ECE 551
System on Chip Design

Combinational Logic with SystemVerilog

Garrett S. Rose
Fall 2018
Outline

- Review continuous assignments and `always_comb` block
- Module instantiation and hierarchy
- Procedural modeling with `if-else` and `case` statements
- Built-in types – different integer types
- Enumerations and type definitions
assign and always_comb

- 2 ways to describe combinational logic procedurally: assign and always_comb
  - assign statements:
    - Great for one-line logic expressions
    - Evaluated like an always block: waits for change to any variable on right side

  ```
  // assign out_net = (condition) ? out_ifTrue : out_ifFalse;
  assign result = (select_plus) ? a + b : a - b;
  ```

  - always_comb blocks:
    - Essentially, a continuously looping statement
    - Waits at top of block for change to any right side variable in any assignment
    - Allows use of more complex procedural statements (e.g., case)

  ```
  always_comb // no begin since one if-else statement
  if (select_plus)
    result = a + b;
  else result = a - b;
  ```
Multiple assign Statements

- A single `assign` statement can be used for multiple logic expressions
- We’re sharing an `assign` statement in the example below
  - However, these are two separate `assign` statements!
  - The comma separates each statement while using the same `assign`
- Each `assign` statement is executed *concurrently*

```verilog
module compare
    (output logic eq, neq,
     input logic [3:0] value);

    assign neq = -eq,       // neq is the complement of eq
       eq  = (value == 0);  // eq evaluates TRUE when value==0

endmodule: compare
```
Module Instantiation

- Assume we gather the previous two examples into two base modules:

```verilog
module sum_and_dif
  (output logic [3:0] result,
   input logic [3:0] a, b,
   input logic select_plus);

  assign result = (select_plus) ? a + b : a - b;
endmodule: sum_and_dif

module compare
  (output logic eq, neq,
   input logic [3:0] value);

  assign neq = -eq,
         eq  = (value == 0);
endmodule: compare
```

- Note the style used in the above module description
The adder/subtractor and comparator modules are now instantiated in higher level module `add_sub_compare`:

```verilog
module add_sub_compare
output logic [3:0] result,
input logic [3:0] a, b,
output logic neq, eq,
input logic plus_minus);

sum_and_dif alu0(result, a, b, plus_minus);
compare cmp0(eq, neq, result);
endmodule: add_sub_compare
```

---

**Diagram:**

```
module add_sub_compare
  module sum_and_dif
    module compare
```

---

**Table:**

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>plus_minus</th>
<th>module sum_and_dif</th>
<th>module compare</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>eq</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>result</td>
</tr>
</tbody>
</table>
Avoid Incorrect Specifications

- The `always_comb` block is stateless – nothing is remembered
- Synthesis tools *should* check to make sure `always_comb` is stateless
- Take care to eliminate state in `always_comb` yourself
  - The block is sensitive to all input (right side) variables
  - Make sure all possible cases are covered – uncovered cases infer memory!

```vhdl
module notCombinational       // this is INCORRECT!
  (input  logic [3:0] a, b,
   output logic [3:0] sum,
   input  logic    hold);

always_comb
  if (~hold)                // the ‘~’ specifies NOT - OK
    sum = a + b;            // else condition not specified
endmodule: notCombinational
```
Procedural Modeling – if-else

- Consider the following Boolean expression as SystemVerilog assignment:
  
  assign f = (a & b) | (b & c) | (a & c);

- This implements the Boolean expression: \( f = a \cdot b + b \cdot c + a \cdot c \)

- Inside an `always_comb` block, one can use `if` statements

```systemverilog
// technically OK
module if1(
  input logic a, b, c,
  output logic f);

  always_comb begin
    f = 0;
    if (a & b) f = 1;
    if (c & (a ^ b)) f = 1;
  end
endmodule: if1

// safer – cases explicit
module if2(
  input logic a, b, c,
  output logic f);

  always_comb begin
    if (a & b) f = 1;
    else if (c & (a ^ b)) f = 1;
    else f = 0;
  end
endmodule: if2
```
Procedural Modeling – case

- Same example can be implemented using a case statement inside always_comb

```
module basicCase
  (input logic a, b, c,
   output logic f);

  always_comb begin
    case ({a, b, c})
      3'b000: f = 0;
      3'b001: f = 0;
      3'b010: f = 0;
      3'b011: f = 1;
      3'b100: f = 0;
      3'b101: f = 1;
      3'b110: f = 1;
      3'b111: f = 1;
    endcase
  end
endmodule: basicCase
```

```
// alternative – use default
always_comb begin
  case ({a, b, c})
    3'b000: f = 0;
    3'b001: f = 0;
    3'b010: f = 0;
    3'b011: f = 1;
    3'b100: f = 0;
    3'b101: f = 1;
    default: f = 1; // default case
  endcase
end
```

```
// alternative – combine cases
always_comb begin
  case ({a, b, c})
    3'b000, 3'b001: f = 0; // if 000 or 001
    3'b010: f = 0;
    3'b100: f = 0;
    default: f = 1; // default case
  endcase
end
```
Don’t Care Specifications – unique case

- Consider the following ALU description – not all cases are implemented
- Not a combinational circuit – uncovered cases suggest memory inference!

```verilog
module alu
  (input logic [7:0] a, b,
   output logic [7:0] result,
   input logic [2:0] op);

always_comb // oops!
  case (op)
    3’100: result = a + b; // add
    3’010: result = a - b; // sub
    3’001: result = a & b; // bitwise and
    3’110: result = a | b; // bitwise or
    3’011: result = a ^ b; // bitwise xor
  endcase
endmodule: alu
```
Don’t Care Specifications – unique case

- Remedy this issue using the **unique case**
- Synthesis tool recognizes only combinations it must implement are those listed
- The tool will implement any unlisted combinations as don’t cares
  - The result is still combinational logic!

```verbatim
module aluUnique
    (input logic [7:0] a, b,
    output logic [7:0] result,
    input logic [2:0] op);

always_comb
    unique case (op)
       3'100: result = a + b; // add
       3'010: result = a - b; // sub
       3'001: result = a & b; // bitwise and
       3'110: result = a | b; // bitwise or
       3'011: result = a ^ b; // bitwise xor
endcase
endmodule: aluUnique
```
Simplify Function Specifications – casez

- Sometimes we want to implement entire rows in a K-map on one line
- casez allows for ? wildcards, meaning the bit position is either 1 or 0

```verilog
module simpleKmap
  (input logic a, b, c,
   output logic f);

  always_comb
    casez ({a, b, c})
      3'1??: f = 0;
      3'01?: f = 1;
      3'000: f = 1;
      3'?01: f = 0;
    endcase
endmodule: simpleKmap
```
priority case

- Operation of `case` is to match *first* selection item equal to it
- Priority specifies to the tool that “at least one” item will match, maybe more
- If some combinations not specified, they’re treated as don’t cares

```vhdl
module pcaseA
  (output bit out,
   input bit a, b);

  always_comb
  priority casez ({a,b})
  2'b1?: out = 1'b0;  //1
  2'b?0: out = 1'b0;  //2
  2'b?1: out = 1'b1;  //3

endcase
endmodule: pcaseA
// if {a,b}=11, out=0
// line 1 executes first

module pcaseB
  (output bit out,
   input bit a, b);

  always_comb
  priority casez ({a,b})
  2'b?1: out = 1'b1;  //3
  2'b?0: out = 1'b0;  //2
  2'b1?: out = 1'b0;  //1

endcase
endmodule: pcaseB
// if {a,b}=11, out=1
// line 3 executes first - swapped
```
Example: A Priority Encoder

- Idea is that multiple devices may request access to a service simultaneously, but only one request can be granted at any time
- priority casez can be used to enforce priority order of requesting devices

```verilog
module prEncoder
    (input logic r0, r1, r2,
     output logic [2:0] gnt);
    logic [2:0] req;

    assign req = {r2, r1, r0}; // conc 3 request lines

    always_comb begin
        gnt = 0;
        priority casez (req)
            req[0]: gnt[0] = 1; // highest priority - serve 1st
            req[1]: gnt[1] = 1; // medium priority
            req[2]: gnt[2] = 1; // lowest priority
        endcase
    endmodule: prEncoder
```
Parameterized Modules

- Can include parameters in module definition – constants after compilation
- Upon compilation/synthesis, parameters either set to default or overridden when instance specifies new value for given parameter

```verilog
module aluParam
  #(parameter W = 8)
  (input logic [W-1:0] a, b,
   output logic [W-1:0] result,
   input logic op);

always_comb
  unique case (op)
    3'100: result = a + b; // add
    3'010: result = a - b; // sub
    3'001: result = a & b; // bitwise and
    3'110: result = a | b; // bitwise or
    3'011: result = a ^ b; // bitwise xor
  endcase
endmodule: aluParam
```
SystemVerilog Types – Example: Integers

<table>
<thead>
<tr>
<th>type name</th>
<th>2 or 4 state</th>
<th>size (bits)</th>
<th>signed/unsigned (default)</th>
<th>initial value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>shortint</td>
<td>2</td>
<td>16</td>
<td>signed</td>
<td>0</td>
<td>Same as short in C</td>
</tr>
<tr>
<td>int</td>
<td>2</td>
<td>32</td>
<td>signed</td>
<td>0</td>
<td>Same as int in C</td>
</tr>
<tr>
<td>longint</td>
<td>2</td>
<td>64</td>
<td>signed</td>
<td>0</td>
<td>Same as longint in C</td>
</tr>
<tr>
<td>bit</td>
<td>2</td>
<td>user-def</td>
<td>unsigned</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>byte</td>
<td>2</td>
<td>8</td>
<td>signed or ascii</td>
<td>0</td>
<td>Same as “signed bit [7:0] varName;”</td>
</tr>
<tr>
<td>logic</td>
<td>4</td>
<td>user-def</td>
<td>unsigned</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>reg</td>
<td>4</td>
<td>user-def</td>
<td>unsigned</td>
<td>X</td>
<td>Verilog type; superseded by logic</td>
</tr>
<tr>
<td>integer</td>
<td>4</td>
<td>32</td>
<td>signed</td>
<td>X</td>
<td>Not same as int in C, 4-state</td>
</tr>
<tr>
<td>time</td>
<td>4</td>
<td>64</td>
<td>unsigned only</td>
<td>0</td>
<td>Only used for testbenches/simulation</td>
</tr>
</tbody>
</table>

2-state: each bit can have one of two values (0, 1)
4-state: each bit can have one of four values (X, Z, 0, 1)
Enumerations and Type Definitions

- Enumerations list all possible values for a user defined variable type

```plaintext
enum {CHEVY, FORD, TOYOTA} car1, car2;
```

- An `enum` statement will default to type `int` upon compilation but the type after compile/synthesis can also be explicitly specified

```plaintext
module datapath_enum
    enum logic [2:0] {ADD,SUB,AND,OR,XOR} op;

    ...

    always_comb
    unique case (op)
        ADD: result = a + b; // add
        SUB: result = a - b; // sub
        AND: result = a & b; // bitwise and
        OR:  result = a | b; // bitwise or
        XOR: result = a ^ b; // bitwise xor
    endcase
endmodule: datapath_enum
```
New Data Types – typedef

typedef enum bit [2:0] {ADD=3'b100, SUB=3'b010, AND=3'b001, OR=3'b110, XOR=3'b011} aluInst_t;

module instrDecode
    (output aluInst_t decode, ...
endmodule: instrDecode

module alu
    (input logic [7:0] a, b,
     output logic [7:0] result,
     input aluInst_t op);

always_comb
    unique case (op)
        ADD: result = a + b; // add
        SUB: result = a - b; // sub
        AND: result = a & b; // bitwise and
        OR:  result = a | b; // bitwise or
        XOR: result = a ^ b; // bitwise xor
    endcase
endmodule: alu
New Data Types – ALU Example Continued

- Assume following top module continues from previous slide
- New data type aluInst_t is global to this SystemVerilog file

```verilog
top

module top;
    logic [7:0] inA, inB, aluOut;
    aluInst_t iDecode;

    ...

    instDecode id0(iDecode,...);
    alu a0(inA,inB,aluOut,iDecode);
endmodule: top
```
Structures

- A new data type can also be a `struct` – similar to C style `struct`
- In this example, `packed` makes all variables into one contiguous word/vector

```c
typedef enum logic [2:0] {
    ADD=3'b100,
    SUB=3'b010,
    AND=3'b001,
    OR=3'b110,
    XOR=3'b011
} aluInst_t;

// we'll use a struct to construct full command
// including operation and two operands
typedef struct packed {
    aluInst_t oper;
    logic [7:0] inA, inB;
} command_t;
```
Our ALU with Structure

- Access structure elements with '.' – similar to C

```vhdl
module instrDecode
    (output command_t inst,
     ...);
endmodule: instrDecode

module alu
    (input command_t c,       // inputs and operation
     output logic [7:0] result);

always_comb
    unique case (c.oper)            // access variable in 'c'
        ADD: result = c.inA + c.inB;  // add
        SUB: result = c.inA - c.inB;  // sub
        AND: result = c.inA & c.inB;  // bitwise and
        OR:  result = c.inA | c.inB;  // bitwise or
        XOR: result = c.inA ^ c.inB;  // bitwise xor
endcase
endmodule: alu
```