
Cloud TPU v5e is purpose-built to bring the cost-efficiency and performance required for medium- and large-scale training and inference. TPU v5e delivers up to 2x higher training performance per dollar and up to 2.5x inference performance per dollar for LLMs and gen AI models compared to Cloud TPU v4. At less than half the cost of TPU v4, TPU v5e makes it possible for more organizations to train and deploy larger, more complex AI models.
Review

• Verilog is a common RTL design entry for synthesis
  • Examples of combinatorial logic
  • Continuous and non-continuous assignments

**reg** – can ‘hold’ a value. Often, but not always represents registers

LHS of signals assigned within ‘**always**’ statement

**wire** – cannot hold a value, has to be driven. Represents connections.

LHS of signals assigned outside ‘**always**’ statements (continuous assignment)

• Sequential logic uses flip-flops and (sometimes) latches
Sequential Logic in Verilog
State Elements in Verilog

Always blocks are the only way to specify the “behavior” of state elements. Synthesis tools will turn state element behaviors into state element instances.

D-flip-flop with synchronous set and reset example:

```verilog
module dff(q
  input d, clk, set, rst;
  output reg q;
);
always @(posedge clk)
  if (rst)
    q <= 1'b0;
  else if (set)
    q <= 1'b1;
  else
    q <= d;
endmodule
```

Unlike logic gates, there are no primitive flip-flops in Verilog. Although, it is possible to instantiate FPGA or standard-cell specific flip-flops.
The Sequential always Block

**Combinational**

```verilog
module comb(input a, b, sel, output reg out);
    always @(*) begin
        if (sel) out = b;
        else out = a;
    end
endmodule
```

**Sequential**

```verilog
module seq(input a, b, sel, clk, output reg out);
    always @(posedge clk) begin
        if (sel) out <= b;
        else out <= a;
    end
endmodule
```
Latches vs. Flip-Flops

Flip-Flop

```verilog
module flipflop
(input clk,
 input d,
 output reg q);

always @(posedge clk)
begin
  q <= d;
end
endmodule
```

Latch

```verilog
module latch
(input clk,
 input d,
 output reg q);

always @(clk or d)
begin
  if ( clk )
    q <= d;
end
endmodule
```
The use of **posedge** and **negedge** makes an **always** block sequential (edge-triggered).

- **Assign any signal or variable from only one** **always** **block.**
- **Be wary of race conditions:** **always** **blocks with same trigger execute concurrently**...
• Verilog supports two types of assignments within always blocks, with subtly different behaviors.

- Blocking assignment (=): evaluation and assignment are immediate

```verbatim
always @(*) begin
    x = a | b;   // 1. evaluate a|b, assign result to x
    y = a ^ b ^ c; // 2. evaluate a^b^c, assign result to y
    z = b & ~c;  // 3. evaluate b&(~c), assign result to z
end
```

- Nonblocking assignment (<=): all assignments deferred to end of simulation time step after all right-hand sides have been evaluated (even those in other active always blocks)

```verbatim
always @(*) begin
    x <= a | b;   // 1. evaluate a|b, but defer assignment to x
    y <= a ^ b ^ c; // 2. evaluate a^b^c, but defer assignment to y
    z <= b & ~c;  // 3. evaluate b&(~c), but defer assignment to z

// 4. end of time step: assign new values to x, y and z
end
```

Sometimes, as above, both produce the same result. Sometimes, not!
Assignment Styles for Sequential Logic

What we want:
Register-based digital delay line
(a.k.a. shift-register)

Will non-blocking and blocking assignments both produce the desired result?

module nonblocking(
    input in, clk,
    output reg out
);
    reg q1, q2;
    always @(posedge clk) begin
        q1 <= in;
        q2 <= q1;
        out <= q2;
    end
endmodule

module blocking(
    input in, clk,
    output reg out
);
    reg q1, q2;
    always @(posedge clk) begin
        q1 = in;
        q2 = q1;
        out = q2;
    end
endmodule
Use Nonblocking for Sequential Logic

```
always @(posedge clk) begin
    q1 <= in;
    q2 <= q1;  // uses old q1
    out <= q2;  // uses old q2
end
```

```
always @(posedge clk) begin
    q1 = in;
    q2 = q1;  // uses new q1
    out = q2;  // uses new q2
end
```

(“old” means value before clock edge, “new” means the value after most recent assignment)

“At each rising clock edge, q1, q2, and out simultaneously receive the old values of in, q1, and q2.”

“At each rising clock edge, q1 = in.
After that, q2 = q1.
After that, out = q2.
Therefore out = in.”

- Blocking assignments do not reflect the intrinsic behavior of multi-stage sequential logic
- Guideline: use nonblocking assignments for sequential always blocks
Example: A Simple Counter

// 4-bit counter with enable and synchronous clear
module counter(input clk, enb, clr,
               output reg [3:0] count);

always @(posedge clk) begin
  count <= clr ? 4'b0 : (enb ? count+1 : count);
end
endmodule
Example - Parallel to Serial Converter

module ParToSer(
  input [3:0] X;
  input ld, clk;
  output reg out;
);

reg [3:0] Q;
wire [3:0] NS;
assign NS = (ld) ? X : {Q[0], Q[3:1]};
always @ (posedge clk)
  Q <= NS;
assign out = Q[0];
endmodule

Specifies the muxing with "rotation"

forces Q register (flip-flops) to be rewritten every cycle

connect output
Simplified Verilog Guidelines

• Combinational logic:
  • Continuous Assignment:
    ```verilog
    assign a = b & c;
    ```
  • Always block with @(*)
    ```verilog
    always @(*) begin
    a = b & c;  // blocking statement
    end
    ```

• Sequential logic:
  • Always block with @(posedge clk)
    ```verilog
    always @(posedge clk) begin
    a <= b & c;  // nonblocking statement
    end
    ```
Verilog in EECS 151/251A

• We use behavioral modeling at the bottom of the hierarchy

• Use instantiation to 1) build hierarchy and, 2) map to FPGA and ASIC resources not supported by synthesis.

• Favor continuous assign and avoid always blocks unless:
  • No other alternative: ex: state elements, case
  • Helps readability and clarity of code: ex: large nested if else

• Use named ports.

• Refer to the class style manual!

• Verilog is a big language. This is only an introduction.
  • Harris & Harris book chapter 4 is a good source.
  • **Be careful of what you read on the web.** Many bad examples out there.
  • We will be introducing more useful constructs throughout the semester. Stay tuned!
Verilog vs. SystemVerilog

- **always** statements in Verilog can be used to infer flip-flops, latches or logic
  - Depends on the sensitivity list and the statement
  - Easy to create confusion

- System Verilog adds disambiguation:
  - **always_ff** for flip-flops
  - **always_latch** for latches
  - **always_comb** for combinational logic
Administrivia

• Discussion on Friday 8am cancelled
• Lab 2 finished
  • If you couldn’t attend the lab, please get checked off in office hours this week
• Lab 3 is this week
  • Cycle wraps up on Monday
• Homework 2 due this Friday
  • Homework 3 posted on Thursday
Verilog Testbenches
Simulating the Circuit

- Once we have a circuit in Verilog (device under test, or DUT), we would like to test it.
  - Instantiate the DUT and supply its inputs via a testbench.
    - Simple
    - Comprehensive
    - Random

- `initial` statement supplies the stimuli.
• **Device under test (DUT)**
• **Stimulus**: Internal generator or imported vectors
• **Response**: Logged or compared internally
Testbench basics

• Example clock
  
  ```verilog
  reg clk;

  initial clk = 0;
  always #(`CLOCK_PERIOD/2) clk <= ~clk;
  ```

• Example inputs

  ```verilog
  initial begin
  in <= 4’h0;
  @(negedge clk) in<= 4’h1;
  ...
  (sets up inputs on the negedge, so they are ready at the posedge)
  ```

Only small DUTs can be tested exhaustively
Verilog Simulators

• Elements:
  • Processes: built-in gates, always blocks, continuous assignments, etc
  • Events: Events require a simulator to take an action. E.g: transition
  • Simulation time: Elapsed time (e.g. in cycles)
SystemVerilog Stimuli

• So far, the test vectors were created by us (“Directed testing”)
  • Can we exhaustively test a design?

• SystemVerilog provides constructs for constrained random stimuli
  • Random stimulus: `rand bit [3:0] RandomInput`
    • Declares a variable `RandomInput` which can take a uniform random value 0-15
Coverage-Driven Verification in SystemVerilog

• Like in software, we can have
  • Code coverage
  • Functional coverage

• Code coverage:
  • Line coverage – lines exercised during simulation
  • Branch coverage – were if-else and case statements exercised
  • Expression or condition coverage – whether all conditions have been exercised
  • State or transition coverage – coverage of state machines
  • Toggle coverage – whether the variables have taken all transitions (0-to-1 and 1-to-0)

• Functional coverage: whether the design features have been exercised
  – needs a coverage model
• Constructs: covergroup, coverpoint, bins
Properties and Assertions

• Properties are linear-time temporal logic conditions
  • Logical expressions, sequences

• Assertion checks whether a property holds
  • Immediate or Concurrent

• Covers observe whether the property can be satisfied with the test
module m(input logic c, clk);
  logic a = 1'b0;
  logic b = 1'b1;
  always @(posedge clk) begin
    a <= c;
    b <= !c;
  end
  // assertion
  // SystemVerilog assertion
  a1: assert property (@(posedge clk) a != b)
    else $error("a != b does not hold");
endmodule : m
Final Thoughts on Verilog Examples

Verilog looks like C, but it describes hardware:
Entirely different semantics: multiple physical elements with parallel activities and temporal relationships.

A large part of digital design is knowing how to write Verilog that gets you the desired circuit. First understand the circuit you want then figure out how to code it in Verilog. If you try to write Verilog without a clear idea of the desired circuit, you will struggle.

As you get more practice, you will know how to best write Verilog for a desired result.

Be suspicious of the synthesis tools! Check the output of the tools to make sure you get what you want.
Clicker Question

• How many stimuli to exhaustively test a 32-b adder?
  A) 32
  B) 64
  C) 65,536
  D) 4,294,967,296
  E) 18,446,744,073,709,551,616
Combinational Logic
Combinational Logic

- The outputs depend *only* on the current values of the inputs.
  - Memoryless: compute the output values using the current inputs.
Combinational Logic Example

Truth Table Description:

<table>
<thead>
<tr>
<th>x0</th>
<th>x1</th>
<th>y</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

Boolean Equations:

\[ y = x_0 \text{ OR } x_1 \]
\[ = x_0 + x_1 \]

Gate Representations:
Relationship Among Representations

- **Truth Table**
  - Unique

- **Boolean Expression**
  - Convenient for manipulation

- **Gate Representation**
  - Close to Implementation
Boolean Algebra
Boolean Algebra Background

• Logic: The study of the principles of reasoning.
• The 19th Century Mathematician, George Boole, developed a math. system (algebra) involving logic, Boolean Algebra.
  • His variables took on TRUE, FALSE.
• Later Claude Shannon (father of information theory) showed (in his Master's thesis!) how to map Boolean Algebra to digital circuits.
• Alan Turing has shown that a Boolean machine can compute any computable number
Boolean Algebra Fundamentals

• Two elements \( \{0, 1\} \)

• Two binary operators: AND (\( \cdot \)) OR (\(+\))

• One unary operator: NOT (\( \overline{\cdot} \), \( \acute{\cdot} \))
Boolean Operations

- Given two variables \((x, y)\), 16 logic functions

<table>
<thead>
<tr>
<th>(X)</th>
<th>(Y)</th>
<th>(F_0)</th>
<th>(F_1)</th>
<th>(F_2)</th>
<th>(F_3)</th>
<th>(F_4)</th>
<th>(F_5)</th>
<th>(F_6)</th>
<th>(F_7)</th>
<th>(F_8)</th>
<th>(F_9)</th>
<th>(F_A)</th>
<th>(F_B)</th>
<th>(F_C)</th>
<th>(F_D)</th>
<th>(F_E)</th>
<th>(F_F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Laws of Boolean Algebra

• Identities, null elements:
  • \(X + 0 = X\), \(X \cdot 1 = X\)
  • \(X + 1 = 1\), \(X \cdot 0 = 0\)

• Idempotency:
  • \(X + X = X\), \(X \cdot X = X\)

• Complements:
  • \(X + X' = 1\), \(X \cdot X' = 0\)

• Commutativity:
  • \(X + Y = Y + X\), \(X \cdot Y = Y \cdot X\)

• Associativity:
  • \((X + Y) + Z = X + (Y + Z) = X + Y + Z\)
  • \((X \cdot Y) \cdot Z = X \cdot (Y \cdot Z) = X \cdot Y \cdot Z\)

• Distributivity:
  • \(X \cdot (Y + Z) = (X \cdot Y) + (X \cdot Z)\)
  • \(X + (Y \cdot Z) = (X + Y) \cdot (X + Z)\)

• Duality:
  • \(\text{AND} \rightarrow \text{OR} \) and vice versa
  • \(0 \rightarrow 1\) and vice versa
  • Leave literals unchanged

\[
\{F(x_1, x_2, \ldots, x_n, 0, 1, +, \cdot)\}^D = \{F(x_1, x_2, \ldots, x_n, 1, 0, \cdot, +)\}
\]

• Literals are variables or their complements
Proving Distributive Law

- $X \cdot (Y+Z) = (X\cdot Y) + (X \cdot Z)$

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>(Y+Z)</th>
<th>X • (Y+Z)</th>
<th>(X•Y)</th>
<th>(X•Z)</th>
<th>(X•Y) + (X•Z)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
Proving Distributive Law

\[ X \cdot (Y+Z) = (X \cdot Y) + (X \cdot Z) \]

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>(Y+Z)</th>
<th>X • (Y+Z)</th>
<th>(X•Y)</th>
<th>(X•Z)</th>
<th>(X•Y) + (X•Z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
DeMorgan's Law

- Theorem for complementing a complex function.

\[(x + y)' = x' \cdot y'\]

\[(x \cdot y)' = x' + y'\]
DeMorgan's Law

• Procedure for complementing a complex function.

\[(x + y)' = x' \cdot y'\]

\[
\begin{array}{c|c|c|c|c|c}
  x & y & x' & y' & (x + y)' & x' \cdot y' \\
  0 & 0 & 1 & 1 & 1 & 1 \\
  0 & 1 & 1 & 0 & 0 & 0 \\
  1 & 0 & 0 & 1 & 0 & 0 \\
  1 & 1 & 0 & 0 & 0 & 0 \\
\end{array}
\]

\[(x \cdot y)' = x' + y'\]

\[
\begin{array}{c|c|c|c|c|c}
  x & y & x' & y' & (x \cdot y)' & x' + y' \\
  0 & 0 & 1 & 1 & 1 & 1 \\
  0 & 1 & 1 & 0 & 1 & 1 \\
  1 & 0 & 0 & 1 & 1 & 1 \\
  1 & 1 & 0 & 0 & 0 & 0 \\
\end{array}
\]
Summary

• Sequential logic uses flip-flops and (sometimes) latches
  • Flip-flops and latches are inferred in Verilog
    • Always blocks

• Blocking and non-blocking assignments

• Verification effort is equal to the design effort
  • Testbenches, simulation, coverage, assertions

• Practice is the best way to learn a new language...

• Combinational logic block outputs depend only on its inputs

• Boolean algebra can be used for manipulation and simplification of Boolean equations