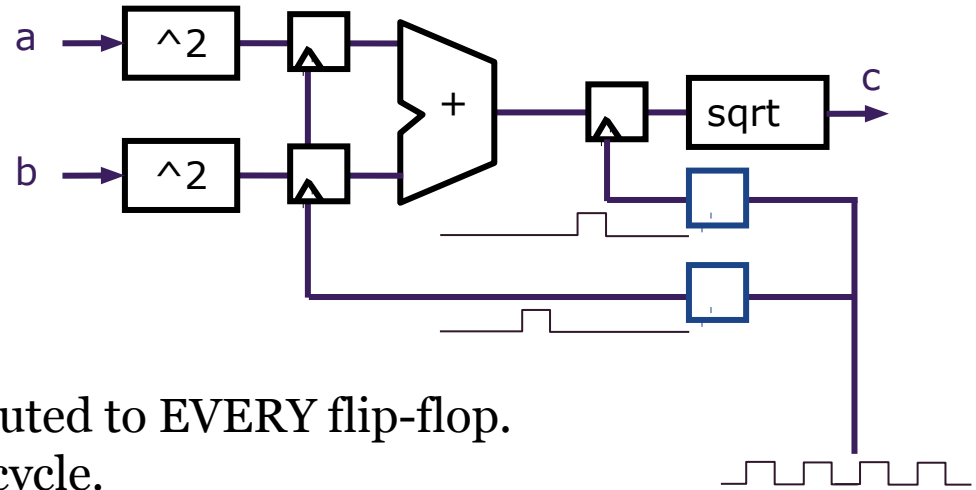
```
|calc
   @1
      $valid = ...;
   ?$valid
      @1
         $aa_sq[31:0] = $aa * $aa;
         $bb_sq[31:0] = $bb * $bb;
      @2
         $cc_sq[31:0] = $aa_sq + $bb_sq;
      @3
         $cc[31:0] = sqr
```

Validity provides:

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



Clock Gating

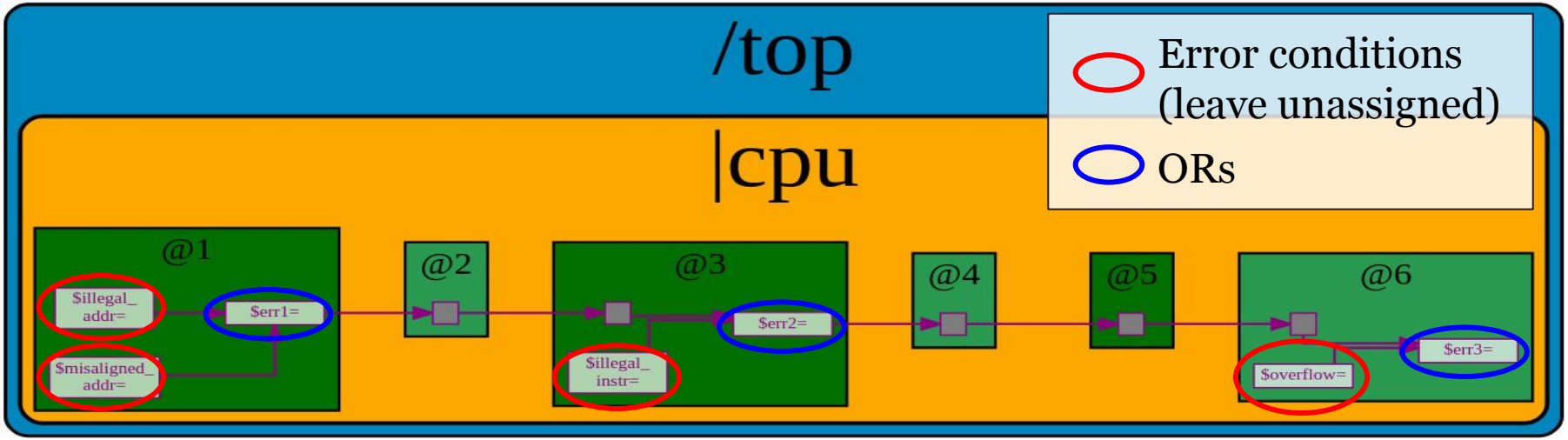


- Motivation:
  - Clock signals are distributed to EVERY flip-flop.
  - Clocks toggle twice per cycle.
  - This consumes power.
- Clock gating avoids toggling clock signals.
- FPGAs generally use very coarse clock gating + clock enables.
- TL-Verilog can produce fine-grained gating or enables.

Lab 3: Errors in WARP-V

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

1) See if you can produce this:



which ORs together various error conditions that can occur on an instruction. (OR is "||")

2) Add a `?$valid` condition.
(<ctrl>-"]" - indent)

Indentation is 3 spaces

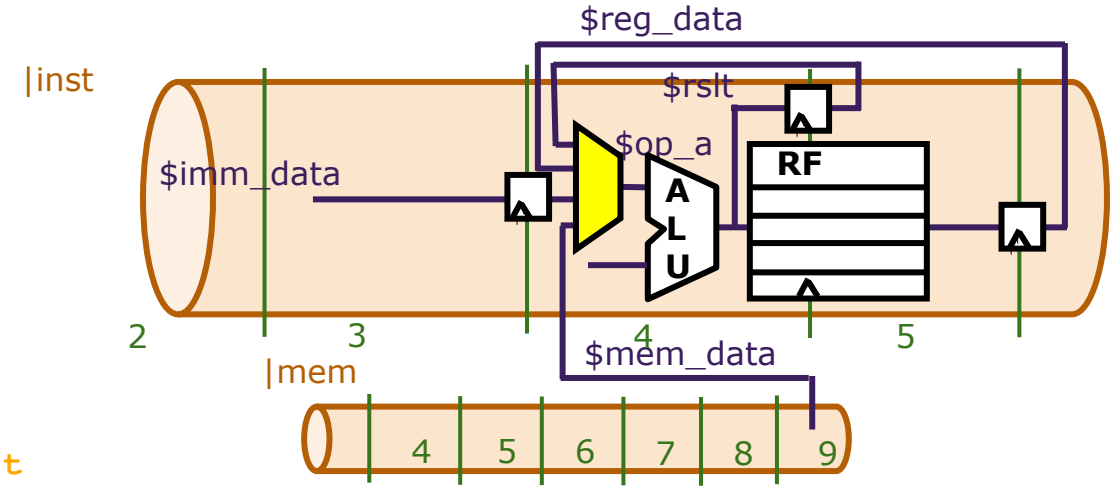
For reference:

```
\TLV
   |calc
      @0
         $aa_sq[31:0] = $aa * $aa;
      @1
         $bb_sq[31:0] = $bb * $bb;
```

Agenda

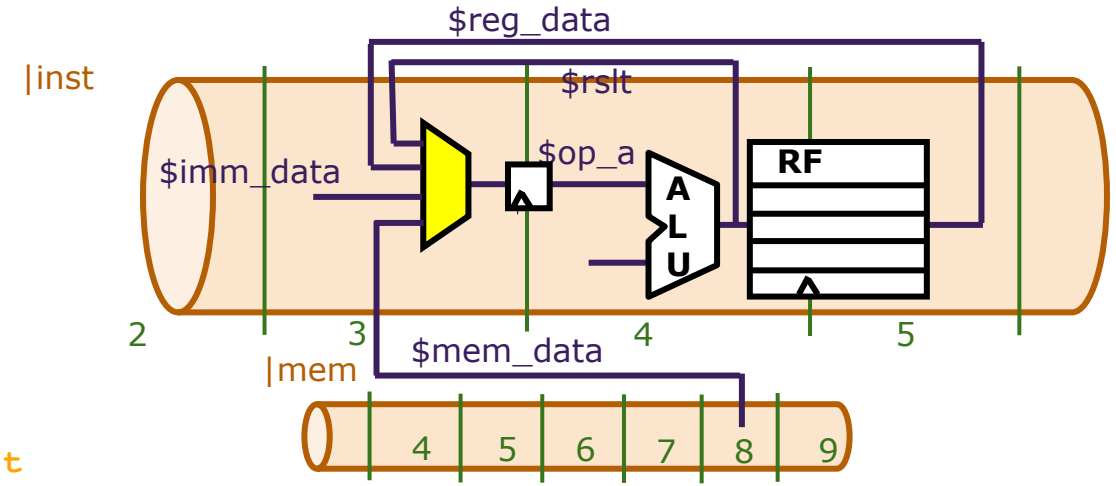
- RISC-V Overview
- IP Design Methodology
- Design Concepts using TL-Verilog in Makerchip.com
  - Combinational Logic
  - Sequential Logic
  - Pipelines
  - Validity
  - Pipeline Interactions
  - Hierarchy
  - Elaboration
  - Interfacing with Verilog/SystemVerilog
  - State
  - Transactions
  - Verification
- Summary & Certification Challenge

WARP-V Operand Mux



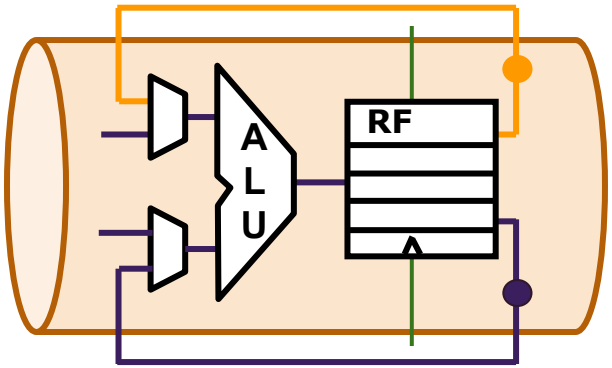

```
|inst
   ?$op_a_valid
      @4
         $op_a[63:0] =
            ($op_a_src == IMM) ? $imm_data     :
            ($op_a_src == BYP) ? >>1$rslt      :
            ($op_a_src == REG) ? >>2$reg_data  :
            ($op_a_src == MEM) ? /top|mem>>5$mem_data :
                                 64'b0;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

WARP-V Operand Mux Retimed

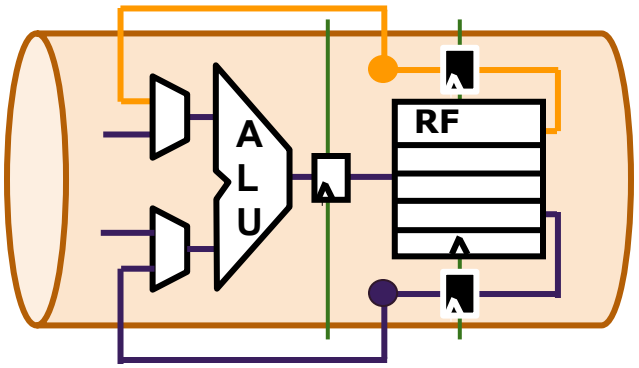



```
|inst
   ?$op_a_valid
      @3
         $op_a[63:0] =
            ($op_a_src == IMM) ? $imm_data     :
            ($op_a_src == BYP) ? >>1$rslt      :
            ($op_a_src == REG) ? >>2$reg_data  :
            ($op_a_src == MEM) ? /top|mem>>5$mem_data :
                                 64'b0;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

Register Bypass



1-Cycle



3-Cycle

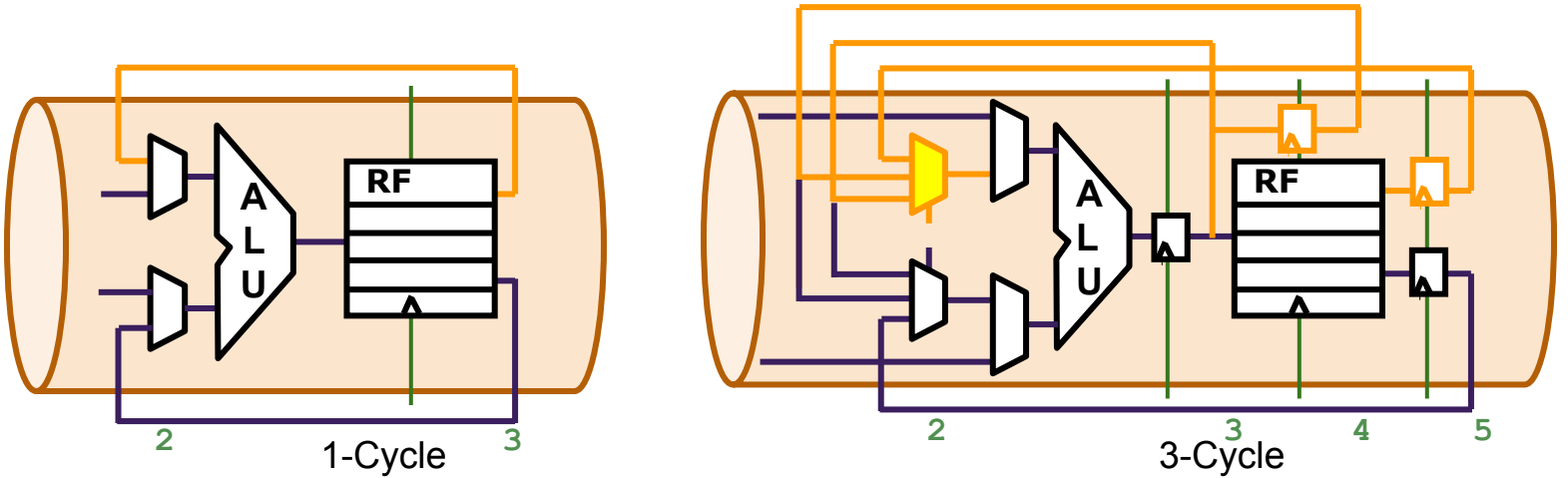
No bypass (ISA spec):

```
$reg1_value[M4_WORD_RANGE] =
   /cpu/regs[$reg]>>1$value;
```

Delay RF write.

Broken implementation!

Register Bypass



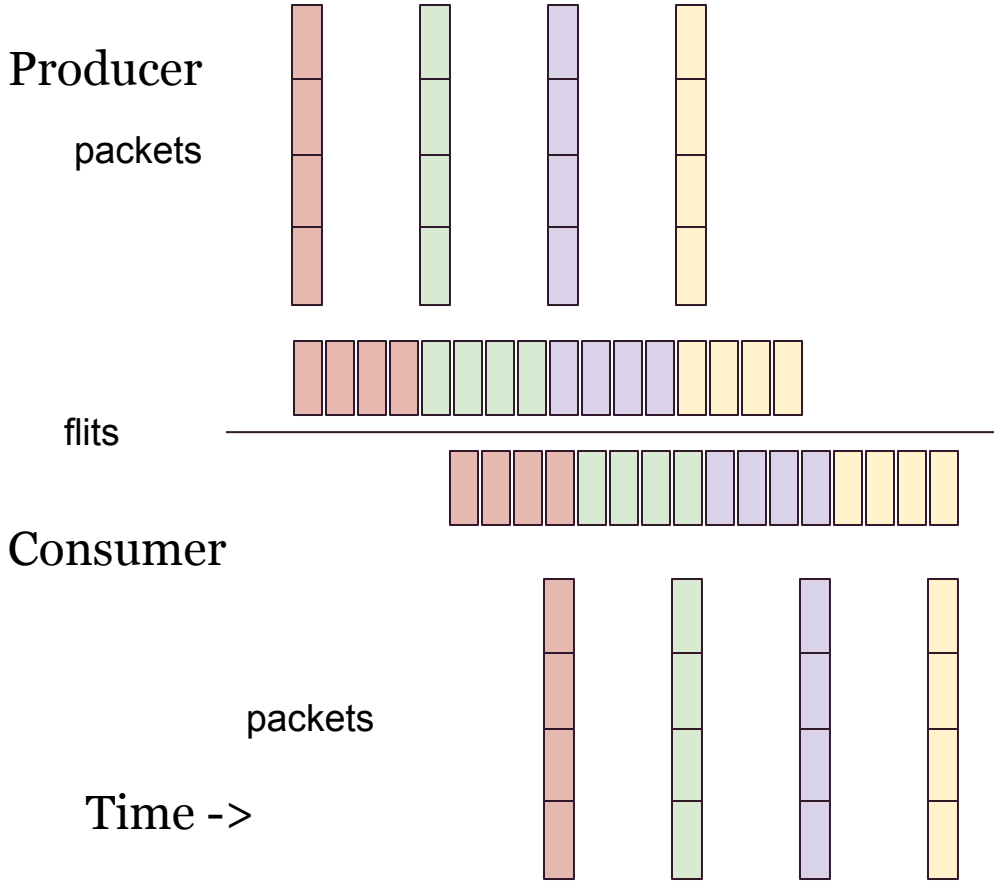
No bypass (ISA spec):

```
@3
   $reg1_value[M4_WORD_RANGE] =
      /cpu/regs[$reg]>>1$value;
```

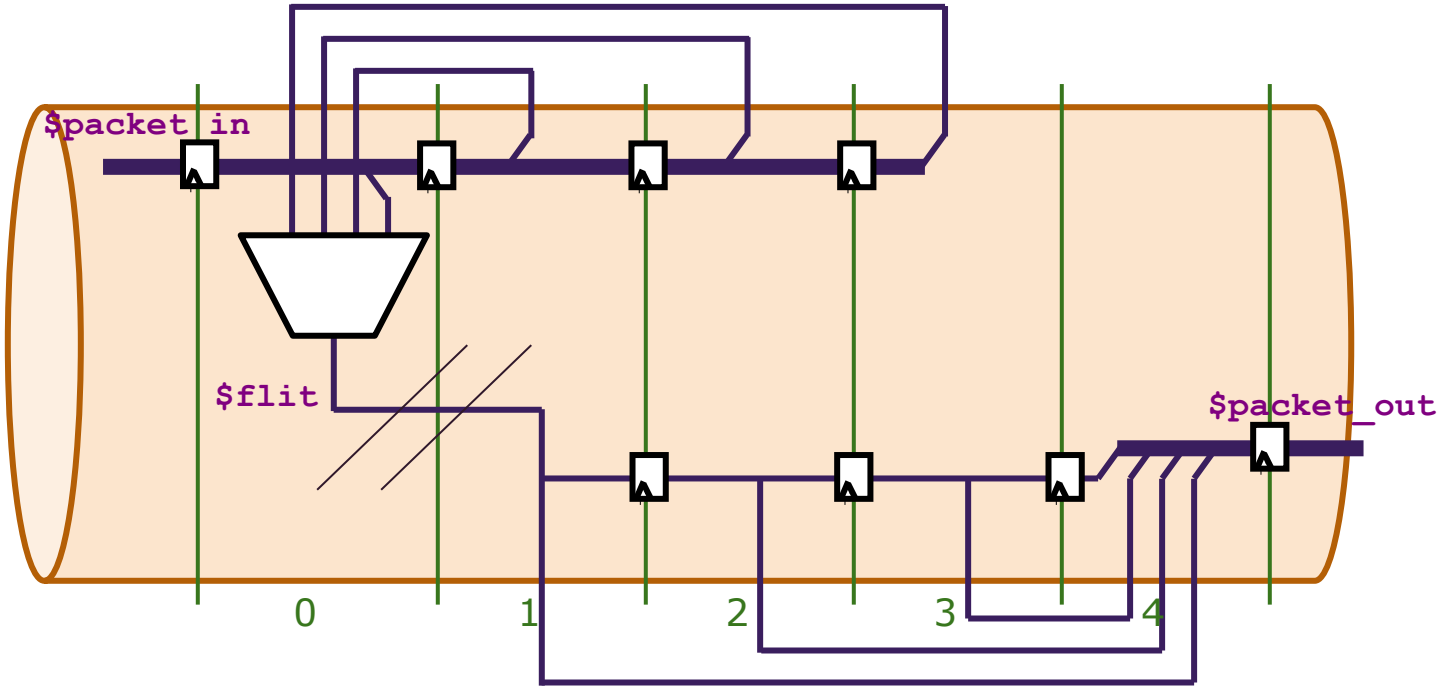
Two bypass stages:

```
@4
   $reg1_value[M4_WORD_RANGE] =
      (>>1$valid && (>>1$dest_reg == $reg1)) ? >>1$rslt :
      (>>2$valid && (>>2$dest_reg == $reg1)) ? >>2$rslt :
      /regs[$reg1]>>3$value;
```

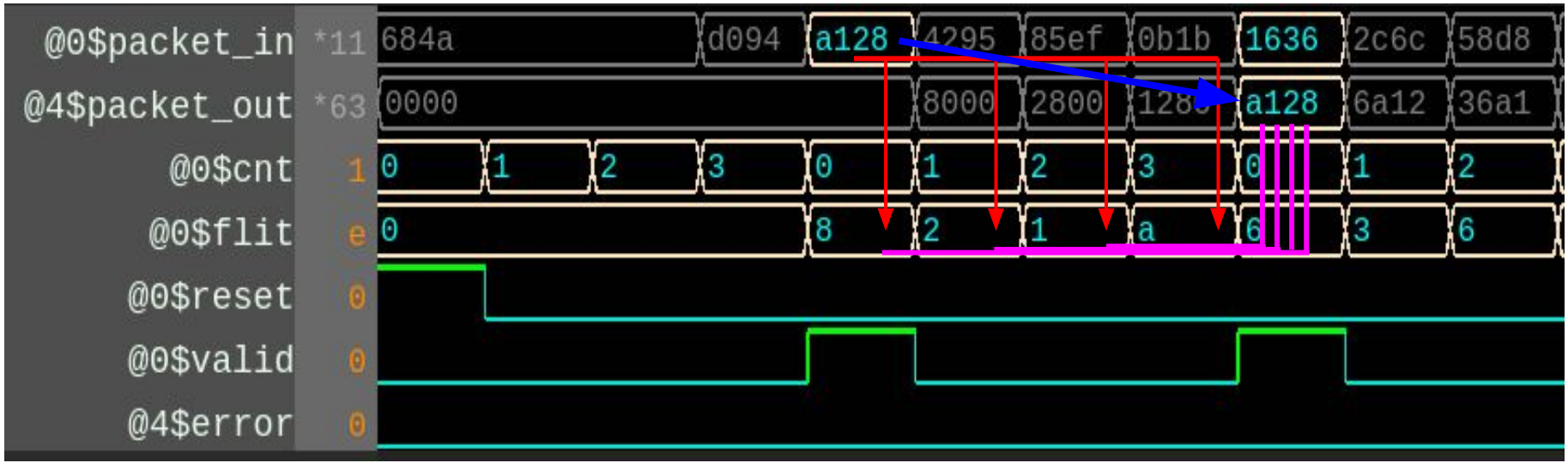
Time-Division Multiplexing Example



Time-Division Multiplexing Example

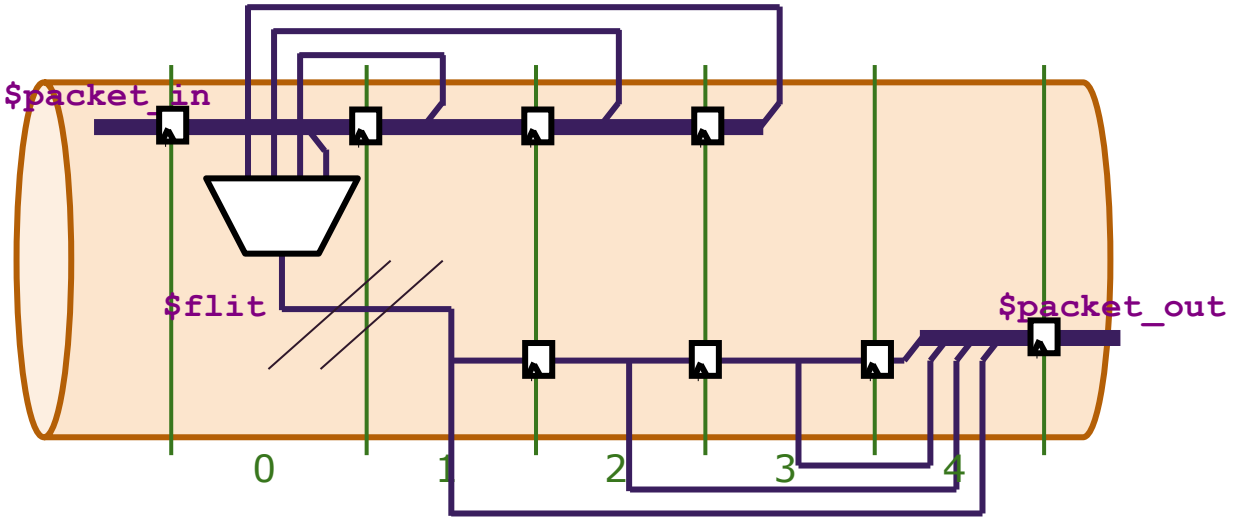


Time-Division Multiplexing Example



Lab: Time-Division Multiplexing

1. Load “Examples”/“Webinar”/“TDM Lab”.
2. Fill in the TL-Verilog for `$flit[3:0] = ...`



3. `$packet_out[15:0] = ...`

Agenda

- RISC-V Overview
- IP Design Methodology
- Design Concepts using TL-Verilog in Makerchip.com
  - Combinational Logic
  - Sequential Logic
  - Pipelines
  - Validity
  - Pipeline Interactions
  - Hierarchy
  - Elaboration
  - Interfacing with Verilog/SystemVerilog
  - State
  - Transactions
  - Verification
- Summary & Certification Challenge

Hierarchy in TL-Verilog

Verilog has many constructs for design hierarchy.

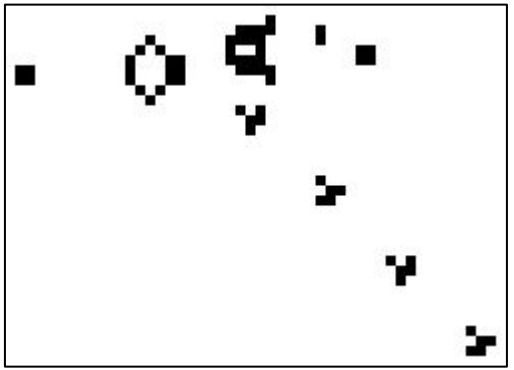
- modules
- packed arrays
- unpacked arrays
- for loops
- generate for loops

TL-Verilog provides

- `/scope[7:0]`

to generate appropriate Verilog.

Hierarchy -- Conway's Game of Life

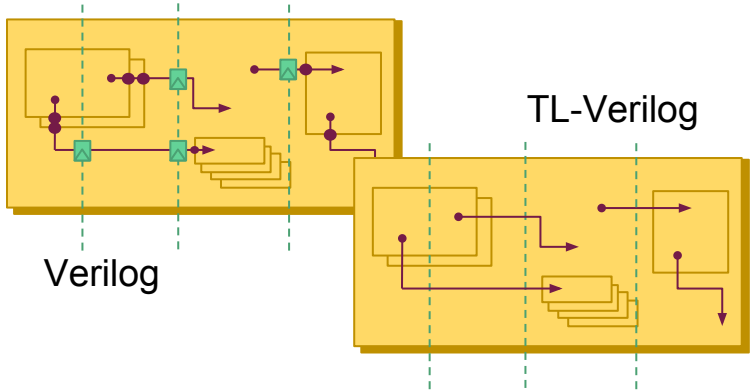


```
|default
   /yy[Y_SIZE-1:0]
      /xx[X_SIZE-1:0]
         @1
            // Sum left + me + right.
            $row_cnt[1:0] = ...;
            // Sum three $row_cnt's: above + mine + below.
            $cnt[3:0] = ...;

            // Init state.
            $init_alive[0:0] = *RW_rand_vect[...];

            $alive = $reset    ? $init_alive :
                     >>1$alive ? ($cnt >= 3 && $cnt <= 4) :
                                 ($cnt == 3);
```

(makerchip.com/sandbox/0/0Nkf06)

Interfaces in TL-Verilog (or lack thereof)

- Verilog modules have explicit interfaces.
- Cross-module references are restrictive and discouraged.



- TL-Verilog scope requires no interfaces.
- Signals are referenced where they are produced.

Eg:

```
$core2_sig[1:0] =
   /core[2]|my_pipe/trans>>1$my_sig[3:2];
```

Accumulate:

```
$any_valid = | /slice[*]$valid;
```

⇒ No interface parameterization for TL-Verilog IP!

Hierarchy in WARP-V

Decode: Extracting src reg fields

```
968     // Output signals.
969     /src[2:1]
970        // Reg valid for this source, based on instruction type.
971        $is_reg = /instr$is_r_type || /instr$is_r4_type || (/instr$is_i_type && (#sr
972        $reg[M4_REGS_INDEX_RANGE] = (#src == 1) ? /instr$raw_rs1 : /instr$raw_rs2;
973
```

Execute: Referencing register values

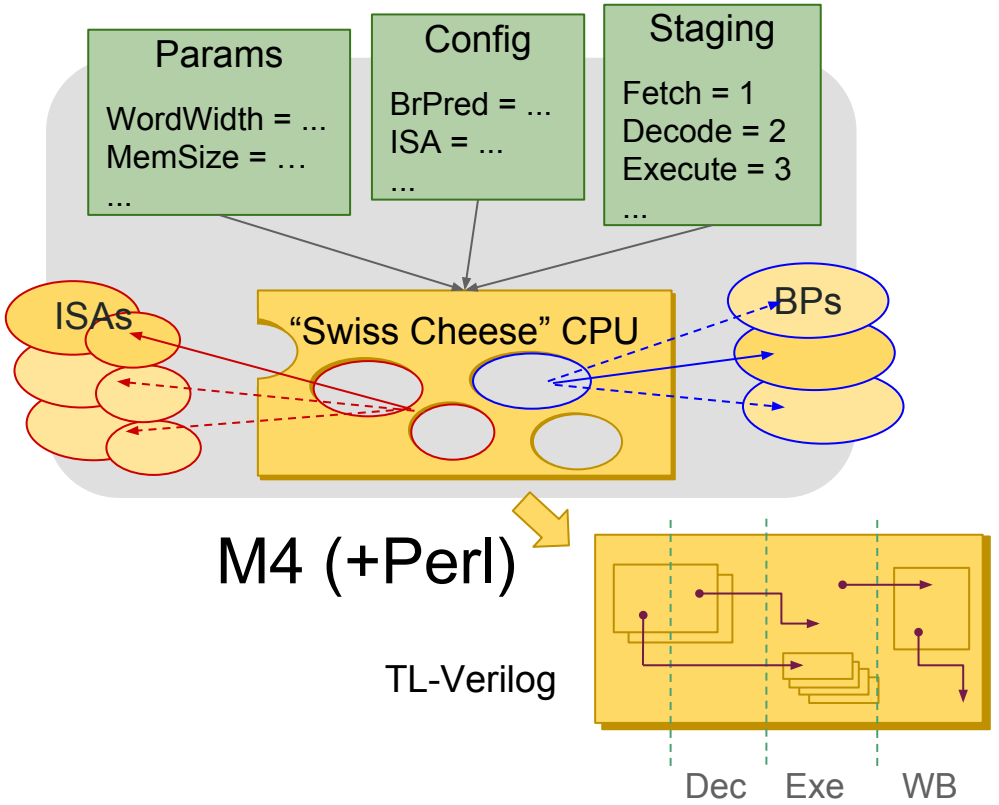
```
1029        $addi_rslt[M4_WORD_RANGE]    = /src[1]$reg_value + $raw_i_imm;  // Note: this
1030        $ori_rslt[M4_WORD_RANGE]     = /src[1]$reg_value | $raw_i_imm;
1031        $add_sub_rslt[M4_WORD_RANGE] = /src[1]$reg_value + /src[2]$reg_value;
```

Agenda

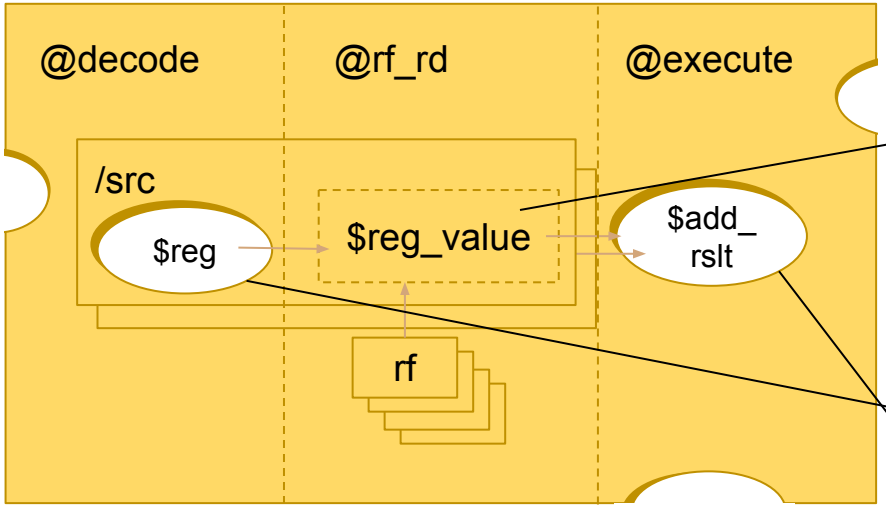
- RISC-V Overview
- IP Design Methodology
- Design Concepts using TL-Verilog in Makerchip.com
  - Combinational Logic
  - Sequential Logic
  - Pipelines
  - Validity
  - Pipeline Interactions
  - Hierarchy
  - Elaboration
  - Interfacing with Verilog/SystemVerilog
  - State
  - Transactions
  - Verification
- Summary & Certification Challenge

WARP-V Configuration

- Native elaboration features are TBD for TL-Verilog.
- Macro preprocessing w/ M4 provides a workable solution in the meantime:
  - Parameterization (incl. staging)
  - Library inclusion
  - Modularity & reuse
  - Configuration (component selection)
  - Code construction



“Swiss-Cheese” Design




```
// Read each src operand from reg file.
@rf_rd  // stage
   /src[2:1]  // for each src operand
      $reg_value[M4_WORD_RANGE] =
            |cpu/rf[$reg]$Value;  // RF rd
```

ISA-Specific:

```
@decode
   /src[*]
      $reg[5:0] = (#src == 1)
                  ? $instr[19:15]
                  : $instr[24:20];
@execute
   $add_rslt[31:0] =
        /src[1]$reg_value +
        /src[2]$reg_value;
```

- Single ISA-specific instantiation fills multiple holes
- “Lexically re-entrant” scopes

Parameterized Register Bypass

1, 2, 3, or 4 cycles (based on M4_REG_BYPASS_STAGES)

```
$reg_value[M4_WORD_RANGE] =
   // Bypass stages:
   m4_ifexpr(M4_REG_BYPASS_STAGES >= 1, ['(/instr>>1$dest_reg_valid && (/instr>>1$dest_reg == $reg)) ? /instr>>1$rslt :'])
   m4_ifexpr(M4_REG_BYPASS_STAGES >= 2, ['(/instr>>2$dest_reg_valid && (/instr>>2$dest_reg == $reg)) ? /instr>>2$rslt :'])
   m4_ifexpr(M4_REG_BYPASS_STAGES >= 3, ['(/instr>>3$dest_reg_valid && (/instr>>3$dest_reg == $reg)) ? /instr>>3$rslt :'])
   /instr/regs[$reg]>>M4_REG_BYPASS_STAGES$Value;
```

(Could be a loop)

Agenda

- RISC-V Overview
- IP Design Methodology
- Design Concepts using TL-Verilog in Makerchip.com
  - Combinational Logic
  - Sequential Logic
  - Pipelines
  - Validity
  - Pipeline Interactions
  - Hierarchy
  - Elaboration
  - Interfacing with Verilog/SystemVerilog
  - State
  - Transactions
  - Verification
- Summary & Certification Challenge

Verilog within TL-Verilog

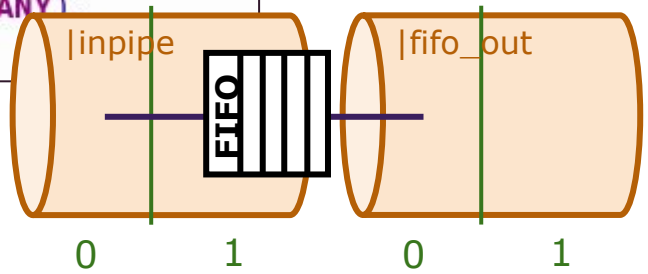
Verilog functions, macros, modules, assertions, and any other Verilog code, can be used in \TLV context.

Module:

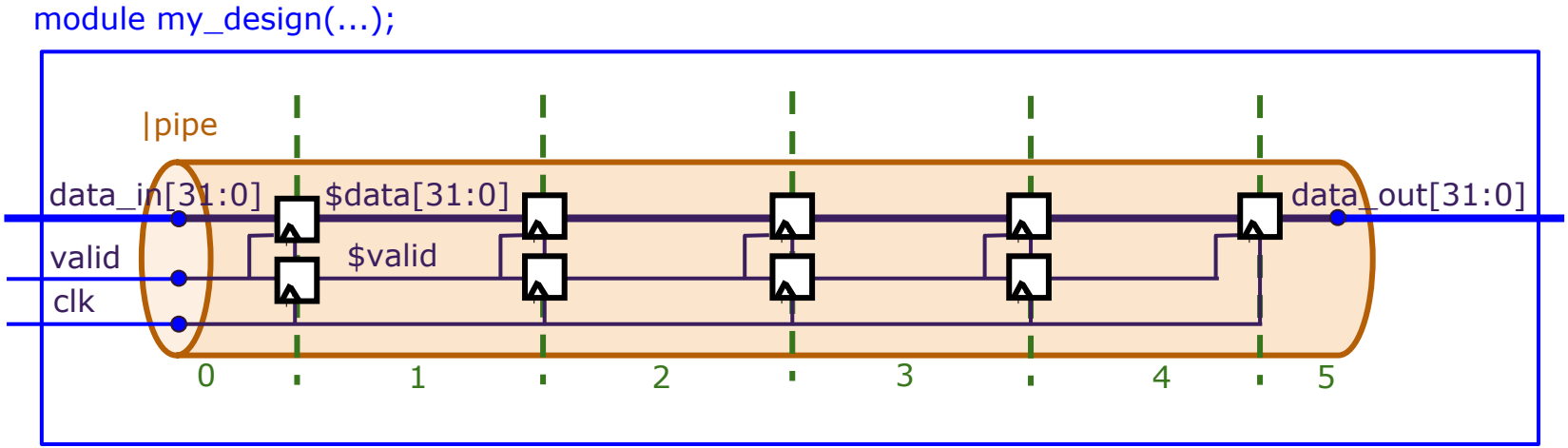
```
simple_bypass_fifo #(.WIDTH(8), .DEPTH(6))
   fifo(.clk(clk), .reset(/ring_stop|inpipe>>1$reset),
        .push(/ring_stop|inpipe>>1$trans_valid),
        .data_in(/ring_stop|inpipe>>1$ANY),
        .pop(/ring_stop|fifo_out>>0$trans_valid),
        .data_out(/ring_stop|fifo_out>>0$$ANY)
        .cnt($$cnt[2:0]));
```

Etc.:

```
\SV_plus
   always_ff @(posedge clk)
      if ($valid) \$display("\%d", $sig1);
\always_comb
   \$display("\%d", $sig2);
```



TL-Verilog within Verilog/SystemVerilog

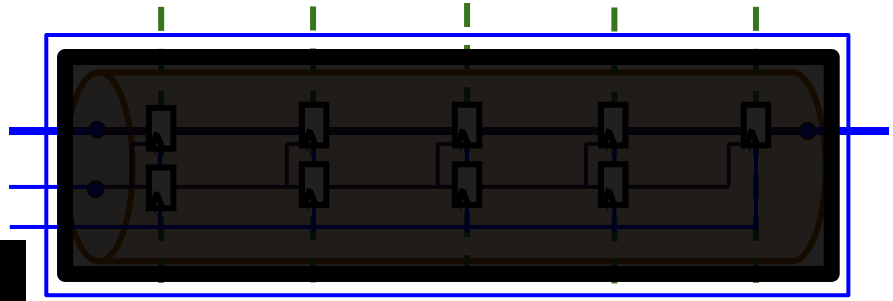


File Structure

```
\TLV_version 1b tl-x.org
\SV
module my_design (
   input clk,
   input valid,
   input [31:0] data_in,
   output [31:0] data_out
);
\TLV
```

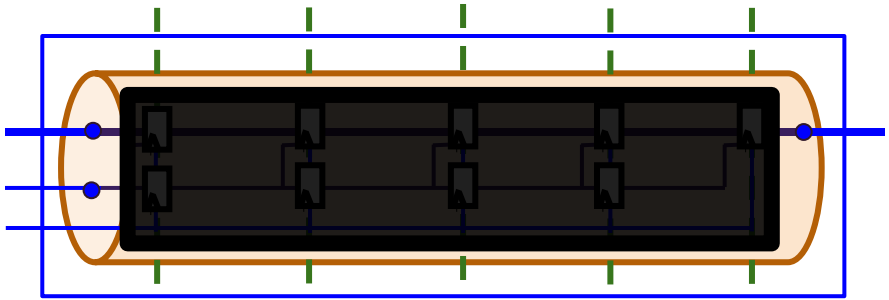
[redacted]

```
\SV
endmodule
```



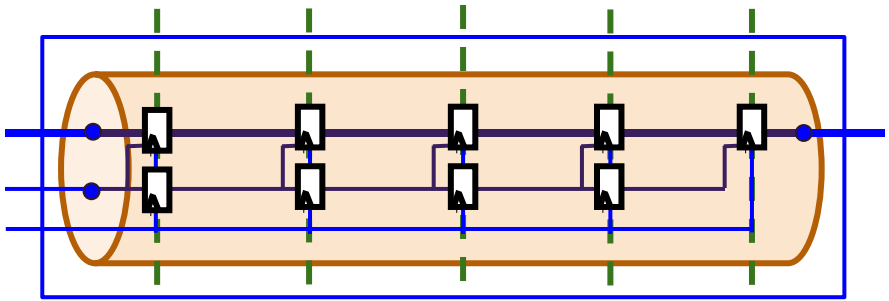
Interfacing with Verilog/SystemVerilog

```
\TLV_version 1b tl-x.org
\SV
module my_design (
   input clk,
   input valid,
   input [31:0] data_in,
   output [31:0] data_out
);
\TLV
   |pipe
      @0
!        $valid = *valid;
         ?$valid
!           $data[31:0] = *data_in;
[redacted]
      @5
!        *data_out = $data;
\SV
endmodule
```



Remaining Logic

```
\TLV_version 1b tl-x.org
\SV
module my_design (
   input clk,
   input valid,
   input [31:0] data_in,
   output [31:0] data_out
);
\TLV
   |pipe
      @0
!        $valid = *valid;
         ?$valid
!           $data[31:0] = *data_in;
      // Logic
      // There is none.
      @5
!        *data_out = $data;
\SV
endmodule
```



Makerchip File Structure

[Show in Makerchip]

Checkpoint

You can now develop anything w/ TL-Verilog!

For free:

- Open-source: TLV-Comp
- Commercial: SandPiper™ w/ Starter License

...but it gets even better (on Makerchip or w/ educational or paid license).

Agenda

- RISC-V Overview
- IP Design Methodology
- Design Concepts using TL-Verilog in Makerchip.com
  - Combinational Logic
  - Sequential Logic
  - Pipelines
  - Validity
  - Pipeline Interactions
  - Hierarchy
  - Elaboration
  - Interfacing with Verilog/SystemVerilog
  - State
  - Transactions
  - Verification
- Summary & Certification Challenge

State

Low-level view of state:

- State is: Flip-flops and memories, aka “state elements”
- State of the machine = values held in state elements.
- State is modified by combinational logic each cycle to get to new state.

High-level view of state:

State is:

- In a CPU: memory, reg file, CSRs
- In a game of chess: the board, the next player ID

State is modified by: transactions

Transactions are:

- CPU: instructions
- Chess: a move

In-flight transactions are also state

Save/Restore

Important test/debug capability.

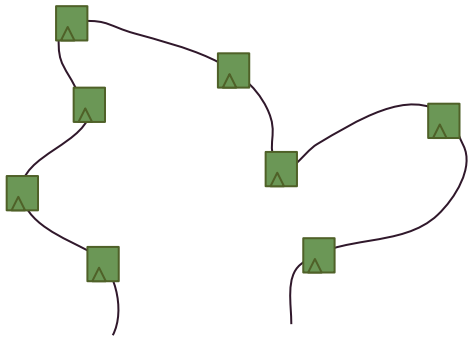
What do we save? -- State

Low-level:

- Every flip-flop (SCAN) & array

High-level:

- High-level state (arrays & regs)
  + in-flight transactions



SCAN Chain

Quiescing:

- Stop issuing transactions
- In-flight transactions => 0 (`?$valid` => 0)
- Save high-level state only.

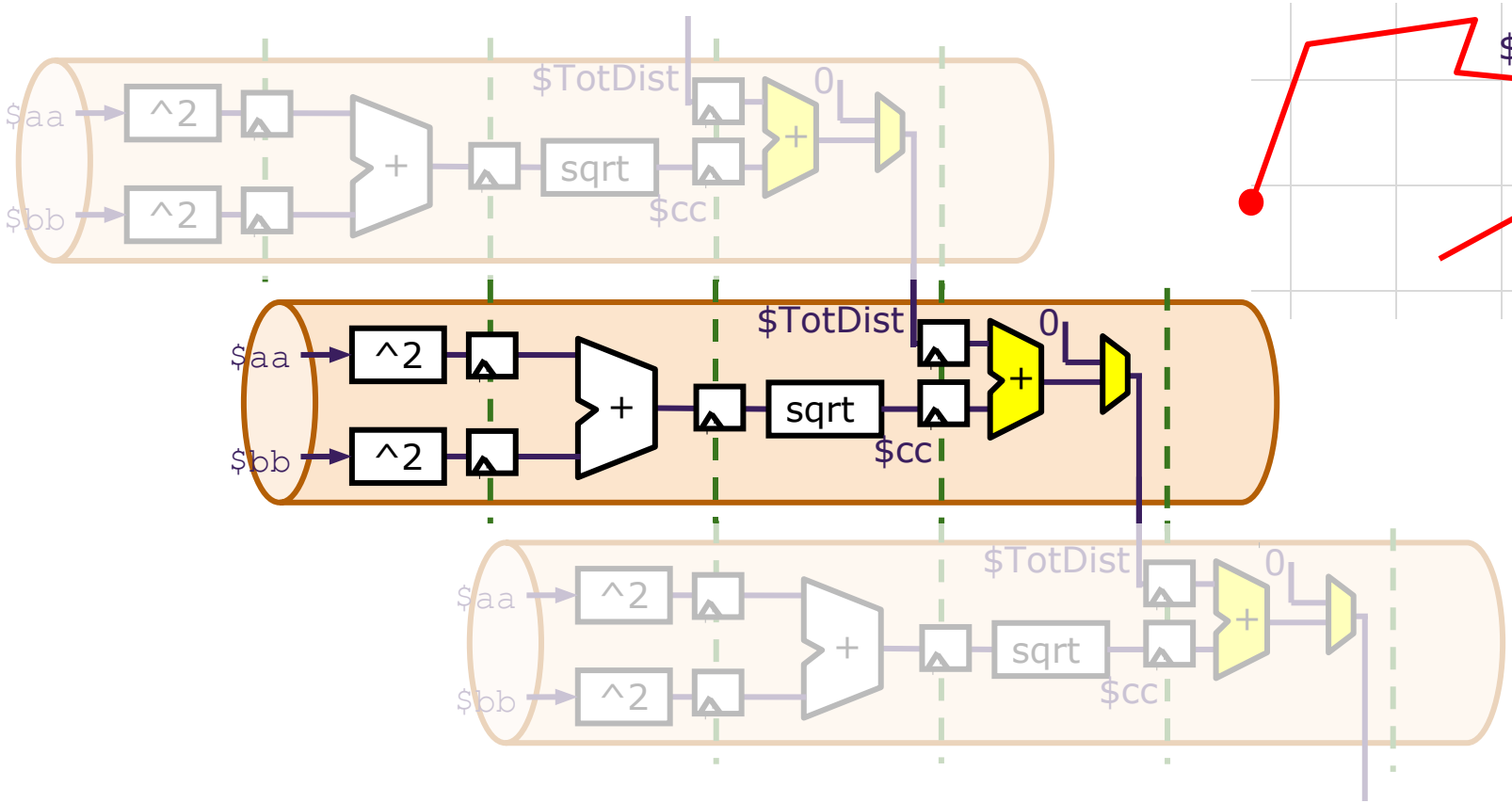
Can capture periodic checkpoints and reproduce bugs in simulation.

Categorizing Flip-Flops

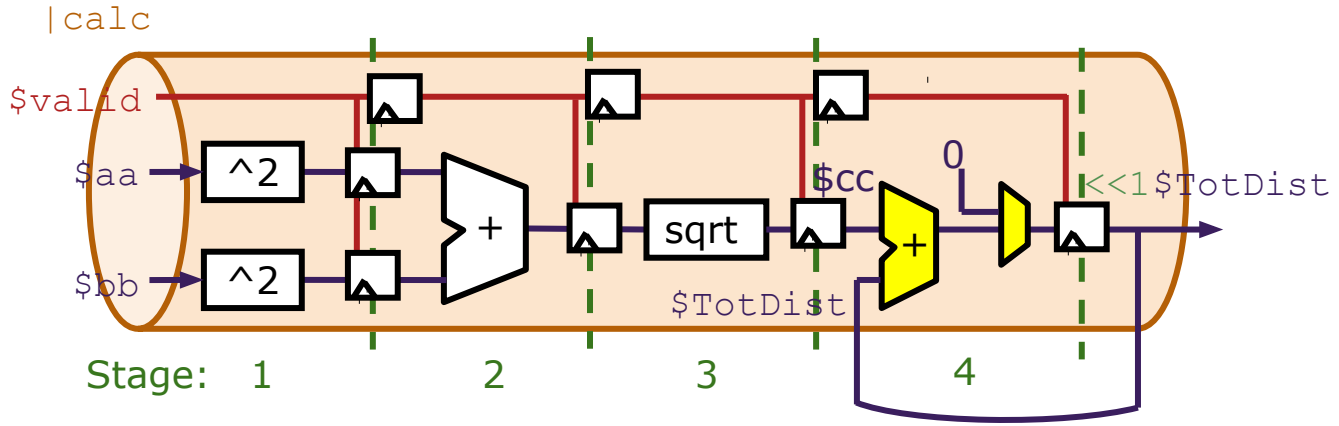
	State	Staging
Example	`$RegValue[63:0]`	`$instr_immediate[6:0]`
Nature	Persistent	Transient
Value under ?$valid == 0	Retained	DONT_CARE
Reset	Fixed value	DONT_CARE (`?$valid` == 0 suggested)
Quiescent Value	Retained	DONT_CARE (`?$valid` == 0)

- Valuable distinction for reset and debug.

State - Distance Accumulator



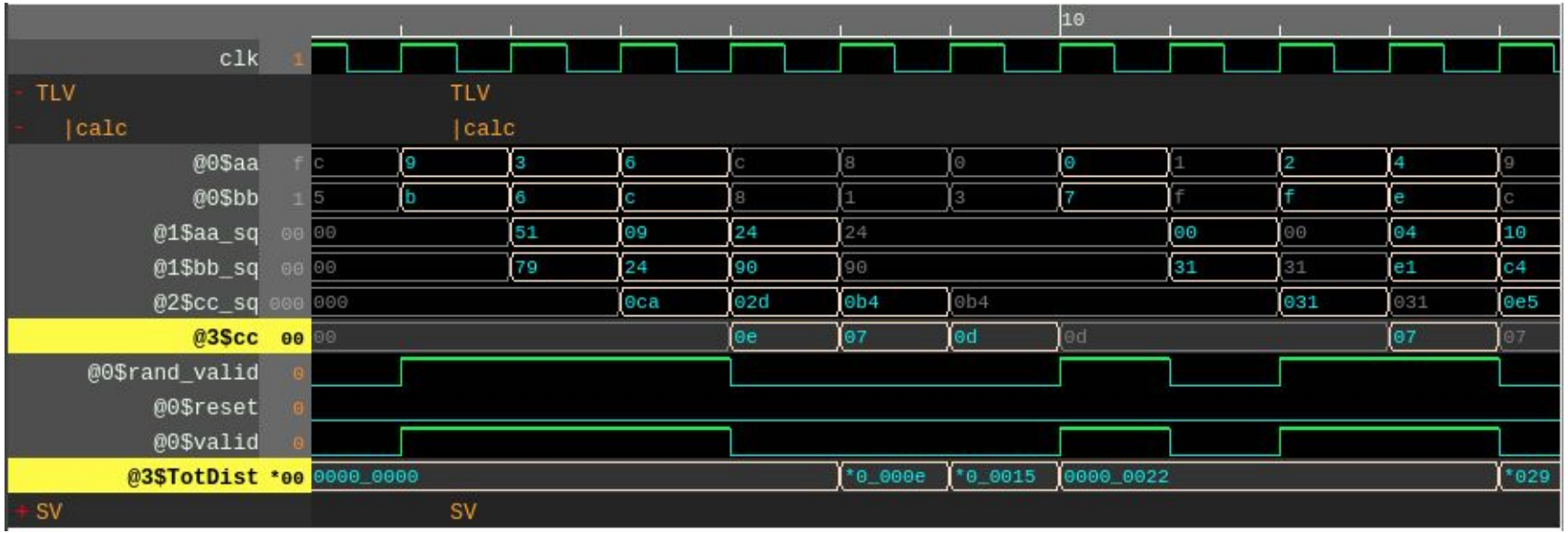
State - Distance Accumulator




```
|calc
   ?$valid
      @1
         $aa_sq[31:0] = $aa * $aa;
         $bb_sq[31:0] = $bb * $bb;
      @2
         $cc_sq[31:0] = $aa_sq + $bb_sq;
      @3
         $cc[31:0] = sqrt($cc_sq);
      @4
         $TotDist[31:0] <= $reset ? 0 : $TotDist + $cc;
```

Shorthand for: `<<1$TotDist[31:0] =`

$TotDist Waveform

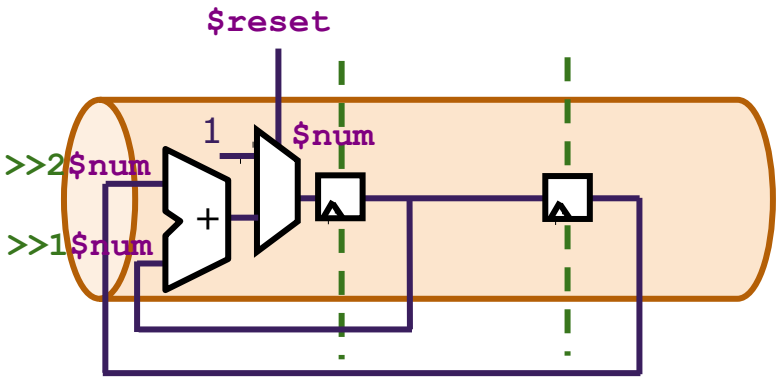


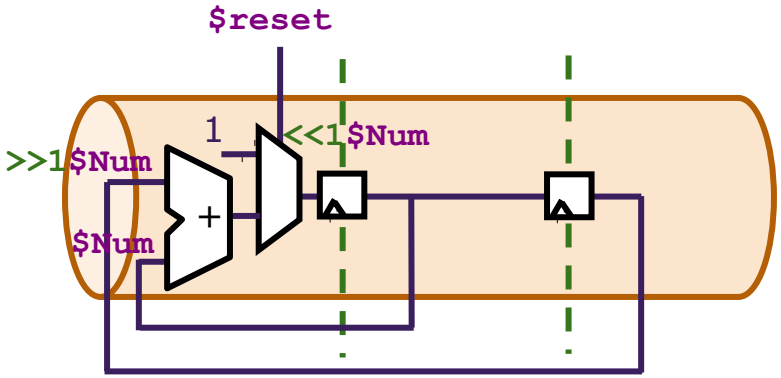
State in WARP-V

```
// =======
// Next PC
// =======

$Pc[M4_PC_RANGE] <=
   $reset ? M4_PC_CNT'b0 :
   >>M4_BRANCH_BUBBLES$valid_mispred_branch ? >>M4_BRANCH_BUBBLES$branch_target :
   >>M4_JUMP_BUBBLES$valid_jump ? >>M4_JUMP_BUBBLES$jump_target :
   >>m4_eval(M4_REPLAY_LATENCY-1)$replay ? >>m4_eval(M4_REPLAY_LATENCY-1)$Pc :
   $returning_ld ? $RETAIN :  // Returning load, so next PC is the previous next PC
            $Pc + M4_PC_CNT'b1;
```

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);
```

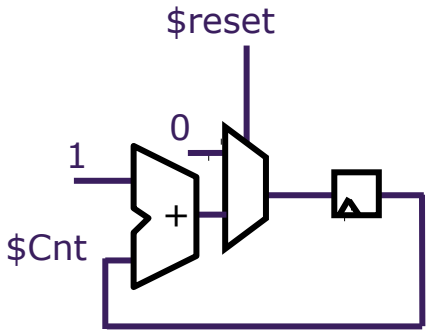
```
$Num[31:0] <= $reset ? 1 : ($Num + >>1$Num);
```

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

Lab: Counter as State

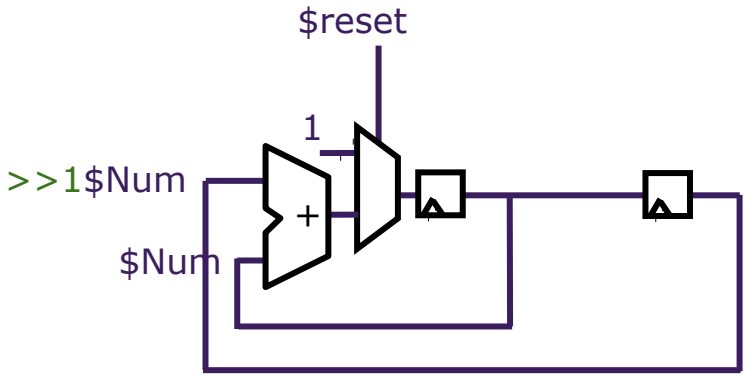
Lab:

1. Design a free-running counter:



2. Compile and explore.

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




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

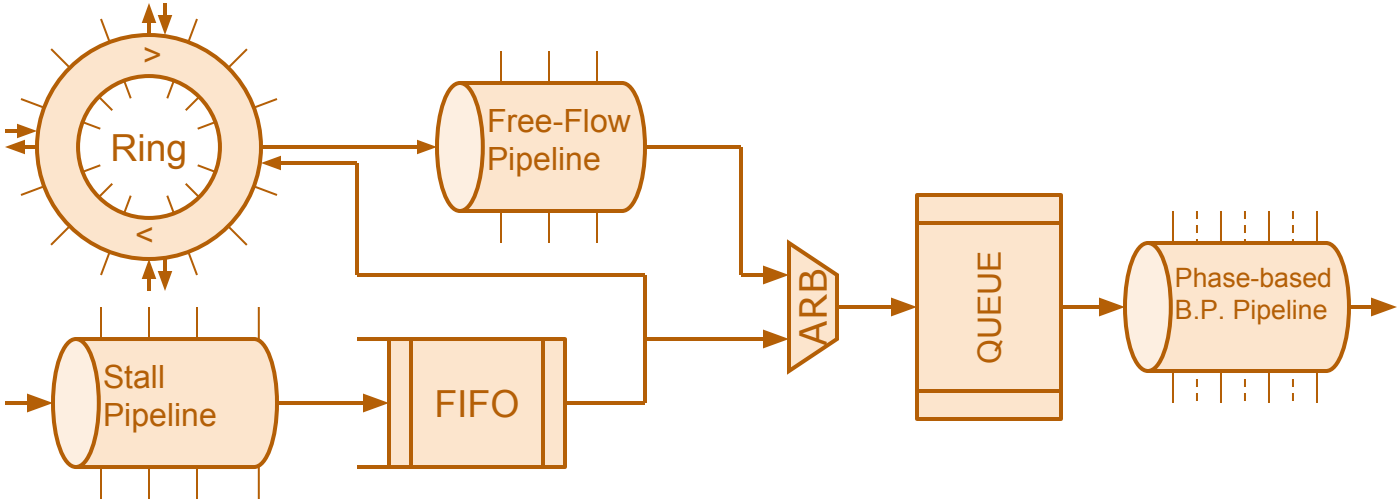
3-space indentation
(no tabs)

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

Agenda

- RISC-V Overview
- IP Design Methodology
- Design Concepts using TL-Verilog in Makerchip.com
  - Combinational Logic
  - Sequential Logic
  - Pipelines
  - Validity
  - Pipeline Interactions
  - Hierarchy
  - Elaboration
  - Interfacing with Verilog/SystemVerilog
  - State
  - Transactions
  - Verification
- Summary & Certification Challenge

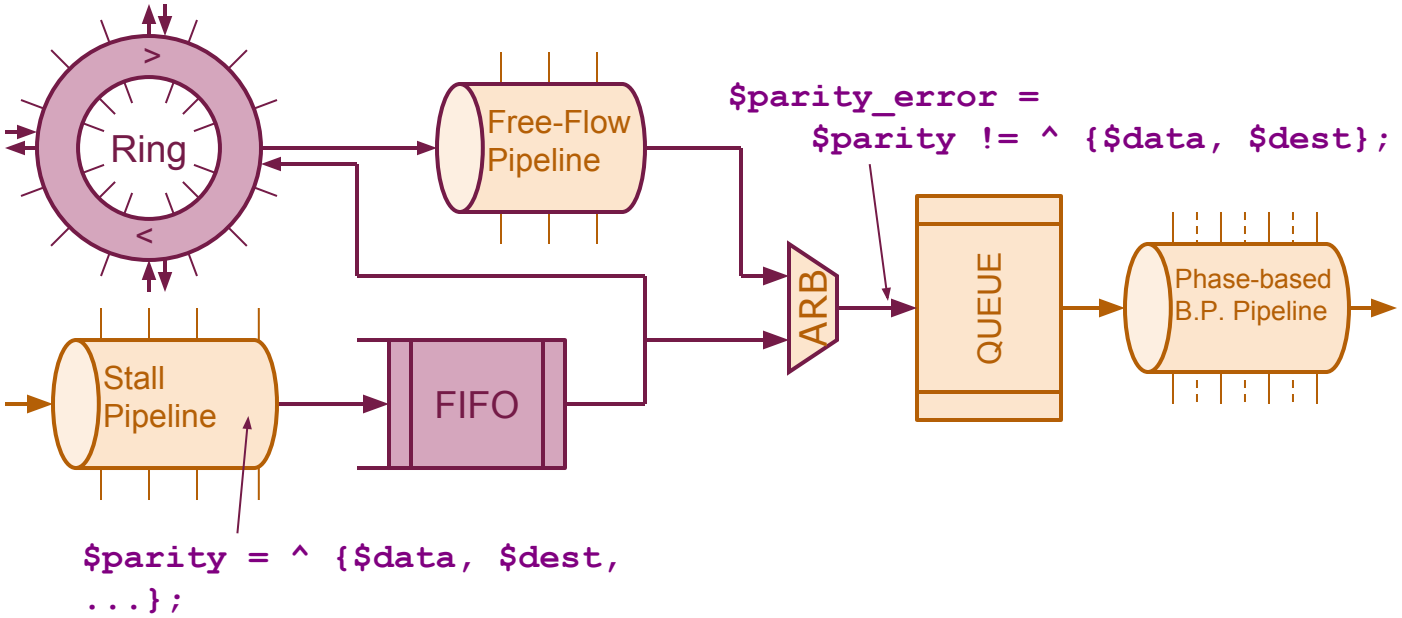
Transaction Flow



- Flow constructed from pre-verified library components. (~100 lines)
- Transaction logic added into this context.

Transaction Flow: Scenario

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



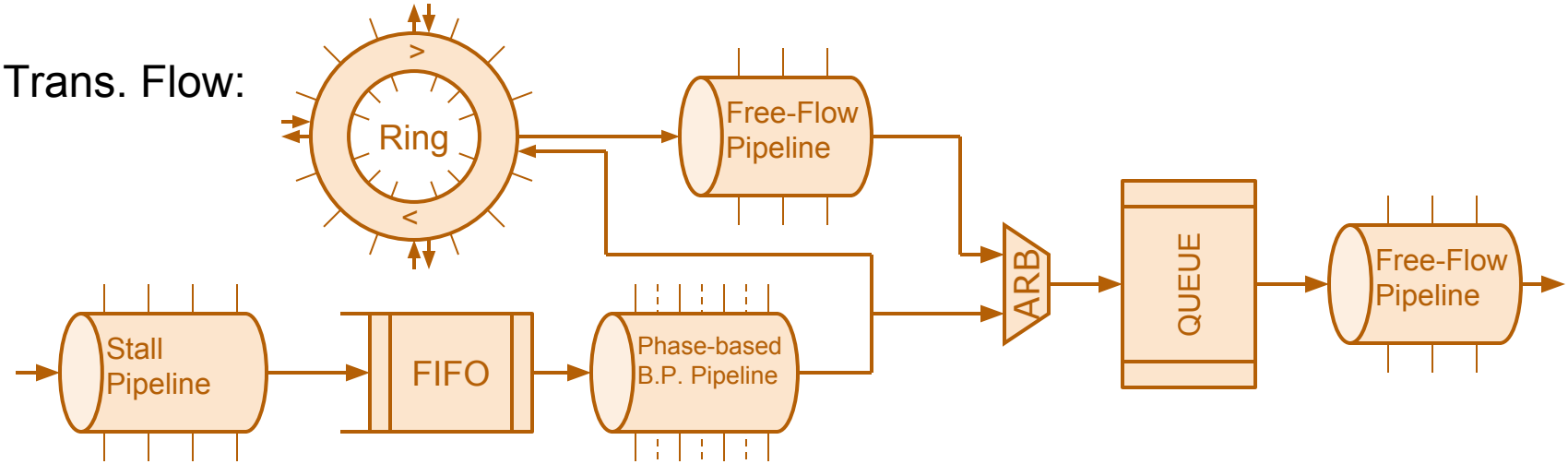
2 lines of TL-Verilog vs. 100s of lines of RTL change (across files).

Demo in Makerchip

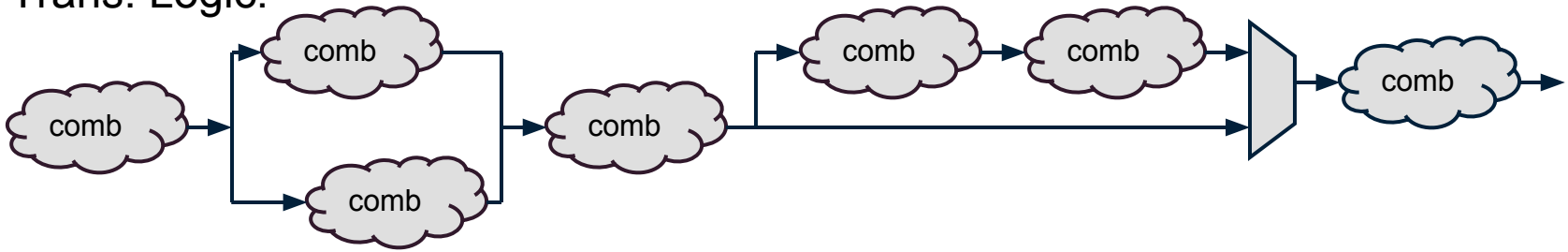
[Demo Flow Tutorial in Makerchip]

Transaction Flow

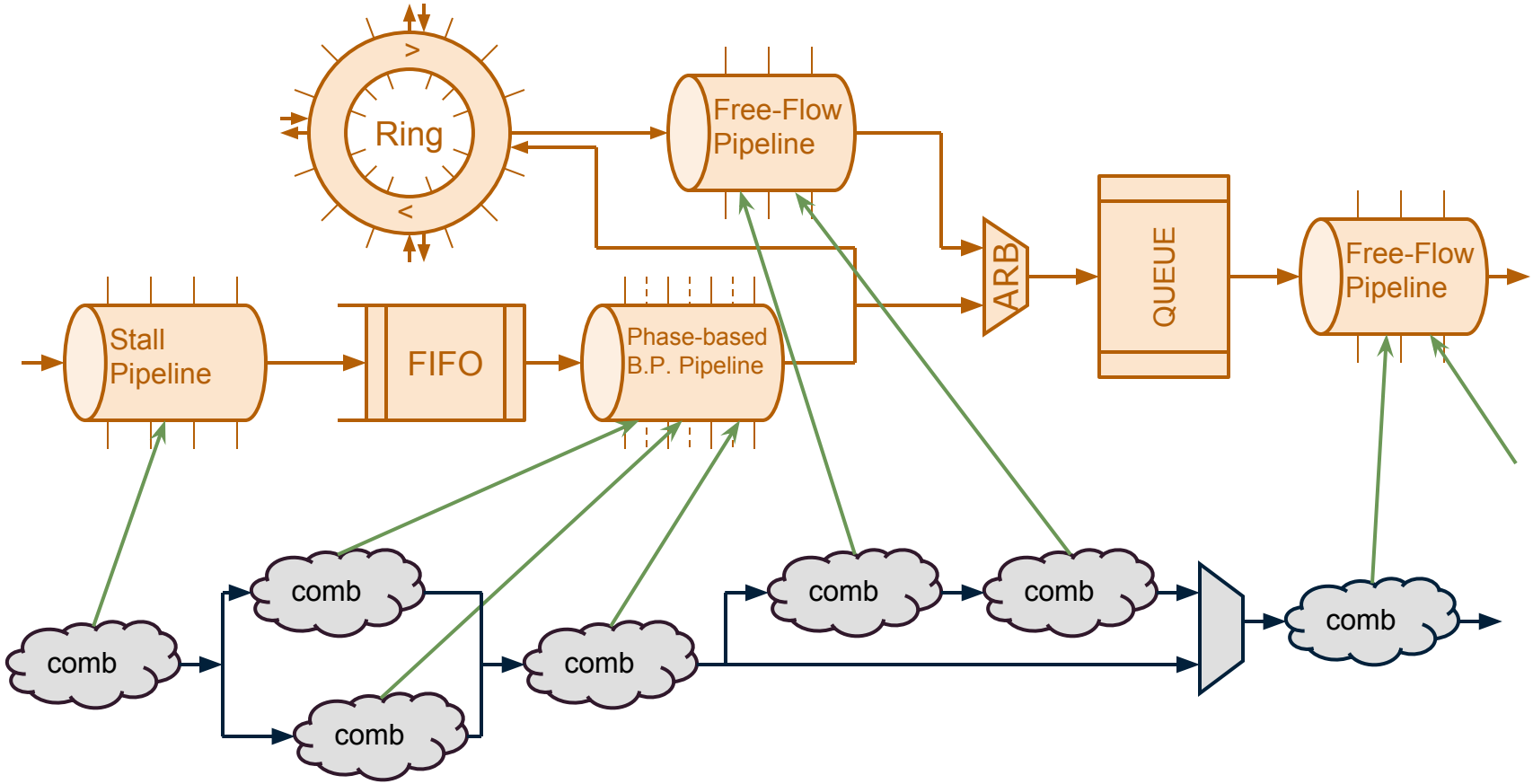
Trans. Flow:



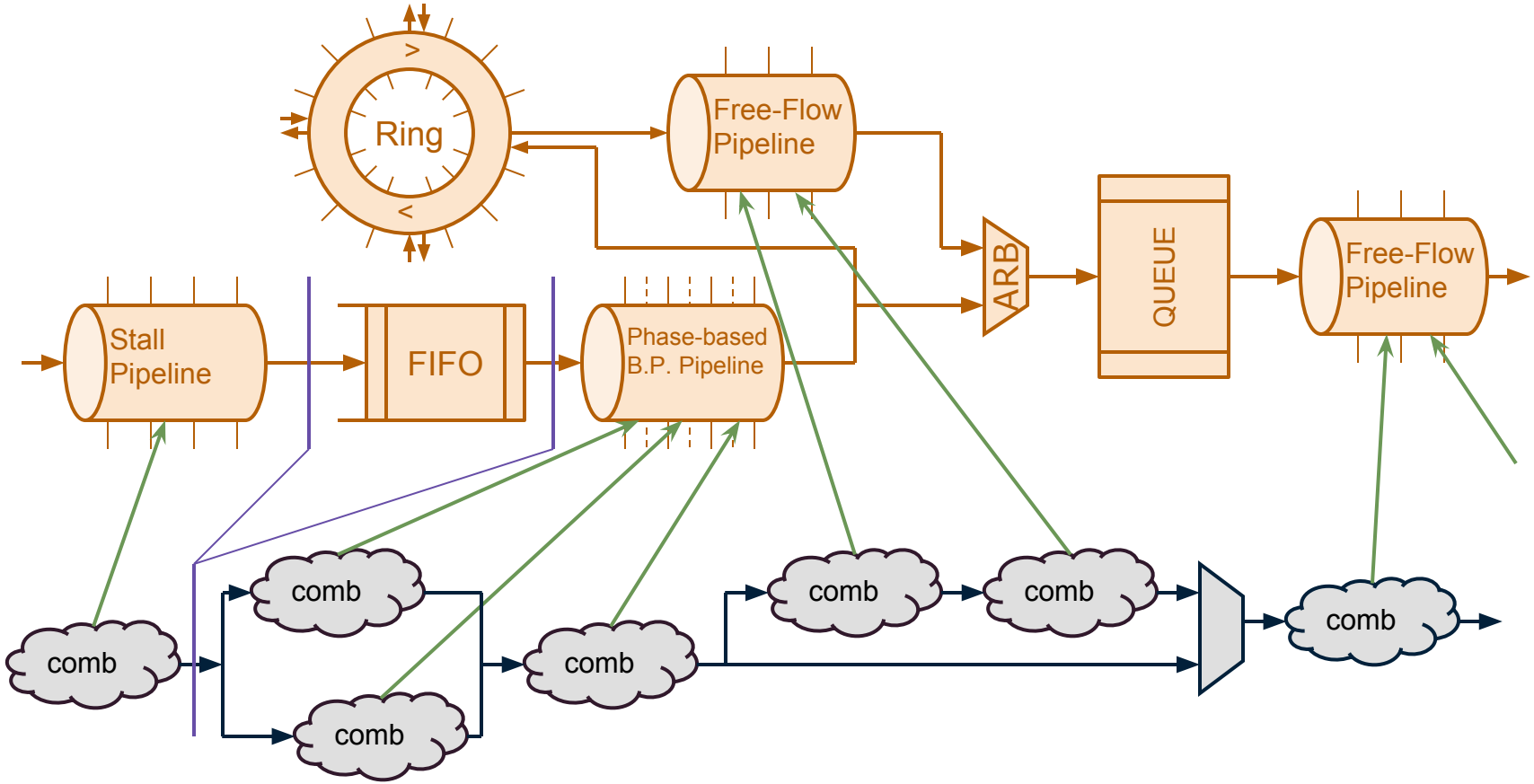
Trans. Logic:



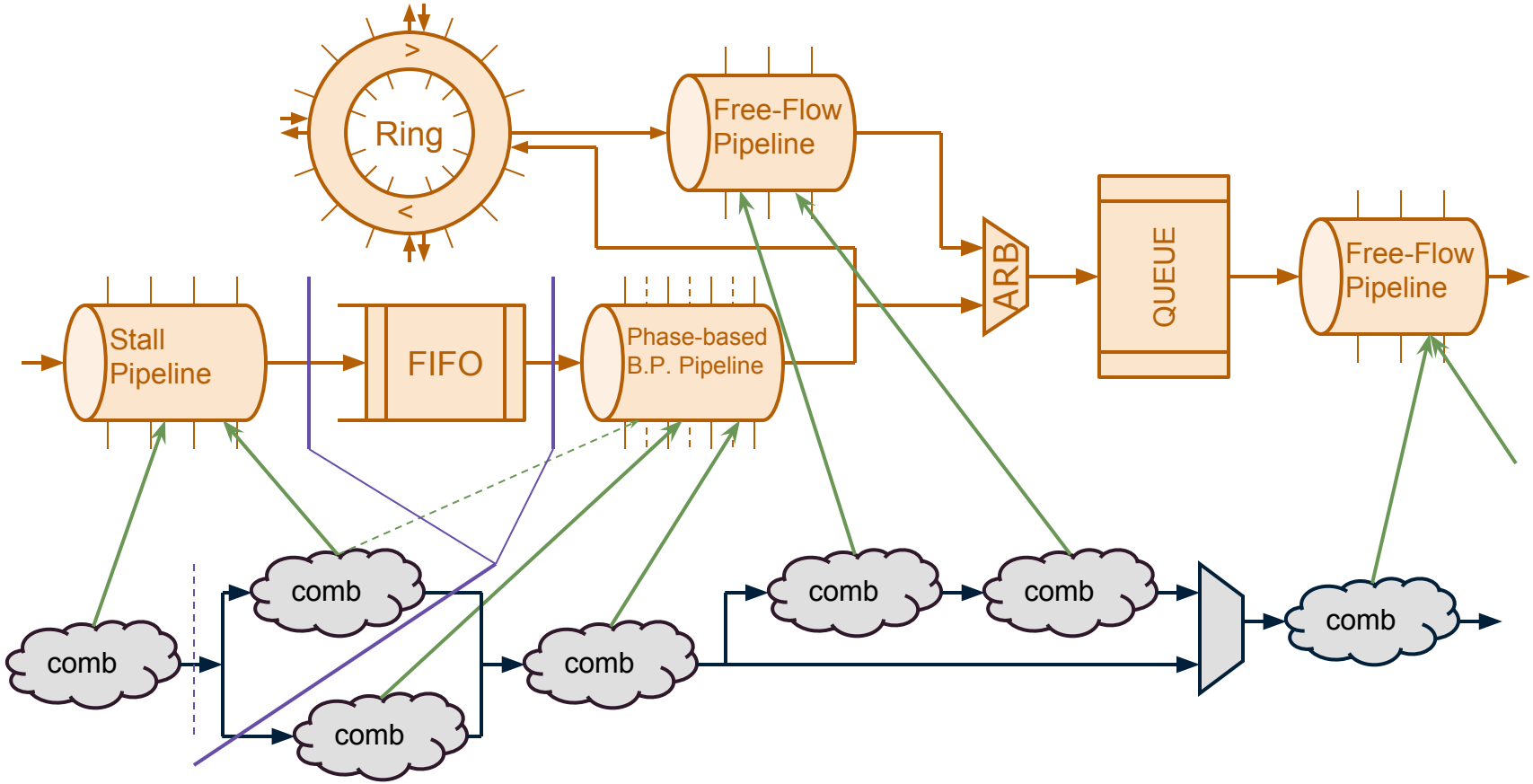
Transaction Flow



Transaction Flow: Retiming



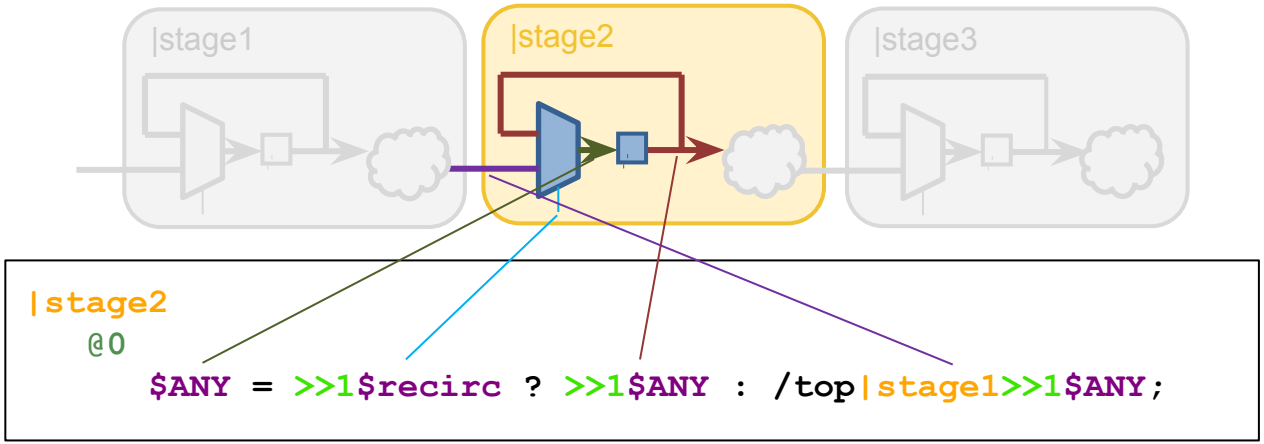
Transaction Flow: Retiming



Transaction Flow: Mechanism -- $ANY

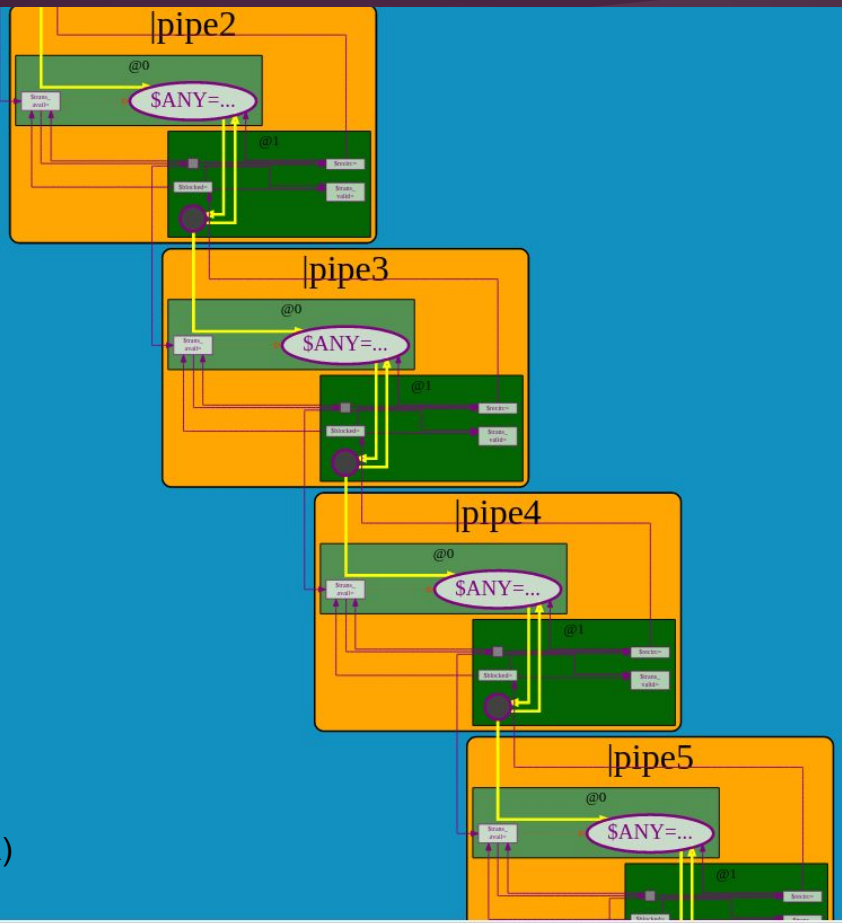
- Example: Back-pressured pipeline

Bold wires carry transactions, and are referenced as **$ANY**.



- **$ANY** is a wildcard rule. If a pipesignal is needed that is not available, it can be produced by a **$ANY** expression.

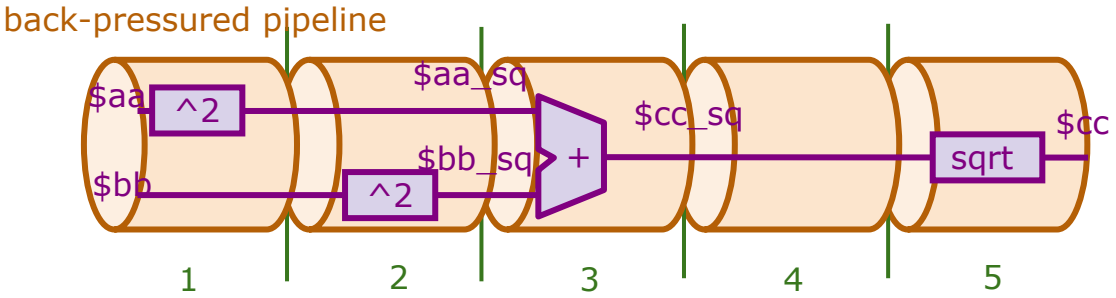
Back-Pressured Pipeline



(makerchip.com/sandbox/0/0Elh3R)

Lab: Back-Pressured Pythagorean

"Lexical Re-entrance" enables insertion of logic into flow.



- Open "Example"/"Backpressured Pipeline Macro"/"Backpressured Pythagorean Calculation"
- Modify to match above.
- Find **`$cc_sq`** in Diagram and highlight its inputs (<Ctrl>-click for multiple).

Agenda

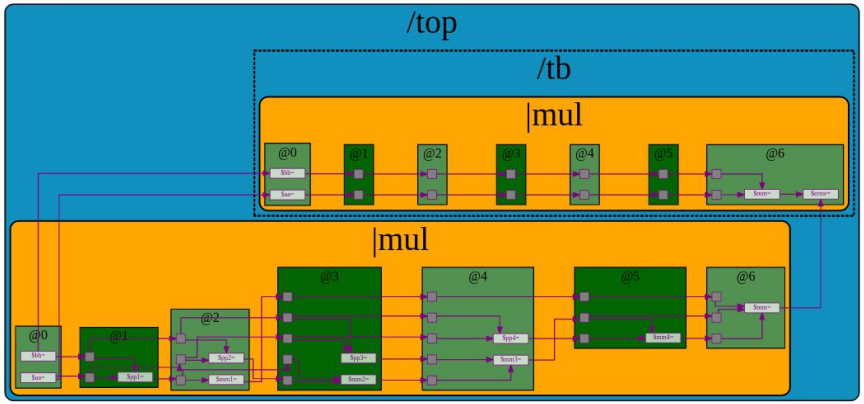
- RISC-V Overview
- IP Design Methodology
- Design Concepts using TL-Verilog in Makerchip.com
  - Combinational Logic
  - Sequential Logic
  - Pipelines
  - Validity
  - Pipeline Interactions
  - Hierarchy
  - Elaboration
  - Interfacing with Verilog/SystemVerilog
  - State
  - Transactions
  - Verification
- Summary & Certification Challenge

Verification Methodology

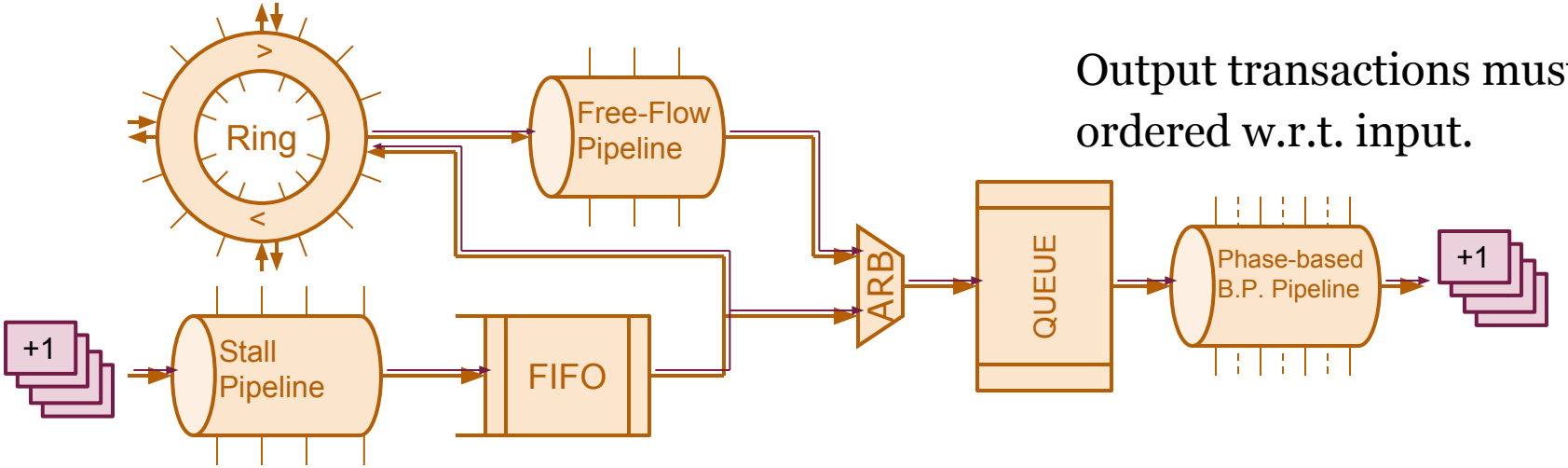
No proposed changes to verification methodology today, but potential for tomorrow:

- Timing-abstract and transaction-aware assertions/checkers/coverage, resilient to logic retiming
- Verify transaction and flow separately
  - Transaction: Dummy flow
  - Flow: Dummy transaction
- Synthesizable testbench

Partial-Products Multiply



Checker Example



Output transactions must be ordered w.r.t. input.

- Can be difficult to reconcile which transaction is coming out.
- Checker
  - count per dest at input; include count in transaction
  - count per source at output (maybe `$src` isn't available in H/W -- no problem)
  - compare counts

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

Agenda

- RISC-V Overview
- IP Design Methodology
- Design Concepts using TL-Verilog in Makerchip.com
  - Combinational Logic
  - Sequential Logic
  - Pipelines
  - Validity
  - Pipeline Interactions
  - Hierarchy
  - Elaboration
  - Interfacing with Verilog/SystemVerilog
  - State
  - Transactions
  - Verification
- Summary & Certification Challenge

Implications

Abstract context:

- Transactions & Transaction Flow
- State
- Validity

Help to:

- Organize design
- Reason about design
- Visualize design (more to come)
- Safely modify design

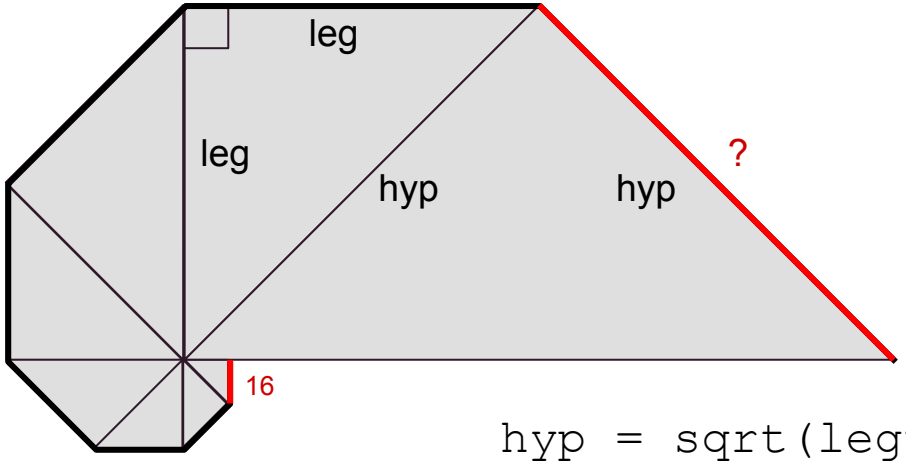
Separation of concerns

- Behavior from implementation
- State flops from staging flops
- Transaction flow logic from transaction logic

Certification Challenge

Now it's time to show your skills.

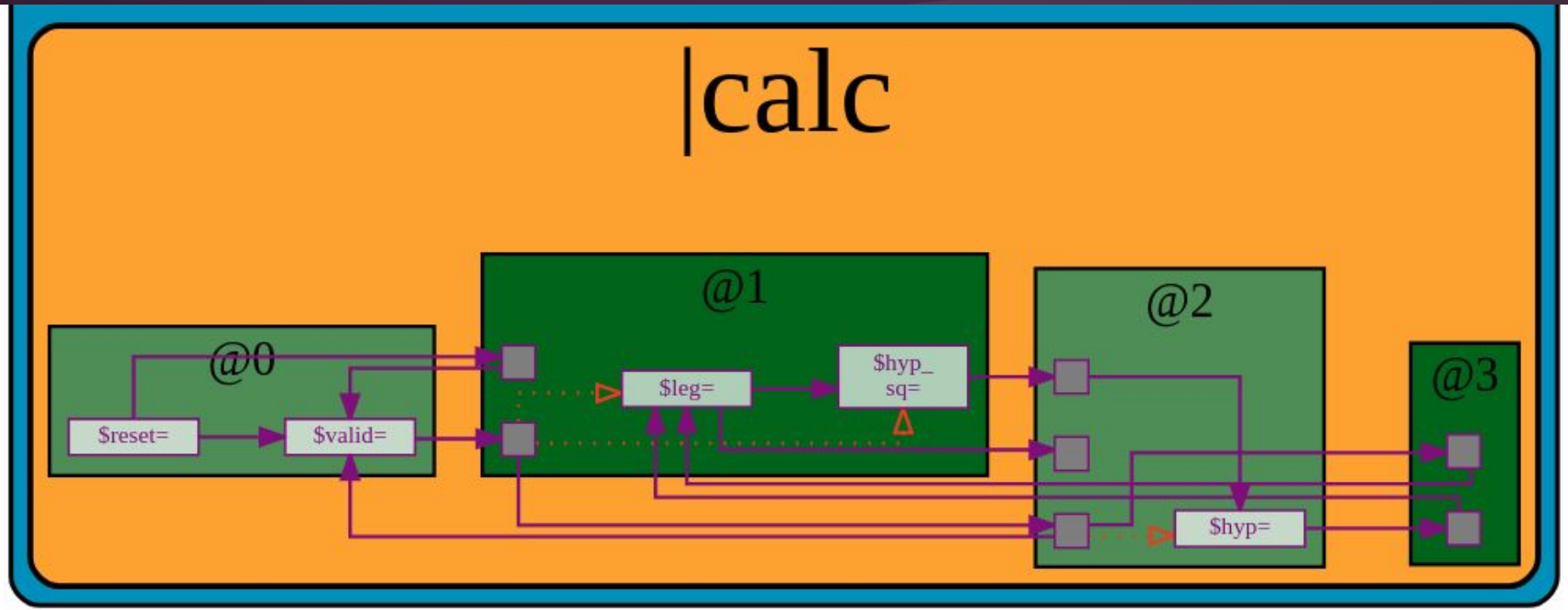
You'll create a circuit to compute the unknown distance.



$$hyp = sqrt(leg^2 * 2)$$

- Find "Webinar Certification Challenge" in "Examples" and load.
- Then be sure your project has been cloned and bookmarked.

Certification Challenge Diagram




```
$leg[15:0] = >>???$valid ? >>???$hyp : 16'd16;  // 16, then prev $hyp.
$hyp_sq[32:0] = ($leg ** 2) * 2;  // Pythagorean thm w/ leg1 == leg2.
$hyp[15:0] = sqrt($hyp_sq);       // "
```

$$hyp = sqrt(leg^2 * 2)$$

Certification Waveform

Calculate w/ 2-cycle latency.

Calculation valid only when **`$valid`**.

Certification Submission

- Find the unknown distance in the log (in decimal vs. hexadecimal in waveform).
- Submit your answer (distance) and course feedback to:

  kunalpghosh@gmail.com.

Live webinar only

Parting Thoughts

Change is a community effort.

Contact me (steve.hoover@redwoodeda.com) about:

- Interest in WARP-V and other open-source development.
- Projects (incl. Google Summer of Code).
- Internship/co-op in Massachusetts (exceptional resumes only).
- Interest in TL-X.org (language standard).
- Pilot program and SandPiper incentives (mention course).
- Questions, thoughts, or just a kind word.

Help me spread the word:

- Show your professors/colleagues.
- Follow me on LinkedIn/Twitter (@RedwoodEDA).
- Share Makerchip on social media (via “Social” menu).