1.0 Verilog Synthesis Methodology

Finbarr O'Regan (finbarr@ee.ucd.ie) October 2001

Synthesis is a contraint driven process i.e. the synthesis script needs timing constraints

Follow the following methodology for best results

1. Draw a simple block diagram, labelling all signals, widths etc.
2. Draw a timing diagram with as much detail as possible
3. Code the HDL according to the synthesizable templates
4. Do a quick, low effort, compile- just to see if it is synthesizable *before* simulating. Compare this to the block diagram. Look at the inference report:

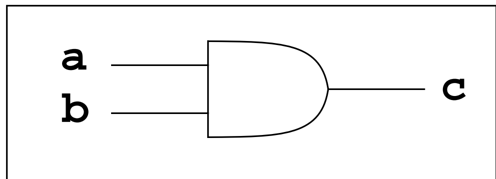
- count the number of flip flops - is it the same as the number of flip flops in the code.
- check for latches - did you want them. If not, latches are inferred in combinational procedures - the inferrence report tells you which combinational procedure and the name of the latch. Fully specify all variables in all cases to eliminate latches.
- Check the case statement inferrence. Was it full/parallel?
- Check any incomplete event list warnings?
- Check to see if there are any combinational feedback loops (typically only after a compile). Combinational feedback loops can be identified by the signal names in the timing loop.
- Check the schematic - any ports unconnected?
- Check to see if Designware and Ambitware components have been built correctly. Are these the components that you wanted? How many did you want?
- Never ignore any warning that the synthesis tool flags. All warnings need to be understood and typically signed off.

5. Simulate and compare with the timing diagram

- If your design doesn't meet timing by more than 10% of the clock period, then go back to the code. If you are within 10% of the clock period, then try a different compile strategy.

2.0 Synthesizeable Templates

2.1 Combinational Logic

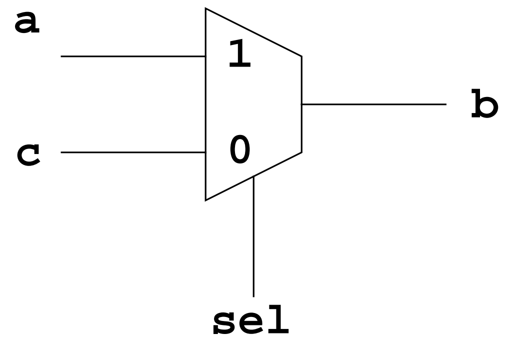



```
// Using a reg
// -----------------------------
wire a,b;
reg c;
always @ (a or b)
   c = a & b;

// Using a wire
// -----------------------------
wire a,b,c;
assign c = a & b;

// using a built in primitive (without instance name)
// -----------------------------
reg a,b;
wire c;
and (c,a,b);  // output is always first in the list

// using a built in primitive (with instance name)
// -----------------------------
reg a,b;
wire c;
and u1 (c,a,b);  // output is always first in the list

// if c is an output
// -----------------------------
output c;
reg a,b;
assign c = a & b;
```

2.2 Multiplexers

2.2.1 Multiplexer using a procedure

```
// 1. using an always
always@(a or b or sel)
   if (sel == 1'b1)
      c = a;
   else
      c = b;
```



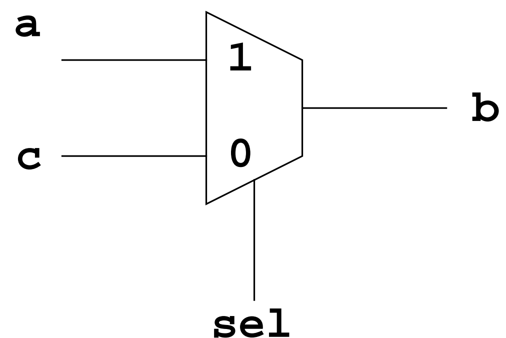
Use default assignments to prevent latches: every

2.2.2 Multiplexer using the ternary operator

```
// 2. using the ternary operator
wire c = sel ? a : b;
```

2.2.3 Multiplexer using the case statement

```
// 3. using the case statement
always @ (a or b or sel)
   case (sel)
      1'b1: c = a;
      1'b0: c = b;
   endcase
```



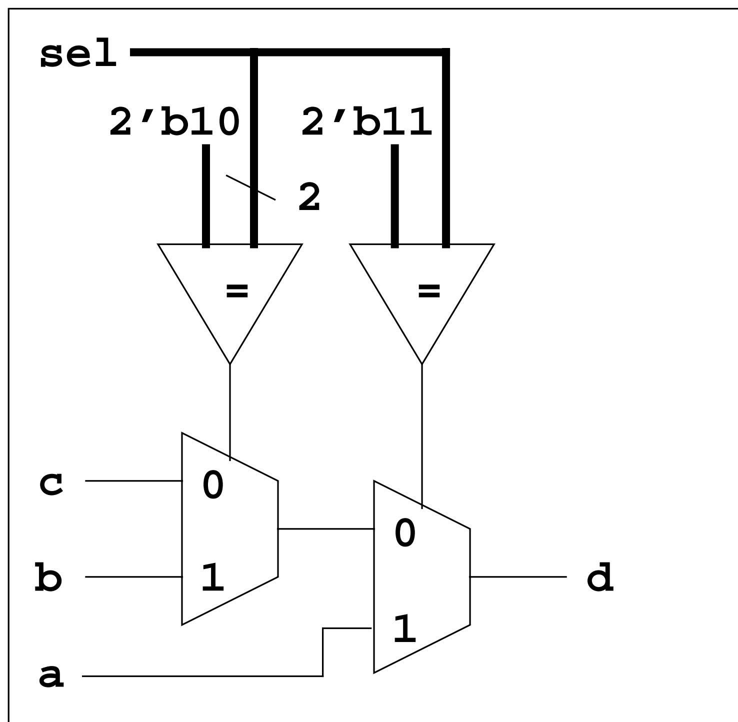
2.3 Priority Decoders

2.3.1 Priority Decoder using a case statement

```
// 1. using a case statement
always @ (sl or a or b or c)
   case (sel)
      2'b11:  d = a;
      2'b10:  d = b;
      default: d = c;
   endcase
```

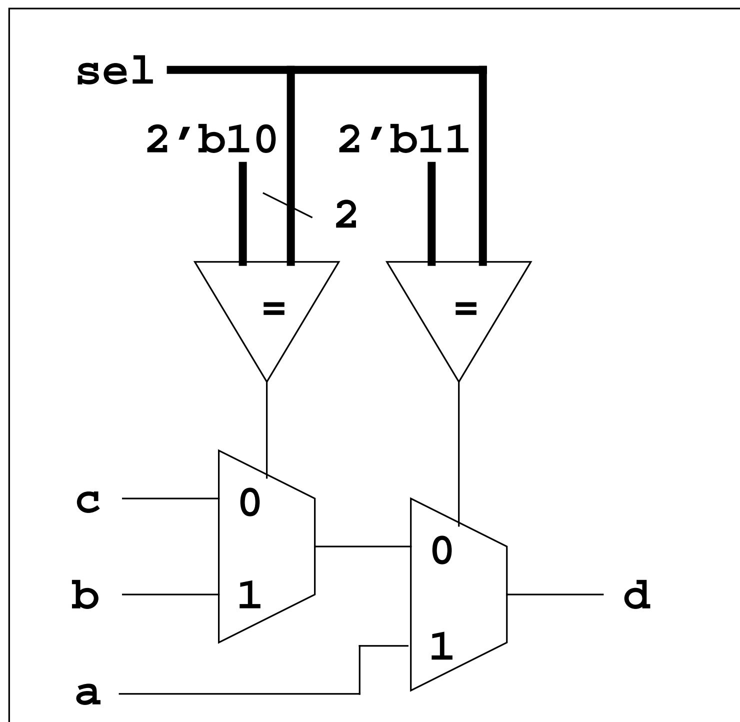
1. Both case and if statements result in priority structures.

2. The order of the variables determines the priority



2.3.2 Priority Decoder using an if/else statement

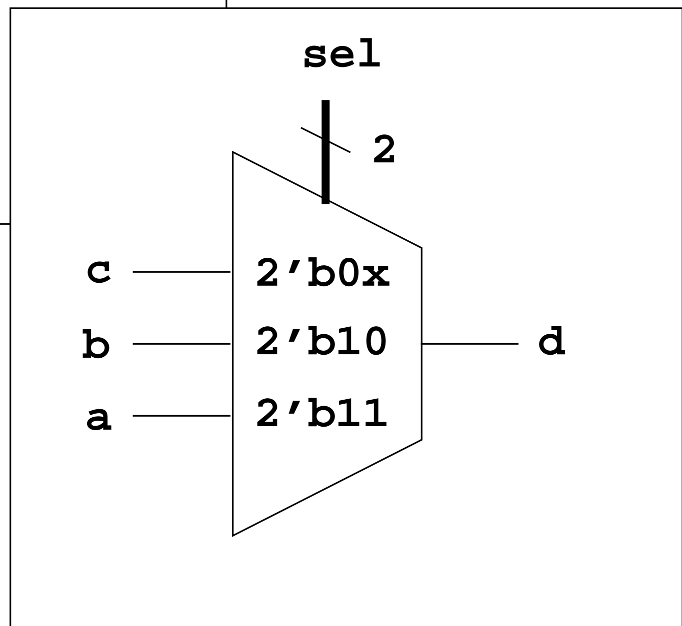
```
// 2. using an if statement
always @ (sl or a or b or c)
   if (sel == 2'b11)
      d = a;
   else if (sel ==2'b10)
      d = b;
   else
      d = c;
```



2.4 Parallel Priority Decoders

2.4.1 Parallel Priority Decoders Using a Synthesis Directive

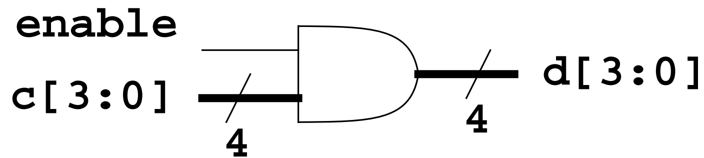
```
// using a synthesis directive
always @ (sl or a or b or c)
   case (sel) // parallel_case
      2'b11: d = a;
      2'b10: d = b;
      default:d = c;
   endcase
```



2.5 Bus Logic, Splitting and Reordering

2.5.1 Bus Enabling

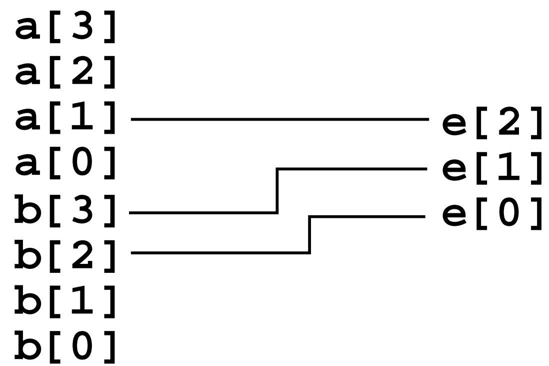
```
// A1. using a wire
wire [3:0] d = ({4{enable}} & c);
// A2. using a reg
reg [3:0] d;
always @ (c or enable)
   d = c & {4{enable}};
```



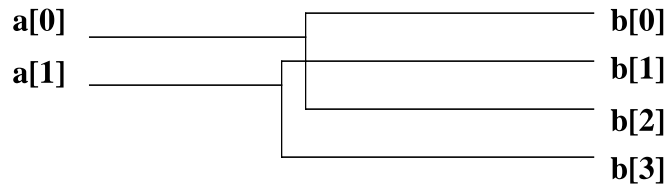
2.5.2 Bus Concatenation

```
// B1. using a wire
wire [2:0] e = {a[1],b[3:2]};
// B2. using a reg
reg [2:0] e;
always @ (a or b)
   e = {a[1],b[3:2]};
```

e = {a[1],b[3:2]}



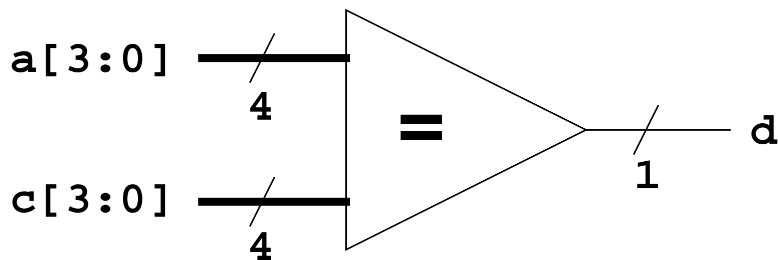
2.5.3 Bus Replication




```
// bus replication
wire [1:0] a;
wire [3:0] b;
assign b = {2{a}};
```

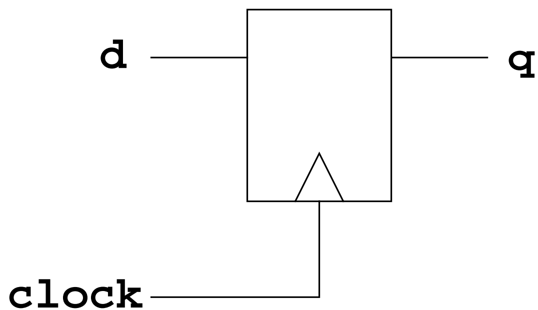
2.6 Comparators

```
// 1. using a wire
wire d;
assign d = (a == c);

// 2. using a reg
reg d;
always @ (a or c)
   d = (a == c);
```



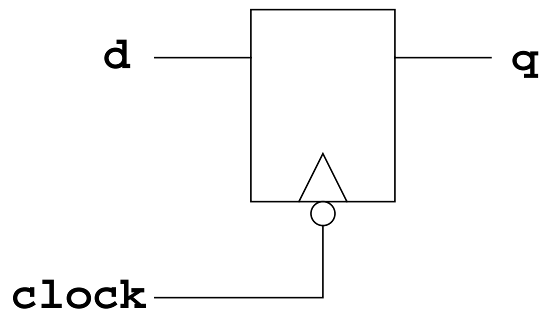
2.7 D Type Flip Flops

```
// 1. positive edge triggered D flip flop
always @ (posedge clock)
   q <= d;
```



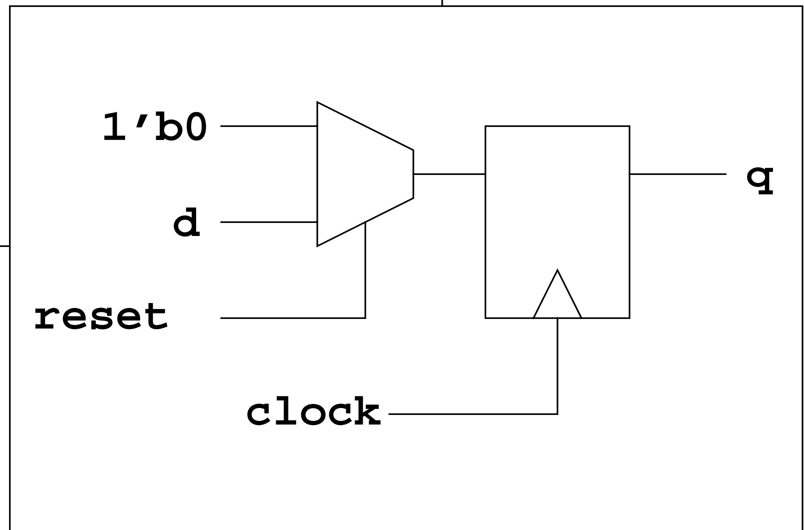
Use non-blocking assignments (<=) in clocked procedures.

```
// 2. negative edge triggered D flip flop
always @ (negedge clock)
   q <= d;
```

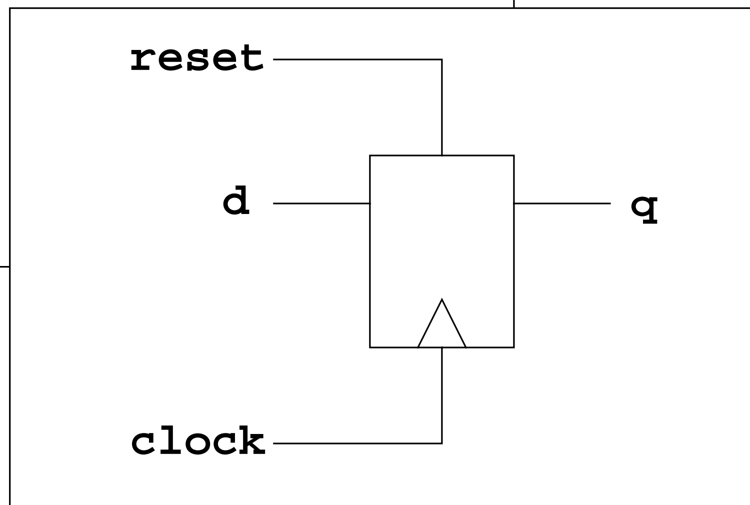


2.8 Resettable D Type Flip Flops

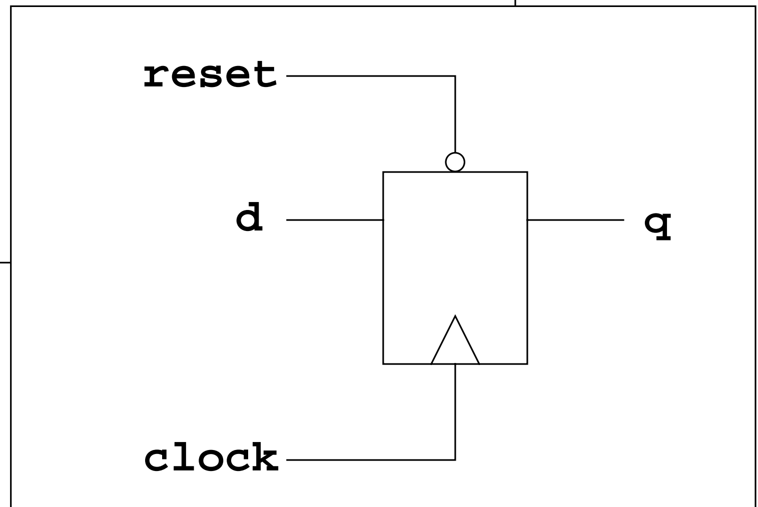
```
// 1. synchronously resettable D flip flop
always @ (posedge clock)
   if (reset)
      q <= 1'b0;
   else
      q <= d;
```




```
// 2. asynchronously resettable D flip flop
// (active high async reset)
always @ (posedge clock or posedge reset)
   if (reset)
      q <= 1'b0;
   else
      q <= d;
```

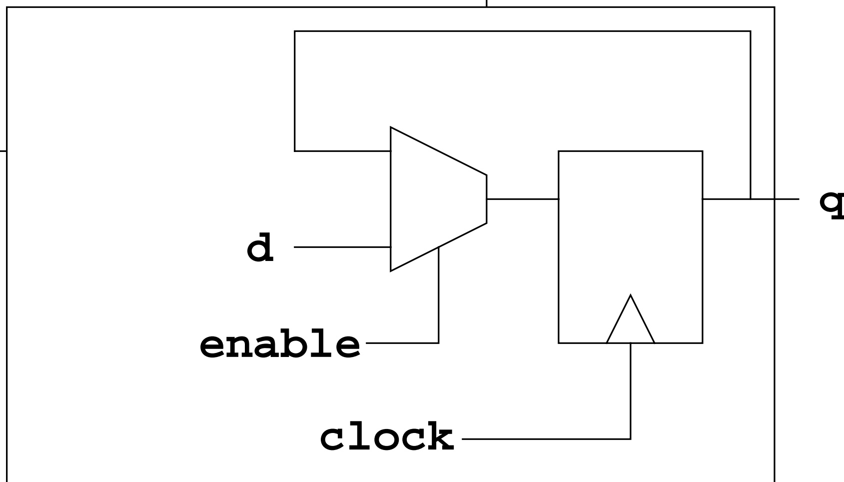


```
// 3. asynchronously resettable D flip flop
// (active low reset)
always @ (posedge clock or negedge reset)
   if (~reset)
      q <= 1'b0;
   else
      q <= d;
```

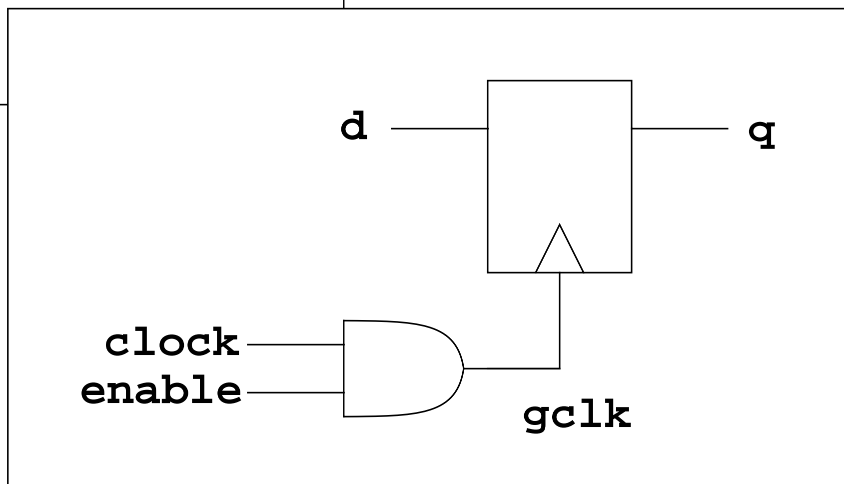


2.9 Data Enabled and Clock Gated Flip Flops

```
// 1. data enabled flip flop
always @ (posedge clock)
   if (enable)
      q <= d;
```



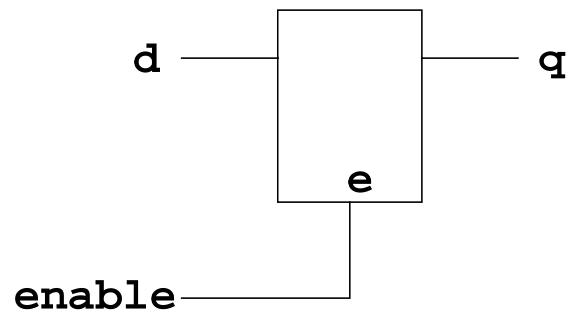

```
// 2. D flip flop with gated clock
wire gclk = (clock && enable);
always @ (posedge gclk)
      q <= d;
```



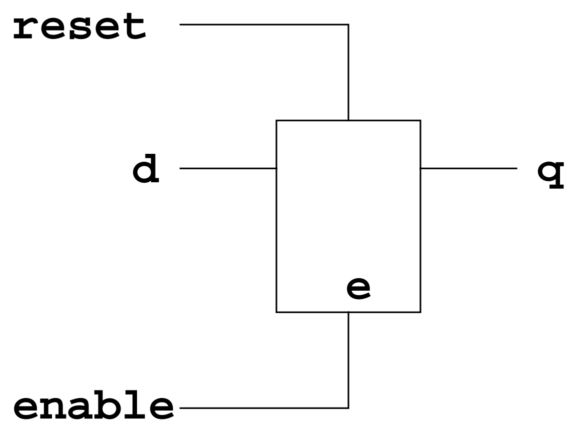
enable signal must be glitch free

2.10 Latches

```
// 1. latch
always @ (enable or d)
   if (enable)
      q = d;
```

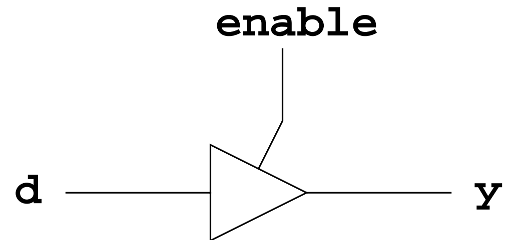



```
// 2.resettable latch
always @ (enable or d or reset)
   if (reset)
      q = 1'b0;
   else if (enable)
      q = d;
```



2.11 Tri-state Drivers

```
// 1. using a reg
reg y;
always @ (d or enable)
   if (enable)
      y = d;
   else
      y = 1'bz;
```



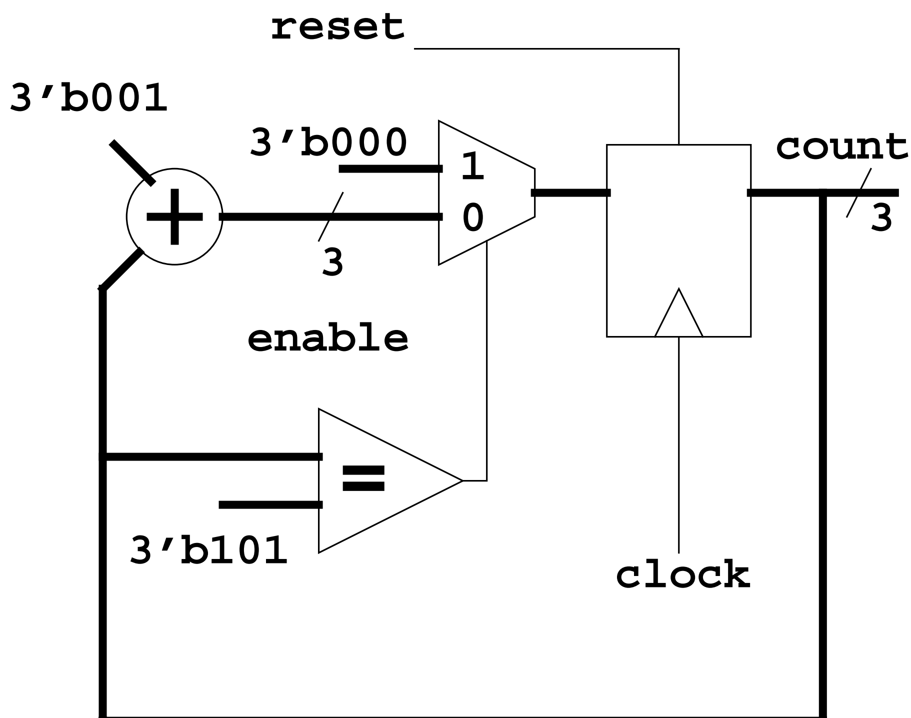

```
// 2. using a wire
wire y;
assign y = enable ? d : 1'bz;
```

```
// 3. using a primitive
bufif1 (y,d,enable);
```

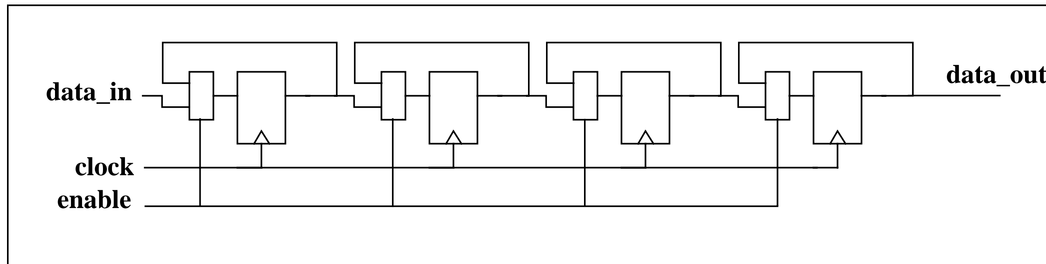
2.12 Counter

3 bit asynchronously resettable counter which counts 0, 1, 2, 3, 4, 5,

```
// 3 bit asynchronously resettable
// partial range counter
always @ (posedge clock or posedge reset)
   if (reset)
      count <= 3'b0;
   else
      if (count == 3'b101)
         count <= 3'b0;
      else
         count <= count + 3'b001;
```

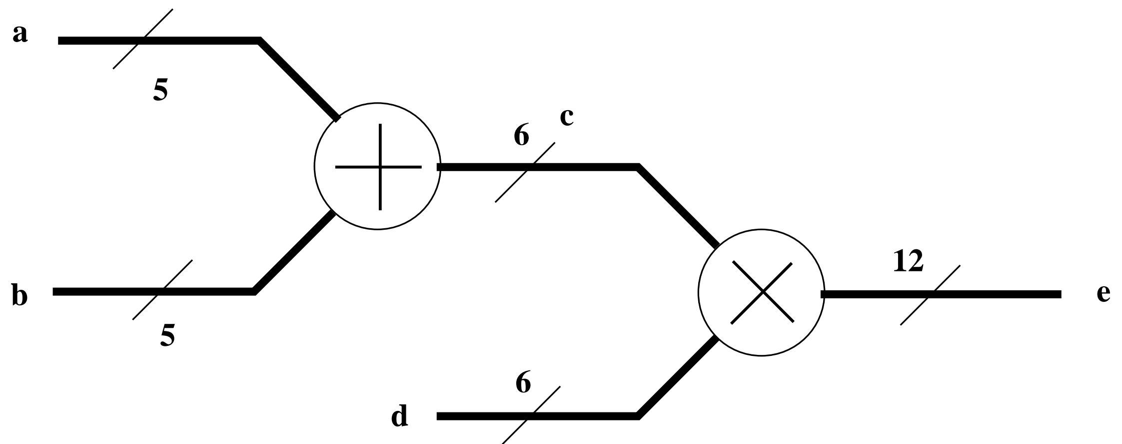


2.13 Enabled Shift Register




```
module enabled_shift_reg (clock,enable,data_in,data_out);
input clock;
input enable;
input [3:0] data_in;
output [3:0] data_out;

reg [3:0] data_out;
reg [3:0] shift_reg_1;
reg [3:0] shift_reg_2;
reg [3:0] shift_reg_2;


always @ (posedge clock)
   if (enable)
      begin
      shift_reg_1 <= data_in;
      shift_reg_2 <= shift_reg_1;
      shift_reg_3 <= shift_reg_2;
      data_out <= shift_reg_3;
      end

endmodule
```

2.14 Unsigned Adders and Multipliers



Note that the * and + and - signs give you unsigned arithmetic. If you need signed arithmetic, you may need special instaces recognizable to the synthesis tool.

- Adding two five bit numbers gives a siz bit result (the extra bit is the carry out bit).
- The multiplication of two number means that the output is twice the width of the inputs.

```
wire [5:0]  c = a + b;
wire [11:0] e = c * d;
```

3.0 Coding Guidelines

These coding guidelines assume that you are able to write correct synthesizeable code. You can always check the synthesizeablilty of your code by parsing it using the synthesis tool.

- Use non-blocking assignments (<=) in clocked procedures. Don't use blocking assignments (=).

```
always @ (posedge clock)
   q <= d;
```

- Use blocking assignments (=) in combinational procedures:

```
always @ (a or b or sl)
   if (sl)
      d = a;
   else
      d = b;
```

- Make sure that the event lists are complete

```
always @ (a or b) // this event list is missing signal sl
   if (sl)
      d = a;
   else
      d = b;
```

- Take care of indentation. Develop your own identation guidelines and stick to them. Make sure others can read them. It helps readability and debugging greatly if it is done properly.
- Comment code properly. The theory about good commenting is that you should be able to remove all functional code and the comments remaining should almost document the block you are designing.

```
// example of bad comments
// add a and b together
always @ (a or b)
   c = a + b;

// Good commenting
// 8 bit unsigned adder for data signals 'a' and 'b'
// output is sent to UART2
always @ (a or b)
   c = a + b;
```

- Always completely specify literals.

```
always @ (c)
   if (c == 4'b0101)
      a = 2'bxx;
   else
      a = 2'b10;
```

- Use named port mapping when instantiating.

```
state_machine u1 (
   .sm_in       (in1),
   .sm_clock    (clk),
   .reset       (reset),
   .sm_out      (data_mux)
   );
```

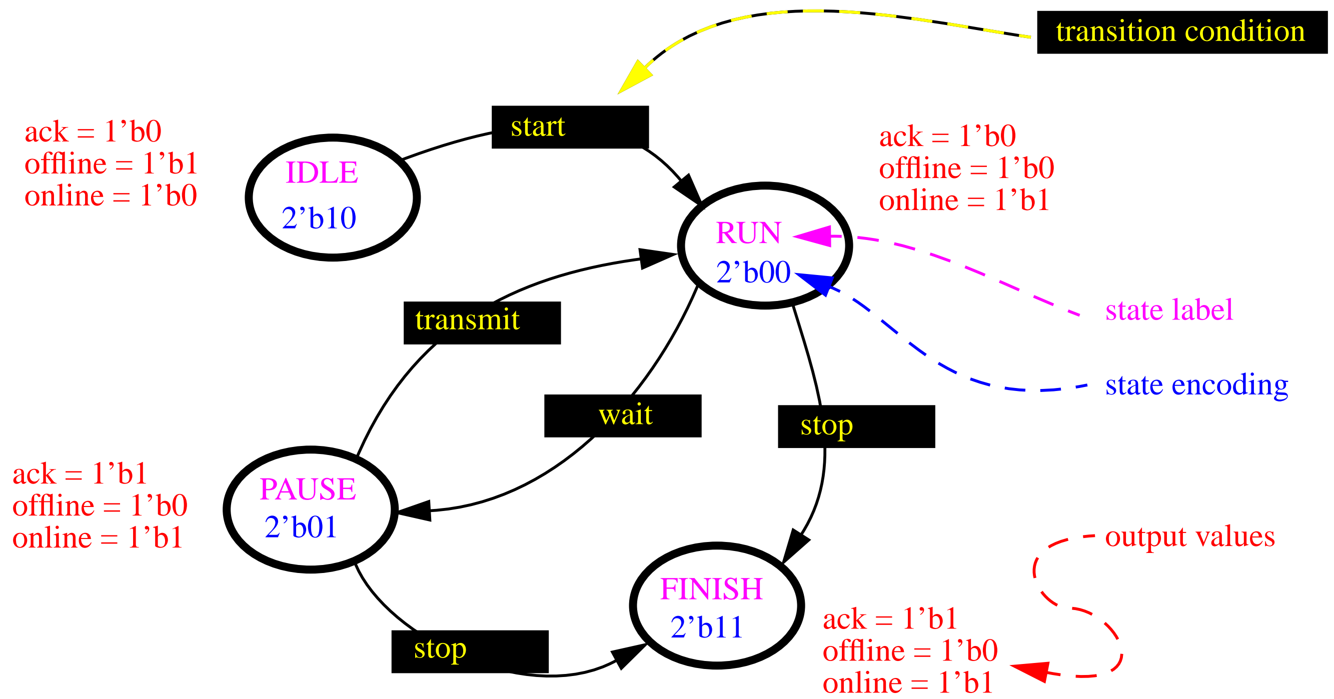
- Don't make the code any more complicated than it needs to be. Your priorities should be correctness, then readability and finally code efficiency.

4.0 State Machine Guidelines

4.1 Guidelines

- Draw a state diagram.
- Label the states.
- Allocate state encoding.
- Label the transition conditions.
- Label the output values.
- Use parameters for the state variables.
- Use two procedures (one clocked for the state register and one combinational for the next state logic and the output decode logic).
- Use a case statement in the combinational procedure.
- Have a reset strategy (asynchronous or synchronous).
- Use default assignments and then corner cases.
- Keep state machine code separate from other code (i.e. don’t mix other logic in with the state machine clocked and combinational procedures).

4.2 State Diagram



4.3 State Machine Verilog Code

```
module state_machine (clock,reset,start,transmit,wait,stop,ack,offline,online);
// parameter declarations
parameter pIDLE   = 2'b10; // state labels and state encoding
parameter pRUN    = 2'b00;
parameter pPAUSE  = 2'b01;
parameter pFINISH = 2'b11;

// IO declaration section
input clock;
input reset;
input start, transmit, wait, stop;
output ack, offline, online;
// interal variables declaration section
reg [1:0] state, next_state;
reg ack, offline, online;

// clocked procedure with synchronous reset
always @ (posedge clock)
if (reset) // reset strategy
   state <= pIDLE;
else
   state <= next_state;

// combinational procedure with case statement and output logic
always @ (start or transmit or stop or wait or state)
   begin
   next_state = state; // default assignment to state and output variables
   ack     = 1'b0;
   offline = 1'b0;
   online  = 1'b1;
   case (state)
      pIDLE:
         begin
         offline = 1'b1;
         online  = 1'b0;
         if (start)
            next_state = pRUN;
         end
      pRUN:
         begin
         if (wait)
            next_state = pPAUSE;
         if (stop)
            next_state = pFINISH; // this has priority over the wait transition
         end
      pPAUSE:
         begin
         ack     = 1'b1;
         if (transmit)
            next_state = pRUN;
         if (stop)
            next_state = pFINISH;
         end
      pFINISH:
         ack = 1'b1;
   endcase
   end

endmodule
```