CME 341: Introduction to Behavioral Verilog

Brian Berscheid

Department of Electrical and Computer Engineering
University of Saskatchewan

Today's agenda

Introduction to Behavioral Verilog

Behavioral coding

- Alternative to structural coding: specify the *behavior* of the circuit
  - ▷ How does the behavior of the output signals depend on the input signals?
- Synthesizer translates description of behavior into individual gates + connections
- Very powerful; can represent large amount of logic with small amount of code
- We need to follow certain conventions to ensure the translation is done as intended
  - ▷ It is important to keep in mind how the synthesizer works and not just treat Verilog as a sequential programming language (like C, python, etc.)

Procedures

- Behavioral logic is built within *procedures*: blocks of code which specify two things
  - ▷ When to execute
  - ▷ What to do when executing
- In CME 341, we will study two main types of procedures:
  - ▷ `initial`: executes once at start of simulation (not normally synthesizable)
  - ▷ `always`: executes every time a certain condition is met
- `initial` procedures have already been seen in an example testbench
- Most of our efforts in CME 341 will be focused on `always` procedures

Procedures: key facts to remember

- Any variables being modified within a procedure must be defined as type `reg`, not `wire`!
  - ▷ Confusingly, type `reg` doesn't necessarily build a register in hardware; it is just a data type that allows behavioral assignment!
- Procedures are one statement long by default
  - ▷ Can include multiple statements if wrapped in `begin ... end`
- To set a variable to some expression, just use:

```
variable = expression; // ``assign'' not needed (or allowed)
```

- Verilog is a parallel language... procedures run in parallel!

Recall: examples of initial procedures in testbench code

```
reg n_set, n_clear; // define registers

initial  // runs once at simulation startup
begin
#10 n_set = 1'b0;   // schedule n_set to be 0 at t=10 ns
#10 n_set = 1'b1;   // schedule n_set to be 1 at t=10+10=20 ns
#55 n_set = 1'b1;   // schedule n_set to be 1 at t=20+55=75 ns
end

initial
#65 n_set = 1'b0;   // schedule n_set to be 0 at t=65 ns
```

Syntax of an always procedure

```
always @ (sensitivity_list)
   statement;

// OR

always @ (sensitivity_list)
   begin
      statement_1;
      statement_2;
      // etc...
   end
```

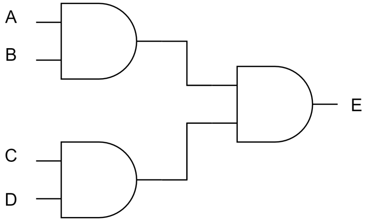
- Sensitivity list defines “when to do it?”
- Statement(s) define “what to do?”

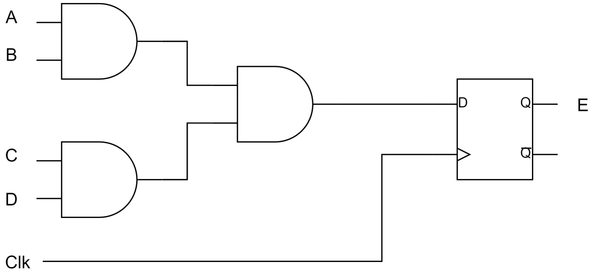
Combinational vs synchronous[1] logic

- `always` procedures can be used to build combinational or synchronous logic
- Synchronous: happening at the same time (from ancient Greek)
  - ▷ Refers to logic that operates based on a clock signal (typically at the positive edge)
  - ▷ Key element is a "flip-flop" or "register": passes signal from input to output on clock
- Combinational: logic which creates outputs based on combinations of inputs
  - ▷ No notion of time or clock signal involved
  - ▷ Standard logic gates: AND, OR, NAND, NOR, etc.
- Note that synchronous logic generally combines flip-flops with combinational logic gates

[1] Also known as sequential logic

Combinational vs synchronous logic





Combinational Logic

Synchronous Logic

Sensitivity lists for combinational vs synchronous logic

- The sensitivity list defines whether an `always` block generates synchronous or combinational logic
  - ▷ Combinational logic: has form `always @ (a or b or c)` or `always @ (*)`

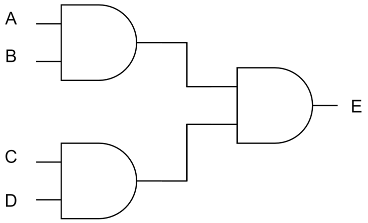
    Means "run procedure whenever a or b or c changes"

    * is a shortcut which automatically creates a sensitivity list containing all signals on RHS of assignments (strongly recommended)
  - ▷ Synchronous logic: has form `always @ (posedge clk)`

    Means "run procedure whenever a positive edge of clk occurs"
- We will study both types in CME 341, but will start with combinational logic
  - ▷ Many common rules and principles apply to both types

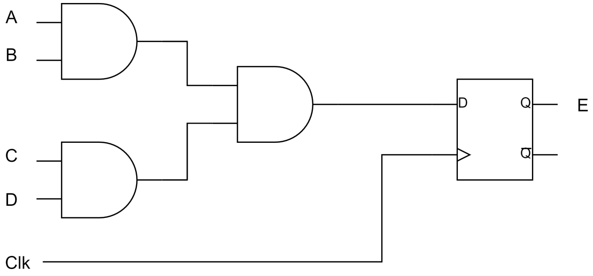
Example: combinational vs synchronous logic

```
always @ (*)
   E = (A & B & C & D);
```



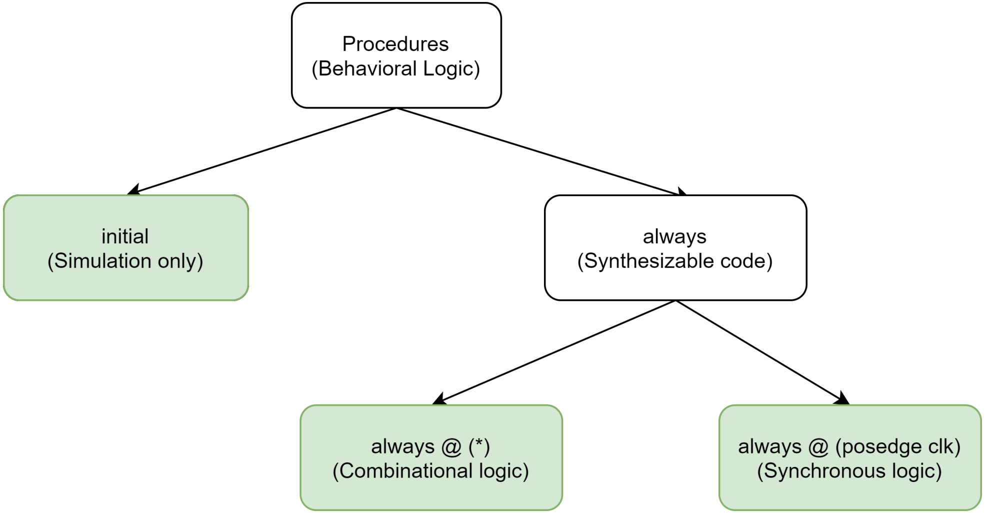
Combinational Logic

```
always @ (posedge Clk)
   E = (A & B & C & D);
```



Synchronous Logic

Summary: taxonomy of procedures



Synthesis templates

- Synthesizer parses the Verilog code looking for structures that match *templates*
  - ▷ Template: a specific coding structure that maps to hardware in a predicable way
- Templates are where the power of behavioral coding really becomes evident
- Understanding how code maps into hardware through templates is critical to digital design
- We will study two of the most important templates / coding constructs in the following sections:
  - ▷ `if ... else`
  - ▷ `case`

If-else statement

If-else statement: basics

- Represents hardware which takes one of two actions depending on the result of a test
- Similar structure to many programming languages, but don't forget that it is building hardware!
- Syntax:

```
always @ (*)
  if (conditional_test)     // no semicolon
     statement_if_true;
  else                      // no semicolon
     statement_if_false;
```

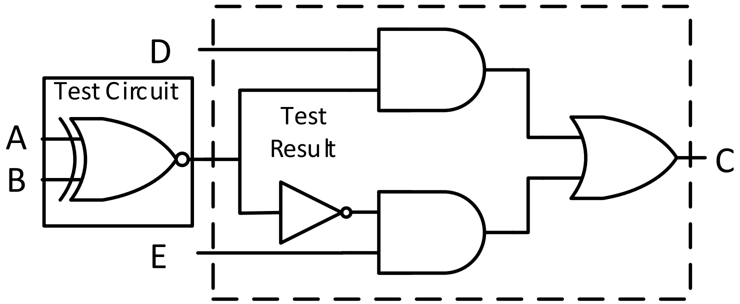
- Conditional test can be anything that evaluates to a single-bit (boolean) value:
  - ▷ Review operators section if necessary
  - ▷ Reduction operators and relational operators are commonly used here

If-else statement: example 1

```
reg C;

always @ (*)  // equivalent to (A or B or C or D)
  if (A == B)
    C = D;
  else
    C = E;
```

If-else statement: hardware for example 1

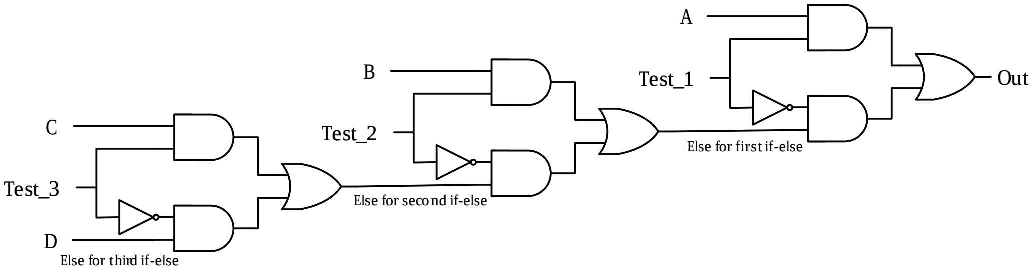


Nesting of if-else

- if-else statements can be nested to represent more complex logic
- Place another if statement in the else condition as shown below... can nest as deep as desired

```
always @ (*)
  if(Test_1)
    Out = A;
  else if(Test_2)
    Out = B;
  else if(Test_3)
    Out = C;
  else
    Out = D;
```

Hardware for nested if-else



Challenge

Try to write a Verilog module to solve the following problem

Generate a circuit which analyzes an 8-bit input value called price and generates a 2-bit output called flag whose value is chosen based on the value of price.

If the price is odd and greater than 13, flag should be set to 2. If the price is even, flag should be set to 0. Otherwise, flag should be set to 1.

Danger: latch warning!

- When writing if-else conditions, make sure all cases are covered (make sure you have an else for each statement)
  - ▷ It is legal but dangerous to omit the else, especially in a combinational circuit
  - ▷ If you omit the else, the compiler will add one for you which leaves the values at their previous states
  - ▷ This adds feedback to the circuit and risks creating a latch (if circuit is combinational)
  - ▷ Recall from previous discussions that latches can behave unpredictably and even go unstable
  - ▷ Quartus will produce a warning if this happens

- This is a VERY COMMON source of errors for people new to Verilog!

Inferred latch example

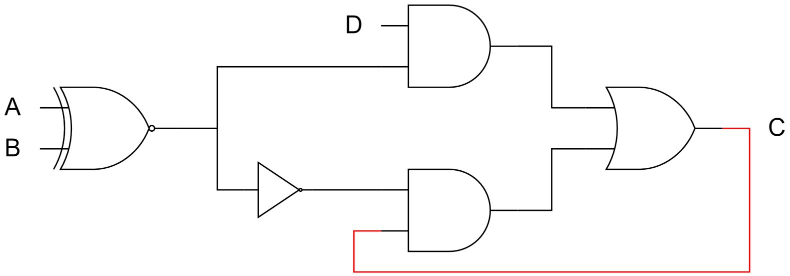
Suppose you write the following code:

```
always @ (*)
  if (A == B)
    C = D;
```

Inferred latch example

Compiler adds else case

```
always @ (*)
if (A == B)
  C = D;
else
  C = C;
```



Case statement

What is a case statement?

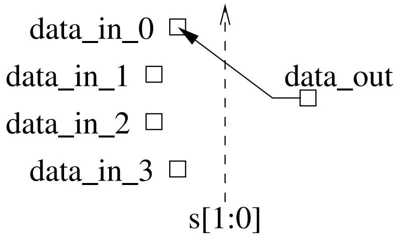
- A case statement analyzes an input vector and take one of multiple actions depending on the value of the input
- Like the if-else statement, it must reside within a procedure (combinational or synchronous)
- Similar to the if-else construct in functionality, but allows for more than 2 possibilities
- Synthesizes to an N-to-1 multiplexer hardware structure or "multi-throw switch"

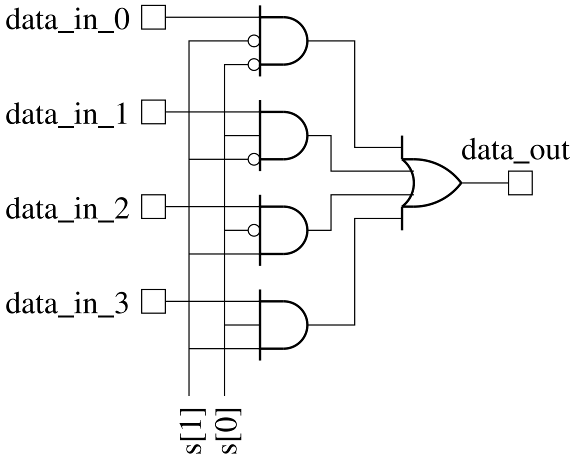
Case statement example

```
module data_selector (
  input wire data_in_0, data_in_1, data_in_2, data_in_3,
  input wire [1:0] s,
  output reg data_out
);

always @ *
  case(s)
    2'b00: data_out = data_in_0;
    2'b01: data_out = data_in_1;
    2'b10: data_out = data_in_2;
    2'b11: data_out = data_in_3;
  endcase
endmodule
```

Hardware representation of case statement





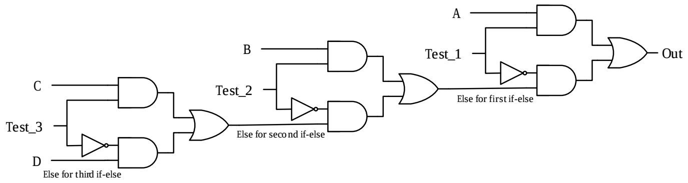
General case statement syntax

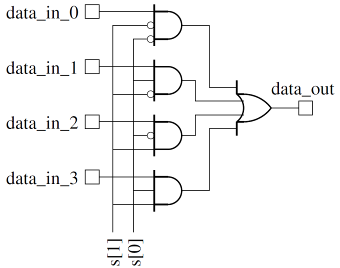
```
case (select_expression)
  alternative_1: statement_1;
  alternative_2: statement_2;
  // etc...
  default: statement_default;
endcase
```

Key points to note:

- `select_expression` generally a vector
- `alternative_#` is a value that the vector may take on
- Compiler checks whether `select_expression == alternative_#` is true. If so, `statement_#` will be executed.
- Only statement corresponding to matching alternative will be executed
- If none of the specified items match, `default` statement will execute

Compare and contrast hardware: case vs nested if-else





- Similar functionality is possible using the two structures
- Case structure is more “flat”, whereas if-else is more hierarchical
  - ▷ Can be an important distinction when we move to synchronous logic

Key points to keep in mind

- Make sure the select expression and case alternatives are same length
- Can include `begin... end` to execute multiple statements within an item
- Can specify multiple case expressions separated by commas to execute the same assignment for multiple conditions:

  ```
  3'b001, 3'b011:  a = b;
  ```

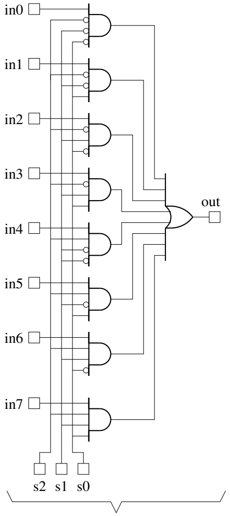
- If multiple items match, the first will be the one that is used
  - ▷ Don't rely on this! Make sure your case items are unique!

A second example

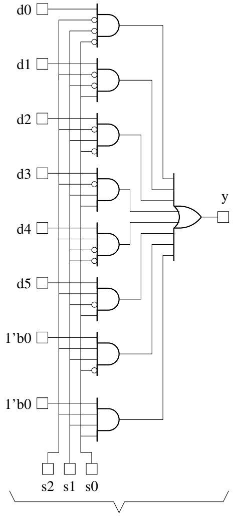
```
always @ *
  case( { s2, s1, s0 } )
    3'b010:  y = d2;
    3'b100:  y = d4;
    3'b101:  y = d5;
    3'b101:  y = d0; // this will be ignored
    3'b011:  y = d3;
    3'b000:  y = d0;
    3'b001:  y = d1;
    default: y = 1'b0;
  endcase
```

- Synthesis translator uses a template hardware structure for the case logic
- Actual case items and assignment values are plugged into the template

Hardware for second example



initial template used by translator



after translator is finished

casex statement

- Verilog includes a variation of the case statement known as 'casex', which handles don't care values (x) in the case items.
- When the compiler sees a don't care value, it expands it out into all of the possible values before proceeding with the template matching
- Be careful - it is easy to duplicate cases and create unintentional behavior in this way.

casex example

```
module data_selector (
  input a, b, c, d, e,
  input [2:0] sel,
  output reg y );

always @ *
  casex ( sel )
    3'b1xx:  y = a;
    3'bxx0:  y = b;
    3'b101:  y = c;
    3'b001:  y = d;
    3'b010:  y = e;
  endcase
endmodule
```

casex example - equivalent case statement

```
module data_selector (
  input a, b, c, d, e,
  input [2:0] sel,
  output reg y );

always @ *
  casex ( sel )
    3'b1xx:  y = a;
    3'bxx0:  y = b;
    3'b101:  y = c;
    3'b001:  y = d;
    3'b010:  y = e;
  endcase
endmodule
```

```
module data_selector (
  input a, b, c, d, e,
  input [2:0] sel,
  output reg y );

always @ *
  case ( sel )
    3'b100, 3'b101, 3'b110, 3'b111:  y = a;
    3'b000, 3'b010, 3'b100, 3'b110:  y = b;
    3'b101:  y = c;
    3'b001:  y = d;
    3'b010:  y = e;
  endcase
endmodule
```

Danger: latch warning! (part 2)

- If you don't specify all of the possibilities for the case select variable (and don't provide a default case), the synthesis tool won't know what to connect to the corresponding input.
- As in the if-else scenario, it is possible that the compiler will add a "trivial assignment" `out = out;` creating feedback and the potential for a combinational latch
  - ▷ Unpredictable behavior
  - ▷ Differences between simulation and hardware
- To avoid such problems, be sure to always include a default case!

Challenge

Try to write a Verilog module to solve the following problem

Generate a combinational circuit which accepts a 4-bit input called "value" and outputs a 3-bit value called "ones" which indicates the number of bits in "value" which are currently set to 1.

Thank you!
Have a great day!