# VERILOG HDL 11

*Key terms and concepts:* syntax and semantics • operators • hierarchy • procedures and assignments • timing controls and delay • tasks and functions • control statements • logic-gate modeling • modeling delay • altering parameters • other Verilog features: PLI

*History:* Gateway Design Automation developed Verilog as a simulation language • Cadence purchased Gateway in 1989 • Open Verilog International (OVI) was created to develop the Verilog language as an IEEE standard • Verilog LRM, IEEE Std 1364-1995 • problems with a normative LRM

## 11.1 A Counter

*Key terms and concepts:* Verilog **keywords** • simulation language • compilation • interpreted, compiled, and native code simulators

```
`timescale 1ns/1ns // Set the units of time to be nanoseconds.     //1
module counter;                                                    //2
  reg clock; // Declare a reg data type for the clock.             //3
  integer count; // Declare an integer data type for the count.    //4
initial // Initialize things; this executes once at t=0.          //5
  begin                                                            //6
    clock = 0; count = 0; // Initialize signals.                   //7
    #340 $finish; // Finish after 340 time ticks.                 //8
  end                                                              //9
/* An always statement to generate the clock; only one statement
follows the always so we don't need a begin and an end. */       //10
always                                                           //11
  #10 clock = ~ clock; // Delay (10ns) is set to half the clock
cycle.                                                           //12
/* An always statement to do the counting; this executes at the same
time (concurrently) as the preceding always statement. */        //13
always                                                           //14
  begin                                                          //15
```

```
    // Wait here until the clock goes from 1 to 0.                  //16
    @ (negedge clock);                                              //17
    // Now handle the counting.                                     //18
    if (count == 7)                                                 //19
      count = 0;                                                    //20
    else                                                            //21
      count = count + 1;                                            //22
    $display("time = ",$time," count = ", count);                   //23
  end                                                               //24
endmodule                                                           //25
```

## 11.2 Basics of the Verilog Language

*Key terms and concepts:* **identifier** • Verilog is case-sensitive • system tasks and functions begin with a dollar sign '$'

```
identifier ::= simple_identifier | escaped_identifier
simple_identifier ::= [a-zA-Z][a-zA-Z_$]


escaped_identifier ::=
  \ {Any_ASCII_character_except_white_space} white_space
white_space ::= space | tab | newline
```

```
module identifiers;                                                 //1
/* Multiline comments in Verilog                                    //2
  look like C comments and // is OK in here. */                     //3
// Single-line comment in Verilog.                                  //4
reg legal_identifier,two__underscores;                              //5
reg _OK,OK_,OK_$,OK_123,CASE_SENSITIVE, case_sensitive;             //6
reg \/clock ,\a*b ; // Add white_space after escaped identifier.    //7
//reg $_BAD,123_BAD; // Bad names even if we declare them!          //8
initial begin                                                       //9
legal_identifier = 0; // Embedded underscores are OK,              //10
two__underscores = 0; // even two underscores in a row.           //11
_OK = 0; // Identifiers can start with underscore                 //12
OK_ = 0; // and end with underscore.                              //13
OK$ = 0; // $ sign is OK, but beware foreign keyboards.           //14
OK_123 =0; // Embedded digits are OK.                             //15
CASE_SENSITIVE = 0; // Verilog is case-sensitive (unlike VHDL).    //16
```

```
case_sensitive = 1;                                          //17
\/clock = 0; // An escaped identifier with \ breaks rules,   //18
\a*b = 0; // but be careful to watch the spaces!             //19
$display("Variable CASE_SENSITIVE= %d",CASE_SENSITIVE);      //20
$display("Variable case_sensitive= %d",case_sensitive);      //21
$display("Variable \/clock = %d",\/clock );                  //22
$display("Variable \\a*b = %d",\a*b );                       //23
end                                                          //24
endmodule                                                    //25
```

### 11.2.1 Verilog Logic Values

*Key terms and concepts:* predefined **logic-value system or value set** • four logic values: `'0'`, `'1'`, `'x'`, and `'z'` (lowercase) • uninitialized or an unknown logic value (either `'1'`, `'0'`, `'z'`, or in a state of change) • high-impedance value (usually treated as an `'x'` value) • internal logic-value system resolves conflicts between drivers on the same node

### 11.2.2 Verilog Data Types

*Key terms and concepts:* **data types** • **nets** • **wire** and **tri** (identical) • **supply1** and **supply0** (positive and negative power) • default initial value for a `wire` is `'z'` • **integer**, **time**, **event**, and **real** data types • **register** data type (keyword `reg`) • default initial value for a `reg` is `'x'` • a `reg` is not always equivalent to a register, flip-flop, or latch • **scalar** • **vector** • **range** • access (or **expand**) bits in a vector using a **bit-select**, or as a contiguous subgroup of bits using a **part-select** • no multidimensional arrays • **memory** data type is an array of registers • integer arrays • time arrays • no real arrays

```
module declarations_1;                                       //1
wire pwr_good, pwr_on, pwr_stable; // Explicitly declare wires.  //2
integer i; // 32-bit, signed (2's complement).               //3
time t; // 64-bit, unsigned, behaves like a 64-bit reg.      //4
event e; // Declare an event data type.                      //5
real r; // Real data type of implementation defined size.    //6
// An assign statement continuously drives a wire:           //7
assign pwr_stable = 1'b1; assign pwr_on = 1; // 1 or 1'b1    //8
assign pwr_good = pwr_on & pwr_stable;                       //9
initial begin                                                //10
i = 123.456;   // There must be a digit on either side       //11
r = 123456e-3; // of the decimal point if it is present.     //12
t = 123456e-3; // Time is rounded to 1 second by default.    //13
```

```
$display("i=%0g",i," t=%6.2f",t," r=%f",r);                    //14
#2 $display("TIME=%0d",$time," ON=",pwr_on,                    //15
  " STABLE=",pwr_stable," GOOD=",pwr_good);                    //16
$finish; end                                                   //17
endmodule                                                      //18

module declarations_2;                                          //1
reg Q, Clk; wire D;                                             //2
// Drive the wire (D):                                          //3
assign D = 1;                                                   //4
// At a +ve clock edge assign the value of wire D to the reg Q: //5
always @(posedge Clk) Q = D;                                    //6
initial Clk = 0; always #10 Clk = ~ Clk;                        //7
initial begin #50; $finish; end                                 //8
always begin                                                    //9
$display("T=%2g", $time," D=",D," Clk=",Clk," Q=",Q); #10;end   //10
endmodule                                                      //11

module declarations_3;                                          //1
reg a,b,c,d,e;                                                  //2
initial begin                                                   //3
  #10; a = 0;b = 0;c = 0;d = 0; #10; a = 0;b = 1;c = 1;d = 0;   //4
  #10; a = 0;b = 0;c = 1;d = 1; #10; $stop;                     //5
end                                                             //6
always begin                                                    //7
  @(a or b or c or d) e = (a|b)&(c|d);                          //8
  $display("T=%0g",$time," e=",e);                              //9
end                                                            //10
endmodule                                                      //11

module declarations_4;                                          //1
wire Data; // A scalar net of type wire.                        //2
wire [31:0] ABus, DBus; // Two 32-bit-wide vector wires:        //3
// DBus[31] = leftmost = most-significant bit = msb             //4
// DBus[0] = rightmost = least-significant bit = lsb            //5
// Notice the size declaration precedes the names.              //6
// wire [31:0] TheBus, [15:0] BigBus; // This is illegal.       //7
reg [3:0] vector; // A 4-bit vector register.                   //8
reg [4:7] nibble; // msb index < lsb index is OK.               //9
integer i;                                                     //10
initial begin                                                  //11
i = 1;                                                         //12
vector = 'b1010; // Vector without an index.                   //13
nibble = vector; // This is OK too.                            //14
```

```
#1; $display("T=%0g",$time," vector=", vector," nibble=", nibble);//15
#2; $display("T=%0g",$time," Bus=%b",DBus[15:0]);                 //16
end                                                               //17
assign DBus [1] = 1; // This is a bit-select.                     //18
assign DBus [3:0] = 'b1111; // This is a part-select.             //19
// assign DBus [0:3] = 'b1111; // Illegal : wrong direction.      //20
endmodule                                                         //21
```

```
module declarations_5;                                            //1
reg [31:0] VideoRam [7:0]; // An 8-word by 32-bit wide memory.    //2
initial begin                                                     //3
VideoRam[1] = 'bxz; // We must specify an index for a memory.     //4
VideoRam[2] = 1;                                                  //5
VideoRam[7] = VideoRam[VideoRam[2]]; // Need 2 clock cycles for this//6
VideoRam[8] = 1; // Careful! the compiler won't complain about this!//7
// Verify what we entered:                                        //8
$display("VideoRam[0] is %b",VideoRam[0]);                        //9
$display("VideoRam[1] is %b",VideoRam[1]);                        //10
$display("VideoRam[2] is %b",VideoRam[2]);                        //11
$display("VideoRam[7] is %b",VideoRam[7]);                        //12
end                                                               //13
endmodule                                                         //14
```

```
module declarations_6;                                            //1
integer Number [1:100]; // Notice that size follows name          //2
time Time_Log [1:1000]; // - as in an array of reg.               //3
// real Illegal [1:10]; // Illegal. There are no real arrays.     //4
endmodule                                                         //5
```

### 11.2.3 Other Wire Types

*Key terms and concepts:* `wand`, `wor`, `triand`, and `trior` model wired logic • ECL or EPROM, • one area in which the logic values `'z'` and `'x'` are treated differently • `tri0` and `tri1` model resistive connections to VSS or VDD • `trireg` is like a `wire` but associates some capacitance with the net and models charge storage • scalared and vectored are properties of vectors • `small`, `medium`, and `large` model the charge strength of `trireg`

### 11.2.4 Numbers

*Key terms and concepts:* **constant numbers** are integer or real constants • **integer constants** are written as `width'radix value`• **radix** (or base): **decimal** (`d` or `D`), **hex** (`h` or `H`), **octal** (`o` or `O`), or **binary** (`b` or `B`) • **sized** or **unsized** (implementation dependent) • `1'bx` and `1'bz` for `'x'`

and 'z' • **parameter** (local scope) • **real constants** 100.0 or 1e2 (IEEE Std 754-1985) • reals round to the nearest integer, ties away from zero

```
module constants;                                                    //1
parameter H12_UNSIZED = 'h 12; // Unsized hex 12 = decimal 18.       //2
parameter H12_SIZED = 6'h 12; // Sized hex 12 = decimal 18.          //3
// Note: a space between base and value is OK.                       //4
// Note: ‘’ (single apostrophes) are not the same as the ' character.//5
parameter D42 = 8'B0010_1010; // bin 101010 = dec 42                 //6
// OK to use underscores to increase readability.                    //7
parameter D123 = 123; // Unsized decimal (the default).             //8
parameter D63 = 8'o 77; // Sized octal, decimal 63.                  //9
// parameter ILLEGAL = 1'o9; // No 9's in octal numbers!            //10
// A = 'hx and B = 'ox assume a 32 bit width.                       //11
parameter A = 'h x, B = 'o x, C = 8'b x, D = 'h z, E = 16'h ????; //12
// Note the use of ? instead of z, 16'h ???? is the same as 16'h
zzzz.                                                               //13
// Also note the automatic extension to a width of 16 bits.         //14
reg [3:0] B0011,Bxxx1,Bzzz1;real R1,R2,R3; integer I1,I3,I_3;       //15
parameter BXZ = 8'b1x0x1z0z;                                        //16
initial begin                                                       //17
B0011 = 4'b11; Bxxx1 = 4'bx1; Bzzz1 = 4'bz1;// Left padded.         //18
R1 = 0.1e1; R2 = 2.0; R3 = 30E-01; // Real numbers.                 //19
I1 = 1.1; I3 = 2.5; I_3 = -2.5; // IEEE rounds away from 0.         //20
end                                                                 //21
initial begin #1;                                                   //22
$display                                                            //23
("H12_UNSIZED, H12_SIZED (hex) = %h, %h",H12_UNSIZED, H12_SIZED); //24
$display("D42 (bin) = %b",D42," (dec) = %d",D42);                   //25
$display("D123 (hex) = %h",D123," (dec) = %d",D123);                //26
$display("D63 (oct) = %o",D63);                                     //27
$display("A (hex) = %h",A," B (hex) = %h",B);                       //28
$display("C (hex) = %h",C," D (hex) = %h",D," E (hex) = %h",E);     //29
$display("BXZ (bin) = %b",BXZ," (hex) = %h",BXZ);                   //30
$display("B0011, Bxxx1, Bzzz1 (bin) = %b, %b, %b",B0011,Bxxx1,Bzzz1);//31
$display("R1, R2, R3 (e, f, g) = %e, %f, %g", R1, R2, R3);          //32
$display("I1, I3, I_3 (d) = %d, %d, %d", I1, I3, I_3);              //33
end                                                                 //34
endmodule                                                           //35
```

### 11.2.5 Negative Numbers

*Key terms and concepts:* Integers are **signed** (two's complement) or **unsigned** • Verilog only "keeps track" of the sign of a negative constant if it is (1) assigned to an `integer` or (2) assigned to a `parameter` without using a base (essentially the same thing) • in other cases a negative constant is treated as an unsigned number • once Verilog "loses" a sign, keeping track of signed numbers is your responsibility

```
module negative_numbers;                                          //1
parameter PA = -12, PB = -'d12, PC = -32'd12, PD = -4'd12;        //2
integer IA , IB , IC , ID ; reg [31:0] RA , RB , RC , RD ;        //3
initial begin #1;                                                 //4
IA = -12; IB = -'d12; IC = -32'd12; ID = -4'd12;                  //5
RA = -12; RB = -'d12; RC = -32'd12; RD = -4'd12; #1;              //6
$display("          parameter    integer   reg[31:0]");           //7
$display ("-12     =",PA,IA,,,RA);                                //8
$displayh("         ",,,,PA,,,,IA,,,,,RA);                        //9
$display ("-'d12   =",,PB,IB,,,RB);                              //10
$displayh("         ",,,,PB,,,,IB,,,,,RB);                       //11
$display ("-32'd12 =",,PC,IC,,,RC);                              //12
$displayh("         ",,,,PC,,,,IC,,,,,RC);                       //13
$display ("-4'd12  =",,,,,,,,,,PD,ID,,,RD);                      //14
$displayh("         ",,,,,,,,,,,PD,,,,ID,,,,,RD);                //15
end                                                               //16
endmodule                                                         //17
```

```
           parameter    integer   reg[31:0]
-12     =        -12        -12  4294967284
             fffffff4   fffffff4    fffffff4
-'d12   = 4294967284        -12  4294967284
             fffffff4   fffffff4    fffffff4
-32'd12 = 4294967284        -12  4294967284
             fffffff4   fffffff4    fffffff4
-4'd12  =          4        -12  4294967284
                   4   fffffff4    fffffff4
```

### 11.2.6 Strings

*Key terms and concepts:* ISO/ANSI defines characters, but not their appearance • problem characters are quotes and accents • **string constants** • **define** directive is a compiler directive (global scope)

```
module characters; /*                                                  //1
" is ASCII 34 (hex 22), double quote.                                  //2
' is ASCII 39 (hex 27), tick or apostrophe.                            //3
/ is ASCII 47 (hex 2F), forward slash.                                 //4
\ is ASCII 92 (hex 5C), back slash.                                    //5
` is ASCII 96 (hex 60), accent grave.                                  //6
| is ASCII 124 (hex 7C), vertical bar.                                 //7
There are no standards for the graphic symbols for codes above 128.    //8
´ is 171 (hex AB), accent acute in almost all fonts.                   //9
“ is 210 (hex D2), open double quote, like 66 (in some fonts).         //10
” is 211 (hex D3), close double quote, like 99 (in some fonts).        //11
‘ is 212 (hex D4), open single quote, like 6 (in some fonts).          //12
’ is 213 (hex D5), close single quote, like 9 (in some fonts).         //13
*/ endmodule                                                           //14
```

```
module text;                                                           //1
parameter A_String = "abc"; // string constant, must be on one line   //2
parameter Say = "Say \"Hey!\"";                                        //3
// use escape quote \" for an embedded quote                           //4
parameter Tab = "\t";   // tab character                               //5
parameter NewLine = "\n"; // newline character                        //6
parameter BackSlash = "\\"; // back slash                             //7
parameter Tick = "\047"; // ASCII code for tick in octal              //8
// parameter Illegal = "\500";// illegal - no such ASCII code         //9
initial begin                                                          //10
$display("A_String(str) = %s ",A_String," (hex) = %h ",A_String);      //11
$display("Say = %s ",Say," Say \"Hey!\"");                             //12
$display("NewLine(str) = %s ",NewLine," (hex) = %h ",NewLine);         //13
$display("\\(str) = %s ",BackSlash," (hex) = %h ",BackSlash);          //14
$display("Tab(str) = %s ",Tab," (hex) = %h ",Tab,"1 newline...");      //15
$display("\n");                                                        //16
$display("Tick(str) = %s ",Tick," (hex) = %h ",Tick);                  //17
#1.23; $display("Time is %t", $time);                                  //18
```

```
end                                                          //19
endmodule                                                    //20

module define;                                               //1
`define G_BUSWIDTH 32 // Bus width parameter (G_ for global).  //2
/* Note: there is no semicolon at end of a compiler directive. The
character ` is ASCII 96 (hex 60), accent grave, it slopes down from
left to right. It is not the tick or apostrophe character ' (ASCII 39
or hex 27)*/                                                 //3
wire [`G_BUSWIDTH:0]MyBus; // A 32-bit bus.                  //4
endmodule                                                    //5
```

## 11.3 Operators

*Key terms and concepts:* three types of operators: unary, binary, or a single ternary operator • similar to C programming language (but no ++ or --)

**Verilog unary operators**

| Operator | Name | Examples |
|---|---|---|
| ! | logical negation | !123 is 'b0 [0, 1, or x for ambiguous; legal for real] |
| ~ | bitwise unary negation | ~1'b10xz is 1'b01xx |
| & | unary reduction and | & 4'b1111 is 1'b1, & 2'bx1 is 1'bx, & 2'bz1 is 1'bx |
| ~& | unary reduction nand | ~& 4'b1111 is 1'b0, ~& 2'bx1 is 1'bx |
| \| | unary reduction or | Note: |
| ~\| | unary reduction nor | Reduction is performed left (first bit) to right |
| ^ | unary reduction xor | Beware of the non-associative reduction operators |
| ~^ ^~ | unary reduction xnor | z is treated as x for all unary operators |
| + | unary plus | +2'bxz is +2'bxz [+m is the same as m; legal for real] |
| - | unary minus | -2'bxz is x [-m is unary minus m; legal for real] |

```
module operators;                                            //1
parameter A10xz = {1'b1,1'b0,1'bx,1'bz}; // Concatenation and  //2
parameter A01010101 = {4{2'b01}}; // replication, illegal for real.//3
// Arithmetic operators: +, -, *, /, and modulus %           //4
parameter A1 = (3+2) %2; // The sign of a % b is the same as sign of
a.                                                           //5
// Logical shift operators: << (left), >> (right)            //6
parameter A2 = 4 >> 1; parameter A4 = 1 << 2; // Note: zero fill.  //7
```

**Verilog operators (in increasing order of precedence)**

?: (conditional) [legal for real; associates right to left (others associate left to right)]
| | (logical or) [A smaller operand is zero-filled from its msb (0-fill); legal for real]
&& (logical and)[0-fill, legal for real]
| (bitwise or) ~| (bitwise nor) [0-fill]
^ (bitwise xor) ^~ ~^ (bitwise xnor, equivalence) [0-fill]
& (bitwise and) ~& (bitwise nand) [0-fill]
== (logical) != (logical) === (case) !== (case) [0-fill, logical versions are legal for real]
< (lt) <= (lt or equal) > (gt) >= (gt or equal) [0-fill, all arelegal for real]
<< (shift left) >> (shift right) [zero fill; no -ve shifts; shift by x or z results in unknown]
+ (addition) - (subtraction) [if any bit is x or z for + - * / % then entire result is unknown]
* (multiply) / (divide) % (modulus) [integer divide truncates fraction; + - * / legal for real]
Unary operators: ! ~ & ~& | ~| ^ ~^ ^~ + -

```
// Relational operators: <, <=, >, >=                                    //8
initial if (1 > 2) $stop;                                                //9
// Logical operators: ! (negation), && (and), || (or)                    //10
parameter B0 = !12; parameter B1 = 1 && 2;                               //11
reg [2:0] A00x; initial begin A00x = 'b111; A00x = !2'bx1;end            //12
parameter C1 = 1 || (1/0); /* This may or may not cause an              //13
error: the short-circuit behavior of && and || is undefined. An          //14
evaluation including && or || may stop when an expression is known       //15
to be true or false. */                                                  //16
// == (logical equality), != (logical inequality)                        //17
parameter Ax = (1==1'bx); parameter Bx = (1'bx!=1'bz);                    //18
parameter D0 = (1==0); parameter D1 = (1==1);                            //19
// === case equality, !== (case inequality)                              //20
// The case operators only return true (1) or false (0).                 //21
parameter E0 = (1===1'bx); parameter E1 = 4'b01xz === 4'b01xz;            //22
parameter F1 = (4'bxxxx === 4'bxxxx);                                    //23
// Bitwise logical operators:                                            //24
// ~ (negation), & (and), | (inclusive or),                              //25
// ^ (exclusive or), ~^ or ^~ (equivalence)                              //26
parameter A00 = 2'b01 & 2'b10;                                           //27
// Unary logical reduction operators:                                    //28
// & (and), ~& (nand), | (or), ~| (nor),                                 //29
// ^ (xor), ~^ or ^~ (xnor)                                              //30
parameter G1= & 4'b1111;                                                 //31
// Conditional expression f = a ? b : c [if (a) then f=b else f=c]       //32
```

```
// if a=(x or z), then (bitwise) f=0 if b=c=0, f=1 if b=c=1, else f≠X33
reg H0, a, b, c; initial begin a=1; b=0; c=1; H0=a?b:c; end        //34
reg[2:0] J01x, Jxxx, J01z, J011;                                   //35
initial begin Jxxx = 3'bxxx; J01z = 3'b01z; J011 = 3'b011;         //36
J01x = Jxxx ? J01z : J011; end // A bitwise result.               //37
initial begin #1;                                                  //38
$display("A10xz=%b",A10xz," A01010101=%b",A01010101);              //39
$display("A1=%0d",A1," A2=%0d",A2," A4=%0d",A4);                   //40
$display("B1=%b",B1," B0=%b",B0," A00x=%b",A00x);                  //41
$display("C1=%b",C1," Ax=%b",Ax," Bx=%b",Bx);                      //42
$display("D0=%b",D0," D1=%b",D1);                                  //43
$display("E0=%b",E0," E1=%b",E1," F1=%b",F1);                      //44
$display("A00=%b",A00," G1=%b",G1," H0=%b",H0);                    //45
$display("J01x=%b",J01x); end                                      //46
endmodule                                                          //47
```

### 11.3.1 Arithmetic

*Key terms and concepts:* arithmetic on *n*-bit objects is performed modulo $2^n$ • arithmetic on vectors (`reg` or `wire`) are predefined • once Verilog “loses” a sign, it cannot get it back

```
module modulo; reg [2:0] Seven;                                    //1
initial begin                                                      //2
#1 Seven = 7; #1 $display("Before=", Seven);                       //3
#1 Seven = Seven + 1; #1 $display("After =", Seven);               //4
end                                                                //5
endmodule                                                          //6
```

```
module LRM_arithmetic;                                             //1
integer IA, IB, IC, ID, IE; reg [15:0] RA, RB, RC;                 //2
initial begin                                                      //3
IA = -4'd12; RA =  IA / 3; // reg is treated as unsigned.          //4
RB = -4'd12; IB =  RB / 3; //                                      //5
IC = -4'd12 / 3; RC = -12 / 3; // real is treated as signed        //6
ID =    -12 / 3; IE =  IA / 3; // (two’s complement).              //7
end                                                                //8
initial begin #1;                                                  //9
$display("                        hex    default");               //10
$display("IA = -4'd12    = %h%d",IA,IA);                           //11
$display("RA = IA / 3    =     %h      %d",RA,RA);                 //12
$display("RB = -4'd12    =     %h      %d",RB,RB);                 //13
$display("IB = RB / 3    = %h%d",IB,IB);                           //14
$display("IC = -4'd12 / 3 = %h%d",IC,IC);                          //15
```

```
$display("RC = -12 / 3   =     %h      %d",RC,RC);                  //16
$display("ID = -12 / 3   = %h%d",ID,ID);                           //17
$display("IE =  IA / 3   = %h%d",IE,IE);                           //18
end                                                                //19
endmodule                                                          //20
```

```
                        hex    default
IA = -4'd12     = fffffff4       -12
RA = IA / 3     =     fffc      65532
RB = -4'd12     =     fff4      65524
IB = RB / 3     = 00005551      21841
IC = -4'd12 / 3 = 55555551 1431655761
RC = -12 / 3    =     fffc      65532
ID = -12 / 3    = fffffffc         -4
IE =  IA / 3    = fffffffc         -4
```

## 11.4 Hierarchy

*Key terms and concepts:* **module** • the **module interface** interconnects two Verilog modules using **ports** • ports must be explicitly declared as **input**, **output**, or **inout** • a `reg` cannot be `input` or `inout` port (to connection of a `reg` to another `reg`) • **instantiation** • ports are linked using **named association** or **positional association** • **hierarchical name** ( `m1.weekend`) • The compiler will first search downward (or inward) then upward (outward)

**Verilog ports.**

| Verilog port | input | output | inout |
|---|---|---|---|
| **Characteristics** | `wire` (or other net) | `reg` or `wire` (or other net)<br>We *can* read an output port inside a module | `wire` (or other net) |

```
module holiday_1(sat, sun, weekend);                               //1
  input sat, sun; output weekend;                                  //2
  assign weekend = sat | sun;                                      //3
endmodule                                                          //4
```

```
`timescale 100s/1s // Units are 100 seconds with precision of 1s. //1
module life; wire [3:0] n; integer days;                           //2
  wire wake_7am, wake_8am; // Wake at 7 on weekdays else at 8.     //3
  assign n = 1 + (days % 7); // n is day of the week (1-7)         //4
always@(wake_8am or wake_7am)                                      //5
```

```
  $display("Day=",n," hours=%0d ",($time/36)%24," 8am = ",          //6
    wake_8am,"  7am = ",wake_7am,"  m2.weekday = ", m2.weekday);    //7
  initial days = 0;                                                 //8
  initial begin #(24*36*10);$finish; end // Run for 10 days.        //9
  always #(24*36) days = days + 1; // Bump day every 24hrs.        //10
  rest m1(n, wake_8am); // Module instantiation.                   //11
// Creates a copy of module rest with instance name m1,            //12
// ports are linked using positional notation.                     //13
  work m2(.weekday(wake_7am), .day(n));                             //14
// Creates a copy of module work with instance name m2,            //15
// Ports are linked using named association.                       //16
endmodule                                                         //17

module rest(day, weekend); // Module definition.                    //1
// Notice the port names are different from the parent.            //2
  input [3:0] day; output weekend; reg weekend;                     //3
  always begin #36 weekend = day > 5; end // Need a delay here.     //4
endmodule                                                          //5

module work(day, weekday);                                         //1
  input [3:0] day; output weekday; reg weekday;                    //2
  always begin #36 weekday = day < 6; end // Need a delay here.    //3
endmodule                                                          //4
```

## 11.5 Procedures and Assignments

*Key terms and concepts:* a **procedure** is an `always` or `initial` statement, a `task`, or a `function`) • statements within a **sequential block** (between a `begin` and an `end`) that is part of a procedure execute sequentially, but the procedure executes concurrently with other procedures • **continuous assignments** appear outside procedures • **procedural assignments** appear inside procedures

```
module holiday_1(sat, sun, weekend);                               //1
  input sat, sun; output weekend;                                  //2
  assign weekend = sat | sun; // Assignment outside a procedure.   //3
endmodule                                                          //4

module holiday_2(sat, sun, weekend);                               //1
  input sat, sun; output weekend; reg weekend;                     //2
```

```
  always #1 weekend = sat | sun; // Assignment inside a procedure. //3
endmodule                                                          //4

module assignments                                                 //1
//... Continuous assignments go here.                              //2
always // beginning of a procedure                                 //3
  begin // beginning of sequential block                           //4
  //... Procedural assignments go here.                            //5
  end                                                              //6
endmodule                                                          //7
```

### 11.5.1 Continuous Assignment Statement

*Key terms and concepts:* a **continuous assignment statement** assigns to a `wire` like a real logic gate drives a real wire,

```
module assignment_1();                                             //1
wire pwr_good, pwr_on, pwr_stable;reg Ok, Fire;                    //2
assign pwr_stable = Ok & (!Fire);                                  //3
assign pwr_on = 1;                                                 //4
assign pwr_good = pwr_on & pwr_stable;                             //5
initial begin Ok = 0; Fire = 0; #1 Ok = 1; #5 Fire = 1;end         //6
initial begin $monitor("TIME=%0d",$time," ON=",pwr_on, " STABLE=", //7
   pwr_stable," OK=",Ok," FIRE=",Fire," GOOD=",pwr_good);          //8
  #10 $finish; end                                                 //9
endmodule                                                         //10

module assignment_2; reg Enable; wire [31:0] Data;                 //1
/* The following single statement is equivalent to a declaration and
continuous assignment. */                                          //2
wire [31:0] DataBus = Enable ? Data : 32'bz;                       //3
assign Data = 32'b10101101101011101111000010100001;               //4
  initial begin                                                    //5
    $monitor("Enable=%b DataBus=%b ", Enable, DataBus);            //6
    Enable = 0; #1; Enable = 1; #1;end                             //7
endmodule                                                          //8
```

### 11.5.2 Sequential Block

*Key terms and concepts:* a **sequential block** is a group of statements between a **begin** and an **end** • to declare new variables within a sequential block we must name the block • a sequential block is a statement, so that we may nest sequential blocks • a sequential block in an **always**

**statement** executes repeatedly • an **initial statement** executes only once, so a sequential block in an `initial` statement only executes once at the beginning of a simulation

```
module always_1; reg Y, Clk;                                    //1
always // Statements in an always statement execute repeatedly: //2
begin: my_block // Start of sequential block.                   //3
  @(posedge Clk) #5 Y = 1; // At +ve edge set Y=1,              //4
  @(posedge Clk) #5 Y = 0; // at the NEXT +ve edge set Y=0.     //5
end // End of sequential block.                                 //6
always #10 Clk = ~ Clk; // We need a clock.                     //7
initial Y = 0; // These initial statements execute             //8
initial Clk = 0; // only once, but first.                       //9
initial $monitor("T=%2g",$time," Clk=",Clk,"  Y=",Y);          //10
initial #70 $finish;                                           //11
endmodule                                                      //12
```

### 11.5.3 Procedural Assignments

*Key terms and concepts:* the value of an expression on the RHS of an assignment within a procedure (a **procedural assignment**) updates a `reg` (or memory element) immediately • a `reg` holds its value until changed by another procedural assignment • a **blocking assignment** is one type of procedural assignment

```
blocking_assignment ::= reg-lvalue = [delay_or_event_control]
expression
```

```
module procedural_assign; reg Y, A;                             //1
always @(A)                                                     //2
  Y = A; // Procedural assignment.                              //3
initial begin A=0; #5; A=1; #5; A=0; #5; $finish;end            //4
initial $monitor("T=%2g",$time,,"A=",A,,,"Y=",Y);               //5
endmodule                                                       //6
```

```
T= 0 A=0  Y=0
T= 5 A=1  Y=1
T=10 A=0  Y=0
```

## 11.6 Timing Controls and Delay

*Key terms and concepts:* statements in a sequential block are executed, in the absence of any delay, at the same simulation time—the current **time step** • delays are modeled using a **timing control**

### 11.6.1 Timing Control

*Key terms and concepts:* a **timing control** is a delay control or an event control • a **delay control** delays an assignment by a specified amount of time • **timescale compiler directive** is used to specify the units of time and precision • `` `timescale 1ns/10ps `` • (`s`, `ns`, `ps`, or `fs` and the multiplier must be 1, 10, or 100) • intra-assignment delay • delayed assignment • an **event control** delays an assignment until a specified event occurs • **posedge** is a transition from `'0'` to `'1'` or `'x'`, or a transition from `'x'` to `'1'` (transitions to or from `'z'` don't count) • events can be declared (as **named events**), triggered, and detected

```
x = #1 y; // intra-assignment delay
#1 x = y; // delayed assignment
```

```
begin // Equivalent to intra-assignment delay.
  hold = y; // Sample and hold y immediately.
  #1; // Delay.
  x = hold; // Assignment to x. Overall same as x = #1 y.
end
```

```
begin // Equivalent to delayed assignment.
  #1; // Delay.
  x = y; // Assign y to x. Overall same as #1 x = y.
end
```

```
event_control ::= @ event_identifier | @ (event_expression)
```

```
event_expression ::= expression |event_identifier
  | posedge expression | negedge expression
  | event_expression or event_expression
```

```
module delay_controls; reg X, Y, Clk, Dummy;                          //1
always #1 Dummy=!Dummy; // Dummy clock, just for graphics.            //2
// Examples of delay controls:                                        //3
always begin #25 X=1;#10 X=0;#5; end                                  //4
// An event control:                                                  //5
always @(posedge Clk) Y=X; // Wait for +ve clock edge.                //6
always #10 Clk = !Clk; // The real clock.                             //7
initial begin Clk = 0;                                                //8
  $display("T   Clk X Y");                                            //9
  $monitor("%2g",$time,,,Clk,,,,X,,Y);                               //10
  $dumpvars;#100 $finish; end                                        //11
endmodule                                                            //12
```

```
module show_event;                                                    //1
reg clock;                                                            //2
event event_1, event_2; // Declare two named events.                  //3
always @(posedge clock) -> event_1; // Trigger event_1.               //4
always @ event_1                                                      //5
begin $display("Strike 1!!"); -> event_2;end // Trigger event_2.      //6
always @ event_2 begin $display("Strike 2!!");                        //7
$finish; end // Stop on detection of event_2.                         //8
always #10 clock = ~ clock; // We need a clock.                       //9
initial clock = 0;                                                   //10
endmodule                                                            //11
```

### 11.6.2 Data Slip

```
module data_slip_1 (); reg Clk, D, Q1, Q2;                            //1
/************* bad sequential logic below ***************/           //2
always @(posedge Clk)  Q1 = D;                                        //3
always @(posedge Clk)  Q2 = Q1; // Data slips here!                  //4
/************* bad sequential logic above ***************/           //5
initial begin Clk = 0; D = 1; end always #50 Clk = ~Clk;              //6
initial begin $display("t   Clk D Q1 Q2");                            //7
$monitor("%3g",$time,,Clk,,,,D,,Q1,,,Q2);end                         //8
initial #400 $finish; // Run for 8 cycles.                            //9
```

```
initial $dumpvars;                                                    //10
endmodule                                                             //11
```

```
always @(posedge Clk)  Q1 = #1 D; // The delays in the assignments  //1
always @(posedge Clk)  Q2 = #1 Q1; // fix the data slip.            //2
```

### 11.6.3 Wait Statement

*Key terms and concepts:* **wait statement** suspends a procedure until a condition is true • beware "infinite hold" • level-sensitive

```
wait (Done) $stop; // Wait until Done = 1 then stop.
```

```
module test_dff_wait;                                                 //1
reg D, Clock, Reset; dff_wait u1(D, Q, Clock, Reset);                 //2
initial begin D=1; Clock=0;Reset=1'b1; #15 Reset=1'b0; #20 D=0;end    //3
always #10 Clock = !Clock;                                            //4
initial begin $display("T  Clk D Q Reset");                           //5
  $monitor("%2g",$time,,Clock,,,,D,,Q,,Reset); #50 $finish;end        //6
endmodule                                                             //7
```

```
module dff_wait(D, Q, Clock, Reset);                                  //1
output Q; input D, Clock, Reset; reg Q; wire D;                       //2
always @(posedge Clock) if (Reset !== 1) Q = D;                       //3
always begin wait (Reset == 1) Q = 0; wait (Reset !== 1); end         //4
endmodule                                                             //5
```

```
module dff_wait(D,Q,Clock,Reset);                                     //1
output Q; input D,Clock,Reset; reg Q; wire D;                         //2
always @(posedge Clock) if (Reset !== 1) Q = D;                       //3
// We need another wait statement here or we shall spin forever.      //4
always begin wait (Reset == 1) Q = 0; end                             //5
endmodule                                                             //6
```

### 11.6.4 Blocking and Nonblocking Assignments

*Key terms and concepts:* a procedural assignment (**blocking procedural assignment statement**) with a timing control delays or **blocks** execution • **nonblocking procedural assignment statement** allows execution to continue • registers are updated at end of current time step • synthesis tools don't allow blocking and nonblocking procedural assignments to the same `reg` within a sequential block

```
module delay;                                                         //1
reg a,b,c,d,e,f,g,bds,bsd;                                            //2
initial begin                                                         //3
```

```
a = 1; b = 0; // No delay control.                                    //4
#1 b = 1;     // Delayed assignment.                                   //5
c = #1 1;     // Intra-assignment delay.                               //6
#1;           // Delay control.                                        //7
d = 1;        //                                                       //8
e <= #1 1;    // Intra-assignment delay, nonblocking assignment        //9
#1 f <= 1;    // Delayed nonblocking assignment.                      //10
g <= 1;       // Nonblocking assignment.                              //11
end                                                                   //12
initial begin #1 bds = b; end // Delay then sample (ds).              //13
initial begin bsd = #1 b; end // Sample then delay (sd).              //14
initial begin $display("t a b c d e f g bds bsd");                    //15
$monitor("%g",$time,,a,,b,,c,,d,,e,,f,,g,,bds,,,,bsd); end           //16
endmodule                                                             //17
```

```
t a b c d e f g bds bsd
0 1 0 x x x x x x   x
1 1 1 x x x x x 1   0
2 1 1 1 x x x x 1   0
3 1 1 1 1 x x x 1   0
4 1 1 1 1 1 1 1 1   0
```

### 11.6.5 Procedural Continuous Assignment

*Key terms and concepts:* **procedural continuous assignment statement** (or quasicontinuous assignment statement) is a special form `assign` within a sequential block

**Verilog assignment statements.**

| Type of Verilog assignment | Continuous assignment statement | Procedural assignment statement | Nonblocking procedural assignment statement | Procedural continuous assignment statement |
|---|---|---|---|---|
| Where it can occur | outside an `always` or `initial` statement, task, or function | inside an `always` or `initial` statement, task, or function | inside an `always` or `initial` statement, task, or function | `always` or `initial` statement, task, or function |
| Example | `wire [31:0] DataBus; assign DataBus = Enable ? Data : 32'bz` | `reg Y; always @(posedge clock) Y = 1;` | `reg Y; always Y <= 1;` | `always @(Enable) if(Enable) assign Q = D; else deassign Q;` |
| Valid LHS of assignment | net | register or memory element | register or memory element | net |
| Valid RHS of assignment | <expression> net, reg or memory element | <expression> net, reg or memory element | <expression> net, reg or memory element | <expression> net, reg or memory element |
| Book | 11.5.1 | 11.5.3 | 11.6.4 | 11.6.5 |
| Verilog LRM | 6.1 | 9.2 | 9.2.2 | 9.3 |

```
module dff_procedural_assign;                                          //1
reg d,clr_,pre_,clk; wire q; dff_clr_pre dff_1(q,d,clr_,pre_,clk);     //2
always #10 clk = ~clk;                                                 //3
initial begin clk = 0; clr_ = 1; pre_ = 1; d = 1;                      //4
  #20; d = 0; #20; pre_ = 0; #20; pre_ = 1; #20; clr_ = 0;             //5
  #20; clr_ = 1; #20; d = 1; #20; $finish;end                          //6
initial begin                                                          //7
  $display("T  CLK PRE_ CLR_ D Q");                                    //8
  $monitor("%3g",$time,,,clk,,,,pre_,,,,clr_,,,,d,,q);end              //9
endmodule                                                             //10

module dff_clr_pre(q,d,clear_,preset_,clock);                          //1
output q; input d,clear_,preset_,clock; reg q;                         //2
always @(clear_ or preset_)                                            //3
```

```
  if (!clear_) assign q = 0; // active-low clear                    //4
  else if(!preset_) assign q = 1; // active-low preset              //5
  else deassign q;                                                  //6
always @(posedge clock) q = d;                                      //7
endmodule                                                           //8

module all_assignments                                              //1
//... continuous assignments.                                       //2
always // beginning of procedure                                   //3
  begin // beginning of sequential block                            //4
  //... blocking procedural assignments.                            //5
  //... nonblocking procedural assignments.                         //6
  //... procedural continuous assignments.                          //7
  end                                                               //8
endmodule                                                           //9
```

## 11.7 Tasks and Functions

*Key terms and concepts:* a **task** is a procedure called from another procedure • a task may call other tasks and functions • a **function** is a procedure used in an expression • a function may not call a task • tasks may contain timing controls but functions may not

```
Call_A_Task_And_Wait (Input1, Input2, Output);
Result_Immediate = Call_A_Function (All_Inputs);
```

```
module F_subset_decode; reg [2:0]A, B, C, D, E, F;                  //1
initial begin A = 1; B = 0; D = 2; E = 3;                           //2
  C = subset_decode(A, B); F = subset_decode(D,E);                  //3
  $display("A B C D E F"); $display(A,,B,,C,,D,,E,,F);end           //4
function [2:0] subset_decode; input [2:0] a, b;                     //5
  begin if (a <= b) subset_decode = a; else subset_decode = b; end  //6
endfunction                                                         //7
endmodule                                                           //8
```

## 11.8 Control Statements

*Key terms and concepts:* `if`, `case`, `loop`, `disable`, `fork`, and `join` statements control execution

### 11.8.1 Case and If Statement

*Key terms and concepts:* an **if statement** represents a two-way branch • a **case statement** represents a multiway branch • a **controlling expression** is matched with **case expressions** in each of the **case items** (or arms) to determine a match • the `case` statement must be inside a sequential block (inside an `always` statement) and needs some delay • a **casex statement** handles both `'z'` and `'x'` as don't care • the **casez statement** handles only `'z'` bits as don't care • bits in case expressions may be set to `'?'` representing don't care values

```
if(switch) Y = 1; else Y = 0;
```

```
module test_mux; reg a, b, select; wire out;                          //1
mux mux_1(a, b, out, select);                                         //2
initial begin #2; select = 0; a = 0; b = 1;                           //3
  #2; select = 1'bx; #2; select = 1'bz; #2; select = 1; end           //4
initial $monitor("T=%2g",$time," Select=",select,"  Out=",out);       //5
initial #10 $finish;                                                  //6
endmodule                                                             //7
```

```
module mux(a, b, mux_output, mux_select);input a, b, mux_select;      //1
output mux_output; reg mux_output;                                    //2
always begin                                                          //3
case(mux_select)                                                      //4
  0: mux_output = a;                                                  //5
  1: mux_output = b;                                                  //6
  default mux_output = 1'bx; // If select = x or z set output to
x.                                                                    //7
endcase                                                               //8
#1; // Need some delay, otherwise we'll spin forever.                 //9
end                                                                  //10
endmodule                                                            //11
```

```
casex (instruction_register[31:29])
  3b'??1 : add;
  3b'?1? : subtract;
  3b'1?? : branch;
endcase
```

### 11.8.2 Loop Statement

*Key terms and concepts:* A **loop statement** is a **for**, **while**, **repeat**, or **forever** statement •

```
module loop_1;                                                    //1
integer i; reg [31:0] DataBus; initial DataBus = 0;               //2
initial begin                                                     //3
/************** Insert loop code after here. ******************/
/* for(Execute this assignment once before starting loop; exit loop
if this expression is false; execute this assignment at end of loop
before the check for end of loop.) */
for(i = 0; i <= 15; i = i+1) DataBus[i] = 1;                      //4
/*************** Insert loop code before here. ****************/
end                                                               //5
initial begin                                                     //6
$display("DataBus = %b",DataBus);                                 //7
#2; $display("DataBus = %b",DataBus); $finish;                    //8
end                                                               //9
endmodule                                                        //10


i = 0;
/* while(Execute next statement while this expression is true.) */
while(i <= 15) begin DataBus[i] = 1; i = i+1; end


i = 0;
/* repeat(Execute next statement the number of times corresponding to
the evaluation of this expression at the beginning of the loop.) */
repeat(16) begin DataBus[i] = 1; i = i+1; end


i = 0;
/* A forever statement loops continuously. */
forever begin : my_loop
  DataBus[i] = 1;
  if (i == 15) #1 disable my_loop; // Need to let time advance to
exit.
  i = i+1;
end
```

### 11.8.3 Disable

*Key terms and concepts:* The **disable statement** stops the execution of a labeled sequential block and skips to the end of the block • difficult to implement in hardware

```
forever
begin: microprocessor_block // Labeled sequential block.
  @(posedge clock)
  if (reset) disable microprocessor_block; // Skip to end of block.
  else Execute_code;
end
```

### 11.8.4 Fork and Join

*Key terms and concepts:* The **fork statement** and **join statement** allows the execution of two or more parallel threads in a **parallel block** • difficult to implement in hardware

```
module fork_1                                          //1
event eat_breakfast, read_paper;                       //2
initial begin                                          //3
  fork                                                 //4
  @eat_breakfast; @read_paper;                         //5
  join                                                 //6
end                                                    //7
endmodule                                              //8
```

## 11.9 Logic-Gate Modeling

*Key terms and concepts:* Verilog has a set of built-in logic models and you may also define your own models.

### 11.9.1 Built-in Logic Models

*Key terms and concepts:* **primitives**: `and, nand, nor, or, xor, xnor`• strong drive strength is the default • the first port of a primitive gate is always the output port • remaining ports are the input ports

**Definition of the Verilog primitive 'and' gate**

| 'and' | 0 | 1 | x | z |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | x | x |
| x | 0 | x | x | x |
| z | 0 | x | x | x |

```
module primitive;                                          //1
nand (strong0, strong1) #2.2                               //2
  Nand_1(n001, n004, n005),                                //3
  Nand_2(n003, n001, n005, n002);                          //4
nand (n006, n005, n002);                                   //5
endmodule                                                  //6
```

### 11.9.2 User-Defined Primitives

*Key terms and concepts:* a **user-defined primitive** (**UDP**) uses a truth-table specification • the first port of a UDP must be an `output` port (no vector or `inout` ports) • inputs in a **UDP truth table** are `'0'`, `'1'`, and `'x'` • any `'z'` input is treated as an `'x'` • default output is `'x'` • any next state goes between an input and an output in UDP table • shorthand notation for levels • `(ab)` represents a change from a to b • `(01)` represents a rising edge • shorthand notations for edges

```
primitive Adder(Sum, InA, InB);                            //1
output Sum; input Ina, InB;                                //2
table                                                      //3
// inputs : output                                         //4
00 : 0;                                                    //5
01 : 1;                                                    //6
10 : 1;                                                    //7
11 : 0;                                                    //8
```

```
endtable                                                          //9
endprimitive                                                     //10

primitive DLatch(Q, Clock, Data);                                 //1
output Q; reg Q; input Clock, Data;                               //2
table                                                             //3
//inputs : present state : output (next state)                    //4
1 0 : ? : 0; // ? represents 0,1, or x (input or present state).  //5
1 1 : b : 1; // b represents 0 or 1 (input or present state).     //6
1 1 : x : 1; // Could have combined this with previous line.      //7
0 ? : ? : -; // - represents no change in an output.             //8
endtable                                                          //9
endprimitive                                                     //10

primitive DFlipFlop(Q, Clock, Data);                              //1
output Q; reg Q; input Clock, Data;                               //2
table                                                            //3
//inputs : present state : output (next state)                    //4
r    0 : ? : 0 ; // rising edge, next state = output = 0          //5
r    1 : ? : 1 ; // rising edge, next state = output = 1          //6
(0x) 0 : 0 : 0 ; // rising edge, next state = output = 0          //7
(0x) 1 : 1 : 1 ; // rising edge, next state = output = 1          //8
(?0) ? : ? : - ; // falling edge, no change in output            //9
? (??) : ? : - ; // no clock edge, no change in output          //10
endtable                                                        //11
endprimitive                                                    //12
```

## 11.10 Modeling Delay

*Key terms and concepts:* built-in delays • ASIC cell library models include logic delays as a function of fanout and estimated wiring loads • after layout, we can back-annotate • delay calculator calculates the net delays in **Standard Delay Format**, **SDF** • sign-off quality ASIC cell libraries

### 11.10.1 Net and Gate Delay

*Key terms and concepts:* minimum, typical, and maximum delays • first triplet specifies the min/typ/max rising delay (`'0'` or `'x'` or `'z'` to `'1'`) and the second triplet specifies the falling

delay (to `'0'`) • for a high-impedance output, we specify a triplet for rising, falling, and the delay to transition to `'z'` (from `'0'` or `'1'`), the delay for a three-state driver to turn off or float

```
#(1.1:1.3:1.7) assign delay_a = a; // min:typ:max
wire #(1.1:1.3:1.7) a_delay; // min:typ:max
wire #(1.1:1.3:1.7) a_delay = a; // min:typ:max


nand #3.0 nd01(c, a, b);
nand #(2.6:3.0:3.4) nd02(d, a, b); // min:typ:max
nand #(2.8:3.2:3.4, 2.6:2.8:2.9) nd03(e, a, b);
// #(rising, falling) delay


wire #(0.5,0.6,0.7) a_z = a; // rise/fall/float delays
```

### 11.10.2 Pin-to-Pin Delay

*Key terms and concepts:* A **specify block** allows **pin-to-pin delays** across a module • `x => y` specifies a **parallel connection** (or parallel path) • x and y must have the same number of bits • `x *> y` specifies a **full connection** (or full path) • every bit in x is connected to y • x and y may be different sizes • **state-dependent path delay**

```
module DFF_Spec; reg D, clk;                                          //1
DFF_Part DFF1 (Q, clk, D, pre, clr);                                  //2
initial begin D = 0; clk = 0; #1; clk = 1; end                        //3
initial $monitor("T=%2g", $time," clk=", clk," Q=", Q);               //4
endmodule                                                             //5

module DFF_Part(Q, clk, D, pre, clr);                                 //1
  input clk, D, pre, clr; output Q;                                   //2
  DFlipFlop(Q, clk, D); // No preset or clear in this UDP.            //3
  specify                                                             //4
    specparam                                                         //5
    tPLH_clk_Q = 3, tPHL_clk_Q = 2.9,                                 //6
    tPLH_set_Q = 1.2, tPHL_set_Q = 1.1;                               //7
  (clk => Q) = (tPLH_clk_Q, tPHL_clk_Q);                              //8
  (pre, clr *> Q) = (tPLH_set_Q, tPHL_set_Q);                         //9
  endspecify                                                         //10
endmodule                                                            //11

`timescale 1 ns / 100 fs                                              //1
module M_Spec; reg A1, A2, B; M M1 (Z, A1, A2, B);                    //2
```

```
initial begin A1=0;A2=1;B=1;#5;B=0;#5;A1=1;A2=0;B=1;#5;B=0;end      //3
initial                                                             //4
  $monitor("T=%4g",$realtime," A1=",A1," A2=",A2," B=",B," Z=",Z); //5
endmodule                                                           //6

`timescale 100 ps / 10 fs                                           //1
module M(Z, A1, A2, B); input A1, A2, B; output Z;                  //2
or (Z1, A1, A2); nand (Z, Z1, B); // OAI21                         //3
/*A1 A2 B Z  Delay=10*100 ps unless indicated in the table below.  //4
  0  0  0 1                                                        //5
  0  0  1 1                                                        //6
  0  1  0 1  B:0->1 Z:1->0 delay=t2                                 //7
  0  1  1 0  B:1->0 Z:0->1 delay=t1                                 //8
  1  0  0 1  B:0->1 Z:1->0 delay=t4                                 //9
  1  0  1 0  B:1->0 Z:0->1 delay=t3                                //10
  1  1  0 1                                                       //11
  1  1  1 0 */                                                     //12
specify specparam t1 = 11, t2 = 12; specparam t3 = 13, t4 = 14;   //13
  (A1 => Z) = 10; (A2 => Z) = 10;                                 //14
  if (~A1) (B => Z) = (t1, t2); if (A1) (B => Z) = (t3, t4);      //15
endspecify                                                        //16
endmodule                                                         //17
```

## 11.11 Altering Parameters

*Key terms and concepts:* parameter override in instantiated module • parameters have local scope • **defparam** statement and hierarchical name

```
module Vector_And(Z, A, B);                                       //1
  parameter CARDINALITY = 1;                                      //2
  input [CARDINALITY-1:0] A, B;                                   //3
  output [CARDINALITY-1:0] Z;                                     //4
  wire [CARDINALITY-1:0] Z = A & B;                               //5
endmodule                                                         //6

module Four_And_Gates(OutBus, InBusA, InBusB);                    //1
  input [3:0] InBusA, InBusB; output [3:0] OutBus;                //2
  Vector_And #(4) My_AND(OutBus, InBusA, InBusB); // 4 AND gates  //3
endmodule                                                         //4

module And_Gates(OutBus, InBusA, InBusB);                         //1
  parameter WIDTH = 1;                                            //2
  input [WIDTH-1:0] InBusA, InBusB; output [WIDTH-1:0] OutBus;    //3
```

```
  Vector_And #(WIDTH) My_And(OutBus, InBusA, InBusB);              //4
endmodule                                                          //5

module Super_Size; defparam And_Gates.WIDTH = 4; endmodule         //1
```

## 11.12 A Viterbi Decoder

### 11.12.1 Viterbi Encoder

```
/*****************************************************/
/* module viterbi_encode                             */
/*****************************************************/
/* This is the encoder. X2N (msb) and X1N form the 2-bit input
message, XN. Example: if X2N=1, X1N=0, then XN=2. Y2N (msb), Y1N, and
Y0N form the 3-bit encoded signal, YN (for a total constellation of 8
PSK signals that will be transmitted). The encoder uses a state
machine with four states to generate the 3-bit output, YN, from the
2-bit input, XN. Example: the repeated input sequence XN = (X2N, X1N)
= 0, 1, 2, 3 produces the repeated output sequence YN = (Y2N, Y1N,
Y0N) = 1, 0, 5, 4. */
module viterbi_encode(X2N,X1N,Y2N,Y1N,Y0N,clk,res);
input X2N,X1N,clk,res; output Y2N,Y1N,Y0N;
wire X1N_1,X1N_2,Y2N,Y1N,Y0N;
dff dff_1(X1N,X1N_1,clk,res); dff dff_2(X1N_1,X1N_2,clk,res);
assign Y2N=X2N; assign Y1N=X1N ^ X1N_2; assign Y0N=X1N_1;
endmodule
```

### 11.12.2 The Received Signal

```
/*****************************************************/
/* module viterbi_distances                          */
/*****************************************************/
/* This module simulates the front end of a receiver. Normally the
received analog signal (with noise) is converted into a series of
distance measures from the known eight possible transmitted PSK
signals: s0,...,s7. We are not simulating the analog part or noise in
this version, so we just take the digitally encoded 3-bit signal, Y,
from the encoder and convert it directly to the distance measures.
d[N] is the distance from signal = N to signal = 0
d[N] = (2*sin(N*PI/8))**2 in 3-bit binary (on the scale 2=100)
Example: d[3] = 1.85**2 = 3.41 = 110
inN is the distance from signal = N to encoder signal.
```

```
Example: in3 is the distance from signal = 3 to encoder signal.
d[N] is the distance from signal = N to encoder signal = 0.
If encoder signal = J, shift the distances by 8-J positions.
Example: if signal = 2, in0 is d[6], in1 is D[7], in2 is D[0], etc.
*/
module viterbi_distances
  (Y2N,Y1N,Y0N,clk,res,in0,in1,in2,in3,in4,in5,in6,in7);
input clk,res,Y2N,Y1N,Y0N; output in0,in1,in2,in3,in4,in5,in6,in7;
reg [2:0] J,in0,in1,in2,in3,in4,in5,in6,in7;reg [2:0] d [7:0];
initial begin d[0]='b000;d[1]='b001;d[2]='b100;d[3]='b110;
d[4]='b111;d[5]='b110;d[6]='b100;d[7]='b001;end
always @(Y2N or Y1N or Y0N) begin
J[0]=Y0N;J[1]=Y1N;J[2]=Y2N;
J=8-J;in0=d[J];J=J+1;in1=d[J];J=J+1;in2=d[J];J=J+1;in3=d[J];
J=J+1;in4=d[J];J=J+1;in5=d[J];J=J+1;in6=d[J];J=J+1;in7=d[J];
end endmodule
```

### 11.12.3 Testing the System

```
/*****************************************************/
/* module viterbi_test_CDD                           */
/*****************************************************/
/* This is the top-level module, viterbi_test_CDD, that models the
communications link. It contains three modules: viterbi_encode,
viterbi_distances, and viterbi. There is no analog and no noise in
this version. The 2-bit message, X, is encoded to a 3-bit signal, Y.
In this module the message X is generated using a simple counter.
The digital 3-bit signal Y is transmitted, received with noise as an
analog signal (not modeled here), and converted to a set of eight
3-bit distance measures, in0, ..., in7. The distance measures form
the input to the Viterbi decoder that reconstructs the transmitted
signal Y, with an error signal if the measures are inconsistent.
CDD = counter input, digital transmission, digital reception */
module viterbi_test_CDD;
wire Error; // decoder out
wire [2:0] Y, Out; // encoder out, decoder out
reg [1:0] X; // encoder inputs
reg Clk, Res; // clock and reset
wire [2:0] in0,in1,in2,in3,in4,in5,in6,in7;
always #500 $display("t   Clk X Y Out Error");
initial $monitor("%4g",$time,,Clk,,,,X,,Y,,Out,,,,Error);
initial $dumpvars; initial #3000 $finish;
always #50 Clk = ~Clk; initial begin Clk = 0;
```

```
X = 3; // No special reason to start at 3.
#60 Res = 1;#10 Res = 0;end // Hit reset after inputs are stable.
always @(posedge Clk) #1 X = X + 1; // Drive the input with a
counter.
viterbi_encode v_1
  (X[1],X[0],Y[2],Y[1],Y[0],Clk,Res);
viterbi_distances v_2
  (Y[2],Y[1],Y[0],Clk,Res,in0,in1,in2,in3,in4,in5,in6,in7);
viterbi v_3
  (in0,in1,in2,in3,in4,in5,in6,in7,Out,Clk,Res,Error);
endmodule
```

### 11.12.4 Verilog Decoder Model

```
/*****************************************************/
/*       module dff                                  */
/*****************************************************/
/* A D flip-flop module. */


module dff(D,Q,Clock,Reset); // N.B. reset is active-low.
output Q; input D,Clock,Reset;
parameter CARDINALITY = 1; reg [CARDINALITY-1:0] Q;
wire [CARDINALITY-1:0] D;
always @(posedge Clock) if (Reset !== 0) #1 Q = D;
always begin wait (Reset == 0); Q = 0; wait (Reset == 1); end
endmodule


/* Verilog code for a Viterbi decoder. The decoder assumes a rate
2/3 encoder, 8 PSK modulation, and trellis coding. The viterbi module
contains eight submodules: subset_decode, metric, compute_metric,
compare_select, reduce, pathin, path_memory, and output_decision.
  The decoder accepts eight 3-bit measures of ||r-si||**2 and, after
an initial delay of thirteen clock cycles, the output is the best
estimate of the signal transmitted. The distance measures are the
Euclidean distances between the received signal r (with noise) and
each of the (in this case eight) possible transmitted signals s0 to
s7.
  Original by Christeen Gray, University of Hawaii. Heavily modified
by MJSS; any errors are mine. Use freely. */
/*****************************************************/
/*   module viterbi                                  */
```

```
/*****************************************************/
/* This is the top level of the Viterbi decoder. The eight input
signals {in0,...,in7} represent the distance measures, ||r-si||**2.
The other input signals are clk and reset. The output signals are
out and error. */


module viterbi
    (in0,in1,in2,in3,in4,in5,in6,in7,
    out,clk,reset,error);
input [2:0] in0,in1,in2,in3,in4,in5,in6,in7;
output [2:0] out; input clk,reset; output error;
wire sout0,sout1,sout2,sout3;
wire [2:0] s0,s1,s2,s3;
wire [4:0] m_in0,m_in1,m_in2,m_in3;
wire [4:0] m_out0,m_out1,m_out2,m_out3;
wire [4:0] p0_0,p2_0,p0_1,p2_1,p1_2,p3_2,p1_3,p3_3;
wire ACS0,ACS1,ACS2,ACS3;
wire [4:0] out0,out1,out2,out3;
wire [1:0] control;
wire [2:0] p0,p1,p2,p3;
wire [11:0] path0;


  subset_decode u1(in0,in1,in2,in3,in4,in5,in6,in7,
    s0,s1,s2,s3,sout0,sout1,sout2,sout3,clk,reset);
  metric u2(m_in0,m_in1,m_in2,m_in3,m_out0,
    m_out1,m_out2,m_out3,clk,reset);
  compute_metric u3(m_out0,m_out1,m_out2,m_out3,s0,s1,s2,s3,
    p0_0,p2_0,p0_1,p2_1,p1_2,p3_2,p1_3,p3_3,error);
  compare_select u4(p0_0,p2_0,p0_1,p2_1,p1_2,p3_2,p1_3,p3_3,
    out0,out1,out2,out3,ACS0,ACS1,ACS2,ACS3);
  reduce u5(out0,out1,out2,out3,
    m_in0,m_in1,m_in2,m_in3,control);
  pathin u6(sout0,sout1,sout2,sout3,
    ACS0,ACS1,ACS2,ACS3,path0,clk,reset);
  path_memory u7(p0,p1,p2,p3,path0,clk,reset,
    ACS0,ACS1,ACS2,ACS3);
  output_decision u8(p0,p1,p2,p3,control,out);
endmodule


/*****************************************************/
/* module subset_decode                              */
/*****************************************************/
```

```
/* This module chooses the signal corresponding to the smallest of
each set {||r-s0||**2,||r-s4||**2}, {||r-s1||**2, ||r-s5||**2},
{||r-s2||**2,||r-s6||**2}, {||r-s3||**2,||r-s7||**2}. Therefore
there are eight input signals and four output signals for the
distance measures. The signals sout0, ..., sout3 are used to control
the path memory. The statement dff #(3) instantiates a vector array
of 3 D flip-flops. */
module subset_decode
    (in0,in1,in2,in3,in4,in5,in6,in7,
    s0,s1,s2,s3,
    sout0,sout1,sout2,sout3,
    clk,reset);
input [2:0] in0,in1,in2,in3,in4,in5,in6,in7;
output [2:0] s0,s1,s2,s3;
output sout0,sout1,sout2,sout3;
input clk,reset;
wire [2:0] sub0,sub1,sub2,sub3,sub4,sub5,sub6,sub7;


  dff #(3) subout0(in0, sub0, clk, reset);
  dff #(3) subout1(in1, sub1, clk, reset);
  dff #(3) subout2(in2, sub2, clk, reset);
  dff #(3) subout3(in3, sub3, clk, reset);
  dff #(3) subout4(in4, sub4, clk, reset);
  dff #(3) subout5(in5, sub5, clk, reset);
  dff #(3) subout6(in6, sub6, clk, reset);
  dff #(3) subout7(in7, sub7, clk, reset);


  function [2:0] subset_decode; input [2:0] a,b;
    begin
      subset_decode = 0;
      if (a<=b) subset_decode = a; else subset_decode = b;
    end
  endfunction


  function set_control; input [2:0] a,b;
    begin
      if (a<=b) set_control = 0; else set_control = 1;
    end
  endfunction


assign s0 = subset_decode (sub0,sub4);
```

```
assign s1 = subset_decode (sub1,sub5);
assign s2 = subset_decode (sub2,sub6);
assign s3 = subset_decode (sub3,sub7);
assign sout0 = set_control(sub0,sub4);
assign sout1 = set_control(sub1,sub5);
assign sout2 = set_control(sub2,sub6);
assign sout3 = set_control(sub3,sub7);
endmodule


/*****************************************************/
/*   module compute_metric                           */
/*****************************************************/
/* This module computes the sum of path memory and the distance for
each path entering a state of the trellis. For the four states,
there are two paths entering it; therefore eight sums are computed
in this module. The path metrics and output sums are 5 bits wide.
The output sum is bounded and should never be greater than 5 bits
for a valid input signal. The overflow from the sum is the error
output and indicates an invalid input signal.*/
module compute_metric
    (m_out0,m_out1,m_out2,m_out3,
    s0,s1,s2,s3,p0_0,p2_0,
    p0_1,p2_1,p1_2,p3_2,p1_3,p3_3,
    error);
  input [4:0] m_out0,m_out1,m_out2,m_out3;
  input [2:0] s0,s1,s2,s3;
  output [4:0] p0_0,p2_0,p0_1,p2_1,p1_2,p3_2,p1_3,p3_3;
  output error;


  assign
    p0_0 = m_out0 + s0,
    p2_0 = m_out2 + s2,
    p0_1 = m_out0 + s2,
    p2_1 = m_out2 + s0,
    p1_2 = m_out1 + s1,
    p3_2 = m_out3 + s3,
    p1_3 = m_out1 + s3,
    p3_3 = m_out3 + s1;


  function is_error; input x1,x2,x3,x4,x5,x6,x7,x8;
  begin
    if (x1||x2||x3||x4||x5||x6||x7||x8) is_error = 1;
```

```
    else is_error = 0;
  end
  endfunction


  assign error = is_error(p0_0[4],p2_0[4],p0_1[4],p2_1[4],
    p1_2[4],p3_2[4],p1_3[4],p3_3[4]);
endmodule


/*****************************************************/
/*   module compare_select                           */
/*****************************************************/
/* This module compares the summations from the compute_metric
module and selects the metric and path with the lowest value. The
output of this module is saved as the new path metric for each
state. The ACS output signals are used to control the path memory of
the decoder. */
module compare_select
    (p0_0,p2_0,p0_1,p2_1,p1_2,p3_2,p1_3,p3_3,
    out0,out1,out2,out3,
    ACS0,ACS1,ACS2,ACS3);
  input [4:0] p0_0,p2_0,p0_1,p2_1,p1_2,p3_2,p1_3,p3_3;
  output [4:0] out0,out1,out2,out3;
  output ACS0,ACS1,ACS2,ACS3;


  function [4:0] find_min_metric; input [4:0] a,b;
    begin
      if (a <= b) find_min_metric = a; else find_min_metric = b;
    end
  endfunction


  function set_control; input [4:0] a,b;
    begin
      if (a <= b) set_control = 0; else set_control = 1;
    end
  endfunction


assign out0 = find_min_metric(p0_0,p2_0);
assign out1 = find_min_metric(p0_1,p2_1);
assign out2 = find_min_metric(p1_2,p3_2);
assign out3 = find_min_metric(p1_3,p3_3);
```

```
assign ACS0 = set_control (p0_0,p2_0);
assign ACS1 = set_control (p0_1,p2_1);
assign ACS2 = set_control (p1_2,p3_2);
assign ACS3 = set_control (p1_3,p3_3);
endmodule


/*****************************************************/
/*   module path                                     */
/*****************************************************/
/* This is the basic unit for the path memory of the Viterbi
decoder. It consists of four 3-bit D flip-flops in parallel. There
is a 2:1 mux at each D flip-flop input. The statement dff #(12)
instantiates a vector array of 12 flip-flops. */
module path(in,out,clk,reset,ACS0,ACS1,ACS2,ACS3);
input [11:0] in; output [11:0] out;
input clk,reset,ACS0,ACS1,ACS2,ACS3;wire [11:0] p_in;

dff #(12) path0(p_in,out,clk,reset);

  function [2:0] shift_path; input [2:0] a,b; input control;
    begin
      if (control == 0) shift_path = a;else shift_path = b;
    end
  endfunction


assign p_in[11:9] = shift_path(in[11:9],in[5:3],ACS0);
assign p_in[ 8:6] = shift_path(in[11:9],in[5:3],ACS1);
assign p_in[ 5:3] = shift_path(in[8: 6],in[2:0],ACS2);
assign p_in[ 2:0] = shift_path(in[8: 6],in[2:0],ACS3);
endmodule


/*****************************************************/
/*   module path_memory                              */
/*****************************************************/
/* This module consists of an array of memory elements (D
flip-flops) that store and shift the path memory as new signals are
added to the four paths (or four most likely sequences of signals).
This module instantiates 11 instances of the path module. */
module path_memory
    (p0,p1,p2,p3,
    path0,clk,reset,
    ACS0,ACS1,ACS2,ACS3);
output [2:0] p0,p1,p2,p3; input [11:0] path0;
```

```
input clk,reset,ACS0,ACS1,ACS2,ACS3;
wire [11:0]out1,out2,out3,out4,out5,out6,out7,out8,out9,out10,out11;
    path x1 (path0,out1 ,clk,reset,ACS0,ACS1,ACS2,ACS3),
         x2 (out1, out2 ,clk,reset,ACS0,ACS1,ACS2,ACS3),
         x3 (out2, out3 ,clk,reset,ACS0,ACS1,ACS2,ACS3),
         x4 (out3, out4 ,clk,reset,ACS0,ACS1,ACS2,ACS3),
         x5 (out4, out5 ,clk,reset,ACS0,ACS1,ACS2,ACS3),
         x6 (out5, out6 ,clk,reset,ACS0,ACS1,ACS2,ACS3),
         x7 (out6, out7 ,clk,reset,ACS0,ACS1,ACS2,ACS3),
         x8 (out7, out8 ,clk,reset,ACS0,ACS1,ACS2,ACS3),
         x9 (out8, out9 ,clk,reset,ACS0,ACS1,ACS2,ACS3),
         x10(out9, out10,clk,reset,ACS0,ACS1,ACS2,ACS3),
         x11(out10,out11,clk,reset,ACS0,ACS1,ACS2,ACS3);
assign p0 = out11[11:9];
assign p1 = out11[ 8:6];
assign p2 = out11[ 5:3];
assign p3 = out11[ 2:0];
endmodule


/*****************************************************/
/*   module pathin                                   */
/*****************************************************/
/* This module determines the input signal to the path for each of
the four paths. Control signals from the subset decoder and compare
select modules are used to store the correct signal. The statement
dff #(12) instantiates a vector array of 12 flip-flops. */
module pathin
    (sout0,sout1,sout2,sout3,
    ACS0,ACS1,ACS2,ACS3,
    path0,clk,reset);
  input sout0,sout1,sout2,sout3,ACS0,ACS1,ACS2,ACS3;
  input clk,reset; output [11:0] path0;
  wire [2:0] sig0,sig1,sig2,sig3;wire [11:0] path_in;


  dff #(12) firstpath(path_in,path0,clk,reset);


  function [2:0] subset0; input sout0;
    begin
      if(sout0 == 0) subset0 = 0;else subset0 = 4;
    end
  endfunction
```

```
  function [2:0] subset1; input sout1;
    begin
      if(sout1 == 0) subset1 = 1; else subset1 = 5;
    end
  endfunction


  function [2:0] subset2; input sout2;
    begin
      if(sout2 == 0) subset2 = 2; else subset2 = 6;
    end
  endfunction


  function [2:0] subset3; input sout3;
    begin
      if(sout3 == 0) subset3 = 3; else subset3 = 7;
    end
  endfunction


  function [2:0] find_path; input [2:0] a,b; input control;
    begin
      if(control==0) find_path = a; else find_path = b;
    end
  endfunction


assign sig0 = subset0(sout0);
assign sig1 = subset1(sout1);
assign sig2 = subset2(sout2);
assign sig3 = subset3(sout3);
assign path_in[11:9] = find_path(sig0,sig2,ACS0);
assign path_in[ 8:6] = find_path(sig2,sig0,ACS1);
assign path_in[ 5:3] = find_path(sig1,sig3,ACS2);
assign path_in[ 2:0] = find_path(sig3,sig1,ACS3);
endmodule


/*****************************************************/
/*   module metric                                   */
/*****************************************************/
/* The registers created in this module (using D flip-flops) store
the four path metrics. Each register is 5 bits wide. The statement
dff #(5) instantiates a vector array of 5 flip-flops. */
```

```
module metric
    (m_in0,m_in1,m_in2,m_in3,
    m_out0,m_out1,m_out2,m_out3,
    clk,reset);
input [4:0] m_in0,m_in1,m_in2,m_in3;
output [4:0] m_out0,m_out1,m_out2,m_out3;
input clk,reset;
  dff #(5) metric3(m_in3, m_out3, clk, reset);
  dff #(5) metric2(m_in2, m_out2, clk, reset);
  dff #(5) metric1(m_in1, m_out1, clk, reset);
  dff #(5) metric0(m_in0, m_out0, clk, reset);
endmodule


/*****************************************************/
/*   module output_decision                          */
/*****************************************************/
/* This module decides the output signal based on the path that
corresponds to the smallest metric. The control signal comes from
the reduce module. */


module output_decision(p0,p1,p2,p3,control,out);
  input [2:0] p0,p1,p2,p3; input [1:0] control; output [2:0] out;
  function [2:0] decide;
  input [2:0] p0,p1,p2,p3; input [1:0] control;
  begin
    if(control == 0) decide = p0;
    else if(control == 1) decide = p1;
    else if(control == 2) decide = p2;
    else decide = p3;
    end
  endfunction


assign out = decide(p0,p1,p2,p3,control);
endmodule


/*****************************************************/
/*   module reduce                                   */
/*****************************************************/
/* This module reduces the metrics after the addition and compare
operations. This algorithm selects the smallest metric and subtracts
it from all the other metrics. */
```

```
module reduce
    (in0,in1,in2,in3,
    m_in0,m_in1,m_in2,m_in3,
    control);
  input [4:0] in0,in1,in2,in3;
  output [4:0] m_in0,m_in1,m_in2,m_in3;
  output [1:0] control; wire [4:0] smallest;


  function [4:0] find_smallest;
    input [4:0] in0,in1,in2,in3; reg [4:0] a,b;
      begin
        if(in0 <= in1) a = in0; else a = in1;
        if(in2 <= in3) b = in2; else b = in3;
        if(a <= b) find_smallest = a;
        else find_smallest = b;
      end
  endfunction


  function [1:0] smallest_no;
  input [4:0] in0,in1,in2,in3,smallest;
    begin
      if(smallest == in0) smallest_no = 0;
      else if (smallest == in1) smallest_no = 1;
      else if (smallest == in2) smallest_no = 2;
      else smallest_no = 3;
    end
  endfunction


assign smallest = find_smallest(in0,in1,in2,in3);
assign m_in0 = in0 - smallest;
assign m_in1 = in1 - smallest;
assign m_in2 = in2 - smallest;
assign m_in3 = in3 - smallest;
assign control = smallest_no(in0,in1,in2,in3,smallest);
endmodule
```

## 11.13 Other Verilog Features

*Key terms and concepts:* **system tasks** and functions are part of the IEEE standard

### 11.13.1 Display Tasks

*Key terms and concepts:* **display system tasks** • `$display` (format works like C) • `$write` • `$strobe`

```
module test_display; // display system tasks:
initial begin $display ("string, variables, or expression");
/* format specifications work like printf in C:
    %d=decimal %b=binary %s=string %h=hex %o=octal
    %c=character %m=hierarchical name %v=strength %t=time format
    %e=scientific %f=decimal %g=shortest
examples: %d uses default width %0d uses minimum width
    %7.3g uses 7 spaces with 3 digits after decimal point */
// $displayb, $displayh, $displayo print in b, h, o formats
// $write, $strobe, $monitor also have b, h, o versions


$write("write"); // as $display, but without newline at end of line


$strobe("strobe"); // as $display, values at end of simulation cycle


$monitor(v); // disp. @change of v (except v= $time,$stime,$realtime)
$monitoron; $monitoroff; // toggle monitor mode on/off


end endmodule
```

### 11.13.2 File I/O Tasks

*Key terms and concepts:* **file I/O system tasks** • `$fdisplay` • `$fopen` • `$fclose` • **multichannel descriptor** • 32 flags • channel 0 is the standard output (screen) and is always open • `$readmemb` and `$readmemh` read a text file into a memory • file may contain only spaces, new lines, tabs, form feeds, comments, addresses, and binary (`$readmemb`) or hex (`$readmemh`)

```
module file_1; integer f1, ch; initial begin f1 = $fopen("f1.out");
if(f1==0) $stop(2); if(f1==2)$display("f1 open");
ch = f1|1; $fdisplay(ch,"Hello"); $fclose(f1);end endmodule
```

```
> vlog file_1.v
> vsim -c file_1
# Loading work.file_1
VSIM 1> run 10
# f1 open
# Hello
VSIM 2> q
> more f1.out
Hello
>
```

```
mem.dat
@2 1010_1111 @4 0101_1111 1010_1111 // @address in hex
x1x1_zzzz 1111_0000 /* x or z is OK */
```

```
module load; reg [7:0] mem[0:7]; integer i; initial begin
$readmemb("mem.dat", mem, 1, 6); // start_address=1, end_address=6
for (i= 0; i<8; i=i+1) $display("mem[%0d] %b", i, mem[i]);
end endmodule
```

```
> vsim -c load
# Loading work.load
VSIM 1> run 10
# ** Warning: $readmem (memory mem) file mem.dat line 2:
#    More patterns than index range (hex 1:6)
#    Time: 0 ns  Iteration: 0  Instance:/
# mem[0] xxxxxxxx
# mem[1] xxxxxxxx
# mem[2] 10101111
# mem[3] xxxxxxxx
# mem[4] 01011111
# mem[5] 10101111
# mem[6] x1x1zzzz
# mem[7] xxxxxxxx
VSIM 2> q
>
```

### 11.13.3 Timescale, Simulation, and Timing-Check Tasks

*Key terms and concepts:* **timescale tasks**: `$printtimescale` and `$timeformat` • **simulation control tasks**: `$stop` and `$finish` • **timing-check tasks** • **edge specifiers** •

'**edge** [01, 0x, x1] clock'is equivalent to '**posedge** clock' • edge transitions with 'z' are treated the same as transitions with 'x'• **notifier register** (changed when a timing-check task detects a violation)

**Timing-check system task parameters**

| Timing task argument | Description of argument | Type of argument |
|---|---|---|
| reference_event | to establish reference time | module input or inout (scalar or vector net) |
| data_event | signal to check against reference_event | module input or inout (scalar or vector net) |
| limit | time limit to detect timing violation on data_event | constant expression or specparam |
| threshold | largest pulse width ignored by timing check $width | constant expression or specparam |
| notifier | flags a timing violation (before -> after): x->0, 0->1, 1->0, z->z | register |

```
edge_control_specifier ::= edge [edge_descriptor {, edge_descriptor}]
edge_descriptor ::= 01 | 0x | 10 | 1x | x0 | x1
```

```
// timescale tasks:
module a; initial $printtimescale(b.c1); endmodule
module b; c c1 (); endmodule
`timescale 10 ns / 1 fs
module c_dat; endmodule
```

```
`timescale 1 ms / 1 ns
module Ttime; initial $timeformat(-9, 5, " ns", 10); endmodule
/* $timeformat [ ( n, p, suffix , min_field_width ) ] ;
units = 1 second ** (-n), n = 0->15, e.g. for n = 9, units = ns
p = digits after decimal point for %t e.g. p = 5 gives 0.00000
suffix for %t (despite timescale directive)
min_field_width is number of character positions for %t */
```

```
module test_simulation_control; // simulation control system tasks:
initial begin $stop; // enter interactive mode (default parameter 1)
$finish(2); // graceful exit with optional parameter as follows:
// 0 = nothing 1 = time and location 2 = time, location, and
statistics
end endmodule
```

```
module timing_checks (data, clock, clock_1,clock_2);                  //1
input data,clock,clock_1,clock_2;reg tSU,tH,tHIGH,tP,tSK,tR;          //2
specify // timing check system tasks:                                 //3
/* $setup (data_event, reference_event, limit [, notifier]);         //4
violation = (T_reference_event)-(T_data_event) < limit */             //5
$setup(data, posedge clock, tSU);                                     //6
/* $hold (reference_event, data_event, limit [, notifier]);          //7
violation =                                                           //8
  (time_of_data_event)-(time_of_reference_event) < limit */           //9
$hold(posedge clock, data, tH);                                      //10
/* $setuphold (reference_event, data_event, setup_limit,             //11
    hold_limit [, notifier]);                                        //12
parameter_restriction = setup_limit + hold_limit > 0 */              //13
$setuphold(posedge clock, data, tSU, tH);                            //14
/* $width (reference_event, limit, threshold [, notifier]);          //15
violation =                                                          //16
  threshold < (T_data_event) - (T_reference_event) < limit           //17
reference_event = edge                                               //18
data_event = opposite_edge_of_reference_event */                     //19
$width(posedge clock, tHIGH);                                        //20
/* $period (reference_event, limit [, notifier]);                    //21
violation = (T_data_event) - (T_reference_event) < limit             //22
reference_event = edge                                               //23
data_event = same_edge_of_reference event */                         //24
$period(posedge clock, tP);                                          //25
/* $skew (reference_event, data_event, limit [, notifier]);          //26
violation = (T_data_event) - (T_reference_event) > limit */          //27
$skew(posedge clock_1, posedge clock_2, tSK);                        //28
/* $recovery (reference_event, data_event, limit, [, notifier]);     //29
violation = (T_data_event) - (T_reference_event) < limit */          //30
$recovery(posedge clock, posedge clock_2, tR);                       //31
/* $nochange (reference_event, data_event, start_edge_offset,        //32
  end_edge_offset [, notifier]);                                     //33
reference_event = posedge | negedge                                  //34
```

```
violation = change while reference high (posedge)/low (negedge)   //35
+ve start_edge_offset moves start of window later                 //36
+ve end_edge_offset moves end of window later */                  //37
$nochange (posedge clock, data, 0, 0);                            //38
endspecify endmodule                                              //39


primitive dff_udp(q, clock, data, notifier);
output q; reg q; input clock, data, notifier;
table //   clock data  notifier:state:  q
           r     0     ?      :  ?  :   0 ;
           r     1     ?      :  ?  :   1 ;
           n     ?     ?      :  ?  :   - ;
           ?     *     ?      :  ?  :   - ;
           ?     ?     *      :  ?  :   x ; endtable // notifier
endprimitive


`timescale 100 fs / 1 fs
module dff(q, clock, data); output q; input clock, data; reg
notifier;
dff_udp(q1, clock, data, notifier); buf(q, q1);
specify
  specparam tSU = 5, tH = 1, tPW = 20, tPLH = 4:5:6, tPHL = 4:5:6;
    (clock *> q) = (tPLH, tPHL);
  $setup(data, posedge clock, tSU, notifier); // setup: data to clock
  $hold(posedge clock, data, tH, notifier); // hold: clock to data
  $period(posedge clock, tPW, notifier); // clock: period
endspecify
endmodule
```

### 11.13.4 PLA Tasks

*Key terms and concepts:* The **PLA modeling tasks** model two-level logic • `eqntott` logic equations • **array format** (`'1'` or `'0'` in **personality array**) • `espresso` input plane format • **plane format** allows `'1'`, `'0'`, `'?'` or `'z'` (either may be used for don't care) in personality array

```
b1 = a1 & a2; b2 = a3 & a4 & a5 ; b3 = a5 & a6 & a7;


array.dat
1100000
```

```
0011100
0000111
```

```
module pla_1 (a1,a2,a3,a4,a5,a6,a7,b1,b2,b3);
input a1, a2, a3, a4, a5, a6, a7 ; output b1, b2, b3;
reg [1:7] mem[1:3]; reg b1, b2, b3;
initial begin
  $readmemb("array.dat", mem);
  #1; b1=1; b2=1; b3=1;
  $async$and$array(mem,{a1,a2,a3,a4,a5,a6,a7},{b1,b2,b3});
end
initial $monitor("%4g",$time,,b1,,b2,,b3);
endmodule
```

```
b1 = a1 & !a2; b2 = a3; b3 = !a1 & !a3; b4 = 1;
```

```
module pla_2; reg [1:3] a, mem[1:4]; reg [1:4] b;
initial begin
  $async$and$plane(mem,{a[1],a[2],a[3]},{b[1],b[2],b[3],b[4]});
  mem[1] = 3'b10?; mem[2] = 3'b??1; mem[3] = 3'b0?0; mem[4] = 3'b???;
  #10 a = 3'b111; #10 $displayb(a, " -> ", b);
  #10 a = 3'b000; #10 $displayb(a, " -> ", b);
  #10 a = 3'bxxx; #10 $displayb(a, " -> ", b);
  #10 a = 3'b101; #10 $displayb(a, " -> ", b);
end endmodule
```

```
111 -> 0101
000 -> 0011
xxx -> xxx1
101 -> 1101
```

### 11.13.5 Stochastic Analysis Tasks

*Key terms and concepts:* The **stochastic analysis tasks** model queues

```
module stochastic; initial begin // stochastic analysis system tasks:
```

```
/* $q_initialize (q_id, q_type, max_length, status) ;
q_id is an integer that uniquely identifies the queue
```

**Status values for the stochastic analysis tasks.**

| Status value | Meaning |
|---|---|
| 0 | OK |
| 1 | queue full, cannot add |
| 2 | undefined `q_id` |
| 3 | queue empty, cannot remove |
| 4 | unsupported `q_type`, cannot create queue |
| 5 | `max_length` <= 0, cannot create queue |
| 6 | duplicate `q_id`, cannot create queue |
| 7 | not enough memory, cannot create queue |

```
q_type 1=FIFO 2=LIFO
max_length is an integer defining the maximum number of entries */


$q_initialize (q_id, q_type, max_length, status) ;


/* $q_add (q_id, job_id, inform_id, status) ;
job_id = integer input
inform_id = user-defined integer input for queue entry */
$q_add (q_id, job_id, inform_id, status) ;


/* $q_remove (q_id, job_id, inform_id, status) ; */
$q_remove (q_id, job_id, inform_id, status) ;


/* $q_full (q_id, status) ;
status = 0 = queue is not full, status = 1 = queue full */
$q_full (q_id, status) ;


/* $q_exam (q_id, q_stat_code, q_stat_value, status) ;
q_stat_code is input request as follows:
1=current queue length 2=mean inter-arrival time 3=max. queue length
4=shortest wait time ever
5=longest wait time for jobs still in queue 6=ave. wait time in queue
```

```
q_stat_value is output containing requested value */
$q_exam (q_id, q_stat_code, q_stat_value, status) ;


end endmodule
```

### 11.13.6 Simulation Time Functions

*Key terms and concepts:* The **simulation time functions** return the time

```
module test_time; initial begin // simulation time system functions:
$time ;
// returns 64-bit integer scaled to timescale unit of invoking module


$stime ;
// returns 32-bit integer scaled to timescale unit of invoking module


$realtime ;
// returns real scaled to timescale unit of invoking module


end endmodule
```

### 11.13.7 Conversion Functions

*Key terms and concepts:* The **conversion functions** for reals handle real numbers:

```
module test_convert; // conversion functions for reals:
integer i; real r; reg [63:0] bits;
initial begin #1 r=256;#1 i = $rtoi(r);
#1; r = $itor(2 * i) ; #1 bits = $realtobits(2.0 * r) ;
#1; r = $bitstoreal(bits) ;end
initial $monitor("%3f",$time,,i,,r,,bits); /*
$rtoi converts reals to integers w/truncation e.g. 123.45 -> 123
$itor converts integers to reals e.g. 123 -> 123.0
$realtobits converts reals to 64-bit vector
$bitstoreal converts bit pattern to real
Real numbers in these functions conform to IEEE Std 754. Conversion
rounds to the nearest valid number. */
endmodule
```

```
module test_real;wire [63:0]a; driver d (a); receiver r (a);
initial $monitor("%3g",$time,,a,,d.r1,,r.r2);endmodule


module driver (real_net);
output real_net; real r1; wire [64:1] real_net = $realtobits(r1);
initial #1 r1 = 123.456; endmodule


module receiver (real_net);
input real_net; wire [64:1] real_net; real r2;
initial assign r2 = $bitstoreal(real_net);
endmodule
```

### 11.13.8 Probability Distribution Functions

*Key terms and concepts:* probability distribution functions • `$random` • `uniform` • `normal` • `exponential` • `poisson` • `chi_square` • `t` • `erlang`

```
module probability; // probability distribution functions:          //1
/* $random [(seed)] returns random 32-bit signed integer           //2
seed = register, integer, or time */                               //3
reg [23:0] r1,r2; integer r3,r4,r5,r6,r7,r8,r9;                    //4
integer seed, start, \end , mean, standard_deviation;              //5
integer degree_of_freedom, k_stage;                                //6
initial begin seed=1; start=0; \end =6; mean=5;                    //7
standard_deviation=2; degree_of_freedom=2; k_stage=1; #1;          //8
r1 = $random % 60; // random -59 to 59                             //9
r2 = {$random} % 60; // positive value 0-59                       //10
r3=$dist_uniform (seed, start, \end ) ;                            //11
r4=$dist_normal (seed, mean, standard_deviation) ;                 //12
r5=$dist_exponential (seed, mean) ;                                //13
r6=$dist_poisson (seed, mean) ;                                    //14
r7=$dist_chi_square (seed, degree_of_freedom) ;                    //15
r8=$dist_t (seed, degree_of_freedom) ;                             //16
r9=$dist_erlang (seed, k_stage, mean) ;end                         //17
initial #2 $display ("%3f",$time,,r1,,r2,,r3,,r4,,r5);             //18
initial begin #3; $display ("%3f",$time,,r6,,r7,,r8,,r9);end       //19
/* All parameters are integer values.                              //20
Each function returns a pseudo-random number                       //21
e.g. $dist_uniform returns uniformly distributed random numbers    //22
mean, degree_of_freedom, k_stage                                   //23
```

```
(exponential, poisson, chi-square, t, erlang) > 0.                //24
seed = inout integer initialized by user, updated by function     //25
start, end ($dist_uniform) = integer bounding return values */    //26
endmodule                                                         //27
```

```
2.000000        8       57          0          4          9
3.000000           7          3          0          2
```

### 11.13.9 Programming Language Interface

*Key terms and concepts:* The C language **Programming Language Interface** (**PLI**) allows you to access the internal Verilog data structure • three generations of PLI routines • task/function (TF) routines (or utility routines) • access (ACC) routines access delay and logic values • Verilog Procedural Interface (VPI) routines are a superset of the TF and ACC routines

## 11.14 Summary

*Key terms and concepts:* concurrent processes and sequential execution • difference between a `reg` and a `wire` • scalars and vectors • arithmetic operations on `reg` and `wire` • data slip • delays and events

**Verilog on one page**

| Verilog feature | Example |
|---|---|
| Comments | `a = 0; // comment ends with newline`<br>`/* This is a multiline or block comment */` |
| Constants: string and numeric | **`parameter`** `BW = 32 // local, BW`<br>`` `define G_BUS 32 // global, `G_BUS ``<br>`4'b2  1'bx` |
| Names (case-sensitive, start with letter or '_') | `_12name A_name $BAD NotSame notsame` |
| Two basic types of logic signals: wire and reg | **`wire`** `myWire;` **`reg`** `myReg;` |
| Use continuous assignment statement with wire | **`assign`** `myWire = 1;` |
| Use procedural assignment statement with reg | **`always`** `myReg = myWire;` |
| Buses and vectors use square brackets | **`reg`** `[31:0] DBus; DBus[12] = 1'bx;` |
| We can perform arithmetic on bit vectors | **`reg`** `[31:0] DBus; DBus = DBus + 2;` |
| Arithmetic is performed modulo $2^n$ | **`reg`** `[2:0] R; R = 7 + 1; // now R = 0` |
| Operators: as in C (but not `++` or `--`) | |
| Fixed logic-value system | `1, 0, x (unknown), z (high-impedance)` |
| Basic unit of code is the module | **`module`** `bake (chips, dough, cookies);`<br>**`input`** `chips, dough;` **`output`** `cookies;`<br>**`assign`** `cookies = chips & dough;`<br>**`endmodule`** |
| Ports | `input or input/output ports are wire`<br>`output ports are wire or reg` |
| Procedures happen at the same time<br>and may be sensitive to an edge, **posedge**, **negedge**, or to a level. | **`always`** `@rain sing;` **`always`** `@rain dance;`<br>**`always`** `@(`**`posedge`** `clock) D = Q; // flop`<br>**`always`** `@(a` **`or`** `b) c = a & b; // and gate` |
| Sequential blocks model repeating things:<br>`always`: executes forever<br>`initial`: executes once only at start of simulation | **`initial`** `born;`<br>**`always`** `@alarm_clock` **`begin`** `: a_day`<br>`metro=commute; bulot=work;`<br>`dodo=sleep;`<br>**`end`** |
| Functions and tasks | **`function ... endfunction`**<br>**`task ... endtask`** |
| Output | `$display("a=%f",a);$dumpvars;$monitor(a)` |
| Control simulation | `$stop; $finish // sudden/gentle halt` |
| Compiler directives | `` `timescale 1ns/1ps // units/resolution `` |
| Delay | `#1 a = b;  // delay then sample b`<br>`a = #1 b;  // sample b then delay` |