



Delay and Conditional
Statement (Part 2) 



Chap 6: Introduction to HDL (f)



Credit to: Dr. MD Rizal Othman
Faculty of Electrical & Electronics Engineering
Universiti Malaysia Pahang

Ext: 6036

If...else statement

- If-else statements check a condition to decide whether or not to execute a portion of code.
- If a condition is satisfied, the code is executed. Else, it runs this other portion of code.
- begin and end must be used, when more than one statement needs to be executed for an if condition

Syntax : if

```
if (condition)
statements;
```

Syntax : if-else

```
if (condition)
statements;
else
statements;
```

Syntax : nested if-else-if

```
if (condition)
statements;
else if (condition)
statements;
.....
.....
else
statements;
```

If...else statement

reg f, g; // a new reg variable, g

```
always @(sel or a or b)
```

```
begin
```

```
  if (sel == 1)
```

```
    begin
```

```
      f = a;
```

```
      g = ~a;
```

```
    end
```

```
  else
```

```
    begin
```

```
      f = b;
```

```
      g = a & b;
```

```
    end
```

```
end
```

If statements can be nested if you have more complex behaviour to describe:

```
reg f, g;
```

```
always @(sel or sel_2 or a or b)
```

```
  if (sel == 1)
```

```
    begin
```

```
      f = a;
```

```
      if (sel_2 == 1)
```

```
        g = ~a;
```

```
      else
```

```
        g = ~b;
```

```
    end
```

```
  else
```

```
    begin
```

```
      f = b;
```

```
      if (sel_2 == 1)
```


```
        g = a & b;
```

```
      else
```

```
        g = a | b;
```

```
    end
```

Case Statement

- Case statements are used where we have one variable which needs to be checked for multiple values.
 - like an address decoder, where the input is an address and it needs to be checked for all the values that it can take.
 - Instead of using multiple nested if-else statements, one for each value we're looking for, we use a single case statement (*this is similar to switch statements in languages like C++*)
- 

Case Statement

- Case statements begin with the reserved word **case** and end with the reserved word **endcase**
- The cases, followed with a colon and the statements you wish executed
- It's also a good practice to have a **default** case. Just like with a finite state machine (FSM), if the Verilog machine enters into a non-covered statement, the machine hangs. Defaulting the statement with a return to idle keeps us safe.

Case Statement

```
module mux (a,b,c,d,sel,y);  
input a, b, c, d;  
input [1:0] sel;  
output y;
```

```
reg y;
```

```
always @ (a or b or c or d or sel)
```

```
case (sel)
```

```
0 : y = a;
```

```
1 : y = b;
```

```
2 : y = c;
```

```
3 : y = d;
```

```
default : y=0;
```

```
endcase
```

```
endmodule
```



casez and casex statement

- Special versions of the case statement allow the x and z logic values to be used as "don't care":
 - casez : Treats z as don't care.
 - casex : Treats x and z as don't care.



casez statement

```
module casez_example();  
reg [3:0] opcode;  
reg [1:0] a,b,c;  
reg [1:0] out;  
  
always @ (opcode or a or b or c)  
casez(opcode)  
    4'b1zzx : out = a; // Don't care about lower 2:1 bit, bit 0 match    with x  
    4'b01?? : out = b; // bit 1:0 is don't care  
    4'b001? : out = c; // bit 0 is don't care  
    default : out = 1;  
endcase  
endmodule
```


case statement

```
module casez_example();  
reg [3:0] opcode;  
reg [1:0] a,b,c;  
reg [1:0] out;
```

```
always @ (opcode or a or b or c)
```

```
case(opcode)
```

```
    4'b1zzx : out = a; // Don't care 2:0 bits
```

```
    4'b01?? : out = b; // bit 1:0 is don't care
```

```
    4'b001? : out = c; // bit 0 is don't care
```

```
    default : out = 0;
```


```
endcase
```

```
endmodule
```

Task

- Tasks are used in all programming languages, generally known as procedures or subroutines.
- The lines of code are enclosed in task....end task brackets.
- Data is passed to the task, the processing done, and the result returned.
- Included in the main body of code, they can be called many times, reducing code repetition.

Task

- tasks are defined in the module in which they are used. It is possible to define a task in a separate file and use the compile directive 'include' to include the task in the file which instantiates the task.
 - tasks can include timing delays, like posedge, negedge, # delay and wait.
 - tasks can have any number of inputs and outputs.
 - The variables declared within the task are local to that task. The order of declaration within the task defines how the variables passed to the task by the caller are used.
 - tasks can call another task or function.
 - tasks can be used for modeling both combinational and sequential logic.
 - A task must be specifically called with a statement, it cannot be used within an expression as a function can.
- 

Task

```
module task_calling (temp_a, temp_b, temp_c,
temp_d);
input [7:0] temp_a, temp_c;
output [7:0] temp_b, temp_d;
reg [7:0] temp_b, temp_d;

always @ (temp_a)
begin
    convert (temp_a, temp_b);
end
always @ (temp_c)
begin
    convert (temp_c, temp_d);
end


task convert;
input [7:0] temp_in; output [7:0] temp_out;
begin
    temp_out = (9/5) * ( temp_in + 32)
end
endtask
```

Task

```
module task_calling (temp_a, temp_b,  
temp_c, temp_d);  
input [7:0] temp_a, temp_c;  
output [7:0] temp_b, temp_d;  
reg [7:0] temp_b, temp_d;  
`include "mytask.v"  
  
always @ (temp_a)  
begin  
    convert (temp_a, temp_b);  
end  
  
always @ (temp_c)  
begin  
    convert (temp_c, temp_d);  
end  
  
endmodule
```

```
module simple_task();  
  
task convert;  
input [7:0] temp_in;  
output [7:0] temp_out;  
begin  
    temp_out = (9/5) *( temp_in + 32)  
end  
endtask  
  
endmodule
```

Functions

- function is same as a task, with very little differences:
 - functions are defined in the module in which they are used. It is possible to define functions in separate files and use compile directive 'include to include the function in the file which instantiates the task.
 - functions can not include timing delays, like posedge, negedge, # delay, which means that functions should be executed in "zero" time delay.
 - functions can have any number of inputs but only one output.
 - The variables declared within the function are local to that function. The order of declaration within the function defines how the variables passed to the function by the caller are used.
 - functions can be used for modeling combinational logic.
 - functions can call other functions, but can not call tasks.
- 

Functions

```
//calling functions
```

```
module function_calling (a,  
b, c, d, e, f);
```

```
input a, b, c, d, e ;
```

```
output f;
```

```
wire f;
```

```
`include "myfunction.v"
```

```
assign f = (myfunction  
(a,b,c,d)) ? e :0;
```

```
Endmodule
```

```
module simple_function();
```

```
function myfunction;
```

```
input a, b, c, d;
```

```
begin
```

```
    myfunction = ((a+b) + (c-  
d));
```

```
end
```

```
endfunction
```

```
endmodule
```