Hardware:

Verilog:

language features

→ Module definitions are disjoint

syntax:

```
module module name (list of ports);
.....
.....
endmodule
```

→ module can be instantiated in another module

→ instantiation allows hierarchy

assign statement ⇒ Behavioural description.

↳ Continuous assignment.

LHS is net type (wire)

RHS is both net & register type.

typically assign statements are right after port & net declarations. Used to mainly model combinational circuits

Datatypes: two types

→ Net : Default value : z (high impedence)

1) Continuously driven
2) cannot store value
3) used to model connections in cont assignments and instantiations

→ Register : Default value : x (unknown logic state).

1) Retains value
2) storage elements & sometimes translate into combinatorial circuits.

Net: wire, wor, wand, tri, supply0, supply1

tri & net type are similar datatypes.

Register: reg, integer, real, time.

Synthesis: Design process of hardware/modules.
Simulation: Process such as timing & delays to verify synthesis.

Asynchronous: reset signal in sensitivity list
Synchronous: only clock in sensitivity list.

integer: register datatype to manipulate quantities.
useful in loop counting
treated as 2's complement signed integer
Default size is 32-bits.
real: Used to store floating point
rounded off when stored in integer.
time: Used to track time. ($time = system function)
Syntax: time curr_time. gives current time.
curr_time = $time.
Vectors: multiple bit quantities.
no ranges → single bit value
[range1:range2] variable.
⇒ Variable of size $2^{range1} - 1$: range1 to range2
ranges can be arbitrary values
ex: [7:0]a =>

*** [1:8]a =>(a)

important difference between [7:0]a & [1:10]a.

part-selects: reg [31:0] IR ***
reg [7:0] a => a = IR[31:24]

Multi dimensional arrays:

ex: reg [31:0] register_bank[15:0] => 16 32-bit registers.
to access above example we need two indices.
eg: register_bank[5] => gives 32-bit value at 5th index.
register_bank[5][4] => gives 4th bit of 5th 32 bit register.

Parameter: Constant with given name.

ex:
```
module Counter (clear, clk, count);
  parameter N=7
   input clk, clear;
   output [0:N] count;        --> By default output is of type net (wire)
    reg [0:N] count;
   always @(negedge clk) begin
     if (clear)
          count<=0;           --> Any variable inside always block must be of type reg
     else
          count<=count+1;
   end
endmodule.
```

`timescale` specifies precision for delays.

time scale directives:

syntax: `timescale <reference_time_unit>/<time_precision>

Specifies unit of time for measurement

Instantiation: positional association
explicit association

Operators:

Arithmatic operators : Unary & Binary.

Unary → (+) & (-) Binary → (+), (-), (*), (/), (%), (**)

Logical operator: Only logic values !, &&, ||.

Relational operators: !=, ==, >=, <=, <, >.

Bitwise operators: ~, &, |, ^ → exclusive OR, ~^ → ex-NOR

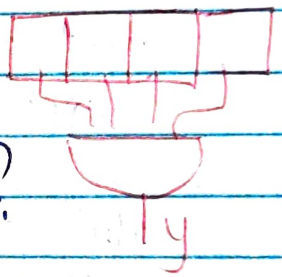
** Reduction operators: Converts multiple bit input to single bit o/p

& → bitwise AND ~& → bitwise NAND ^ → bitwise EX-OR

| → bitwise OR ~| → bitwise NOR ~^ → bitwise EX-NOR

ex: wire [3:0] x ; wire y;

assign y = &x; ⇒



→ Reduction operator.

ex: a=0111 b=1100

c=0100

f1 = ^a ⇒ 0 ⊕ 1 ⊕ 1 ⊕ 1 = 1

f2 = &(a^b) ⇒ 0 1 1 1

1 1 0 0

0 1 1 ⇒ &(1011) = 0

f3 = ^a & ~^b ⇒ ^a ⇒ 0 ⊕ 1 ⊕ 1 ⊕ 1 = 1

~^b ⇒ 0 ⊙ 1 ⊙ 0 ⊙ 0 = 1

⇒ f3 = 1 & 1 = 1

Shift Operators: >> → shift right << → shift left.

*** >>> → arithmetic shift right.

ex: 2's complement no system.

(+)/(-) sign binary numbers.

for 2's Complement no's

shift right ⇒ ÷2 } meanings

shift left ⇒ *2.

for -ve no: → shift with 1s

for +ve no: → Shift with 0s

Conditional Operators:

expression ? true_expr : false_expr

ex: assign a = (b>c) ? b : c;

→ means assign a to b (or) c based on (b>c) result.

Concatenation Operator: { , , , }

Replication Operator: { const { val } }

ex: {3{3'b001}} => 001001001

** presence of z or x in a reg or wire in arithmetic expression result whole expression to be x (unknown)

Behavioral Modeling:

ex1: 16x1 multiplexer.

a) Pure behavioral modelling;

```
module mux16to1(in,sel,out);
  input [15:0] in;
  input [3:0] sel;
    output out;
  assign out = in[sel];
endmodule
```

//testbench for 16to1 mux.

```
module muxtest;
  reg [15:0] A;
  reg [3:0] S;
  wire F;
mux16to1 m1(.A(in),.S(sel),.F(out));
initial begin
    $dumpfile("mux16to1.vcd");
    $dumpvars(0,muxtest);
    $monitor($time,"A=%h,S=%h,F=%b",A,S,F);
```

//testbench continued;

```
#5 A=16'h3fCa;S=4'h0;
#5 S=4'h1;
#5 S=4'h6;
#5 S=4'hC;
#5 $finish;
end    //end doesn't have
endmodule //semicolon
```

b) Behavioral modeling of 4x1 mux & structural modeling of 16x1

```
module mux4to1(in,sel,out);
  input [3:0]in;
   input[1:0]sel;
   output out;
   assign out = in[sel];
endmodule

module mux16to1 (in,sel,out);
   input [15:0]in;
   input [3:0]s;
   output out;
    wire [3:0]w;
  mux4to1 m1 (in[3:0], s[1:0], w[0]);
  mux 4to1  m2 (in[7:4], s[1:0], w[1]);
  mux 4to1 m3(in[11:8]; s[1:0], w[2]);
  mux4to1 m4 (in[15:12],s[1:0], w[3]);
  mux4to1 m5 (w[3:0], s[3:2], out);
endmodule
```

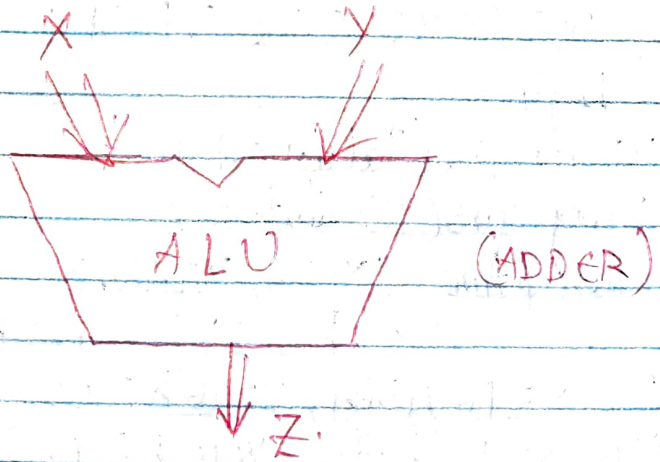
c) Behavioral modeling of 2x1 mux & structural model of 4x1 mux

```
module mux8x1(in,sel,out);
  input [1:0]in;
   input sel;
   Output out;
  assign out= in[sel];
  endmodule
```

```
module mux4to1 (in, Sel, out);
    input [3:0] in;
    input [1:0] sel;
    output out; wire [1:0] w;
  mux2x1 m1 (in[1:0], sel[0], w[0]);
   mux2x1 m2 (in[3:2], sel[0], w[1]);
   mux2x1 m3 (w, sel[1], out);
endmodule
```

d) Structural implementation of 2x1 mux:

```
module mux2x1 (in,sel, out);
    input [1:0] in;
     input sel;
     output out; wire [1:0] w;
     AND g1 (w[0], sel, in[0]);
     AND g2 (w[1], sel, in[1]);
      OR g3 (out, w[0], w[1]);
endmodule
```



Generation of status flags: sign, zero, carry, parity, overflow

- 1/0 — 0/Nonzero — Cout — 1's count — overflow

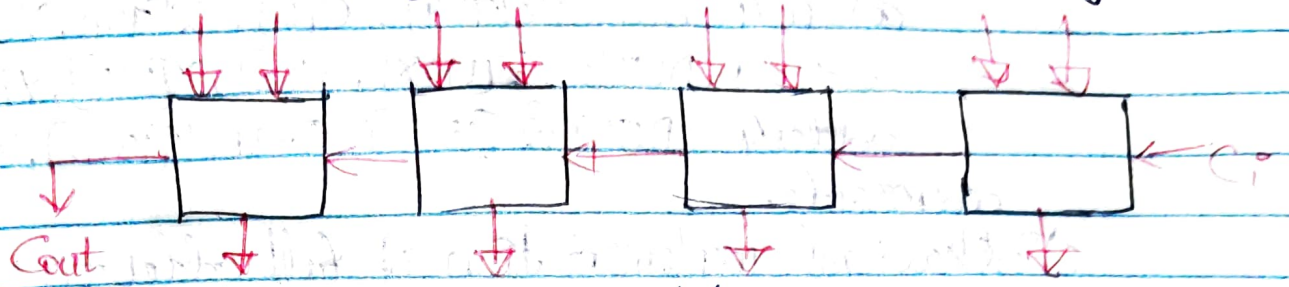
16-bit adder - example

a) Behavioral description

```
module ALU (X, Y, Z, sign, zero, carry, parity, overflow);
 input [15:0] x, y;
 output [15:0] z;
 output sign, zero, carry, parity, overflow;

 assign {carry, z} = x+y;
 assign sign = z[15];
 assign zero = ~|z;
 assign parity = ~^[z];
 assign overflow = (x[15] & y[15] & ~z[15]) | (~x[15] & ~y[15] & z[15]);
endmodule.
//testbench
 module ALUtest;
  reg [15:0] x, y;
  wire [15:0] z;
  wire S, ZR, C, P, Of;
  ALU DUT (x, y, z, S, ZR, C, P, Of);
initial begin
    $dumpfile("alu.vcd");
    $dumpvars("0, alutest");
 $monitor($time, "x=%h, y=%h, z=%h, S=%b, ZR=%b,
                  C=%b, P=%b, Of=%b", x, y, z, S, ZR, C, P, Of);
    #5 x=16'h8fff; y=16'h8000;
    #5 x=16'hfffe; y=16'h0002;
    #5 x=16'hAAAA; y=16'h5555;
    #5 $finish;
     end endmodule.
```

b) 16-bit adder using 4-bit adders and ripple carry.



behavioral description of adder 4-bit:

```
module adder4 (S, Cout, A, B, Cin);
  input [3:0] A,B;
  input Cin;
  output [3:0] S;
  output Cout;
 assign {Cout, S} = A+B+Cin;
endmodule
```

Structural description of 16-bit adder from 4-bit adder.

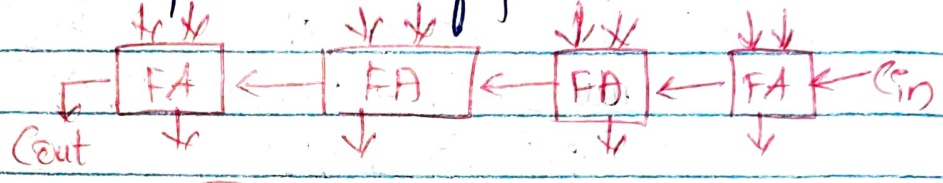
```
module ALU (X, Y, Z, Sign, Carry, Zero, parity, overflow);
  input [15:0] X, Y;
  output [15:0] Z;
  output sign, carry, zero, parity, overflow;
  wire [3:1] C;

  assign  sign = Z[15];
  assign  zero = ~|[Z];
  assign  parity = ~^(Z);
  assign  overflow = (X[15] & Y[15] & ~Z[15]) |
                     (~X[15] & ~Y[15], Z[15]);

// module instantiations;
```

```
adder4 DUT1 (Z[3:0], C[1], X[3:0], Y[3:0], 1'b0);
adder4 DUT2 (Z[7:4], C[2], X[7:4], Y[4:4], C[1]);
adder4 DUT3 (Z[11:8], C[3], X[11:8], Y[11:8], C[2]);
adder4 DUT4 (Z[15:12], carry, X[15:12], Y[15:12], C[3]);
endmodule.
```

c) Behavioral implementation of full adder to 4bit adder.



Ripple Carry adder:

```
module adder4 (S, C, A, B, Cin);
  input [3:0] A, B;
  input Cin;
  output [3:0] S;
  output C; wire [3:1] c;
 full adder fa1 (s[0], c[1], A[0], B[0], Cin);
 full adder fa2 (s[1], c[2], A[1], B[1], c[1]);
 full adder fa3 (s[2], c[3], A[2], B[2], c[2]);
 full adder fa4 (s[3], C, A[3], B[3], c[3]);
endmodule

module fulladder (sum, carry, a, b, cin);
  input a, b, cin;
  output sum, carry;
  assign {carry, sum} = a+b+cin;
endmodule.
```

d) Structural model of full adder:

```
module fulladder(sum, carry, a, b, cin);
  input a, b, cin;
  output sum, carry;
  wire s1, c1, c2;
  XOR g1(s1, a, b);
  XOR g2(sum, s1, cin);
  AND g3(c1, a, b);
  AND g4(c2, s1, cin);
  XOR g5(carry, c1, c2);
endmodule
```

Ripple carry adder – slow, simple
Carry lookahead adder – fast, complex

$$g_i = A_i \cdot B_i$$
$$P_i = A_i \oplus B_i$$

Carry look ahead principle $C_{i+1} = g_i + P_i C_i$

$$C_1 = g_0 + P_0 C_0$$
$$C_2 = g_1 + P_1 C_1$$
$$C_3 = g_2 + P_2 C_2$$
$$C_4 = g_3 + P_3 C_3$$

} using these expressions we can find future carry

$$\Rightarrow S_0 = A_0 \oplus B_0 \oplus C_0 = P_0 \oplus C_0$$
$$S_1 = P_1 \oplus C_1$$
$$S_2 = P_2 \oplus C_2$$
$$S_3 = P_3 \oplus C_3$$

* Circuit first generates g_i & P_i terms and then carry and sum conditions are implemented

Verilog description styles.

```
module modulename;
  signal declarations
  assign statements
  procedural statements.
endmodule
```

examples with assign statements.

```
module genemux(data, select, out);
input [15:0] data;
input [3:0] select;
output out;
assign out = data[select];
endmodule.
```

** conditional operator generates MUX

every conditional operator generates 2x1 MUX
Based on the size of i/o the MUX count increases.

```
module gen_decoder (out, in, sel);
input in;
input [1:0] sel;
output [0:3] out;
assign out[sel] = in;
endmodule
```

Level sensitive D-type latch

```
module level_sensitive_latch (D, En, Q);
input D, En;
Output Q;
```

```
    assign Q = En ? D : Q;
endmodule
```

If input is somehow in output then memory element / latch is generated by synthesizing tool.

SR Latch:

```
module sr_latch (Q, Q̄, S, R);
    input S, R;
    output Q, Q̄;
    assign Q = ~(R & Q̄);
    assign Q̄ = ~(S & Q);
endmodule
```

Procedural assignments:

Procedural blocks:

initial blocks: Only in test benches / not in synthesis

Always blocks: Both synthesis & test benches

Sequential statements: executed in the order of appearance.

initialization and declaration can be merged.

Only reg type can be used inside procedural blocks.

This is because of mapping to hardware registers when the condition of event expression is not evaluated.

a) Sequential statement blocks:

```
begin
    ;
end
```

b) if ... else

c) if ... elseif ... elseif ... else

c) case statements

```
Case (<expression>)
  expr1 :
  expr2 :
    .
    .
  exprn :
  default :
endCase.
```

two variation:

Casez: z values are don't care

Casex: z, x values are don't care.

ex:

```
reg [3:0] state;
integer next_state;
Casex (State)
  4'b1xxx: Next_state=0;
  4'b01xx : next_state=1;
  4'b001x : next_state=2;
  4'b xxx1 : next_state=3;
 default : next_state=0;
endCase
```

d) while loop;

```
while (<expression>)
    sequential statements;
```

ex:

```
integer mycount;
initial begin
    while (mycount<=255)
      begin
```

```
        $display ("My count: %d", mycount);
        mycount = mycount+1;
    end
end
```

e) For loop: (initialization, termination → control)

```
for(expr1; expre2; expr3)
    sequential statements;
```

ex:

```
integer mycount;
reg [100:0] data;
integer i;
initial
    for (mycount = 0; mycount<=255; mycount=count+1)
        $display("My count: %d", mycount);
initial
    for(i=0; i<=100; i=i+1);
        data[i] = 1'b0;
```

f) repeat:

```
repeat (<expression>)
    sequential statement;
```

ex:

```
initial begin
    repeat(5)
        $display ("Hi");
end
```

g) forever

ex: forever # 5 clk = ~clk.

```
forever
    sequential statements;
```

Procedural Statements: Examples:

```
module 2x1mux (in1, in0, sel, out);
    input in1, in0, sel;
    output out;
      reg out;
    always @ (in1 or in0 or s)
          if (s)
              out = in1;
          else
              out = in0;
endmodule
```

→ or can be replaced with ','

→ Sensitivity list can be '*'

Synchronous D type negedge triggered ff.

```
module dff_ne (D, clk, Q, Qbar);
    input D, clk;
    output reg Q, Qbar;
    always @ (negedge clk)
          begin
              Q = D;
              Qbar = ~D;
          end
endmodule
```

Procedural assignments:

used to update only reg, real, integer, time.

two types

- blocking assignments: "="
- non-blocking assignments: "<="

Swapping two variables: snippet only

```
always @(posedge clk)
    a=b;
always @(posedge clk)
    b=a;
```

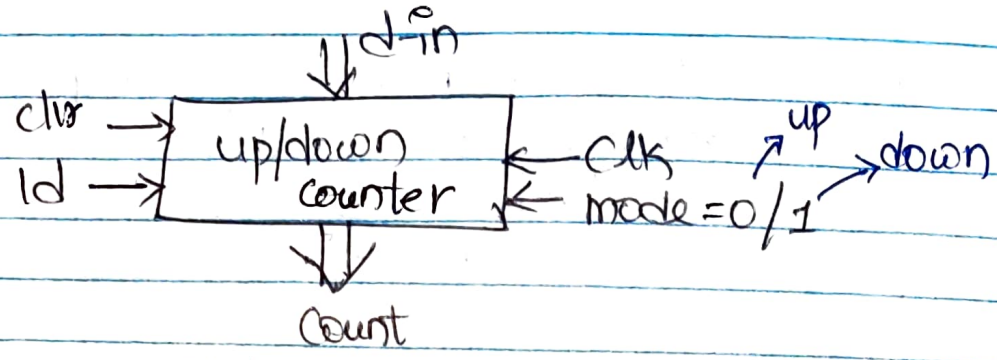
* This gives rise to RACE. Behaviour depends on simulator.

```
always @(posedge clk)
    a<=b;
always @(posedge clk)
    b<=a;
```

** Since non-blocking assignments read first assign later, variable values are swapped correctly. NO RACE condition here.

Behavioral description of 8x1 multiplexer.

```
module mux8x1 (in, sel, out);
  input [7:0] in;
  input [2:0] sel;
  output out;
  always @(*)
    begin
      case(sel)
        3'b000: out = in[0];
        3'b001: out = in[1];
        3'b010: out = in[2];
        3'b011: out = in[3];
        3'b100: out = in[4];
        3'b101: out = in[5];
        3'b110: out = in[6];
        3'b111: out = in[7];
        default: out = 1'bx
      endcase
    end
endmodule.
```



Synchronous Up/Down Counter

```
module counter.(mode, clr, clk, ld, din, count);
  input mode, clr, ld, clk;
  input [7:0] din;
  output reg [7:0] count;
  always @(posedge clk)
      if (ld)
          count <= din;
      else if (clr)
          count <= 0;
      else if (mode)
          count <= count+1;
      else
          count <= count-1;
endmodule
```

Parameterized design : an N-bit counter

```
module counter( clear, clock, count);
  Parameter N=7;
  output reg [N:0] count;
  always @(negedge clock)
      if (clear) count <= 0;
      else count <= count+1;
endmodule.
```

Using more than one clk: Complex frequencies & logics:

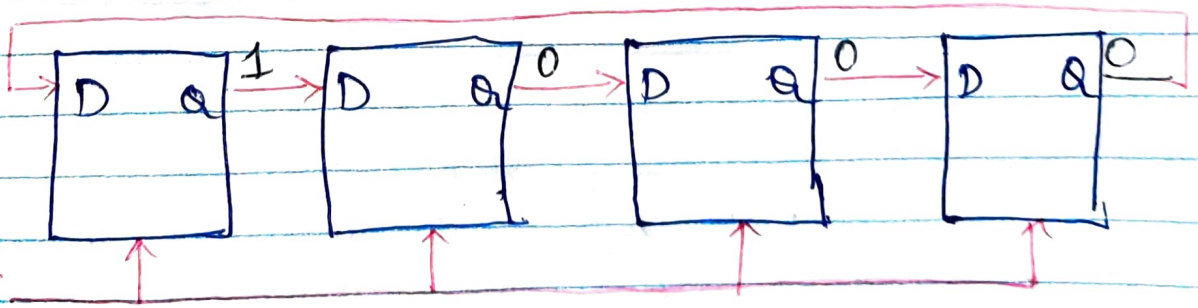
```
module multipleclk(clk1, clk2, a, b, c, f1, f2);
    input clk1, clk2, a, b, c;
    output reg f1, f2;
    always @(posedge clk1)
        f1 <= a&b;
    always @(posedge clk2)
        f2 <= a^b^c;
endmodule
```

Another example with single clk for multiple blocks.

```
module singleclk(clk, a, b, t, f);
    input clk, a, b;
    output reg f;
    reg t;
    always @(posedge clk)
        f <= t&b;
    always @(negedge clk)
        t <= a^b;
endmodule
```

* Single clk with multiple edge controlled blocks.

Ring counter: Shift register with D-flip flops connected in ring. as below:



8-bit ring counter example:

```
module ring_counter (clk, init, count);
    input clk, init;
    output reg [7:0] count;
    always @ (posedge clk)
      begin
        if (init)
            count = 8'b10000000;
        else
          begin
            count<=count<<1;
            count[0]<=count[7];
          end
      end
endmodule
```

** using non-blocking assignment for proper rotation

Never mix blocking and non-blocking assignments in the same procedural blocks (always/initial).

Generate Blocks:

Syntax

```
generate
  .....
  .....
  .....
endgenerate
```

Used as a loop to instantiate multiple modules. Can be used to create parameterized modules. Generate block can contain modules, UDPs, procedural blocks.

Special variable genvar : ignored in simulation/synthesis

ex: Bitwise exclusive OR:

```
module biwiseexor_gen (f, a, b);
    parameter N=16;
    input [N-1:0] a, b;
    output [N-1:0] f;
    genvar p;
    generate for(p=0; p<N; p=p+1);
        begin xorlp
            xor xg (f[p], a[p], b[p]);
        end
    endgenerate
endmodule
```

accessing gates is as follows:

```
xorlp[0].xg
*  xorlp[1].xg
** <begin block name>[index].gatename.
```

All the gates are stored as vector inside the blockname. and can be accessed with indices.

Q2: N-bit Ripple Carry Adder:

```
module NRCA (a, b, C, sum, Cout);
    parameter N=8;
    input [N-1:0] a, b;
    output [N-1:0] sum;
    wire [N-1:0] Carry;
    assign Carry[0] = C;
    assign Cout = Carry[N]
```

```
genvar i;
generate for(i=0; i<N; i++)
    begin for_loop
        wire t1, t2, t3;
        xor G1(t1, a[i], b[i])
        xor G2(sum[i], t[i], carry[i])
        and G3(t2, a[i], b[i])
        and G4(t3, t1, carry[i]);
        OR G5(carry[i+1], t2, t3);
    end
endgenerate
end module.
```

User Defined Primitives:

Behavioral way specification

truth table → combination circuit

state transition → sequential circuit.

syntax:

```
truth table
    <input1> <input2> <input> :<output>;
state table
<input1> <input2> <input3> :<present> <next>.
```

Rules:

→ No vectors allowed as inputs.

→ Order should be taken care in same order.

→ single scalar output in one UDP.

→ Can be specified for don't care combinations.

ex: full adder primitive:

```
primitive sum_udp(sum, a,b,c);   // primitive declaration
  input a, b, c;                 // input order & scalar
  output sum;                    // output
  table                          // table for combinational logic
    // a   b   c     :  sum      // Comment line
       0   0   0     :   0
       0   0   1     :   1
       0   1   0     :   1
       0   1   1     :   0
       1   0   0     :   1
       1   0   1     :   0
       1   1   0     :   0
       1   1   1     :   1
  endtable                       // ending table
endprimitive                     // ending primitive
```

including primitives is same as module instantiation.

```
module fa_udp(sum, cout, a,b,c);
  input a,b,c;
  output sum, cout;

  sum_udp  SUM(sum, a,b,c);
  carry_udp  CARRY(cout, a,b,c);

endmodule:
```

Module description with don't cares.

```
primitive and4_udp (f,a,b,c,d);
    input a,b,c,d;
    output f;
    table
    // a  b  c  d  f
       0  ?  ?  ?  : 0;
       ?  0  ?  ?  : 0;
       ?  ?  0  ?  : 0;
       ?  ?  ?  0  : 0;
       1  1  1  1  : 1;
    endtable
endprimitive

module fourand_udp (f, a,b,c,d);
    input a,b,c;
    output f;
    and4_udp AND (f, a,b,c,d);
endmodule
```

Primitives are easy to implement in combinational circuits.

```
primitive mux_4x1 (f, S1, S0, a,b,c,d);
    input S1,S0, a,b,c,d;
    output f;
    table
    // S1 S0  a  b  c  d  f
       0  0   0  ?  ?  ?  : 0;
       0  0   1  ?  ?  ?  : 1;
```

```
    0   1   ?   0   ?   ?  :  0 ;
    0   1   ?   1   ?   ?  :  1 ;
    1   0   ?   ?   0   ?  :  0 ;
    1   0   ?   ?   1   ?  :  1 ;
    1   1   ?   ?   ?   0  :  0 ;
    1   1   ?   ?   ?   1  :  1 ;
  endtable
endprimitive
```

Modeling sequential circuits using primitives.

Level sensitive D-latch:

```
primitive D latch (q, d, clk, clr);
  input d, clk, clr;
  output reg q;        // reg declaration for output in
                       // sequential circuits.
  initial
    q = 0;             // initial value to preset state
  table
  // d   clk  clr   q   q-new.
    ?    ?    1  :  ?  :  0;
    0    1    0  :  ?  :  0;
    1    1    0  :  ?  :  1;
    ?    0    0  :  ?  :  -;   // - -> retains state
  endtable
endprimitive.
```