Calling a function

- A function call is an operand within an expression
- The order of evaluation of the arguments to a function call is undefined.
- Example:

  word=control? {getbyte(msbyte),getbyte(lsbyte)}:0;

Function declarations

- The following example defines a function called getbyte, using a range specification:

```
function [7:0] getbyte;
input [15:0] address;
begin
        // code to extract low-order byte from addressed word
        . . .
        getbyte = result_expression;
end
endfunction
```

- Or using the second form of a function declaration, the function could be defined as follows:

```
function [7:0] getbyte (input [15:0] address);
begin
        // code to extract low-order byte from addressed word
        . . .
        getbyte = result_expression;
end
endfunction
```

Function rules

- Functions are more limited than tasks. The following rules govern their usage:
  a) A function definition shall not contain any time-controlled statements, that is, any statements containing **#**, **@**, or **wait**.
  b) Functions shall not enable tasks.
  c) A function definition shall contain at least one input argument.
  d) A function definition shall not have any argument declared as output or inout.
  e) A function shall not have any nonblocking assignments or procedural continuous assignments.
  f) A function shall not have any event triggers.

Returning a value from a function

- The function definition shall implicitly declare a variable, internal to the function, with the same name as the function.
  - is the same type as the type specified in the function declaration
  - It is illegal to declare another object with the same name as the function in the scope where the function is declared

  getbyte = result_expression;

User Define Primitives (UDP)

- Verilog has built in primitives like gates, trans-mission gates, and switches. This is rather small number of primitives, if we need more complex primitives, then Verilog provides UDP, or simply User Defined Primitives. Using UDP we can model
  - Combinational Logic
  - Sequential Logic
- We can include timing information along with this UDP to model complete ASIC library models.

Example: a function factorial that returns an integer value

```
module tryfact;
// define the function
function automatic integer factorial;
input [31:0] operand;
integer i;
if (operand >= 2)
            factorial = factorial (operand - 1) * operand;
else
            factorial = 1;
endfunction
// test the function
integer result;
integer n;
initial begin
            for (n = 0; n <= 7; n = n+1) begin
                        result = factorial(n);
                        $display("%0d factorial=%0d", n, result);
            end
end
endmodule   //tryfact
```

The simulation results are as follows:

```
0 factorial=1
1 factorial=1
2 factorial=2
3 factorial=6
4 factorial=24
5 factorial=120
6 factorial=720
7 factorial=5040
```

UDP

- UDP begins with reserve word primitive and ends with endprimitive. This should follow by ports/terminals of primitive. This is kind of same as we do for module definition. UDP’s should be defined outside module and endmodule

```
//This code shows how input/output p
// and primitve is declared
primitive udp syntax ( a, b, c, d );
output a;
input b,c,d;
// UDP function code here
endprimitive
```

- In the above code, udp syntax is the primitive name, it contains ports a, b,c,d.

Constant function calls

```
module ram_model (address, write, chip_select, data);
            parameter data_width = 8;
            parameter ram_depth = 256;
            localparam addr_width = clogb2(ram_depth);
            input [addr_width - 1:0] address;
            input write, chip_select;
            inout [data_width - 1:0] data;
            //define the clogb2 function
            function integer clogb2;
            input [31:0] value;
            begin
                        value = value - 1;
                        for (clogb2 = 0; value > 0; clogb2 = clogb2 + 1)
                                    value = value >> 1;
            end
endfunction

reg [data_width - 1:0] data_store[0:ram_depth - 1];
//the rest of the ram model
```

- An instance of this ram_model with parameters assigned is as follows:

```
            ram_model #(32,421) ram_a0(a_addr,a_wr,a_cs,a_data);
```

UDP Symbols

?	: 0 or 1 or X : ? var can be 0 or 1 or x
b	: 0 or 1 : Same as ?, but x is not included
f	: (10) : Falling edge on an input
r	: (01) : Rising edge on an input
p	: (01)(0x)(x1)(1z)(z1): Rising edge incl x and
n	: (10)(1x)(x0)(0z)(z0): Falling edge inclx and
(vw)	: change from v to w: v,w can be 0,1,x,? or
-	: no change : No Change

UPD rules

- A UDP can contain only one output and up to 10 inputs max.
  - Storage: 1-5 is less 1K; 10 is 623K
- Output Port should be the first port followed by one or more input ports.
- All UDP ports are scalar, i.e. Vector ports are not allowed.
- UDP's can not have bidirectional ports.
- The output terminal of a sequential UDP requires an additional declaration as type reg.
- It is illegal to declare a reg for the output
- terminal of a combinational UDP
- May not appear between keywork module andendmodule

UDP Body

- Functionality of primitive (both combinational and sequential) is described inside a table, and it ends with reserve word endtable as shown in code below. For sequential UDP, we can use initial to assign initial value to output.

```
primitive udp body ( a, b, c );
output a;
input b,c;
          // A = B | C;
table  // B C : A
             ? 1 : 1;
             1 ? : 1;
             0 0 : 0;
endtable
endprimitive
```

- A UDP cannot use 'z' in input table

Port Rules

- A UDP can contain only one output and up to 10 inputs max.
- Output Port should be the first port followed by one or more input ports.
- All UDP ports are scalar, i.e. Vector ports are not allowed.
- UDP's can not have bidirectional ports.
- The output terminal of a sequential UDP requires an additional declaration as type reg.
- It is illegal to declare a reg for the output terminal of a combinational UDP

Example

```
primitive multiplexer(mux, control, dataA, dataB
)
output mux ;
input control, dataA, dataB ;
table
// control dataA dataB mux
        0 1 0 : 1 ;
        0 1 1 : 1 ;
        0 1 x : 1 ;
        0 0 0 : 0 ;
        0 0 1 : 0 ;
        0 0 x : 0 ;
        1 0 1 : 1 ;
        1 1 1 : 1 ;
        1 x 1 : 1 ;
        1 0 0 : 0 ;
        1 1 0 : 0 ;
        1 x 0 : 0 ;
        x 0 0 : 0 ;
        x 1 1 : 1 ;
endtable
endprimitive
```

```
primitive multiplexer(mux, control,
dataA, dataB )
output mux ;
input control, dataA, dataB ;
table
// control dataA dataB mux
// control dataA dataB mux
        0 1 ? : 1 ; // ? = 0,1,x
        0 0 ? :0 ;
        1 ? 1 :1 ;
        1 ? 0 :0 ;
        x 0 0 :0 ;
        x 1 1 :1 ;
endtable
endprimitive
```

UDP instantiation

```
'include "udp body.v"
module udp body tb();
reg b,c;
wire a;
udp body udp (a,b,c);
initial begin
        $monitor(" Bb== 0; c = 0;
        #1 b = 1;
        #1 b = 0;
        #1 c = 1;
        #1 b = 1'bx;
        #1 c = 0;
        #1 b = 1;
        #1 c = 1'bx;
        #1 b = 0;
        #1 $finish;
end
endmodule
```

Level Sensitive UDP

- Level-sensitive sequential behavior is represented the same way as combinational behavior, except that the output is declared to be of type **reg**, and there is an additional field in each table entry. This new field represents the current state of the UDP.
  - The output is declared as reg to indicate that there is an internal state. The output value of the UDP is always the same as the internal state.
  - A field for the current state has been added. This field is separated by colons from the inputs and the output.
- Sequential UDPs have an additional field inserted between the input fields and the output field, compared to combinational UDP. This additional field represents the current state of the UDP and is considered equivalent to the current output value. It is delimited by colons.

Table/Initial

- Table is used for describing the function of UDP. Verilog reserve world **table** marks the start of table and reserve word **endtable** marks the end of table.
  - Each line inside a table is one condition, as and when an input changes, the input condition is matched and the output is evaluated to reflect the new change in input.
- **initial** statement is used for initialization of sequential UDP's. This statement begins with the keyword **initial**. The statement that follows must be an assignment statement that assigns a single bit literal value to the output terminal reg.

```
primitive udp initial (a,b,c);
output a;
input b,c;
reg a;
initial a = 1'b1;
table
//udp initial behaviour
endtable
endprimitive
```

Example

```
primitive d_edge_ff(q, clock, data);
output q; reg q;
input clock, data;
table
            // obtain out on rising clock edge
            // clock data q q+
            (01) 0 : ? : 0 ;
            (01) 1 : ? : 1 ;
            (0?) 1 : 1 : 1 ;
            (0?) 0 : 0 : 0 ;
            // ignore negative edge of clock
            (?0) ? : ? : - ; - indicates node v
            // ignore data changes on steady clock
            ? (??) : ? : - ;
endtable
endprimitive
```

Example

```
primitive latch(q, clock, data) ;
output q; reg q ;
input clock, data;
table
// clock data q q+
        0 1 : ? : 1 ;
        0 0 : ? : 0 ;
        1 ? : ? : - ; // - = no change
endtable
endprimitive
```

Initial in UDP

- contents limited to one procedural assignment statement
- the procedural assignment statement must assign a value to a **reg** whose identifier matches the identifier of an output terminal
- the procedural assignment statement must assign one of the following values: 1b1 1b0 1bx 1 0

```
primitive srff (q,s,r);
output q;
input s,r;
reg q;
initial q = 1b1;
Table
  // s r q q+
     1 0 : ? : 1 ;
     f 0 : 1 : - ;
     0 r : ? : 0 ;
     0 f : 0 : - ;
     1 1 : ? : 0 ;
endtable
endprimitive
```

Edge Sensitive

- In level-sensitive behavior, the values of the inputs and the current state are sufficient to determine the output value. Edge-sensitive behavior differs in that changes in the output are triggered by specific transitions of the inputs.
- As in the combinational and the level-sensitive entries, a ? implies iteration of the entry over the values 0, 1, and x. A dash (-) in the output column indicates no value change.
- All unspecified transitions default to the output value x. Thus, in the previous example, transition of clock from 0 to x with data equal to 0 and current state equal to 1 result in the output q going to x.
- All transitions that should not affect the output must be explicitly specified. Otherwise, they will cause the value of the output to change to x. If the UDP is sensitive to edges of any input, the desired output state must be specified for all edges of all inputs.

Edge triggered

```
primitive jk edge ff(q, clock, j, k, preset
output q; reg q;
input clock, j, k, preset, clear;
table
      //clock jk pc state output/next state
      ? ?? 01 : ? : 1 ; //preset logic
      ? ?? *1: 1: 1 ;
      ? ?? 10 : ? : 0 ; //clear logic
      ? ?? 1*: 0: 0 ;
      r 00 00 : 0 : 1 ; //normal clocking c
      r 00 11: ?: - ;
      r 01 11: ?: 0 ;
      r 10 11: ?: 1 ;
      r 11 11: 0: 1 ;
      r 11 11: 1: 0 ;
      f ?? ??: ?: - ;
      b *? ?? : ? : - ; //j and k transi
      b ?* ??: ?: - ;
endtable
endprimitive
```

```
primitive latch(q, clock, data)
output q; reg q ;
input clock, data ;
table
// clock data state output/next state
            0 1 :?: 1 ;
            0 0 :?: 0 ;
            1 ? :?: - ; // - = no change
            // ignore x on clock when data equals state
            x 0 :0: - ;
            x 1 :1: - ;
endtable
endprimitive
```

Initial in UDP

```
primitive dff1 (q,clk,d);
input clk,d;
output q;
reg q;
initial q = 1b1;
table
   // clkd q q+
   p 0 : ? : 0 ;
   p 1 : ? : 1 ;
   n ? : ? : - ;
   ? * : ? : - ;
endtable
endprimitive
```

```
module dff (q,qb,clk,d);
input clk,d;
output q,qb;
dff1 g1 (qi,clk,d);
buf #3 g2 (q,qi);
not #5 g3 (qb,qi);
endmodule
```