Sequential Logic Design

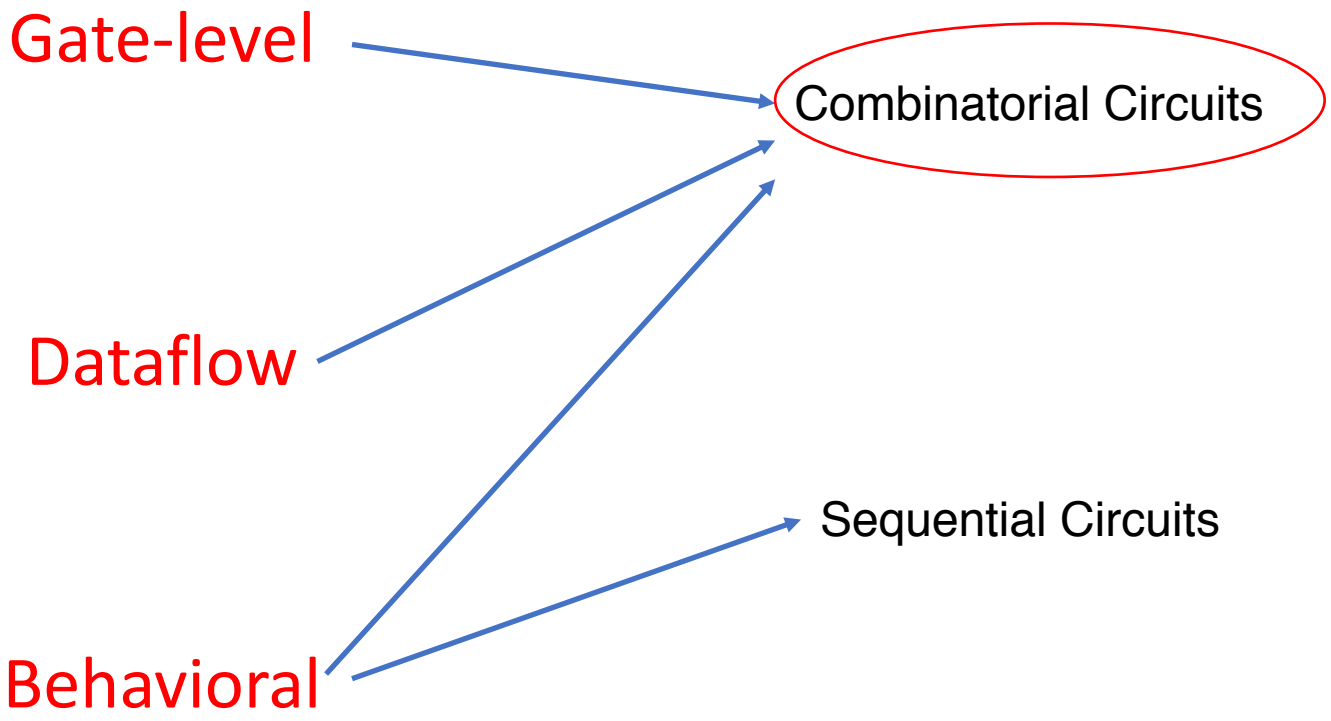
Lecture 5

Outline

- Create scalar and wide combinatorial circuits using gate-level, dataflow, and behavioral modeling
- Write models to read switches and push buttons, and output on LEDs
- Simulate and understand the design output
- Synthesize, implement and generate bitstreams
- Download bitstreams into the board and verify functionality

Verilog HDL

Three kinds of modeling styles



Gate-Level Modeling

Built-in primitive gates modeling are supported

Multiple-input (one output)

and | **nand** | **or** | **nor** | **xor** | **xnor** [instance name] (out, in1, ..., inN);

Multiple-output (one input)

buf | **not** [instance name] (out1, out2, ..., out2, input);

Tristate (one input, one control signal, one output)

bufif0 | **bufif1** | **notif0** | **notif1** [instance name] (outputA, inputB, controlC);

Pull gates (single output, no input)

pullup | **pulldown** [instance name] (output A);

// [] is optional and | is selection

Multiple Instances of the Same Type of Gate in One Construct

```
and [inst1] (out11, in11, in12), [inst2] (out21, in21, in22, in23), [inst3] (out31, in31, in32, in33);
```

Allowing the delays to be expressed when instantiating gates

```
and #5 A1(out1, in1, in2); // the rise and fall delays are 5 units
and #(2,5) A2(out2, in1, in2); // the rise delay is 2 units and the fall delay is 5 units
notif1 #(2, 5, 4) A3(out3, in2, ctrl1); //the rise delay is 2, fall delay is 5, turn-off delay is 4 units
```

Dataflow Modeling

- Mainly used to describe combinational circuits.
- The basic mechanism used is the continuous assignment.
- In a continuous assignment, a value is assigned to a data type called **net**.
- The syntax of a continuous assignment is:

```
assign [delay] LHS_net = RHS_expression;
```

The statement is evaluated at any time any of the source operand value changes and the result is assigned to the destination net after the delay unit.

```
assign out1 = in1 & in2; // perform and function on in1 and in2 and assign the result to out1
assign out2 = not in1;
assign #2 z[0] = ~(ABAR & BBAR & EN); // perform function and assign the result after 2 units
```

Target in the Continuous Assignment Expression

1. A scalar net

```
assign out1 = in1 & in2;
```

2. Vector net
3. Constant bit-select of a vector

```
assign #2 z[0] = ~(ABAR & BBAR & EN);
```

4. Constant part-select of a vector
5. Concatenation of any of the above

```
wire COUT, CIN; // scalar net declaration
wire [3:0] SUM, A, B; // vector nets declaration
assign {COUT,SUM} = A + B + CIN; // A and B vectors are added with CIN and the result is
                                 // assigned to a concatenated vector of a scalar and vector nets
```

Note: multiple continuous assignment statements are not allowed on the same destination net.

Behavioral Modeling

- Used to describe complex circuits, primarily sequential circuits, but also pure combinatorial circuits.
- The statements for modeling the behavior of a design are:

  initial [timing_control] procedural_statements; //both may contain multiple procedural statements

  always [timing_control] procedural_statements;

- They are executed concurrently, Both **initial** and **always** statements are executed at time=0 and then only **always** statements are executed during the rest of the time
- The **initial** statement is non-synthesizable and is normally used in testbenches
- The **always** statement is synthesizable

Numbers Representation

[size]'[signed][radix]value ([ ] indicates optional)

size The number of binary bits the number is comprised of Default is 32 bits.

' A separator

signed including S Indicates if the value is signed Default is unsigned.

Radix b : binary, o : octal, h : hex, d : decimal (also default)

Note: X represents unknown, Z represents high impedance (none of the above is case dependent)

Number Representation

In Verilog HDL a signal can have the following four basic values:

i. 0 : logic-0 or false
ii. 1 : logic-1 or true
iii. x : unknown
iv. z : high-impedance

Integer numbers can be written in

i. simple decimal or

i.e. 16, -34

ii. base format

[size] 'base value

number of bits

Unsigned sequence of digits valid for the specified base

o (octal), b (binary), d (decimal), h (hexadecimal)

```
wire [4:0] 5'O37      // 5-bit octal representation

reg [3:0] 4'B1x_01    // 4-bit binary
```

Three types of constants in Verilog HDL:

i. Integer
ii. Real
iii. String

Note:

- If the size specified is larger than the value for the specified constant,
  - the number is padded to the left with 0's
  - for the case where the left most bit is x or z then the padding is done with x or z.
- If the size specified is smaller then the extra left most bits are ignored.
- If the size is not specified then it will use 32-bit data.

Procedural Statements

A module may contain an arbitrary number of **initial** or **always** statements and may contain one or more procedural statements within them. Procedural statements are executed sequentially

Procedural Assignment

[delay] register_name = [delay] expression; // blocking

[delay] register_name <= [delay] expression; // non-blocking

Example- Procedural Assignment:

```
// time 0: a=0; time 10: a=1; time 15 (#10+#5): a=2;
begin
a = 0;
#10 a = 1;
#5 a = 2;
end

// time 0: a=0; time 5: a=2; time 10: a=1;
begin
a <= 0;
#10 a <= 1;
#5 a <= 2;
end

// both assignments are evaluated before a or b changes
begin
a <= b;
b <= a;
end
```

Conditional Statement

condition ? Expression1 : expression2;

Example- modeling tri-state buffer:

assign data_out = (enable) ? data_reg : 8'bz;

Example

(a) ? 4'b110x : 4'b1000;

	0	1	x	z
0	0	x	x	x
1	x	1	x	x
x	x	x	x	x
z	x	x	x	x

- If one of the expressions is of real type then the result of the whole expression should be 0 (zero).
- If expressions have different lengths, then length of an entire expression will be extended to the length of the longer expression.
- Trailing 0s will be added to the shorter expression.
- The conditional operator can be nested and its behavior is identical with the case statement behavior

Case Statements

case ()

<case1>: <statement>;

<case1>: <statement>;

….

default : <statement>

loop Statements

Forever, repeat, while, for

Wait Statements

Example

case (i)

0 : y = a;

1 : y = b;

2 : y = c;

default :$display(“Error”);

endcase

https://www.csee.umbc.edu/portal/help/VHDL/verilog/sequential.html

A Multiplexer - Two Ways

reg [4:0] mux;
reg [1:0] addr;

```
mux = (addr == 2'b00) ? i0 :
  ((addr == 2'b01) ? i1 :
  ((addr == 2'b10) ? i2 :
  ((addr == 2'b11) ? i3 :
  4'bz)));
```

```
case (addr)
  2'b00: mux = i0;
  2'b01: mux = i1;
  2'b10: mux = i2;
  2'b11: mux = i3;
  default: mux = 4'bz;
endcase
```

To generate a combinatorial circuit, the **always** block

(i) should not be edge sensitive,

(ii) every branch of the conditional statement should define all output, and

(iii) every case of case statement should define all output and must have a default case.

The destination (LHS) should be of **reg** type; either scalar or vector

```
reg m; // scalar reg type
reg [7:0] switches; // vector reg type
```

Example 2-to-1 Mux

```
always @ (x or y or s)  //The sensitive list, always block executes when changed
begin
        if(s==0)
                m=y;
        else
                m=x;
end
```

Lab 2

1. **Create a two-bit wide 2-to-1 multiplexer using gate-level modeling.**
   a. Synthesize the design.
   b. Implement the design.
   c. Generate the bitstream, download it into the ZedBoard to verify the functionality.
2. **Model a two-bit wide 2-to-1 multiplexer using dataflow modeling with net delays of 3 ns.**
   a. Simulate the design for 100 ns and analyze the output. Refer to last slide for the testbench
   b. Synthesize the design.
   c. Implement the design.
   d. Generate the bitstream, download it into the ZedBoard, and verify the functionality.
3. **Create a two-bit wide 2-to-1 multiplexer using behavioral modeling.**
   a. Synthesize the design.
   b. Implement the design.
   c. Generate the bitstream, download it into the ZedBoard to verify the functionality.
4. **Create a two-bit wide 3-to-2 multiplexer using two 2-1 multiplexers from parts 1, 2, 3, or 4**
   a. Synthesize the design.
   b. Implement the design.
   c. Generate the bitstream, download it into the ZedBoard to verify the functionality.

Use the following testbench for Lab 2 Part 2.a:

```
initial
   begin
     x = 0; y = 0; s = 0;
     #10 x = 1;
     #10 y = 1;
     #10 x = 3; y = 0;
     #10 x = 2; y = 3;
     #10 s = 1;
     #10 x = 1;
     #10 y = 1;
     #10 x = 3; y = 0;
     #10 x = 2; y = 3;
              #20;
   end
end
```

Where x and y are the 2-bit inputs and s is the select input

Addition- Ripple Carry Adder

5. Create a 4-bit ripple carry adder using dataflow modeling

a. Create and add a Verilog module with three one-bit wide inputs (*a, b, cin*) and one-bit wide two outputs (*s* and *cout*)

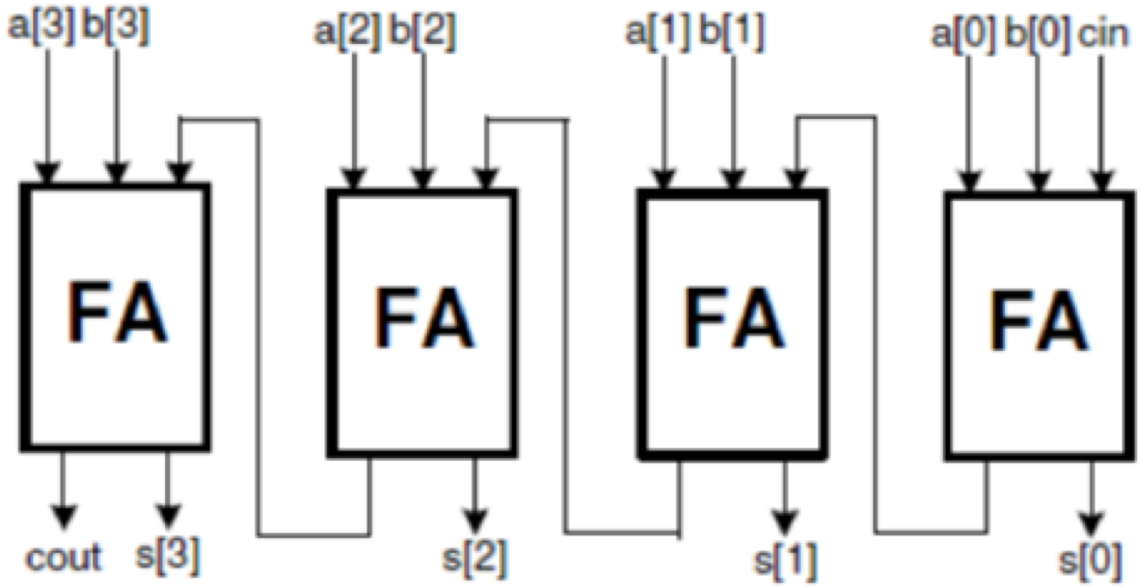
b. Simulate the design for 80 ns using the provided testbench and verify that the design works

c. Create a 4-bit adder by instantiating the adder created in part 5-a.

d. Create the XDC file reflecting the LEDs and Switches of your choice for implementation.

e. Synthesize, implement the design, generate the bitstream and download it into the ZedBoard to verify the functionality.

b	a	c_i	c_o	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Full Adder Dataflow Testbench

```
reg a, b, cin;
wire cout, s;

   fulladder_dataflow DUT (.a(a), .b(b), .cin(cin), .cout(cout), .s(s));

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

Carry Lookahead Adder

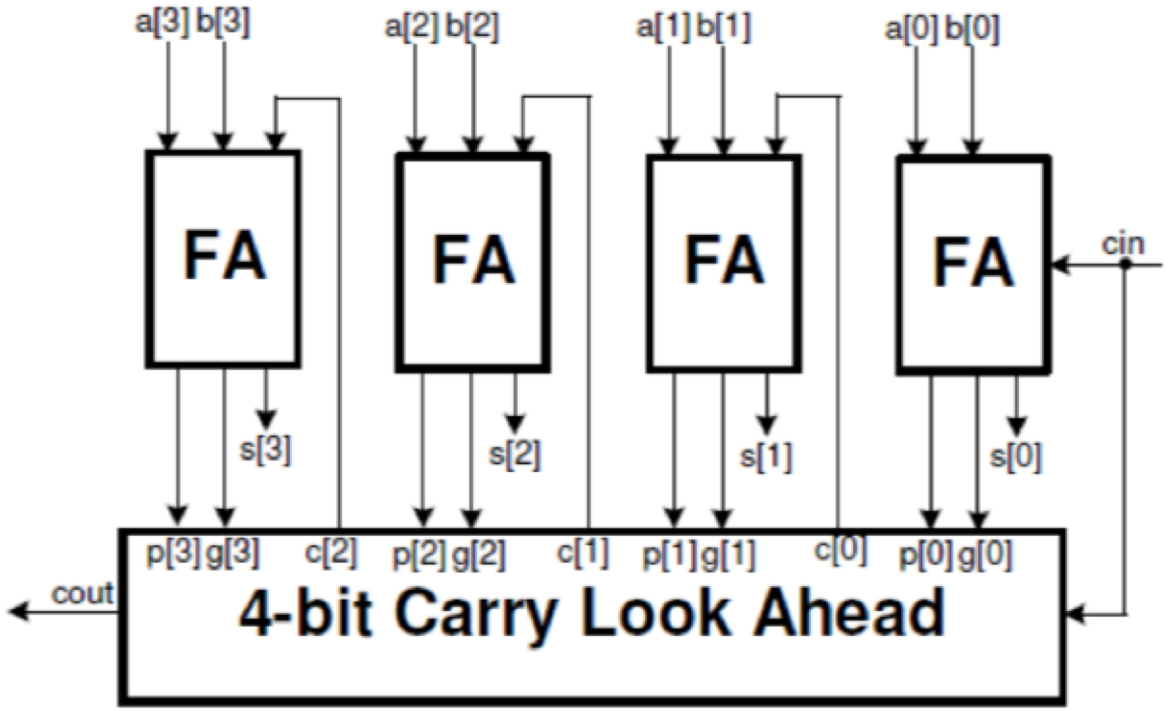
6. Create a 4-bit carry Lookahead adder using dataflow modeling

a. Modify the ripple carry adder as necessary

b. Create the XDC file reflecting the LEDs and Switches of your choice for implementation.

c. Synthesize and implement the design.

d. generate the bitstream and download it into the ZedBoard to verify the functionality.



Where:

$P_i = A_i + B_i$

$G_i = A_iB_i$

$C_{i+1} = G_i + P_iC_i$