

Building a RISC-V Core

RISC-V Based MYTH Workshop

MYTH - Microprocessor for You in Thirty Hours



Steve Hoover

Founder, Redwood EDA

July 31, 2020



Day 3-5 Agenda

Day 3: Digital logic with TL-Verilog in Makerchip IDE

Day 4: Coding a RISC-V CPU subset

Day 5: Pipelining and completing your CPU



Agenda

Day 3: Digital logic with TL-Verilog in Makerchip IDE

- Logic gates
- Makerchip platform
- Combinational logic
- Sequential logic
- Pipelined logic
- State

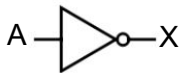








Logistics

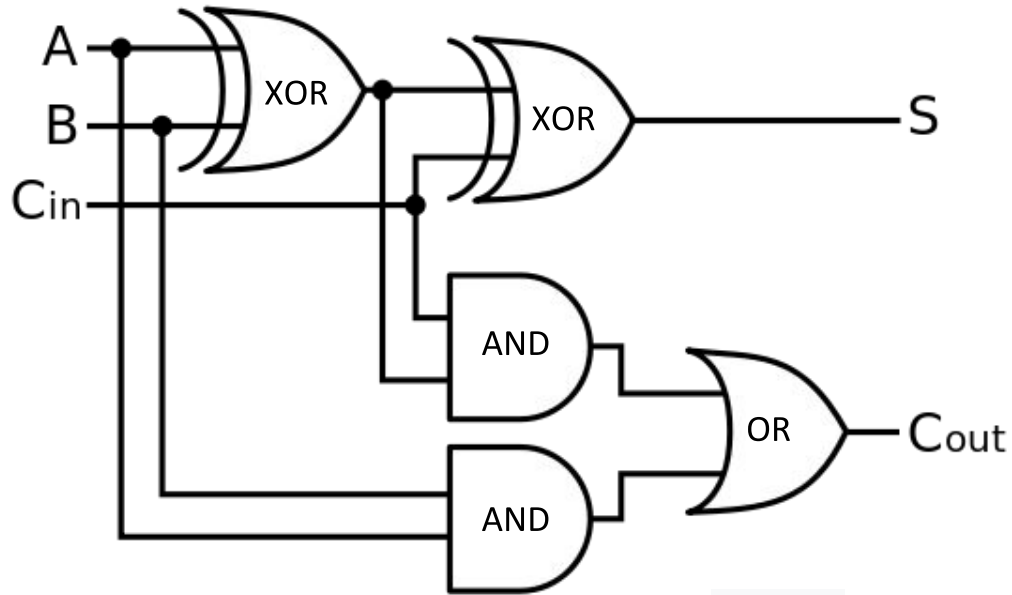
Live updates, lab help, links, etc.:

https://github.com/stevehoover/RISC-V_MYTH_Workshop

Logic Gates

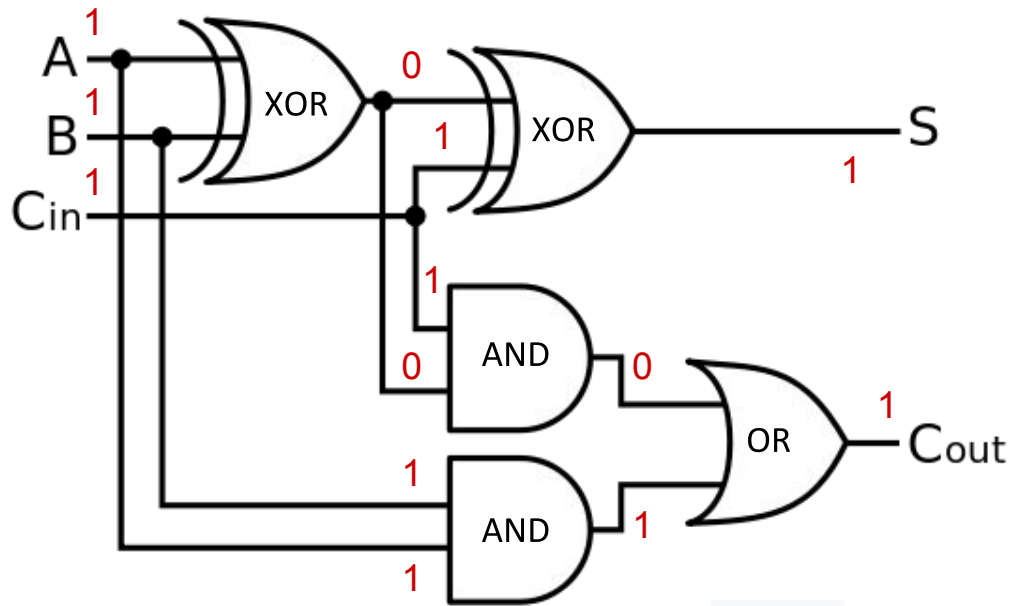
Name	NOT	AND	OR	XOR	NAND	NOR	XNOR																																																																																																
Symbol																																																																																																							
Truth Table	<table><tr><th>A</th><th>X</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	X	0	1	1	0	<table><tr><th>A</th><th>B</th><th>X</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	X	0	0	0	0	1	0	1	0	0	1	1	1	<table><tr><th>A</th><th>B</th><th>X</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	X	0	0	0	0	1	1	1	0	1	1	1	1	<table><tr><th>A</th><th>B</th><th>X</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	X	0	0	0	0	1	1	1	0	1	1	1	0	<table><tr><th>A</th><th>B</th><th>X</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	X	0	0	1	0	1	1	1	0	1	1	1	0	<table><tr><th>A</th><th>B</th><th>X</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	X	0	0	1	0	1	0	1	0	0	1	1	0	<table><tr><th>A</th><th>B</th><th>X</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	X	0	0	1	0	1	0	1	0	0	1	1	1
A	X																																																																																																						
0	1																																																																																																						
1	0																																																																																																						
A	B	X																																																																																																					
0	0	0																																																																																																					
0	1	0																																																																																																					
1	0	0																																																																																																					
1	1	1																																																																																																					
A	B	X																																																																																																					
0	0	0																																																																																																					
0	1	1																																																																																																					
1	0	1																																																																																																					
1	1	1																																																																																																					
A	B	X																																																																																																					
0	0	0																																																																																																					
0	1	1																																																																																																					
1	0	1																																																																																																					
1	1	0																																																																																																					
A	B	X																																																																																																					
0	0	1																																																																																																					
0	1	1																																																																																																					
1	0	1																																																																																																					
1	1	0																																																																																																					
A	B	X																																																																																																					
0	0	1																																																																																																					
0	1	0																																																																																																					
1	0	0																																																																																																					
1	1	0																																																																																																					
A	B	X																																																																																																					
0	0	1																																																																																																					
0	1	0																																																																																																					
1	0	0																																																																																																					
1	1	1																																																																																																					

Combinational Circuit



[CC BY-SA 3.0](#)
[en:User:Cburnett](#)

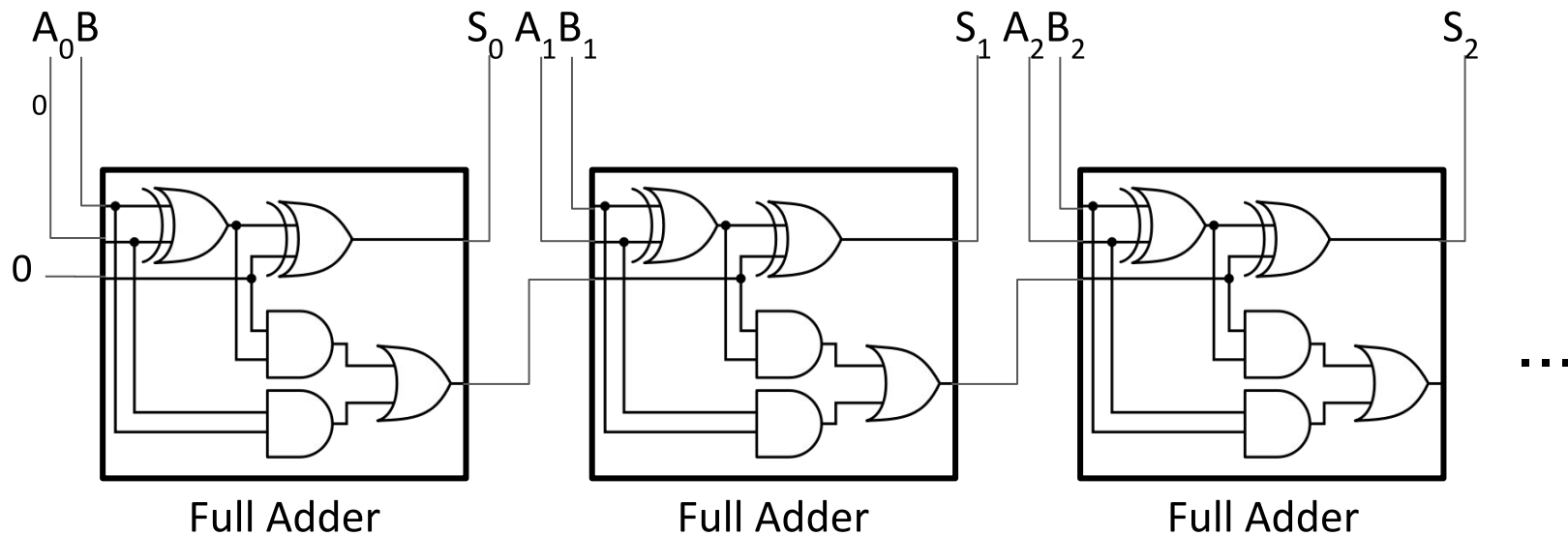
Combinational Circuit



[CC BY-SA 3.0](#)
[en:User:Cburnett](#)

Adder








$$S = A + B$$



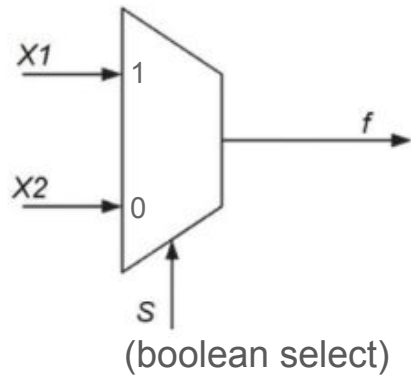
Adder



Boolean Operators

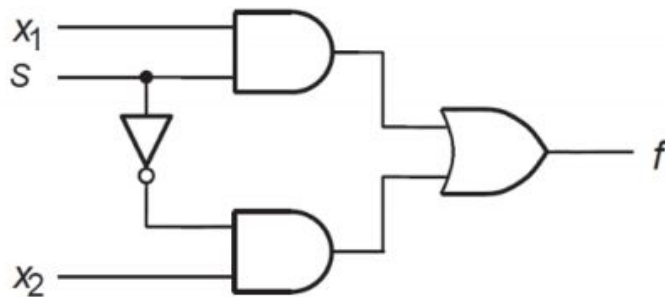
Op	Bool Arith	Bool Calc	Verilog	Gate
NOT	\overline{A}	$\neg A$	$\sim A$ (or $!A$)	
AND	$A \bullet B$	$A \wedge B$	$A \& B$ (or $\&\&$)	
OR	$A + B$	$A \vee B$	$A B$ (or $ $)	
XOR	$A \oplus B$	$A \oplus B$	$A \wedge B$	
NAND	$\overline{A \bullet B}$	$\neg (A \wedge B)$	$! (A \& B)$	
NOR	$\overline{A + B}$	$\neg (A \vee B)$	$! (A B)$	
XNOR	$\overline{A \oplus B}$	$\neg (A \oplus B)$	$! (A \wedge B)$	

Multiplexer (MUX)

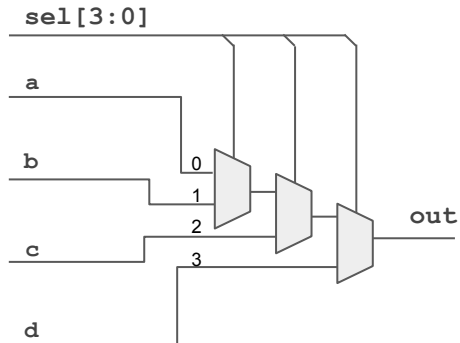
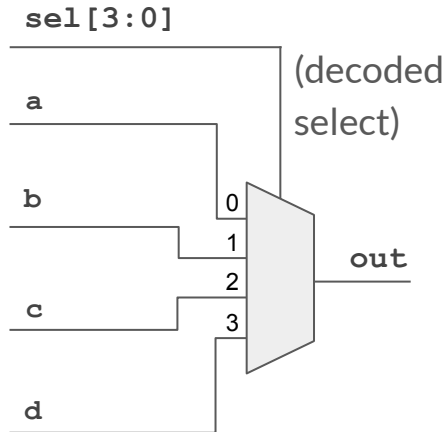


Verilog:

```
assign f = s ? X1 : X2;
```



Chaining Ternary Operator



Verilog:

```
assign f =  
    sel[0]  
    ? a  
    : (sel[1]  
        ? b  
        : (sel[2]  
            ? c  
            : d  
        )  
    );
```

Equivalently:

```
assign f =  
    sel[0]  
    ? a :  
    sel[1]  
    ? b :  
    sel[2]  
    ? c :  
    //default  
    d;
```

(So, highest
priority first.)

Makerchip

makerchip PROJECT TUTORIALS HELP saved 4 minutes ago

EDITOR NAV-TLV LOG

```
@1
Saa_sq[7:0] = $aa[3:0] ** 2;
Sbb_sq[7:0] = $bb[3:0] ** 2;
@2
Scc_sq[8:0] = Saa_sq + Sbb_sq;
@3
Scc[4:0] = sqrt(Scc_sq);
```

Last updated 10 minutes ago

TUTORIAL-VALID

|calc

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

Last updated 10 minutes ago

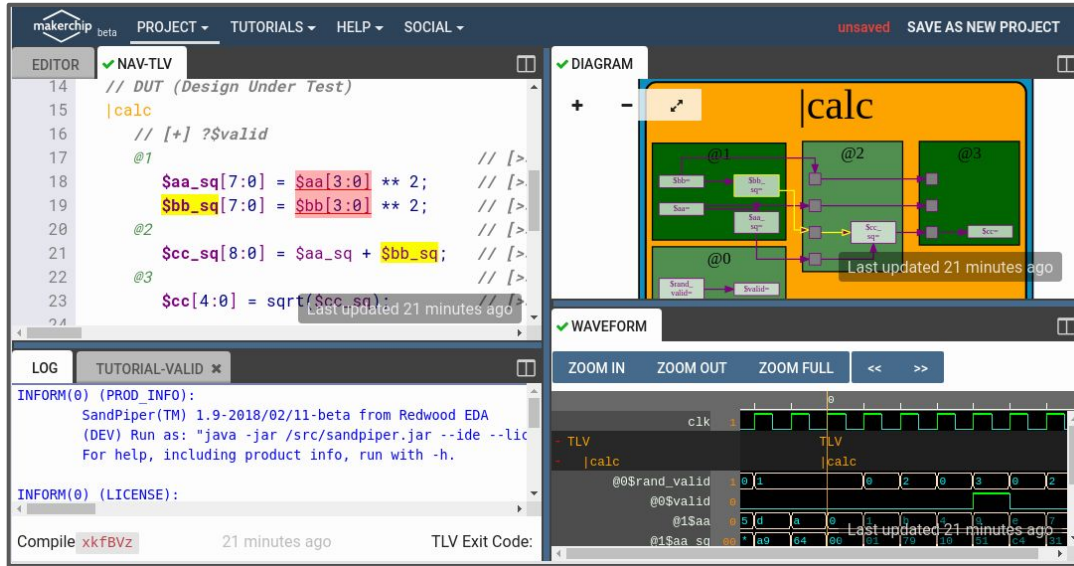
WAVEFORM

ZOOM IN ZOOM OUT ZOOM FULL << >>

Signal	Value
clk	1
TLV	1
calc	1

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”.

1. Reproduce this screenshot:
2. Open “Tutorials” “Validity Tutorial”.
3. In tutorial, click

Load Pythagorean Example

4. Split panes  and move tabs.
5. Zoom/pan in Diagram w/ mouse wheel and drag.
6. Zoom Waveform w/ “Zoom In” button.
7. Click `$bb_sq` to highlight.

Lab: Combinational Logic

A) Inverter

1. Open “Examples” (under “Tutorials”).
2. Load “Default Template”.
3. Make an inverter.

On line 16, in place of:

```
//...
```

type:

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

(Preserve 3-space indentation, no tabs)

4. Compile (“E” menu) & Explore

Note:

There was no need to declare `$out` and `$in1` (unlike Verilog).

There was no need to assign `$in1`. Random stimulus is provided, and a warning is produced.

B) Other logic

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.

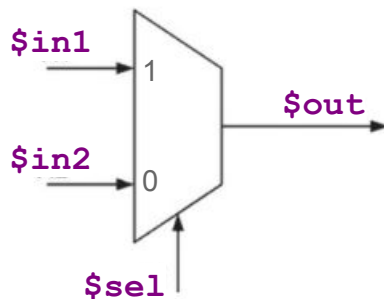
1. Try:

```
$out[4:0] = $in1[3:0] + $in2[3:0];
```

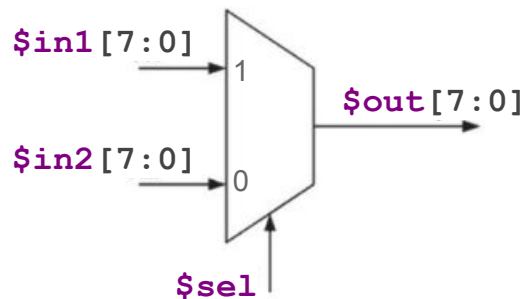
2. View Waveform.

Lab: Mux

`$out = $sel ? $in1 : $in2`
creates a multiplexer.



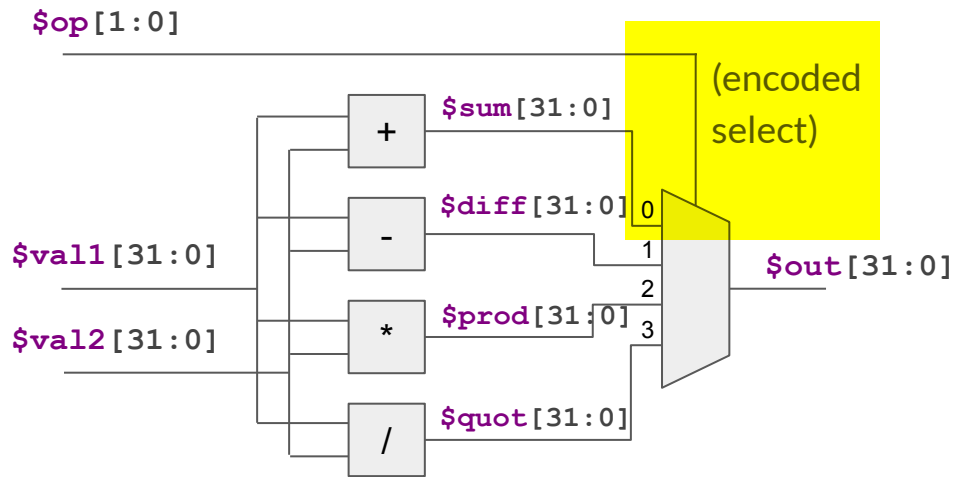
Modify this multiplexer to operate on vectors.



Note that bit ranges can generally be assumed on the left-hand side, but with no assignments to these signals, they must be explicit.

Lab: Combinational Calculator

This circuit implements a calculator that can perform $+$, $-$, $*$, $/$ on two input values.



1. Implement this.

2. Use:

```
$val1[31:0] = $rand1[3:0];  
$val2[31:0] = $rand2[3:0];
```

for inputs to keep values small.

3. We'll return to this, so "Save as new project", bookmark, and open a new Makerchip IDE in a new tab.

Sequential Logic

Sequential logic is sequenced by a clock signal.



A D-flip-flop transitions next state to current state on a rising clock edge.

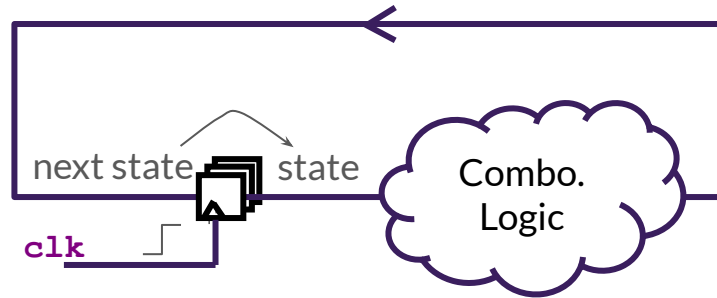


The circuit is constructed to enter a known state in response to a reset signal.



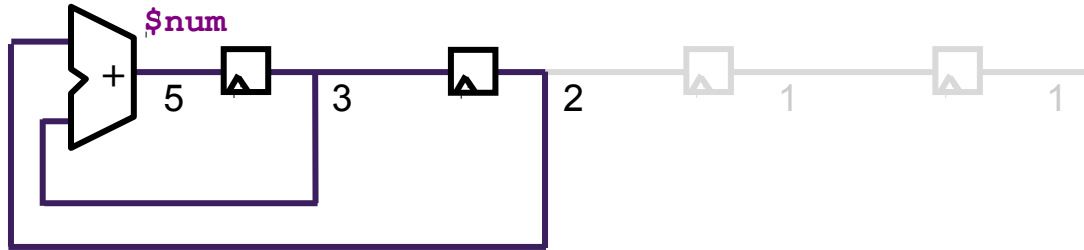
Sequential Logic

The whole circuit can be viewed as a big state machine.



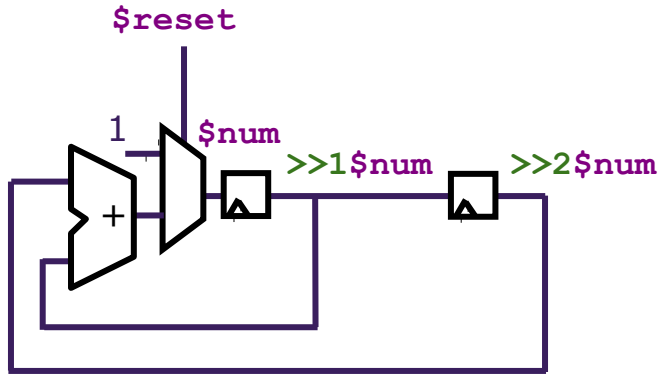
Sequential Logic - Fibonacci Series

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



Fibonacci Series - Reset

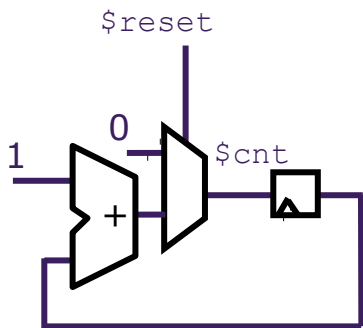
Next value is sum of previous two: 1, 1, 2, 3, 5, 8, 13, ...



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

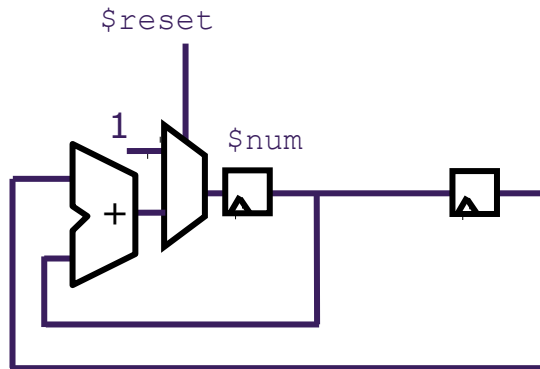
Lab: Counter

1. Design a free-running counter:



2. Include this code in your saved calculator sandbox for later (and confirm that it auto-saves).

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)

Values in Verilog

16 ' h F0

16-bit hexadecimal value

' 0: All 0s (width based on context).

' X: All DONT-CARE bits.

16 ' d5: 16-bit decimal 5.

5 ' b00XX1: 5-bit value with DONT-CARE bits.

1: 32-bit (signed) 1.

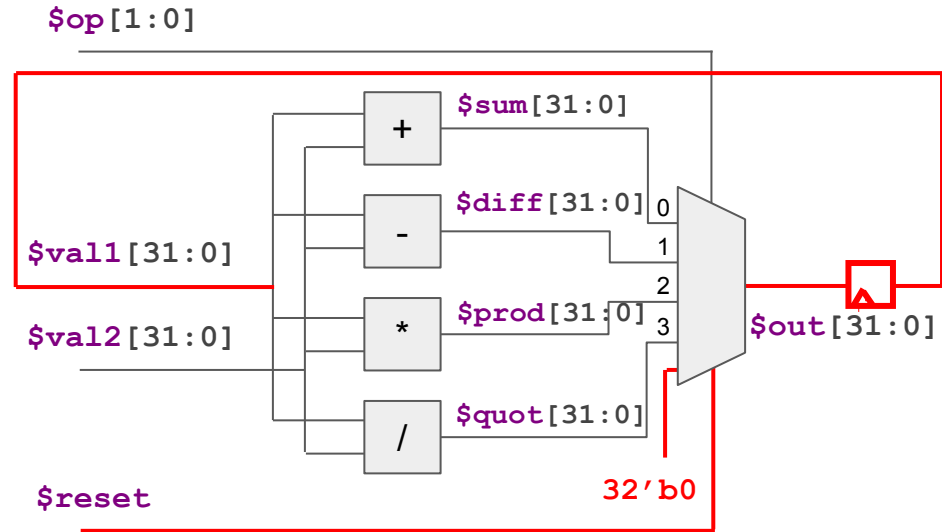
Our simulator configuration:

- will zero-extend or truncate when widths are mismatched (without warning)
- uses 2-state simulation (no X's)

Lab: Sequential Calculator

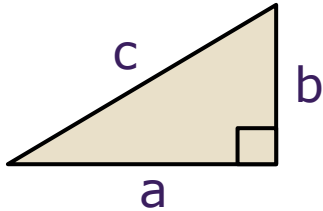
A real calculator remembers the last result, and uses it for the next calculation.

1. Return to the calculator.
2. Update the calculator to perform a new calculation each cycle where
 $\$val1[31:0]$ = the result of the previous calculation.
3. Reset $\$out$ to zero.
4. Copy code and save outside of Makerchip (just to be safe).

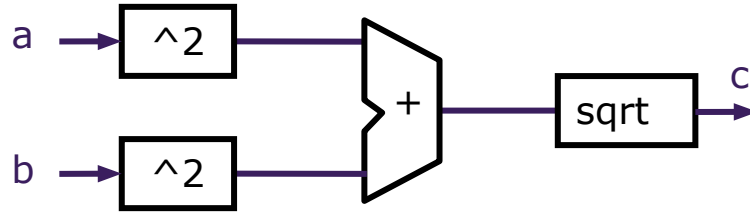


A Simple Pipeline

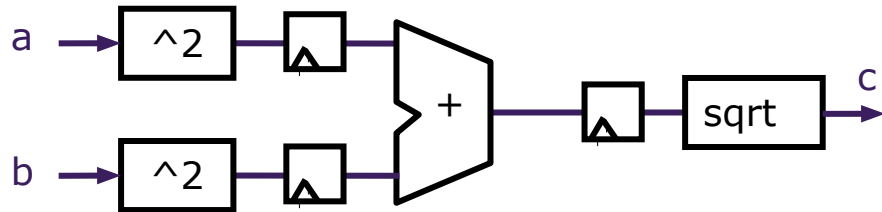
Let's compute Pythagoras's Theorem in hardware.



$$c = \text{sqrt}(a^2 + b^2)$$

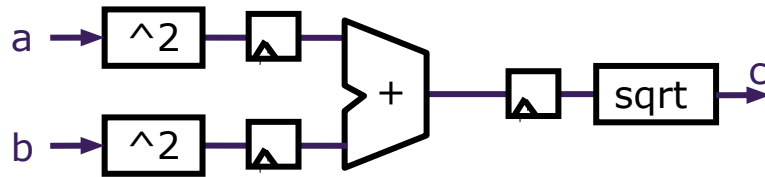


We distribute the calculation over three cycles.

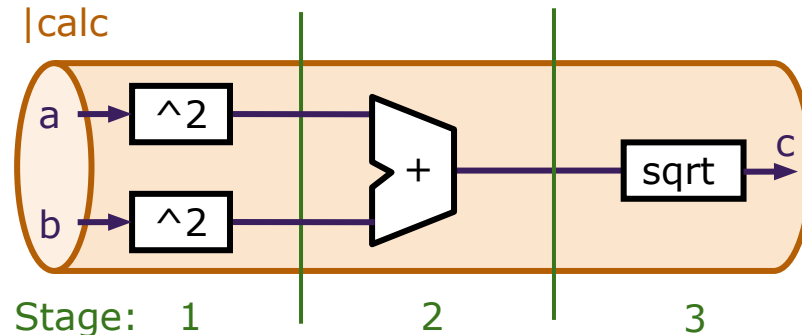


A Simple Pipeline - Timing-Abstract

RTL:

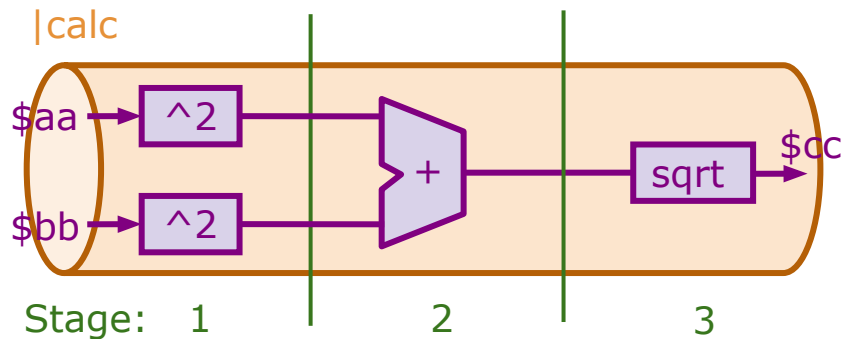


Timing-abstract:



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

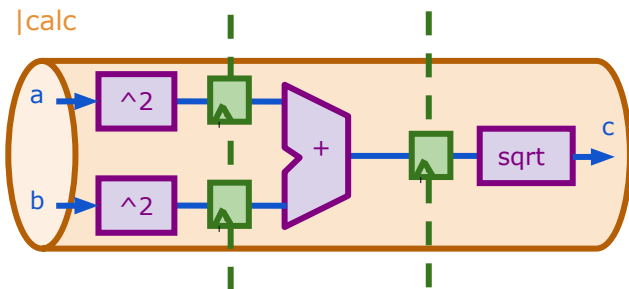
A Simple Pipeline - TL-Verilog



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

~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);
```

```
// 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);
```

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;

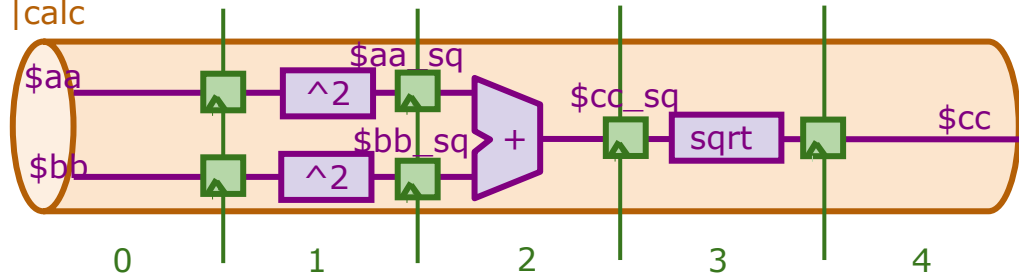
@2

\$cc_sq[31:0] = \$aa_sq + \$bb_sq;

@4

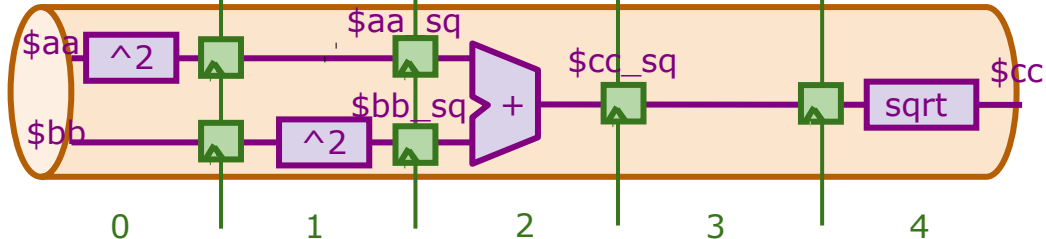
\$cc[31:0] = sqrt(\$cc_sq);

|calc



==

|calc



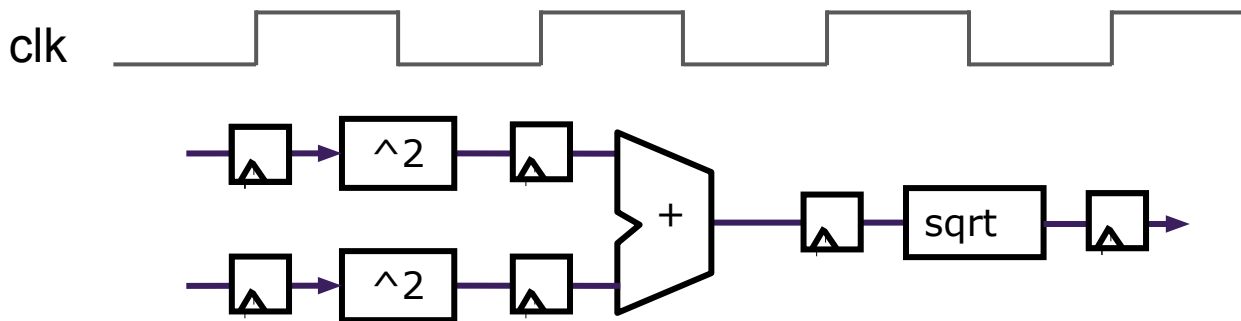
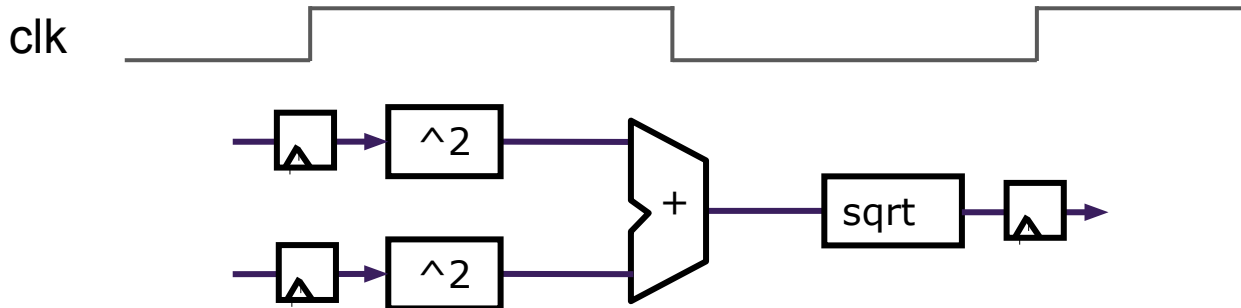
Staging is a physical attribute. No impact to behavior.

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!

High Frequency



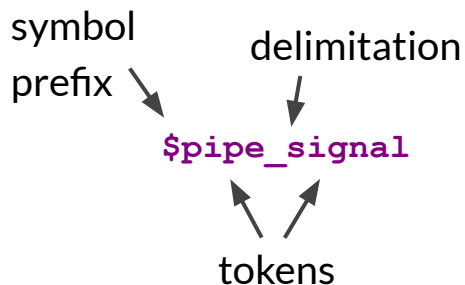


Makerchip - a level deeper

(Exploration of pipelined logic within Makerchip.)

Identifiers and Types

Type of an identifier determined by symbol prefix and case/delimitation style. E.g.:



First token must start with two alpha chars. These determine delimitation style

- `$lower_case`: pipe signal
- `$CamelCase`: state signal (technically, this is "Pascal case")
- `$UPPER_CASE`: keyword signal

Numbers end tokens (after alphas)

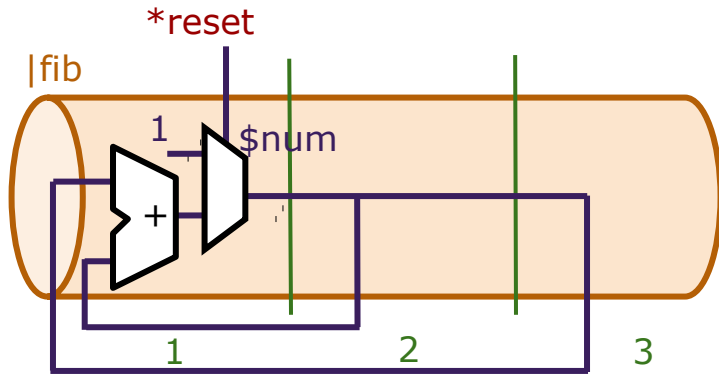
- `$base64_value`: good
- `$bad_name_5`: bad

Numeric identifiers

- `>>1`: ahead by 1

Fibonacci Series in a Pipeline

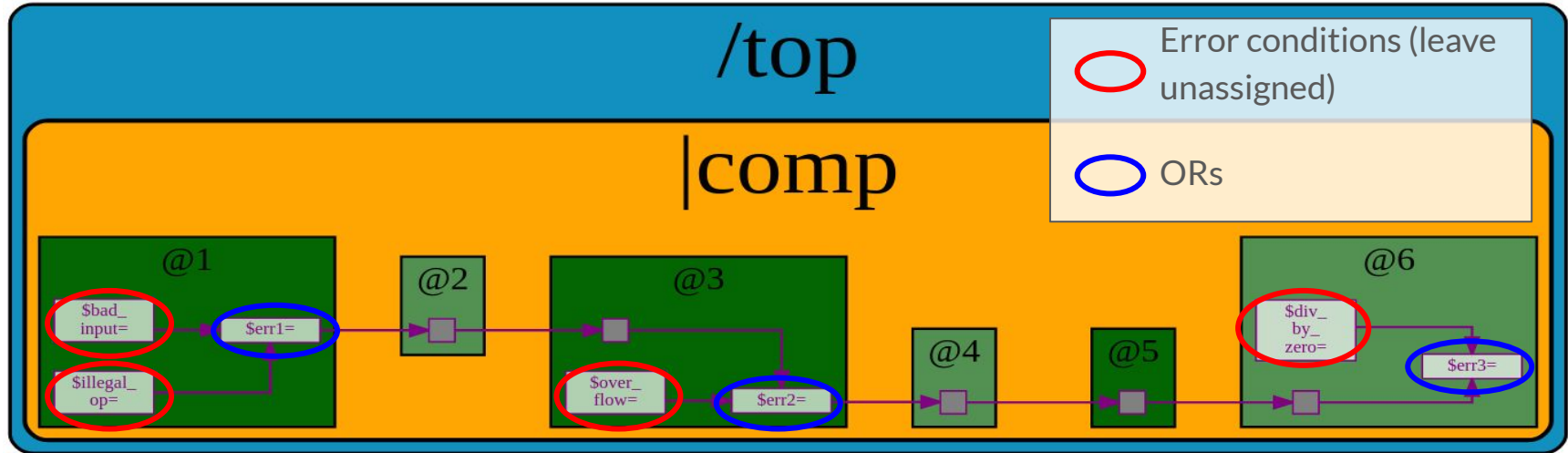
Next value is sum of previous two: 1, 1, 2, 3, 5, 8, 13, ...



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

Lab: Pipeline

See if you can produce this:



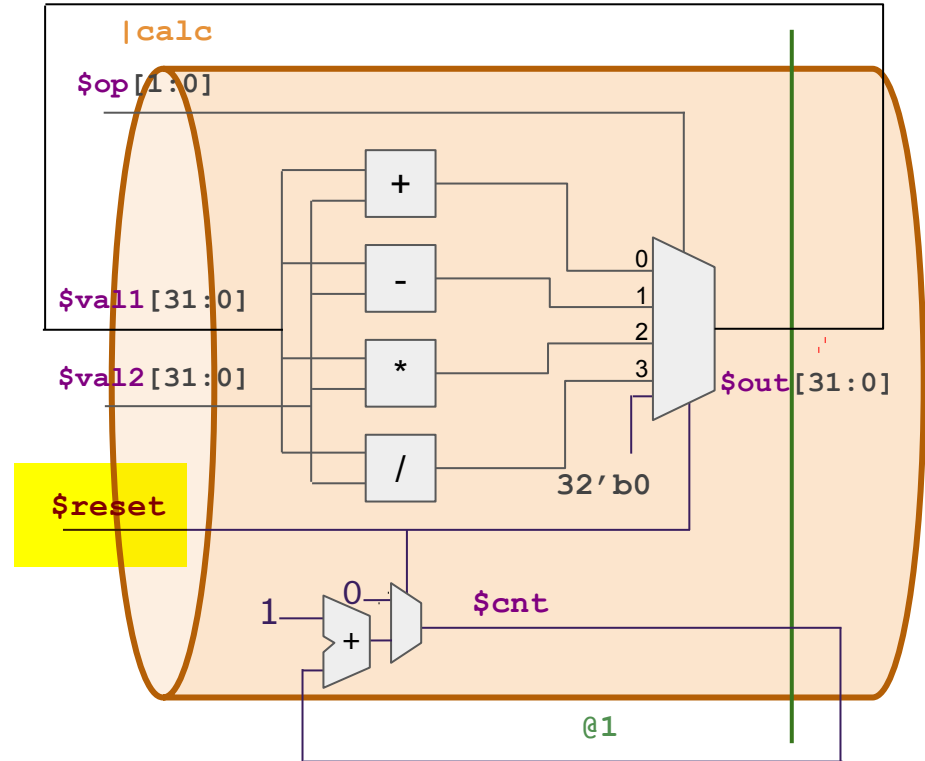
which ORs together (|) various error conditions that can occur within a computation pipeline.

Open in Makerchip

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

Lab: Counter and Calculator in Pipeline

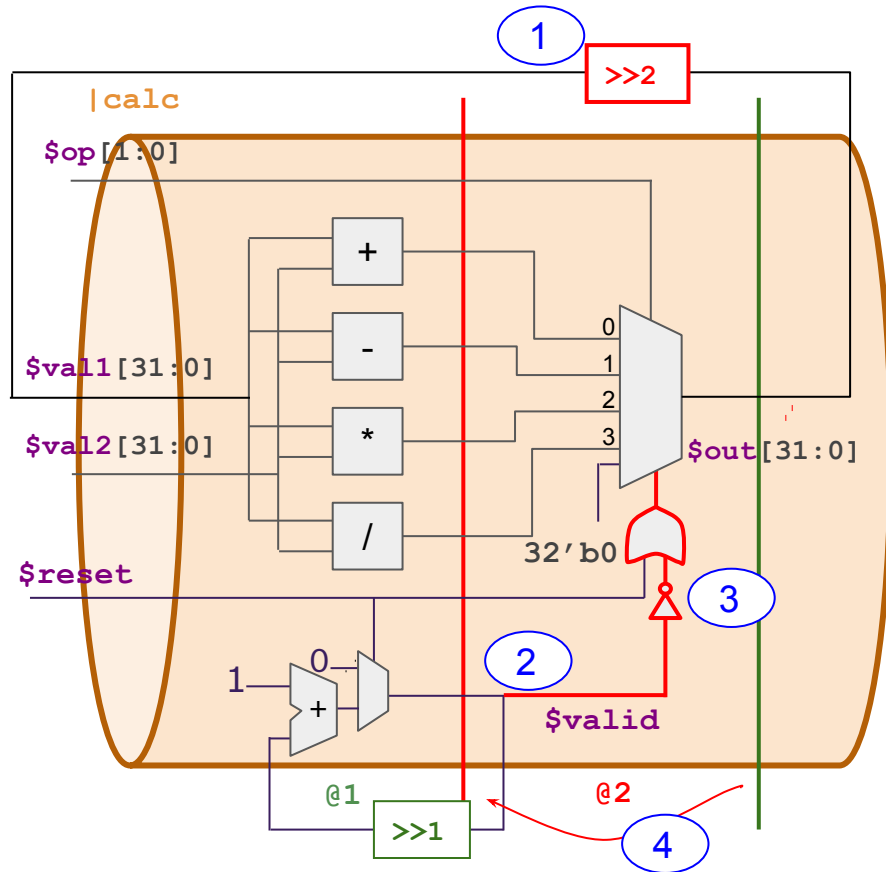
1. Put calculator and counter in stage @1 of a |calc pipeline.
2. Check log, diagram, and waveform.
3. Confirm save.



Lab: 2-Cycle Calculator

At high frequency, we might need to calculate every other cycle.

1. Change alignment of `$out` (to calculate every other cycle).
2. Change counter to single-bit (to indicate every other cycle).
3. Connect `$valid` (to clear alternate outputs).
4. Retime mux to @2 (to ease timing; no functional change).
5. Verify behavior in waveform.
6. Save.

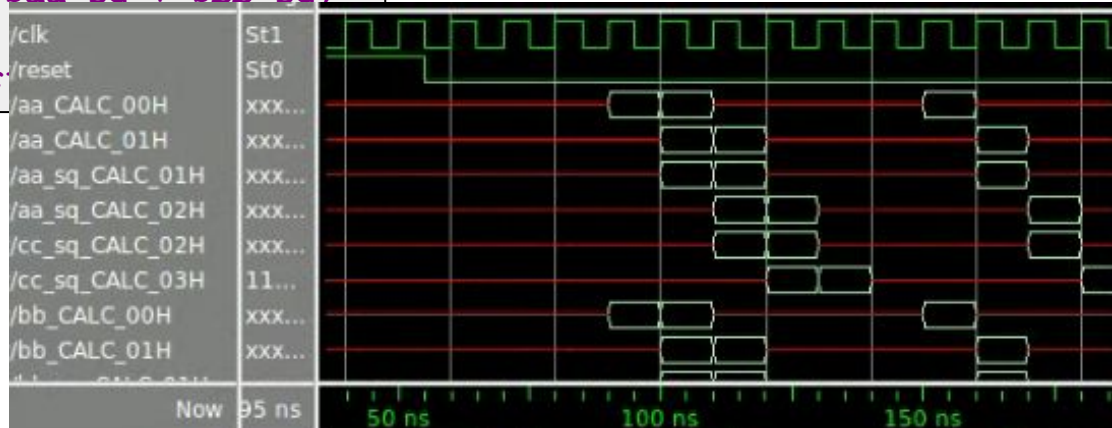


Validity

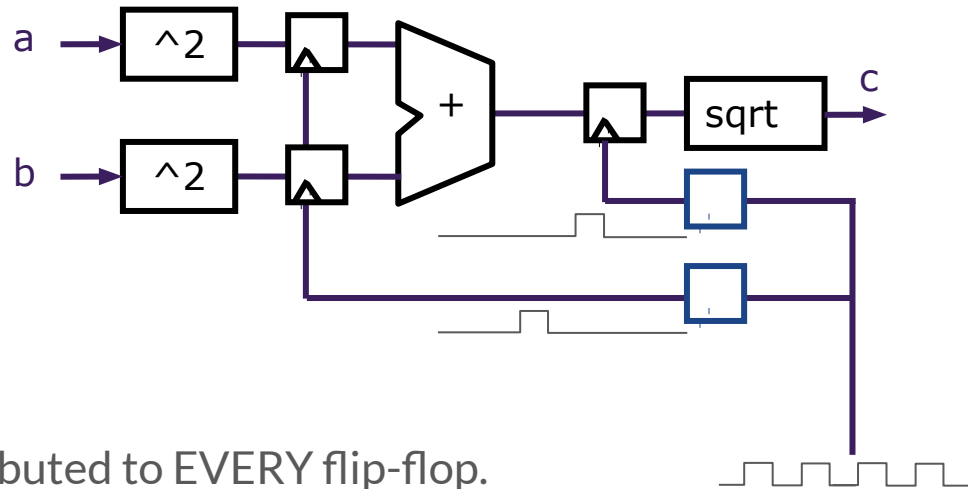
```
|calc
@1
    $valid = ...;
    ?$valid
@1
    $aa_sq[31:0] = $aa * $aa;
    $bb_sq[31:0] = $bb * $bb;
@2
    $cc_sq[31:0] = $aa_sq + $bb_sq;
@3
    $cc[31:0] = sqr
```

Validity provides:

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

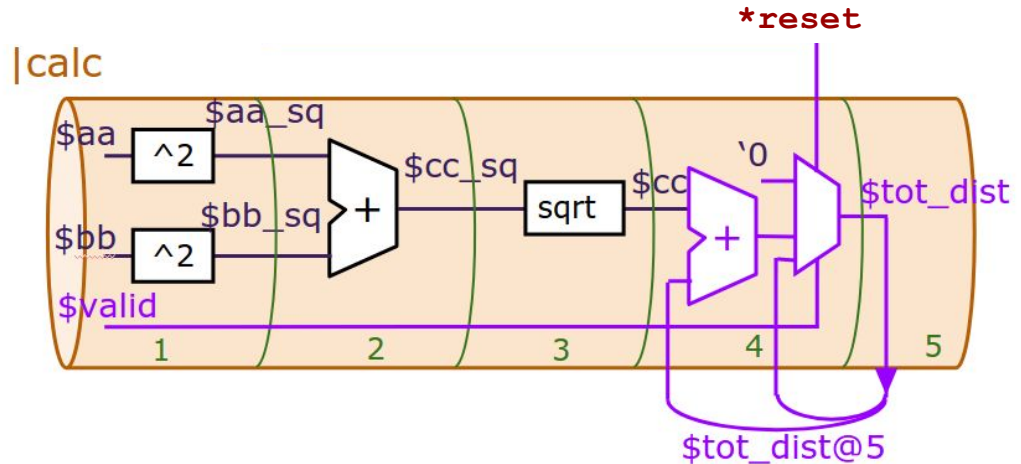
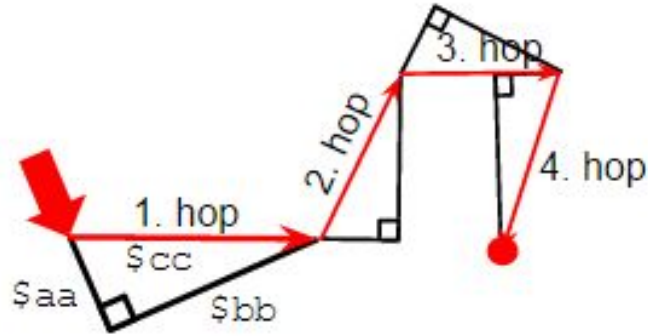


Clock Gating



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

Total Distance (Makerchip walkthrough)



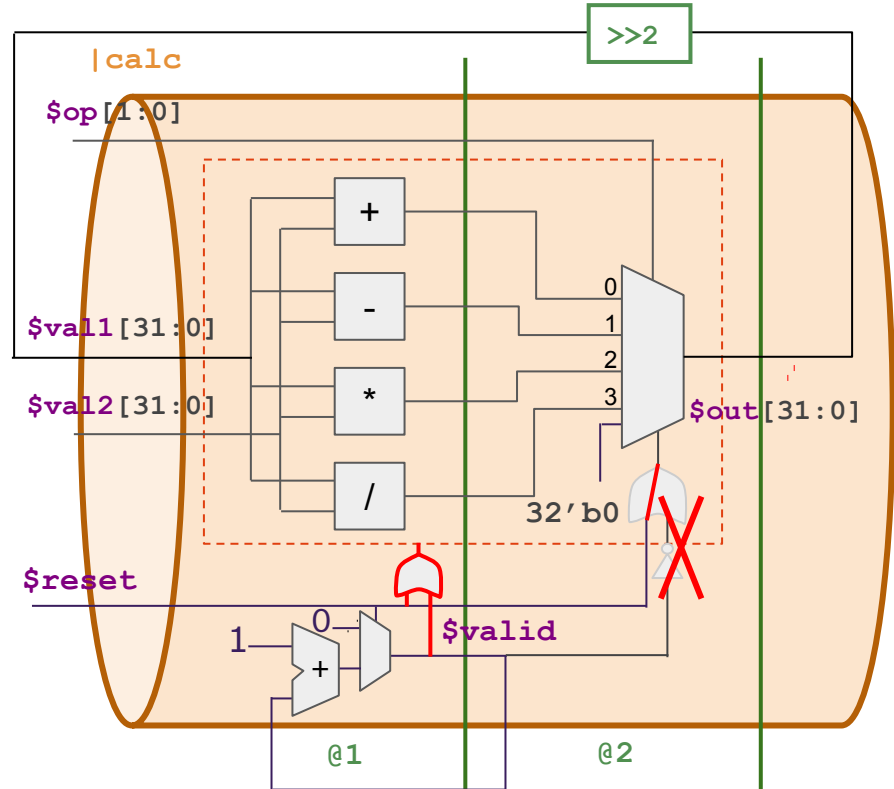
Lab: 2-Cycle Calculator with Validity

1. Use:
 $\$valid_or_reset = \$valid \mid \mid \$reset;$
as a when condition for calculation
instead of zeroing $\$out$.

For reference:

```
|calc
@1
  $valid = ...;
  ?? $valid
@1
  $aa_sq[31:0] = $aa * $aa;
  ...
```

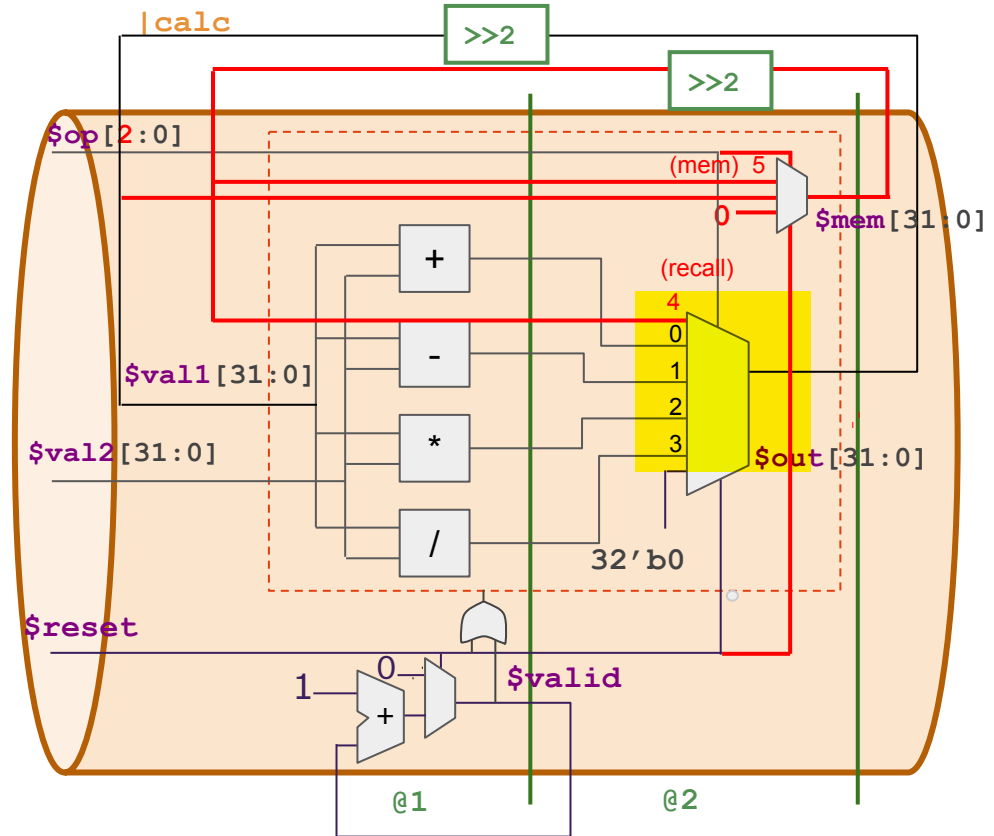
2. Verify behavior in waveform.



Lab: Calculator with Single-Value Memory

Calculators support “mem” and “recall”, to remember and recall a value.

1. Extend $\$op$ to 3 bits.
2. Add memory MUX.
3. Select recall value in output MUX.
4. Verify behavior in waveform.



Bonus Content

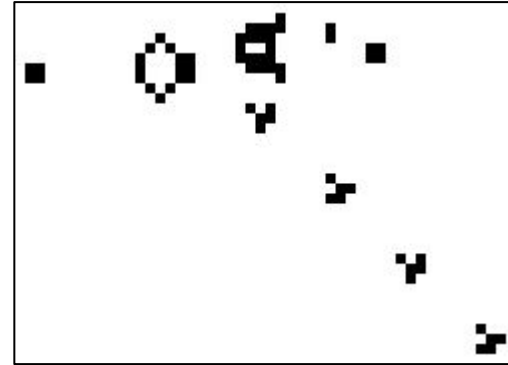


Hierarchy

```
|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] = ...;

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



Hierarchy

```
|default
  /yy[Y_SIZE-1:0]
  /xx[X_SIZE-1:0]
  @1
    $alive = ...;
|somewhere_else
  // ...
// "Lexical re-entrance"
|default
  @1
    /yy[*] // == [Y_SIZE-1:0]
    // Row vector.
    $row[X_SIZE-1:0] = /xx[*]$alive;
    /xx[*]
      $alive_to_right = /xx[(#xx + 1) % X_SIZE]$alive;
      $reset = /top|reset>>1$reset;

// http://makerchip.com/sandbox/0/0y8h1B
```

Lab: Hierarchy

Do the “Hierarchy” tutorial in a new Makerchip window.

The screenshot displays the Makerchip IDE interface. On the left, the 'EDITOR' pane shows a code snippet for a module named `m4_makerchip_module`. The code includes a Verilog-like structure with a `calc` block containing logic for calculating the square root of the sum of squares. A cursor is positioned over the 'TUTORIALS' menu item in the top navigation bar.

On the right, the 'TUTORIAL-HIER' pane is active, showing a 'LOAD PYTHAGOREAN EXAMPLE' button. Below the button, a text block explains the logic, mentioning input coordinates `$a` and `$bb`, and the behavioral hierarchy `/coord[1:0]`. The text states that `$aa` and `$bb` are now `/coord[0]$value` and `/coord[1]$value`, and that the new expression of this logic is illustrated in Figure 1.

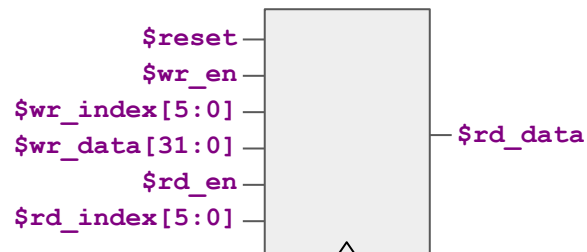
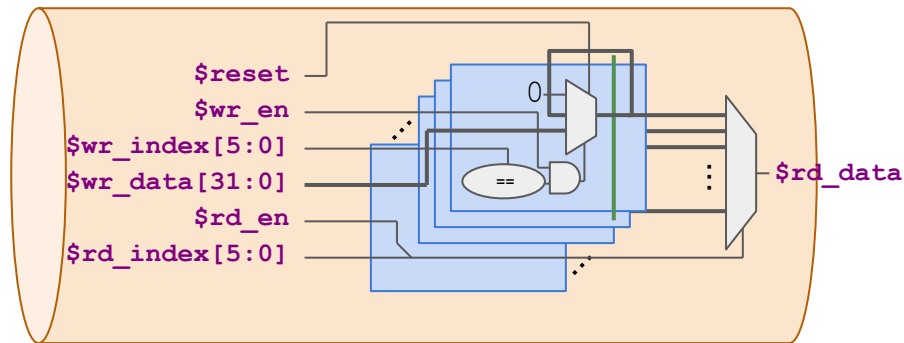
Figure 1: Pythagorean Theorem Logic. The diagram shows a block labeled `calc` with a sub-block `>coord[1:0]`. Inside this sub-block, two `rand` blocks generate values for `$value`, which are then squared (`^2`) to produce `$sq`. These `$sq` values are summed (`+`) to produce `$cc_sq`, which is then passed through a `sqrt` block to produce `$cc`. The final result `$cc` is printed (`print`). The diagram is divided into five numbered sections (0 to 4) representing different stages of the logic flow.

Arrays

A low-level implementation
of a 1-read, 1-write, array:

```
|pipe
@1
// Write
/entry[63:0]
  $wr = |pipe$wr_en && (#entry == |pipe$wr_index);
  $value[31:0] =
    |pipe$reset ? 32'b0 :
    $wr          ? |pipe$wr_data :
    $RETAIN;      // AKA: >>1$value

@2
// Read
?$rd_en
  $rd_data[31:0] = /entry[$rd_index]$value;
```



-

