Embedded Systems: Week 4 - Verilog Hardware Description Language (Verilog HDL)

Course Overview: Welcome to Week 4, where we unlock the power of **Verilog Hardware Description Language (Verilog HDL)**, an essential tool for designing modern embedded systems. In the realm of digital design, Verilog is more than just a programming language; it's a specialized language used to describe the structure and behavior of electronic circuits. This module will equip you with the fundamental concepts and practical skills needed to model, simulate, and synthesize digital hardware using Verilog. We will explore various modeling styles, from low-level gate descriptions to high-level behavioral representations, and understand how your Verilog code translates into actual silicon. Mastering Verilog is a crucial step towards building complex embedded systems that integrate custom hardware.

Learning Objectives: Upon successful completion of this module, you will be able to:

- **Explain** the purpose and significance of Hardware Description Languages (HDLs) in the digital design flow.
- **Identify and differentiate** fundamental Verilog lexical conventions, including data types, literals, and operators.
- **Apply** various Verilog modeling styles—gate-level, dataflow, and behavioral—to describe digital circuits.
- **Implement** both combinational and sequential logic circuits using appropriate Verilog constructs.
- **Understand** the critical distinction between blocking and non-blocking assignments in behavioral modeling for correct hardware inference.
- **Develop** basic Verilog testbenches to simulate and verify the functional correctness of designed hardware modules.
- **Grasp** the fundamental principles of logic synthesis and distinguish between synthesizable and non-synthesizable Verilog constructs.

Module 4.1: Introduction to Hardware Description Languages (HDLs)

This section introduces the foundational concept of HDLs, explaining why they are indispensable tools in modern digital design and how they differ from conventional software programming languages.

- **4.1.1 What are Hardware Description Languages (HDLs)?**
    1. **Purpose:** HDLs are specialized computer languages used to formally describe the structure and behavior of electronic circuits, ranging from simple gates to complex microprocessors and entire systems-on-chip (SoCs). They serve as a textual, machine-readable representation of hardware.
    2. **Why HDLs are Needed:**
        - **Complexity Management:** Modern digital circuits contain millions or even billions of transistors. Drawing schematics manually for such

complexity is impractical and error-prone. HDLs allow designers to describe hardware at a higher level of abstraction.

- **Verification:** HDLs enable simulation, where the described hardware's behavior can be tested rigorously before actual fabrication. This is crucial for catching design flaws early, which is significantly cheaper than fixing them post-fabrication.
- **Synthesis:** HDLs can be translated by specialized tools (synthesizers) into physical gate-level netlists, which are then used to create the actual silicon layout. This automates the conversion from behavioral description to physical implementation.
- **Portability and Reusability:** HDL descriptions are technology-independent (initially), allowing the same design to be targeted to different fabrication processes (e.g., ASICs or FPGAs). Modules can be reused in different projects.
- **Documentation:** HDL code serves as precise and unambiguous documentation of the hardware's design.

- **4.1.2 Comparison with Software Programming Languages** While HDLs share syntax similarities with software languages like C, their underlying semantics are fundamentally different due to the nature of hardware.
  1. **Parallelism/Concurrency:**
     - **Software:** Instructions in typical software programs execute sequentially, one after another, unless explicitly managed for parallelism (e.g., threads, processes).
     - **Hardware (HDL):** Operations described in an HDL are inherently **concurrent** or **parallel**. All parts of a digital circuit (gates, registers) operate simultaneously unless explicitly designed to be sequential (e.g., a state machine). This is a critical distinction that impacts how HDL code is written and interpreted.
  2. **Time:**
     - **Software:** Time is conceptual; execution speed depends on processor speed.
     - **Hardware (HDL):** Time is explicit and fundamental. Propagation delays, clock cycles, and timing relationships are inherent to the description. HDLs include constructs for precise timing control (e.g., `#` delay operator).
  3. **Hardware Description vs. Algorithm Execution:**
     - **Software:** Describes algorithms that execute on a general-purpose processor. The focus is on *what* to compute.
     - **Hardware (HDL):** Describes the physical structure and behavior of the hardware itself. The focus is on *how* data flows and *how* operations are performed by dedicated circuitry.
  4. **Compilation vs. Synthesis:**
     - **Software:** Compiled into machine code for a specific CPU architecture.
     - **Hardware (HDL):** Synthesized into a gate-level netlist (a list of interconnected logic gates) that can be mapped onto a specific

hardware technology (e.g., FPGA Lookup Tables, ASIC standard cells).

- **4.1.3 Verilog HDL in the Digital Design Flow** Verilog plays a central role in the modern electronic design automation (EDA) flow:
    1. **Specification:** Initial requirements and desired functionality.
    2. **Architectural Design:** High-level block diagram and data flow.
    3. **HDL Modeling (Verilog Coding):** Describing the hardware using Verilog. This can be at various levels of abstraction (gate, dataflow, behavioral, structural).
    4. **Simulation & Verification:** Using Verilog simulators to test the functional correctness of the design by applying stimuli (inputs) and observing responses (outputs). Testbenches (also written in Verilog) are crucial here.
    5. **Logic Synthesis:** Translating the synthesizable Verilog code into a technology-dependent netlist of logic gates (e.g., a list of AND, OR, NOT gates, flip-flops) optimized for a target technology (ASIC or FPGA).
    6. **Physical Design (Place & Route):** Arranging the gates on the chip (place) and connecting them with wires (route) according to timing and area constraints.
    7. **Post-Layout Simulation:** Simulating the design with actual wire delays extracted from the physical layout for final timing verification.
    8. **Fabrication:** Manufacturing the actual silicon chip.
    9. **Testing:** Verifying the fabricated chip.

Module 4.2: Verilog Basics and Lexical Conventions

Before writing Verilog code, it's essential to understand its fundamental building blocks: how elements are named, what types of data can be represented, and how operations are performed.

- **4.2.1 Keywords, Identifiers, Comments, White Spaces**
    - **Keywords:** Reserved words in Verilog that have special meaning (e.g., `module`, `endmodule`, `input`, `output`, `wire`, `reg`, `assign`, `always`, `initial`). These cannot be used as identifiers. All keywords are lowercase.
    - **Identifiers:** Names given to objects in the design, such as modules, ports, signals (wires, registers), and parameters. They must start with a letter or underscore, followed by letters, numbers, underscores, or dollar signs. They are case-sensitive (e.g., `my_signal` is different from `My_Signal`).
    - **Comments:** Used to explain the code and are ignored by the compiler/synthesizer.
        - **Single-line comments:** Start with `//` and extend to the end of the line.
        - **Multi-line comments:** Enclosed between `/*` and `*/`.
    - **White Spaces:** Spaces, tabs, newlines are generally ignored by the Verilog compiler, used to improve code readability.

- **4.2.2 Data Types: Nets, Registers, and Other Types** Verilog categorizes data into types based on how they store and transmit values, reflecting actual hardware behavior.
  - **Nets (or Wires):**
    - **Representation:** Represent physical connections between hardware elements (e.g., wires on a circuit board).
    - **Behavior:** Nets do not store a value; their value is continuously driven by the output of a connected component (a gate, a module instance, or an `assign` statement). If nothing drives a net, its value is high-impedance (`z`). If multiple drivers conflict, the value becomes unknown (`x`).
    - **Declaration:** `wire` is the most common net type. Others include `tri` (tristate), `wand`, `wor` (for wired-AND/OR logic).
    - **Example:** `wire enable_signal;` or `wire [7:0] data_bus;` (for an 8-bit bus).
  - **Registers (or Variables):**
    - **Representation:** Represent data storage elements in hardware (e.g., flip-flops, latches, registers, or temporary variables in behavioral blocks).
    - **Behavior:** Registers *store* a value until a new value is explicitly assigned to them. They hold their last assigned value.
    - **Declaration:** `reg` is the most common register type.
    - **Usage:** Used inside `initial` and `always` procedural blocks. A `reg` variable can hold a value, but it does *not* necessarily imply a hardware register. If assigned combinatorially within an `always` block, it might infer combinational logic or a latch. If assigned on a clock edge, it infers a flip-flop.
    - **Example:** `reg control_state;` or `reg [15:0] count_value;`
  - **Integer:** A general-purpose register variable for integer arithmetic. Used primarily for loop counters or general-purpose variables in behavioral blocks, not typically for hardware inference. Defaults to 32-bit signed.
  - **Time:** A 64-bit unsigned quantity used to store simulation time (often used with system tasks like `$time`).
  - **Real/Realtime:** For floating-point numbers. Not synthesizable; used only in simulations (e.g., for analog models or complex calculations in testbenches).
  - **Parameters:** Constant values declared using the `parameter` keyword. Used for defining fixed sizes (e.g., bit-widths, array sizes) or timing constants. They improve code readability and reusability.
    - **Example:** `parameter DATA_WIDTH = 8;`
- **4.2.3 Literals: Number and String Representation**
  - **Number Literals:**
    - **Syntax:** `size'base_format value`
      - `size`: Optional, decimal number specifying the bit width (e.g., `8` for 8 bits).
      - `base_format`: Required, specifies the number base:

                - `'b` or `'B` for binary (e.g., `4'b1011`)
                - `'o` or `'O` for octal (e.g., `12'o77`)
                - `'d` or `'D` for decimal (e.g., `16'd255`, or simply `255` if size is not specified)
                - `'h` or `'H` for hexadecimal (e.g., `8'hFF`)
            - `value`: The number itself.
        - **Default Size:** If `size` is omitted, the default size is system-dependent (usually 32 bits).
        - **Underscore (`_`):** Allowed in numbers for readability, ignored by compiler (e.g., `16'b1010_1010_1111_0000`).
        - **Unknown (`x`) and High-Impedance (`z`):** Can be used in numbers (e.g., `4'b10xz`, `8'hFz`). These are crucial for modeling unknown or high-impedance states in hardware.
    - **String Literals:** Enclosed in double quotes (e.g., `"Hello World!"`). Primarily used for display messages in testbenches.
- **4.2.4 Operators: The Actions of Hardware** Verilog provides a rich set of operators to describe computations and logical relationships.
    - **Arithmetic Operators:** `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), `%` (modulo), `**` (power - non-synthesizable).
    - **Relational Operators:** `>`, `<`, `>=`, `<=`, `==` (equality), `!=` (inequality), `===` (case equality, includes x/z), `!==` (case inequality, includes x/z). Used for comparisons.
    - **Logical Operators:** `&&` (logical AND), `||` (logical OR), `!` (logical NOT). Operate on single-bit (boolean) operands; return 0 or 1. If an operand is `x` or `z`, the result can be `x`.
    - **Bitwise Operators:** `&` (bitwise AND), `|` (bitwise OR), `~` (bitwise NOT), `^` (bitwise XOR), `~^` or `^~` (bitwise XNOR). Operate bit by bit on operands; return multi-bit results.
    - **Reduction Operators:** `&` (reduction AND), `|` (reduction OR), `~` (reduction NOT), `^` (reduction XOR), `~^` or `^~` (reduction XNOR). Operate on a single multi-bit operand and produce a single-bit result (e.g., `&A` checks if all bits of `A` are 1).
    - **Shift Operators:** `<<` (left shift), `>>` (right shift), `<<<` (arithmetic left shift), `>>>` (arithmetic right shift). Used for bit manipulation. Arithmetic shifts preserve the sign bit for signed numbers.
    - **Concatenation Operator:** `{}` (e.g., `{A, B, C}` concatenates signals A, B, and C into a larger vector).
    - **Replication Operator:** `{num_copies {vector}}` (e.g., `{4{1'b1}}` creates `4'b1111`).
    - **Conditional Operator (Ternary Operator):** `condition ? true_expression : false_expression`. A single-line `if-else` equivalent, widely used in dataflow modeling for multiplexers.
        - **Example:** `assign out = sel ? in1 : in0;`

Module 4.3: Modeling Techniques in Verilog

Verilog offers several distinct modeling styles, each suitable for describing hardware at different levels of abstraction. Understanding these styles is crucial for effective and efficient hardware design.

- **4.3.1 Gate-Level Modeling: The Lowest Abstraction**
  - **Concept:** Describes a circuit in terms of interconnected basic logic gates (primitives) as provided by Verilog's built-in gate types. This is the lowest level of abstraction where you explicitly define each gate and its connections. It directly corresponds to a schematic diagram.
  - **Built-in Primitives:** Verilog provides predefined gate primitives:
    - `and`, `nand`, `or`, `nor`, `xor`, `xnor`: Two or more inputs, one output.
    - `buf`, `not`: One input, one or more outputs (buffers and inverters).
    - `bufif0`, `bufif1`, `notif0`, `notif1`: Tristate buffers and inverters, enabled when the control signal is 0 or 1.
  - **Instantiation:** Gates are instantiated (placed) within a `module` using the primitive name, followed by an optional instance name, and then port connections.
    - **Port Ordering:** `gate_type (output, input1, input2, ...);` (e.g., `and A1 (out_signal, in1, in2);`)
    - **Named Port Connection:** `gate_type ( .output_port(output_signal), .input1_port(input1_signal), ...);` (e.g., `and A1 (.out(out_signal), .i1(in1), .i2(in2));` - safer for readability and maintainability).
  - **Strengths:** Direct mapping to physical gates, useful for very small, custom logic blocks where exact gate structure is critical.
  - **Weaknesses:** Extremely tedious and error-prone for complex designs, very difficult to debug and modify. Not scalable.

Example: 2-input XOR Gate from NAND gates:
Verilog
```
module XOR_gate_structural (
    output wire Y,
    input wire A,
    input wire B
);
    wire N1, N2, N3; // Internal wires for intermediate signals

    nand #(2,3) NAND1 (N1, A, B);        // Gate instance with delay
    nand NAND2 (N2, A, N1);
    nand NAND3 (N3, N1, B);
    nand NAND4 (Y, N2, N3);
```

endmodule

  - (Note: `#` represents a delay, usually only relevant for simulation.)

- **4.3.2 Dataflow Modeling: Describing Concurrent Data Assignment**
  - **Concept:** Describes a circuit in terms of how data flows through it and how signals are continuously assigned values based on expressions. This is a higher level of abstraction than gate-level, focusing on the relationship between inputs and outputs. It implicitly infers combinational logic.
  - **`assign` Statement:** The primary construct for dataflow modeling. It is a **continuous assignment** statement. The expression on the right-hand side is continuously evaluated, and its result is immediately assigned to the net (wire) on the left-hand side. Any change in an operand on the right-hand side triggers a re-evaluation and update.
    - **Syntax:** `assign net_name = expression;`
    - **Left-Hand Side (LHS):** Must be a `net` type (e.g., `wire`).
    - **Right-Hand Side (RHS):** Can be any valid expression involving nets, registers, or literals.
  - **Net Declaration Assignments:** You can combine the declaration of a wire with an initial assignment.
    - **Example:** `wire sum = A ^ B;`
  - **Implicit Nets:** If a net is used without explicit declaration (e.g., in an `assign` statement), Verilog implicitly declares it as a `wire`. While convenient, this is generally bad practice and can lead to errors. Always explicitly declare all nets.
  - **Strengths:** Concise and readable for combinational logic, directly synthesizable, good for modeling arithmetic and logical operations, multiplexers, decoders.
  - **Weaknesses:** Cannot describe sequential logic (flip-flops, latches) or complex control flow (loops, state machines).

Example: 2-to-1 Multiplexer using `assign`:

Verilog
```
module MUX2_1_dataflow (
   output wire Y,
   input wire D0,
   input wire D1,
   input wire S  // Select line
);
   assign Y = S ? D1 : D0; // Conditional operator
endmodule
```

  -

Example: 4-bit Ripple Carry Adder Dataflow:

Verilog
```
module RippleCarryAdder_4bit_dataflow (
   output wire [3:0] Sum,
   output wire       CarryOut,
   input wire  [3:0] A,
   input wire  [3:0] B,
   input wire        CarryIn
);
   wire c1, c2, c3; // Internal carries

   // Full Adder 0
   assign {c1, Sum[0]} = A[0] + B[0] + CarryIn;
   // Full Adder 1
   assign {c2, Sum[1]} = A[1] + B[1] + c1;
   // Full Adder 2
   assign {c3, Sum[2]} = A[2] + B[2] + c2;
   // Full Adder 3
   assign {CarryOut, Sum[3]} = A[3] + B[3] + c3;

endmodule
```

    - (Note: `{CarryOut, Sum[3]}` uses concatenation for multi-bit assignment. Verilog's `+` operator infers an adder.)
- **4.3.3 Behavioral Modeling: Describing Sequential and Complex Logic**
    - **Concept:** Describes a circuit's behavior at an algorithmic level, focusing on *what* the circuit does rather than *how* it's implemented with gates. This is the highest level of abstraction for hardware description in Verilog. It uses procedural blocks (`initial`, `always`) to describe operations that execute sequentially within the block, but concurrently with other blocks.
    - **Procedural Blocks:**
        - **`initial` block:** Executes only once at the beginning of simulation (time 0). Primarily used in testbenches for setting up initial conditions, applying stimuli, and generating reports. **Not synthesizable.**
        - **`always` block:** Executes repeatedly whenever a signal in its **sensitivity list** changes value (for combinational logic) or on a specific clock edge (for sequential logic). This is the primary block for synthesizable behavioral code.
            - **Syntax:** `always @(sensitivity_list)`
            - **Sensitivity List:** Specifies the events that trigger the execution of the `always` block.
                - `always @(*)`: For **combinational logic**. This is SystemVerilog syntax that automatically includes all inputs used in the block. In older Verilog-2001, you had to list all inputs manually (e.g., `always @(in1 or in2 or select)`).
                - `always @(posedge clk)`: For **rising-edge triggered sequential logic** (flip-flops).
                - `always @(negedge clk)`: For **falling-edge triggered sequential logic**.

            - `always @(posedge clk or negedge reset_n)`: For sequential logic with an asynchronous reset.
    - **Blocking vs. Non-blocking Assignments:** This is a crucial concept for correct hardware modeling.
        - **Blocking Assignment (`=`):**
            - **Behavior:** Assignments execute **sequentially** within the procedural block. The right-hand side is evaluated, and the value is assigned to the left-hand side *before* the next statement in the block executes. It "blocks" the flow until complete.
            - **When to Use:** Primarily for **sequential logic** where you want immediate updates within a sequence, or for procedural variables within an `initial` block in testbenches. **Generally avoid for multi-statement combinational logic within `always` blocks as it can lead to unintended sequential behavior (implied latches) or simulation-synthesis mismatches.**
        - **Non-blocking Assignment (`<=`):**
            - **Behavior:** The right-hand side is evaluated *at the current time*, but the assignment to the left-hand side is *scheduled to occur at the end of the current time step* (or at the next clock edge for sequential logic). All non-blocking assignments within an `always` block are processed **concurrently** at the scheduled time.
            - **When to Use: Always use for modeling sequential logic (flip-flops, registers) inside `always @(posedge clk)` blocks.** This ensures that all flip-flops update simultaneously based on values from the start of the clock cycle, mirroring true hardware behavior. Also recommended for multi-statement combinational logic within `always @(*)` blocks to avoid implicit latches.
    - **Procedural Assignments to `reg` Type Variables:** Variables (like `reg`) can only be assigned values inside `initial` or `always` blocks. Nets (wires) cannot be assigned in procedural blocks.
    - **Control Flow Statements (within procedural blocks):**
        - `if-else if-else`: Standard conditional branching.
        - `case-casex-casez`: Multi-way branching, useful for state machines or decoders. `casex` treats `x` or `z` in the case expression or case items as don't cares. `casez` treats `z` as don't cares.
        - `for`, `while`, `repeat`, `forever`: Looping constructs. Primarily synthesizable for **fixed loop counts** (unrolled loops) or finite iterations. `forever` is generally for testbenches (e.g., clock generation).
    - **Event Control (`@`) and Timing Control (`#`):**

        - `@ (event)`: Event control. Waits for a specified event to occur (e.g., `posedge clk`, `negedge reset`, `value_change`).
        - `# delay`: Timing control. Specifies a time delay. Used in simulation for modeling gate delays or for generating waveforms in testbenches. **Not synthesizable.**
    - **Strengths:** Highly expressive, allows modeling complex sequential and combinational logic concisely, ideal for state machines, data paths, and control logic.
    - **Weaknesses:** Careful use of blocking/non-blocking assignments is critical to avoid simulation-synthesis mismatches. Requires clear understanding of hardware inference.
- **4.3.4 Structural Modeling: Connecting Modules Hierarchically**
    - **Concept:** Describes a circuit as an interconnection of instances of other (sub)modules or primitives. This promotes modularity and hierarchy, allowing complex designs to be broken down into smaller, manageable, and reusable blocks.
    - **Module Instantiation:**
        - **Syntax:** `module_name instance_name (port_connections);`
        - **Module Definition:** First, you define the sub-module (e.g., `full_adder_module`).
        - **Instantiation:** Then, you "place" instances of this sub-module within a higher-level module. Each instance gets a unique name.
    - **Connecting Modules (Port Mapping):**
        - **Positional Port Mapping:** `(port1_signal, port2_signal, ...)` - Connects signals based on their order in the sub-module's port list. **Error-prone** if port order changes in the sub-module.
        - **Named Port Mapping:** `(.sub_module_port_name(connecting_signal), ...)` - Connects signals by explicitly naming the sub-module's port. **Recommended** for clarity and robustness against port list changes.
    - **Hierarchy:** Verilog supports hierarchical design, where a top-level module instantiates lower-level modules, which can in turn instantiate even lower-level modules. This mirrors the hierarchical structure of real hardware.
    - **Strengths:** Promotes modularity, reusability, simplifies debugging, crucial for large-scale designs.
    - **Weaknesses:** Requires defining all sub-modules first.

Example: 4-bit Ripple Carry Adder using Structural Modeling (instantiating Full Adder modules):

Verilog

```
// Define a Full Adder module (could be behavioral or dataflow)
module FullAdder (
    output wire sum,
    output wire carry_out,
    input wire a,
    input wire b,
    input wire carry_in
);
    assign {carry_out, sum} = a + b + carry_in; // Behavioral/Dataflow
endmodule

// Define the 4-bit Ripple Carry Adder using structural modeling
module RippleCarryAdder_4bit_structural (
    output wire [3:0] Sum,
    output wire       CarryOut,
    input wire  [3:0] A,
    input wire  [3:0] B,
    input wire        CarryIn
);
    wire c1, c2, c3; // Internal carries between full adders

    // Instantiate four Full Adder modules
    FullAdder FA0 (
        .sum(Sum[0]),
        .carry_out(c1),
        .a(A[0]),
        .b(B[0]),
        .carry_in(CarryIn)
    );

    FullAdder FA1 (
        .sum(Sum[1]),
        .carry_out(c2),
        .a(A[1]),
        .b(B[1]),
        .carry_in(c1)
    );

    FullAdder FA2 (
        .sum(Sum[2]),
        .carry_out(c3),
        .a(A[2]),
        .b(B[2]),
        .carry_in(c2)
    );

    FullAdder FA3 (
        .sum(Sum[3]),
        .carry_out(CarryOut),
        .a(A[3]),
        .b(B[3]),
        .carry_in(c3)
    );

endmodule
```

- ○

Module 4.4: Combinational Logic Design using Verilog

This section focuses on translating combinational logic functions into Verilog code, primarily using `assign` statements and `always @(*)` blocks.

- **4.4.1 Review of Combinational Logic Properties:**
  - Output depends *only* on current inputs.
  - No memory elements.
  - No clock signal is required for function, only for synchronization if part of a larger synchronous system.
  - Delay is due to gate propagation.
- **4.4.2 Implementing Common Combinational Circuits:**
  - **Multiplexers (Muxes):**

Dataflow (`assign`): Best and most concise way. Uses conditional operator.

Verilog
```
module MUX4_1 (output wire Y, input wire D0, D1, D2, D3, input wire [1:0] Sel);
  assign Y = Sel[1] ? (Sel[0] ? D3 : D2) : (Sel[0] ? D1 : D0);
endmodule
```

■

Behavioral (`always @(*)` with `case` or `if-else`):

Verilog
```
module MUX4_1_behavioral (output reg Y, input wire D0, D1, D2, D3, input wire [1:0] Sel);
  always @(*) begin
    case (Sel)
      2'b00: Y = D0;
      2'b01: Y = D1;
      2'b10: Y = D2;
      2'b11: Y = D3;
      default: Y = 1'bx; // Handle unknown select lines
    endcase
  end
endmodule
```

■

Demultiplexers (Demuxes):

Verilog
```
module DEMUX1_4 (
  output wire Y0, Y1, Y2, Y3,
  input wire DataIn,
  input wire [1:0] Sel
);
  assign Y0 = (Sel == 2'b00) ? DataIn : 1'b0;
  assign Y1 = (Sel == 2'b01) ? DataIn : 1'b0;
  assign Y2 = (Sel == 2'b10) ? DataIn : 1'b0;
  assign Y3 = (Sel == 2'b11) ? DataIn : 1'b0;
endmodule
```

-

Decoders:

Verilog
```
module Decoder2_4 (
   output wire [3:0] Out,
   input wire [1:0] In
);
   assign Out[0] = (In == 2'b00); // Only 1 if true, else 0
   assign Out[1] = (In == 2'b01);
   assign Out[2] = (In == 2'b10);
   assign Out[3] = (In == 2'b11);
endmodule
```

-

Encoders (Priority Encoder example):

Verilog
```
module PriorityEncoder4_2 (
   output reg [1:0] EncodedOut,
   output reg Valid,
   input wire [3:0] In
);
   always @(*) begin
      Valid = 1'b1; // Default to valid
      case (In)
         4'b0001: EncodedOut = 2'b00;
         4'b001x: EncodedOut = 2'b01; // x for don't care, e.g., 0010, 0011
         4'b01xx: EncodedOut = 2'b10;
         4'b1xxx: EncodedOut = 2'b11;
         default: begin Valid = 1'b0; EncodedOut = 2'bxx; end // No input active or multiple
      endcase
   end
endmodule
```

-
- **Adders (Half, Full, Ripple-Carry):**
  - Already demonstrated in Dataflow and Structural sections. Behavioral could also use `always @(*)` and the `+` operator.

Comparators:
Verilog
```
module MagnitudeComparator_8bit (
   output wire A_gt_B, A_eq_B, A_lt_B,
   input wire [7:0] A, B
);
   assign A_eq_B = (A == B);
   assign A_gt_B = (A > B);
   assign A_lt_B = (A < B);
endmodule
```

   ○

Module 4.5: Sequential Logic Design using Verilog

This section focuses on modeling circuits that have memory, where outputs depend on past inputs, achieved through explicit clocking and the proper use of non-blocking assignments.

- **4.5.1 Review of Sequential Logic Properties:**
    - Outputs depend on current inputs AND past inputs (state).
    - Contains memory elements (flip-flops, latches).
    - Requires a clock signal for synchronous operation.
    - Behavior is defined by state transitions.
- **4.5.2 Registers, Latches, and Flip-flops:**

D-Flip-flop (DFF): The most fundamental sequential element in synchronous design.
Verilog
```
module D_FF (
   output reg Q,
   input wire D,
   input wire clk
);
   always @(posedge clk) begin
      Q <= D; // Non-blocking assignment for sequential logic
   end
endmodule
```

   ○

D-Flip-flop with Asynchronous Reset:
Verilog
```
module D_FF_AsyncReset (
   output reg Q,
   input wire D,
   input wire clk,
   input wire reset_n // Active low reset
);
```

```
    always @(posedge clk or negedge reset_n) begin
        if (!reset_n) // Reset condition (active low)
            Q <= 1'b0;
        else
            Q <= D;
    end
endmodule
```

-
- **Implied Latches:** A common pitfall. A latch is inferred when a `reg` variable in an `always @(*)` (combinational) block is not assigned a value under *all* possible conditions (e.g., missing an `else` branch in an `if` statement, or a `default` case in a `case` statement). Latches are generally undesirable in synchronous designs as they can cause unpredictable timing behavior.
  - **How to Avoid:**
    1. **Always assign a default value** to the `reg` at the beginning of the `always @(*)` block.
    2. **Ensure all `if` statements have `else` branches.**
    3. **Ensure all `case` statements have `default` branches.**
    4. **Use `assign` statements** for simple combinational logic.

- **4.5.3 Counters:**

Up Counter (Synchronous, N-bit):

Verilog
```
module UpCounter (
    output reg [7:0] count,
    input wire clk,
    input wire reset_n, // Asynchronous active-low reset
    input wire enable
);
    always @(posedge clk or negedge reset_n) begin
        if (!reset_n) begin
            count <= 8'b0; // Reset to 0
        end else if (enable) begin
            count <= count + 1; // Increment on enable
        end
    end
endmodule
```

-

Modulo-N Counter (e.g., Modulo-10 counter, counts 0 to 9):

Verilog
```
module Modulo10Counter (
    output reg [3:0] count, // 4 bits needed for 0-9
    input wire clk,
    input wire reset_n
);
   always @(posedge clk or negedge reset_n) begin
      if (!reset_n) begin
         count <= 4'b0;
      end else if (count == 4'd9) begin // Check for max count
         count <= 4'b0; // Rollover to 0
      end else begin
         count <= count + 1; // Increment
      end
   end
endmodule
```

   - ○
- **4.5.4 Shift Registers:**

Serial In, Serial Out (SISO):

Verilog

```
module SISO_ShiftRegister (
   output reg [3:0] Q, // 4-bit register
   input wire DataIn,
   input wire clk,
   input wire reset_n
);
   always @(posedge clk or negedge reset_n) begin
      if (!reset_n) begin
         Q <= 4'b0;
      end else begin
         Q <= {Q[2:0], DataIn}; // Shift right, new bit enters MSB
      end
   end
endmodule
```

   - ○

Serial In, Parallel Out (SIPO):

Verilog

```
module SIPO_ShiftRegister (
   output wire [3:0] ParallelOut,
   input wire DataIn,
   input wire clk,
   input wire reset_n
);
   reg [3:0] shift_reg; // Internal register

   assign ParallelOut = shift_reg; // Parallel output is just the internal register

   always @(posedge clk or negedge reset_n) begin
      if (!reset_n) begin
```

```
            shift_reg <= 4'b0;
        end else begin
            shift_reg <= {shift_reg[2:0], DataIn}; // Shift right, DataIn enters MSB
        end
    end
endmodule
```

○

Module 4.6: Testbenches and Simulation

Writing Verilog for hardware is only half the battle. The other half is ensuring it works correctly. Testbenches are crucial for verifying the functional correctness of your designed hardware modules through simulation.

- **4.6.1 Purpose of Testbenches:**
  - **Verification:** The primary goal is to verify that the "Design Under Test" (DUT) behaves as expected under various input conditions.
  - **Stimulus Generation:** Apply input signals (stimuli) to the DUT over time.
  - **Output Monitoring:** Observe and capture the outputs of the DUT.
  - **Self-Checking:** Optionally compare DUT outputs against expected values to automate verification.
  - **Debugging:** Provide waveforms and messages to help debug design flaws.
  - **No Synthesis:** Testbenches are purely for simulation and are never synthesized into hardware. Therefore, they can use non-synthesizable constructs (e.g., `#` delays, `initial` blocks, file I/O).

4.6.2 Structure of a Basic Testbench: A testbench is typically a top-level Verilog module with no inputs or outputs. It instantiates the DUT and generates the necessary input waveforms.

Verilog
```
module my_design_tb; // No ports for a testbench

    // 1. Declare signals (wires/regs) for connecting to the DUT's ports
    // These signals act as the inputs/outputs to your DUT
    reg clk;
    reg reset_n;
    reg [7:0] data_in_A;
    reg [7:0] data_in_B;
    wire [7:0] gcd_result_out;
    wire done_flag;

    // 2. Instantiate the Design Under Test (DUT)
    // Connect the testbench signals to the DUT's ports
    GCD_Processor DUT (
        .A_in(data_in_A),
        .B_in(data_in_B),
        .clk(clk),
        .reset_n(reset_n),
        .result_out(gcd_result_out),
        .done_flag(done_flag)
    );

    // 3. Clock Generation (using an always block with forever)
    parameter CLK_PERIOD = 10; // 10 ns clock period (5 ns high, 5 ns low)
    initial begin
        clk = 1'b0; // Initial clock value
        forever #(CLK_PERIOD / 2) clk = ~clk; // Toggle clock every half period
    end

    // 4. Initial Block for Stimulus Application and Reset Generation
    initial begin
        // Apply reset (active low)
        reset_n = 1'b0;
        data_in_A = 8'b0;
        data_in_B = 8'b0;
        #(CLK_PERIOD * 2); // Hold reset for 2 clock cycles
        reset_n = 1'b1; // Release reset

        // Apply stimuli for GCD(48, 18) = 6
        data_in_A = 8'd48;
        data_in_B = 8'd18;
        #(CLK_PERIOD * 1); // Wait for one clock cycle for inputs to register

        // Wait for the done flag, or a maximum time
        wait(done_flag);
        $display("GCD(48, 18) = %0d, Expected: 6", gcd_result_out); // Display result

        // Apply new stimuli for GCD(100, 75) = 25
        data_in_A = 8'd100;
        data_in_B = 8'd75;
        #(CLK_PERIOD * 1); // Wait
        wait(done_flag);
        $display("GCD(100, 75) = %0d, Expected: 25", gcd_result_out);

        #(CLK_PERIOD * 5); // Allow some time for final signals to settle
        $finish; // End simulation
    end

    // 5. Monitoring Outputs (using system tasks)
    initial begin
        $display("Time\tClock\tReset\tA_in\tB_in\tGCD_Result\tDone");
        $monitor("%0t\t%b\t%b\t%0d\t%0d\t%0d\t\t%b", $time, clk, reset_n, data_in_A,
data_in_B, gcd_result_out, done_flag);
```

end

endmodule

-
- **4.6.3 System Tasks for Simulation:**
    - `$display("format_string", arg1, ...);`: Prints messages to the console during simulation. Behaves like `printf` in C.
    - `$monitor("format_string", arg1, ...);`: Prints messages to the console *whenever any of its arguments change value*. Useful for continuously tracking signals. Only one `$monitor` can be active at a time.
    - `$strobe("format_string", arg1, ...);`: Similar to `$display`, but prints values at the very end of the current simulation time step, after all values have stabilized. Useful for seeing final results for a given time step.
    - `$time`: Returns the current simulation time.
    - `$finish;`: Terminates the simulation.
    - `$stop;`: Halts the simulation, allowing interaction (e.g., examining waveforms).
    - `$dumpfile("filename.vcd");`, `$dumpvars(0, testbench_instance);`: Used to create a Value Change Dump (VCD) file for waveform viewing in a waveform viewer.

4.6.4 Self-Checking Testbenches (Briefly): More advanced testbenches include logic to automatically check if the DUT's outputs match expected values, reporting "PASS" or "FAIL". This reduces manual verification effort.

Verilog

```
// Example within initial block after getting result
if (gcd_result_out == 8'd6) begin
  $display("TEST PASSED for GCD(48, 18)");
end else begin
  $display("TEST FAILED for GCD(48, 18): Got %0d, Expected 6", gcd_result_out);
end
```

-

Module 4.7: Synthesis Concepts

This section bridges the gap between your Verilog code and actual physical hardware, explaining how a logic synthesizer translates your description into a circuit of interconnected gates.

- **4.7.1 What is Logic Synthesis?**
    - **Definition:** Logic synthesis is the automated process of translating a high-level HDL description (like Verilog) of a digital circuit into an optimized, technology-specific gate-level netlist.

    - **Netlist:** A netlist is a description of the circuit in terms of basic logic gates (AND, OR, NOT, flip-flops) and their interconnections, typically provided by a "standard cell library" for ASICs or specific "logic elements" (like Lookup Tables and Flip-flops) for FPGAs.
    - **Goals of Synthesis:**
        - **Functionality:** Ensure the synthesized netlist implements the exact logical behavior described in the HDL code.
        - **Optimization:** Minimize area (number of gates), maximize speed (meet timing constraints, reduce critical path), and minimize power consumption, based on user-defined constraints.
        - **Technology Mapping:** Map the optimized logic onto the specific gates available in the chosen target technology library.
- **4.7.2 Synthesizable vs. Non-Synthesizable Constructs** Not all Verilog constructs describe physical hardware. Some are purely for simulation and will be ignored or cause errors during synthesis.
    - **Synthesizable Constructs (Hardware Realizable):** These describe structures that can be built using physical gates and wires.
        - `module` declarations, `input`, `output`, `inout` ports.
        - `wire`, `reg` (when used appropriately to infer combinational or sequential logic).
        - `assign` statements (for combinational logic).
        - `always @(*)` for combinational logic (ensuring no implied latches).
        - `always @(posedge clk)` or `always @(negedge clk)` for sequential logic (flip-flops, registers).
        - `if-else`, `case` statements (when fully specified to avoid latches).
        - Arithmetic, logical, bitwise, relational, reduction, shift, concatenation, conditional operators.
        - `parameter` (defines constants).
        - `for` loops with **fixed, calculable bounds** (synthesizer "unrolls" them into combinational logic).
    - **Non-Synthesizable Constructs (Simulation-Only):** These describe testbench behavior, timing, or high-level abstract concepts that don't have direct hardware equivalents.
        - `initial` blocks.
        - Timing control (`#delay`).
        - System tasks (`$display`, `$monitor`, `$finish`, `$time`, file I/O).
        - `real`, `realtime` data types.
        - `force`, `release` statements (for overriding signals in simulation).
        - `forever`, `while`, `repeat` loops without fixed bounds (synthesizer cannot unroll them).
        - Recursive functions/tasks.
        - `specify` blocks (for timing specification).
- **4.7.3 Common Synthesis Issues:**
    - **Implied Latches:** As discussed in Module 4.5, if a `reg` variable in an `always @(*)` block is not assigned a value under all possible input conditions, the

synthesizer will infer a latch to hold its value. Latches can lead to race conditions and are harder to analyze for timing, making them generally undesirable in synchronous designs.

    - **Solution:** Always provide a default assignment or ensure all conditional paths are covered (`else` for `if`, `default` for `case`).

  - **Combinational Loops:** Occur when a signal's value depends on itself through a purely combinational path, creating an oscillation or undefined state.
    - **Example:** `assign A = B & C; assign C = A | D;` (creates a loop: A -> C -> A).
    - **Problem:** Such loops are unstable and problematic in hardware.
    - **Solution:** Avoid direct combinational feedback loops. If feedback is necessary, it *must* go through a sequential element (flip-flop) to break the loop and synchronize it to a clock.
  - **Over-constrained or Under-constrained Designs:** If timing constraints are too tight, synthesis might fail. If too loose, the resulting hardware might be slower than necessary.
  - **Unintended Logic Sharing:** Synthesizers try to optimize. If not careful, distinct parts of your logic might get unintentionally merged.
  - **Poorly Written RTL (Register Transfer Level) Code:** Ambiguous or inefficient Verilog code can lead to sub-optimal synthesis results (larger area, slower speed). Clear, concise, and structured RTL coding style is critical.
- **4.7.4 Mapping to Target Technology:**
  - **ASIC (Application-Specific Integrated Circuit):** Synthesizers map your design onto **standard cells** from a specific foundry's library. These are pre-designed and characterized basic gates (AND, OR, NOT), flip-flops, adders, etc., with known area, delay, and power characteristics.
  - **FPGA (Field-Programmable Gate Array):** Synthesizers map your design onto the FPGA's programmable logic blocks, typically consisting of **Look-Up Tables (LUTs)** (which can implement any Boolean function), **flip-flops**, and dedicated logic like adders or multipliers. The physical interconnections are configured electronically.