

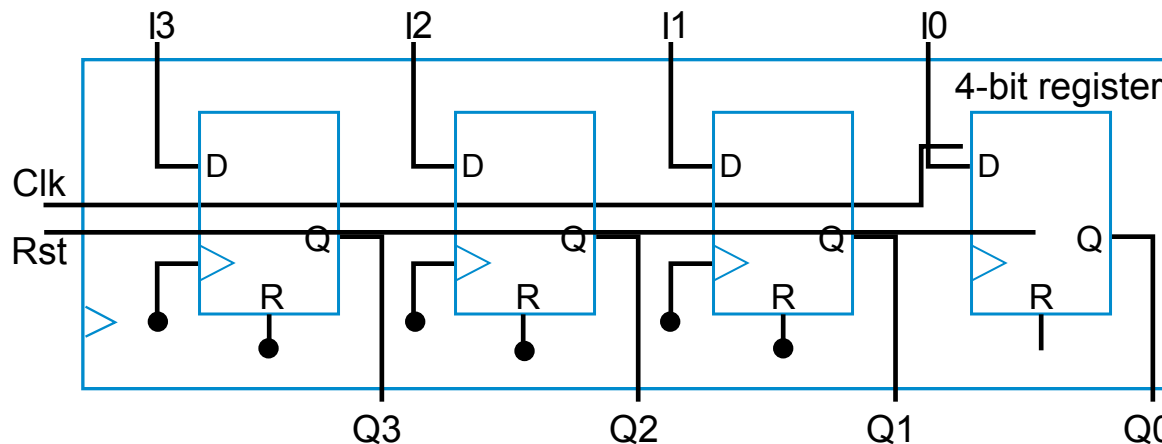
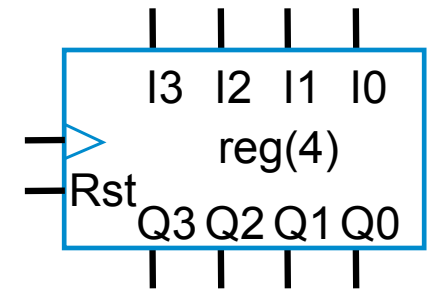
# 3 – Registers, Datapath Components, and Verilog Review

ECE 474A/574A  
COMPUTER-AIDED LOGIC DESIGN

# Register Behavior

2

- Sequential circuits have storage
- **Register**: most common storage component
  - ▣ N-bit register stores N bits
  - ▣ Structure may consist of connected flip-flops

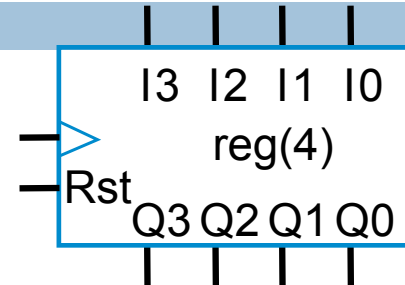


# Register Behavior

## Vectors

3

- Typically just describe register behaviorally
  - ▣ Declare output Q as reg variable to achieve storage
- Uses **vector** types
  - ▣ Collection of bits
    - More convenient than declaring separate bits like I3, I2, I1, I0
  - ▣ Vector's bits are numbered
    - Options: [0:3], [1:4], etc.
    - [3:0]
      - Most-significant bit is on left
  - ▣ Assign with binary constant
    - (more on next slide)



```
module Reg4(I3,I2,I1,I0,Q3,...);
  input I3, I2, I1, I0;
```

```
    I3 I2 I1 I0
```

```
module Reg4(I, Q, ...);
```

```
  input [3:0] I;
```

```
  I: [ ][ ][ ][ ]
```

```
    I[3]I[2]I[1]I[0]
```

```
`timescale 1 ns/1 ns

module Reg4(I, Q, Clk, Rst);

  input [3:0] I;
  output [3:0] Q;
  reg [3:0] Q;
  input Clk, Rst;

  always @(posedge Clk) begin
    if (Rst == 1 )
      Q <= 4'b0000;
    else
      Q <= I;
  end
endmodule
```

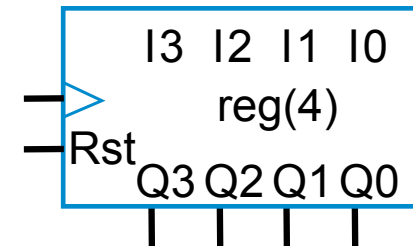
# Register Behavior

## Constants

| | | |

4

- Binary constant
  - 4'b0000
    - 4: size, in number of bits
    - 'b: binary base
    - 0000: binary value
- Other constant bases possible
  - d: decimal base, o: octal base, h: hexadecimal base
  - 12'hFA2
    - 'h: hexadecimal base
    - 12: 3 hex digits require 12 bits
    - FA2: hex value
  - Size is always in bits, and optional
    - 'hFA2 is OK
  - For decimal constant, size and 'd optional
    - 8'd255 or just 255
    - In previous uses like "A <= 1;" 1 and 0 are actually decimal numbers. 'b1 and 'b0 would explicitly represent bits
- Underscores may be inserted into value for readability
  - 12'b1111\_1010\_0010
  - 8\_000\_000



```
`timescale 1 ns/1 ns

module Reg4(I, Q, Clk, Rst);

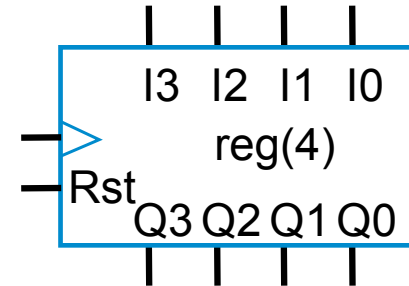
    input [3:0] I;
    output [3:0] Q;
    reg [3:0] Q;
    input Clk, Rst;

    always @(posedge Clk) begin
        if (Rst == 1 )
            Q <= 4'b0000;
        else
            Q <= I;
        end
    end
endmodule
```

# Register Behavior

5

- Procedure's event control involves Clk input
  - ▣ *Not* the I input. Thus, synchronous
  - ▣ "posedge Clk"
    - Event is not just any change on Clk, but specifically change from 0 to 1 (positive edge)
    - negedge also possible
- Process has synchronous reset
  - ▣ Resets output Q only on rising edge of Clk
- Process writes output Q
  - ▣ Q declared as reg variable, thus *stores* value too



```
`timescale 1 ns/1 ns

module Reg4(I, Q, Clk, Rst);

    input [3:0] I;
    output [3:0] Q;
    reg [3:0] Q;
    input Clk, Rst;

    always @(posedge Clk) begin
        if (Rst == 1 )
            Q <= 4'b0000;
        else
            Q <= I;
        end
    end
endmodule
```

# Register Behavior

## Testbench

6

- reg/wire declarations and module instantiation similar to previous testbenches

- Module uses two procedures

- One generates 20 ns clock

- 0 for 10 ns, 1 for 10 ns
- Note: always procedure repeats

- Other provides values for inputs Rst and I (i.e., vectors)

- initial procedure executes just once, does not repeat
- (more on next slide)

```
`timescale 1 ns/1 ns

module Testbench();

    reg [3:0] I_s;
    reg Clk_s, Rst_s;
    wire [3:0] Q_s;

    Reg4 CompToTest(I_s, Q_s, Clk_s, Rst_s);

    // Clock Procedure
    always begin
        Clk_s <= 0;
        #10;
        Clk_s <= 1;
        #10;
    end // Note: Procedure repeats

    // Vector Procedure
    initial begin
        Rst_s <= 1;
        I_s <= 4'b0000;
        @(posedge Clk_s);
        #5 Rst_s <= 0;
        I_s <= 4'b0000;
        @(posedge Clk_s);
        #5 Rst_s <= 0;
        I_s <= 4'b1010;
        @(posedge Clk_s);
        #5 Rst_s <= 0;
        I_s <= 4'b1111;
    end
endmodule
```

# Register Behavior

## Testbench

7

- Variables/nets can be shared between procedures
  - ▣ Only one procedure should write to variable
    - Variable can be read by many procedures
    - Clock procedure writes to Clk\_s
    - Vector procedures reads Clk\_s
- Event control "@(posedge Clk\_s)"
  - ▣ May be prepended to statement to synchronize execution with event occurrence
    - Statement may be just ";" as in example
    - In previous examples, the "statement" was a sequential block (begin-end)
  - ▣ Test vectors thus don't include the clock's period hard coded
- Care taken to change input values away from clock edges

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

```
module Testbench();
```

```
    reg [3:0] I_s;  
    reg Clk_s, Rst_s;  
    wire [3:0] Q_s;
```

```
    Reg4 CompToTest(I_s, Q_s, Clk_s, Rst_s);
```

```
    // Clock Procedure
```

```
    always begin
```

```
        Clk_s <= 0;
```

```
        #10;
```

```
        Clk_s <= 1;
```

```
        #10;
```

```
    end    // Note: Procedure repeats
```

```
    // Vector Procedure
```

```
    initial begin
```

```
        Rst_s <= 1;
```

```
        I_s <= 4'b0000;
```

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

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

```
        I_s <= 4'b0000;
```

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

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

```
        I_s <= 4'b1010;
```

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

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

```
        I_s <= 4'b1111;
```

```
    end
```

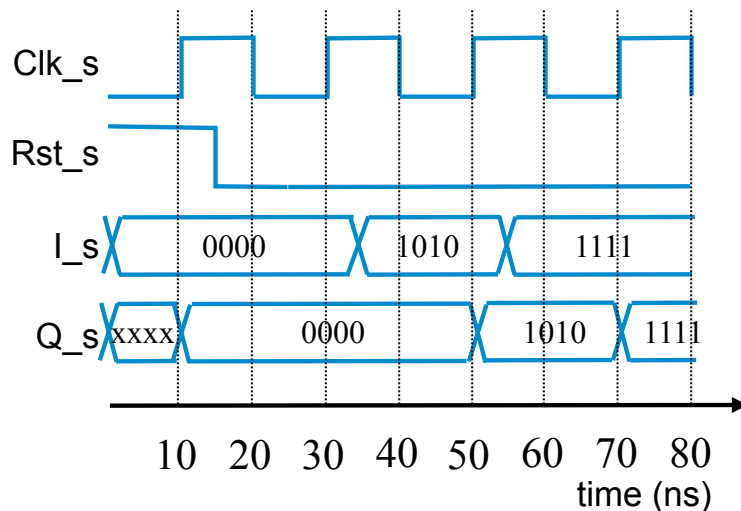
```
endmodule
```

# Register Behavior

## Testbench

8

- Simulation results
  - ▣ Note that Q\_s updated only on rising clock edges
  - ▣ Note Q\_s thus unknown until first clock edge
    - Q\_s is reset to “0000” on first clock edge



...

```
// Vector Procedure
initial begin
    Rst_s <= 1;
    I_s <= 4'b0000;
    @(posedge Clk_s);
    #5 Rst_s <= 0;
    I_s <= 4'b0000;
    @(posedge Clk_s);
    #5 Rst_s <= 0;
    I_s <= 4'b1010;
    @(posedge Clk_s);
    #5 Rst_s <= 0;
    I_s <= 4'b1111;
end
```

...

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

...

*Initial value of a bit is the unknown value x*

*Remember that Q\_s is connected to Q, and I\_s to I, in the testbench*



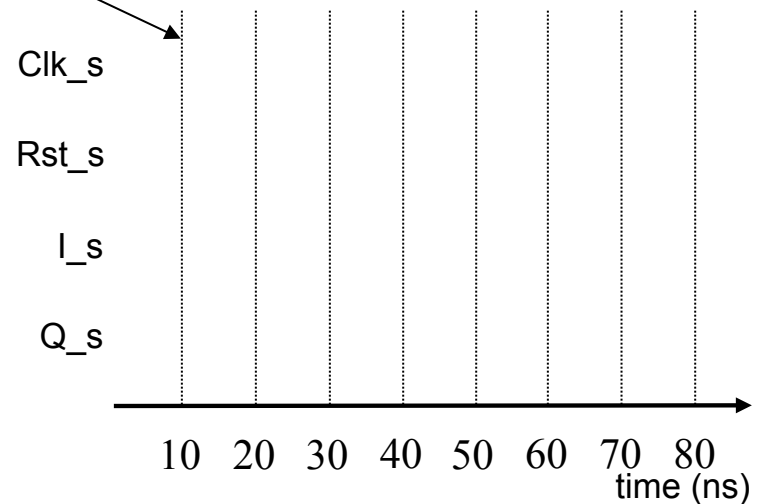
# Common Pitfalls

9

- Using "always" instead of "initial" procedure
  - ▣ Causes repeated procedure execution
- Not including any delay control or event control in an always procedure
  - ▣ May cause infinite loop in the simulator
    - Simulator executes those statements over and over, never executing statements of another procedure
    - Simulation time can never advance
  - ▣ Symptom – Simulator appears to just hang, generating no waveforms

```
// Vector Procedure
always begin
    Rst_s <= 1;
    I_s <= 4'b0000;
    @(posedge Clk_s);
    ...
    @(posedge Clk_s);
    #5 Rst_s <= 0;
    I_s <= 4'b1111;
end
```

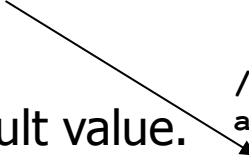
```
// Vector Procedure
always begin
    Rst_s <= 1;
    I_s <= 4'b0000;
end
```



# Common Pitfalls

10

- Not initializing all module inputs
  - ▣ May cause undefined outputs
  - ▣ Or simulator may initialize to default value. Switching simulators may cause design to fail.
  - ▣ *Tip:* Immediately initialize all module inputs when first writing procedure



```
// Vector Procedure
always begin
Rst_s <= 1;
  I_s <= 4'b0000;
  @(posedge Clk_s);

  ...
  @(posedge Clk_s);
  #5 Rst_s <= 0;
  I_s <= 4'b1111;
end
```

# Common Pitfalls

11

- Forgetting to explicitly declare as a wire an identifier used in a port connection
  - e.g., Q\_s
  - Verilog **implicitly declares** identifier as a net of the default net type, typically a *one-bit* wire
    - Intended as shortcut to save typing for large circuits
    - May not give warning message during compilation
    - Works fine if a one-bit wire was desired
    - But may be mismatch – in this example, the wire should have been four bits, not one bit
    - Unexpected simulation results
  - Always explicitly declare wires
    - Best to avoid use of Verilog's implicit declaration shortcut

```
`timescale 1 ns/1 ns

module Testbench();

    reg [3:0] I_s;
    reg Clk_s, Rst_s;
wire [3:0] Q_s;

    Reg4 CompToTest(I_s, Q_s, Clk_s, Rst_s);

    ...
endmodule
```

# 4-Bit Adder

12

- 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

13

- "+" 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 with Carry-In and Carry-Out

14

- 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

15

- 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

16

### 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

17

- $A5 = \{1'b0, A\} \rightarrow$  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;

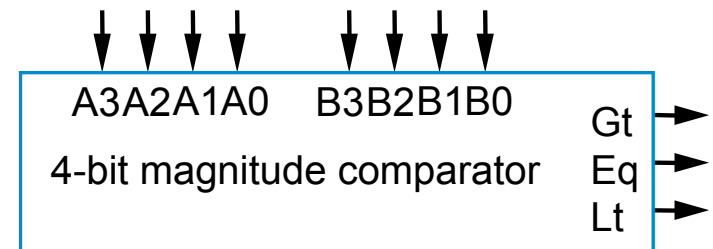
    reg [4:0] A5, B5, S5;

    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 Unsigned/Signed Magnitude Comparator

18

- 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);

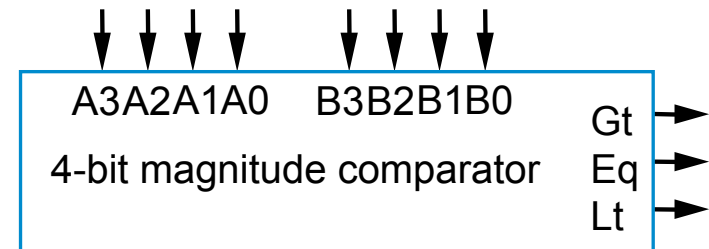
    input [3:0] A;
    input signed [3:0] B;
    output Gt, Eq, Lt;
    reg Gt, Eq, Lt;

    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
end
endmodule
```

# 4-bit Unsigned/Signed Magnitude Comparator

19

- 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
end
endmodule
```