BECE407P ASIC DESIGN

Module-2:VERILOG HDL Coding Style for Synthesis

Dr. PRITAM BHATTACHARJEE

Assistant Professor (Senior Grade 2)

School of Electronics Engineering (SENSE)

Vellore Institute of Technology – Chennai

pritam.bhattacharjee@vit.ac.in, +91 8132863424

Introduction

- Verilog HDL used for digital design
- RTL design for synthesis
- Importance of coding style

Introduction

Objective

Understand and apply proper Verilog HDL coding styles to ensure your designs are:

- **Synthesis-friendly**
- **Hardware efficient**
- **Portable across tools and FPGAs/ASICs**

Why Coding Style Matters

- Ensures RTL code maps correctly to real hardware.
- Prevents simulation-synthesis mismatches.
- Improves timing closure, performance, and tool optimization.
- Avoids unintentional latches, race conditions, and resource bloat.

Key Synthesis Constructs

- ❖ Supported: `assign`, `always`, `if`, `case`
- ❖ Avoid: `initial`, `#delays`, `$display` (simulation-only)
- ❖ Data types: `wire`, `reg`, `integer`

Data Type Table

Type	Usage	Notes
wire	Continuous assign	For combinational logic
reg	Procedural assign	For storage, FF outputs
integer	Loop/indexing	Avoid for synthesis outputs

Key Synthesis Constructs - Examples

A simple code snippet that demonstrates supported constructs for synthesis—showing usage of procedural blocks, blocking/non-blocking assignments, supported data types, and why to avoid simulation-only elements.

```
module eg (input wire clk, input wire reset, input wire enable, input wire [3:0] data_in, output reg [3:0] out);

    // Supported data type for synthesis
    reg [3:0] internal_reg;

    // Combinational logic using always @(*)
    always @(*) begin
        // Blocking assignment for combinational logic
        if (enable)
            internal_reg = data_in + 1;   // Supported
        else
            internal_reg = 4'b0;          // Supported
     end

    // Sequential logic using always @(posedge clk)
    always @(posedge clk or posedge reset) begin
        if (reset)
            out <= 4'b0;         // Non-blocking assignment for sequential logic
        else
            out <= internal_reg;
     end

    // Avoid this in synthesizable code:
    // initial out = 0;  // NOT supported for synthesis
    // Avoid $display, #delay, $finish, etc.
endmodule
```

Key Synthesis Constructs - Examples

- **Procedural Blocks:**
  - always @(*) for combinational logic.
  - always @(posedge clk or posedge reset) for sequential (flip-flop) logic.
- **Assignments:**
  - Blocking (=) for combinational logic inside always @(*).
  - Non-blocking (<=) for sequential logic inside always @(posedge clk).
- **Supported Data Types:**
  - reg, wire, and vector types.
- **Simulation-Only Constructs Avoided:**
  - No initial blocks, # delays, or system tasks like $display.
- **Hierarchical Design:**
  - All inputs/outputs clearly defined at the module interface.

Combinational vs Sequential

Type	Use	Syntax
Combinational	always @(*)	Blocking (=)
Sequential	always @(posedge clk)	Non-blocking (<=)

Combinational Logic

- Use always @(*) or assign statements
- Assign every output in all branches to prevent latches

Sequential Logic (Flip-Flops)

- Use always @(posedge clk) (and/or posedge reset)
- Use non-blocking assignments (<=)

Coding Guidelines

- Use hierarchical modules
- Define all control paths
- Avoid incomplete `if/case`
- Reset all state elements

- Break large designs into modules (hierarchy).
- Define all control paths (avoid incomplete `if/case`).
- Assign default values to outputs in each process.
- Implement and connect resets for all state elements.
- Group related combinational logic for tool optimization.

Coding Guidelines - Examples

Break Large Designs into Modules (Hierarchy)

```
// Top-level module
module top_system (
    input wire clk,
    input wire reset,
    input wire start,
    input wire [7:0] data_in,
    output wire [7:0] result
);
    wire [7:0] processed_data;

    // Submodule: Data Processor
    data_processor u_data_proc (
        .clk(clk),
        .reset(reset),
        .start(start),
        .data_in(data_in),
        .data_out(processed_data)
    );

    // Submodule: Result Calculation
    result_unit u_result (
        .clk(clk),
        .reset(reset),
        .data_in(processed_data),
        .result(result)
    );
endmodule
```

Define All Control Paths (Avoid Incomplete if/case)

```
module data_processor (
    input wire clk,
    input wire reset,
    input wire start,
    input wire [7:0] data_in,
    output reg [7:0] data_out
);
    reg [1:0] state, next_state;

    localparam IDLE = 2'd0, LOAD = 2'd1, PROC = 2'd2;

    // State Register with Reset
    always @(posedge clk or posedge reset) begin
        if (reset)
            state <= IDLE;
        else
            state <= next_state;
    end

    // Next-State Logic: Complete case statement
    always @(*) begin
        next_state = state;
        case (state)
            IDLE:  if (start) next_state = LOAD;
            LOAD:             next_state = PROC;
            PROC:             next_state = IDLE;
            default:          next_state = IDLE;    // Handles unintentional states
        endcase
    end
```

Coding Guidelines - Examples

Assign Default Values to Outputs in Each Process

```
 // Output Logic: Assign default and
override as necessary
    always @(*) begin
        data_out = 8'h00; // Default value
         case (state)
            LOAD: data_out = data_in;
            PROC: data_out = data_in + 8'd5;
        endcase
    end
endmodule
```

Implement and Connect Resets for All State Elements

Reset is shown in all state registers of the examples, ensuring ALL flip-flops are reset to known conditions.

Group Related Combinational Logic for Tool Optimization

```
module result_unit (
    input wire clk,
    input wire reset,
    input wire [7:0] data_in,
    output reg [7:0] result
);
    reg [7:0] sum, diff;

    // Grouped combinational logic
    always @(*) begin
        sum  = data_in + 8'd3;
        diff = data_in - 8'd1;
    end

    // Sequential logic for outputs
    always @(posedge clk or posedge reset) begin
        if (reset)
            result <= 0;
        else
            result <= sum ^ diff; // Using grouped signals
    end
endmodule
```

Coding Guidelines - Examples

Assign Default Values to Outputs in Each Process

```
 // Output Logic: Assign default and
override as necessary
    always @(*) begin
        data_out = 8'h00; // Default value
         case (state)
            LOAD: data_out = data_in;
            PROC: data_out = data_in + 8'd5;
        endcase
    end
endmodule
```

Summary of Coding Practice Demonstrated

- **Hierarchy:** Design is split into clear modules (`top_system`, `data_processor`, `result_unit`).
- **Complete Control Paths:** All `if` and `case` statements include `else`/`default` to avoid unintended latches.
- **Default Assignments:** Outputs are assigned default values at the start of each process.
- **Resets:** Every stateful element (register/FF) uses an explicit reset.
- **Logic Grouping:** Related combinational calculations are grouped in the same process for clearer, synthesizable paths and improved optimization.

These practices ensure your designs are robust, maintainable, and synthesis-optimized.

Implement and Connect Resets for All State Elements

Reset is shown in all state registers of the examples, ensuring ALL flip-flops are reset to known conditions.

Group Related Combinational Logic for Tool Optimization

```
module result_unit (
    input wire clk,
    input wire reset,
    input wire [7:0] data_in,
    output reg [7:0] result
);
    reg [7:0] sum, diff;

    // Grouped combinational logic
    always @(*) begin
        sum  = data_in + 8'd3;
        diff = data_in - 8'd1;
    end

    // Sequential logic for outputs
    always @(posedge clk or posedge reset) begin
        if (reset)
            result <= 0;
        else
            result <= sum ^ diff; // Using grouped signals
    end
endmodule
```

State Machines

- Use case statements
- Enumerated types
- One-hot, binary encoding
- Assign defaults

State Machines - Examples

Finite State Machines (FSMs) are essential for designing control logic in digital systems. Well-coded FSMs are critical for robust, synthesizable hardware. Here’s a detailed explanation:

State Machines - Examples

Finite State Machines (FSMs) are essential for designing control logic in digital systems. Well-coded FSMs are critical for robust, synthesizable hardware. Here's a detailed explanation:

1. Use Enumerated Types or One-Hot Encoding as Required by the Synthesis Tool

- **Enumerated Types:** With Verilog-2001 and later, you can use `typedef enum` for more readable state encoding. The synthesizer chooses the optimal encoding by default (often binary), or you can specify.
- **One-Hot Encoding:** Each state has a single high bit (e.g., for four states: `4'b0001`, `4'b0010`, `4'b0100`, `4'b1000`). Some tools and FPGAs optimize for this, yielding faster, simpler logic.
- **Example:** Enumerated States (Preferred Practice)

```
typedef enum logic [1:0] {
    IDLE  = 2'b00,
    LOAD  = 2'b01,
    PROC  = 2'b10,
    DONE  = 2'b11
} state_t;

state_t state, next_state;
```

Example: One-Hot Encoding (Tool-Specific, e.g., for FPGAs)

```
parameter IDLE = 4'b0001, LOAD = 4'b0010,
PROC = 4'b0100, DONE = 4'b1000;
reg [3:0] state, next_state;
```

State Machines - Examples

2. Assign All Outputs in Every State (Avoid Latches)

- Outputs should be assigned a definite value in every state—otherwise, synthesis infers a latch (undesirable).
- Always initialize or set default output values at the start of the combinational process, then override in specific states.

Example: Output Assignment in FSM (No Latches)

```
always @(*) begin
    out_signal = 1'b0;      // Default
    data_out   = 8'h00;     // Default

    case (state)
        IDLE: begin
            out_signal = 1'b0;
        end
        LOAD: begin
            data_out = data_in;
        end
        PROC: begin
            data_out = data_in + 8'd1;
        end
        DONE: begin
            out_signal = 1'b1;
        end
        default: begin
            // Covers unintentional state
            out_signal = 1'b0;
            data_out   = 8'h00;
        end
    endcase
end
```

Every output is set regardless of the current state.

State Machines - Examples

2. Assign All Outputs in Every State (Avoid Latches)

- Outputs should be assigned a definite value in every state—otherwise, synthesis infers a latch (undesirable).
- Always initialize or set default output values at the start of the combinational process, then override in specific states.

Example: Output Assignment in FSM (No Latches)

```
always @(*) begin
    out_signal = 1'b0;      // Default
    data_out   = 8'h00;     // Default

    case (state)
        IDLE: begin
            out_signal = 1'b0;
        end
        LOAD: begin
            data_out = data_in;
        end
        PROC: begin
            data_out = data_in + 8'd1;
        end
        DONE: begin
            out_signal = 1'b1;
        end
        default: begin
            // Covers unintentional state
            out_signal = 1'b0;
            data_out   = 8'h00;
        end
    endcase
end
```

Every output is set regardless of the current state.

3. Implement Clearly Defined Reset States (Synchronous or Asynchronous)

- **Synchronous Reset:** State register reset synchronously with the clock.
- **Asynchronous Reset:** State register reset immediately, triggered by the reset signal.

Example: Synchronous State Register with Reset

```
always @(posedge clk) begin
    if (reset)
        state <= IDLE;    // Defined reset state
    else
        state <= next_state;
end
```

State Machines - Examples

2. Assign All Outputs in Every State (Avoid Latches)

- Outputs should be assigned a definite value in every state—otherwise, synthesis infers a latch (undesirable).
- Always initialize or set default output values at the start of the combinational process, then override in specific states.

Example: Output Assignment in FSM (No Latches)

```
always @(*) begin
    out_signal = 1'b0;     // Default
    data_out   = 8'h00;    // Default

    case (state)
        IDLE: begin
            out_signal = 1'b0;
        end
        LOAD: begin
            data_out = data_in;
        end
        PROC: begin
            data_out = data_in + 8'd1;
        end
        DONE: begin
            out_signal = 1'b1;
        end
        default: begin
            // Covers unintentional state
            out_signal = 1'b0;
            data_out   = 8'h00;
        end
    endcase
end
```

Every output is set regardless of the current state.

3. Implement Clearly Defined Reset States (Synchronous or Asynchronous)

- Synchronous Reset: State register reset synchronously with the clock.
- Asynchronous Reset: State register reset immediately, triggered by the reset signal.

Example: Synchronous State Register with Reset

```
always @(posedge clk) begin
    if (reset)
        state <= IDLE;    // Defined reset state
    else
        state <= next_state;
end
```

Example: Asynchronous Reset

```
always @(posedge clk or posedge reset) begin
    if (reset)
        state <= IDLE;
    else
        state <= next_state;
end
```

State Machines - Examples

2. Assign All Outputs in Every State (Avoid Latches)

- Outputs should be assigned a definite value in every state—otherwise, synthesis infers a latch (undesirable).
- Always initialize or set default output values at the start of the combinational process, then override in specific states.

Example: Output Assignment in FSM (No Latches)

```
always @(*) begin
    out_signal = 1'b0;      // Default
    data_out   = 8'h00;     // Default

    case (state)
        IDLE: begin
            out_signal = 1'b0;
        end
        LOAD: begin
            data_out = data_in;
        end
        PROC: begin
            data_out = data_in + 8'd1;
        end
        DONE: begin
            out_signal = 1'b1;
        end
        default: begin
            // Covers unintentional state
            out_signal = 1'b0;
            data_out   = 8'h00;
        end
    endcase
end
```

Every output is set regardless of the current state.

Best Practice: Explicitly define the reset state to ensure the FSM starts in a known state after reset.

3. Implement Clearly Defined Reset States (Synchronous or Asynchronous)

- **Synchronous Reset:** State register reset synchronously with the clock.
- **Asynchronous Reset:** State register reset immediately, triggered by the reset signal.

Example: Synchronous State Register with Reset

```
always @(posedge clk) begin
    if (reset)
        state <= IDLE;    // Defined reset state
    else
        state <= next_state;
end
```

Example: Asynchronous Reset

```
always @(posedge clk or posedge reset) begin
    if (reset)
        state <= IDLE;
    else
        state <= next_state;
end
```

Complete FSM Example incorporating these Guidelines

```
typedef enum logic [1:0] {IDLE = 2'b00, LOAD = 2'b01, PROC = 2'b10, DONE = 2'b11}
fsm_state_t;

module fsm_example (input wire clk, input wire reset, input wire start, input wire
[7:0] data_in, output reg [7:0] data_out, output reg done);
    fsm_state_t state, next_state;

    // State register: synchronous reset
    always @(posedge clk) begin
        if (reset)
            state <= IDLE;
        else
            state <= next_state;
    end

     // Next-state logic
    always @(*) begin
        next_state = state;
        case (state)
            IDLE:  next_state = (start) ? LOAD : IDLE;
            LOAD:  next_state = PROC;
            PROC:  next_state = DONE;
            DONE:  next_state = IDLE;
            default: next_state = IDLE; // Robustness
        endcase
    end

     // Output logic: assign all outputs in each state
    always @(*) begin
        data_out = 8'h00; // default
        done     = 1'b0;  // default

         case (state)
            IDLE: ;
            LOAD: data_out = data_in;
            PROC: data_out = data_in + 8'd4;
            DONE: done     = 1'b1;
        endcase
    end
endmodule
```

Summary of Key Points

- **Enumerate or clearly encode states**—use `typedef enum` or one-hot as needed.
- **Assign outputs for every state**—never leave outputs unassigned.
- **Define resets explicitly**—FSM must enter a known state after reset.

Adhering to these principles ensures your FSMs are robust, synthesis-friendly, and behave as intended on hardware.

Synthesizable Verilog code and Testbench for "overlapping sequence detection of 0010 using state machine"

overlapping sequence detector that detects the binary pattern 0010 using a Mealy machine.

- •S0: Initial state (no bits detected yet)
- •S1: Detected '0'
- •S2: Detected '00'
- •S3: Detected '001'

Current State	Input	Next State	Output (Z)
S0	0	S1	0
S0	1	S0	0
S1	0	S2	0
S1	1	S0	0
S2	0	S2	0
S2	1	S3	0
S3	0	S1	1
S3	1	S0	0

Verilog code

```
module seq_det_0010 (
    input wire clk,
    input wire rst_n,
    input wire x,          // serial input bit stream
    output reg z           // output
);

    // State Encoding
 parameter S0 = 2'd0, S1 = 2'd1, S2 = 2’d2, S3 = 2’d3;
 reg [1:0] state, next_state;

    // Sequential logic: state update
    always @(posedge clk or negedge rst_n) begin
        if (!rst_n)
            state <= S0;
        else
            state <= next_state;
    end

// Combinational logic: next state and output logic
    always @(*) begin
        next_state = state;
        z = 0;
        case (state)
            S0: begin
                if (x == 0)
                    next_state = S1;
                else
                    next_state = S0;
                z = 0;
            end
            S1: begin
                if (x == 0)
                    next_state = S2;
                else
                    next_state = S0;
                z = 0;
            end
            S2: begin
                if (x == 0)
                    next_state = S2;
                else
                    next_state = S3;
                z = 0;
            end
            S3: begin
                if (x == 0) begin
                    next_state = S1;
                    z = 1;  // Sequence 0010 detected
                end else begin
                    next_state = S0;
                    z = 0;
                end
            end
        endcase
    end

endmodule
```

Synthesizable Verilog code and Testbench for "overlapping sequence detection of 0010 using state machine"

Testbench

```
`timescale 1ns/1ps
module tb_seq_det_0010();

    reg clk, rst_n, x;
    wire z;

    seq_det_0010 DUT(.clk(clk), .rst_n(rst_n), .x(x), .z(z));

    // Clock generation
    initial clk = 0;
    always #5 clk = ~clk;

    // Stimulus
    initial begin
        rst_n = 0;
        x = 0;
        #12 rst_n = 1;

        // Test sequence: 0 0 1 0 0 0 1 0 (should detect "0010" twice, with overlap)
        #10 x = 0;
        #10 x = 0;
        #10 x = 1;
        #10 x = 0;  // "0010" detected, z = 1
        #10 x = 0;
        #10 x = 0;
        #10 x = 1;
        #10 x = 0;  // "0010" detected again, with overlap
        #40;        // End
        $finish;
    end
```

Clock & Reset Design

- Prefer **single clock domain** per block.
- Use **synchronous reset** where possible.
- Register **all block outputs** for easy timing analysis.
- Synchronize external asynchronous resets to the clock.

Clock & Reset Design

Example: Synthesis-Ready Verilog Code for Clock and Reset

```
module reg_block (
    input  wire clk,           // Clock input
    input  wire reset,         // Synchronous reset
    input  wire enable,
    input  wire [7:0] d_in,
    output reg  [7:0] q_out
);
    always @(posedge clk) begin
        if (reset)            // Synchronous reset active
            q_out <= 8'b0;    // Initialize to a known value
        else if (enable)
            q_out <= d_in;    // Register update on clock edge
    end
endmodule
```

Clock & Reset Design

Example: Synthesis-Ready Verilog Code for Clock and Reset

```
module reg_block (
    input  wire clk,           // Clock input
    input  wire reset,         // Synchronous reset
    input  wire enable,
    input  wire [7:0] d_in,
    output reg  [7:0] q_out
);
    always @(posedge clk) begin
        if (reset)            // Synchronous reset active
            q_out <= 8'b0;    // Initialize to a known value
        else if (enable)
            q_out <= d_in;    // Register update on clock edge
    end
endmodule
```

Key Points in Clock & Reset Coding for Synthesis

- **Single Clock Domain**:
  - Use **one clock per block** or module whenever possible.
  - Avoid unnecessary complexity and clock domain crossings in a unit.
- **Synchronous Reset (Preferred)**:
  - The reset logic updates state only on a positive clock edge.
  - Synchronous resets are more predictable and easier to constrain/timing optimize in ASIC/FPGA flow.
  - Example here uses `if (reset)` within an `always @(posedge clk)` block.

Clock & Reset Design

Example: Synthesis-Ready Verilog Code for Clock and Reset

```
module reg_block (
    input  wire clk,           // Clock input
    input  wire reset,         // Synchronous reset
    input  wire enable,
    input  wire [7:0] d_in,
    output reg  [7:0] q_out
);
    always @(posedge clk) begin
        if (reset)            // Synchronous reset active
            q_out <= 8'b0;    // Initialize to a known value
        else if (enable)
            q_out <= d_in;    // Register update on clock edge
    end
endmodule
```

Asynchronous Reset (If Required, Use Carefully):

- If hardware constraints need asynchronous reset, write:

```
 always @(posedge clk or posedge reset) begin
     if (reset)//Asynchronously sets value immediately
         q_out <= 8'b0;
     else if (enable)
         q_out <= d_in;
     end
```

- Asynchronous resets are instantly responsive but can introduce metastability if not properly synchronized for de-assertion.

Clock & Reset Design

Example: Synthesis-Ready Verilog Code for Clock and Reset

```
module reg_block (
    input  wire clk,           // Clock input
    input  wire reset,         // Synchronous reset
    input  wire enable,
    input  wire [7:0] d_in,
    output reg  [7:0] q_out
);
    always @(posedge clk) begin
        if (reset)           // Synchronous reset active
            q_out <= 8'b0;   // Initialize to a known value
        else if (enable)
            q_out <= d_in;   // Register update on clock edge
    end
endmodule
```

- **Register Block Outputs**:
  - All output signals (especially across module boundaries) should be **registered** to help with static timing analysis and placement/routing.
- **No Clock Gating Logic in RTL**:
  - Avoid gating clock signals with combinational logic or enable signals.
  - Instead, use enable signals inside sequential blocks, as shown, for power/cell optimization. Clock gating will be implemented automatically in the synthesis/physical tools when requested.
- **All FFs Initialized**:
  - Explicit reset path ensures all registers begin in a known state, crucial for reliable hardware bring-up and functional simulation.

Memory Block Coding

- Inferred RAM/ROM
- Synchronous read/write
- Use arrays (`reg [n:0] mem [m:0];`)

Memory Block Coding

Synthesizable memory coding allows the synthesis tools to infer hardware RAM/ROM from your Verilog RTL. Correct memory coding style ensures proper mapping to physical resources (block RAM, LUT RAM, registers) in ASICs and FPGAs.

Memory Block Coding

Synthesizable memory coding allows the synthesis tools to infer hardware RAM/ROM from your Verilog RTL. Correct memory coding style ensures proper mapping to physical resources (block RAM, LUT RAM, registers) in ASICs and FPGAs.

1. Single-Port RAM (Synchronous Write, Asynchronous Read)

```
module single_port_ram (
    input  wire clk,
    input  wire we,
    input  wire [3:0] addr,
    input  wire [7:0] din,
    output reg  [7:0] dout
);
    reg [7:0] mem [0:15]; // 16 × 8-bit memory

    // Synchronous write, asynchronous read
    always @(posedge clk) begin
        if (we)
            mem[addr] <= din;
    end

    always @(*) begin
        dout = mem[addr]; // Asynchronous read
    end
endmodule
```

- Declares a 16×8 memory using `reg [7:0] mem[0:15]`.
- Write happens on the clock's rising edge (synchronous).
- Read is combinational (asynchronous).
- Many FPGAs infers distributed RAM with this template. Some ASIC flows require synchronous read.

Memory Block Coding

Synthesizable memory coding allows the synthesis tools to infer hardware RAM/ROM from your Verilog RTL. Correct memory coding style ensures proper mapping to physical resources (block RAM, LUT RAM, registers) in ASICs and FPGAs.

2. Single-Port RAM with Synchronous Read (Fully Synchronous FF/Block RAM)

```
module sync_ram (
    input  wire clk,
    input  wire we,
    input  wire [3:0] addr,
    input  wire [7:0] din,
    output reg  [7:0] dout
);
    reg [7:0] mem [0:15];

    always @(posedge clk) begin
        if (we)
            mem[addr] <= din; // Synchronous write
        dout <= mem[addr];    // Synchronous read
    end
endmodule
```

- Both read and write occur on clock edge.
- The output holds the data present at the addressed location after the clock edge.
- Most FPGAs and ASICs infer true dual-port/block RAM with this style.

Memory Block Coding

Synthesizable memory coding allows the synthesis tools to infer hardware RAM/ROM from your Verilog RTL. Correct memory coding style ensures proper mapping to physical resources (block RAM, LUT RAM, registers) in ASICs and FPGAs.

3. Dual-Port RAM (Two Independent Ports)

```
module dual_port_ram (
    input  wire clk,
    // Port A signals
    input  wire we_a,
    input  wire [3:0] addr_a,
    input  wire [7:0] din_a,
    output reg  [7:0] dout_a,
    // Port B signals
    input  wire we_b,
    input  wire [3:0] addr_b,
    input  wire [7:0] din_b,
    output reg  [7:0] dout_b
);
    reg [7:0] mem [0:15];

    always @(posedge clk) begin
        // Port A
        if (we_a)
            mem[addr_a] <= din_a;
        dout_a <= mem[addr_a];
        // Port B
        if (we_b)
            mem[addr_b] <= din_b;
        dout_b <= mem[addr_b];
    end
endmodule
```

- Both ports can read/write independently at different addresses.
- FPGAs map this to hardware block RAM with two ports.
- Always avoid simultaneous write to same address from both ports—behavior is undefined.

Memory Block Coding

Synthesizable memory coding allows the synthesis tools to infer hardware RAM/ROM from your Verilog RTL. Correct memory coding style ensures proper mapping to physical resources (block RAM, LUT RAM, registers) in ASICs and FPGAs.

3. Dual-Port RAM (Two Independent Ports)

```
module dual_port_ram (
    input  wire clk,
    // Port A signals
    input  wire we_a,
    input  wire [3:0] addr_a,
    input  wire [7:0] din_a,
    output reg  [7:0] dout_a,
    // Port B signals
    input  wire we_b,
    input  wire [3:0] addr_b,
    input  wire [7:0] din_b,
    output reg  [7:0] dout_b
);
    reg [7:0] mem [0:15];

    always @(posedge clk) begin
        // Port A
        if (we_a)
            mem[addr_a] <= din_a;
        dout_a <= mem[addr_a];
        // Port B
        if (we_b)
            mem[addr_b] <= din_b;
        dout_b <= mem[addr_b];
    end
endmodule
```

- **No Dedicated Dual-Port RAM Hardware:** ASIC standard cell libraries typically **do not include configurable block RAMs** like FPGAs. Instead, memories must be constructed from either:
  - **Custom memory macros generated by memory compilers** (offered by foundries or EDA vendors),
  - Or, if small, **arrays of flip-flops/registers** synthesized from standard-cell D flip-flops.
- **Memory Compiler Macros:**
  - Dual-port or multi-port SRAMs are often instantiated as **memory hard macros**. These are pre-designed and characterized blocks that are not synthesized from behavioural code, but instead are *instantiated* as black-box macros with specific library views (.lib, .lef, etc.).
  - When writing portable Verilog, you use a generic "coding style" for a dual-port RAM (matching the macro's behaviour), and during synthesis, you either:
    - Replace/infer ("black-box replace") that code with the foundry's optimized dual-port memory,
    - Or guide the synthesis tool (via constraints/pragmas or scripts) to use a specific macro matching your interface.

Memory Block Coding

Synthesizable memory coding allows the synthesis tools to infer hardware RAM/ROM from your Verilog RTL. Correct memory coding style ensures proper mapping to physical resources (block RAM, LUT RAM, registers) in ASICs and FPGAs.

3. Dual-Port RAM (Two Independent Ports)

```
module dual_port_ram (
    input  wire clk,
    // Port A signals
    input  wire we_a,
    input  wire [3:0] addr_a,
    input  wire [7:0] din_a,
    output reg  [7:0] dout_a,
    // Port B signals
    input  wire we_b,
    input  wire [3:0] addr_b,
    input  wire [7:0] din_b,
    output reg  [7:0] dout_b
);
    reg [7:0] mem [0:15];

    always @(posedge clk) begin
        // Port A
        if (we_a)
            mem[addr_a] <= din_a;
        dout_a <= mem[addr_a];
        // Port B
        if (we_b)
            mem[addr_b] <= din_b;
        dout_b <= mem[addr_b];
    end
endmodule
```

- **Small DPRAMs with Flops:**
  - If your memory is small and a macro is not used/inferred, the synthesis tool will **build the RAM using arrays of registers/flip-flops**. This is inefficient for anything above a few words/depth, but functionally correct for ASIC.
  - This means that a dual-port RAM (with two independent read/write ports) would require duplicating underlying registers such that every collision/priority/clocking rule is satisfied with combinatorial and sequential logic — an approach that is area and power expensive.

Memory Block Coding

Synthesizable memory coding allows the synthesis tools to infer hardware RAM/ROM from your Verilog RTL. Correct memory coding style ensures proper mapping to physical resources (block RAM, LUT RAM, registers) in ASICs and FPGAs.

3. Dual-Port RAM (Two Independent Ports)

```
module dual_port_ram (
    input  wire clk,
    // Port A signals
    input  wire we_a,
    input  wire [3:0] addr_a,
    input  wire [7:0] din_a,
    output reg  [7:0] dout_a,
    // Port B signals
    input  wire we_b,
    input  wire [3:0] addr_b,
    input  wire [7:0] din_b,
    output reg  [7:0] dout_b
);
    reg [7:0] mem [0:15];

    always @(posedge clk) begin
        // Port A
        if (we_a)
            mem[addr_a] <= din_a;
        dout_a <= mem[addr_a];
        // Port B
        if (we_b)
            mem[addr_b] <= din_b;
        dout_b <= mem[addr_b];
    end
endmodule
```

Key Points

- **Performance and Area:**
  - ASIC dual-port macros are highly optimized for area and timing.
  - If not mapped to a macro, synthesized RAMs will be much larger and slower than FPGA block RAMs.
- **Design Flow:**
  - Synopsys Design Compiler, Cadence Genus, or similar tools allow you to specify which memories should be "black-boxed" and then replaced with foundry-specific macros during Place & Route.
- **Verification:**
  - Since macros are black boxes, their functionality is normally simulated using behavioural models (Verilog) that match the macro's interface.

Memory Block Coding

Synthesizable memory coding allows the synthesis tools to infer hardware RAM/ROM from your Verilog RTL. Correct memory coding style ensures proper mapping to physical resources (block RAM, LUT RAM, registers) in ASICs and FPGAs.

3. Dual-Port RAM (Two Independent Ports)

```
module dual_port_ram (
    input  wire clk,
    // Port A signals
    input  wire we_a,
    input  wire [3:0] addr_a,
    input  wire [7:0] din_a,
    output reg  [7:0] dout_a,
    // Port B signals
    input  wire we_b,
    input  wire [3:0] addr_b,
    input  wire [7:0] din_b,
    output reg  [7:0] dout_b
);
    reg [7:0] mem [0:15];

    always @(posedge clk) begin
        // Port A
        if (we_a)
            mem[addr_a] <= din_a;
        dout_a <= mem[addr_a];
        // Port B
        if (we_b)
            mem[addr_b] <= din_b;
        dout_b <= mem[addr_b];
    end
endmodule
```

Summary Table

Platform	Behavioural Verilog →	Synthesis Output	Performance/Area
FPGA	Dual-port RAM code	Block RAM (built-in)	Efficient, small, fast
ASIC	Dual-port RAM code	Macro or registers	Macro = best; FFs = area/speed penalty

In ASICs, synthesis of dual-port RAM typically maps to a specialized memory macro if available and specified. Otherwise, the tool will resort to using arrays of registers, which is functional but area-inefficient. Proper dual-port SRAMs are almost always instantiated as hard macros in final ASIC designs, not purely synthesized from RTL.

One must work with the ASIC vendor to obtain the correct macro and match the code interface to the macro's definition for efficient implementation.

Memory Block Coding

Synthesizable memory coding allows the synthesis tools to infer hardware RAM/ROM from your Verilog RTL. Correct memory coding style ensures proper mapping to physical resources (block RAM, LUT RAM, registers) in ASICs and FPGAs.

4. ROM (Read-Only Memory)

```
module rom (
    input  wire [2:0] addr,
    output reg  [7:0] dout
);
    always @(*) begin
        case (addr)
            3'd0: dout = 8'hA5;
            3'd1: dout = 8'h1C;
            3'd2: dout = 8'hBE;
            3'd3: dout = 8'hEF;
            // ... (initialize as needed)
            default: dout = 8'h00;
        endcase
    end
endmodule
```

- ROM contents are "hardcoded" using a case statement.
- Can also use `initial begin ...` for simulation or synthesis with inferred ROMs in some tools.

```
module rom_init (
    input  wire [3:0] addr,
    output reg  [7:0] data);
    reg [7:0] rom [0:15]; // 16 × 8 ROM

    // Initialize ROM contents
    initial begin
        rom[0]  = 8'hDE;
        rom[1]  = 8'hAD;
        rom[2]  = 8'hBE;
        rom[3]  = 8'hEF;
        rom[4]  = 8'h12;
        rom[5]  = 8'h34;
        rom[6]  = 8'h56;
        rom[7]  = 8'h78;
        rom[8]  = 8'h9A;
        rom[9]  = 8'hBC;
        rom[10] = 8'hCD;
        rom[11] = 8'hEF;
        rom[12] = 8'h00;
        rom[13] = 8'h11;
        rom[14] = 8'h22;
        rom[15] = 8'h33;
    end

    always @(*) begin
        data = rom[addr];
    end
endmodule
```

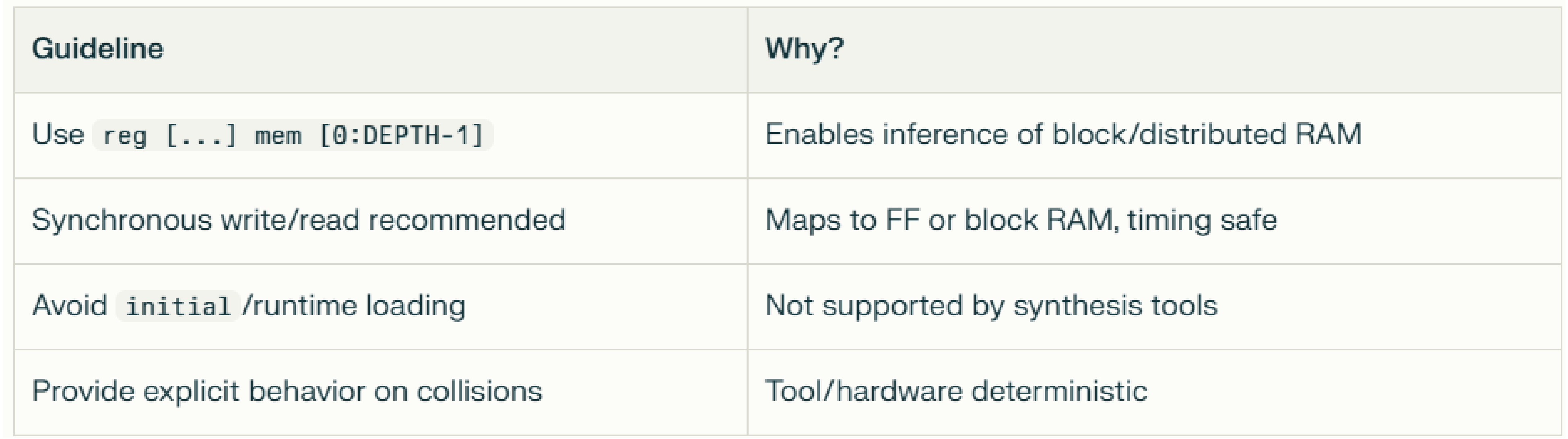
- The `initial begin ... end` block loads constant values into the ROM array at elaboration time, which is fully supported for simulation.
- Some ASIC flows can also infer a ROM from this construct, mapping it to embedded ROM resources. Some synthesis tools might require a specific setting to preserve initial values.
- Not all ASIC tools support this for synthesis—always check your tool documentation and target technology capabilities.

Memory Block Coding

Synthesizable memory coding allows the synthesis tools to infer hardware RAM/ROM from your Verilog RTL. Correct memory coding style ensures proper mapping to physical resources (block RAM, LUT RAM, registers) in ASICs and FPGAs.

Best Practice Guidelines

- **Module Memories, Use Arrays:** Use reg `[WIDTH-1:0] mem [0:DEPTH-1]` for RAM/ROM.
- **Synchronous Write, Synchronous/Asynchronous Read:** Synchronous clocking ensures reliable synthesis and timing closure.
- **No initial or dynamic allocation at runtime:** Use static definitions and initialization for synthesizability.
- **Explicit Behaviour for Write/Read-Write:** Always define what happens on simultaneous read/write to the same address—if not defined, results differ across vendors.
- **Arrays are Not Synthesizable at the Top Module Port:** Keep memory arrays internal to modules, not directly exposed at I/O.



Guideline	Why?
Use `reg [...] mem [0:DEPTH-1]`	Enables inference of block/distributed RAM
Synchronous write/read recommended	Maps to FF or block RAM, timing safe
Avoid `initial` /runtime loading	Not supported by synthesis tools
Provide explicit behavior on collisions	Tool/hardware deterministic

Tri-state & I/O Ports

- Restrict tri-state logic to top-level
- Conditional assignments only where essential
- Avoid internally

Best Practice Summary

Practice	**Reason**
Register I/O	Timing closure
Non-blocking for seq. logic	Synthesizer-friendly
Avoid unwanted latches	Stability
Use @(*)for comb. logic	Accurate sensitivity