INTRODUCTION TO VERILOG HDL

OVERVIEW

- Verilog was adopted as an official standard as IEEE Standard 1364-1995 in 1995.
- An enhanced version, called Verilog-2001, was adopted in 2001 as IEEE Standard 1364-2001.
- Originally intended for simulation, today Verilog is designed to facilitate describe digital hardware for simulation and synthesis.
- Verilog is a great low level language.
- The syntax is regular and easy to remember. It is the fastest HDL language to learn and use.
- However Verilog lacks user defined data types and lacks the interface-object separation of the VHDL's entity-architecture model.

Verilog constructs

Entity declaration	**module** circuit (a, b, C, D);
	    input a;
	    output b;
	    input [3:0] C;
	    output [0:7] D;
	endmodule

Internal signals, variables, constants	**wire** int a ; **wire** [3:0] int b ; **integer** [7:0] counter; **reg** [0:7] temp; **parameter** C 3'b000 ;

Component instantiation	**module** system1 (……) ;
	
	    circuit U_comp (A, B);
	
	endmodule

Concurrent signal assignment	**assign** Dataout = Datain;

Sequential block	**always @** (a)
	begin
	    ……
	end

Control flow	
• if	**if** (en == 1) f = x1;

• if …else	**if** (sel == 0)
	    begin f = x1; g = x2; **end**
	else
	    begin f = x2; g = x1; **end**

• case	**case** (y)
	    0 : f <= stateA;
	    1 : f <= stateB;
	    default : f <= stateC;
	endcase

BASICS OF VERILOG

Verilog design unit

```
module module_name (  ports  );
    {parameter declarations}
    input <port list> ;        // input/output declarations:
    output  <port list> ;
    wire  <list> ;          // nets & variables:
    reg (or integer) <list> ;

    {assign continuous statement;}  // behaviour statements:
    {initial block;}
    {always blocks;}
    {gate instantiations;}
    {module instantiations;}
endmodule
```

- Verilog describes a digital circuit or system as a set of modules.
- The entity used in Verilog description of hardware components is a module
- Following the module header is a declarative part, where module ports, nets and variables are declared.
- A port in Verilog may be *input*, *output*, or *inout*. Ports provide the module with a means to connect to other modules.
- Nets are typically declared by the keyword *wire*, connection between hardware elements.
- A port is automatically declared as *wire* if it is defined as *input*, *output*, or *inout*.
- Variables, declared as *reg*, are used for behavioural descriptions, and are very much like variables in software languages.

- Usually each line of Verilog text must terminate with a semicolon, one exception of which is the terminating *endmodule* keyword.
- Verilog is case-sensitive. It allows letters, numbers and special character "_" to be used for names.
- Names (or identifiers) are used for modules, parameters, ports, variables, and instances of modules, and must begin with letters.
- Keywords cannot be used for names (i.e. *and, not, xor, xnor*).

```
module circuitA (  Cin, x, y, X, Y, Cout, s, Bus, S  );

   input   Cin, x, y;
   input   [3:0]   X, Y;
   output Cout, s;
   output [3:0]   S;
   inout    [7:0]  Bus;

   wire  d;
   reg     e;
   ...
endmodule
```

Representation of Numbers in Verilog

- Verilog uses a 4-value logic, that is, 0, 1, z, and x.
- Numbers can be given as binary (b), octal (o), hex (h), or decimal (d).

  <size-in-bits> ′ <radix-identifier> <significant-digits>

- E.g., 2217 can be represented as 12’b100010101001, 12’h8A9, or 12’d2217.
- Unsigned numbers are given without specifying the size, e.g. ‘b1000100110 or ‘h116 or ‘d2217 (will not be zero-padded)
- Negative numbers, e.g. if -5 is specified as -4’b101, it will be interpreted as a four-bit 2’s complement of 5, which is 1011.
- The number 12’b100010101001 may be written as 12’b1000_1010_1001 to improve read-ability in the code.

- A constant used in Verilog may be given as 8’hz3, which is the same as 8’bzzzz0011.
- 8’hx denotes an unknown 8-bit number.

Operators in Verilog

Operator type	Operator symbol	Operation
Bitwise	~, &, |, ^, ~^	not, and, or, xor, xnor
Logical	!, &&, ||	not, and, or
Arithmetic	+, -, *, /	add, sub, mult, divide
Relational	>, <, >=, <=	Gt, Lt, Gt or eq, Lt or eq
Equality	==, != ===, !==	Logical equality, logical inequality, case equality, case inequality
Shift	>>, <<	Right shift, Left shift
Concatenation	{ , }	
Replication	{{,}}	
Conditional	?:	

Consider that A, B, and C to be operands, either vectors or scalar (1-bit).

- The bitwise operator produces the same number of bits as the operands. E.g, A = a1a0, B = b1b0, C = c1c0, then A|B results in c1=a1|b1 and c0 = a0|b0.
- The logical operator generates a one-bit result. Used in conditional statements.
  - A||B results in 1 unless both A and B are zeros
  - A && B will produce a result of 1 if both A and B are non-zeros.
  - !A gives a 1 if all its bits are 0, otherwise it results in a 1.
- The relational operator outputs a 1 or 0 based on the (specified) comparison of A and B.
- The shift operators perform logical 1-bit shifts to the right or left, with zeros shifted in.

- In the case of the conditional operator, the operation A?B:C produce a result that is equal to B if A evaluates to 1, otherwise the result is C.
- The precedence of Verilog operators is similar to that found in arithmetic and Boolean algebra.

HDL MODELLING OF DIGITAL CIRCUITS

- Different circuit complexities (e.g., simple modules to complete systems) require different kinds of specification or levels of abstraction.
- Three modelling styles in HDL-based design of digital systems
  - Structural modelling
  - Dataflow modelling
  - Behavioural modelling

- Dataflow modelling – output signals are specified in terms of input signal transformation. This style is similar to Boolean equations. This model-ling style allows a digital system to be designed in terms of its function.

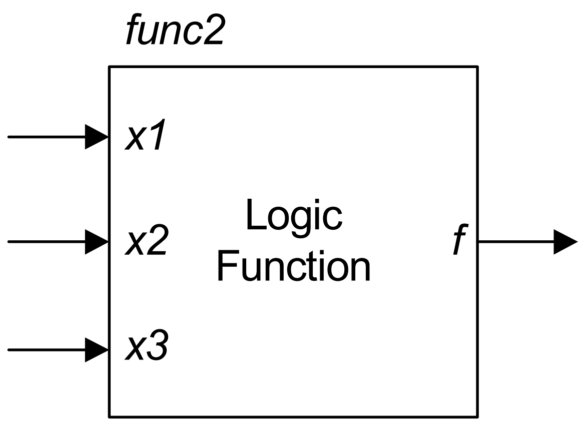
- Structural modelling – using primitives and lower-level module instantiation. This modelling allows for the hierarchical modular design approach in design. It is used to describe a schematic or logic diagram. The functionality of the design is hidden inside the components.

- Behavioural modelling – describes the function or expected behaviour of the design in an algorithmic manner. This style is the closest to a natural language description of the circuit functionality.

Dataflow Modelling

```
module func2 (x1, x2, x3, f) ;
   input     x1, x2, x3;
   output f ;

   assign f  = ( ~x1 & ~x2 & x3)
              | (x1 & ~x2 & ~x3)
              | (x1 & ~x2 & x3)
              | (x1 & x2 & ~x3) ;
endmodule
```



Modelling of full-adder using concurrent statements

```
module fulladder (Cin, x, y, S, Cout) ;
    input Cin, x, y;
    output S, Cout ;

    assign S  = ( x ^ y ^ Cin ) ;
    assign Cout = (x & y) | (Cin & x) | (Cin & y);
endmodule
```

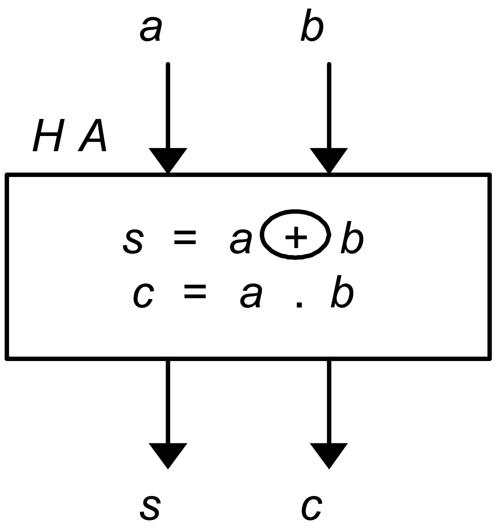
Notes on Concurrent/ Continuous signal assignment statements

- In Verilog, concurrent assignment statements are called continuous assignment statements.
- Lines 5 and 6 in the above Verilog code are continuous assignment statements, by the fact that they begin with the assign keyword.
- They are executed concurrently, and the line order is not important.
- Besides concurrent statements, there are also sequential (in VHDL jargon) or procedural (in Verilog jargon) statements.
- Differing from concurrent statements, sequential statements are evaluated in the order in which they appear in the code.
- Verilog syntax require them to be in an always block.

HDL dataflow description of a half-adder

```
module HA (a, b, s, c) ;
  input     a, b;
  output    s, c ;

  assign s  =  a ^ b  ;
  assign c  =  a & b ;
endmodule
```

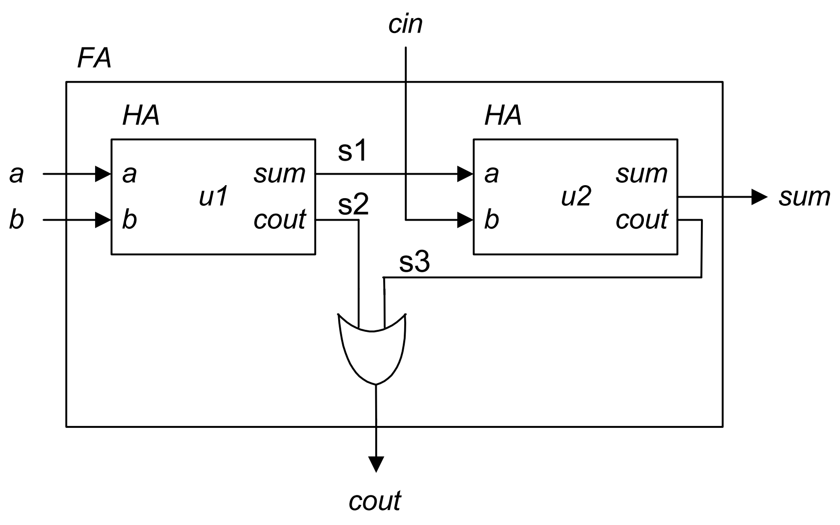


Structural Modelling

Modular design of full-adder using half-adders

```
module FA (cin, a, b, sum, cout) ;
   input      cin, a, b;
   output sum, cout ;
   wire  s1, s2, s3;

   HA  u1 (a, b, s1, s2);
   HA  u2 (s1, cin, sum, s3);
   assign cout = s2 | s3;
endmodule
```



- In Verilog structural modelling, module instantiation is used.
- The instantiation statement associates the signals in the instantiated module (HA, in this case) with the ports of the design unit (FA in this case).
- Here, positional association is applied, where each signal in the instantiation statement is mapped by position to the corresponding signal in the module.

Behavioural Modelling

- At higher levels of design abstraction, a digital module is often modelled behaviourally,
- The function or operation of the module is described in an algorithmic manner.
- The HDL code will contain statements that are executed sequentially in a predefined order (or procedure).
- The order of the sequential (or procedural) statements in the HDL code is important and may affect the semantics of the code.

Behavioural Modelling in Verilog

- Behavioural modelling in Verilog uses constructs similar to C language constructs.
- Sequential statements, like if-else and case statements, are called procedural statements.
- Procedural statements be contained inside a construct called an always block
- An always block execute sequentially in the order they are listed in the source code.
- The @ symbol is called the event control operator. The part after the @ symbol, is the event control expression, also referred to as the sensitivity list.
- This variable holds its value until the next time an event occurs on inputs in the sensitivity list.

```
module Vcircuit (A, B, z) ;
   input     [3:0] A, B;
   output z ;
   reg  z;

   always @ (A or B)
   begin
     z = 0 ;
     if (A == B)  z = 1;
   end
endmodule
```

- Verilog syntax requires any signal assigned a value inside an always block has to be a variable of type reg ; hence z is declared as reg.
- Since z depends on A and B, these signals are included in the sensitivity list.
- Blocking assignments, denoted by "=" symbol is used. The assignment completes and updates its LHS before the next statement is evaluated.
- We will cover non-blocking assignment, denoted by <= symbol later on.

VERILOG MODELLING OF BASIC COMBINATIONAL LOGIC

- Rather than using gates or logic equations, the circuits will be modelled in terms of their behaviour, applying behavioural modelling.
- As the circuit models are described, some new behavioural Verilog constructs are introduced.
- These constructs are similar to those found in programming languages, including if-else-if, case statements, and loops.
- They all control the activity of flow within the behavioural description.

Using IF-ELSE Verilog construct

The conditional IF statement executes a statement if a condition is true. There are two other variants available: IF-ELSE and IF-ELSE-IF statement.

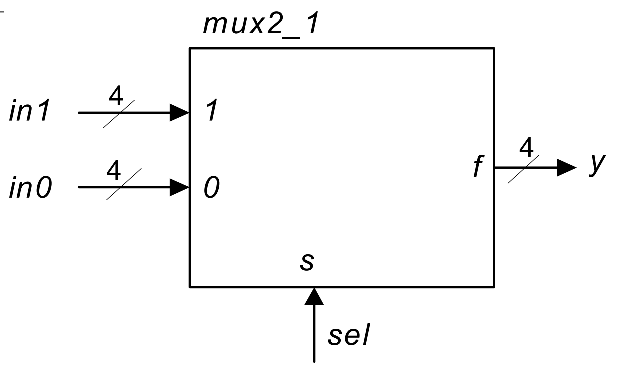
```
module mux2_1 (in0, in1, sel, y) ;
   input  in0, in1, sel;
   output y ;
   reg  y;

   always @ (in0 or in1 or sel)
        if (sel == 0)   y = in0;
        else  y = in1;
endmodule
```

Example 2-19

```
module mux2_1 (in0, in1, sel, y) ;
   input      [3:0]     in0, in1;
   input                sel ;
   output     [3,0]     y ;
   reg        [3,0]     y ;

   always @ (in0 or in1 or sel)
      if (sel == 0)
         y = in0;
      else
         y = in1;
endmodule
```



A 4-1 MUX using if-else-if

```
module mux4_1 ( X, S, y) ;
   input       [3:0]     X ;
   input       [1:0]     S ;
   output                y ;
   reg                   y ;

   always @ (X or S)
        if (S == 0)         y = X[0] ;
        else if (S == 1)  y = X[1] ;
        else if (S == 2)  y = X[2] ;
        else if (S == 3)  y = X[3] ;
endmodule
```

Using CASE statement

- Similar to the switch statement in C.
- It searches from top to bottom to find a match between the case expression and a case item.
- The case statement executes the statement associated with the first match found
- It does not consider any remaining possibilities.

A 4-1 MUX using case statement

```
module mux4_1 ( X, S, y) ;
   input       [3:0]      X ;
   input       [1:0]      S ;
   output                 y ;
   reg                    y ;
       always @ (X or S)
        case (S )
            0 :  y = X[0] ;
            1 :    y = X[1] ;
            2 :    y = X[2] ;
            3 :  y = X[3] ;
        endcase
endmodule
```

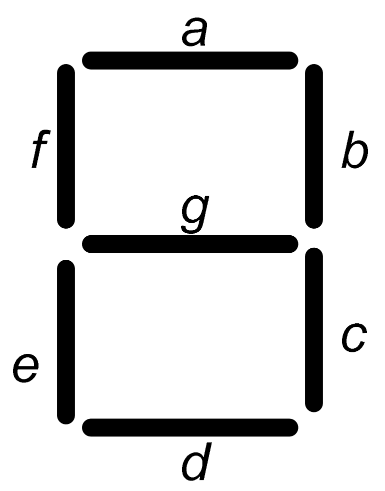
A 2-to-4 Decoder with Enable

```
module dec2_4 ( A, en, W) ;
   input  [1:0]       A ;
   input              en ;
   output [0:3]       W ;
   reg     [0:3]      W ;
   always @ (A or en)
   if (en == 0)  W = 4'b0000;
      else
       case (A )
            0 :    W = 4'b1000 ;
            1 :    W = 4'b0100 ;
            2 :    W = 4'b0010 ;
            3 :    W = 4'b0001 ;
       endcase
endmodule
```

Draw the I/O block diagram of the decoder described by this Verilog code.

```
module dec2_4 ( A, en, W) ;
   input  [1:0]      A ;
   input             en ;
   output [0:3]      W ;
   reg    [0:3]      W ;
   always @ (A or en)
        case ( {en, A} )
             3b'100 :   W = 4'b1000 ;
             3b'101 :   W = 4'b0100 ;
             3b'110 :   W = 4'b0010 ;
             3b'111 :   W = 4'b0001 ;
        default :  W = 'b0 ;
       endcase
endmodule
```

Example 2-13: BCD-to-7-segment display decoder



bcd	leds (abcdefg)
0000	1111110
0001	0110000
0010	1101101
0011	1111001
0100	0110011
0101	1011011
0110	1011111
0111	1110000
1000	1111111
1001	1111011
1010	Don’t care
1011	Don’t care
1100	Don’t care
1101	Don’t care
1110	Don’t care
1111	Don’t care

```
module SEG7 ( bcd, leds) ;
  input     [3:0]    bcd ;
  output    [1:7]    leds ;
  reg       [1:7]    leds ;

  always @ (bcd)
     case ( bcd )    //  abcdefg
        0 :   leds = 7'b1111110 ;
        1 :   leds = 7'b0110000;
        2 :   leds = 7'b1101101;
        3 :   leds = 7'b1111001;
        4 :   leds = 7'b0110011 ;
        5 :   leds = 7'b1011011;
        6 :   leds = 7'b1011111;
        7 :   leds = 7'b1110000;
        8 :   leds = 7'b1111111 ;
        9 :   leds = 7'b1111011;
     default :   leds = 7'bxxxxxxx ;
     endcase
endmodule
```

<u>Exercise</u>: Modify the above Verilog code to describe a HEX-to-7 segment display decoder.

Important Notes:

- The **if-else** and **case** statements are procedural statements, and therefore must be contained in an **always** block.
- The **always** construct infers an implied memory if there are conditions under which the output of a combinational circuit is not assigned a value
- The output retains its old value, unwanted when describing combinational logic.
- Important rules
  - all inputs should be listed in the sensitivity list, and
  - all combinational circuit outputs must have assigned values.

Using Continuous Assignment Statements with the Conditional Operator

- Verilog also provides an alternative way to describe these combinational logic circuits using continuous assignment statements.
- Declared with the keyword **assign**
- LHS target of the assignment.
- RHS expression of an **assign** statement
- The *conditional operator,* denoted by **"? :"** symbol can be used.
- E.g., "*A* **?** *B* **:** *C*" reads as:
  "*<u>if</u> A is true <u>then</u> the result of the expression is B <u>else</u> the result is C*".

Example 2-24: Description of 2-to-1 MUX with conditional operator

```
module mux2_1 (in0, in1, sel, y) ;
   input  in0, in1, sel;
   output y ;

   assign y = sel ? in1 : in0 ;

endmodule
```

Example 2-25:

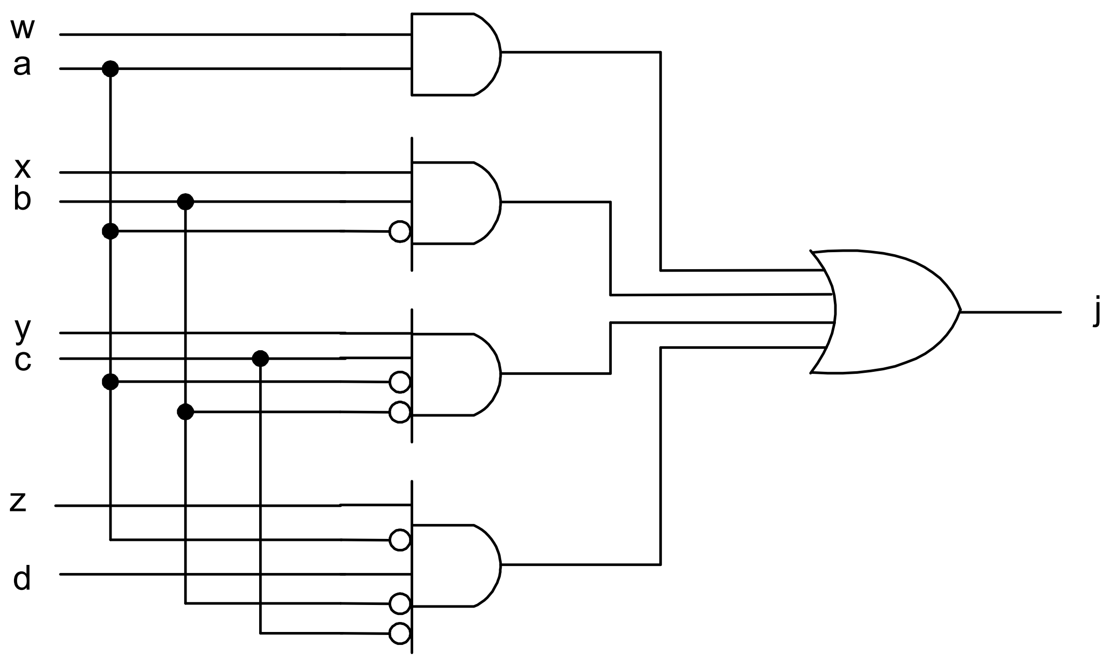
```
module mux_circuit (A, B, C, D, S, Y) ;
   input      [3:0]     A, B, C, D;
   input      [1:0]     S;
   output     [3:0]     Y;

   assign Y = S[1] ? (S[0] ? D : C) : (S[0] ? B : A)  ;

endmodule
```

Exercise: Sketch the functional block diagram the circuit described by the Verilog code below. Give the Boolean equation of Y_3 .

Example 2-26: Priority Encoder

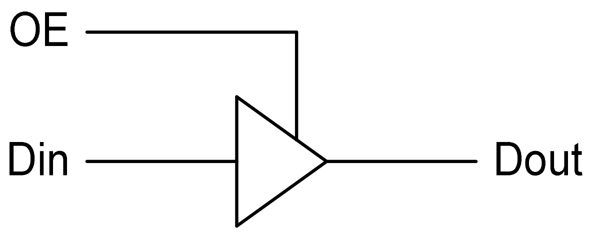


```
module p_encoder (a, b, c, d, w, x, y, z) ;
  input  a, b, c, d, w, x, y, z;
  output j ;

  assign   j  =  (a == 1) ? w :
          (b == 1) ? x :
          (c == 1) ? y :
          (d == 1) ? z : 0 ;
endmodule
```

Example 2-27: Tristate Buffer

The *high-impedance state, Z* or *z*.
A high-impedance state implies an open circuit.
Using *tri-state gates*.
A tri-state gate has one more input called *output enable, OE*.



Din	OE	Dout
X	0	Z
0	1	0
1	1	1

```
module tristatebuffer (Din, oe, Dout) ;
  input  Din, oe;
  output Dout ;

  assign   Dout  =  oe ? Din : 1‘bz ;
endmodule
```

Exercise: Write the Verilog code for the tristate buffer using sequential statement, instead of the continuous statement construct.

APPLYING HIERARCHICAL MODULAR DESIGN APPROACH

Involve the following steps.

- Derive the subsystems functional block diagrams.
- Design and simulate all subsystems.
- Apply structural modeling style to integrate these modules into the top-level module.
- Simulate this top-level module to verify that the design works.

```
module DPU (din, dsel, seg, dout) ;
   input      [15:0]   din;
   input      [1:0]    dsel;
   output     [1:7]    seg ;
   output     [7:0]    dout ;
   wire       [3:0]    Ain, Bin;
   wire       [2:0]    ALUop;
   reg        [3:0]    ALUout;
   reg        [7:0]    dout;

   assign  Ain = din[3:0];
   assign  Bin = din[7:4];
   assign  ALUop = din[10:8];

   SEG7  converter (ALUout, seg);
   ...
```

```
// ALU:
  always @ (ALUop or Ain or Bin)
    case (ALUop)
      3’b000    :  ALUout = Ain;
      3’b001    :  ALUout = Ain | Bin;
      3’b010    :  ALUout = Ain ^ Bin;
      3’b011    :  ALUout = Ain & Bin;
      3’b100    :  ALUout = Ain - Bin;
      3’b101    :  ALUout = Ain + Bin;
      default   :  ALUout = Bin;
    endcase
  …
```

```
// DEBUG_INTERFACE:
  always @ (dsel or Ain or Bin or ALUout or ALUop)
    case (dsel)
      2'b00 :  dout = { 4'b0000 , Ain };
      2'b01 :  dout = { 4'b0000 , Bin };
      2'b10 :  dout = { 4'b0000 , ALUout };
      2'b11 :  dout = { 5'b00000 , ALUop };
    endcase

endmodule
```