N26F300
VLSI SYSTEM DESIGN
(GRADUATE LEVEL)

Fall 2022

Verilog/SystemVerilog (I)

Outline

Behavioral Style of Modeling

Introduction

Boolean –Equation Based

Continuous Assignment

Operators

If-else and Case

Loop Statements

Behavioral Model

- Describe the functionality of a design
  - What the design will do
  - NOT how to build it in hardware
- Specify input-output model of logic circuit
  - Suppress details about internal structure and physical implementation

Why behavioral model?

- Increasing complexity of digital design
  - Need to evaluate the tradeoffs of various architectures
  - Avoid gate-level details by using higher level of abstract
- Encourage
  - Rapid prototyping
  - Fast function verification
  - Leave optimization to a synthesis tool

Example

- Sometime in the design process of CPU
  - For ALU
    - Not care about either ripple-carry adder or carry-lookahead adder
    - ONLY concern whether ADD instruction procesude the sum of two values stored in the register file
- Behavioral model
  - Ignores all timing information
  - Leaves specific timing to the lower level of models

Boolean-Equation-Based Behavioral Models for Combinational Logic

```
module AOI_5 _CA0 (y_out, x_in1, x_in2, x_in3, x_in4, x_in5);
  input        x_in1, x_in2, x_in3, x_in4, x_in5;
  output       y_out;

  assign y_out = ~((x_in1 & x_in2) | (x_in3 & x_in4 & x_in5));

endmodule
```

```
module AOI_5 _CA1 (y_out, x_in1, x_in2, x_in3, x_in4, x_in5, enable);
  // md ciletti
  input          x_in1, x_in2, x_in3, x_in4, x_in5, enable;
  output         y_out;

  assign y_out = enable ? ~((x_in1 & x_in2) | (x_in3 & x_in4 & x_in5)) : 1'bz;

endmodule
```

Continuous Assignment

- A handy way to model combinational logic
- Operands + operators
- Drive values to a net

  -
    ```
    wire out, eq;
        assign out = a&b;
        assign eq = (a==b);
    ```
  - `wire #10 inv = ~in;`
  - `wire [7:0] c=a+b;`
- Avoid logic loops
  - `assign a= b+ a;`
  - **asynchronous design**

Operators

{ }	concatenation
+, -, *, /	arithmetic
%	modulus
>, >=, <, <=	relational
!	logical NOT
&&	logical AND
||	logical OR
==	logical equality
!=	logical inequality
?:	conditional

~	bit-wise NOT
&	bit-wise AND
|	bit-wise OR
^	bit-wise XOR
^~, ~^	bit-wise XNOR
&	reduction AND
|	reduction OR
~&	reduction NAND
~|	reduction NOR
^	reduction XOR
~^, ^~	reduction XNOR
<<	shift left
>>	shift right

Reduction Operator

- Logical, bit-wise and unary operators

Example: a=1011, b=0010

logical	**bit-wise**	**unary**
a | | b = 1	**a | b = 1011**	**|a = 1**
a && b = 1	**a & b = 0010**	**&a = 0**

If a = 4’bx000, b=4’bx111,

|a =

&b =

Operators (SystemVerilog)

- Post-increment and pre-increment

Statement	Operation	Description
j = i++	Post-increment	j = i, i = i +1
j = ++i	Pre-increment	i = i +1, j = i
j = i--	Post-decrement	j = i, i = i -1
j= --i	Pre-decrement	i = i -1, j = i

- ++ and –– operators: blocking assignment
- Avoid race conditions by not mixing blocking and nonblocking statements
- synthesizable but depending on synthesis compiler

Operators (SystemVerilog)

- Extra assignment operators

Statement	Operation	Description
+=	Add and assign	b += 5 ≡ b = b + 5
–=	Subtract and assign	c -= 5 ≡ c = c – 5
*=	Multiply and assign	
/=	Divide and assign	
%=	Reminder and assign	
&=	Bitwise AND and assign	
|=	Bitwise OR and assign	
^=	Bitwise XOR and assign	
<<=	Bitwise left shift and assign	
>>=	Bitwise right shift and assign	

Operators (SystemVerilog)

- Assignment operators : blocking assignments
- Generally speaking, synthesizable, but some restrictions may be placed by synthesis compilers on multiplication and division
- Some compilers may not support the use of assignment operators in compound expressions

```
c += 5;    // synthesizable

c =  (m+=5); // compound form, not synthesizable
```

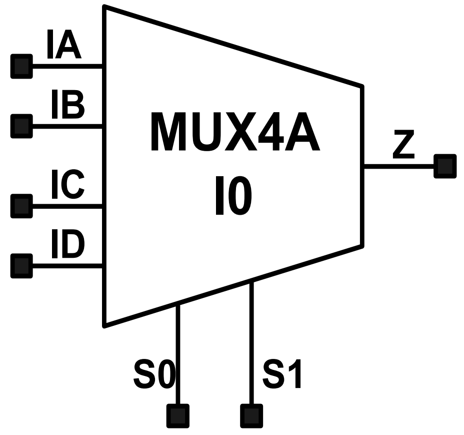
Conditional operator

- Typical conditional operator
  - <condition_expr> ? <true_expr> : <false_expr>;
  - Acts like a software if-then-else, a switching control
- Nested conditional operator
  - True_expr or false expr can itself be a conditional operation

Examples

- Conditional operator

```
assign z=({s1, s0} == 2'b00)? IA:
         ({s1, s0} == 2'b01)? IB:
         ({s1, s0} == 2'b10)? IC:
         ({s1, s0} == 2'b11)? ID: 1'bx;
```



```
assign s = ( op == ADD)? a+b: a-b;
```

Equality Operators

- == is the equality operator.

```
a = 2'b1x;
b = 2'b1x;
if(a == b)
    $display("a is equal to b");
else
    $display("a is not equal to b");
```

==	0	1	X	Z
0	1	0	X	X
1	0	1	X	X
X	X	X	X	X
Z	X	X	X	X

- === is the identity operator.

```
a = 2'b1x;
b = 2'b1x;
if(a === b)
    $display("a is identical to b");
else
    $display("a is not identical to b");
```

===	0	1	X	Z
0	1	0	0	0
1	0	1	0	0
X	0	0	1	0
Z	0	0	0	1

Concatenation Operator

- { }
- Target operands may be wires or regs
- wire [3:0] A = 4’b0010;
- reg [7:0] B = 8’b0000_1111;
- wire [7:0] C = {A, B[5:2]};
- wire [15:0] A_ones = 1111_1111_1111_1111;
- wire [15:0] B_ones = {16{1’b1}};
- wire [23:0] = {A,B[5:2],{16{1’b1}}};

Selection Statements

- Conditional if-else Statement
  - if construct
  - if-else construct
  - if-else-if construct
  - unique-if, unique0-if, and priority-if
- Case Statement
  - case construct
  - casez and casex constructs
  - unique-case, unique0-case, and priority-case

If-else Statements

- If and If-Else Statements

```
initial
if (index > 0) // Beginning of Outer if
     if (rega > regb) // Beginning of the 1st inner if
             result = rega;
     else
             result = 0; // End of the 1st inner if
else
     if (index == 0)
             $display("Note : Index is zero");
    else
             $display("Note : Index is negative");
```

Multiway Branching (Case)

```
reg [2:0]  opcode;

....
        case (opcode)
          3'b000 : result = rega + regb;
          3'b001 : result = rega - regb;
          3'b010 : result = rega * regb;
          3'b100 : result = rega / regb;
          default : begin
             result = 'bx;
             $display ("no match");
             end
        endcase
```

Enhanced if-else decisions

- Reduce ambiguities in synthesis
  - Evaluate in order or
  - Mutually exclusive in branches ➔ Evaluate in parallel
- Trap potential design error in early design phase
  - Let the designer’s intention be known to synthesis

```
logic [2:0] sel;

always_comb begin
   if          (sel == 3’b001) mux_out = a;
   else if     (sel == 3’b010) mux_out = b;
   else if     (sel == 3’b100) mux_out = c;
end
```

Enhanced if-else decisions

```
logic [2:0] sel;

always_comb begin
   unique if   (sel[0]) mux_out = a;
   else if     (sel[1]) mux_out = b;
   else if     (sel[2]) mux_out = c;
end
```

- If all decision conditions are mutually exclusive, all branches can be executed in parallel. No violation report!
- Otherwise, there in an overlap in the decision conditions. The decision must evaluated in the order listed. Violation report!

unique-if, unique0-if, and priority-if

- The keywords `unique`, `unique0`, and `priority` can be used before an `if` to perform certain violation checks.
- If the keywords `unique` or `priority` are used, a violation report shall be issued if no condition matches unless there is an explicit `else`.

```
logic a [2:0];
// values 3,5,6,7 cause a violation report
always_comb begin
  unique if ((a==0) || (a==1)) $display("0 or 1");
  else if   (a == 2) $display("2");
  else if   (a == 4) $display("4");
end
```

```
// covers all other possible values, so no violation report
always_comb begin
  priority if (a[2:1]==0) $display("0 or 1");
  else if     (a[2] == 0) $display("2 or 3");
  else        $display("4 to 7");
end
```

`unique` and `unique0`

- The **order of the decision statements is not important** because all the conditions are evaluated in parallel.
- All of the decision conditions must be mutually exclusive, otherwise a violation report must be generated
- If no branch is executed, a violation report must be generated

	Overlap	Order	No Match
`unique`	violation	-	violation
`unique0`	violation	-	-
`priority`	-	V	violation

```
// unique01.sv - unique-if

module unique01;

  logic [3:0] a;

  initial
    for (a=0; a<4; a++)
      begin
        $display("a = %b", a);

        unique if (a[0] == 0)
          $display("a[0] = 0");
        else if (a[1] == 1)
          $display("a[1] = 1");

      end

endmodule
```

```
a = 0000
a[0] = 0

a = 0001
ncsim: *W,NOCOND: Unique if violation:  Every if clause was false...

a = 0010
ncsim: *W,MCONDE: Unique if violation:  Multiple true if clauses at...
a[0] = 0

a = 0011
a[1] = 1
```

```
// unique02.sv - unique0-if

module unique02;

  logic [3:0] a;

  initial
    for (a=0; a<4; a++)
      begin
        $display("a = %b", a);

        unique0 if (a[0] == 0)
          $display("a[0] = 0");
        else if (a[1] == 1)
          $display("a[1] = 1");

      end

endmodule
```

```
file: unique02.sv
        unique0 if (a[0] == 0)
                  |
ncvlog: *E,MISEXX (unique02.sv,12|17): expecting an '=' or '<=' sign in an
assignment [9.2(IEEE)].
        module worklib.unique02:sv
                errors: 1, warnings: 0
```

`priority`

- The **order of the decision statements is important**, because all the conditions are evaluated in sequential order.
- If no branch is executed, a violation report must be generated

	Overlap	Order	No Match
`unique`	violation	-	violation
`unique0`	violation	-	-
`priority`	-	V	violation

```
// priority01.sv - priority-if

module priority01;

  logic [3:0] a;

  initial
    for (a=0; a<4; a++)
      begin
        $display("a = %b", a);

        priority if (a[0] == 0)
          $display("a[0] = 0");
        else if (a[1] == 1)
          $display("a[1] = 1");

      end

endmodule
```

```
a = 0000
a[0] = 0

a = 0001
ncsim: *W,NOCOND: Priority if violation:  Every if clause was false.

a = 0010
a[0] = 0

a = 0011
a[1] = 1
```

unique-case, unique0-case, and priority-case

- The case, casez, and casex keywords can be qualified by priority, unique, or unique0 keywords to perform certain violation checks.
- A priority-case shall act on the first match only. Unique-case and unique0-case assert that there are no overlapping case_items.

```
bit [2:0] a;
unique case(a)      // values 3,5,6,7 cause a violation report
  0,1: $display("0 or 1");
  2  : $display("2");
  4  : $display("4");
endcase
```

```
// unique03.sv - unique-case

module unique03;

  logic [3:0] a;

  initial
    for (a=0; a<4; a++)
      begin
        $display("a = %b", a);

        unique case(a)
          4'b???0: $display("a[0] = 0");       // ? = z
          4'b??1?: $display("a[1] = 1");
        endcase

      end

endmodule
```

	?	X	Z
case	Z	X	Z
case x	?	?	?
case z	?	X	?

```
a = 0000
ncsim: *W,NOCOND: Unique case violation:  Every case item expression was false...

a = 0001
ncsim: *W,NOCOND: Unique case violation:  Every case item expression was false...

a = 0010
ncsim: *W,NOCOND: Unique case violation:  Every case item expression was false...

a = 0011
ncsim: *W,NOCOND: Unique case violation:  Every case item expression was false...
```

```
// unique04.sv - unique-casez

module unique04;

  logic [3:0] a;

  initial
    for (a=0; a<4; a++)
      begin
        $display("a = %b", a);

        unique casez(a)
          4'b???0: $display("a[0] = 0");
          4'b??1?: $display("a[1] = 1");
        endcase

      end

endmodule
```

```
a = 0000
a[0] = 0

a = 0001
ncsim: *W,NOCOND: Unique casez violation:  Every case item expression was false.

a = 0010
ncsim: *W,MCONDE: Unique casez violation:  Multiple matching case item expressions at
{line=13:pos=10 and line=14:pos=10}.
a[0] = 0

a = 0011
a[1] = 1
```

```
// priority02.sv - priority-casez

module priority02;

  logic [3:0] a;

  initial
    for (a=0; a<4; a++)
      begin
        $display("a = %b", a);

        priority casez(a)
          4'b???0: $display("a[0] = 0");
          4'b??1?: $display("a[1] = 1");
        endcase

      end

endmodule
```

```
a = 0000
a[0] = 0

a = 0001
ncsim: *W,NOCOND: Priority casez violation:  Every case item expression was false.

a = 0010
a[0] = 0

a = 0011
a[1] = 1
```

Loop Statements

- for-loop
- repeat loop
- foreach loop
- while loop
- do…while loop
- forever loop

For-Loop

```
module chip (….);   // Verilog style loops
reg [7:0] i;
int j, k;

always (@posedge clk) begin
 for (i=0; i<=15; i=i+1)
   for (j=511;j>=0; j=j-1) begin
                                    ……
   end
 end

 always (@posedge clk) begin
  for (k=1; k<=1024; k= k+2) begin
    ……
  end
 end
```

For-Loop (Enhanced)

```
module chip (….);   // SystemVerilog style loops

always_ff (@posedge clk) begin
 for (bit [4:0] i=0; i<=15; i++)      // local loop variable
     . . .
end

always_ff (@posedge clk) begin
  for (int i=1; k<=1024; i+=2) begin   // local loop variable
    ……
  end
 end
```

- Local loop variables (LLVs) prevent interference
- LLVs are automatic – no static storage, created when invoked, destroyed when exits
- LLVs cannot be dumped to VCD files

For-Loop

- Reference outside of a loop

```
always_comb begin
  int lo_bit;  // localabe to the block
  for (lo_bit=0; lo_bit<=63; lo_bit++)  begin
     if (data[lo_bit]) break;
     . . .
  end

   if (lo_bit > 7) // retain its last value
     ……

  end
```

For-Loop

- Hierarchically referencing variable declared in for-loops

```
always_ff @(posedge clk) begin
   for (int i=1; k<=1024; i+=2) begin
        ……   // i cannot be referenced hierarchically
   end
end
```

```
always_ff @(posedge clk) begin: loopTest
     int i; // i can be referenced hierarchically
     for (i=1; k<=1024; i+=2) begin
        ……   // i cannot be referenced hierarchically
     end
end
```

i can be referenced with the last portion of the hierarchy path ending with .loopTest.i.

`foreach` loop

- The foreach-loop construct specifies iteration over the elements of an array.
- Its argument is an identifier that designates any type of array followed by a comma-separated list of loop variables enclosed in square brackets.
- Each loop variable corresponds to one of the dimensions of the array.
- The foreach-loop is similar to a repeat-loop that uses the array bounds to specify the repeat count instead of an expression.

`foreach` loop

```
string words [2] = '{ "hello", "world" };
int prod [1:8] [1:3];

foreach(words[j])
  $display(j, words[j]);     // print each index and value

foreach(prod[k, m])
  prod[k][m] = k * m;        // initialize
```

```
//      1   2   3          3     4        1    2      -> Dimension numbers
int A [2][3][4]; bit [3:0][2:1] B [5:1][4];

foreach( A [ i, j, k ] ) ...

foreach( B [ q, r, , s ] ) ...
```

```
// foreach01.sv foreach loop

module foreach01;

  logic [3:0] a [2][3];

  initial begin

    foreach (a[i, j])
      a[i][j] = i * 3 + j;

    foreach (a[i, j])
      $display("a[%0d][%0d] = %b", i, j, a[i][j]);

  end

endmodule
```

```
a[0][0] = 0000
a[0][1] = 0001
a[0][2] = 0010
a[1][0] = 0011
a[1][1] = 0100
a[1][2] = 0101
```

```
// foreach02.sv foreach loop

module foreach02;

  logic [3:0] a [2][3];

  initial begin

    foreach (a[i, j])
      a[i][j] = i * 3 + j;

    foreach (a[i,])
      foreach (a[,j])
        $display("a[%0d][%0d] = %b", i, j, a[i][j]);
  end

endmodule
```

```
a[0][0] = 0000
a[0][1] = 0001
a[0][2] = 0010
a[1][0] = 0011
a[1][1] = 0100
a[1][2] = 0101
```

```
// foreach03.sv foreach loop

module foreach03;

  logic [7:0] a [];

  initial begin

    a = new[3];
    foreach (a[i])
      begin
        a[i] = $random;
        $display("a[%0d] = %b", i, a[i]);
      end

    a = new[5];
    foreach (a[i])
      begin
        a[i] = $random;
        $display("a[%0d] = %b", i, a[i]);
      end

  end

endmodule
```

```
a[0] = 00100100
a[1] = 10000001
a[2] = 00001001

a[0] = 01100011
a[1] = 00001101
a[2] = 10001101
a[3] = 01100101
a[4] = 00010010
```

`do…while` loop

- The while-loop executes a statement until an expression becomes false. If the expression starts out false, the statement shall not be executed at all.
- The do…while-loop differs from the while-loop in that a do…while-loop tests its control expression at the end of the loop.

```
// while01.sv - do while loop

module while01();

  byte a = 0;

  initial
    begin

      do begin
        $display ("Current value of a = %g", a);
        a++;
      end while  (a < 10);

      #1 $finish;
    end

endmodule
```

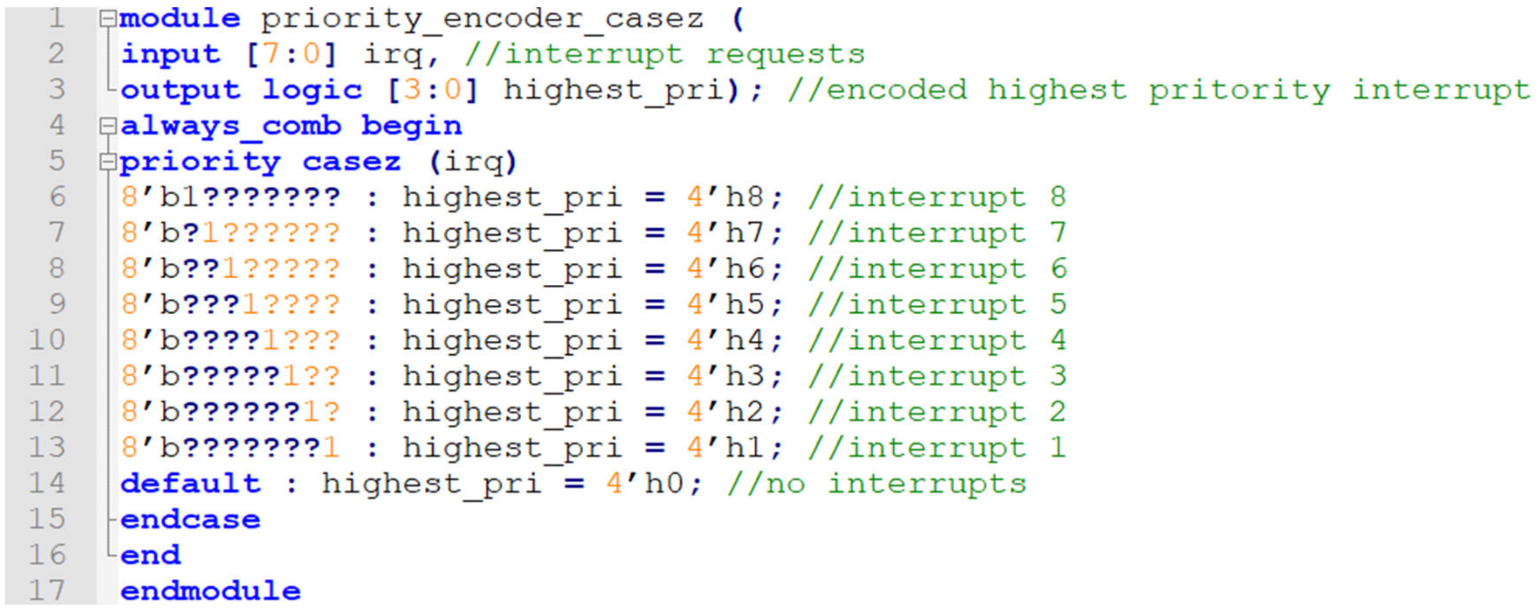
```
Current value of a = 0
Current value of a = 1
Current value of a = 2
Current value of a = 3
Current value of a = 4
Current value of a = 5
Current value of a = 6
Current value of a = 7
Current value of a = 8
Current value of a = 9
```

填寫第二階段的分組名單(每個人各自填寫)

- 後選課階段(第二階段)
- 確認分組成員
- 設定Moodle的分組
- 作為後續線上活動的安排依據
- Due: 2:00pm 9/30

Supplement on casez and casex

- casez: a case expression allows to use ? or Z as don't care instead of as logic value



```
module priority_encoder_casez (
input [7:0] irq, //interrupt requests
output logic [3:0] highest_pri); //encoded highest pritority interrupt
always_comb begin
priority casez (irq)
8'b1??????? : highest_pri = 4'h8; //interrupt 8
8'b?1?????? : highest_pri = 4'h7; //interrupt 7
8'b??1????? : highest_pri = 4'h6; //interrupt 6
8'b???1???? : highest_pri = 4'h5; //interrupt 5
8'b????1??? : highest_pri = 4'h4; //interrupt 4
8'b?????1?? : highest_pri = 4'h3; //interrupt 3
8'b??????1? : highest_pri = 4'h2; //interrupt 2
8'b???????1 : highest_pri = 4'h1; //interrupt 1
default : highest_pri = 4'h0; //no interrupts
endcase
end
endmodule
```

- Synthesizable, but not able to differentiate "don't know" bit and "don't case" one; not recommended.

Notes on casez and casex

- casez treats bits having a Z value as don't case.
- casex treats bit have a X or Z value as don't case.
- Therefore, synthesizable.
- However, errors can arise when an uninitialized signal properly takes on an X values, but the unknown signal is then treated by a casex statement as a don't case.
- Simulation/synthesis results mismatch, as the synthesized hardware will only have zeros and ones, never any X values.
- Not recommended to be used in synthesis!

case with inside: avoid the problem

```
module insidedemo(input [2:0] OPCODE,
  output logic [3:0]  ENABLE_BUS);
  always_comb
    case (OPCODE) inside
      3'b0xx: ENABLE_BUS = 4'd0;
      3'b100: ENABLE_BUS = 4'd1;
      3'b101: ENABLE_BUS = 4'd2;
      default: ENABLE_BUS = 4'd7;
    endcase
endmodule

module casexdemo (input [2:0] OPCODE,
  output logic [3:0]  ENABLE_BUS);
  always_comb
    casex (OPCODE)
      3'b0xx: ENABLE_BUS = 4'd0;
      3'b100: ENABLE_BUS = 4'd1;
      3'b101: ENABLE_BUS = 4'd2;
      default: ENABLE_BUS = 4'd7;
    endcase
endmodule

module tb_insidedemo;
  reg [2:0]  OPCODE;
  logic  [3:0]   ENABLE_BUS1, ENABLE_BUS2;
  insidedemo  UUT1(OPCODE, ENABLE_BUS1);
  casexdemo   UUT2(OPCODE, ENABLE_BUS2);
  initial begin
    $monitor("OPCODE = %b, BUS1 = %d, BUS2 = %d", OPCODE, ENABLE_BUS1, ENABLE_BUS2);
    OPCODE = 3'bx;
    #1 OPCODE = 3'b0xx;
    #1 OPCODE = 3'b000;
    #1 OPCODE = 3'b001;
    #1 OPCODE = 3'b100;
  end
endmodule
```

- X or Z inputs are not treated as don’t case.
- Make error conditions easier to be detected and debugged.

```
/* Results: difference at time 0. Unknow opcode
takes the default clause with case inside but
matches the first branch for casex.
# OPCODE = xxx, BUS1 = 7, BUS2 = 0;
# OPCODE = 0xx, BUS1 = 0, BUS2 = 0;
# OPCODE = 000, BUS1 = 0, BUS2 = 0;
# OPCODE = 001, BUS1 = 0, BUS2 = 0;
# OPCODE = 100, BUS1 = 1, BUS2 = 1;
```

*** Adapted from "Digital-integrated Circuit Using Verilog and SystemVerilog,” 2015