Modeling Combinational Logic: 2

Modeling Combinational Logic

- SV has 3 ways to represent combinational logic at a synthesizable RTL level:
  - continuous assignments, always procedures, and functions.
- Details:
  - Continuous assignment statements
  - **always** procedures, with strict coding guidelines
  - **always_comb** procedure with simulation rules
  - **always @*** procedure
  - Using functions to model combinational logic

1. Continuous Assignments

- A continuous assignment drives an expression or the result of an operation onto a net (Verilog) or a variable (SV).
- An explicit continuous assignment is a statement that begins with the **assign** keyword.
  - **assign** sum = a + b;

assign sum = a + b;

- The left-hand side of the assignment is updated
  - whenever any change of value occurs on the right-hand side,
  - which is whenever a or b changes.
- This continuous updating of the left-hand side whenever the right-hand side changes is what models the behavior of combinational logic.

Delays

- The continuous assignment syntax allows for a propagation delay to be specified between
  - when a change on the right-hand side occurs and
  - when the left-hand side is updated.
- Synthesis compilers expect zero-delay RTL models
  - will ignore delays in continuous assignments.
- This can lead to a mismatch between a design that was verified with delays, and the synthesized implementation that ignored the delays.

Left-side Types (SV)

Left-hand Side =

- The left-hand side of a continuous assignment can be
  - a scalar (1-bit) or vector net or
  - a variable type, or
  - a user-defined type.

Net =

Left-hand Side =

- Net objects : explicitly declared by **wire** or **tri**
  - can be driven by multiple sources, including
    - multiple continuous assignments,
    - multiple connections to output or inout ports of module or
    - primitive instances, or
    - any combination of drivers.

Variable =

- Variable objects :
  - explicitly declared by type **var**
  - can only be assigned a value from a single source,
  - Variables can be:
    - a single input port,
    - a single continuous assignment, or
    - any number of procedural assignments in a procedure
      - multiple procedural assignments are considered to be a single source;
    - synthesis requires that the multiple procedural assignments be in the same procedure.

Best Practice Guideline

- *Use variables on the left-hand side of continuous assignments to prevent unintentional multiple drivers.*
- *Only use wire or tri nets on the left-hand side when it is intended for a signal to have multiple drivers.*

- Only use a net kind (such as wire or tri) when multiple drivers are intended, such as for a shared bus, a tri-state bus or an inout bidirectional module port.

Variables Imply A Only Single Source

- For RTL modeling, an important semantic rule that
  - variables can only have a single source.
- If we write a model using variables of type logic,
  - we must ensure that two models do not attempt to put a value onto the same variable.
- With variable types, a multiple-source coding mistake will be reported as a compilation or elaboration error in both simulation and synthesis.

Vector Width Mismatches

- The left-hand side of a continuous assignment can be a different vector size than the signal or expression result on the right-hand side.
- When this occurs, SystemVerilog automatically adjusts the vector width of the right-hand side to match the size of the left-hand side.
  - If the right-hand side is a larger vector than the left-hand side, the most-significant bits of the right-hand side will be truncated to the size of the left-hand side.
  - If the right-hand side is a smaller vector size, the right-hand side value will be left-extended to the size of the left-hand side.
  - The left extension will extend with zeros if the expression or operation result is unsigned.
  - Sign extension will be used if the right-hand expression or operation result is signed.

Vector Width Mismatches

```
logic [3:0] smallLogic;
logic [5:0] bigLogic;
 ...
smallLogic = 4’b1000;
bigLogic = smallLogic; //result will be 6'b001000

signed logic [3:0] smallSignedLogic;
signed logic [5:0] bigSignedLogic;
 ...
smallSignedLogic = 4'b1000;
bigSignedLogic = smallSignedLogic; // 6'b111000
as the sign is extended (copied) into the two extra bits
smallSignedLogic = 4’b0100;
bigSignedLogic = smallSignedLogic; // 6’b000100
as the sign is extended (copied) into the two extra bits
```

- If a small vector is assigned to a larger vector the resulting value depends on whether the vectors are signed or not.
- If the vectors are unsigned, zeros will be padded on the left to fill in the extra bits.
- If the vectors are signed, the sign bit (the most significant bit) will be copied into all of the extra bits.

Best Practice Guideline

- *Ensure that both sides of continuous assignments and procedural assignments are the same vector width.*
- *Avoid mis-matched vector sizes on the left-hand and right-hand side expressions.*

Explicit and Inferred Continuous Assignments

- An explicit continuous assignment is declared with the **assign** keyword,.
  - This form of continuous assignment can assign to both net and variable types.

- An implicit net declaration continuous assignment combines the declaration of a net kind with a continuous assignment.
  - The continuous nature of this form is inferred, even though the assign keyword is not used.
  - wire [7:0] sum = a + b;

Different Initializations

- An inferred net declaration assignment is:
  - wire [7:0] sum = a + b;
- A variable declaration assignment:
  - int i = 5;

- A variable initialization is only executed one time
- An inferred net declaration assignment is *a process that updates the left-hand net whenever there is a change of value on the right-hand expression*.
  - An inferred net declaration assignment is synthesizable.

Multiple Continuous Assignments

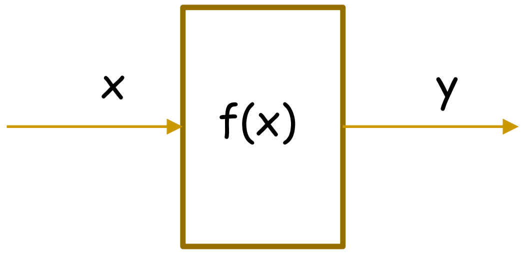
- A module can contain any number of continuous assignments.
- Each continuous assignment is a separate process that runs in parallel with other continuous assignments.
- All continuous assignments begin evaluating the right-hand side at simulation time zero, and run to the end of simulation.

Data Flow Model

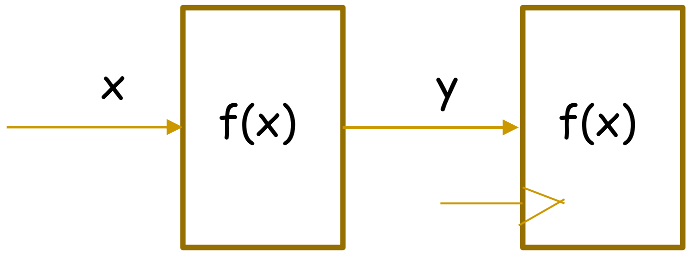
```
module dataflow
#(parameter N =4 )            //bus size
(input logic clk // scalar input
 input logic [N-1:0]  a, b, c,   // scalable input size
 input logic [1,0] factor,       // fixed input size
 output logic [N-1:0] out        // scalable output size
 );
logic [N-1:0] sum, diff, prod;

assign sum = a + b;
assign diff = prod - c;
assign prod = sum * factor;

always_ff @(posedge elk)
   out <= diff;

endmodule: dataflow
```

- Multiple procedural assignments in a module can be used to represent a dataflow behavior.
- In RTL models, dataflow assignments represent the combinational logic through which data flows between registers.
- Because multiple continuous assignments in a module run in parallel, the order of the assignments in the RTL source code makes no difference.

Both Continuous Assignments and Always Procedures



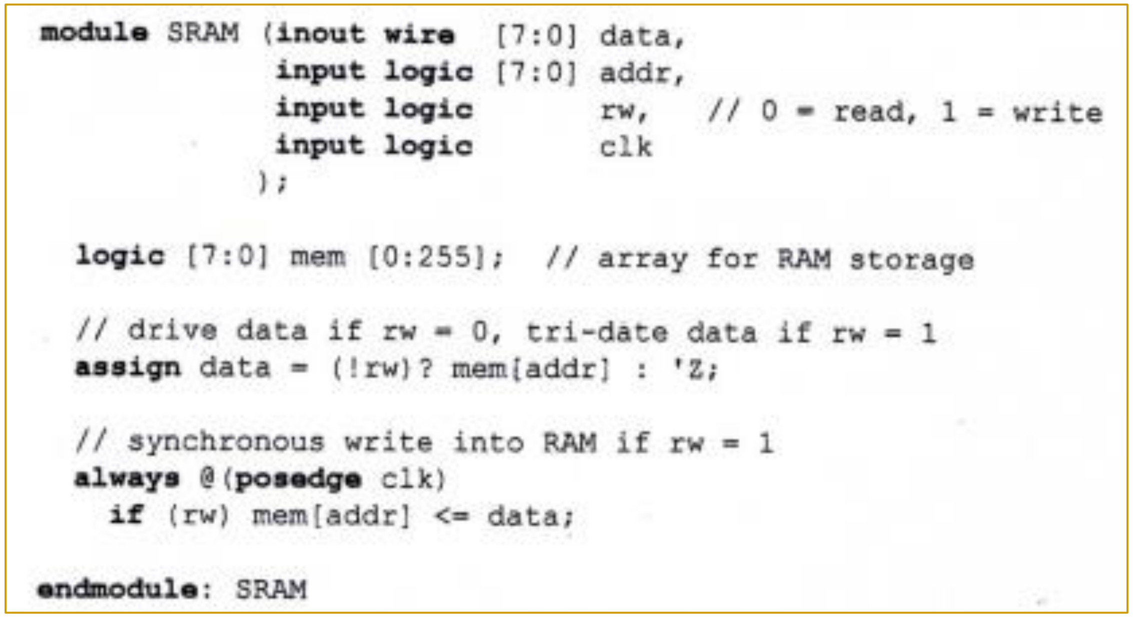
assign y=f(x)



```
always @....
f(x)
```

- A module can contain a mix of continuous assignments and always procedures.
  - A continuous assignment is used to model the output functionality.
  - An always procedure is used to model the input functionality in order to trigger on rising edges of the clock.

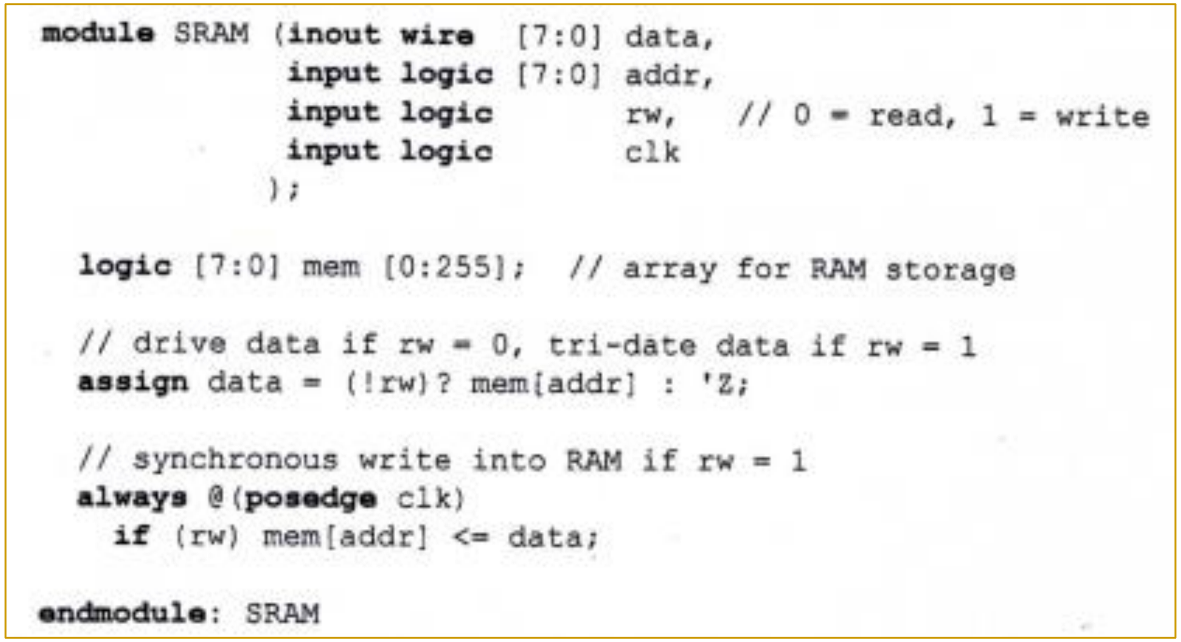
Both Continuous Assignments and Always Procedures



```
module SRAM (inout wire  [7:0] data,
             input logic [7:0] addr,
             input logic       rw,   // 0 = read, 1 = write
             input logic       clk
            );

  logic [7:0] mem [0:255];  // array for RAM storage

  // drive data if rw = 0, tri-date data if rw = 1
  assign data = (!rw)? mem[addr] : 'Z;

  // synchronous write into RAM if rw = 1
  always @(posedge clk)
    if (rw) mem[addr] <= data;

endmodule: SRAM
```

- The data bus is a bidirectional inout port, and must be a net kind, such as wire or tri, in order to have multiple drivers.
- The data bus can be driven by the RAM when it is an output from the RAM, and by some other module when data bus is an input writing into the RAM.
- Only continuous assignment can assign to net data objects.

Both Continuous Assignments and Always Procedures



```
module SRAM (inout wire  [7:0] data,
             input logic [7:0] addr,
             input logic       rw,   // 0 = read, 1 = write
             input logic       clk
            );

  logic [7:0] mem [0:255];   // array for RAM storage

  // drive data if rw = 0, tri-date data if rw = 1
  assign data = (!rw)? mem[addr] : 'Z;

  // synchronous write into RAM if rw = 1
  always @(posedge clk)
    if (rw) mem[addr] <= data;

endmodule: SRAM
```

- Each continuous assignment and each always procedure is a separate process that runs in parallel, beginning at simulation time zero and running throughout simulation.
- The order of continuous assignments and always procedures within a module does not matter because the processes are running in parallel.

2. The **always** Procedures

- The primary RTL modeling construct for combinational logic is the **always** procedure:
  - the general purpose **always**
  - the RTL-specific **always_comb**
- These **always** procedures can take advantage of
  - the **robust set** of operators and programming statements,
  - whereas continuous assignments are limited to using only SV operators.

always & always_comb

- Both always and always_comb procedures are supported by synthesis compilers.

```
always @(a, b) begin
sum = a + b;
end
```

```
always_comb begin
sum = a + b;
end
```

Coding Restrictions for Synthesizing **always**

- When using the general purpose **always** procedure,
  - synthesis compilers impose several coding restrictions that the RTL design engineer must be aware of and adhere to.

Coding Restrictions for Synthesizing always

- Some Restrictions:
  - The procedure sensitivity list should include every signal for which the value can affect the output(s) of the combinational logic.
  - The procedure sensitivity list must be sensitive to all possible value changes of each signal.
  - It cannot contain posedge or negedge keywords.
  - The procedure should execute in zero simulation time.
  - A variable which assigned a value in a combinational logic procedure should not be assigned a value by any other procedure or continuous assignment.

Best Practice Guideline

- Use the RTL-specific **always_comb** to model combinational logic.
- Do not use the generic **always** procedure in RTL models.

- The RTL-specific **always_comb** automatically enforces the coding restrictions listed above.
  - *The sensitivity is inferred,*
  - *a variable assigned in an **always_comb** procedure cannot be assigned by any other procedure or continuous assignment.*

Modeling with **always**

- Though not recommended for RTL modeling, it is common to see this general purpose **always** procedure in legacy Verilog models.

Sensitivity Lists

```
always @(a, b, mode) begin
 if (!mode) result =a+b;
         else result = a-b;
end
```

- The general always requires a sensitivity list to tell simulators.
- A sensitivity list is @(list_of_signals).
- Each signal in the sensitivity list can be separated by a comma, or by the keyword **or**, as in: @ (a or b or mode).
- Some engineers prefer the comma-separated list as the **or** keyword could be mistaken as a logical-OR operation, rather than just a separator between signals in the list.

Complete Sensitivity Lists

```
always @(a, b, mode) begin
 if (!mode) result =a+b;
         else result = a-b;
end
```

- With combinational logic, the outputs of the combinational block are a direct reflection of the current values of the inputs to that block.
- To model this behavior, the always procedure needs to execute its programming statements whenever any signal changes value that affects the outputs of the procedure.
- An input to the combinational always procedure is any signal of which the value is read by the statements in the procedure.
- The inputs to the procedure, the signals that are read within the procedure, are: a, b and mode.

Incomplete Sensitivity Lists: A Modeling Gotcha

- A gotcha is a programming term for code that is syntactically legal, but which does not perform as expected.
- The always procedure allows making this type of coding mistake.
- If one or more inputs to the combinational logic procedure are inadvertently omitted from the sensitivity list,
  - the RTL model will compile, and might even appear to simulate correctly.
- Thorough verification would show, however, that there are periods of time when the output(s) of the combinational logic block are not reflecting a combination of the current input values.

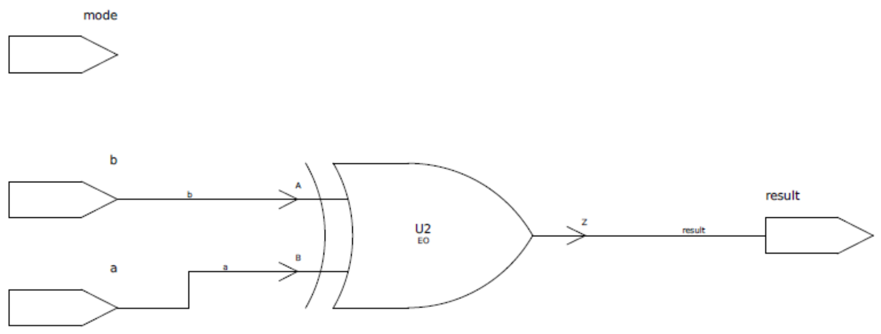
Incomplete Sensitivity Lists

```
always @(a, b) begin
if (!mode) result = a + b; // add when mode = 0
      else   result = a - b;
end
```

- If mode changes value, the result output will not be updated to the new operation result until either a or b changes value.
- The value of result is incorrect during the time between when mode changed and a or b changed.
- It is easy to inadvertently omit a signal in the sensitivity list when so many signals are involved.
- It also common to modify an always block during the development of a design, adding another signal to the logic, but forgetting to add it to the sensitivity list.

Incomplete Sensitivity Lists

```
always @(a, b) begin
if (!mode) result = a + b;
      else   result = a - b;
end
```



- A serious hazard with this coding gotcha is that many synthesis compilers will still implement this incorrect RTL model as gate-level combinational logic, possibly with a warning message that is easy to overlook.
- Though the implementation from synthesis might be what the designer intended, it is not the design functionality that was verified during RTL simulation.

The Obsolete **always @***

```
always @* begin
if (!mode) result = a + b;
        else   result - a - b;
end
```

- The IEEE 1364-2001 standard (Verilog-2001) attempted to address the gotcha of incomplete sensitivity lists with the addition of a special token:
- @* or @ (*) automatically infers a complete sensitivity list.
- The @* token offers a better coding style than explicitly listing signals in a combinational logic sensitivity list.

always @*

- Verilog-2001 added the always @* construct to automatically infer a complete sensitivity list,
  - but this construct is not perfect, and,
  - in some corner cases, simulation and synthesis infer different lists.

- The SystemVerilog always_comb procedural block has very specific rules that ensure that
  - all software tools will infer the same, accurate combinatorial sensitivity list.

The Obsolete always @*

```
always @* begin
if (!mode) result = a + b;
      else   result - a - b;
end
```

- Two subtle problems:
- 1) synthesis compilers impose some restrictions on modeling combinational logic.
  - Using @* infers a sensitivity list, but does not enforce other restrictions for modeling combinational logic.
- 2) a corner case where a complete sensitivity list is not inferred.
  - If a combinational logic procedure calls a function, but does not pass in as function arguments all signals used within the function, an incomplete sensitivity list will be inferred.

Cannot Infer a Complete Sensitivity List

```
@0: a = 0, b = 0, c = 0, always_d = 0, always_com_d = 0
@10: a = 1, b = 0, c = 0, always_d = 0, always_com_d = 1
@20: a = 1, b = 1, c = 0, always_d = 0, always_com_d = 1
@30: a = 1, b = 1, c = 1, always_d = 1, always_com_d = 1
```

```
module test;
logic a, b, c, always_d, always_comb_d;
function logic my_func(input logic m_c);
    my_func = a | b | m_c;
endfunction
always @*
    always_d = my_func(c);
always_comb
    always_comb_d = my_func(c);
initial begin
    $monitor("@%0t: a = %d, b = %d, c = %d, always_d = %d,
     always_com_d = %d", $time, a, b, c, always_d, always_comb_d);
end
```

- Notice that change on a, b does not trigger the always @* block to be reevaluated, but change on c does.

```
 initial begin
   a = 0;
   b = 0;
   c = 0;
   #10 a = 1;
   #10 b = 1;
   #10 c = 1;
 end
endmodule
```

Best Practice Guideline

- *Use the SV always_comb to automatically infer correct combinational logic sensitivity lists.*
- *Do not use the obsolete @* inferred sensitivity list.*

- An always comb procedure will infer an accurate sensitivity list without the hazards of explicit lists or the corner-case problem of @*.

Issues with the General **always**

- The original Verilog language that was introduced in the 1980s only had the general purpose **always** procedure.
- As a general purpose procedure, **always** can be used to model
  - combinational logic, sequential logic, latched logic and various verification processes.
- When a synthesis compiler encounters an **always** procedure,
  - the compiler has no way to know what type of functionality a design engineer intended to model.
- Instead, a synthesis compiler must analyze the contents of the procedure and try to infer a designer's intent.
  - It is all too possible for synthesis to infer a different type of functionality than what an engineer intended.

Issues with the General **always**

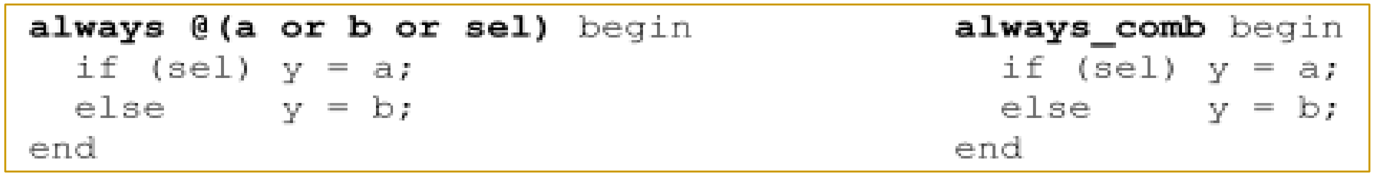
- Another limitation:
  - it does not enforce RTL coding restrictions required by synthesis compilers for representing combinational logic behavior.
- Models using general purpose always procedures might appear to simulate correctly, but might not synthesize to the intended functionality,
  - resulting in lost engineering time by having to rewrite the RTL models and reverify the functionality in simulation before the model can be synthesized.

Modeling with **always_comb**

- SystemVerilog introduced the RTL specific always procedures, such as **always_comb**, to address the limitations of the general purpose always procedure.

```
always_comb begin
if (!mode) result = a + b;
      else   result - a - b;
end
```

always_comb



```
always @(a or b or sel) begin
  if (sel) y = a;
  else     y = b;
end
```

```
always_comb begin
  if (sel) y = a;
  else     y = b;
end
```

- The designer's intent is to represent combinational logic.
- Using always_comb has several major benefits over the generic Verilog always procedure.

always_comb: The First Benefit

```
always @(a or b or sel) begin
  if (sel) y = a;
  else     y = b;
end
```

```
always_comb begin
  if (sel) y = a;
  else     y = b;
end
```

- The first benefit:
  - Tools can infer a combinatorial sensitivity list,
    - because tools know the intent of the procedural block.

always_comb: The First Benefit

```
always @(a or b or sel) begin          always_comb begin
  if (sel) y = a;                        if (sel) y = a;
  else     y = b;                        else     y = b;
end                                    end
```

- A common coding mistake with traditional Verilog is to have an incomplete sensitivity list.
- This is not a syntax error, and
  - results in RTL simulation behaving differently than the post-synthesis gate-level behavior.
- Inadvertently leaving out one signal in the sensitivity list is an easy mistake to make in large, complex decoding logic.
- It is an error that will likely be caught during synthesis,
  - but that only comes after many hours of simulation time have been invested in the project. (Too late !)

always_comb Benefits

- A complete sensitivity list is automatically inferred.
  - This list is fully complete, and avoids the corner case where @* would infer an incomplete sensitivity list.
- Using #, @ or wait to delay execution of a statement in an always_comb procedure is not permitted,
  - enforcing the synthesis guideline for using zero-delay procedures.
  - Using these time controls in always_comb is an error that will be caught during the compilation and elaboration of the RTL models.
- Any variable assigned a value in an always_comb procedure cannot be assigned from another procedure or continuous assignment, which is a restriction required by synthesis compilers.
  - A coding mistake that violates this synthesis restriction will be caught during compilation and elaboration of the RTL models.

always: An RTL Simulation Glitch

- With a general purpose always procedure, a value change must occur to a signal in the sensitivity list in order to trigger an execution of the assignment statements within the procedure.
- If none of the signals in the sensitivity list change value at the start of simulation,
  - the outputs of the combinational logic procedure are not updated to match the values of the inputs to the procedure at the start of simulation.
- The **always** procedure will continue to have incorrect output values until a signal in the sensitivity list changes value.

always_comb: Automatic Evaluation at the Start

- The RTL-specific **always_comb** resolves this simulation glitch.
- An **always_comb** procedure automatically triggers once at the start of simulation,
  - to ensure that all variables assigned in the procedure accurately reflect the values of the inputs to the procedure at simulation time zero.

Using Blocking Assignments

- Only use blocking assignments (=) when modeling combinational logic behavior.

- The blocking assignment (=) immediately updates the variable on the left-hand side,
  - allowing the new value to be available for use by any subsequent statement in a begin-end sequence of statements.
- The immediate update effectively models the behavior of value propagation in combinational logic dataflow.

Using Blocking Assignments

```
always_comb begin
sum = a + b;
prod = sum * factor;
result = prod - c; end
```

- The combinational logic dataflow through multiple assignments in a combinational logic procedural block.
- The variable sum is immediately updated to the result of the operation a + b.
- This new value of sum then flows to the next statement, where the new value is used to calculate a new value for prod.
- This new value for prod then flows to the next line of code and is used to calculate the value of result.

Using Blocking Assignments

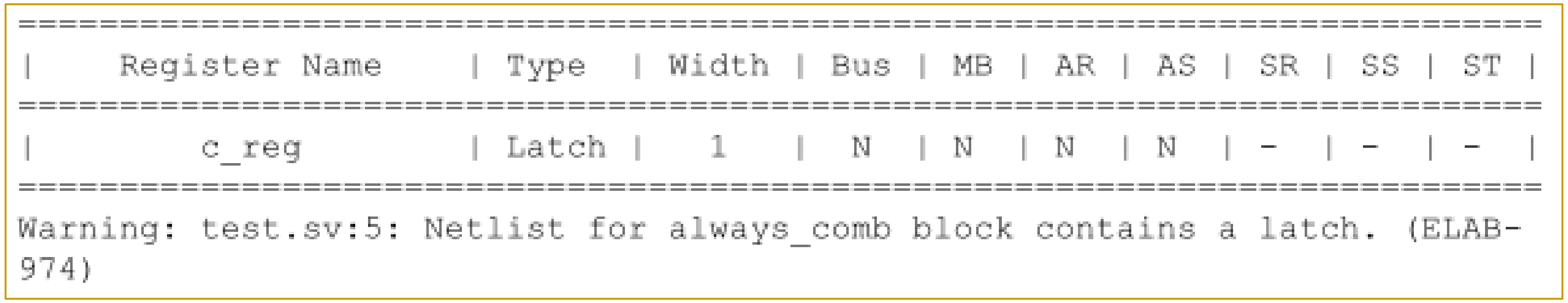
```
always_comb begin
sum = a + b;
prod = sum * factor;
result = prod - c; end
```

- The blocking assignment in each line of code blocks the evaluation of the next line,
  - until the current line has updated its left-hand side variable with a new value.
- The blocking of the evaluation of each subsequent line of code is what ensures that each line is using the new value of variables assigned by the previous lines.

always_comb: The Second Benefit

```
module always_comb_test
      (input logic a, b,
         output logic c);
         always_comb
            if (a) c = b;
endmodule: always_comb_test
```

- As tools know the intent is combination logic, tools can verify that this intent is being met.
- The example does not correctly behave as combinational logic.
- DC issues warning and generates a "Register table" indicating the register type as "Latch".
- Additionally, an elaboration warning was issued.



```
===============================================================================
|     Register Name      | Type  | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|         c_reg          | Latch |   1   |  N  | N  | N  | N  | -  | -  | -  |
===============================================================================
Warning: test.sv:5: Netlist for always_comb block contains a latch. (ELAB-
974)
```

Avoiding Unintentional Latches

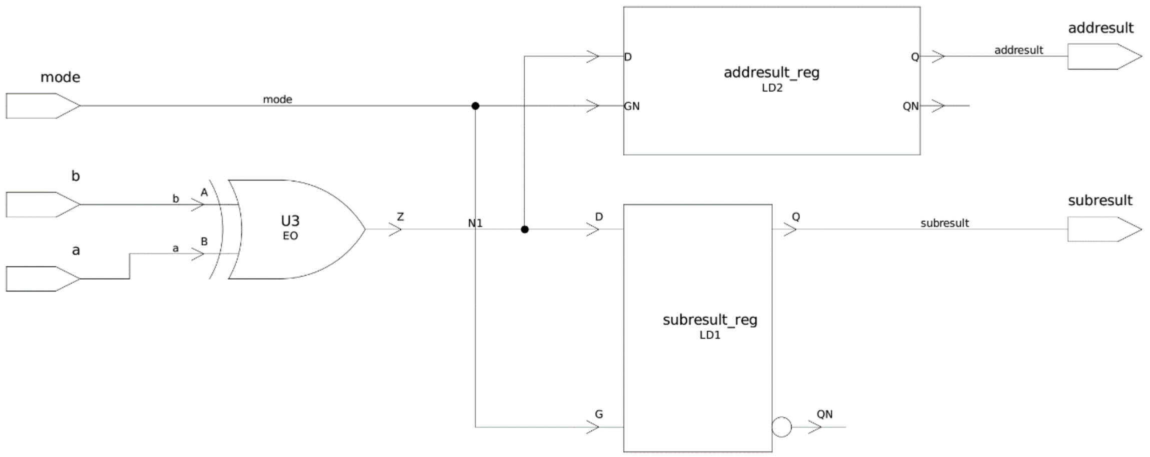
- A common problem in RTL modeling is the inference of latch behavior in code that is intended to represent combinational logic behavior.
- SV language rules require that
  - the left-hand side of procedural assignments must be some type of variable.
  - Net data types are not permitted on the left-hand side of procedural assignments.
- This requirement to use variables can lead to inadvertent latches, where pure combinational logic was intended.
- Latch behavior occurs
  - when a non-clocked always procedure is triggered, and no assignment is made to the variables used by the procedure.

Avoiding Unintentional Latches

```
always_comb begin
if (!mode) add_result = a + b;
else  subtract_result = a - b;
end
```

- 1. A decision statement assigns to different variables in each branch.

Avoiding Unintentional Latches



```
always_comb begin
if (!mode) add_result = a + b;
else  subtract_result = a - b;
end
```

- 1. A decision statement assigns to different variables in each branch.

Avoiding Unintentional Latches

- 2. A decision statement does not execute a branch for every possible value of the decision expression.

```
always_comb begin
case (opcode)
2'b00: result = a + b;
2'b01: result = a - b;
2'b10: result = a * b;
endcase end
```

- If, however, the opcode input should have a value of 2'b11, this example does not make any assignment to the result variable.
- Because result is a variable, it retains its previous value.
- The retention of a value behaves as a latch, even though the intent is that the always_comb procedural behave as combinational logic.

Avoiding Unintentional Latches

```
always_comb begin
case (opcode)
2'b00: result = a + b;
2'b01: result = a - b;
2'b10: result = a * b;
endcase end
```

- A latch will be inferred even when an always_comb procedure is used.
- Synthesis compilers and lint checkers, will however, report a warning or non-fatal error that a latch was inferred in an always_comb procedure.
- This warning is one of the several advantages of always_comb over a general always procedures.
- An always_comb procedure documents the design engineers intent, allowing software tools to report when the code within the procedure does not match that intent.

3. Using Functions for Combinational Logic

- A function return value must be calculated each time the function is called.
- All functions used in RTL models as automatic functions.

```
module algorithmic_multiplier
import definitions_pkg ::*
(input logic [3:0] a, b,
 output logic [7:0] result );
assign result = multiply_f(a, b);
endmodule: algorithmic_ multiplier
```

```
package definitions_pkg;
function automatic [7:0] multiply_f([7:0] a, b);
multiply_f =0;
for (int i=0; i<=3; i++) begin
if (b == 0) continue; // all done, finish looping
else begin
if (b & 1) multiply_f += a; // function name is a var
a <<= 1; // multiply by 2 by shifting left 1 time
b >>= 1; // divide by 2 by shifting left 1 time
and
end
endfunction
endpackage: definitions_pkg
```

Best Practice Guideline

- *When possible, use SystemVerilog operators for complex operations such as multiplication, rather than using loops and other programming statements.*

- It is preferable to use SystemVerilog operators for complex operations such as multiply and divide.
- If the multiply operator (*) had been used, synthesis compilers could map the operator to the most efficient multiplier implementation for a specific target ASIC or FPGA.

Summary

- Combinational logic:
  - The output values are always representing a combination of the input values.
  - If any input changes value, the output is updated to reflect this change.
- Three SV modeling constructs:
  - continuous assignments, always procedures, and functions.

Summary: Continuous Assignments

- The assign continuous assignment statement is a simple way to model combinational logic.
- A continuous assignment is automatically re-evaluated any time the value of a net or variable on the right-hand side of the assignment changes.
- Continuous assignments can use any synthesizable SystemVerilog operator, but cannot use programming statements.
- Continuous assignments can have function calls on the right-hand side of the assignment, and the function can use programming statements.

Summary: **always**

- **always** procedures are an infinite loop.
- To modeling combinational logic behavior, an **always** procedures must start with a sensitivity list that triggers on a change of every net or variable that can affect the outputs of the combinational logic.
- The **always** procedure is a general purpose procedure that can be used to model
  - combinational logic, sequential logic, latched logic, verification loops, and other behaviors.
- A general purpose always procedure requires that
  - the design engineer explicitly define an accurate sensitivity list, and follow strict coding guidelines in order for an always procedure to synthesize as combinational logic.
- Engineers can easily make mistakes with the always general purpose procedure that seem to simulate OK, but do not synthesize as intended.

Summary: **always_comb**

- SV added the always_comb procedure to the original Verilog language to address these issues.
- An always_comb procedure automatically infers a correct sensitivity list, and
  - syntactically or semantically requires adherence to several synthesis restrictions.

References

- RTL Modeling with SystemVerilog for Simulation and Synthesis by Stuart Sutherland
- The Art of Hardware Architecture: Design Methods and Techniques for Digital Circuits by Mohit Arora. Springer.
- http://www.eetimes.com/document.asp?doc_id=1274581
- https://forums.xilinx.com/t5/BRAM-FIFO/FIFO-with-non-power-of-two-depth/td-p/482158
- http://www.verilogpro.com/asynchronous-fifo-design/