ELEC2221
Digital Systems and Signal Processing

Writing synthesizable SystemVerilog code
- Combinational logic
Writing synthesisable code for combinational logic

- when writing code for decoders or multiplexers, use conditional statements
  - if-then-else
  - case

- common mistake:
  conditional statements with incompletely specified signals cannot be mapped into combinational logic:

```plaintext
if(a)
  b = 1'b0; // b is incompletely specified
```

- use `unique` qualifier to ensure completeness checks
  - alternatively in case statements, use the `default` clause (but see later)
  - assign default values to all signals driven by combinational logic blocks

- in arithmetic circuit synthesis
  - arithmetic operations `+ -` are mapped into standard adders/subtractors
  - the multiplication operator `*` can be mapped into embedded multipliers but you have to follow code templates provided by the FPGA vendor
    - mapping `*` into combinational (cellular) multipliers using standard logic elements is very costly – use embedded multipliers instead.
Use **complete** signal specifications for combinational logic

```vhdl
// if-else statement
if ((x==0) || (x==5))
    $display("x is 0 or 5");
else if (x == 2)
    $display("x is 2");
else
    $display(" x is not 0, 2 or 5");
```

```vhdl
// if-else statement with unique qualifier
unique if ((x==0) || (x==5))
    $display("x is 0 or 5");
else if (x == 2)
    $display("x is 2");
else
    $display(" x is not 0, 2 or 5");
// other values of x cause a warning
```

- **unique** asserts that conditions are mutually exclusive and hence it is safe for the expressions to be evaluated in parallel.
  - Compilers may be able to detect potential overlap and issue compile-time warnings.
  - Simulators issue run-time warnings if conditions are evaluated not to be mutually exclusive.
  - Warnings are also issued if no condition is true, or if it is possible that no condition is true.
- If you get such warnings, your combinational logic description is very likely incomplete and hence will not map correctly into hardware!
Case statements

// case statement
logic [1:0] x;
case(x) //
  0,1: $display("x is 0 or 1");
  2: $display("x is 2");
endcase
//nothing is displayed if x==3

logic [1:0] x;
unique case(x) //
  0,1: $display("x is 0 or 1");
  2: $display("x is 2");
endcase
// if x==3 the simulator issues a warning
casex and casez statements for don’t cares in comparisons

```hierarchy
case(x)
    4'b1xxx: $display("y is 8 or more");
    4'bxxx0: $display("y is even");
endcase
```

```hierarchy
unique case(x)
    4'b1xxx: $display("y is 8 or more");
    4'bxxx0: $display("y is even");
endcase
// if e.g. x==4'b0xx1 the simulator
// issues a warning
```

```hierarchy
// casex statement with priority qualifier
priority case(x)
    4'b1xxx: $display("y is 8 or more");
    4'bxxx0: $display("y is even");
endcase
// if e.g. x = 4'b1xx0, the first case is selected
// and there will be no warning
```

```hierarchy
case(x)
    treats x and z as don’t cares
    casez treats z as don’t care

Can use ? as a synonym for z. Might give clearer code.

Only bit values other than don’t care bits are used in comparisons
```

- **priority** asserts that a series of conditions shall be evaluated in the order listed. A warning is issued if no condition is true, or it is possible that no condition is true.
SystemVerilog case inside

NB both casex and casez match "don't cares" on the input selector as well as the choices. This can be difficult to debug.

casez is considered to be better than casex, because 'x values are trapped.

SystemVerilog has introduced case inside. Only "don't cares" on choices are matched.

Has only been supported in tools very recently.

unique case(y) inside
4'b1????: $display("y is 8 or more");
4'b???0: $display("y is even");
endcase

Now this is the preferred form for case statements with don't cares.
module decoder2to4(  
    input logic [1:0] i, output logic [3:0] outp);

always_comb  
begin  
    outp = 4’b0000; // default value  
    unique case (i)  
    // unique will force completeness test  
    0: outp[0] = 1’b1;  
    1: outp[1] = 1’b1;  
    2: outp[2] = 1’b1;  
    3: outp[3] = 1’b1;  
endcase  
end  
endmodule
Arithmetic circuits - 8-bit adder

```
// 8-bit adder – using 9-bit addition to obtain carry out
module adder8 (input logic [7:0] A, B, input logic Cin,
output logic [7:0] Sum, output logic Cout);

always_comb
{Cout, Sum} = A + B + Cin;

endmodule
```
Synthesising tri-state gates

Note: not all FPGAs support tri-state gates; tri-state buses composed using the code below might synthesise to multiplexers which replace tri-state buses and are functionally equivalent.

Note: Use 'wire' type to connect outputs of tri-states together

```verbatim
// tri-state buffer
module tristate (input logic C, inp; output logic outp);

assign outp = C ? inp : 1'bZ ;

endmodule
```
Timing Constraints

- Unlike (most) software, hardware has to meet specifications, other than functionality
  - Power, area, timing
  - Of these, timing is usually the most critical: silicon is cheap; but power is also a major constraint
Clock Speed

- The maximum clock speed is defined by the choice of device.
- The desired clock speed can be set as a design objective:
  ```
  create_clock -name "clock" -period 5.000ns
  [get_ports {clock}]
  ```
- Defines 200 MHz clock
- Synopsys .sdc file format is used by most tools, including Quartus.
- Objective for Synthesis AND Place and Route
Timing Constraints

Clock period is 5 ns,
Clock Frequency is 200 MHz,
Max delay through Comb Logic is 3 ns
Logic Optimisation

• All optimisation, in terms of meeting constraints, assumes that there is more than one way to implement a function:

Assuming 1ns delay per gate, 4ns delay. Does not meet constraint for 200 MHz operation

3ns delay. Meets constraint
External Input and Output delays can be specified:

```
set_input_delay 2.000ns -clock "clock"
```

Thus the maximum delay through the input logic is 2ns, allowing for 1ns setup time.
Other Timing Constraints

• There are a large number of possible constraints, e.g.
  – `set_clock_uncertainty`, defines uncertainty in the clock
  – `set_clock_skew`, defines uncertainty in the clock paths
  – `set_output_delay`, like `set_input_delay`
  – `set_false_path`, defines paths that cannot ever be logically active
  – `set_multicycle_path`, allows a signal to propagate over multiple cycles (dangerous)

• All used for static timing analysis in Quartus and similar tools (not simulation)
What if constraints not met?

• Do nothing? Metastable behaviour, or miss data completely
  – See next lecture for metastability
• Restructure logic, but may need significant changes
• Lose clock synchronisation
  – Large designs may need complex clock trees (i.e. multiple levels of buffers) to cause clock to arrive everywhere simultaneously
  – May need multiple clock domains. Common in big designs.
Retiming

- Move logic between clock cycles:

- Balances delays between flip-flops. Can be done automatically, but makes cycle-by-cycle verification difficult.
Pipelining

• If delay through combinational logic is too great, logic can be divided:

• Clock speed is greater; time for one piece of data to go through (latency) is greater; throughput is also greater – one new result per clock cycle.
• Efficient if the pipeline is kept full
• Difficult to automate (though can be thought of as retiming)