

Example: Assign and Named Mapping

```
// Verilog code for 2-input multiplexer
module INV (input A, output F); // An inverter
    assign F = ~A;
endmodule

module AOI (input A, B, C, D, output F);
    assign F = ~((A & B) | (C & D));
endmodule

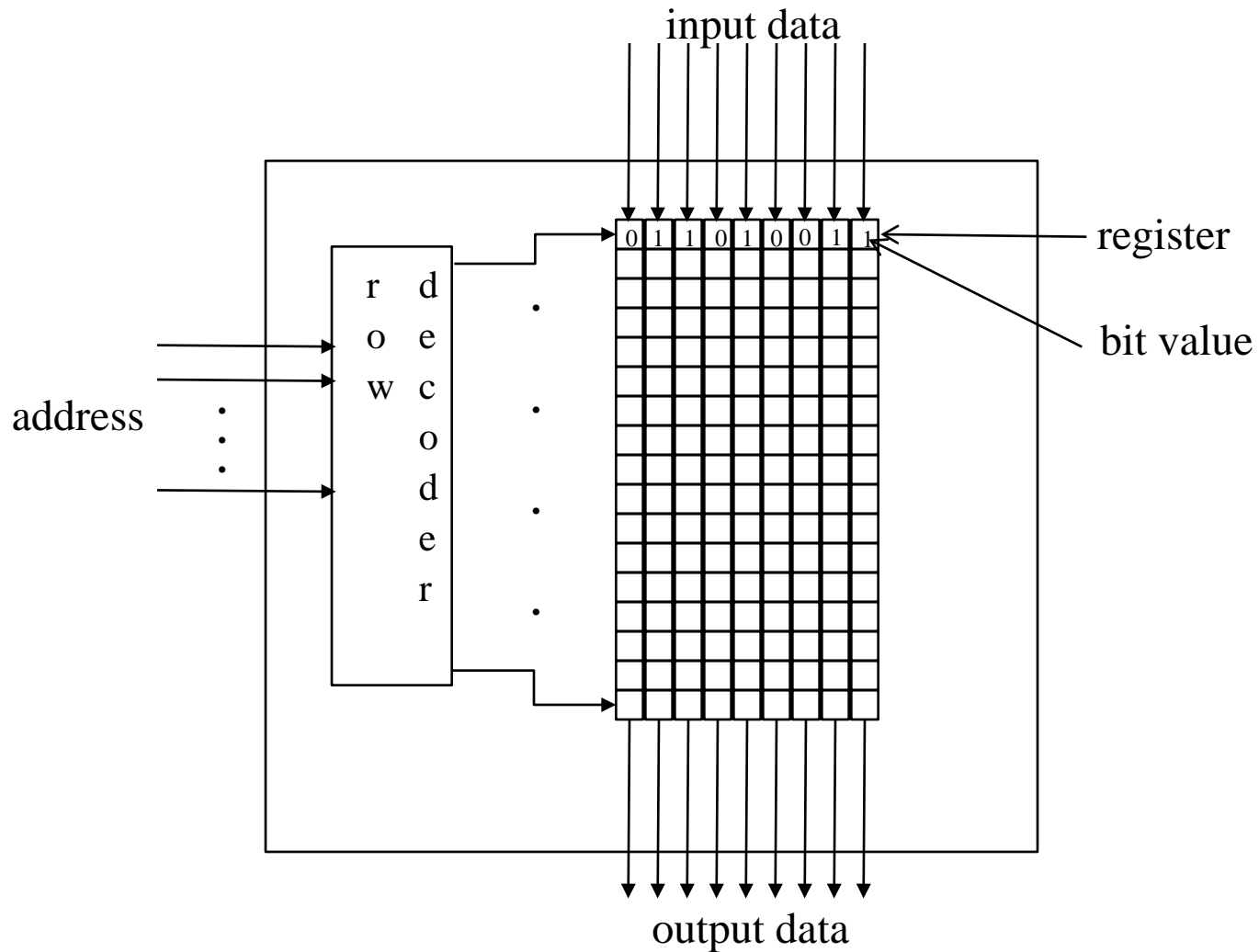
module MUX2 (input SEL, A, B, output F); // 2:1 multiplexer
    // wires SELB and FB are implicit
    // Module instances...
    INV G1 (SEL, SELB);
    AOI G2 (SELB, A, SEL, B, FB);
    INV G3 (.A(FB), .F(F));          // Named mapping
endmodule
```



Register Files

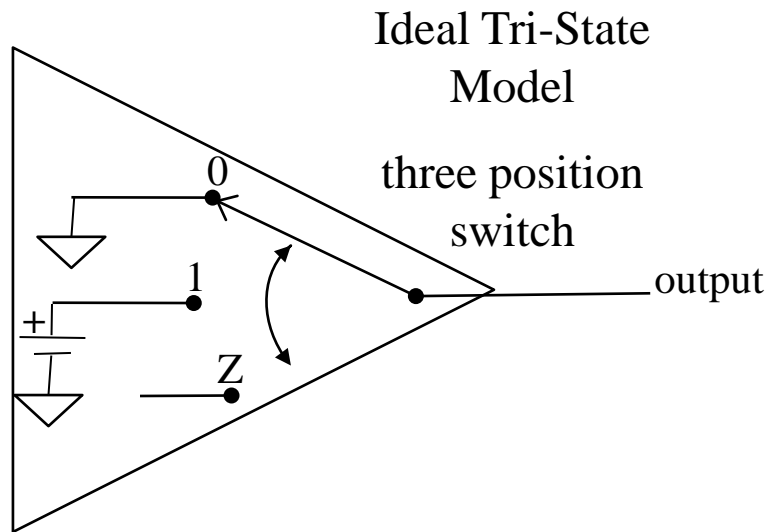
An Introduction to Dynamic Memory Storage

Data Storage – Conceptual Layout

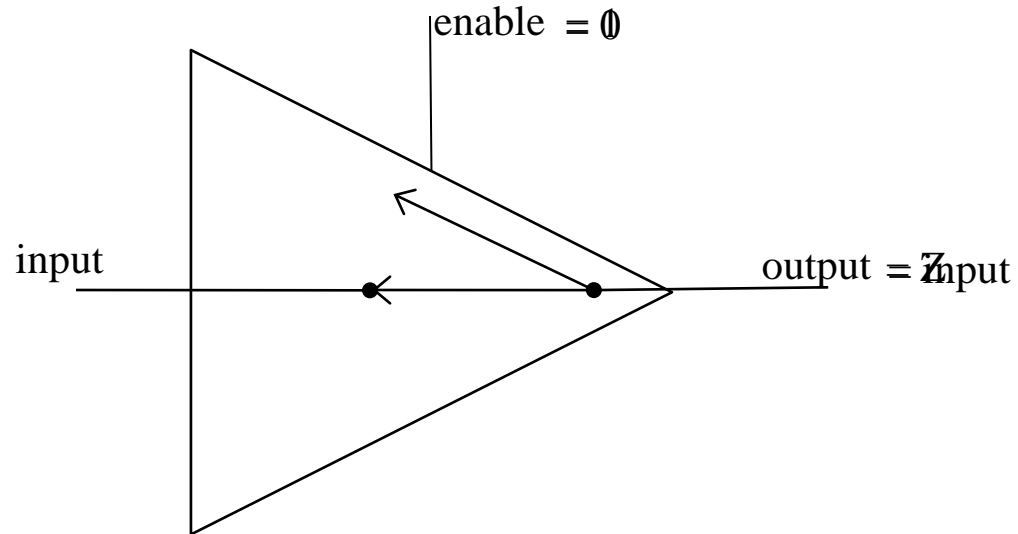


Tri-State Output

- Tri-State output has three possible outputs:
 - 0, 1, Z
- Z is referred to as high-impedance
- Z output allows for multiple outputs to be connected.

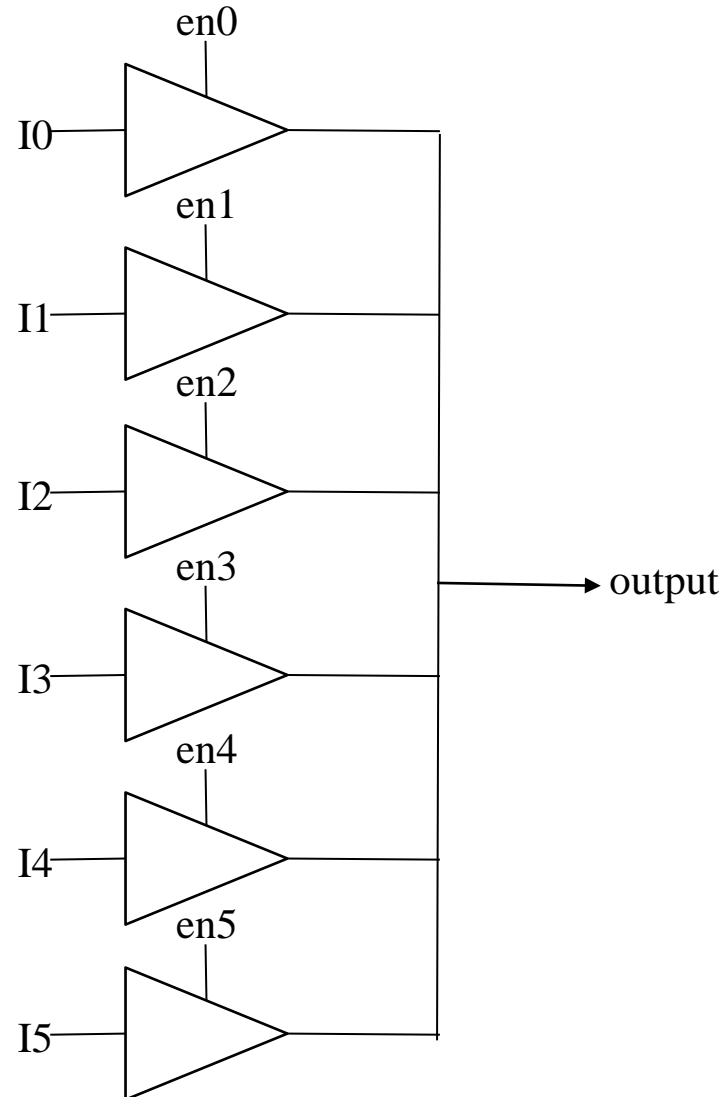


Tri-State Buffer Model



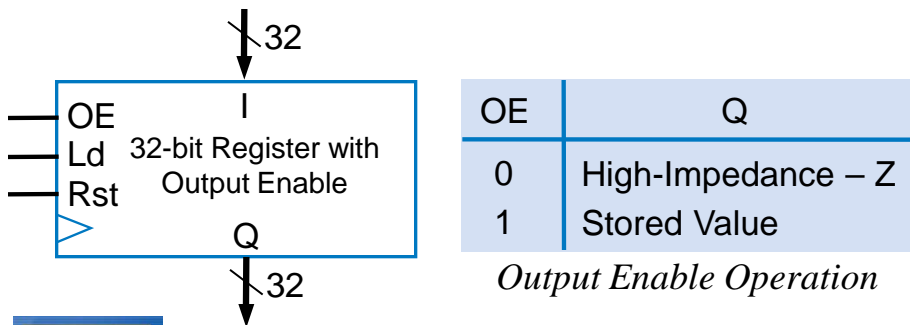
Multiple Tri-State Buffers Tied to One Net

only one enable
should be asserted



32-Bit Register With Output Enable

- *Output procedure*
 - Combinational procedure that controls register output
 - $Oe = 1 \rightarrow$ Output is enabled
 - $Q \leq R;$
 - $Oe = 0 \rightarrow$ Output of register is disabled
 - Output high-impedance
 - $Q \leq 32'hZZZZZZZZ;$



```

`timescale 1 ns/1 ns

module Reg32wOE(I, Q, Oe, Ld, Clk, Rst);

    input [31:0] I;
    output [31:0] Q;
    reg [31:0] Q;
    input Oe, Ld;
    input Clk, Rst;

    reg [31:0] R;

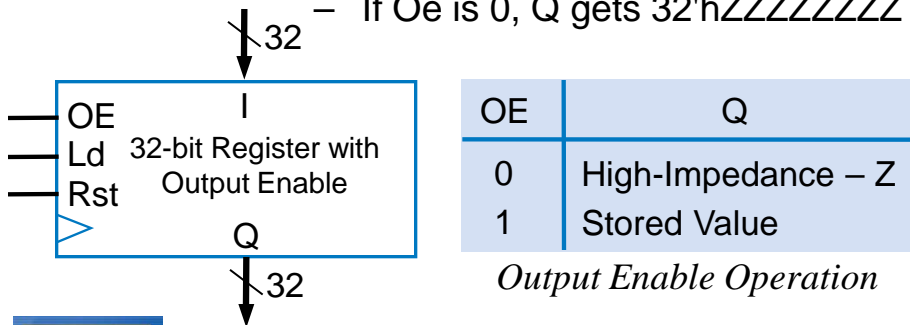
    // Register Procedure
    always @(posedge Clk) begin
        if (Rst == 1)
            R <= 32'd0;
        else if (Ld == 1)
            R <= I;
    end

    // Output Procedure
    always @(R, Oe) begin
        if (Oe == 1)
            Q <= R;
        else
            Q <= 32'hZZZZZZZZ;
    end
endmodule
  
```



32-Bit Register With Output Enable

- Alternative description
 - Replace Output procedure by a single continuous assignment statement (**assign**)
 - Q must be net, not variable
 - Uses **conditional operator** **?** :
 - $A ? B : C$
 - If A is true (non-zero), result is B
 - If A is false (zero), result is C
 - $Q = Oe ? R : 32'hZZZZZZZZ;$
 - If Oe is 1, Q gets R
 - If Oe is 0, Q gets 32'hZZZZZZZZ



```
`timescale 1 ns/1 ns

module Reg32wOE(I, Q, Oe, Ld, Clk, Rst);

    input [31:0] I;
    output [31:0] Q;
    input Oe, Ld;
    input Clk, Rst;

    reg [31:0] R;

    // Register Procedure
    always @(posedge Clk) begin
        if (Rst == 1)
            R <= 32'd0;
        else if (Ld == 1)
            R <= I;
        end

    assign Q = Oe ? R : 32'hZZZZZZZZ;
endmodule
```

Same behavior as previous description, just more compact



32-Bit Register With Output Enable

- Testbench

- Reset register and enable output →
Oe_s <= 1;
- Load register with value
32'h0000000FF
 - Use self-check to verify correctness
- New operator use
 - !=
 - Does bit-by-bit comparison
 - Handles z and x values
 - ==
 - For bit-by-bit equality check
 - Handles z and x values
 - == and != don't handle z or x
 - Returns x (unknown) if z or x present in either operand
 - != and == never return x
- Oe_s <= 0; → Disable output
 - Use self-check to verify that output is high-impedance

```
...
// Vector Procedure
initial begin
    Rst_s <= 1;
    Oe_s <= 1; Ld_s <= 0;
    I_s <= 32'h00000000;
    @(posedge Clk_s);
    #5 Rst_s <= 0;
    @(posedge Clk_s);
    #5 Ld_s <= 1; I_s <= 32'h000000FF;
    @(posedge Clk_s);
    #5;
    if (Q_s != 32'h000000FF)
        $display("Failed output enabled");
    Ld_s <= 0; Oe_s <= 0;
    #5;
    if (Q_s != 32'hZZZZZZZZ)
        $display("Failed output disabled");
end
...
```



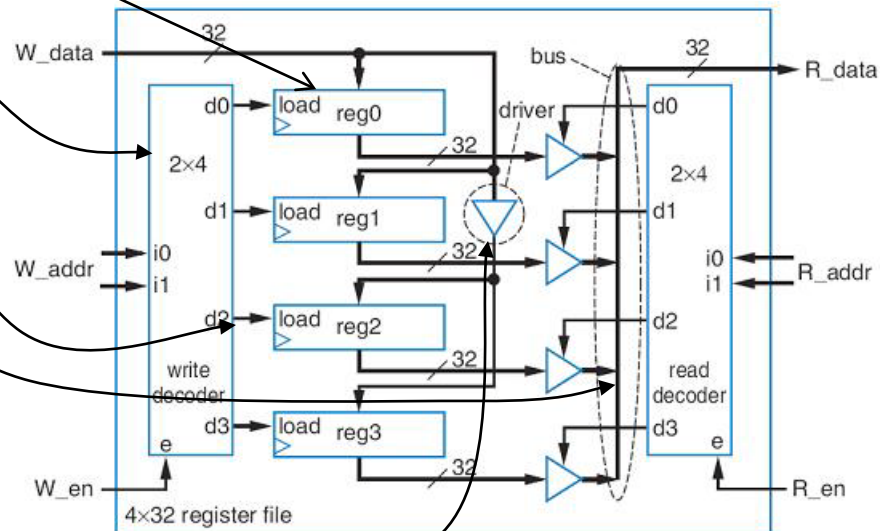
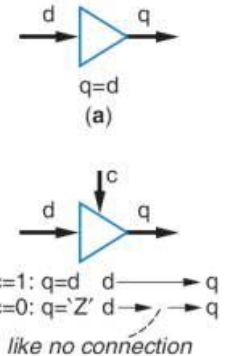
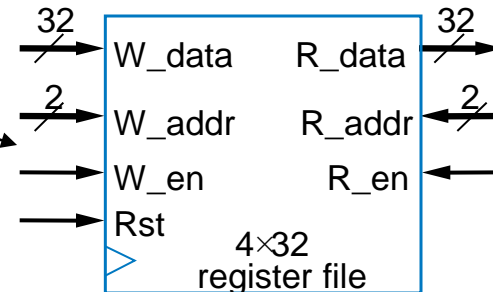
4x32 Register File

- Register Files

- A register file is more efficient than individual registers if we only need to access one or two registers at a time

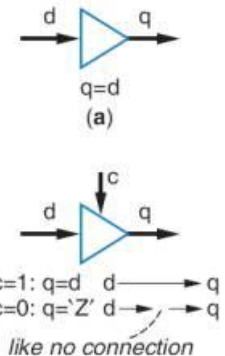
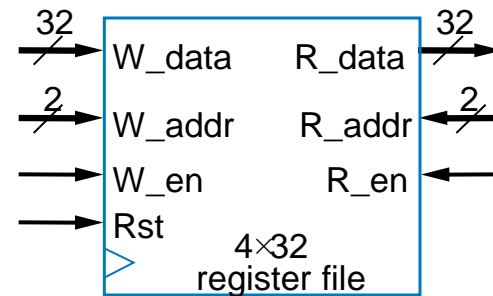
- Consider 4x32 register file (4 registers, each 32-bits wide)

- Need decoder with enable
 - Simple extension of Ch 2 decoder
 - Need 32-bit register with parallel load input and a tri-state buffered outputs
 - Implement as 32-bit register with output enable
 - Output of all registers connected to R_data
 - Only one register should output value to bus
 - All other register should output high-impedance
 - Can omit signal-strengthening driver
 - Synthesis tool would determine when/where to insert driver

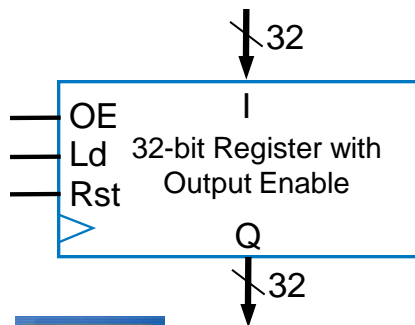


32-Bit Register With Output Enable

- High-impedance
 - Represents an output that is neither driven high nor driven low
 - **high-impedance** → written as **z** or **Z**
 - "S <= z;"
 - Allows for the outputs of several components to be wired together
 - Only one component should output a 0 or 1
 - All other components should output z
 - Typically achieved using three-state buffers

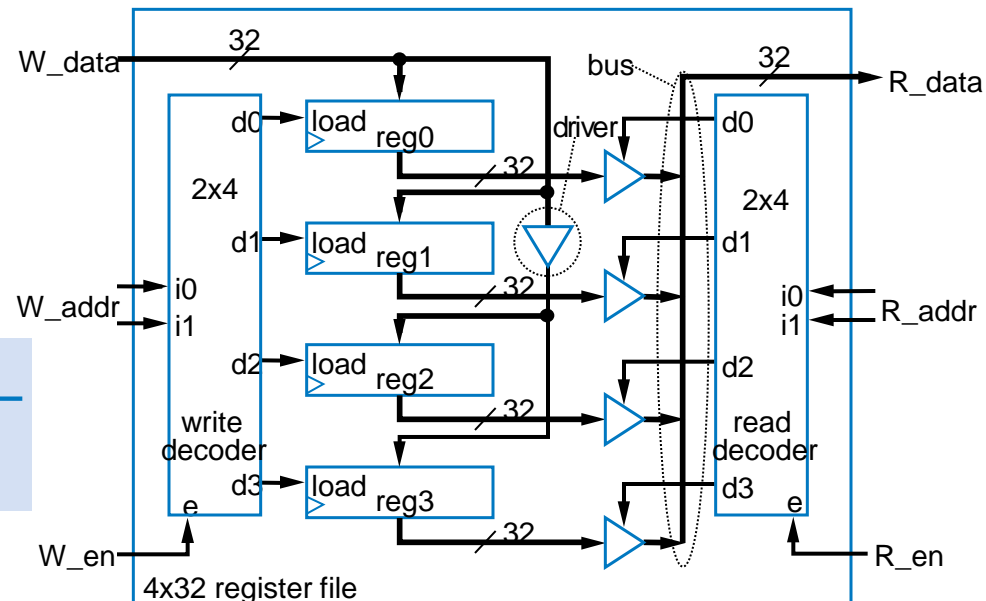


- Register with Output Enable
 - Three-state buffers are part of register



OE	Q
0	High-Impedance – Z
1	Stored Value (0 or 1)

Output Enable Operation



Structural 4x32 Register File

- Register File
 - Structurally connect decoders and registers to create register file

```
`timescale 1 ns/1 ns

module RegFile4x32(R_Addr,W_Addr,R_en,W_en,R_Data,W_Data,Clk,Rst);

    input [1:0] R_Addr, W_Addr;
    input R_en, W_en;
    output [31:0] R_Data;
    input [31:0] W_Data;
    input Clk, Rst;

    wire W_d3, W_d2, W_d1, W_d0;
    wire R_d3, R_d2, R_d1, R_d0;

    Dcd2x4wEn R_Dcd (R_Addr[1],R_Addr[0],R_en,
                    R_d3,R_d2,R_d1,R_d0);
    Dcd2x4wEn W_Dcd (W_Addr[1],W_Addr[0],W_en,
                    W_d3,W_d2,W_d1,W_d0);

    Reg32wOE Reg0 (W_Data,R_Data,R_d0,W_d0,Clk,Rst);
    Reg32wOE Reg1 (W_Data,R_Data,R_d1,W_d1,Clk,Rst);
    Reg32wOE Reg2 (W_Data,R_Data,R_d2,W_d2,Clk,Rst);
    Reg32wOE Reg3 (W_Data,R_Data,R_d3,W_d3,Clk,Rst);
endmodule
```



Structural 4x32 Register File Testbench

```
`timescale 1 ns/1 ns
```

```
module Testbench();
```

```
    reg [1:0] R_Addr_s, W_Addr_s;
```

```
    reg R_en_s, W_en_s;
```

```
    wire [31:0] R_Data_s;
```

```
    reg [31:0] W_Data_s;
```

```
    reg Clk_s, Rst_s;
```

```
    integer Index;
```

```
    RegFile4x32 CompToTest
```

```
        (R_Addr_s, W_Addr_s, R_en_s, W_en_s,  
         R_Data_s, W_Data_s, Clk_s, Rst_s);
```

```
    // Clock Procedure
```

```
    always begin
```

```
        Clk_s <= 0;
```

```
        #10;
```

```
        Clk_s <= 1;
```

```
        #10;
```

```
    end
```

- Writes some values,
then reads and checks

```
    // Vector Procedure
```

```
    initial begin
```

```
        Rst_s <= 1;
```

```
        R_Addr_s <= 0'b00; W_Addr_s <= 0'b00;
```

```
        R_en_s <= 0; W_en_s <= 0;
```

```
        @(posedge Clk_s);
```

```
        #5 Rst_s <= 0;
```

```
        @(posedge Clk_s);
```

```
        #5;
```

```
        // Write values to registers
```

```
        for (Index=0; Index<=3; Index=Index+1) begin
```

```
            W_Addr_s <= Index; W_Data_s <= Index;
```

```
            W_en_s <= 1;
```

```
            @(posedge Clk_s);
```

```
            #5;
```

```
        end
```

```
        W_en_s <= 0;
```

```
        // Check for correct read values from registers
```

```
        for (Index=0; Index<=3; Index=Index+1) begin
```

```
            R_Addr_s <= Index; R_en_s <= 1;
```

```
            @(posedge Clk_s);
```

```
            #5;
```

```
            if( R_Data_s != Index )
```

```
                $display("Failed case %d.", Index);
```

```
        end
```

```
        R_en_s <= 0;
```

```
        #5;
```

```
        if( R_Data_s != 32'hZZZZZZZZ )
```

```
            $display("Failed no read case.");
```

```
    end
```



Behavioral 4x32 Register File

- Register File
 - Can define behaviorally
- Declares a 4-element array
 - Each element 32-bits
 - Element address range defines starting and ending addresses for array elements
 - Specified at end of declaration to distinguish from vector range specification
 - Array elements accessed using index
 - RegFile[0] <= 32'd0; – sets first array element to 32 0s
 - Note that vector may be used as array index: RegFile[W_Addr]

```
`timescale 1 ns/1 ns

module RegFile4x32(R_Addr, W_Addr, R_en, W_en,
                  R_Data, W_Data, Clk, Rst);

    input [1:0] R_Addr, W_Addr;
    input R_en, W_en;
    output reg [31:0] R_Data;
    input [31:0] W_Data;
    input Clk, Rst;

    reg [31:0] RegFile [0:3];

    // Write procedure
    always @(posedge Clk) begin
        if (Rst==1) begin
            RegFile[0] <= 32'd0;
            RegFile[1] <= 32'd0;
            RegFile[2] <= 32'd0;
            RegFile[3] <= 32'd0;
        end
        else if (W_en==1) begin
            RegFile[W_Addr] <= W_Data;
        end
    end

    // Read procedure
    always @* begin
        if (R_en==1)
            R_Data <= RegFile[R_Addr];
        else
            R_Data <= 32'hZZZZZZZZ;
        end
    end
endmodule
```



Behavioral 4x32

Register File

- Note: Must use earlier-described implicit sensitivity list "@"* for Read procedure
 - Because event control may not include an array
 - Could instead include each array element in list (RegFile[0], RegFile[1], ...), but cumbersome, especially for large arrays

```
`timescale 1 ns/1 ns

module RegFile4x32(R_Addr, W_Addr, R_en, W_en,
                  R_Data, W_Data, Clk, Rst);

    input [1:0] R_Addr, W_Addr;
    input R_en, W_en;
    output reg [31:0] R_Data;
    input [31:0] W_Data;
    input Clk, Rst;

    reg [31:0] RegFile [0:3];

    // Write procedure
    always @(posedge Clk) begin
        if (Rst==1) begin
            RegFile[0] <= 32'd0;
            RegFile[1] <= 32'd0;
            RegFile[2] <= 32'd0;
            RegFile[3] <= 32'd0;
        end
        else if (W_en==1) begin
            RegFile[W_Addr] <= W_Data;
        end
    end

    // Read procedure
    always @* begin
        if (R_en==1)
            R_Data <= RegFile[R_Addr];
        else
            R_Data <= 32'hZZZZZZZZ;
        end
    end
endmodule
```



Common Pitfall

Using logical operators instead of bitwise

- Both bitwise and logical AND, OR, and NOT operators exist
 - Easy to mistakenly use logical operator instead bitwise operator, and vice versa
 - May work for single bit inputs, but will produce incorrect results for multi-bit vectors
 - Bitwise Operators:
 - `&`: bitwise AND
 - `|`: bitwise OR
 - `~`: bitwise NOT
 - *Performs operation bit-by-bit resulting in multi-bit vector as wide as largest input operand*
 - Logical Operators:
 - `&&`: logical AND
 - `||`: logical OR
 - `!`: logical NOT (negation)
 - *Performs operation by interpreting input operands as logical values of true or false, resulting in a single bit output of 0 or 1*

```
if( A & 4'b0100 ) begin
    BitSet <= 1;
end
else begin
    BitSet <= 0;
end
```

*Bitwise & operator results
in correct output*

```
if( A && 4'b0100 ) begin
    BitSet <= 1;
end
else begin
    BitSet <= 0;
end
```

*Logical && operator
results in incorrect output*



Common Pitfall

Using logical operators instead of bitwise

- Consider a simple if-else statement that will determine if bit 2 of a 4-bit input A is 1, and set an output BitSet accordingly
 - Using the *bitwise* `&` operator will result in correct output
 - Assume A is 1000: $1000 \& 0100$ results in 0000
 - $1\&0=0, 0\&1=0, 0\&0=0, \text{ and } 0\&0=0$
 - Within an if expression, a value of zero is considered false and BitSet will be assigned the correct value of 0 within else part
 - Using the logical `&&` operator will result in incorrect output
 - Assume A is 1000: $1000 \&\& 0100$ results in 1 – both inputs are non-zero and will be interpreted as true (1), where $1\&\&1=1$
 - Within if expression, 1 is considered true and BitSet will be assigned the incorrect value of 1 within if part

`if(A & 4'b0100) begin
 BitSet <= 1;
end
else begin
 BitSet <= 0;
end`

*Bitwise & operator results
in correct output*

`if(A && 4'b0100) begin
 BitSet <= 1;
end
else begin
 BitSet <= 0;
end`

*Logical && operator
results in incorrect output*

