

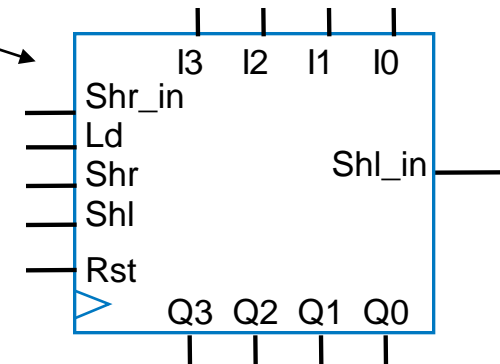
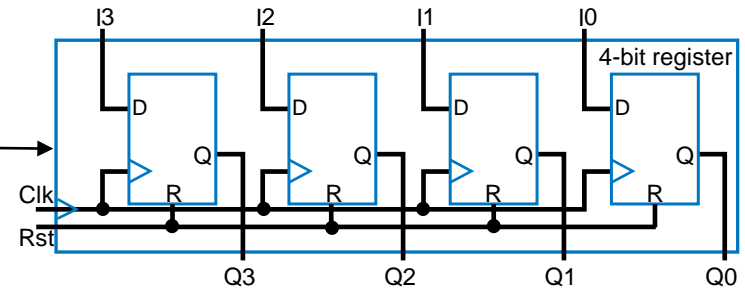


# Multifunction Register Behavior



# Multifunction Register Behavior

- Previously-considered register loaded on every clock cycle
- Now consider register with control inputs, such as load or shift
  - Could describe structurally
    - Four flip-flops, four muxes, and some combinational logic (to convert control inputs to mux select inputs)
  - We'll describe behaviorally



| Ld | Shr | Shl | Operation                               |
|----|-----|-----|---|
| 0  | 0   | 0   | Maintain present value                  |
| 0  | 0   | 1   | Shift left                              |
| 0  | 1   | 0   | Shift right                             |
| 0  | 1   | 1   | Shift right – Shr has priority over Shl |
| 1  | 0   | 0   | Parallel load                           |
| 1  | 0   | 1   | Parallel load – Id has priority         |
| 1  | 1   | 0   | Parallel load – Id has priority         |
| 1  | 1   | 1   | Parallel load – Id has priority         |

| Ld | Shr | Shl | Operation      |
|----|-----|-----|----------------|
| 0  | 0   | 0   | Maintain value |
| 0  | 0   | 1   | Shft left      |
| 0  | 1   | X   | Shift right    |
| 1  | X   | X   | Parallel load  |

*Compact register operation table, clearly showing priorities*



# Multifunction Register

## Behavior

- Use if-else-if construct
  - else-if parts ensure correct priority of control inputs
    - Rst has first priority, then Ld, then Shr, and finally Shl
- Shift by assigning each bit
  - Recall that statement order doesn't matter
- Use reg variable R for storage
  - Best not to try to use port Q – good practice dictates not reading a module's output ports from within a module
- Use **continuous assignment** to update Q when R changes
  - Identifier on left of "=" must be a net, not a variable

```

`timescale 1 ns/1 ns

module MfReg4(I, Q, Ld, Shr, Shl, Shr_in, Shl_in, Clk, Rst);

    input [3:0] I;
    output [3:0] Q;
    input Ld, Shr, Shl, Shr_in, Shl_in;
    input Clk, Rst;

    reg [3:0] R;

    always @(posedge Clk) begin
        if (Rst == 1)
            R <= 4'b0000;
        else if (Ld == 1)
            R <= I;
        else if (Shr == 1) begin
            R[3] <= Shr_in; R[2] <= R[3];
            R[1] <= R[2]; R[0] <= R[1];
        end
        else if (Shl == 1) begin
            R[0] <= Shl_in; R[1] <= R[0];
            R[2] <= R[1]; R[3] <= R[2];
        end
    end
    assign Q = R;
endmodule

```

| Ld | Shr | Shl | Operation      |
|----|-----|-----|----------------|
| 0  | 0   | 0   | Maintain value |
| 0  | 0   | 1   | Shift left     |
| 0  | 1   | X   | Shift right    |
| 1  | X   | X   | Parallel load  |



# Multifunction Register

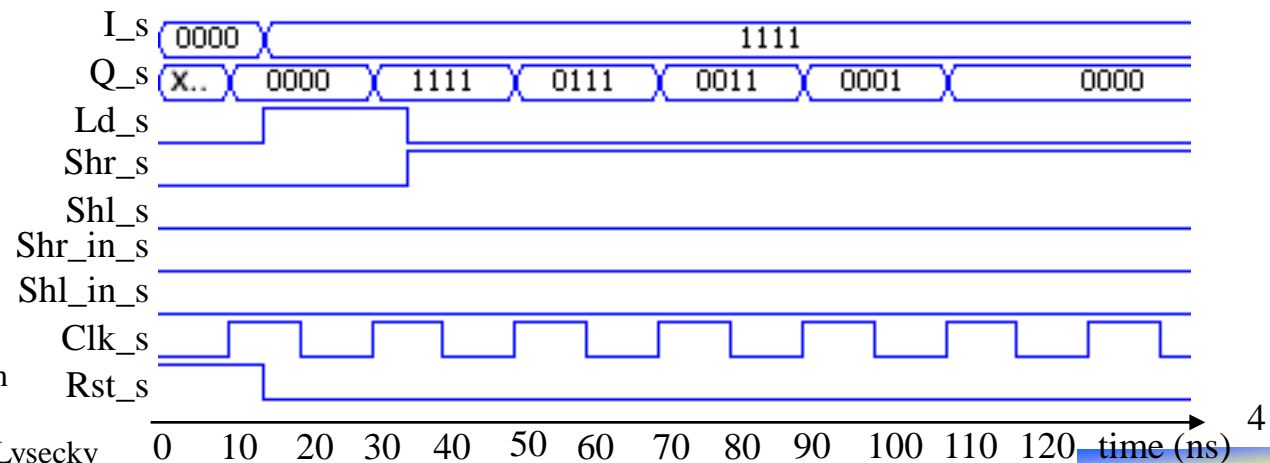
## Behavior

- Testbench should test numerous possible loads and shifts
  - Testbench shown is brief
  - Resets register to 0000
  - Loads 1111
  - Shifts right, shifting in 0
  - Continues shifting right
    - Eventually register is 0000

```

...
// Clock Procedure
...

// Vector Procedure
initial begin
    Rst_s <= 1;
    I_s <= 4'b0000;
    Ld_s <= 0; Shr_s <= 0; Shl_s <= 0;
    Shr_in_s <= 0; Shl_in_s <= 0;
    @(posedge Clk_s);
    #5 Rst_s <= 0;
    I_s <= 4'b1111; Ld_s <= 1;
    @(posedge Clk_s);
    #5 Ld_s <= 0; Shr_s <= 1;
    // Good testbench needs more vectors
end
endmodule
    
```



# Multifunction Register

## Behavior

- **Question:** Does the shown description, with **Q declared as a reg**, and "**Q <= R;**" as the last statement, correctly describe the register?
- **Answer:** No. Q gets the *present* value of R, not the *scheduled* value.

```
`timescale 1 ns/1 ns

module MfReg4(I, Q, Ld, Shr, Shl, Shr_in, Shl_in, Clk, Rst);

    input [3:0] I;
    output [3:0] Q;
    reg [3:0] Q;
    input Ld, Shr, Shl, Shr_in, Shl_in;
    input Clk, Rst;

    reg [3:0] R;

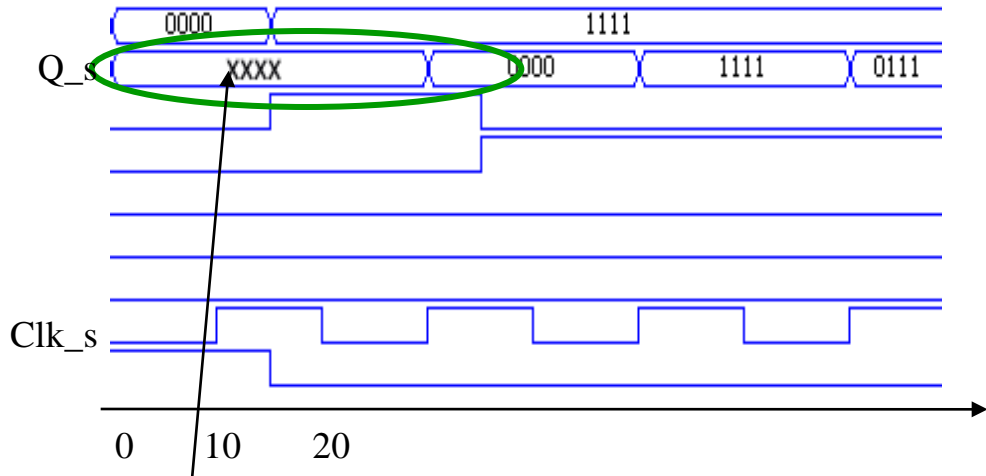
    always @(posedge Clk) begin
        if (Rst == 1)
            R <= 4'b0000;
        else if (Ld == 1)
            R <= I;
        else if (Shr == 1) begin
            R[3] <= Shr_in; R[2] <= R[3];
            R[1] <= R[2]; R[0] <= R[1];
        end
        else if (Shl == 1) begin
            R[0] <= Shl_in; R[1] <= R[0];
            R[2] <= R[1]; R[3] <= R[2];
        end
        Q <= R;
    end
endmodule
```



# Multifunction Register ... Behavior

- **Question:** Does the shown description, with Q declared as a reg, and "Q <= R;" as the last statement, correctly describe the register?
- **Answer:** No. Q gets the present value of R, not the scheduled value.
  - Q thus lags behind R by 1 cycle

```
...
always @(posedge Clk) begin
    if (Rst == 1)
        R <= 4'b0000;
    else if (Ld == 1)
        R <= I;
    else if (Shr == 1) begin
        R[3] <= Shr_in; R[2] <= R[3];
        R[1] <= R[2]; R[0] <= R[1];
    end
    else if (Shl == 1) begin
        R[0] <= Shl_in; R[1] <= R[0];
        R[2] <= R[1]; R[3] <= R[2];
    end
    Q <= R;
end
...
```



*Should have become 0000 on the first clock cycle*

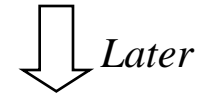


# Common Pitfall

## Not Using Begin-End Block in If Statement

- For more compact code, designers sometimes **don't use begin-end block** for if statement having just one sub-statement
  - As in our register description
- Problem occurs if one adds another sub-statement later without remembering to add **begin-end block**
- Solution – *Always* use begin-end block in if statement

```
always @(posedge Clk) begin
    if (Rst == 1)
        R <= 4'b0000;
```



```
always @(posedge Clk) begin
    if (Rst == 1)
        R <= 4'b0000;
        $display("Reset done.");
```

*Displays even if Rst not 1*

```
always @(posedge Clk) begin
    if (Rst == 1) begin
        R <= 4'b0000;
    end
```



```
always @(posedge Clk) begin
    if (Rst == 1) begin
        R <= 4'b0000;
        $display("Reset done.");
    end
```





# Adder Behavior





# 4-Bit Adder

- 4-bit adder adds two 4-bit binary inputs A and B, sets 4-bit output S
- Could describe structurally
  - Carry-ripple: 4 full-adders
- Behaviorally
  - Simply:  $S \leq A + B$
  - "always" procedure sensitive to A and B
    - Adder is combinational – must include all inputs in sensitivity list
    - Note: procedure resumes if *any* bit in either vector changes

```
`timescale 1 ns/1 ns

module Add4(A, B, S);

    input [3:0] A, B;
    output [3:0] S;
    reg [3:0] S;

    always @(A, B) begin
        S <= A + B;
    end
endmodule
```



# 4-Bit Adder

- "+" is built-in arithmetic operator for addition
  - Built-in arithmetic operators include:
    - + : addition
    - - : subtraction
    - \* : multiplication
    - / : division
    - % : modulus
    - \*\* : power ("a \*\* b" is a raised to the power of b)
  - The operators are intentionally defined to be similar to those in the C programming language

```
`timescale 1 ns/1 ns

module Add4(A, B, S);

    input [3:0] A, B;
    output [3:0] S;
    reg [3:0] S;

    always @(A, B) begin
        S <= A + B;
    end
endmodule
```



# 4-Bit Adder Testbench

- Standard testbench format
  - Needs more vectors than shown
  - Should also be self-checking
- Simulation yields
  - $0011 + 0001 \rightarrow S\_s$  is 0100
    - $3 + 1 = 4$
  - $1100 + 0011 \rightarrow S\_s$  is 1111
    - $12 + 3 = 15$
  - $5 + 2 \rightarrow S\_s$  is 0111
    - Last vector shows use of decimal constant rather than binary

```
`timescale 1 ns/1 ns

module Testbench();

    reg [3:0] A_s, B_s;
    wire [3:0] S_s;

    Add4 CompToTest(A_s, B_s, S_s);

    initial begin
        A_s <= 4'b0011; B_s <= 4'b0001;
        #10;
        A_s <= 4'b1100; B_s <= 4'b0011;
        #10;
        A_s <= 4'd5; // Equivalent to 4'b0101
        B_s <= 4'd2; // Equivalent to 4'b0010
        // Good testbench needs more vectors
    end
endmodule
```



# 4-Bit Adder with Carry-In and Carry-Out

- Adders have carry-in and carry-out bits
  - Extend Add4 with Ci, Co
- $S \leq A + B + Ci$ 
  - Yields correct sum
    - "+" operator handles different bit-widths – extends Ci to 4 bits, padded on left with 0s
  - But carry-out?
    - S is only 4 bits; Co is a fifth bit
- Solution – Do 5-bit add, separate fifth bit (carry-out) from lower four
  - Uses concatenate operator "{}"
  - Uses blocking assignment "="
  - Both to be described now

```
`timescale 1 ns/1 ns

module Add4wCarry(A, B, Ci, S, Co);

    input [3:0] A, B;
    input Ci;
    output [3:0] S;
    reg [3:0] S;
    output Co;
    reg Co;

    reg [4:0] A5, B5, S5;

    always @(A, B, Ci) begin
        A5 = {1'b0, A}; B5 = {1'b0, B};
        S5 = A5 + B5 + Ci;
        S <= S5[3:0];
        Co <= S5[4];
    end
endmodule
```



# 4-Bit Adder with Carry-In and Carry-Out

## Concatenation Operator

- Concatenation operator "{ }"
  - Joins bits from two or more expressions
    - Expressions separated by commas within { }
  - {1b'0, A} → 5-bit value:
    - "0 A[3] A[2] A[1] A[0]"
  - {1b'0, 4b'0011} → "00011"
  - {2b'11, 2b'00, 2b'01} → "110001"

```
`timescale 1 ns/1 ns

module Add4wCarry(A, B, Ci, S, Co);

    input [3:0] A, B;
    input Ci;
    output [3:0] S;
    reg [3:0] S;
    output Co;
    reg Co;

    reg [4:0] A5, B5, S5;

    always @(A, B, Ci) begin
        A5 = {1'b0, A}; B5 = {1'b0, B};
        S5 = A5 + B5 + Ci;
        S <= S5[3:0];
        Co <= S5[4];
    end
endmodule
```



# 4-Bit Adder with Carry-In and Carry-Out

## Blocking and Non Blocking Assignment Statements

- **Blocking assignment** statement

- Uses "="
- Variable is updated before execution proceeds
- Like variable update in C language

- **Non-blocking assignment** statement

- Uses "<="
- Update is scheduled but doesn't occur until later in simulation cycle
- What we've been using until now

- **Guideline**

- Use blocking assignment when computing intermediate values

```
`timescale 1 ns/1 ns

module Add4wCarry(A, B, Ci, S, Co);

    input [3:0] A, B;
    input Ci;
    output [3:0] S;
    reg [3:0] S;
    output Co;
    reg Co;

    reg [4:0] A5, B5, S5;

    always @(A, B, Ci) begin
        A5 = {1'b0, A}; B5 = {1'b0, B};
        S5 = A5 + B5 + Ci;
        S <= S5[3:0];
        Co <= S5[4];
    end
endmodule
```



# 4-Bit Adder with Carry-In and Carry-Out

- $A5 = \{1'b0, A\} \rightarrow$  5-bit version of A
- $B5 = \{1'b0, B\} \rightarrow$  5-bit version of B
- $S5 = A5 + B5 + Ci \rightarrow$  5-bit sum
- Note:
  - Blocking assignment "=" means that above values are updated immediately, rather than being scheduled for update later. Thus, subsequent statements use updated values
- $S \leq S5[3:0] \rightarrow$  4-bit S gets 4 low bits of S5
  - Part selection used to access multiple bits within vector
    - Desired high and low bit positions specified within [] separated by :
- $Co \leq S5[4] \rightarrow$  Co gets 5th bit of S5, which corresponds to the carry-out of  $A+B+Ci$

```
`timescale 1 ns/1 ns

module Add4wCarry(A, B, Ci, S, Co);

    input [3:0] A, B;
    input Ci;
    output [3:0] S;
    reg [3:0] S;
    output Co;
    reg Co;

    reg [4:0] A5, B5, S5;

    always @(A, B, Ci) begin
        A5 = {1'b0, A}; B5 = {1'b0, B};
        S5 = A5 + B5 + Ci;
        S <= S5[3:0];
        Co <= S5[4];
    end
endmodule
```



# 4-Bit Adder with Carry-In and Carry-Out

## Alternative Description

- A more compact description is possible
- Use concatenation on the left side of assignment
  - $\{Co, S\} \leq A + B + Ci$
  - Left side thus 5 bits wide
- Rule
  - For the + operator, all operands extended to width of widest operand, including left side
  - Left side is 5 bits  $\rightarrow$  A, B, and Ci all extended to 5 bits, left padded with 0s
  - E.g., A: 0011, B: 0001, Ci: 1  $\rightarrow$  00011+00001+00000 yields 00100
    - Co gets first 0, S gets 0100
- Though longer, previous description synthesizes to same circuit
  - reg [4:0] A5, B5, S5; – Synthesize into wires

```
`timescale 1 ns/1 ns

module Add4wCarry(A, B, Ci, S, Co);

    input [3:0] A, B;
    input Ci;
    output reg [3:0] S;
    output reg Co;

    always @(A, B, Ci) begin
        {Co, S} <= A + B + Ci;
    end
endmodule
```





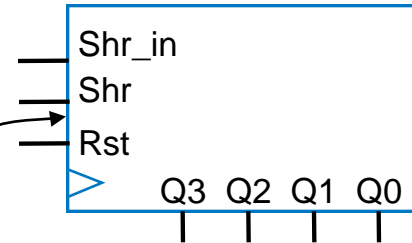


# Shift Register Behavior



# 4-Bit Shift Register

- Shift Register
  - Consider a 4-bit shift register with only a right shift control input
    - Either retains its current value or shift the register contents right
  - Can again describe register behaviorally
    - Perform shifting bit by bit, as in previous multifunction register example
- Could also use concatenation
  - Replace bit-by-bit assignment with single statement using concatenation
  - $R \leq \{Shr\_in, R[3], R[2], R[1]\}$
- What if the register has 32-bits?
  - Both bit-by-bit assignment and concatenation become tedious for large items
  - Could lead to errors



| Shr | Operation    |
|-----|--------------|
| 0   | Retain Value |
| 1   | Shift right  |

```
`timescale 1 ns/1 ns

module ShiftReg4(Q, Shr, Shr_in, Clk, Rst);

    output [3:0] Q;
    input Shr, Shr_in;
    input Clk, Rst;

    reg [3:0] R;

    always @(posedge Clk) begin
        if (Rst == 1)
            R <= 4'b0000;
        else if (Shr == 1) begin
            R[3] <= Shr_in; R[2] <= R[3];
            R[1] <= R[2]; R[0] <= R[1];
        end
    end

    assign Q = R;
endmodule
```



# 32-Bit Shift Register

- Now consider a 32-bit shift register with right shift control input
  - Both bit by bit assignment and concatenation become cumbersome, tedious, and error prone
- Solution:
  - Use loop to perform shifting
- Loop
  - Defines a set of statements that will be repeatedly executed some number of times
  - Loop parameters control execution of loop

```
`timescale 1 ns/1 ns

module ShiftReg32(Q, Shr, Shr_in, Clk, Rst);

    output [31:0] Q;
    input Shr, Shr_in;
    input Clk, Rst;

    reg [31:0] R;
    integer Index;

    always @(posedge Clk) begin
        if (Rst == 1)
            R <= 32'h00000000;
        else if (Shr == 1) begin
            R[31] <= Shr_in;
            for (Index=0; Index<=30; Index=Index+1) begin
                R[Index] <= R[Index+1];
            end
        end
        assign Q = R;
    end
endmodule
```

*"for" loop explained on next slide*



# 32-Bit Shift Register for loop statement

- for loop statement
  - Typically defines loop that executes specified number of times
- Typically involves:
  - index variable declaration
  - Index variable initialization
    - executed only once
  - Loop condition checked
    - Usually involves index
    - Loop exits if not true
  - Loop body statement executed
    - Usually a begin-end block
    - Followed by execution of index variable update
- Loop thus assigns every R bit to next higher bit
  - Last bit handled by statement  $R[31] \leq \text{Shr\_in}$ ;  $\rightarrow$  Assign highest bit to shift input

```
`timescale 1 ns/1 ns

module ShiftReg32(Q, Shr, Shr_in, Clk, Rst);

    output [31:0] Q;
    input Shr, Shr_in;
    input Clk, Rst;

    reg [31:0] R;
    integer Index;

    always @(posedge Clk) begin
        if (Rst == 1)
            R <= 32'h00000000;
        else if (Shr == 1) begin
            R[31] <= Shr_in;
            for (Index=0; Index<=30; Index=Index+1) begin
                R[Index] <= R[Index+1];
            end
        end
        end
        assign Q = R;
    endmodule
```



# 32-Bit Shift Register

## Integer Data Type

- Index declared as integer
- **integer**
  - Another variable data type
  - Previous was “reg”
    - Bit or vector of bits
  - Integer can be negative or positive (signed), 32-bits
  - Use when it makes code clearer
    - Especially if item not destined to become a physical register

```
`timescale 1 ns/1 ns

module ShiftReg32(Q, Shr, Shr_in, Clk, Rst);

    output [31:0] Q;
    input Shr, Shr_in;
    input Clk, Rst;

    reg [31:0] R;
    integer Index;

    always @(posedge Clk) begin
        if (Rst == 1)
            R <= 32'h00000000;
        else if (Shr == 1) begin
            R[31] <= Shr_in;
            for (Index=0; Index<=30; Index=Index+1) begin
                R[Index] <= R[Index+1];
            end
        end
        end
    assign Q = R;
endmodule
```



# 32-Bit Shift Register

## Relational and Logic Operators

- "<=" – built-in relational operator
  - Looks same as non blocking assignment – distinguished by how operator is used
- Built-in relational operators
  - > : greater than
  - < : less than
  - >= : greater than or equal
  - <= : less than or equal

```
`timescale 1 ns/1 ns

module ShiftReg32(Q, Shr, Shr_in, Clk, Rst);

    output [31:0] Q;
    input Shr, Shr_in;
    input Clk, Rst;

    reg [31:0] R;
    integer Index;

    always @(posedge Clk) begin
        if (Rst == 1)
            R <= 32'h00000000;
        else if (Shr == 1) begin
            R[31] <= Shr_in;
            for (Index=0; Index<=30; Index=Index+1) begin
                R[Index] <= R[Index+1];
            end
        end
    end
    assign Q = R;
endmodule
```



# 32-Bit Shift Register

## Relational and Logic Operators

- Built-in logical operators
  - `!` : logical negation
  - `&&` : logical AND
  - `||` : logical OR
- Built-in equality operators
  - `==` : logical equality
  - `!=` : logical inequality
  - `===` : logical equality
    - including x and z bits  
(*more on this later*)
  - `!==` : logical inequality
    - including x and z bits  
(*more on this later*)

```
`timescale 1 ns/1 ns

module ShiftReg32(Q, Shr, Shr_in, Clk, Rst);

    output [31:0] Q;
    input Shr, Shr_in;
    input Clk, Rst;

    reg [31:0] R;
    integer Index;

    always @(posedge Clk) begin
        if (Rst == 1)
            R <= 32'h00000000;
        else if (Shr == 1) begin
            R[31] <= Shr_in;
            for (Index=0; Index<=30; Index=Index+1) begin
                R[Index] <= R[Index+1];
            end
        end
        assign Q = R;
    end
endmodule
```



# 32-Bit Shift Register Testbench

- Testbench
  - Shifting bits individually into the shift register would also be tedious
  - Use for loops to simplify the testbench
- Shift 16 1s into register
  - Set register to shift right with shift input of 1
  - for loop waits 16 clock cycles
    - Loop executes 16 time, each time waiting for rising clock edge
  - Self-check verifies correctly shifted register output
- Shift 16 0s into register
  - for loop waits 16 clock cycles
  - Self-checks again
- Good testbench would have more vectors

```
...
// Vector Procedure
initial begin
    Rst_s <= 1;
    Shr_s <= 0; Shr_in_s <= 0;
    @(posedge Clk_s);
    #5 Rst_s <= 0;
    @(posedge Clk_s);
    #5 Shr_s <= 1; Shr_in_s <= 1;
    for (Index=0; Index<=15; Index=Index+1) begin
        @(posedge Clk_s);
    end
    #5;
    if (Q_s != 32'hFFFF0000)
        $display("Failed Q=FFFF0000");
    Shr_s <= 1; Shr_in_s <= 0;
    for (Index=0; Index<=15; Index=Index+1) begin
        @(posedge Clk_s);
    end
    #5;
    if (Q_s != 32'h0000FFFF)
        $display("Failed Q=0000FFFF");
    Shr_s <= 0;
end
endmodule
```





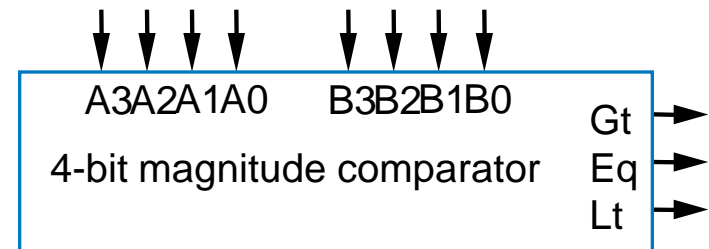


# Comparator



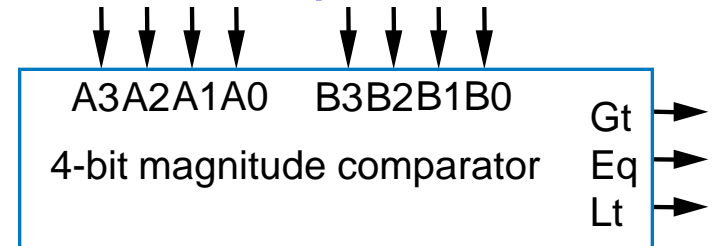
# 4-bit Unsigned/Signed Magnitude Comparator

- Previously
  - Dealt only with unsigned numbers
    - input, output, reg declarations are unsigned unless otherwise specified
- Now consider a simple magnitude comparator that compares a 4-bit unsigned number A with a 4-bit signed number B, with outputs for greater than, less than, and equal
  - A can be 0 to 15 (0000 to 1111)
  - B can be -8 to 7 (1000 to 0111)
  - Need to represent both unsigned and signed numbers



# 4-bit Unsigned/Signed Magnitude Comparator

- Declare A input as before, but declare B input with **signed** keyword
- When comparing A and B using "<", first convert unsigned A to signed value using **\$signed** system function
  - "\$signed(A)" would not work – changes positive number to negative
    - e.g., 1000 would change from meaning 8 to meaning -8
  - Instead, first **extend** A to five bits
    - {1'b0,A} – e.g., 1000 becomes 01000
  - Then convert to signed
    - \$signed({1'b0,A}) – e.g., 01000 as 5-bit signed number is still 8 (due to 0 in highest-order bit)
  - Operands of "<" automatically sign-extended to widest operand's width
    - So B extended to 5-bits with sign bit preserved
  - Comparison is thus correct



```
`timescale 1 ns/1 ns

module Comp4(A, B, Gt, Eq, Lt);

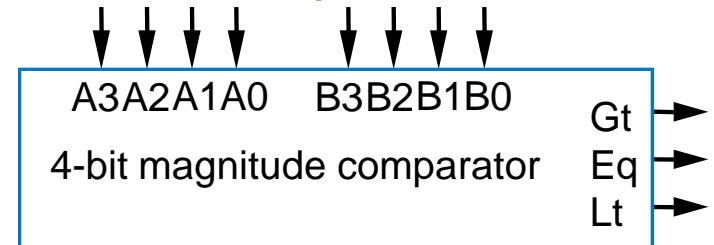
    input [3:0] A;
    input signed [3:0] B;
    output Gt, Eq, Lt;
    reg Gt, Eq, Lt;

    always @(A, B) begin
        if ($signed({1'b0,A}) < B) begin
            Gt <= 0; Eq <= 0; Lt <= 1;
        end
        else if ($signed({1'b0,A}) > B) begin
            Gt <= 1; Eq <= 0; Lt <= 0;
        end
        else begin
            Gt <= 0; Eq <= 1; Lt <= 0;
        end
    end
endmodule
```



# 4-bit Unsigned/Signed Magnitude Comparator

- Performs comparison using if-else-if construct
  - if  $A < B$ :
    - Set Lt to 1, Gt to 0, and Eq to 0
    - If B negative, A will always be greater than B (A is always positive)
  - if  $(A > B)$ 
    - Gt = 1, Lt = 0, and Eq = 0
  - If A is neither greater or less than
    - Eq = 1, Lt = 0, and Gt = 0



```
`timescale 1 ns/1 ns

module Comp4(A, B, Gt, Eq, Lt);

    input [3:0] A;
    input signed [3:0] B;
    output Gt, Eq, Lt;
    reg Gt, Eq, Lt;

    always @(A, B) begin
        if ($signed({1'b0,A}) < B) begin
            Gt <= 0; Eq <= 0; Lt <= 1;
        end
        else if ($signed({1'b0,A}) > B) begin
            Gt <= 1; Eq <= 0; Lt <= 0;
        end
        else begin
            Gt <= 0; Eq <= 1; Lt <= 0;
        end
    end
endmodule
```



# 4-bit Unsigned/Signed Magnitude Comparator

- Testbench should test multiple values for inputs A and B
  - Should perform comparisons for both positive and negative values of B
  - Should have at least one test case in which A is greater than, less than, and equal to B
- Note that reg variable B\_s, used to connect with B, defined as **signed**
- Vectors illustrate use of binary constants as well as decimal constants
  - Negative binary constant achieved using 1 in high-order bit (two's complement form)
  - Negative decimal constant requires negative sign "-" in front of constant

```
`timescale 1 ns/1 ns

module Testbench();

    reg [3:0] A_s;
    reg signed [3:0] B_s;
    wire Gt_s, Eq_s, Lt_s;

    Comp4 CompToTest(A_s, B_s, Gt_s, Eq_s, Lt_s);

    initial begin
        A_s <= 4'b0011; B_s <= 4'b0001;
        #10 A_s <= 4'b1111; B_s <= 4'b0111;
        #10 A_s <= 4'b0111; B_s <= 4'b1011;
        #10 A_s <= 4'b0001; B_s <= 4'b0010;
        #10 A_s <= 4'b0001; B_s <= 4'b0001;
        #10 A_s <= 4'b0000; B_s <= 4'b1111;
        #10 A_s <= 4'd1; B_s <= -4'd1;
        #10 A_s <= 4'd1; B_s <= -4'd8;
        // Good testbench needs more vectors
    end
endmodule
```



# 4-bit Unsigned/Signed Magnitude Comparator

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

- Simulation

- First two vectors compare positive values for both inputs
  - $0011 > 0001 \rightarrow Gt\_s = 1$
  - $1111 > 0111 \rightarrow Gt\_s = 1$
- Third test compares A with negative B
  - $0111 > 1011 \rightarrow Gt\_s = 1$ 
    - $7 > -5$
- Fourth and fifth test should result in the Lt\_s and Eq\_s output asserted, respectively
  - $0001 < 0010 \rightarrow Lt\_s = 1$
  - $0001 = 0001 \rightarrow Eq\_s = 1$
- Next test compares 0 to -1
  - $0000 > 1111 \rightarrow Gt\_s = 1$
- Next test compare 1 to -1
- Last test compares 1 to -8

```
module Testbench();
```

```
    reg [3:0] A_s;  
    reg signed [3:0] B_s;  
    wire Gt_s, Eq_s, Lt_s;
```

```
    Comp4 CompToTest(A_s, B_s, Gt_s, Eq_s, Lt_s);
```

```
    initial begin
```

```
        A_s <= 4'b0011; B_s <= 4'b0001;  
        #10 A_s <= 4'b1111; B_s <= 4'b0111;  
        #10 A_s <= 4'b0111; B_s <= 4'b1011;  
        #10 A_s <= 4'b0001; B_s <= 4'b0010;  
        #10 A_s <= 4'b0001; B_s <= 4'b0001;  
        #10 A_s <= 4'b0000; B_s <= 4'b1111;  
        #10 A_s <= 4'd1; B_s <= -4'd1;  
        #10 A_s <= 4'd1; B_s <= -4'd8;  
        // Good testbench needs more vectors
```

```
    end
```

```
endmodule
```



# Common Pitfall

- Unintentional use of one of many of Verilog's automatic conversions
  - `B_s <= -4'd15`
  - `-4d'15`
    - 4-bit decimal 15 would be 1111
    - Negative of 1111 (15) is 10001 (-15) – Automatically converted to 5 bits
    - Assignment to `B_s` drops the high-order bit, making `B_s=0001`
  - Many similar types of automatic conversions in Verilog
  - Use great caution

```
`timescale 1 ns/1 ns

module Testbench();

    reg [3:0] A_s;
    reg signed [3:0] B_s;
    wire Gt_s, Eq_s, Lt_s;

    Comp4 CompToTest(A_s, B_s, Gt_s, Eq_s, Lt_s);

    initial begin
        ...
        #10 A_s <= 4'd1; B_s <= -4'd1;
        #10 A_s <= 4'd1; B_s <= -4'd15;
        // Good testbench needs more vectors
    end
endmodule
```





## Test Bench With File Input





# Testbench with File Input – 32-Bit Shift Register

- Testbench can read test vectors from an input file
  - Compact
  - Allows for easy integration of new test vectors without modifying testbench or recompiling
  - Can define several separate vector files to test different aspects
- **File:** document located on host computer system, can be read from or written to
  - Verilog has **built-in system procedures for files**
- Four types of procedures in Verilog
  - Initial and always – already seen
  - **Function** – Has at least one input argument, returns a value, no time-controlling statements (executes in one simulation time unit)
  - **Task** – Any number of arguments, no return value, may have time-controlling statements
- Testbench for the 32-Bit shift register reads test vectors from input file
  - Input file specifies bits to be shifted into register
  - Set register to shift right and read input bits from file

```
integer FileId;
reg[8:0] BitChar;
...
// Vector Procedure
initial begin
    FileId = $fopen("vectors.txt", "r");
    if (FileId == 0)
        $display("Could not open input file.");
    else begin
        Rst_s <= 1;
        Shr_s <= 0; Shr_in_s <= 0;
        @(posedge Clk_s);
        #5 Rst_s <= 0;
        @(posedge Clk_s);
        #5 Shr_s <= 1;
        while ($feof(FileId) == 0) begin
            BitChar = $fgetc(FileId);
            if (BitChar == "1") begin
                Shr_in_s <= 1;
                @(posedge Clk_s);
            end
            else if (BitChar == "0") begin
                Shr_in_s <= 0;
                @(posedge Clk_s);
            end
        end
        $fclose(FileId);
    end
    Shr_s <= 0;
end
endmodule
```



# Testbench with File Input – System Procedures for Files

- **\$fopen** – Opens file for access  
Arguments:
  - File name: "vectors.txt"
  - Access type: "r" means read, "w" write, "a" append
  - Returns integer, used to identify opened file (may be more than one file open at one time); 0 means error
- **\$feof** – Returns 0 if end of file has not been reached yet
- **\$fgetc** – Returns next character in file
  - Valid character is 8 bits
  - If error, returns 9-bit value 11111111
    - Thus, variable BitChar is 9 bits, not 8
- **\$fclose** – Closes previously-opened file

```
integer FileId;
reg[8:0] BitChar;
...
// Vector Procedure
initial begin
    FileId = $fopen("vectors.txt", "r");
    if (FileId == 0)
        $display("Could not open input file.");
    else begin
        Rst_s <= 1;
        Shr_s <= 0; Shr_in_s <= 0;
        @(posedge Clk_s);
        #5 Rst_s <= 0;
        @(posedge Clk_s);
        #5 Shr_s <= 1;
        while ($feof(FileId) == 0) begin
            BitChar = $fgetc(FileId);
            if (BitChar == "1") begin
                Shr_in_s <= 1;
                @(posedge Clk_s);
            end
            else if (BitChar == "0") begin
                Shr_in_s <= 0;
                @(posedge Clk_s);
            end
        end
        $fclose(FileId);
    end
    Shr_s <= 0;
end
endmodule
```

*while loop to be described on next slide*



# Testbench with File Input – While Loops

- Uses another form of loop: **while**
  - If **condition** is true, executes loop body statement (usually **begin-end block**)
  - Repeat
- Both while and for loops are common
  - for loop typically used when number of iterations is known (e.g., loop 16 times)
  - while loop typically used when number of iterations not known

```
integer FileId;
reg[8:0] BitChar;
...
// Vector Procedure
initial begin
    FileId = $fopen("vectors.txt", "r");
    if (FileId == 0)
        $display("Could not open input file.");
    else begin
        Rst_s <= 1;
        Shr_s <= 0; Shr_in_s <= 0;
        @(posedge Clk_s);
        #5 Rst_s <= 0;
        @(posedge Clk_s);
        #5 Shr_s <= 1;
        while ($feof(FileId) == 0) begin
            BitChar = $fgetc(FileId);
            if (BitChar == "1") begin
                Shr_in_s <= 1;
                @(posedge Clk_s);
            end
            else if (BitChar == "0") begin
                Shr_in_s <= 0;
                @(posedge Clk_s);
            end
        end
        $fclose(FileId);
    end
    Shr_s <= 0;
end
endmodule
```



# Testbench with File Input – 32-Bit Register

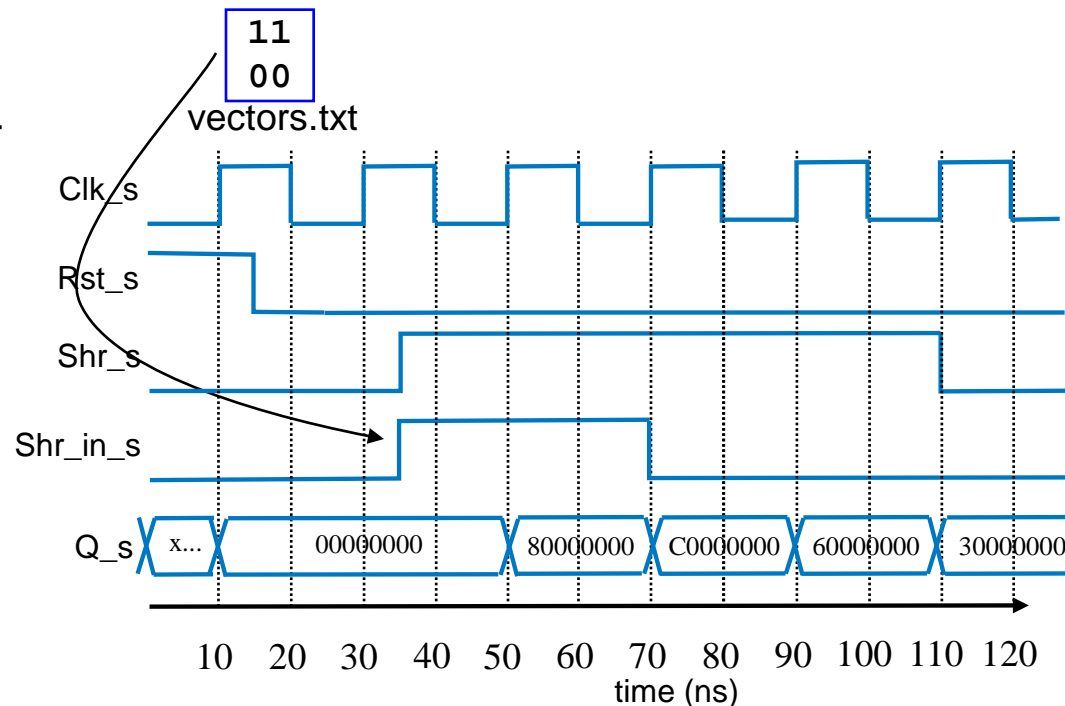
```
integer FileId;
reg[8:0] BitChar;
...
// Vector Procedure
initial begin
    FileId = $fopen("vectors.txt", "r");
    if (FileId == 0)
        $display("Could not open input file.");
    else begin
        Rst_s <= 1;
        Shr_s <= 0; Shr_in_s <= 0;
        @(posedge Clk_s);
        #5 Rst_s <= 0;
        @(posedge Clk_s);
        #5 Shr_s <= 1;
        while ($feof(FileId) == 0) begin
            BitChar = $fgetc(FileId);
            if (BitChar == "1") begin
                Shr_in_s <= 1;
                @(posedge Clk_s);
            end
            else if (BitChar == "0") begin
                Shr_in_s <= 0;
                @(posedge Clk_s);
            end
        end
        $fclose(FileId);
    end
    Shr_s <= 0;
end
endmodule
```

- Open file containing test vectors
  - If file failed to open (maybe doesn't exist), display error message
- Reset register
- Enable shifting
- While we haven't read the entire vector file
  - Read next character from file
  - If 1, shift in a 1
    - Set shift input to 1, wait for clock
  - Else if 0, shift in a 0
  - Can't just assign `Shr_in_s <= BitChar`; one's a bit, one's a 9-bit ASCII encoding of a character
- When read entire file, close file



# Testbench with File Input – 32-Bit Shift Register

- Test vector file can contain as few or as many test vectors as desired
  - Can add new test vectors and simulate without changing testbench
  - Vectors of “1111111111111111” and “0000000000000000” would match previous testbench for 32-bit shift register
  - Consider simulation only for
- Testbench Simulation
  - Waveform shows simulation for test vectors “11” and “00”
  - Value for Q\_s displayed in hexadecimal

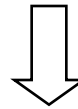


# Common Pitfall

## Unsynthesizable Loop

- Creating an unsynthesizable loop in a description to be synthesized
  - Synthesis must be able to *unroll* the loop into an equivalent straight-line (no loop) sequence of statements
    - To know first statement and last statement of sequence
    - Loop is thus just a shorthand for those statements
  - `Index <= 30`
    - Anything other than constant may prevent unrolling
  - `Index <= Index + 1`
    - Anything other than simple index increment/decrement may prevent unrolling
  - Likewise, while loops (even simple ones) may not be unrolled
- For synthesis, best to use only simple *for* loop with constant bounds, and simple index increment/decrement

```
for (Index=0; Index<=30; Index<=Index+1) begin
    R(Index) <= R[Index+1];
end
```



*Unrolling*

```
R[0] <= R[1];
R[1] <= R[2];
...
R[30] <= R[31];
```

```
Index = 0;
while (Index <= 30) begin
    ... // loop statements
    Index = Index + 1;
end
```

*Some tools can unroll this while loop, better to use a for loop*

