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

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

Diagram of RISC-V architecture showing block components:
- Instruction Memory (IMem) with read (Rd)
- Data Memory (DMem) with read/write (Rd/Wr)
- Decoder (Dec)
- Register File (RF)
- ALU for arithmetic and logic operations
- PC (Program Counter) for branch target
- ld rtn (load return)
- ld data (load data)
- Wr (write)
RISC-V Waterfall Diagram & Hazards

Time ->

ADDI
BGT

PC (P) Ftch (F) Dec (D) Rd (R) Exe (E) Wr (W)

P F D R E W

P F D R E W

mispred

bypass r4

r1 pending

replay for r1

MV r4 <- ...
ADIDI ... <- r4
LD r1 <- ...
ADD

XOR ... <- r1 ...
BLT

<LD r1 rslt>
SW

XOR ... <- r1 ...
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.
RISC-V IP Challenges

Every implementation is constrained differently

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

RTL expresses an implementation.

RTL is not good for IP!
WARP-V: RISC-V CPU Core Generator

“Swiss Cheese” CPU

Params
WordWidth = ...
MemSize = ...

Config
BrPred = ...
ISA = ...

Staging
Fetch = 1
Decode = 2
Execute = 3

ISAs

ISAs

BPs

TL-Verilog

Verilog

Dec    Exe      WB
Dec    Exe      WB
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

<table>
<thead>
<tr>
<th>SystemC + HLS</th>
<th>Chisel, CλaSH, etc.</th>
<th>TL-X (TL-Verilog)</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Integrate w/ C++-based verification models</td>
<td>- Leverage s/w techniques to construct h/w</td>
<td>- H/w modeling (w/ HLS) deserves its own language</td>
</tr>
<tr>
<td>- Synthesize algorithms to gate-level RTL</td>
<td><strong>Driven by:</strong> Academia</td>
<td>- Abstraction as context for details (if details are needed)</td>
</tr>
<tr>
<td>- Tools optimize for constraints</td>
<td><strong>Driven by:</strong> Designers</td>
<td></td>
</tr>
</tbody>
</table>

**Driven by:** EDA Industry
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
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.
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.
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

B) Other logic

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

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

Arithmetic operators operate on vectors as binary numbers.

1. Try:
   (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, ...

![Diagram showing the Fibonacci sequence and logic circuit.](image-url)
Fibonacci Series - Reset

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

$num = \begin{cases} 
\text{reset} & \text{if } reset \\
\text{1} & \text{otherwise} 
\end{cases}

=num[31:0] = \text{reset} ? \text{1} : (\text{>>1}$num + \text{>>2}$num);
Lab: Counter

Lab:

1. Design a free-running counter:

   ![Circuit Diagram]

   $num[31:0] = \text{reset} \ ? 1 : (\gg 1\text{num} + \gg 2\text{num});$

2. Compile and explore.

Reference Example: Fibonacci Sequence

$(1, 1, 2, 3, 5, 8, ...)$
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

Stage 1: Roll
Stage 2: Twist
Stage 3: Salt
Stage 4: Bake
A Simple Pipeline

Pythagoras's Theorem circuit:

\[
\begin{align*}
\text{aa}_{31:0} &= \text{aa} \times \text{aa}; \\
\text{bb}_{31:0} &= \text{bb} \times \text{bb}; \\
\text{cc}_{31:0} &= \text{aa}_{31:0} + \text{bb}_{31:0}; \\
\text{cc}_{31:0} &= \sqrt{\text{cc}_{31:0}};
\end{align*}
\]

Too much for one cycle. Distribute over three cycles.
Timing-Abstraction

RTL:

\[ a \xrightarrow{^2} + \xrightarrow{sqrt} c \]
\[ b \xrightarrow{^2} + \xrightarrow{sqrt} c \]

Timing-abstract:

\[ \overset{|\text{calc}}{a} \xrightarrow{^2} + \xrightarrow{sqrt} c \]
\[ \overset{|\text{calc}}{b} \xrightarrow{^2} + \xrightarrow{sqrt} c \]

Stage: 1 2 3

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);

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);

~3.5x
Retiming -- Easy and Safe

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

$num[31:0] = \text{if } \text{reset} \text{ then } 1 \text{ else } (\gg 1\num + \gg 2\num);$
```

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

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

3-Stage
fetchStage = 1
decStage = 2
exeStage = 3

1-Stage
fetchStage = 1
decStage = 1
exeStage = 1

In WARP-V:

Not Practical with RTL!
WARP-V Parameterized Staging

![Diagram of WARP-V Parameterized Staging]

- IMem Rd
- Dec
- RF Rd
- ALU
- RF Wr
- DMem Rd/Wr
- Pend. replay PC
- Br. target
- Reg. byp.

Id data

Id rtn
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

Validity provides:

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

```
| 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
  $cc[31:0] = sqrt($cc_sq);
```
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.
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)

For reference:
\begin{verbatim}
|calc
@0
$aa_sq[31:0] = $aa * $aa;
@1
$bb_sq[31:0] = $bb * $bb;
\end{verbatim}
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
$op_a[63:0] =
($op_a_src == IMM) ? $imm_data : 
($op_a_src == BYP) ? $rslt : 
($op_a_src == REG) ? $reg_data : 
($op_a_src == MEM) ? $mem_data : 64'b0;
WARP-V Operand Mux Retimed

\[
\text{\$op_a[63:0] = } \\
\text{\$op_a \_src == IMM) ? \$imm\_data : } \\
\text{\$op_a \_src == BYP) ? >>1\$rslt : } \\
\text{\$op_a \_src == REG) ? >>2\$reg\_data : } \\
\text{\$op_a \_src == MEM) ? /top\_mem>>5\$mem\_data : } \\
\text{64'b0;}
\]
Register Bypass

No bypass (ISA spec):

\[ \text{$reg1\_value[M4\_WORD\_RANGE]} = \text{/cpu/regs[$reg]>>1$value}; \]

Delay RF write.

Broken implementation!
No bypass (ISA spec):

$\text{reg1\_value}[\text{M4\_WORD\_RANGE}] = \text{/cpu/regs}[\text{reg}] >> 1 \text{value};$

Two bypass stages:

$\text{reg1\_value}[\text{M4\_WORD\_RANGE}] =$

$(>>1 \text{valid} && (>>1 \text{dest\_reg} == \text{reg1})) \ ? \ >>1 \text{rslt} :$

$(>>2 \text{valid} && (>>2 \text{dest\_reg} == \text{reg1})) \ ? \ >>2 \text{rslt} :$

$/\text{regs}[\text{reg1}] >> 3 \text{value};$
Time-Division Multiplexing Example

Producer

packets

flits

Consumer

packets

Time ->
Time-Division Multiplexing Example

The diagram illustrates a time-division multiplexing example. The packet input ($packet\_in$) is divided into flits ($flit$). Each flit is transmitted through the system and eventually passed to the packet output ($packet\_out$). The numbers 0 to 4 indicate the time slots in which the flits are transmitted.
Time-Division Multiplexing Example
1. Load “Examples”/“Webinar”/“TDM Lab”.
2. Fill in the TL-Verilog for $\text{flit}[3:0] = ...$
3. $\text{packet_out}[15:0] = ...$
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
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

```plaintext
|default
/yy[Y_SIZE-1:0]
/xx[X_SIZE-1:0]
@1

  // Sum left + me + right.
$\text{row\_cnt}[1:0]\ = \ldots;\n
  // Sum three $\text{row\_cnt}'s: \above + \mine + \below.
$\text{cnt}[3:0]\ = \ldots;\n
  // Init state.
$\text{init\_alive}[0:0]\ = \text{*RW\_rand\_vect}[\ldots];\n
$\text{alive} = \text{$reset}\ ? \text{$init\_alive} :\n  \text{\ldots} @1\text{$alive}\ ? \text{($cnt >= 3 && $cnt <= 4) :}\n  \text{($cnt == 3);}\n```

(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:
\[
$\text{core2}_{\text{sig}}[1:0] = \\
/\text{core}[2]|/\text{my}_{\text{pipe}}/\text{trans}>>1$\text{my}_{\text{sig}}[3:2];
\]

Accumulate:
\[
$\text{any}_{\text{valid}} = | /\text{slice}[\ast]$\text{valid};
\]

⇒ No interface parameterization for TL-Verilog IP!
Decode: Extracting src reg fields

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

Macro preprocessing with M4 provides a workable solution in the meantime:
- Parameterization (incl. staging)
- Library inclusion
- Modularity & reuse
- Configuration (component selection)
- Code construction
“Swiss-Cheese” Design

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

```c
// 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
```

```c
@decode
/src[*]
$reg[5:0] = (#src == 1)
? $instr[19:15]
: $instr[24:20];
@endcode

@execute
$add_rslt[31:0] =
/src[1]$reg_value +
/src[2]$reg_value;
```
Parameterized Register Bypass

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

```c
$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:

```verilog
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.:

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

|pipe

data_in[31:0]  $data[31:0]  data_out[31:0]

valid

clk

0  1  2  3  4  5
File Structure

module my_design (  
   input clk,  
   input valid,  
   input [31:0] data_in,  
   output [31:0] data_out  
);
endmodule
Verilog/SystemVerilog Example

module my_design (input clk, input valid, input [31:0] data_in, output [31:0] data_out);
pipe @0
$valid = *valid;
?$valid
$valid = *data_in;
@5
*data_out = $data;
endmodule
module my_design (  
    input clk,  
    input valid,  
    input [31:0] data_in,  
    output [31:0] data_out  
);

@0  
! valid = *valid;
?valid
! $data[31:0] = *data_in;
// Logic
// There is none.
@5  
! *data_out = $data;
endmodule
Makerchip File Structure

[Show in Makerchip]
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
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
Quiescing:
  ● Stop issuing transactions
  ● In-flight transactions => 0 ($valid => 0)
  ● Save high-level state only.
Can capture periodic checkpoints and reproduce bugs in simulation.
<table>
<thead>
<tr>
<th>State</th>
<th>Staging</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Example</strong></td>
<td>$RegValue[63:0] $instr_immediate[6:0]</td>
</tr>
<tr>
<td><strong>Nature</strong></td>
<td>Persistent</td>
</tr>
<tr>
<td><strong>Value under ?valid == 0</strong></td>
<td>Retained</td>
</tr>
<tr>
<td><strong>Reset</strong></td>
<td>Fixed value</td>
</tr>
<tr>
<td><strong>Quiescent Value</strong></td>
<td>Retained</td>
</tr>
</tbody>
</table>

- Valuable distinction for reset and debug.
State - Distance Accumulator

\[ \text{TotDist} = \sqrt{a^2 + b^2 + c^2} \]

\[ \text{TotDist} = \sqrt{\text{TotDist}^2 + \text{TotDist}^2 + \text{TotDist}^2} \]

\[ \text{TotDist} = \sqrt{\text{TotDist}^2 + \text{TotDist}^2 + \text{TotDist}^2} \]

\[ \text{TotDist} = \sqrt{\text{TotDist}^2 + \text{TotDist}^2 + \text{TotDist}^2} \]
State - Distance Accumulator

$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);

$TotDist[31:0] <= reset ? 0 : TotDist + cc;

Shorthand for: $<<1$TotDist[31:0] =
$\text{TotDist}$ Waveform
// ======
// 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, ...

\[
\text{reset}
\]

\[
\text{num}
\]

\[
\text{reset}
\]

\[
\text{num}
\]

\[
\text{reset}
\]

\[
\text{Num}
\]

\[
\text{reset}
\]

\[
\text{Num}
\]

\[
\text{num}[31:0] = \text{reset} \ ? \ 1 : (\gg \text{num} + \gg \text{num});
\]

\[
\text{Num}[31:0] \leq \text{reset} \ ? \ 1 : (\text{Num} + \gg \text{Num});
\]
Lab: Counter as State

Lab:

1. Design a free-running counter:

   ![Circuit Diagram]

   $\text{Num}[31:0] \leftarrow \text{reset} \oplus 1 : (\text{Num} + \gg 1\text{Num})$

2. Compile and explore.

Reference Example: Fibonacci Sequence

$(1, 1, 2, 3, 5, 8, ...)$

(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
- Flow constructed from pre-verified library components. (~100 lines)
- Transaction logic added into this context.
Error rate too high. Require parity protection on FIFO and Ring.

$parity = ^ {\{data, \ dest, \ ...\}};

$parity\_error = \ $parity \neq ^ {\{data, \ \ dest\}};

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

[Demo Flow Tutorial in Makerchip]
Transaction Flow

Phase-based B.P. Pipeline

Transaction Flow

FIFO

Free-Flow Pipeline

Stall Pipeline

Phase-based B.P. Pipeline

ARB

Free-Flow Pipeline

comb

comb

comb

comb

comb

comb

comb
Transaction Flow: Retiming

- Stall Pipeline
- FIFO
- Phase-based B.P. Pipeline
- Free-Flow Pipeline
- QUEUE
- Rings

Comb boxes and arrows indicate flow and connections between the different pipeline stages.
Transaction Flow: Retiming
Transaction Flow: Mechanism -- $\textsf{ANY}$

- Example: Back-pressed pipeline

Bold wires carry transactions, and are referenced as $\textsf{ANY}$.

$\textsf{ANY} = \gg 1\text{recirc} \ ? \ \gg 1\text{ANY} : /\text{top} |\text{stage1} \gg 1\text{ANY}$;

- $\textsf{ANY}$ is a wildcard rule. If a pipesignal is needed that is not available, it can be produced by a $\textsf{ANY}$ expression.
Back-Pressured Pipeline

(makerchip.com/sandbox/0/0Elh3R)
“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
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.

\[ \text{hyp} = \sqrt{\text{leg}^2 \times 2} \]

- Find “Webinar Certification Challenge” in “Examples” and load.
- Then be sure your project has been cloned and bookmarked.
$\text{leg}[15:0] = \text{valid} \rightarrow \text{hyp} : 16'\text{d}16$; // 16, then prev $\text{hyp}$.
$\text{hyp}_\text{sq}[32:0] = (\text{leg} \times \text{leg}) \times 2$; // Pythagorean thm w/ leg1 == leg2.
$\text{hyp}[15:0] = \sqrt{\text{hyp}_\text{sq}}$; // "

\[
\text{hyp} = \sqrt{\text{leg}^2 \times 2}
\]
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.
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).