ECE4/530 Activity 01: Combinatorial Circuit Design

Introduction to combinatorial Verilog objects

01.1 Summary of Concurrent Statements

A Verilog design, in general consists of statements and procedural blocks that are executed concurrently. Three types of statements:

- continuous assignments
- always Block
- module instantiation

1. Continuous statements are useful for simple combinational circuits
   a. Syntax: `assign [signal name] = [expression]`
   b. Logic operators (bit-wise): `AND(&) ; OR (|) ; XOR (^) ; NOT (~)`
2. Always block for combinational circuits
   a. Syntax:

```
always @([sensitivity-list])
   begin [optional label]
      [optional local variable declaration];
      [procedural statement];
      [procedural statement];
   end
```

   - Statements between `begin` and `end` execute sequentially
   - sensitivity list are signals where events on them execute the block
   - Procedural blocks may contain: a logic function without the keyword assign (blocking assignment), an if statement, or a case statement
   - Procedural block outputs can only be assigned to an output of type `reg` that is declared as a module signal declaration before the keyword begin:
     - Example: `reg [signal-name]`

   b. Blocking assignment are those assignments made in a combinatorial procedural block (No clock). example: `[variable-name] = [expression]`
   c. `if` statement syntax

```
if [relational-operator] or [Boolean expr]
     begin
       [ statement 1] ;
       [ statement 2 ] ;
       . . .
     end
else
     begin
       [ statement 1] ;
       [ statement 2 ] ;
       . . .
     end
```

- Relational operators are:
  - greater than `[ > ]` ; greater than or equal `[ >= ]`
  - less than `[ < ]` ; less than or equal `[ <= ]`
  - equal `[ == ]` ; not equal `[ !~ ]`
- if statements can be nested and imposes priority in synthesis
- a 2-1 mux example:

```
module mux ( output reg op, input a, b, sel);
    always @ (a or b or sel)
        if (sel == 1)
             op = a;
        else
             op = b;
endmodule
```

d. `case` statement example

```
case [case_expr]
   [item_01]:
       begin
           [ statement 1] ;
           [ statement 2] ;
           . . .
       end
   [item_02]:
       begin
           [ statement 1] ;
           [ statement 2] ;
           . . .
       end
    . . .
    default :
       begin
           [ statement 1] ;
           [ statement 2] ;
           . . .
       end
endcase
```

- `case` statement is a multiway decision statement that compares the [case_expr] to the items. The execution jumps to the branch whose [item] matches the current value of [case_expr]
- `default` keyword is used to cover all unspecified values
- `begin`/`end` can be omitted if there is only one statement in the branch
- Example: 2-to-4 decoder

```
module decoder_2_t0_4 ( output reg [3:0] y,
                        input [1:0] a,
                        input en );
  always @ (en or a)
    case ({en,a}) // concatenation to form a bus
       3'b100:  y = 4'b0001;
       3'b101:  y = 4'b0010;
       3'b110:  y = 4'b0100;
       3'b111:  y = 4'b1000;
       default: y = 4'b0000;
    endcase
endmodule
```

  - { … } is the concatenation operator. It combines small signals or vectors to form a larger vector

- Examples:

```
wire a1;
wire [3:0] a4;
wire [7:0] b8, c8, d8;
. . .
assign b8 = {a4, a4};
assign c8 = {a1, a1, a4, 2'b00};
assign d8 = {b8[3:0] , c8[3:0]};
```

01.2 Module instantiation for structural modeling

This allows us to build a large system from predesigned component.

Syntax:

```
[module-name] [instance-label]
   (
      [port-name] ([signal-name]),
      [port-name] ([signal-name]),
      . . .
   );
```

- **module-name** specifies which component is used
- **instance-label** gives a unique id for the instance
- The rest is the port connections which defines the connections between the I/O ports of the instantiated module to external signals used in the higher module
- An alternative scheme to associate ports to external signals is connection by ordered list. In this scheme, port names of the instantiated modules are omitted and the external signals listed in the same order as the module port declarations.
- The fist method (explicit) list is clearer and help omit a lot of errors related to the correct order of ports

```
[module-name] [instance-name]
  (
  ([signal-name_for_port_1]),
  ([signal-name_for port_2]),
  . . .
  );
```

01.3 For Loop

For loops can be used in testbenches to iterate through all combination values of input signals.

- It and must be included inside an initial block
- loop index variable must be declared outside of the for loop as an integer
- for loops can be nested using two loop indices
- syntax:

```
for ([initial Assignment]; [end-condition] ; [step_assignment])
  begin
    [procedural_statements;]
    # [delay_time]
  end
```

01.4 Read-Only Memories

Read-only memories (ROM) consist of interconnected arrays used to store an array of binary information. Once the binary information is stored, it can be read any time but cannot be altered. Large ROMs are typically used to store programs and/or data which will not change by the other components of the system. Small ROMs can be used to implement combinatorial circuits.

A ROM usually have m address input lines and n information output pins. It can store 2^m words, each word being n-bit in length. The content can be accessed by placing a value on the address lines and the content of the corresponding word is read at the output pins.

In Verilog, memories can be defined using **arrays**, as illustrated below:

```
module MY_ROM(ADDR, DATA);
    input [3:0] ADDR;
    output reg [3:0] DATA;

// implement memory using case statement

always @(ADDR)
    begin
        case(ADDR)
                4'b0000 : DATA = 4'b0000;
                4'b0001 : DATA = 4'b0001;
                ...
                4'b1110 : DATA = 4'b1101;
                4'b1111 : DATA = 4'b1111;
        endcase
    end
```

.

Where MY_ROM is a 16x4 memory array with 16 locations each location being 4-bit wide. If the memory is to be modeled as read only then two things must happen: (i) memory should only be read and not written into, and (ii) memory should somehow be initialized with the desired content. The constant declarations immediately after the type declaration initializes the memory to the desired content. In this case, binary bits counting from 0000 (decimal 0) to 1111 (decimal 15).

01.6 *Double – Dabble Algorithm*

In software, the **double dabble** algorithm is used to convert binary numbers into binary-coded-decimal (BCD) notation. It is also known as the **shift-and-add-3 algorithm** **shift-and-add-3**, and we want to implement it in hardware using a small number of gates (minimum resources)

Shift and Add-Algorithm

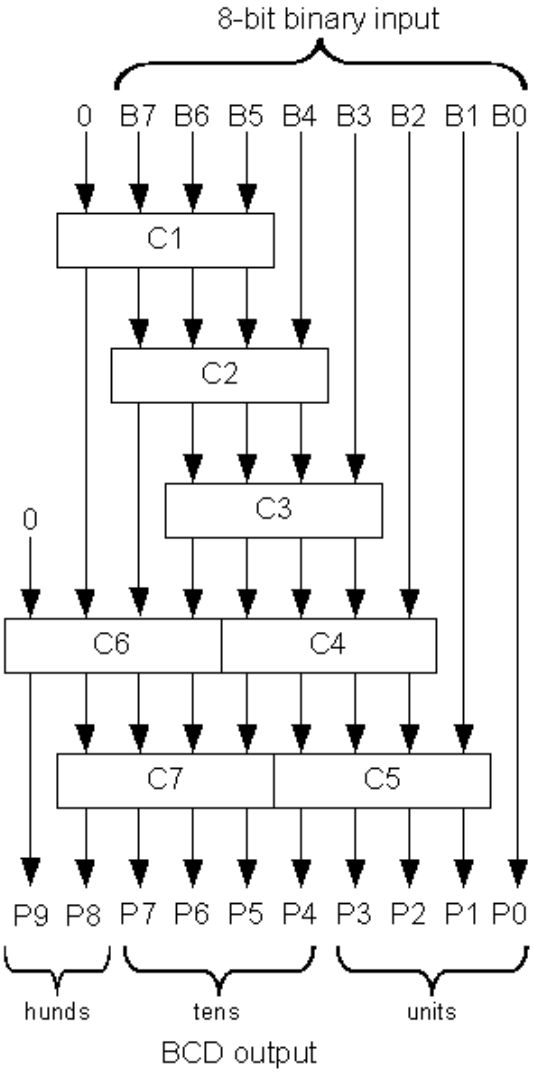
1. Shift the binary number left one bit.
2. If 8 shifts have taken place, the BCD number is in the *Hundreds*, *Tens*, and *Units* column.
3. If the binary value in any of the BCD columns is 5 or greater, add 3 to that value in that BCD column.
4. Go to 1.

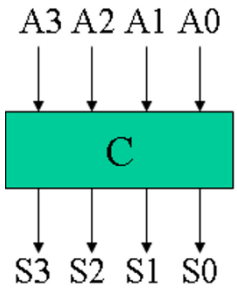
Example: Convert hex FF (Binary 1111_1111) to BCD

Operation	Hundreds	Tens	Units	Binary	
HEX				F	F
Start				1 1 1 1	1 1 1 1
Shift 1			1	1 1 1 1	1 1 1
Shift 2			1 1	1 1 1 1	1 1
Shift 3			1 1 1	1 1 1 1	1
ADD 3			1 0 1 0	1 1 1 1	1
Shift 4		1	0 1 0 1	1 1 1 1	
ADD 3		1	1 0 0 0	1 1 1 1	
Shift 5		1 1	0 0 0 1	1 1 1	
Shift 6		1 1 0	0 0 1 1	1 1	
ADD 3		1 0 0 1	0 0 1 1	1 1	
Shift 7	1	0 0 1 0	0 1 1 1	1	
ADD 3	1	0 0 1 0	1 0 1 0	1	
Shift 7	1 0	0 1 0 1	0 1 0 1		
BCD (Decimal)	2	5	5		

Algorithm Representation for Implementation

for hardware implementation and instead of doing shift and add operations (which consume much resources), we flatten the algorithm as follows





A3	A2	A1	A0	S3	S2	S1	S0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	1
0	1	0	0	0	1	0	0
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0
1	0	1	0	X	X	X	X
1	0	1	1	X	X	X	X
1	1	0	0	X	X	X	X
1	1	0	1	X	X	X	X
1	1	1	0	X	X	X	X
1	1	1	1	X	X	X	X

01.5 Testbench Structure

```
/* a timescale directive is used to specify that
   The simulation time unit is 1 ns and
   The simulation timestep is 10 ps */
`timescale 1 ns/10 ps

// testbench module has no ports
module tb_design;

//signal declarations
   reg  tb_in1 , tb_in2;
   wire tb_out;

// other declaration
   parameter period=50; // a constant value of 50ns


// instantiate the unit under test
   design uut (.in1(tb_in1),
               .in2(tb_in2,
               .out (tb_out)
              );

   //in-line stimulus
   initial
      /* initial block executes only once and
         the statements within executes sequentially */
   begin
      // test vector 1
      tb_in1 = 2'b00;
      tb_in2 = 2'b00;
      # period /* wait for a fixed amount of time
                  before changing inputs */

      // test vector 2
      tb_in1 = 2'b01;
      tb_in2 = 2'b00;
      # period

      // go through all combinations
      . . .

      // wait some time then stop the simulation
      # (period*5)  $stop  /* you may use also use $finish
                              but it will exit the
                              simulation */
   end // of initial block

endmodule
```