# Design of Combinational Logic: Full Adder, Adder, and ALU

## 1 Today's goal

- Learn how to use ISE WebPack.
- Learn the design of combinational logic using Verilog HDL.
- Learn how to write test benches and perform the simulation.
- Learn how to embed a designed circuit into an FPGA.
- Design ALU as a basic component of CPU.

## 2 Today's contents

**Step 1** Write full adder(List 1) and its test bench(List 2).

**Step 2** *Check1* Perform the simulation using the test bench to confirm that the full adder works correctly.

**Step 3** *Check2* Write a UCF(User Constraint File) (List 3), embed it in the FPGA to confirm that the full adder works in the FPGA correctly.

**Step 4** Write full adder using always statement(List 4) and confirm that it works correctly by the simulation using the test bench(List 2).

**Step 5** *Check 3* Write 4-bit adder using 4-full adders(List 5) and confirm that it works correctly by the simulation using its test bench(List 6).

**Step 6** *Check 4* Write ALU(List 7) and confirm that it works correctly by the simulation using its test bench(List 8).

## 3 Full adder

Full adder has 3 input bits a,b, cin and 2 output bits s, cout. The sum of 3 input bits are computed and 2 out bits represent the sum such that s is a lower bit, and count is an upper bit. Assignment statements defines continuous assignments.

List 1: Full adder using assigment statementsfa.v

```
1  module fa(a, b, cin, s, cout);
2
3    input a, b, cin;
4    output s, cout;
5    wire a, b, cin, s, cout;
6
7    assign s = a ^ b ^ cin;
8    assign cout = (a & b) | (b & cin) | (cin & a);
9
10 endmodule
```

## 4 Test bench

Test bench defines the change of inputs. In List 2, module fa is instantiated as fa_0.

List 2: Test benchfa_tb.v for fa.v

```
1  `timescale 1ns / 1ps
2  module fa_tb;
3
4    reg a,b,cin;
5    wire s, cout;
6    fa fa0 (.a(a), .b(b), .cin(cin), .s(s), .cout(cout));
7
8    initial begin
9      a = 0; b = 0; cin = 0;
10     #100 a = 1; b = 0; cin = 0;
11     #100 a = 0; b = 1; cin = 0;
12     #100 a = 1; b = 1; cin = 0;
13     #100 a = 0; b = 0; cin = 1;
14     #100 a = 1; b = 0; cin = 1;
15     #100 a = 0; b = 1; cin = 1;
16     #100 a = 1; b = 1; cin = 1;
17     #100 a = 0; b = 0; cin = 0;
18   end
19 endmodule
```

## 5 UCF(User Constraint file)

The UCF defines the mapping between ports of the module and the pins of an FPGA. NET and LOC correspond to a

name of module port, and a name of FPGA pin.

List 3: UCF for fa.ucf (Spartan-3A/AN)

```
# SWITCH
NET "a" LOC = "V8" | IOSTANDARD = LVTTL | PULLUP;
NET "b" LOC = "U10" | IOSTANDARD = LVTTL | PULLUP
    ;
NET "cin" LOC = "U8" | IOSTANDARD = LVTTL |
    PULLUP;

# LED
NET "s" LOC = "R20" | IOSTANDARD = LVTTL | SLEW =
     SLOW | DRIVE = 8;
NET "cout" LOC = "T19" | IOSTANDARD = LVTTL |
    SLEW = SLOW | DRIVE = 8;
```

List 5: 4-bit adder using adder4.v

```
module adder4(a, b, s);

   input [3:0] a,b;
   output [3:0] s;
   wire [2:0] c;

   fa fa0(.a(a[0]),.b(b[0]),.cin(0),.s(s[0]),.cout(c[0]));
   fa fa1(.a(a[1]),.b(b[1]),.cin(c[0]),.s(s[1]),.cout(c[1]));
   fa fa2(.a(a[2]),.b(b[2]),.cin(c[1]),.s(s[2]),.cout(c[2]));
   fa fa3(.a(a[3]),.b(b[3]),.cin(c[2]),.s(s[3]));

endmodule
```

# 6 Always statement

Always statements in List 4 is used to design combinational logic. "always @ (...)" defines a event list. If the values of signal (net) in the event list change, the following statement is executed.

# 7 Instantiate modules

In List 5, module fa is instantiated four times as fa0, fa1, fa2, and fa3. These modules are connected by wires (nets). Instead, we can simply use " assign s = a+b" instead of using four fa's.

List 4: Full adder using always statementfa.v

```
module fa(a, b, cin, s, cout);

   input a, b, cin;
   output s, cout;
   reg s, cout;

   always @(a or b or cin)
     begin
       s = a ^ b ^ cin;
       cout = (a & b) | (b & cin) | (cin & a);
     end

endmodule
```

List 6: Test bench for 4-bit adderadder4_tb.v

```
`timescale 1ns / 1ps
module adder4_tb;

  reg [3:0] a,b;
  wire [3:0] s;

  adder4 adder4_0(.a(a),.b(b),.s(s));

  initial begin
  a = 4'b0000; b=4'b0000;
  #100 a = 4'b0001;
  #100 a = 4'b0010;
  #100 b = 4'b0111;
  #100 a = 4'b1101;
  #100 a = 4'b1011;
  #100 b = 4'b1001;
  #100 b = 4'b1110;
  #100a = 4'b0000; b=4'b0000;
  end

endmodule
```

# 8 ALU

ALU (List 7) is used to compute a selected function. ALU has 3 input ports, f(5 bits), a(16 bits), b(16 bits), and one output port s. f is used to select a function (operation), and the resulting value is output from s. We assume that a, b, s are signed integers (2's complement). However, array of bits (vector) in Verilog HDL is handled as unsigned integers. Thus, for relational operators, we add 16'h8000 to a and b to get correct results.

List 7: ALU alu.v

```
1  `define ADD 5'b00000
2  `define SUB 5'b00001
3  `define MUL 5'b00010
4  `define SHL 5'b00011
5  `define SHR 5'b00100
6  `define BAND 5'b00101
7  `define BOR 5'b00110
8  `define BXOR 5'b00111
9  `define AND 5'b01000
10 `define OR 5'b01001
11 `define EQ 5'b01010
12 `define NE 5'b01011
13 `define GE 5'b01100
14 `define LE 5'b01101
15 `define GT 5'b01110
16 `define LT 5'b01111
17 `define NEG 5'b10000
18 `define NOT 5'b10001
19 `define BNOT 5'b10010
20
21 module alu(a, b, f, s);
22
23   input [15:0] a, b;
24   input [4:0]   f;
25   output [15:0] s;
26   reg [15:0]    s;
27   wire [15:0] x,y;
28
29   assign x = a + 16'h8000;
30   assign y = b + 16'h8000;
31
32   always @(a or b or x or y or f)
33     case(f)
34       `ADD : s = b + a;
35       `SUB : s = b - a;
36       `MUL : s = b * a;
37       `SHL : s = b << a;
38       `SHR : s = b >> a;
39       `BAND: s = b & a;
40       `BOR : s = b | a;
41       `BXOR: s = b ^ a;
42       `AND : s = b && a;
43       `OR : s = b || a;
44       `EQ : s = b == a;
45       `NE : s = b != a;
46       `GE : s = y >= x;
47       `LE : s = y <= x;
48       `GT : s = y > x;
49       `LT : s = y < x;
50       `NEG : s = -a;
51       `BNOT : s = ~a;
52       `NOT : s = !a;
53       default : s = 16'hxxxx;
54     endcase
55
56 endmodule
```

# 9 Homeworks

In your report, you must show enough explanation and the simulation results.

**Homework 1** Design an 8-bit adder using 8 full adders, and write its test bench. Perform the simulation to confirm that the 8-bit adder works correctly.

**Homework 2** Write test benches for ALU to confirm that each of 19 functions works correctly. You should choose various input b and a for each functin. For example, for bianry arithmetic function, you should choose $\{b > 0, b < 0\} \times \{a > 0, a < 0\}$ (4 cases), and the case that the reult is overflow.

Table 1: Specification of ALU(Arithmetic and Logic Unit)

| | | function f | | outputs |
|---|---|---|---|---|
| binary | arithmetic | ADD | 00000 | b + a (addition) |
| | | SUB | 00001 | b − a (subtraction) |
| | | MUL | 00010 | b * a (multiplication) |
| | shift | SHL | 00011 | b << a (left shift) |
| | | SHR | 00100 | b >> a (right shift) |
| | bitwise | BAND | 00101 | b & a (bitwise and) |
| | | BOR | 00110 | b \| a (bitwise or) |
| | | BXOR | 00111 | b ^ a (bitwise xor) |
| | logic | AND | 01000 | b && a (logical and) |
| | | OR | 01001 | b \|\| a (logical or) |
| | relational | EQ | 01010 | b==a (b is equal toa) |
| | | NE | 01011 | b!=a (b is not equal toa) |
| | | GE | 01100 | b>=a (b is larger than or euqal to a) |
| | | LE | 01101 | b<=a (b is smaller than or equal to a) |
| | | GT | 01110 | b>a (b is larger than a) |
| | | LT | 01111 | b<a (b is smaller than a) |
| unary | arithmetic | NEG | 10000 | −a (negation) |
| | bitwise | BNOT | 10001 | ~a (bitwise not) |
| | logic | NOT | 10010 | !a (logical not) |

List 8: Test bench foralu_tb.v

```
1  `timescale 1ns / 1ps
2
3  module alu_tb;
4
5    reg [15:0] a,b;
6    reg [4:0] f;
7    wire [15:0] s;
8
9    alu alu0(.a(a),.b(b),.f(f),.s(s));
10
11   initial begin
12   a = −3; b= 3; f = 5'b01100;
13   #100 a = −2;
14   #100 a = −1;
15   #100 a = 0;
16   #100 a = 1;
17   #100 a = 2;
18   #100 a = 3;
19   #100 a = 4;
20   #100 a = 5;
21   #100 a = 6;
22   end
23
24 endmodule
```