University of Sheffield

HDL Support

MSc Project - RVFPGA

Dr Tiantai Deng
Lecturer in Electronics and Digital System
t.deng@sheffield.ac.uk
07746660187

A WORLD TOP 100 UNIVERSITY

Content

- Basic Syntax for Verilog HDL & Logic Modelling using Verilog HDL
- Simulation using a Testbench
- Building Blocks and Ips
- Design Choices, Critical Path Mitigation
- Evaluation

Basic Syntax for Verilog HDL

Design hierarchy

- Module - a collection of statements with a defined interface
  - In VHDL entity defines the interface
  - architecture the contents
  - Component declares a black box (module interface so it can be instantiated)
- Instantiation - create one or more instances of a sub module within another module
- Libraries – VHDL supports packages which are a collection of modules, types and constants as does system-Verilog.

What's a Module

- Module Name
  - Port list with declarations of ports
  - Parameters to allow customisation (optional)
- Declaration of local wires/signals/variables
- Instantiation of lower level modules
- Behavioural statements in blocks
- Data flow continuous assignments
- Tasks and functions (typically for simulation)

Port Declarations

- These are its interface to the next level up / outside world (top)
- Have direction input, output, inout
- Can be single bit or bit vectors, etc

- Got to avoid: don't use inout just because you want to read a value that should be an output eg the count value of a counter' create a local signal/wire for internal use and assign from it to create the output.

```
//dataflow level
module Comparator(input [3:0] A,
                  input [3:0] B,
                  output Out);
    assign Out = &((A~^B));
endmodule
```

Example of module declaration

Assignment statements

- Z <= A + B; //non-blocking assignment (within process)
- Z = A + B; //blocking assignment (within process)
- Z <= (sel)? A : B; //conditional assignment (within process)
- assign Z = A + B; //standalone continuous assignment (non-blocking)

NOTE: There is no statement like assign Z <= A+B (syntax quirk)

Comments

- Just like what we do in C/C++!
- //
- /* */

Always remember to comment your code while you do the design! Or you wouldn't know your design after a week!

Modules & ports

- VERILOG

```
module test (
                output E,
                input A,B,C
               );
    assign  E = (A & B) | C;
Endmodule
```

input/output can be either be a net (wire) or REG
inout is always a net.

- VHDL

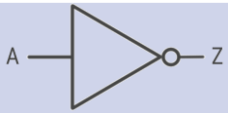
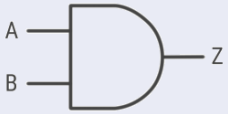
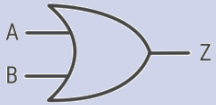
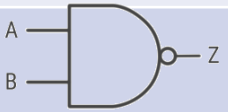
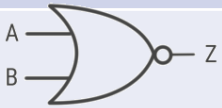
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity test is
  port (
          A,B,C :  in    STD_LOGIC;
          E         : out STD_LOGIC
       );
end test;

architecture beh of test is
begin
    E <= (A and B) or C;
end beh;
```

Combinational & Sequential Logic

Combinatorial vs Sequential

- Combinatorial:
  - AND, OR, NOT, XOR, NAND, etc….
  - DOESN’T STORE ‘state’ and doesn’t need CLOCKING
- Sequential:
  - DFF, registers, memories
  - Changes ‘state’ on an edge of a CLOCK
  - Stores / retains data over time

Bit-wise operations

Operator	Verilog assign	Structural Verilog	VHDL	Gate Symbol
NOT	Z = ~ A	not (Z,A)	Z <= not A	
AND	Z = A & B	and (Z,A,B)	Z <= A and B	
OR (inclusive)	Z = A | B	or (Z,A,B)	Z <= A or B	
XOR (exclusive)	Z = A ^ B	xor (Z,A,B)	Z <= A xor B	
NAND	Z = ~ A & B	nand (Z,A,B)	Z <= A nand B	
NOR	Z = ~ A | B	nor (Z,A,B)	Z <= A nor B	
XNOR	Z = ~ A ^ B	xnor (Z,A,B)	Z <= A xnor B	

Modelling electrical rules of logic

- **Verilog** "wire"

0	logic 0 (low)
1	logic 1 (high)
Z	high impedance
X	unknown/impossible

- **VHDL** "std_logic"

0	logic 0 (low)
1	logic 1 (high)
Z	high impedance
X	unknown/impossible
–	don't care
U	un-initialised
L H	weak low or high
W	weak indeterminate

Logical & Relational Operators

Operator	Verilog	VHDL	Example (Verilog)
NOT	!	not	busy = !ready;
AND	&&	and	congestion = traffic && slowMoving;
OR	||	or	crossRoad = redLight || !traffic;
CONDITIONAL	? :	when else	hexValue = (ascii>64) ? ascii-'A' : ascii-'0';
EQUALITY	==	=	Xmas = (dayOfYear==360);
INEQUALITY	!=	/=	counting = (countValue != maxValue);
GREATER THAN	>	>	
GREATER OR EQUAL	>=	>=	
LESS THAN	<	<	
LESS OR EQUAL	<=	<=	

NOTE: <= is also SIGNAL assignment operator in VHDL

Arithmetic Operators

Operator	Verilog	VHDL	Example (Verilog)
UNARY NEGATION	-	-	Z = -A
ADDITION	+	+	Z = A + B
SUBTRACTION	-	-	Z = A - B
MULTIPLICATION	*	*	Z = A * B
DIVISION	/	/	Z = A / B
MODULUS	%	mod rem	Z = A % B
LOGICAL SHIFT	<< >>	sll srl	Z = A << 3
ARITH SHIFT	<<< >>>	sla sra	Z = sA >> 7
EXPONENTIATION	** (since 2001)	**	Z = 2 ** N
ABSOLUTE VALUE		abs	Z = (A<0) ? –A : A;

University of Sheffield Basic Types

- VHDL
  - boolean
  - std_logic
  - std_logic_vector
  - integer / unsigned / signed
  - ….

- Verilog
  - wire
  - reg

NOTE: **reg** means simulator <u>may</u> remember its value. It dosnt mean define a REGISTER despite its name!

- System Verilog
  - bit — bit[7:0] abyte
  - byte — 8-bit **signed**
  - short int — 16-bit signed
  - int — 32-bit signed
  - long int — 64-bit signed

  TWO STATE 0 or 1

  - logic (or reg) — logic[7:0] abyte
  - integer — 32-bit signed

  FOUR STATE 0 1 X Z

  - time — 64-bit time
  - shortreal — aka float
  - real / realtime — aka double

Verilog "REG" !!Don't get confused!!

- In Verilog **reg** means a declaration of a variable that may potentially hold a value.
- This is in contrast to **wire** which canot hold a value.
- **reg** on its own does not create a digital register. You'd need to use always @ posedge(clk) value<=whatever; to do that.
- It is similar to **int** or **real** just declares a bit type
- The good news is in systemVerilog they both simply become **logic** there is no longer any need to distinguish between reg and wire.

But not so good. When you lose the control over your program, your code maybe hard to crack.

Other Types

- VHDL, Verilog and SystemVerilog also have:
  - Vectors - multiple bits
  - Integer - for integer arithmetic
  - Real - for floating point arithmetic
  - Time - exclusively used in simulation
  - Arrays - single or multidimensional
  - Strings - great for test benches / debugging

Busses

- Collection of individual signals or n-bit signals
- Easy to refer to by name
- Make from various individual signals or parts of other buses
- Extract individual bit or bits
- HDLs make it easy to do all this….

Vectors (Collections of bits)

- Bit strings / bit arrays / logic vectors
  - VERILOG: reg sum[15:0], carry;
  - VHDL: signal sum: std_logic_vector(15 downto 0);
    signal carry : std_logic;
- Why [big:small] or “downto”
  - Because we humans like to see the MSB at the left hand side.
- You can also have arrays of bit strings too…example 4 bytes:
  - VERILOG: reg block[3:0][7:0];
  - VHDL: signal block : array (0 to 3) of std_logic_vector(7 downto 0);

Defining Literals

- VERILOG: reg [7:0] poly = 8’h1f; // 8 hexadecimal bits
  - or use ‘d’ decimal, ‘h’ hex, ‘o’ octal, ‘b’ binary
  - You can use underscore to make more readable eg 32’h1234_c0de

- VHDL: signal poly : std_logic_vector(15 downto 0) := X”2c1f”;
  - For bit strings use double quotes and single bits single quotes
  - Use X for hexadecimal, “0101” automatically binary string, unquoted is decimal
  - ‘1’ or ‘0’ for single bit (std_logic)

You’ll see plenty more example of this in the lab sessions.

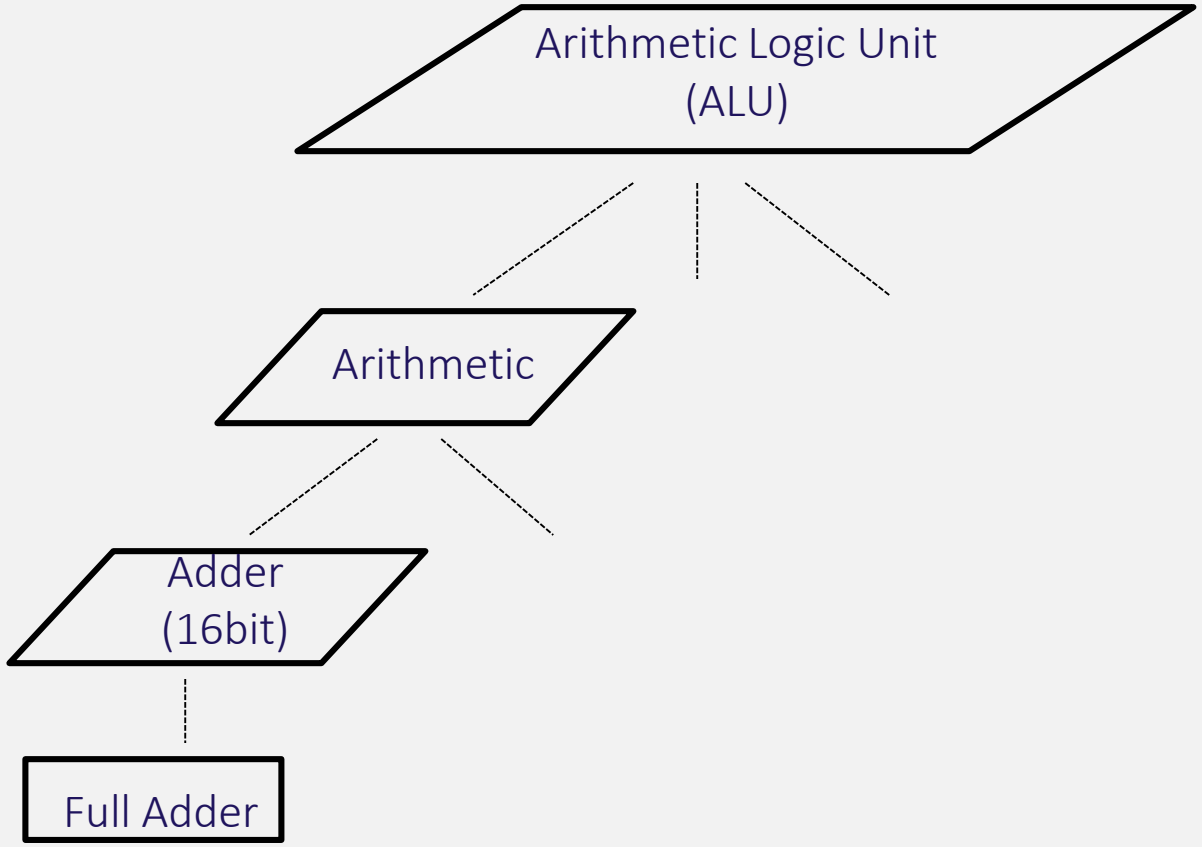
Collection Operators

Operator	Verilog	VHDL	Example (Verilog)
PRIORITY parenthesis	( )	( )	(A+B)*(C+D)
BIT-SELECT	[ ]	( )	signbit = A[31];
PART-SELECT	[:]	(to) (downto)	msb = A[15:8];
CONCATENATION	{ }	&	byteswap = { A[7:0], A[15:8] };
Operations we commonly used in Verilog, there are other operations in Verilog, which doesn't use a lot. We can learn how to use those while encountering one.			

Structural, Data flow & Behavioural Models

Modelling of Logic

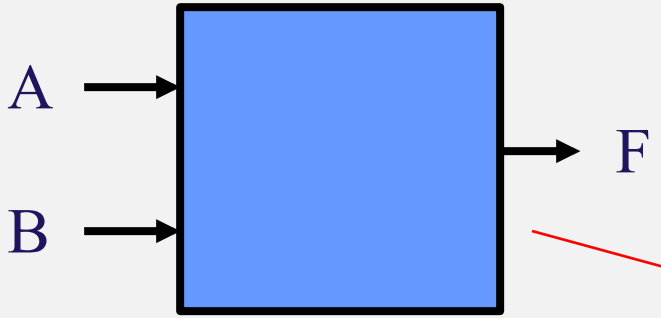
- Combinatorial logic
  - Boolean expressions
  - AND, OR, NOT, XOR gates etc…
- Sequential logic
  - Flip-flops
  - Registers
- Hierarchical Design
  - components
  - abstraction
  - design reuse



Verilog Modules

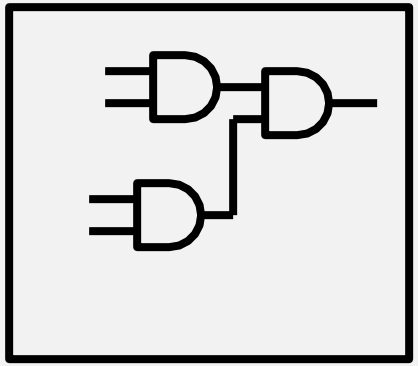
The basic building block in Verilog is called a module. A digital system can be described in an HDL by a set of these modules / entities.

Port declarations detail the interface of a module to other modules or the outside world.



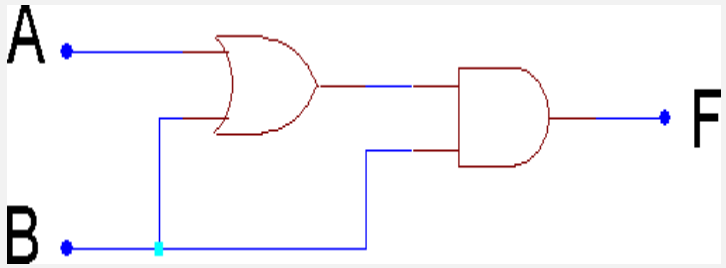

```
module eg1 (A, B, F);
...
...
...
endmodule
```

The module body describes the function of the model and the relationship between the ports.



Structural Description - Verilog

The structure of a circuit can be described in Verilog using predefined gate primitives. These include **not**, **and**, **nand**, **or**, **nor**, **xor**, **xnor**.



Example

Text after // is interpreted as a comment. Keywords are in bold.

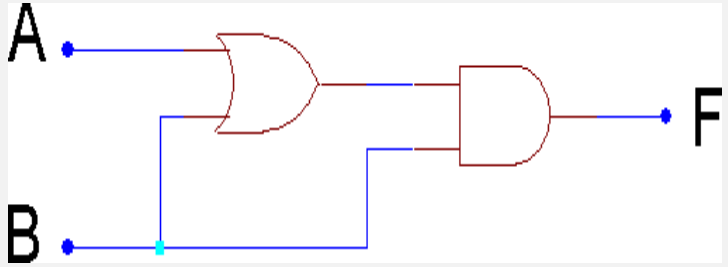
```
module example1 (output F, input A, B);  // module name and ports
wire C;                                  // declare internal connection
or g1(C, A, B);                          // or gate with optional name g1
and g2(F, C, B);                         // and gate with optional name g2
endmodule                                // gate outputs come first in list
```

Data flow Description - Verilog

A behavioural Verilog model can be used to describe the functionality of a circuit, independently of a chosen technology.

Logic expressions can be described behaviourally in Verilog using predefined logical operators:

~	Bitwise NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~ ^	Bitwise XNOR



Example

```
module example2 (output F, input A, B);
assign F = (A | B) & B;        // continuous assignment
endmodule
```

Processes

Behavioral description

- VERILOG

```
module decoder_2x4_df_beh(
      output [0:3] D,
      input A,B, enable
      );
      always @ (A, B, enable) begin
        D[0] <= !((!A) & (!B) & (!enable));
        D[1] <= !((!A) & B & (!enable));
        D[2] <= !(A & (!B) & (!enable));
        D[3] <= !(A & B & (!enable));
        end;
     endmodule
```

Behavioural description declared with keyword **always**, followed by an optional control expression(sensitivity list) and a **begin** ... **end** block of procedural assignment statement.

Continuous assignment (**assign**)

Procedural assignment (non blocking **<=** (concurrent) or blocking **=** (sequential)

You can have as many process blocks in an architecture as you wish but any signal/output should only be defined in ONCE.

Procedural if…then…else…elsif…endif

```
always @ …                    // verilog
begin
  if ( condition )   single_statement;

  if (condition ) begin
     multiple_statements;
     multiple_statements;
  end else begin
     multiple_statements;
     multiple_statements;
  end

  if (condition 1 )
     single_statement;
  else if (condition 2 ) begin
     multiple_statements;
     multiple_statements;
  end else
     single_statement;
end
```

This form of cascaded **"if"** is a priority encoder

```
process (…)      -- VHDL
begin
  if condition then
     multiple_statements;
  end if;

  if condition then
     multiple_statements;
  else
     multiple_statements;
  end if;

  if condition1 then
     multiple_statements;
  elsif condition2 then
     multiple_statements;
  else
     multiple_statements;
  end if;
end process;
```

Procedural case statement

```
always @ (*)              // verilog
case ( selector )
   1 :  single_statement;
   2 : begin
         multiple_statements;
         multiple_statements;
       end
  2:  single_statement;
  labelWhatever: single_statement;

  // can use formatted numbers
  //  with wildcards too
  4'b001?: single_statement;

 default: single_statement;
endcase
```

- Case can be more readable than long cascade of 'if's.
- Also doesn't imply priority so arguably often more appropriate description
- Great with enumerated types for describing state machines
- Ensure case statements are fully complete otherwise infers latches

```
process (All)    -- vhdl
case selector is
  when 0  =>  multiple_statements;
  when 1  =>  multiple_statements;
  when 2 to 5 =>   multiple_statements;
  when 6|8|10 => multiple_statements;
  when label_whatever => statemements;
  when others => multiple_statements;
end case;

// if you don't have any statement for
// an already completed case then
//  when others => null; should be used

// works great with enum types
```

Defining Constants

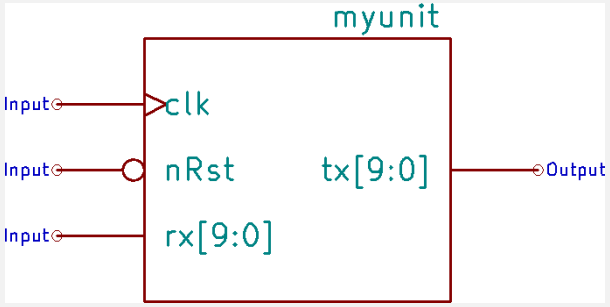
- VHDL
  - **constant** correct_answer **:** integer range 0 to 7 **:=** 3;
  - **constant** space_char **:** std_logic_vector(7 downto 0) **:=** X”20”;
  - **type** atable **is** array (0 to 255) **of** std_logic_vector(7 downto 0) **:=**
    { X”01”, X”23, X”46”, ….. }

- Verilog
  - **`define** SOME_CONSANT_MACRO 1 // normally at near start of file
  - # (**parameter** WIDTH = 5) // within module declaration
  - **localparam** SOME_CONSTANT_LOCAL = 2; // within module body

Using submodules

- Called ‘Instantiation’
- Gives usage a unique name in the hierarchy
- Connects the modules ports to signals within parent module
- Ports can either be connected by name explicitly or in order of declaration
- All inputs must be connected, outputs can be left open / unconnected
- Bit vector port widths MUST match
- Verilog Gotcha: avoid giving a module/instance/port the SAME name

Instantce



Your own module

VERILOG

```
module top(
  input        clk,
  input        rst_n,
  input        enable,
  input  [9:0]  data_rx_1,
  input  [9:0]  data_rx_2,
  output [9:0] data_tx
  );

  wire [9:0] tx1, [9:0] tx2;
  myunit unit_idle (clk,rst_n,data_rx_1, tx1);
  myunit unit_active (.nRst(rst_n), .clk(clk),
                      .rx(data_rx_2), .tx(tx2) );
  assign data_tx  = (enable)  ? tx2 : tx1;

endmodule
```

VHDL

```
entity top is
   port (
            clk, rst_n, enable       : in    std_logic;
            data_rx_1,data_rx_2 : in    std_logic_vector(9 downto 0);
            data_rx                  : out std_logic_vector(9 downto 0)
         );
end top;

architecture beh of top is

   component myunit is
      port (
            clk, nRst  : in   std_logic;
            rx         : in    std_logic_vector(9 downto 0);
            tx         : out std_logic_vector(9 downto 0)
         );
   end component;

   signal tx1, tx2 : std_logic_vector(9 downto 0);
begin
   unit_idle:    myunit port map (clk, rst_n, data_rx_1, tx1);
   unit_active: myunit port map (nRst=>rst_n, clk=>clk,
                                 rx=>data_rx_2, tx=>tx2);
   data_rx <= tx2 when (enable='1') else tx1;

end beh;
```

Parameterization

```
// parameter available since Verilog-2001
module ram(clk,address,dataIn,dataOut,wr);
  parameter DATA_WIDTH = 8;
  parameter ADDR_WIDTH= 7;
  input clk;
  input [ADDR_WIDTH-1:0] address;
  input [DATA_WIDTH-1:0] dataIn;
  output [DATA_WIDTH-1:0] dataOut;
  …
endmodule


// example usage
ram_inst #(.DATA_WIDTH(16), .ADDR_WIDTH(8) )
        ram(clk,address,din,dout,wr);
```

```
-- vhdl has generic
entity ram
  generic ( DATA_WIDTH : positive := 8;
            ADDR_WIDTH : positive :=7 );
  port (
    clk : in std_logic;
    address: in std_logic_vector(ADDR_WIDTH-1 downto 0);
    dataIn : in std_logic_vector(DATA_WIDTH-1 downto 0);
    dataOut : out std_logic_vector(DATA_WIDTH-1 downto 0);
    wr : in std_logic )
end ram;


-- example usage
ram_inst: ram
   generic map ( data_width=>16, addr_width=>8)
   port map ( clk, address, din, dout, wr);
```

Inferring storage elements

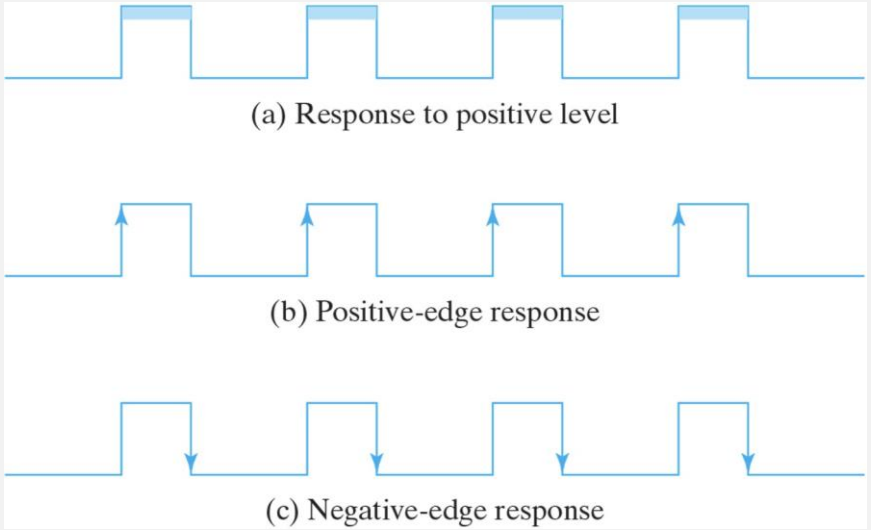
- This is sequential logic so we define which clock edge it should act upon. With the exception of special "DDR" I/O cells we are inferring flip-flops which can only act on one edge.

- VERILOG:

```
   always @ (posedge someclock)
// always @ (negedge someclock)
```

- VHDL:

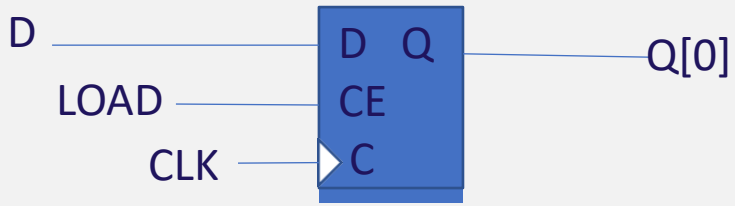
```
process (someclk) begin
        if rising_edge() then …
--      if falling_edge() then …
```



Different Response

A little question: Why DDR is **Double Rate?**

Registers / DFFs



CLK	LOAD	D	Q
0	any	any	Same
1	any	any	Same
_/---	0	any	Same
_/---	1	0/1	D

C = clock, CE = clock enable
This is an fpga 'thing' due to the global nature of the clock network

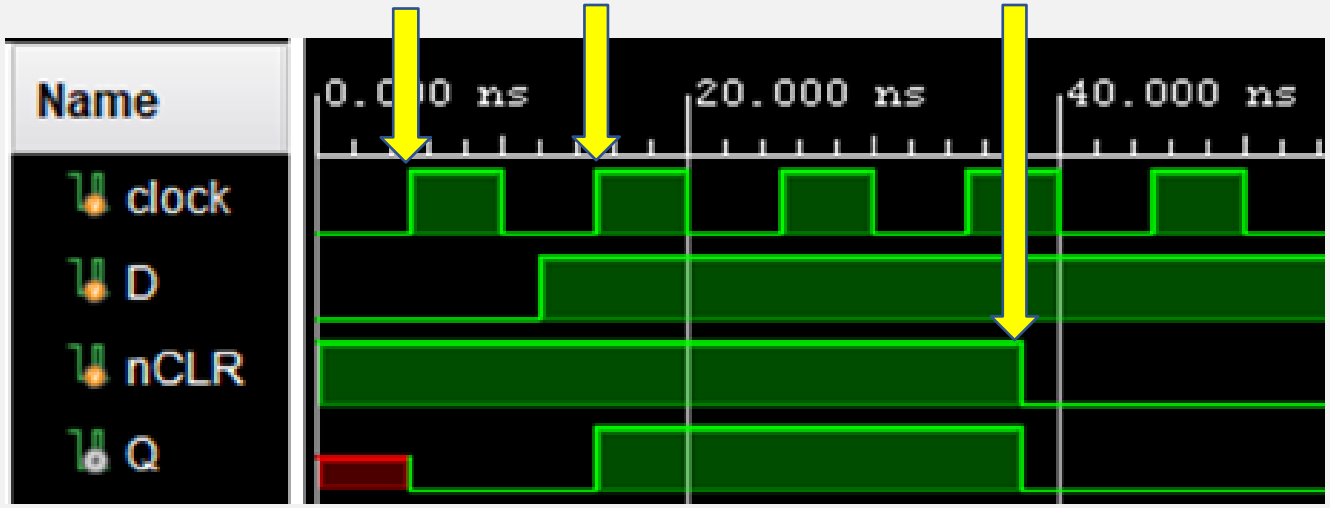
```
// Verilog

always @ (posedge(clk)) begin
    if (load) begin
        Q<=D;
    end
end
```

```
-- VHDL
process (clk) begin
  if rising_edge(clk) then
    if load='1' then
        Q<=D;
    end if;
  end if;
end process;
```

Asynchronous set/clear DFF

Note the always and process sensitivity list differences. It's all too easy to get wrong!

Generally the use of async is deprecated by FPGA manufacturers.



Waveform of module dffr

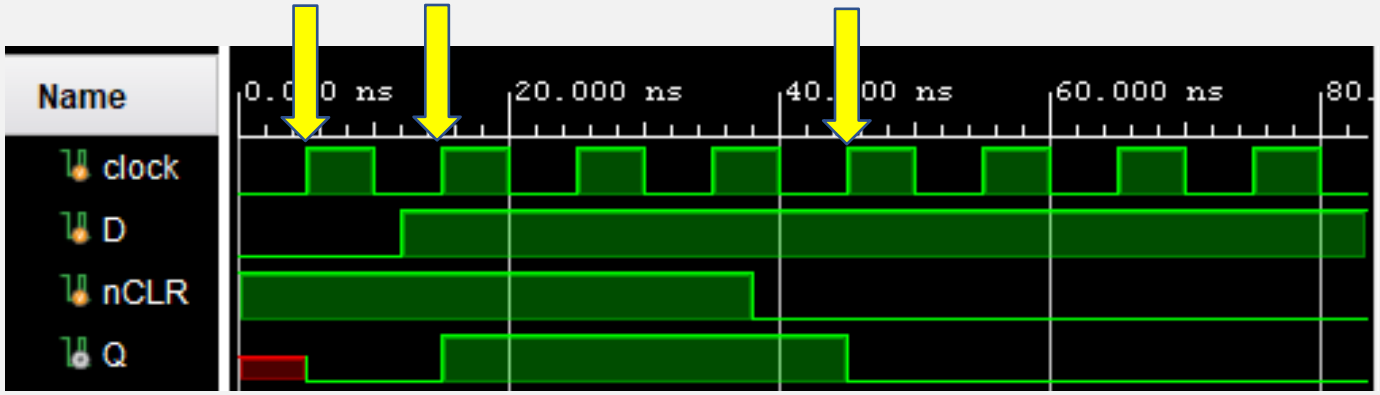
```
// Verilog
module dffr(output reg Q, input D, nCLR, clk);
    always @ (negedge nCLR or posedge clk)
        begin
            if (!nCLR) Q <= 1'b0; else Q <= D;
        end
endmodule
```

```
entity dffr is                                    -- VHDL
port( Q : out std_logic; D,nCLR,clk : in std_logic );
end dffr;
architecture beh of dffr is
begin
    process (clk, nCLR) begin
        if  nCLR = '0'  then
            Q <= '0';
        elsif  rising_edge(clk) then
            Q <= D;
        end if;
    end process;
end beh;
```

University of Sheffield

Synchronous set/clear DFF

- You'll need to compare this with the async version on the previous slide to spot the differences.
- They are subtle which is often a cause of design errors.



Waveform of module dffc

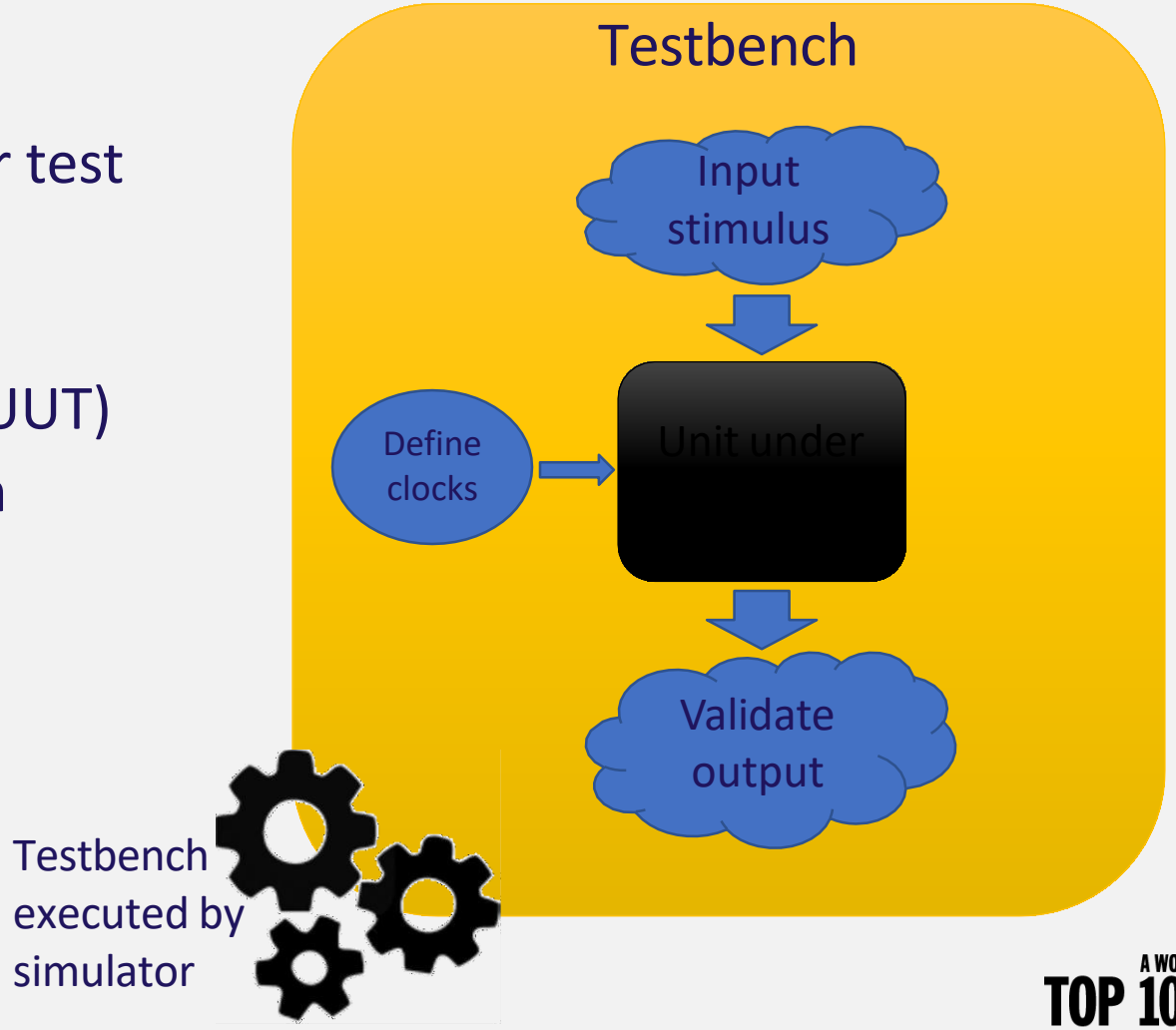
```
// Verilog
module dffc(output reg Q, input D, nCLR, clk);
    always @ (posedge clk)
        begin
            if (!nCLR) Q <= 1'b0; else Q <= D;
        end
endmodule
```

```
entity dffc is                                    -- VHDL
port( Q : out std_logic; D,nCLR,clk : in std_logic );
end dffc;
architecture beh of dffc is
begin
   process (clk)  begin
      if  rising_edge(clk) then
          if  nCLR = '0'  then
              Q <= '0';
          else
              Q <= D;
          end if;
      end if;
   end process;
end beh;
```

Simulation using a Testbench

Test benches / test fixtures

- Tasks
  - Declare signals/variable to be used for test
  - Instantiate unit under test
  - Define clock(s) if any
  - Generate stimulus pattern (inputs to UUT)
  - Validate (compare output of UUT with expected values)
  - Report results



Scope of testing

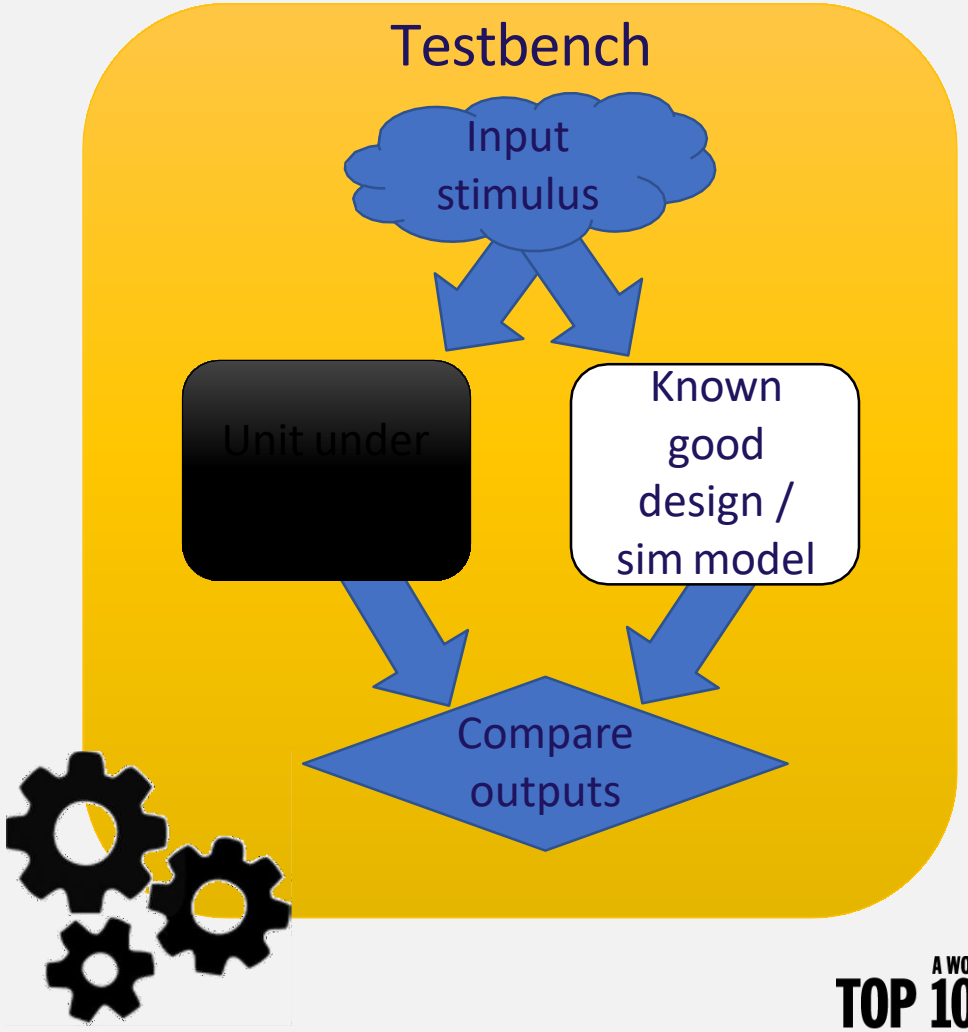
- Module
  - It is most expedient to test individual modules against their expected behaviour
  - Its is usual to have one or more testbench per module in your design
- Sub-System
  - Often a collection of modules can be tested together as a sub-system
  - Typically for debugging the design and to verify the required performance
- Whole System
  - Such simulations can be very slow to execute so normally reserved for ASIC flows – for an FPGA its quicker to simply program the real device

Level of testing

- Behavioural
  - Tests if the register transfer level statements are correct on a cycle per cycle basis
  - Doesn't have any path / gate timing information
  - Would 'pass-the-test' even with an absurdly fast clock
  - Very quick to simulate
- Post implementation
  - Incorporates (back annotates) path & gate timing information to accurately model the expected performance of the device
  - Models are relatively crude but effective
  - Two timing corners best case and worst case
  - Confirms setup and hold times are not violated
  - Needs information on external circuit capacitive loading
  - User constraint 'guestimates' can lead to simulation and device mismatches

Self-checking test benches

- Generate a comprehensive set of test cases
- Compare the output of the UUT against either an alternative design or simulation model
- The reference design doesn't need to meet timing/area constraints as is only a simulation so often can be simply coded
- Finally generate a report to indicate any test failures



Test coverage

- Ideally should test every decision / path through a design for correctness - often too time consuming
- Simulators have ‘force’ commands to allow system to be put into specific state before/during a test this can dramatically reduce testing time.
- Generating an economic set of test vectors for ASIC/FPGA testing is a job in its own right (verification engineer)

Modelling other parts of system

- For a system which needs to respond to external hardware testing often requires this is modelled too.
- For example you may need to model an SPI or I2C sensor device and an actuator controlled by the UUT to verify that it works correctly.
- The HDL can do this too – you can write your own models for parts of a system (not synthesisable) which will simulate some missing hardware.
- This allows the use of the super-set of the language that can be simulated as well as synthesised.

Simple Testbench

Your First Verilog Test Fixture

```
`timescale 1ns / 1ps
```

"#" time unit / simulation resolution

```
module tb_simple( );
```

Module with no ports is a testbench

```
   reg A,B,C;
   wire out;
```

Declare signals for input as reg and outputs as wire

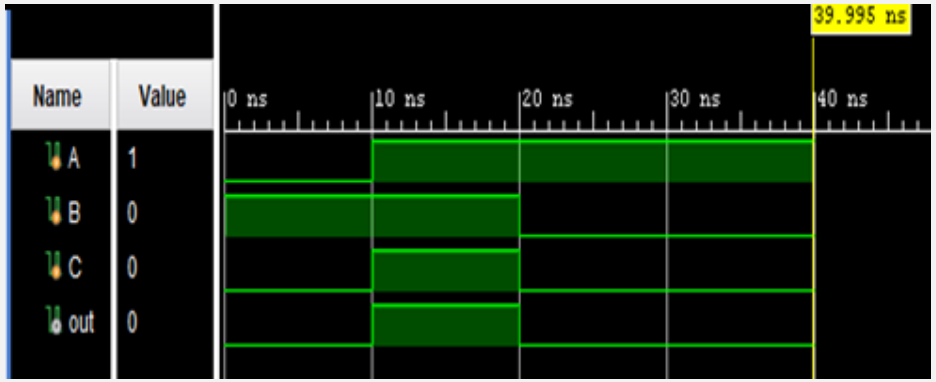
```
   simple uut(out,  A,B,C);
```

Instantiate unit under test

```
   initial begin
            A=1'b0; B=1'b1; C=1'b0;
      #10 A=1'b1; B=1'b1; C=1'b1;
      #10 A=1'b1; B=1'b0; C=1'b0;
      #20 $finish;
   end

endmodule
```

'initial' block containing the desired stimulus and $finish at its end

```
module simple(
   output q,
   input a,b,c
   );
   wire e;
   and a1(e,a,b);
   or   o1(q,e,c);
endmodule
```



Test bench outline

Verilog

```
`timescale 1ns / 1ps

module tb_whatever ();

   ...

endmodule
```

VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity tb_whatever is
end tb_whatever;

architecture Behavioral of tb_whatever is

   ...

begin

   ...

end Behavioral;
```

University of Sheffield

Unit Under Test (UUT)

```
`timescale 1ns / 1ps

module tb_whatever ();

   reg   list_of_inputs;
   wire list_of_outputs;

   test_module_name uut(list_of_ins_and_outs)

   …

endmodule
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity tb_whatever is
end tb_whatever;

architecture Behavioral of tb_whatever is

   component test_module_name
      port ( list_of_out_ports : out std_logic;
             list_of_in_ports    : in   std_logic);
   end component;

   signal list_of_ins_and_outs : std_logic;

begin

  uut:   test_module_name
           port map (list_of_ins_and_outs);

  …

end Behavioral;
```

TOP 100 UNIVERSITY

University of Sheffield

Clock Stimulus

```
`timescale 1ns / 1ps

module tb_whatever ();

   reg clock=0;
   reg   list_of_inputs;
   wire list_of_outputs;

   test_module_name uut(clock, list_of_ins_and_outs)

   always #5 clock=!clock; // #5=half clock period

   …

endmodule
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity tb_whatever is
end tb_whatever;

architecture Behavioral of tb_whatever is
   component test_module_name
      port ( … );
   end component;

  signal clock : std_logic := '0';
  signal list_of_ins_and_outs : std_logic;

begin
   uut: test_module_name
            port map ( clock, list_of_ins_and_outs);

   clock <= !clock after 5 ns; -- half clock period
   …

end Behavioral;
```

University of Sheffield

Test pattern generation

```
`timescale 1ns / 1ps

module tb_whatever ();
   reg clock=0;
   reg list_of_inputs;
   wire list_of_outputs;

   test_module_name uut(clock, list_of_ins_and_outs)

   always #5 clock=!clock; // #5=half clock period

   initial begin
            some_inputs=their_intial_values;
      #10   some_inputs=next_value;
      #10   some_inputs=another_value;
      #100  $finish;
   end
endmodule
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity tb_whatever is
end tb_whatever;

architecture Behavioral of tb_whatever is
   component test_module_name
      port ( clock : in std_logic;
             list_of_out_ports : out std_logic;
             list_of_in_ports    : in  std_logic);
   end component;

  signal clock : std_logic := '0';
  signal list_of_ins_and_outs : std_logic;

begin
   uut: test_module_name port map ( clock, list_of_ins_and_outs);
   clock <= !clock after 5 ns; -- half clock period

   process begin
      some_inputs=their_intial_values;
      wait for 10 ns;
      some_inputs=next_value;
      wait for 10 ns;
      some_inputs=another_value;
      wait for 100 ns;
      wait;
   end process;

end Behavioral;
```

The complete simple TB

```
`timescale 1ns / 1ps

module tb_whatever ();
   reg clock=0;
   reg list_of_inputs;
   wire list_of_outputs;

   test_module_name uut(clock, list_of_ins_and_outs)

   always #5 clock=!clock; // #5=half clock period

   initial begin
            some_inputs=their_intial_values;
      #10   some_inputs=next_value;
      #10   some_inputs=another_value;
      #100  $finish;
   end
endmodule
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity tb_whatever is
end tb_whatever;

architecture Behavioral of tb_whatever is

   component test_module_name
      port ( clock : in std_logic;
             list_of_out_ports : out std_logic;
             list_of_in_ports  : in  std_logic);
   end component;

  signal clock : std_logic := '0';
  signal list_of_ins_and_outs : std_logic;

begin
   uut: test_module_name port map ( clock, list_of_ins_and_outs);
   clock <= !clock after 5 ns; -- half clock period
   process begin
      some_inputs=their_intial_values;
      wait for 10 ns;
      some_inputs=next_value;
      wait for 10 ns;
      some_inputs=another_value;
      wait for 100 ns;
      wait;
   end process;
end Behavioral;
```

simulation “how to” guide

Signal & variable declarations

- Define signals for all the inputs / outputs to / from the UUT
- Define constants typically arrays for your test vectors, keep these in their given format and write a function to manipulate into the format your module needs
- Define parameters for clock periods, etc for easy of editing/reading
- Ensure the initial state of inputs to the UUT is defined (especially any clocks)

```
reg   some_input=0, another_input=1;
wire list_of_outputs;
reg [7:0]DATA[1:8] = {>>{64'h1234_5678_9abc_def0}};
parameter CLOCK_PERIOD = 10;
reg clk = 0;
always #(CLOCK_PERIOD/2) clk = !clk;
```

Delays

- These can either be event triggered or simply a time delay
- To keep the UUT and test bench perfectly in sync it is often desirable to wait specifically for a clock edge in the test bench
- For a Verilog testbench the `timespec has to be defined ONCE typically the first line

#10 next_statement	wait for 10 ns;
always @rising_edge(clk)	wait until rising_edge(clk);

Loops

- Unlike synthesis loops can be defined very similar to how you would use them in a conventional programming language i.e. analogous to control flow
- Very convenient for say looping over a set of test vectors
- Just describe the loop and wait for the appropriate event / clock

```
integer i;
for (i=0; i<16; i=i+1)    // note there is no i++ in Verilog
   begin
         @(posedge clk);
         some_uut_input = SomeFn(some_signal[i]);
         while (!output_valid) @(posedge clk);
   end
```

Process blocks

- Separating independent tasks using multiple process blocks
- Eg stimulus generation, clock generation, testing, other part of system modelling
- It is very normal to have several independent loops in a test bench some of which become activated by others using a Boolean variable to periodically model / verify different parts of the system
- Eg you may have a process that verifies the value of an output is correct but only when the output_valid signal is asserted.
- Embrace multiple process blocks for more readable testbenches

Verilog system tasks

- Display text to the console
  - $display $monitor $strobe $write
- Terminate the simulation / task
  - $finish $stop
- Define the formatting and unit for reporting timestamps of events
  - $timeformat
- Read data from a text file as binary or hex into a testbench array
  - $readmemb, $readmmemh
- File I/O at the character/line level (like C programming language)
  - $fopen, $fseek, $ftell, $fgetc $fgets, $fstrobe $fflush, $ferror $fmonitor $fwrite $fclose

Displaying text

- How to display text messages and monitor signals

```
initial  begin
   $display("\t\ttime,\tclk,\treset,\tenable,\tcount");
   $monitor("%d,\t%b,\t%b,\t%b,\t%d",
                 $time, clk, reset, enable, count);
end
```

```
use  std.textio.all;
…
report "Time, clk, reset, enable, count";
…
process      -- to get the same as Verilog monitor
  variable txt : line;
begin
  write(txt, time);       write(txt, string'(","));
  write(txt, clk);        write(txt, string'(","));
  write(txt, reset);      write(txt, string'(","));
  write(txt, enable);     write(txt, string'(","));
  write(txt, count);      write(txt, string'(","));
  report "values are " & txt.all;
  wait on clk;
end process;
```

These Verilog system tasks all have same syntax but operate slightly differently:

$display output immediately
$monitor output every time value(s) change
$strobe output at the end of time step (eg $finish)

University of Sheffield

Waveform Viewer

- Most simulators have a waveform view like a traditional logic analyser
- This can be used to delve down into signals in the design hierarchy not connected to the top level – typically just expand the hierarchy and drag the signals to the viewer then rerun simulation
- Signals can be grouped, coloured and dividers added to make display more human readable.
- The radix (number base) for bit vectors can be set to binary, hex, octal or even ASCII
- TopTip: if you use enumerated type for state machines the waveform viewer will automatically display the textual names

Value Change Dump (VCD)

- Put your signals and waveforms into a file
- Documents switching activity
- Required for accurate power modelling
- Basic Format
- How to invoke

```
initial  begin
    $dumpfile ("whatever.vcd");
    $dumpvars;
 end
```

There is no equivalent command in VHDL.
Either do it from the simulator command window using the “vcd” command
Or use a Verilog test bench

Refer to the user guide for your simulator for further details (eg ModelSim or xsim)

Generating test vectors

- Eg random number generation in a test bench
- Verilog has some system functions:

  $random $urandom $urandom_range()
- VHDL is predictably more wordy!
- Alternative is to use LFSR, hash, cipher, …

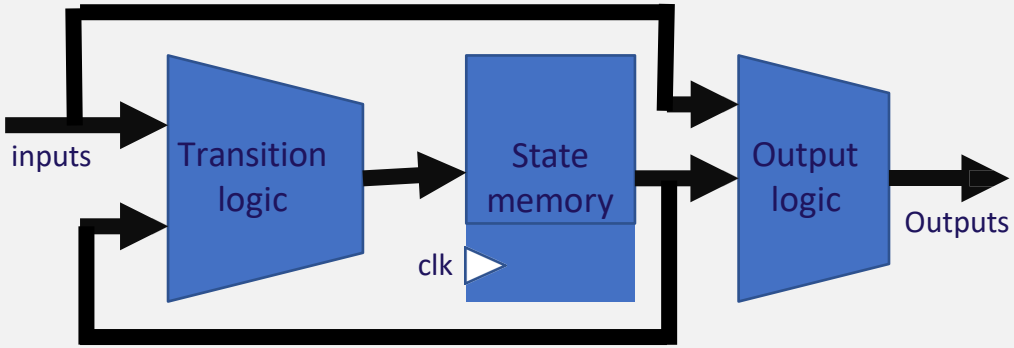
```
// Verilog random in testbench
integer seed = 1234, rv;
initial begin
   repeat (10) begin
      rv = $random(seed) % 31;
      $display("%d", rv);
   end
end
```

```
-- vhdl random in testbench
library ieee;
use ieee.math_real.uniform;
use ieee.math_real.floor;
…
process is
   variable seed1 : positive := 1234;
   variable seed2 : positive := 5678;
   variable x : real;
   variable y : integer;
  begin
    for  n  in  1  to  10  loop
       uniform(seed1, seed2, x);
       y := integer(floor(x * 1024.0));
       report "Random number in 0 .. 1023: "
                & integer'image(y);
    end loop;
   wait;
end process;
```

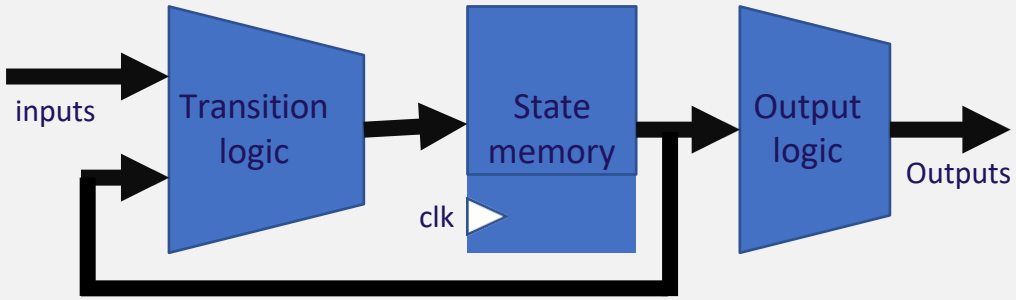
FSM: Finite State Machine

FSM architecture

- It is an abstract machine that can be in exactly one of a finite number of states at any instant in time.
- It changes state in response to inputs and current state; typically clocked.
- It has an initial (default) state (on reset).
- To be useful it requires some definition of output function the form of which defines whether it is a Mealy or Moore machine.



MEALY MACHINE



MOORE MACHINE

FSM state encoding

- The state is encoded as a set of numbers. In Verilog there is no enumerated type so these are declared as parameters (constants). Just number as 1,2,3,4,…. In VHDL use enumerated type without assigning any numerical values.
- By default the tools IGNORE the numbers you enter! (unless you set fsm_encoding="User").
- The tools normally do a very good job assigning the state encoding automatically. An attribute fsm_encoding can be used to override this.
- Vivado options are: Auto, One-Hot, Compact, Sequential, Gray, Johnson, User, Speed or None. Just set this attribute on the state variable/signal to define your design intent. For example:

```
(* fsm_encoding = "one_hot" *) reg [1:0] current_state;
```

University of Sheffield

```
(* fsm_encoding = "one_hot" *) reg [1:0] current_state;
```

FSM state encoding

- Binary: use binary code to represent states
- One-hot: There will be a unique "1" in the encoding, which represent the state
- Gray: only one bit can be changed in the transition of the states
- Johnson: LSB = ~MSB, shift << 1
- Compact: Least area mode, automatically pick the most area efficient way
- Sequential: Optimize the state transition process
- User: Use user's encoding
- Speed: Max performance mode.
- None: The compiler will not regard this part of code as FSM
- Safe mode: Automatically add default sentence if user doesn't setup default
- Auto: Automatically pick up the most suitable one above

A WORLD TOP 100 UNIVERSITY

University of Sheffield

FSM avoiding problems

- Two things to be aware of is that the output from the FSM can contain glitches and if the inputs are not synchronised their transitions can lead to an invalid state being selected.
- Both of these can be fixed by registering/synchronising inputs and outputs at the cost of a clock cycle delay. FSM choices such as one-hot or Gray can also help prevent unwanted glitches.
- For one-hot the tools automatically generate error-correction logic if multiple states are 'hot' see UG901 Synthesis guide FSM_SAFE_STATE for details.
- Typically if you are doing synchronous (clocked) design with a single clock then you wont experience any problems – just take care with off chip inputs & outputs.
- Ensure that your initial state is defined on FPGA initialisation and for any external reset signal

University of Sheffield

FSM Descriptions

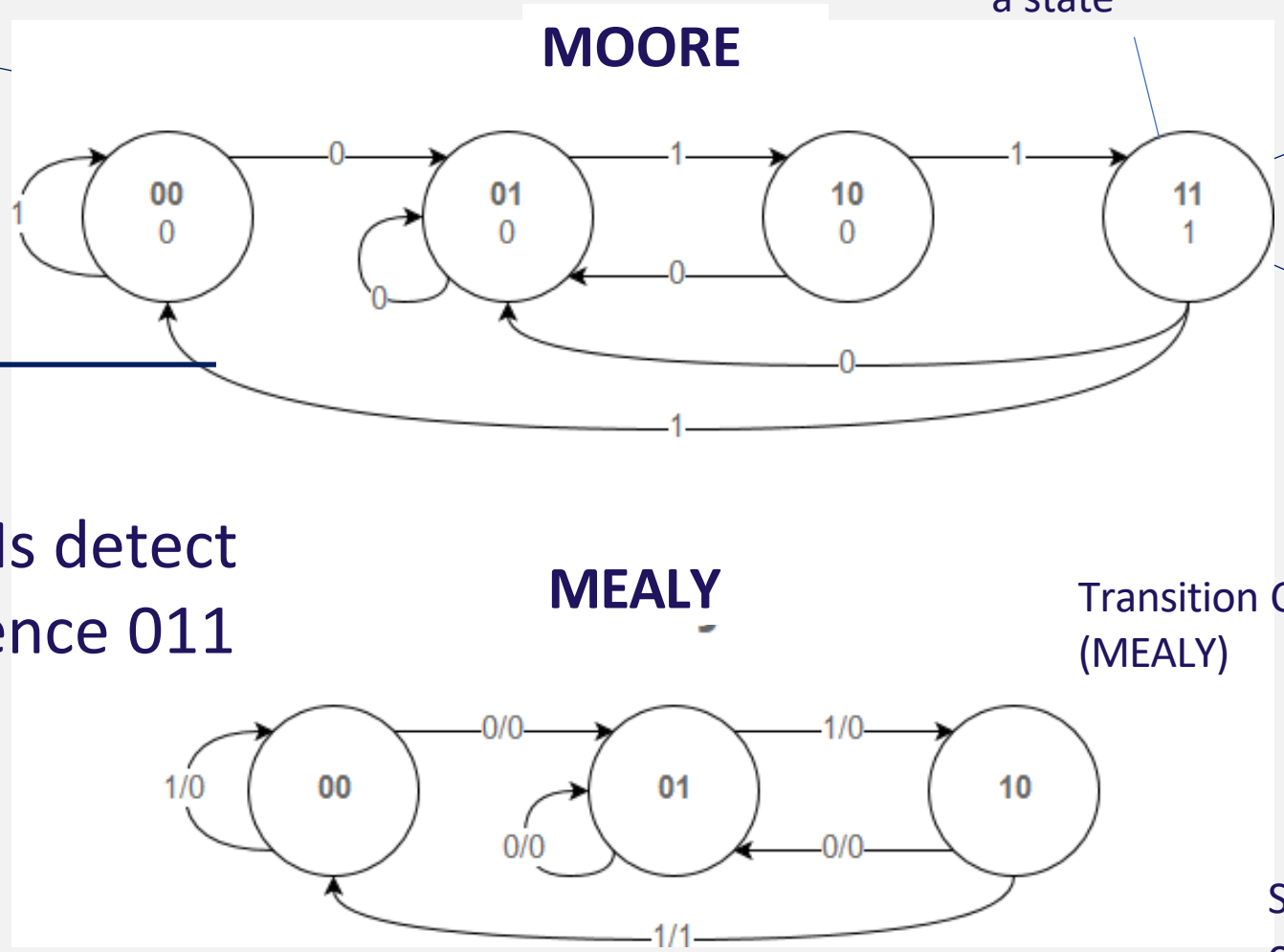
- State diagram (bubbles, arrows, describe output)
- truth tables for next state and output signal
  - Sometime use Karnaugh Maps but usually state transition table
- Timing diagrams (logic analyser view showing a state sequence)
- Transitions can be:
  - event driven (depend on the inputs),
  - automatic progression (clocked in some sequence)
  - self-transitions (automatic on next clock)

State Diagrams Explained

Circle indicate a state

Transition condition (input)

State encoding or name



Output value (MOORE)

Line indicates a possible transition

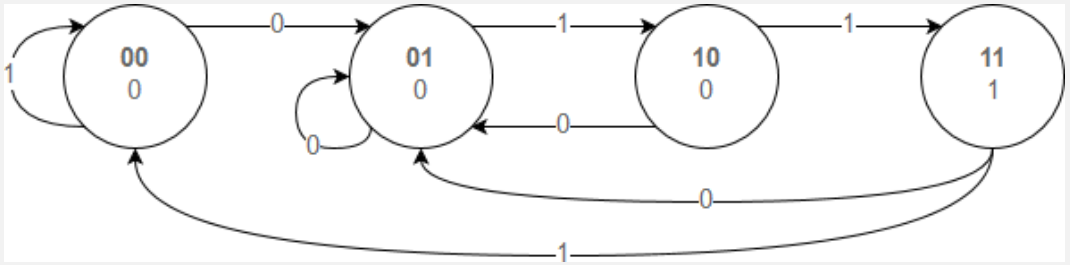
Both these FSMs detect the input sequence 011

Transition Condition "/" Output value (MEALY)

State encoding or name

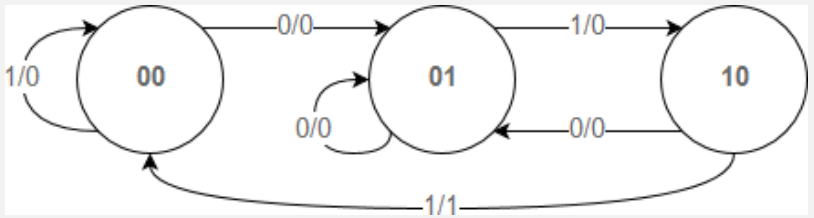
State Transition Table

MOORE



Current State	Next State		Output
	Input=0	Input=1	
00	01	00	0
01	01	10	0
10	01	11	0
11	01	00	1

MEALY



Current State	Input	Next State	Output
00	0	01	0
00	1	00	0
01	0	01	0
01	1	10	0
10	0	01	0
10	1	00	1
11	0	X	X
11	1	X	X

HDL coding strategy to avoid human errors

- Please follow this style of coding until you are very confident with the language - it does make it easier to detect errors.
- There are several other ways to describe a FSM – I'll use some of these in the notes.
- There is a sequential block to update the current_state with the next_state.
- A combinatorial block for transition and output function
- We define the "defaults" for each output and to remain in the current state unless we state otherwise.

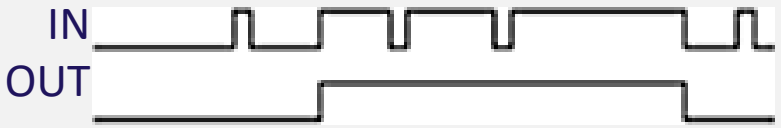
```
module ExampleFsm( input clk, nrst, bitin, output reg Q)
// best to use parameter to give states suitable names
parameter ZeroState=0, FirstState=1, SecondState=2, ThirdState=3;
reg [1:0] current_state=0, next_state; // registers for state variable

always @(posedge clk)     // sequential state update
   if (nrst) current_state <= next_state;
      else  current_state <= ZeroState;

always  @(*)  begin  // combinatorial  transition  &  output  logic
   next_state <= current_state; // by default stay in same state
   Q<=0;  // set the default case for all outputs
   case (current_state)  // case statement to define behaviour
      ZeroState:      if (bitin==0) next_state <= FirstState;
      FirstState:     if (bitin==1) next_state <= SecondState;
      SecondState: if (bitin==1) next_state <= ThirdState;
      ThirdState: begin
                Q<=1;
                if (bitin==1) next_state <= FirstState;
                   else        next_state <= ZeroState;
            end
      default: next_state <= ZeroState;  // always have default
   endcase
end
endmodule
```

FSM Examples

It is quickest to learn about FSMs by example so here are several to get you going

Glitch remover

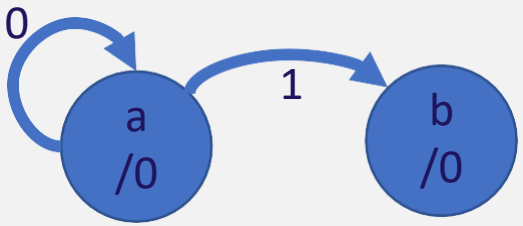
- Design Intent is for a noise elimination circuit which removes any pulses, either a zero or a one, which only last one clock cycle.



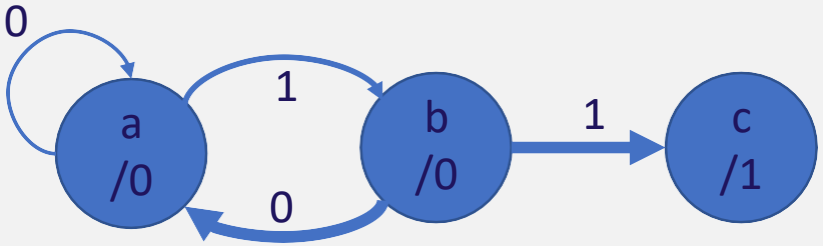
- Outline design:
  - We'll start by choosing to design a Moore machine
  - Assign each state a letter a,b,c,….
  - Work out what transitions should be for each possible input and construct the ASM chart for all possibilities of each successive input bit.

Glitch remover

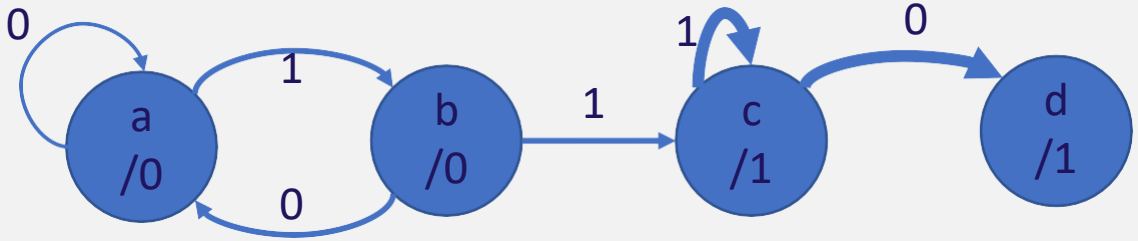
- IN 00, 01



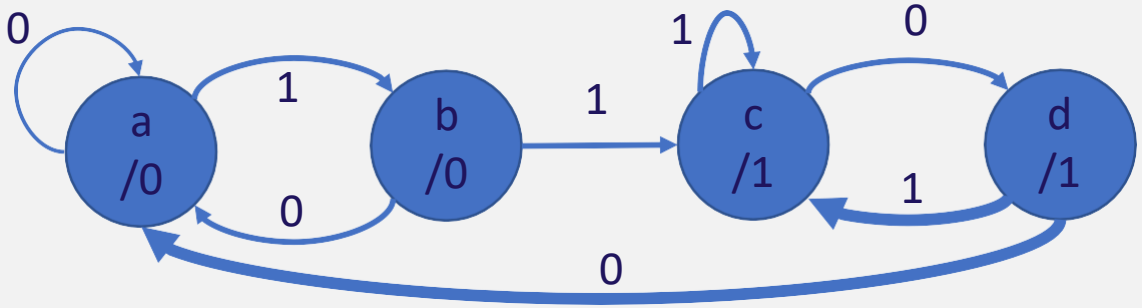
- IN ...00 ...01



- IN ...10 ...11



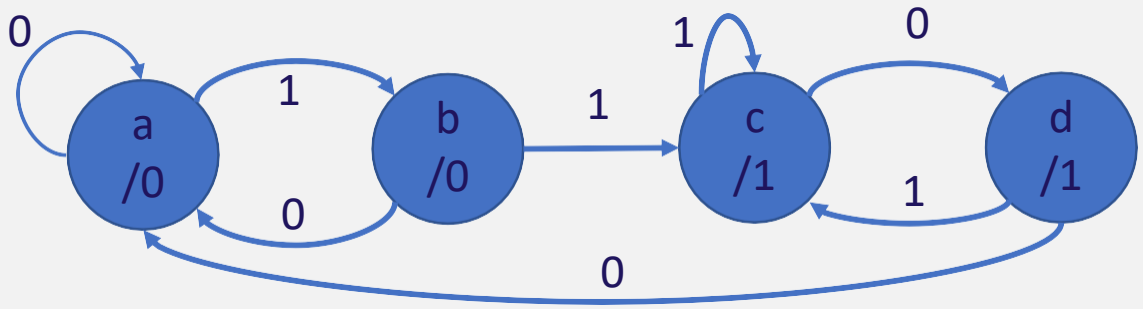
- IN 101 111



Glitch remover

There are three parts to the Verilog description:

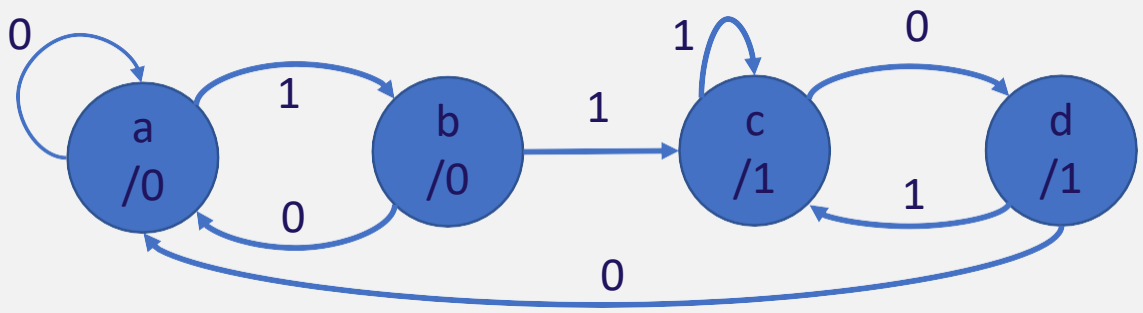
- Declarations
- State transitions
- Output logic




```
/* This is an alternative common FSM description to give you an
example: note that the state transitions is the 'clocked' process. */

// declarations
module GlitchRemover( output reg out,  input in, clk, nrst);
parameter ST_A=0,  ST_B=1, ST_C=2,  ST_D=3;
reg [1:0]  state = ST_A;

// state transitions
always @(posedge clk)
   if (nrst==1'b0) state <= ST_A;
   else
      case (state)
         ST_A:  if (in==1'b1) state <= ST_B;
         ST_B:  if (in==1'b1) state <= ST_A; else state <= ST_C;
         ST_C:  if (in==1'b0) state <= ST_D;
         ST_D:  if (in==1'b1) state <= ST_B; else state <= ST_A;
         default: ;  // do nothing
      endcase

// output logic
always @ (*)
    case (state)
         ST_A: out <= 1'b1;
         ST_B: out <= 1'b0;
         ST_C: out <= 1'b1;
         ST_D: out  = 1'b1;
   endcase
endmodule
```

Glitch remover

We verify it by creating a testbench and simulating in XSIM.




```
`timescale 1ns / 1ps        // Verilog testbench
module tb_glitchremover();

   reg in=0, clk=0, nrst=0;
   wire out;
   GlitchRemover UUT( out, in, clk, nrst );

   always #5 clk=!clk;

   integer i,  times[0:7]={>>{5,1,3,3,1,5,1,5}};
   initial begin
      #15 nrst=1;
      for ( i=0; i<8; i=i+1 )
         #(10*times[i]) in=!in;
      end
endmodule
```

Exercise: See if there are any logic errors in the FSM code?

University of Sheffield

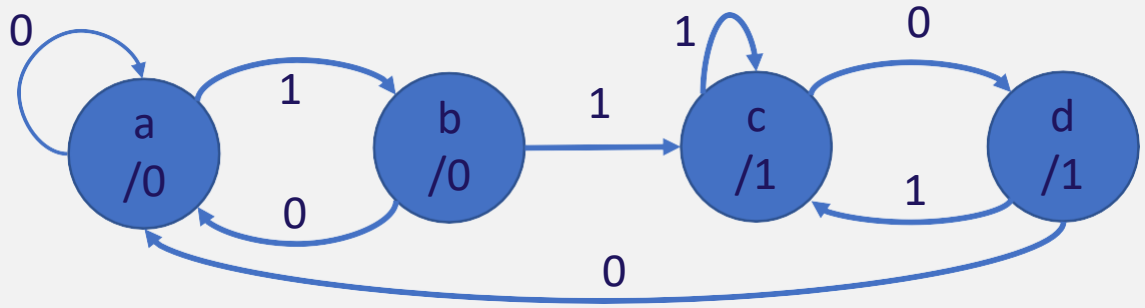
Glitch remover

```
/* This is an alternative common FSM description to give you an
example: note that the state transitions is the 'clocked' process. */

// declarations
module GlitchRemover( output reg out,  input in, clk, nrst);
parameter ST_A=0,  ST_B=1, ST_C=2,  ST_D=3;
reg [1:0]  state = ST_A;

// state transitions
always @(posedge clk)
   if (nrst==1'b0) state <= ST_A;
   else
      case (state)
         ST_A:  if (in==1'b1) state <= ST_B;
         ST_B:  if (in==1'b1) state <= ST_C; else state <= ST_A;
         ST_C:  if (in==1'b0) state <= ST_D;
         ST_D:  if (in==1'b1) state <= ST_C; else state <= ST_A;
         default: ;  // do nothing
      endcase

// output logic
always @ (*)
    case (state)
         ST_A: out <= 1'b0;
         ST_B: out <= 1'b0;
         ST_C: out <= 1'b1;
         ST_D:  out <= 1'b1;
   endcase
endmodule
```

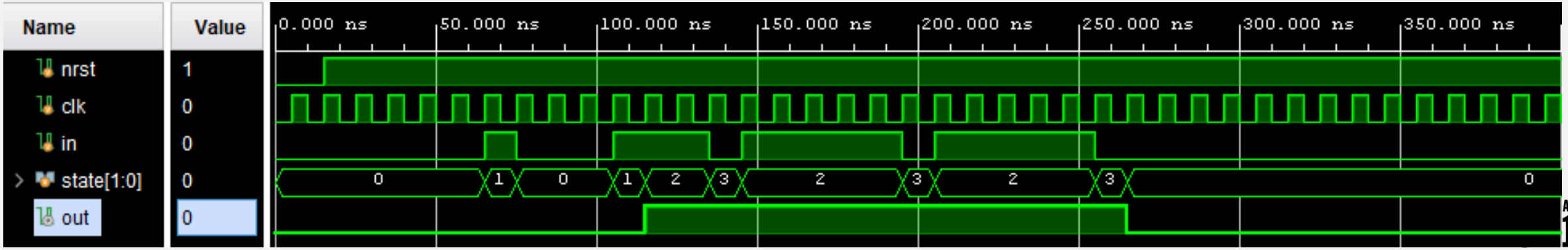
A WORLD TOP 100 UNIVERSITY

Glitch remover

FSM State transitions.

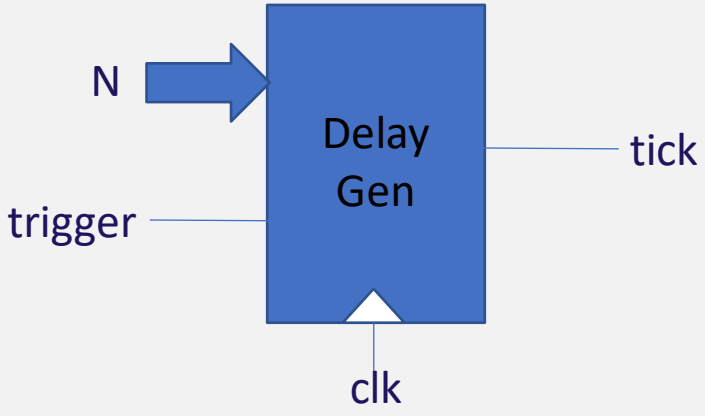


We verify it by creating a testbench and simulating in XSIM.



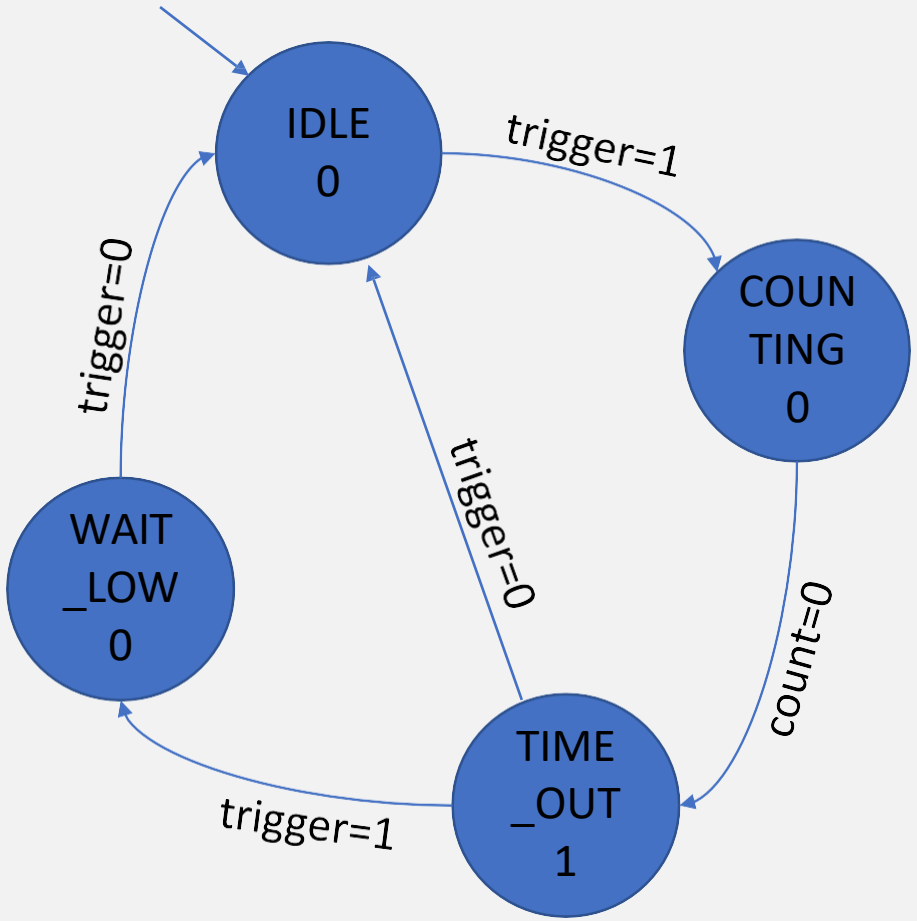
Delay generator

- Design intent is to detect the rising edge of the trigger then after a N cycles delay of the system clock produce a one cycle pulse on the tick signal. The range of N is 0..1023 cycles (10 bits).




```
// Create Date: 02.02.2020 13:13:13
// Module Name: DelayGen
// Engineer: T Good
// Target Devices: Any FPGA
// Tool Versions: Vivado 2019.2
// Description: Generate a single cycle tick pulse 1..1024 cycles after trigger
// Revision: 1

module DelayGen
 #( parameter DELAY_BITS = 10 )
   ( input clk,
     input trigger,
     input [DELAY_BITS-1:0] N,
     output reg tick );
```

Delay generator

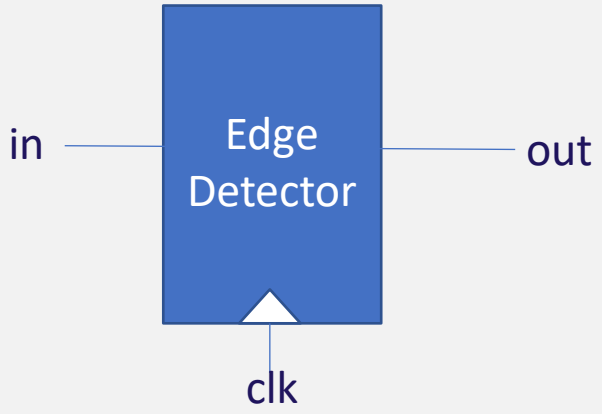



```
module DelayGen
 #( parameter DELAY_BITS = 10 )
   ( input clk, trigger, [DELAY_BITS-1:0] N,  output reg tick );

// declare out counter
reg [DELAY_BITS-1:0] count = {{1'b1}};  // initially 111..111

// declare our FSM states
localparam IDLE=0, COUNTING=1, TIME_OUT=2, WAIT_LOW=3;
reg [1:0] state = IDLE;

always @(posedge clk)
  case (state)
      IDLE:          if (trigger==1'b1) state <= COUNTING;
      COUNTING: if (count==0)
                      begin count<={{1'b1}}; state<=TIME_OUT; end
                      else count <= count - 1;
      TIME_OUT:  if (trigger==1'b0) state <= IDLE;
                      else state <= WAIT_LOW;
      WAIT_LOW: if (trigger==1'b0) state <= IDLE;
      default:       state <= IDLE;
  endcase

assign tick = (state==TIME_OUT) ? 1'b1 : 1'b0;

endmodule
```

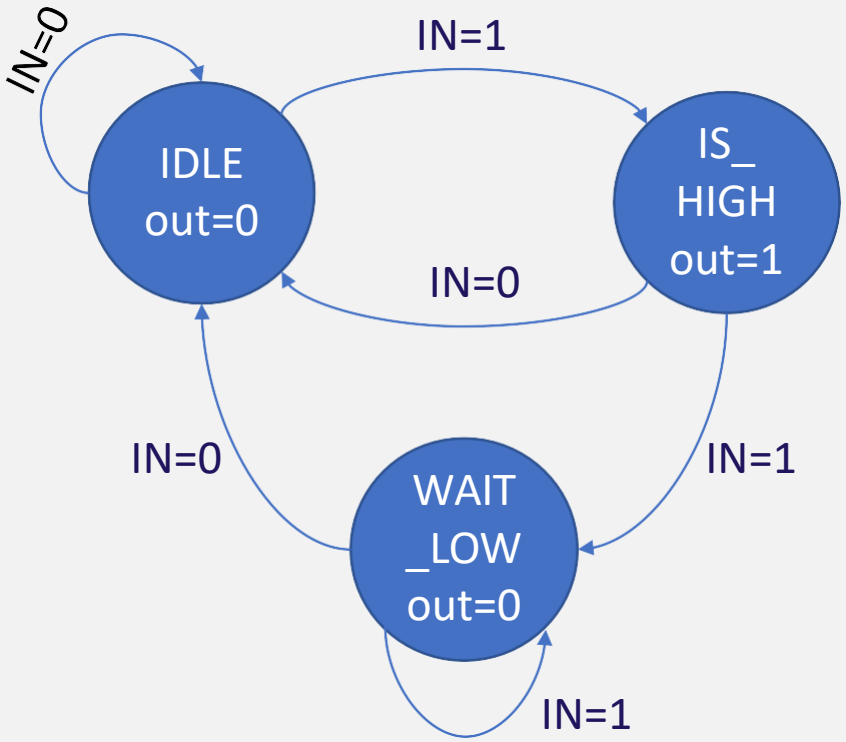
Exercise: Synthesis this and look at report to see what it infers? Hopefully you should see a FSM and a COUNTER.

Pulse generator (edge detector)

- Design intent is a synchronously clocked module which on each positive edge of the input generates an output lasting one clock cycle.



Exercise: Code it in the Vivado



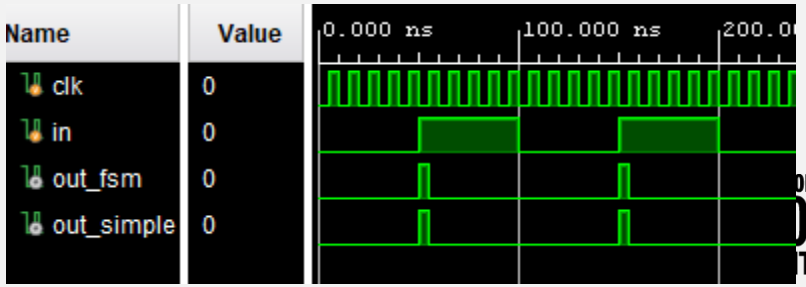
Pulse generator

- This is an example of overkill to describe such a simple module using a FSM.
- For an edge detector it could be described in a few lines.

```
module EdgeDetectorSimple ( input clk, in, output out );
    reg prev = 0;
    always @(posedge clk) prev = in;
    assign out = in & !prev;
endmodule
```

Exercise: Synthesis both versions. Are there any difference in the utilization result?

```
module EdgeDetectorFsm
   ( input clk, in, output reg out );

// declare our FSM states
parameter IDLE=0, IS_HIGH=1, WAIT_LOW=2;
reg [1:0] state = IDLE;

always @(posedge clk)
  case (state)
      IDLE:        if (in==1'b1) state <= IS_HIGH;
      IS_HIGH:     if (in==1'b1) state <= WAIT_LOW;
                   else state <= IDLE;
      WAIT_LOW: if (in==1'b0) state <= IDLE;
      default:     state <= IDLE;
 endcase

 always @ (*)
  case (state)
     IDLE:          out <= 1'b0;
     IS_HIGH:   out   <=   1'b1;
     WAIT_LOW:  out <= 1'b0;
  endcase
endmodule
```



Pulse generator

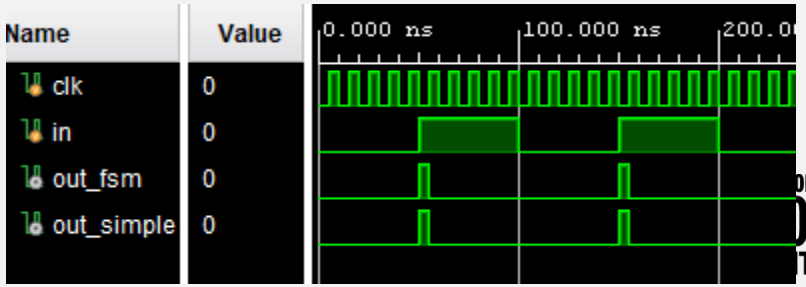
- This is an example of overkill to describe such a simple module using a FSM.
- For an edge detector it could be described in a few lines.

```
module EdgeDetectorSimple ( input clk, in, output out );
    reg prev = 0;
    always @(posedge clk) prev = in;
    assign out = in & !prev;
endmodule
```

Name	Slice LUTs (63400)	Slice Registers (126800)	Bonded IOB (210)	BUFGCTRL (32)
EdgeDetectorSimple	1	1	3	1

Exercise: Synthesis both versions. Are there any difference in the utilization result?

```
module EdgeDetectorFsm
   ( input clk, in, output reg out );

// declare our FSM states
parameter IDLE=0, IS_HIGH=1, WAIT_LOW=2;
reg [1:0] state = IDLE;

always @(posedge clk)
  case (state)
      IDLE:        if (in==1'b1) state <= IS_HIGH;
      IS_HIGH:     if (in==1'b1) state <= WAIT_LOW;
                   else state <= IDLE;
      WAIT_LOW: if (in==1'b0) state <= IDLE;
      default:     state <= IDLE;
 endcase

 always @ (*)
  case (state)
     IDLE:          out <= 1'b0;
     IS_HIGH:   out   <=   1'b1;
     WAIT_LOW:  out <= 1'b0;
  endcase
endmodule
```



University of Sheffield

Building Blocks & IPs

Developing useful building blocks for digital design

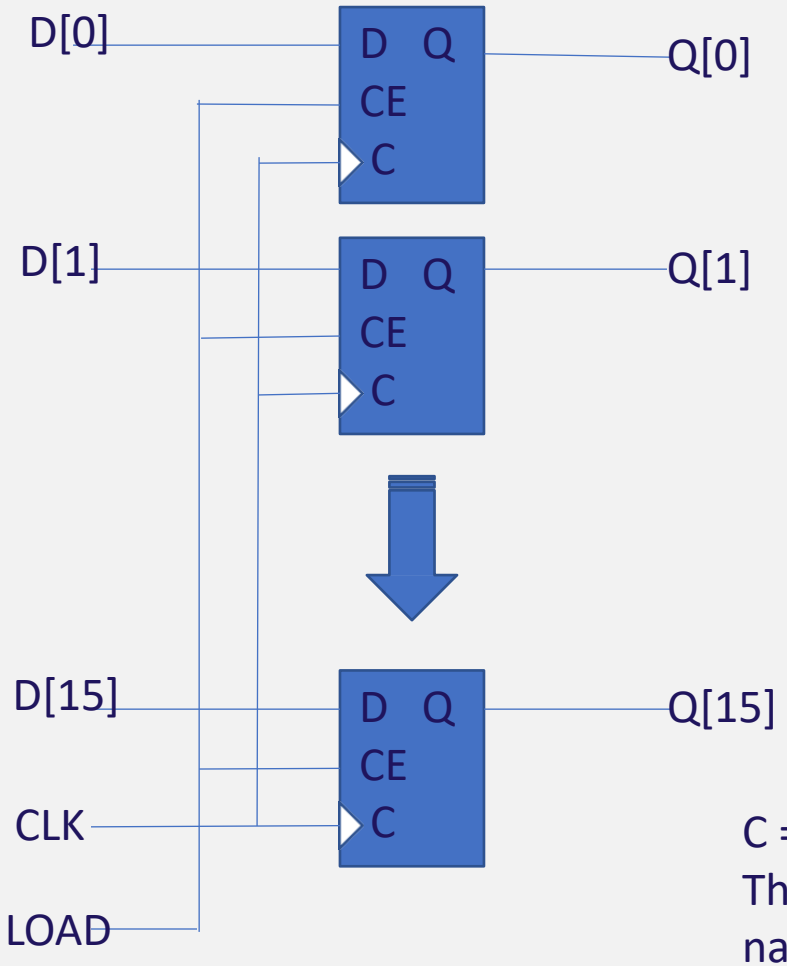
A WORLD TOP 100 UNIVERSITY

Lego Parts for Lego Toy, Building Blocks for Hardware

- Logic building blocks described in different ways.
- Building blocks are concepts such counters, shift register, multiplexers, synchronisers, etc…
- The ways may be symbolic (gates, circuits) or mathematical (Boolean, truth tables) or linguistic (Verilog, vhdl)




Registers / DFFs



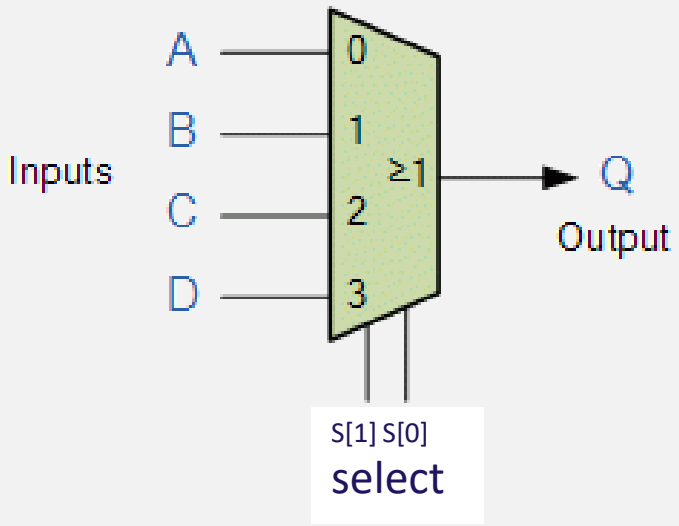
CLK	LOAD	D	Q
0	any	any	Same
1	any	any	Same
_/---	0	any	Same
_/---	1	0/1	D

C = clock, CE = clock enable
This is an fpga 'thing' due to the global nature of the clock network

```
// Verilog

always @ (posedge(clk)) begin
    if (load) begin
        Q<=D;
    end
end
```

```
-- VHDL
process (clk) begin
  if rising_edge(clk) then
    if load='1' then
        Q<=D;
    end if;
  end if;
end process;
```

Multiplexer



S[1]	S[0]	A,B,C,D	Q
0	0	0/1	A
0	1	0/1	B
1	0	0/1	C
1	1	0/1	D

Remember: ensure cases are complete otherwise you'll end up inferring a latch

Exercise:

1. Try describing a mux using conditional continuous assignment statement.
2. Write a N-way mux which using a bit-vector for the input

```
// Verilog
always @ (*)   begin
      case (select)
          2'b00: Q = A;
          2'b01: Q = B;
          2'b10: Q = C;
          2'b11: Q = D;
       endcase
end
```

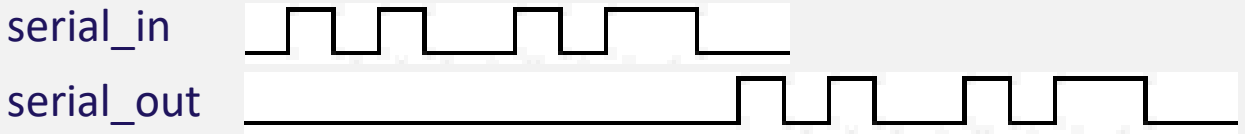
```
-- VHDL
process (All) begin
   case (select) is
     when "00" => Q <= A;
     when "01" => Q <= B;
     when "10" => Q <= C;
     when "11" => Q <= D;
     when others => null;
   end   case;
end process;
```

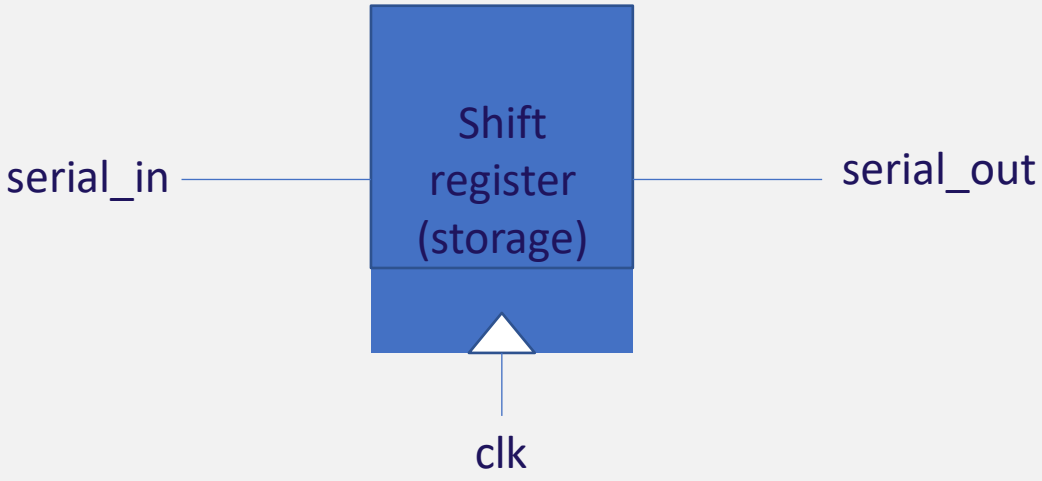
Encoders / Decoders

- In the olden days of discrete logic chips we could buy devices such as 3-to-8 line decoders (74LS138) and 8-bit priority encoders (CD4532B) so these concepts are still thought of as fundamentals.
- In HDL terms these can all be done with a simple case statement or lookup table or ROM to define their behaviour. The tools handle the detail for us automatically.
- We just need to define output as some function of the input. Typically indexing into an array or a case statement is all we need.
- Follow the example of a Priority Encoder or ROM or SevenSeg lab1&2.

Shift Register

- Serial in and serial out (could be multibit)
- Values appear with a shift-length delay in clock cycles (eg 10)



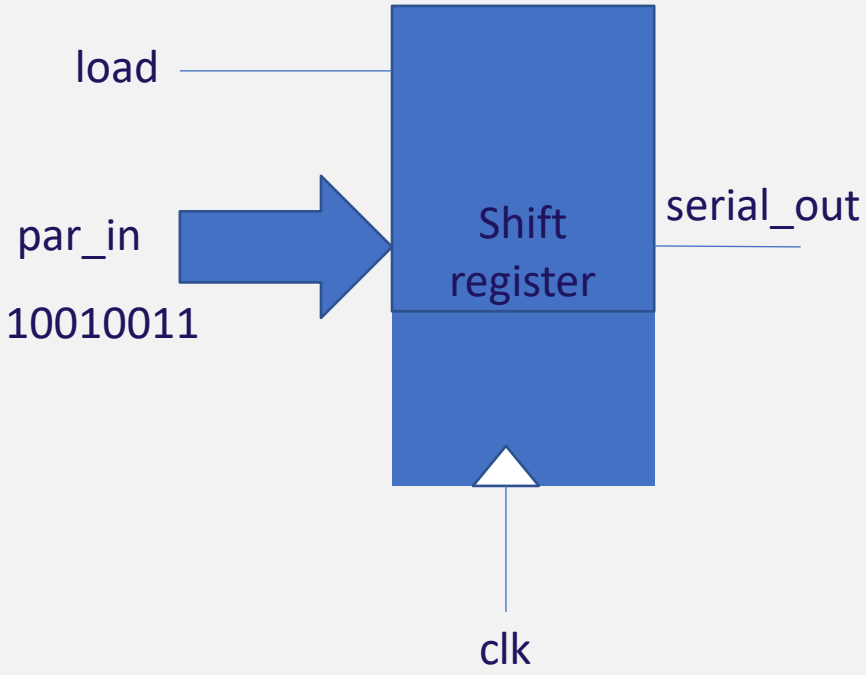



```
parameter shift = 10;      // verilog

reg [shift-1:0] SR = {shift{1'b0}};

always @(posedge clk)
  SR <= {serial_in, SR[shift-1:1]};

assign serial_out = SR[0];
```

```
generic ( width : positive := 10 )          -- vhdl

signal SR : std_logic_vector(width-1 downto 0);
process (clk) begin
  if rising_edge(clk) then
    SR <= serial_in & SR(width-1 downto 1);
  end if;
end process;
serial_out <= SR(0);
```

Parallel to serial – This is similar to Lab 4 TxD

- Shift register: parallel in and serial out with load

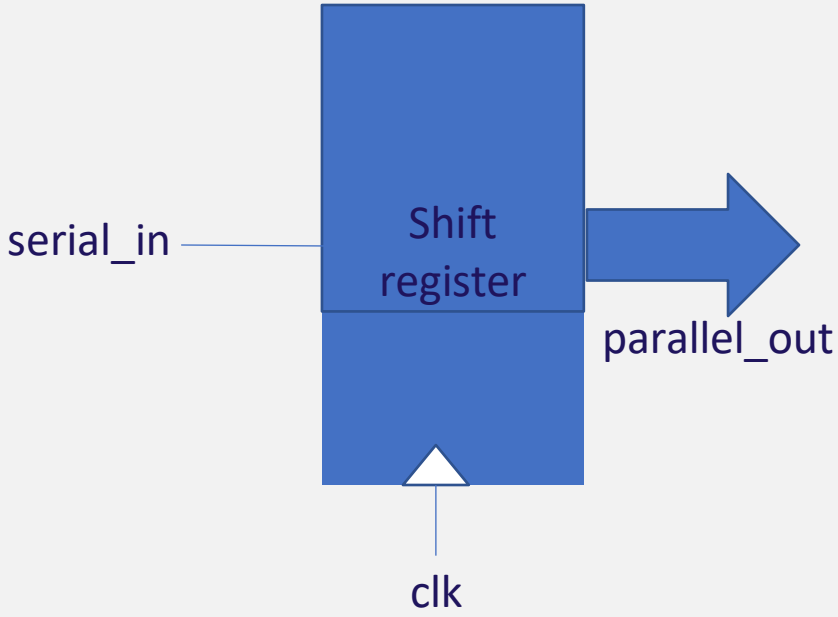


Clock cycle	Load	Internal SR value	Serial out
0	0	00000000	0
1	0	00000000	0
2	1	10010011	1
3	0	01001001	1
4	0	00100100	0
5	0	00010010	0
6	0	00001001	1
7	0	00000100	0
8	0	00000010	0
9	0	00000001	1
10+	0	00000000	0

```
parameter piso_shift = 8;
reg [piso_shift-2:0] SR
            = {piso_shift-1{1'b0}};
reg serial_out = 1'b0;

always @(posedge clk)
   if (load) begin
      SR<= par_in [piso_shift-1:1];
      serial_out  <= SR [0];
   end
   else begin
      SR <= {1'b0, SR[piso_shift-2:1]};
      serial_out   <= SR[0];
   end
```

Serial to Parallel

- This is simplest case, normally would have extra control for framing and handshaking




```
parameter shift = 10;      // verilog

reg [shift-1:0] SR = {shift{1'b0}};

always @(posedge clk)
  SR <= {serial_in, SR[shift-1:1]};

assign parallel_out = SR;
```

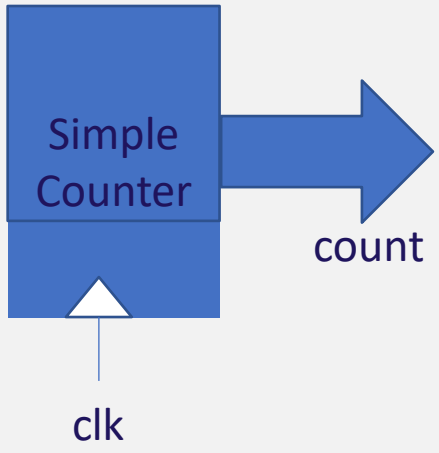
```
generic ( width : positive:= 10 )          -- vhdl

signal SR : std_logic_vector(width-1 downto 0);
process (clk) begin
  if rising_edge(clk) then
    SR <= serial_in & SR(width-1 downto 1);
  end if;
end process;
parallel_out <= SR;
```

Simple Counter

- Count from zero upwards wrapping around at 2^N -1 for example 8-bits
- Every clock cycle the counter increases by one:
- 0,1,2,3…….253,254,255,0,1,2,3….
- Always initialise a simple counter otherwise simulation will fail.

```
// Verilog

reg [7:0] count = 0;

always @ (posedge clk) count <= count +1;
```

```
-- vhdl

signal count : std_logic_vector(7 downto 0)
                    := ( others => '0' );

process (clk) begin
   if rising_edge(clk) then
            count<=count+1;
   end  if;
end process;
```

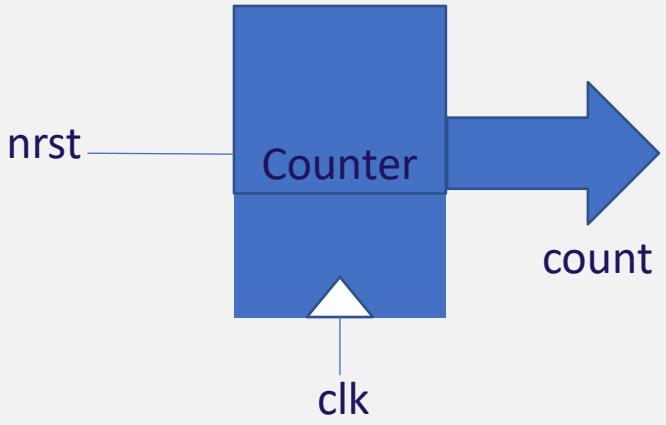


Exercise:
Create a counter that counts from 15 to 0 (i.e. backwards), and automatically back to 15 to count endlessly.

University of Sheffield

Counter with reset

- Adding a synchronous reset and wrapping around at some predefined value
- In VHDL we can use ranged integer to give a concise definition, std_logic_vector or unsigned() can be used too.
- As an alternative can also write count<=(count+1) mod N;

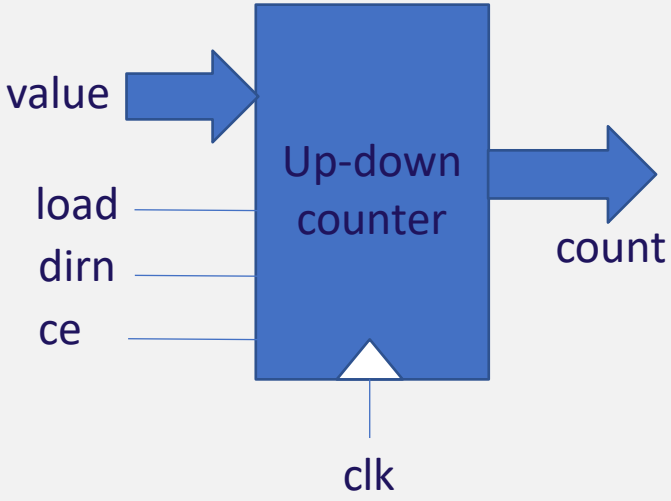



```
module modNctr
  # (parameter N = 10, WIDTH=4)
    ( input clk, nrst,
      output reg [WIDTH-1:0] count = 0 );
always @ (posedge clk) begin
  if (!nrst || (count==N-1))
        count = 0;
  else
        count <= count +1;
  end
endmodule
```

```
constant N : integer := 10;
signal count : integer range 0 to N-1 := 0;

process(clk) begin
   if rising_edge(clk) then
       if (nrst='0') or (count=N-1) then
            count<= 0;
       else
            count<=count+1;
   end  if;
end process;
```

Up Down Counter

- Lets add a dirn signal for direction so dirn=1 for count up and dirn=0 for count down
- Also lets add a load signal to allow it to be loaded with some input value

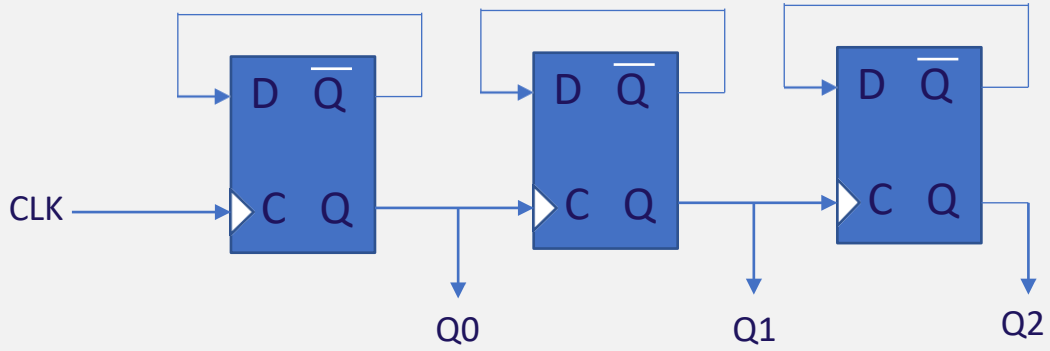



```
module UpDownCtr
 # (parameter WIDTH=4)
   ( input clk, nrst, load, ce, dirn,
    input avalue[WIDTH-1:0],
    output reg [WIDTH-1:0] count = 0 );

  always @(posedge clk)
    if (!nrst)
      count <= 0;
    else if (ce)
      if (load)
        count <= avalue;
      else if (dirn)
        count <= count + 1;
      else
        count<= count - 1;
endmodule
```

```
use IEEE.numeric_std.ALL;
...
constant WIDTH : positive := 4;
signal count :
   unsigned(WIDTH-1 downto 0) := 0;
...
process (clk)
begin
  if rising_edge(clk) then
    if nrst='0' then
      count<= (others => '0');
    elsif ce='1' then
      if load='1' then
        count <= new_count_value;
      else
        if dirn='1' then
          count <= count + 1;
        else
          count <= count - 1;
        end if;
      end if;
    end if;
  end if;
end process;
```

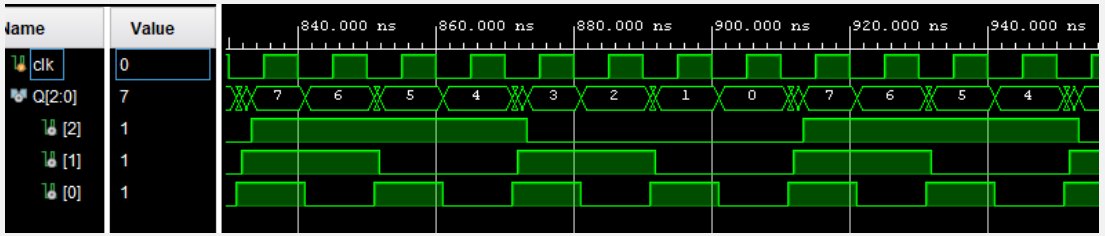
Ripple Counter

- This is an example of an <u>asynchronous</u> sequential circuit using toggle flip-flops
- Popular in ASIC design, but performs poorly on FPGAs, since normally we don't use other signals for clock. Too much uncertainty!




```
module ripple_counter( input clk, output reg [2:0] Q = 0);
   // notice sneaky initialiser above needed for simulation
   always @ (posedge  clk  ) Q[0]=!Q[0];
   always @ (posedge Q[0]) Q[1]=!Q[1];
   always @ (posedge Q[1]) Q[2]=!Q[2];
endmodule
```

BEHAVIOURAL SIMULATION

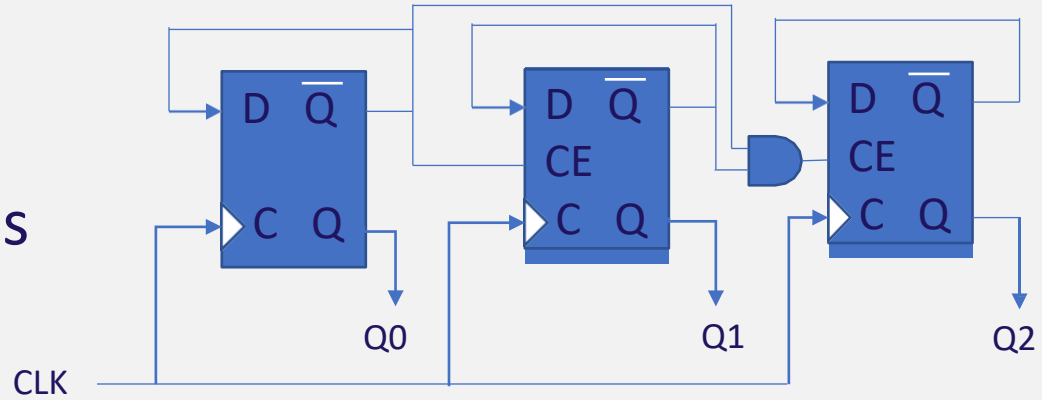


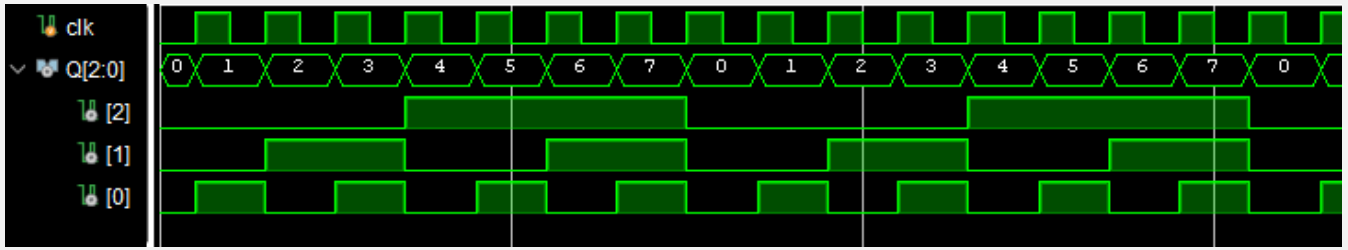
POST ROUTE TIMING SIMULATION

Exercise:
This appears to count backwards what would you change to make it count forwards?

Synchronous version….

- We can still do this with the toggle flip-flops with a single clock
- This is appropriate for an FPGA

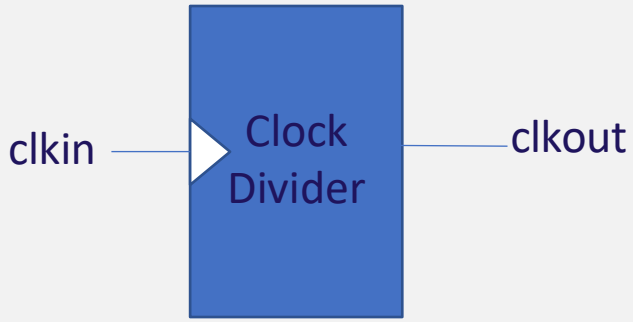





```
module sync_ripple_counter ( input clk, output reg [2:0] Q = 0);
always @ (posedge clk) begin
      Q[0] = !Q[0];
      if (!Q[0])              Q[1] = !Q[1];
      if (!Q[0] & !Q[1])   Q[2] = !Q[2];
   end
endmodule
```

```
-- VHDL
process(clk) begin
  if rising_edge(clk) then
     Q(0) <= not Q(0);
     if Q(0)='0' then
          Q(1) <= not Q(1);
     end if;
     if (Q(0)='0') and (Q(1)='0') then
          Q(2) <= not Q(2);
     end if;
  end if;
end process;
```

Clock Divider



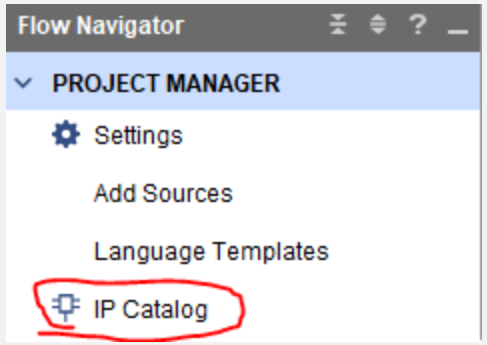
- We can simply use a counter (half-value) and when it wraps then toggle the output which then forms the divided clock.
- The tools will automatically handle all the detail no need to do anything special.
- The alternative is to use the CoreGen Wizard to instantiate a clock-tile which contains a PLL and can synthesise clocks radically different from the source crystal even fractional.

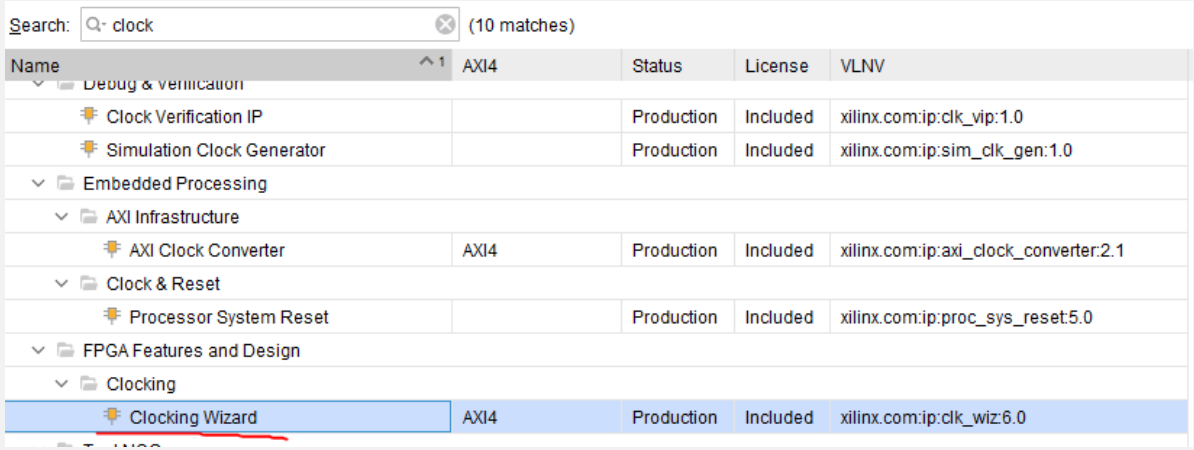
$$\frac{clk\_in}{clk\_out} = 2^N$$

```
module ClkDiv(input clk_in,output clk_out);
   reg [32:0] counter = 0;
   always @(posedge clk_in)
   begin
      counter = counter + 1;
   end

   assign clkout = counter[N];
endmodule
```

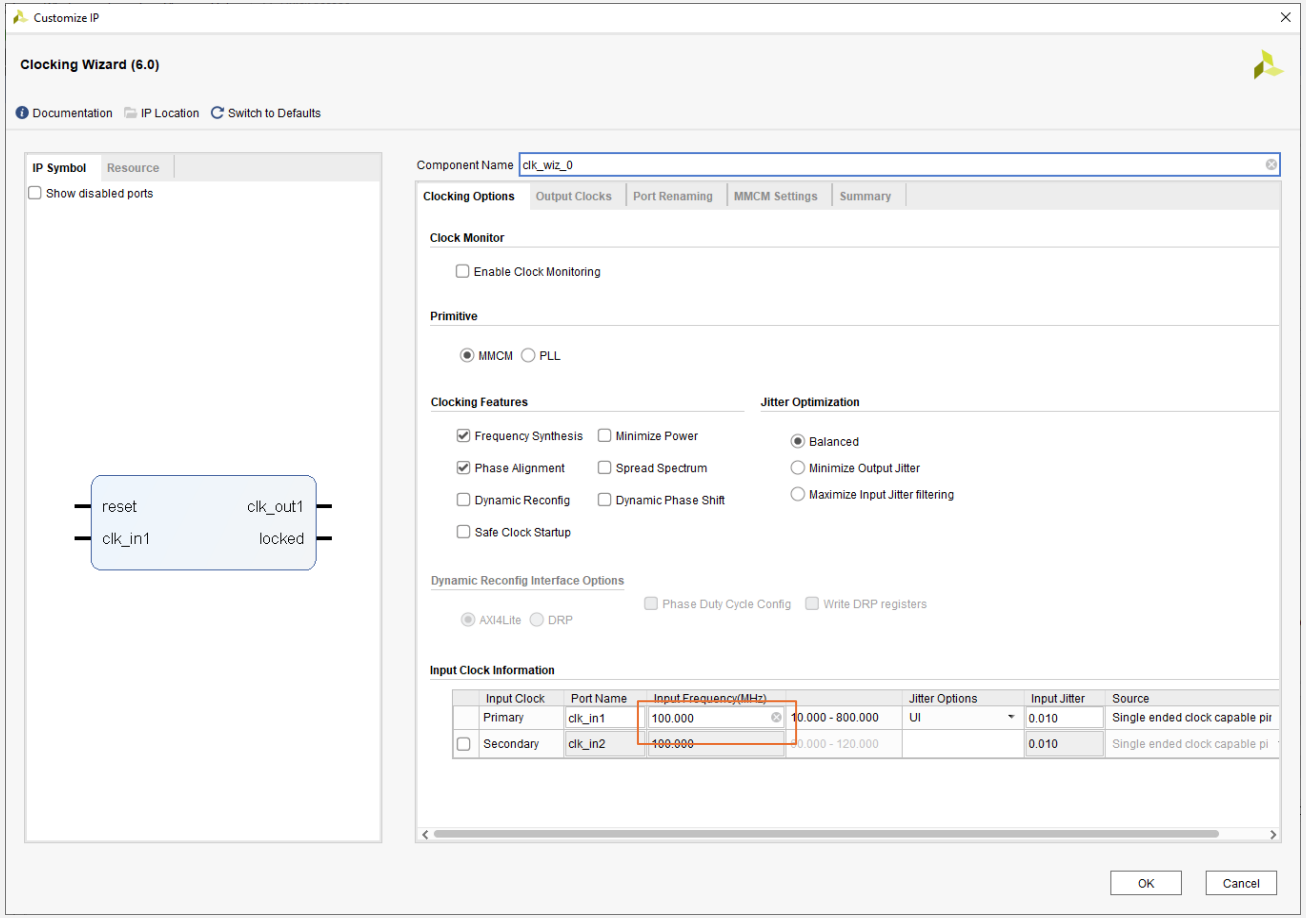
CoreGen Wizard

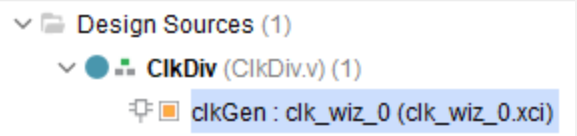
- The alternative is to use the CoreGen Wizard to instantiate a clock-tile which contains a PLL and can synthesise clocks radically different from the source crystal even fractional.
- Vivado -> IP Catalog -> clocking Wizard





CoreGen Wizard - Vivado





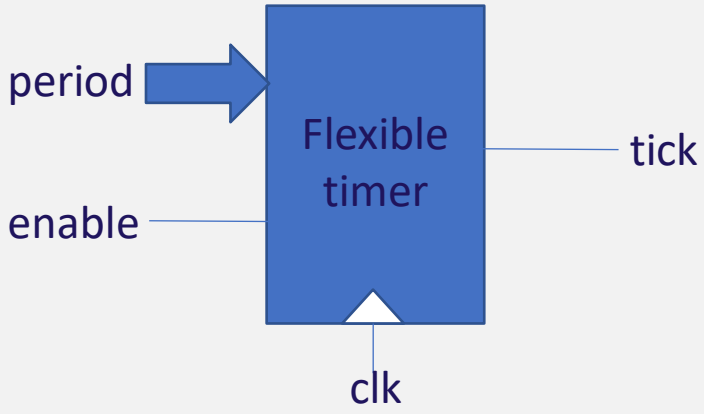

```
module ClkDiv(input clk_in, input rst, output clk_out);

    clk_wiz_0 clkGen(.clk_in1(clk_in),
                     .reset(rst),
                     .clk_out1(clk_out)
                     );

endmodule
```

Using an IP in the Design

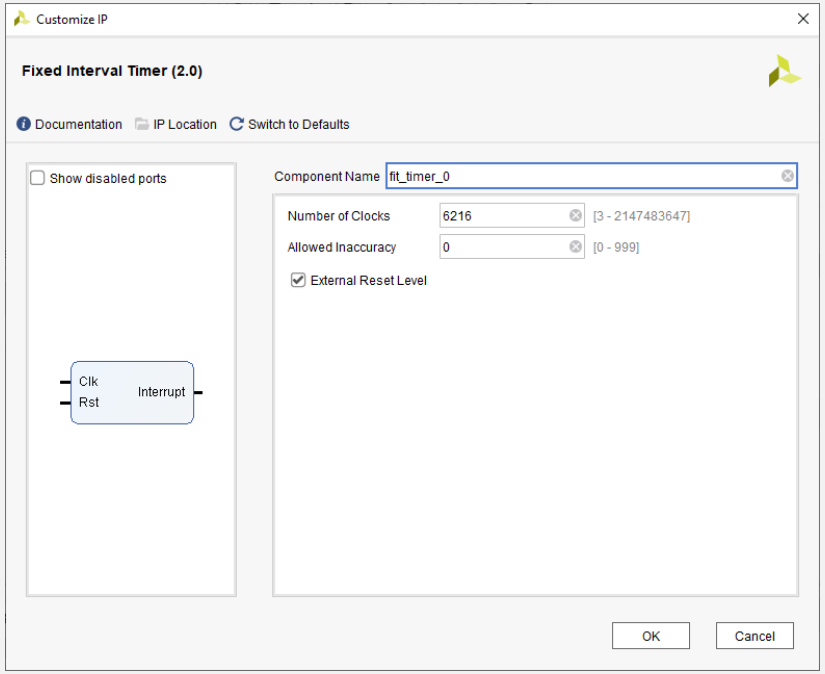
- IP is short for Intellectual Property
- It is supplied by the vendors
- Users can use them in their design if proper licenses have been purchased.
- It is vendor-depended! So if you include an IP in your design. You cannot directly copy your code.
- For different vendors, the IP with the same functionality probably have different names.

Xilinx: clock wizard, Altera: PLL (phase lock loop).

Timer

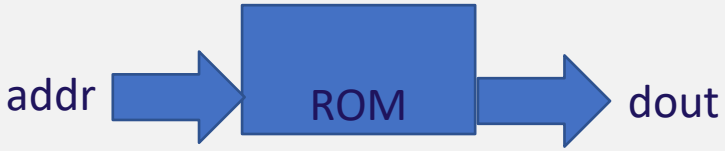


- Lets count down instead of up
- Generates a tick pulse at given period
- The period updates at end of each tick
- Xilinx IP: FLT




```
module FlexiTimer
 # (parameter WIDTH=16)
   ( input clk,
     input [WIDTH-1:0] period,
     output tick);
  reg count [WIDTH-1:0] = 0;
  always @(posedge clk)
       if (count==0) begin
         count=period;  tick=1'b1;
       end
       else  begin
         count <= count - 1; tick = 1`b0;
       end
endmodule
```

ROM

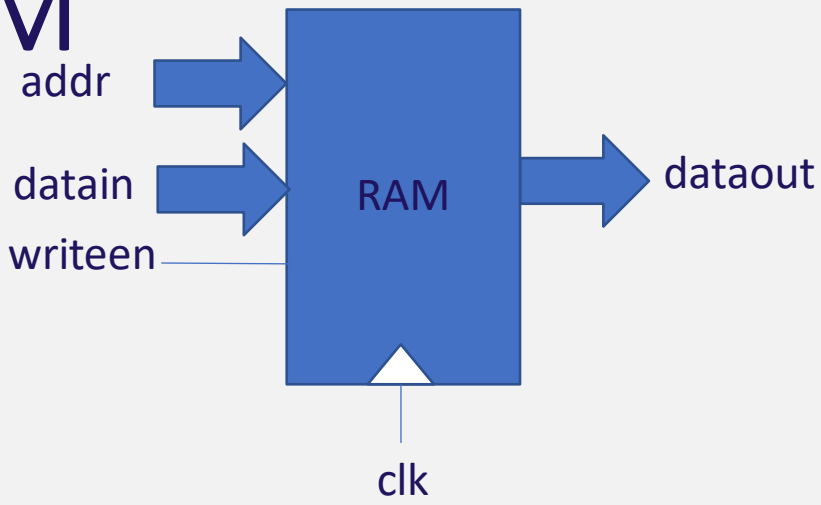


- The trick with ROM's is we often have the contents we want somewhere in some format so the problem becomes how to conveniently define this in an HDL. In Verilog we can load the contents conveniently from text files in binary or hexadecimal using the $readmemb or $readmemh system functions.

- The alternative is to define them as literal initialisation as part of a (constant) array.

```
// 512 off 20-bit values from hex file
reg [19:0] mydata [0:511];
initial $readmemh("rom.txt",mydata);
assign dout = mydata[addr];

/* alternative using serialise and
   concatenation operators */
reg [19:0] mydata [0:511] = {>>{ {
  20'h01234,   20'h232f5,  ....  } }};
assign dout = mydata[addr];
```

RAM

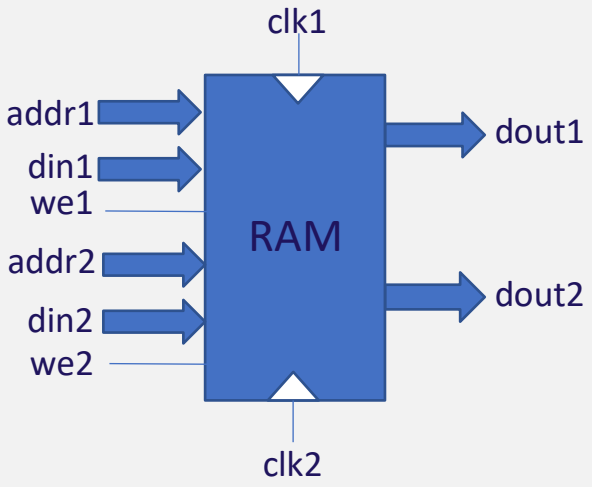



```
module memory
  #( ADDR_BITS=8, DATA_BITS=8 )
  (  output [DATA_BITS-1:0] dout,
     input [ADDR_BITS-1:0] addr,
     input [DATA_BITS-1:0] din,
     input writeen,  input clk );
   reg [DATA_BITS-1:0] memory [0:2**ADDR_BITS-1];
   always @(posedge clk)
       if (writeen) memory[addr] <= din;
   assign dout = memory[addr];
endmodule
```

```
entity SinglePortRam is
  Generic ( N : integer := 8; A : integer := 8 ); -- N=Data width A=address width
  Port ( clk, we : in std_logic;
       addr : in  std_logic_vector(A-1 downto 0);
       din    : in std_logic_vector(N-1 downto 0);
       dout : out std_logic_vector(N-1 downto 0) );
end SinglePortRam;

architecture Behavioral of SinglePortRam is
  type RAM_type is array (0 to 2**A-1) of std_logic_vector(N-1 downto 0);
  signal RAM : RAM_type;
begin
  process (clk) begin
    if rising_edge(clk) then
       if we='1' then
           RAM(conv_integer(addr)) <= din;  -- write data
      end if;
    end if;
  end process;
  dout <= RAM(conv_integer(addr));  -- read data
end Behavioral;
```

- Unlike a ROM it can be written to thus MUST be clocked. The readback can be unclocked or clocked to suit the application.
- In this version we have an address, write-enable, datain and dataout.
- Formally this is a write-first implementation but FPGA can do alternative read-first and nochange with/without registered output

Dual port RAM



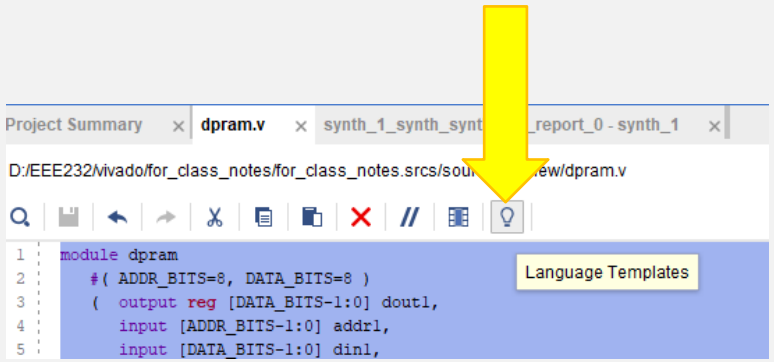
- There are various flavours such as twin port ram which has separate read and write addresses.
- This allows two simultaneous busses to access the same contents.
- To know which you have inferred best to check the synthesis report
- Here a true 2-clock dual port ram version and is described with no-change update rule.
- Which type of memory is best depends on your application and available resources.

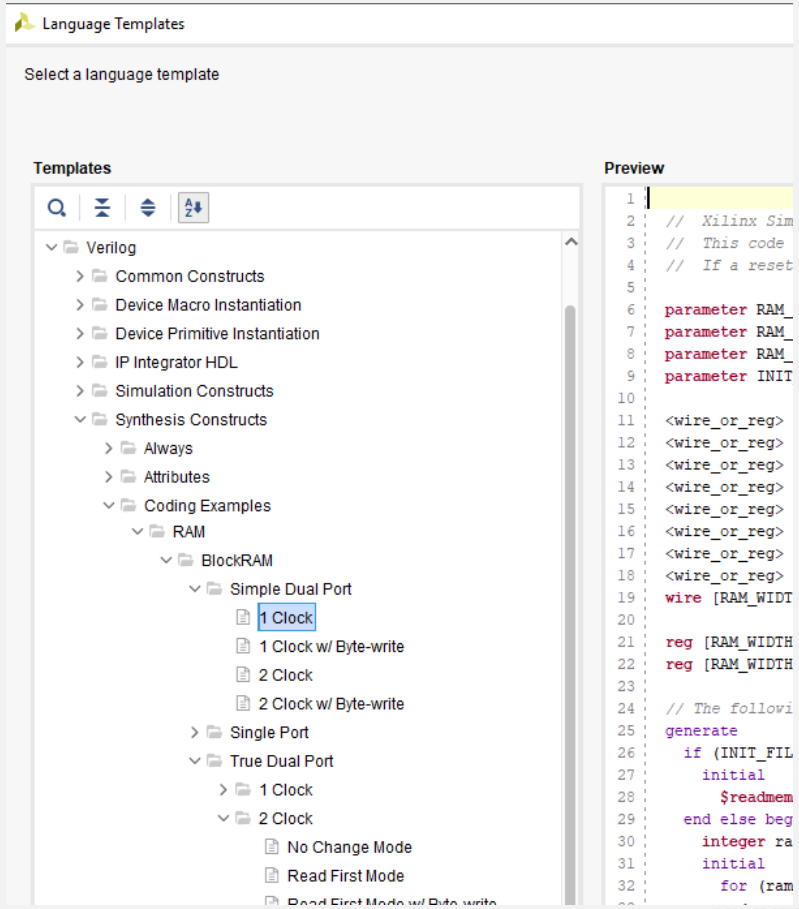
```
module dpram
  #( ADDR_BITS=8, DATA_BITS=8 )
  ( output reg [DATA_BITS-1:0] dout1,
    input [ADDR_BITS-1:0] addr1,
    input [DATA_BITS-1:0] din1,
    input we1,clk1,
    output reg [DATA_BITS-1:0] dout2,
    input [ADDR_BITS-1:0] addr2,
    input [DATA_BITS-1:0] din2,
    input we2, clk2 );
   reg [DATA_BITS-1:0] memory [0:2**ADDR_BITS-1];
   always @(posedge clk1)
        if (we1) memory[addr1] <= din1;
        else dout1 <= memory[addr1];
   always @(posedge clk2)
        if (we2) memory[addr2] <= din2;
        else dout2 <= memory[addr2];
endmodule
```

```
Block RAM: Final Mapping          Report
+-------------------+-------------------+-------------------------------------+---+---+-----------------------+----+----+----------------------------+--------+-------------+
|Module Name | RTL Object | PORT A (Depth x Width) | W | R | PORT B (Depth x Width) | W | R | Ports driving FF | RAMB18 | RAMB36 |
+-------------------+-------------------+-------------------------------------+---+---+-----------------------+----+----+----------------------------+--------+-------------+
|dpram        | memory_reg | 256 x 8(NO_CHANGE)      | W | R | 256 x 8(NO_CHANGE)      | W | R | Port A and B      | 1      | 0      |
+-------------------+-------------------+-------------------------------------+---+---+-----------------------+----+----+----------------------------+--------+-------------+
```

Which type of RAM

- To know what kind of memory you have created check the synthesis report
- This can be distributed or BLOCK memory and single, twin or dual port
- There are many alternatives. Fortunately the built in help will show you how to do this.
  - From the editor window click on the light bulb, then your language (eg Verilog) followed by Synthesis Constructs then coding examples
  - This gives you a language template to modify as you need and can save you LOTS OF TIME.





ROM&RAM using IPs

Customize IP

Distributed Memory Generator (8.0)

Documentation | IP Location | Switch to Defaults

Show disabled ports

a[5:0]
d[15:0]
clk
we
spo[15:0]

Component Name: dist_mem_gen_0

memory config | Port config | RST & Initialization

Options

Depth: 64 [16 - 65536]
Data Width: 16 [1 - 1024]

Memory Type

Memory Type

- ROM
- Single Port RAM
- Simple Dual Port RAM
- Dual Port RAM

OK | Cancel

Component Name: dist_mem_gen_0

memory config | Port config | RST & Initialization

Load COE File

The initial memory content can be set by using a COE file.This will be passed to the core as a Memory Initialisation File (MIF).

Coefficients File: no_coe_file_loaded

COE Options

Default Data : 0 Radix : 16

Reset Options

Reset QSPO | Reset QDPO

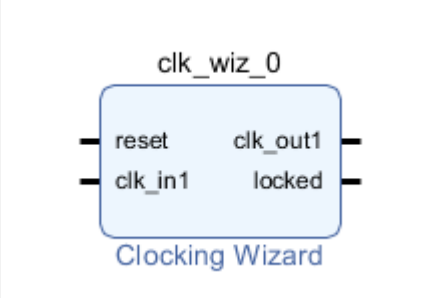
Synchronous Reset QSPO | Synchronous Reset QDPO

ce overrides

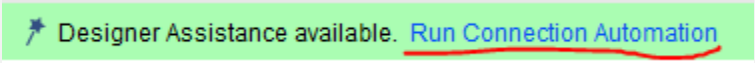
CE Overrides Sync Controls | Sync Controls Overrides CE

IP Integration – block design

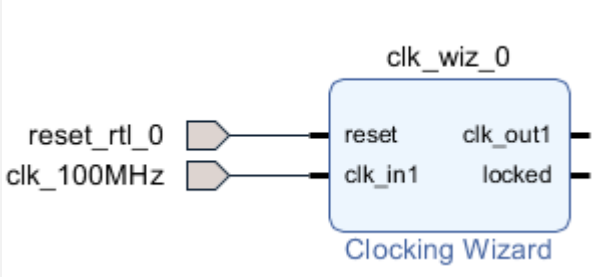
- Create block design in the IP INTEGRATOR
- Click ADD IP button
- Search for Clock Wizard



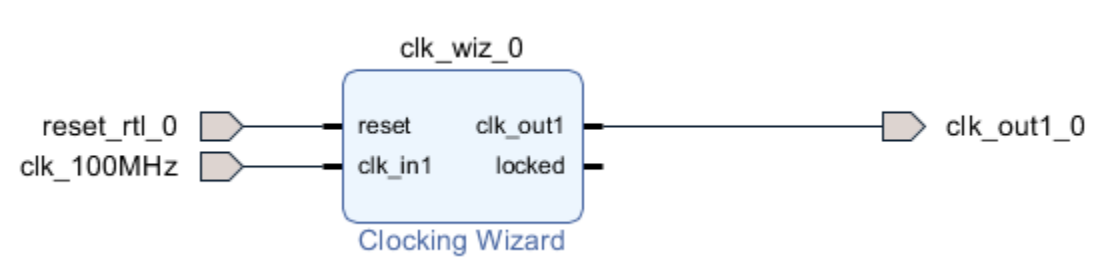
You will be able to see



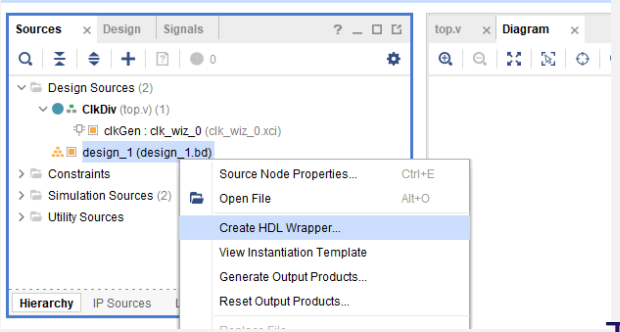
Click Run Connection Automation



You will see that the input has been connected



Then we manually connect the output



Create HDL Wrapper

```
module design_1_wrapper
   (clk_100MHz,
    clk_out1_0,
    reset_rtl_0);
  input clk_100MHz;
  output clk_out1_0;
  input reset_rtl_0;

  wire clk_100MHz;
  wire clk_out1_0;
  wire reset_rtl_0;

  design_1 design_1_i
       (.clk_100MHz(clk_100MHz),
        .clk_out1_0(clk_out1_0),
        .reset_rtl_0(reset_rtl_0));
endmodule
```

Then you can see the generated wrapper

Design Choice & Critical Path Mitigation

Setting tool strategies

- Modern tool offer a large number of options for setting optimisation goals and effort. These at first order prioritise Power, Area or Performance.
- The second level is then to resolve typical FPGA issues around routing congestion or to tweak the priority to focus on a more specific part of the overall performance (eg minimise net delays).
- You only need to be aware that such options exist and tools do much of the optimisation automatically prioritising power, area or time.

Vivado Implementation Defaults
Performance_Explore
Performance_ExplorePostRoutePhysOpt
Performance_ExploreWithRemap
Performance_WLBlockPlacement
Performance_WLBlockPlacementFanoutOpt
Performance_EarlyBlockPlacement
Performance_NetDelay_high
Performance_NetDelay_low
Performance_Retiming
Performance_ExtraTimingOpt
Performance_RefinePlacement
Performance_SpreadSLLs
Performance_BalanceSLLs
Performance_BalanceSLRs
Performance_HighUtilSLRs
Congestion_SpreadLogic_high
Congestion_SpreadLogic_medium
Congestion_SpreadLogic_low
Congestion_SSI_SpreadLogic_high
Congestion_SSI_SpreadLogic_low
Area_Explore
Area_ExploreSequential
Area_ExploreWithRemap
Power_DefaultOpt
Power_ExploreArea
Flow_RunPhysOpt
Flow_RunPostRoutePhysOpt
Flow_RuntimeOptimized
Flow_Quick

Creating effective constraints

- The tools need constraints to operate well, and more recently Xilinx have disabled outputting performance information unless you have defined at least a clock constraint.
- Constraints are used to inform the tools of the designers intent. These are in terms of timing, physical area of chip to be used for some module, locking the I/O pin assignments, etc.
- Xilinx pull all of the constraints together in a single text file (.XDC). Xilinx Constraint Format is a language to specify all the constraints for a design. It is described in detail later. Other vendors use a similar approach albeit the syntax varies slightly the key concepts are the same.
- Over the next few slides the different areas for ‘constraints’ are described.

XDC constraints

- Mostly the timing wizard and IO planner can be used without needing to resort to editing the XDC file directly. (Formally XCF)
- But there are exceptions so familiarity with XDC is recommended.
- -dict{ } allows specifying more than one property for either top level.
- Searches for matching names for either top level ports (get_ports) or internal nodes (get_pins). *Yes, these are confusingly defined (not by me!)*

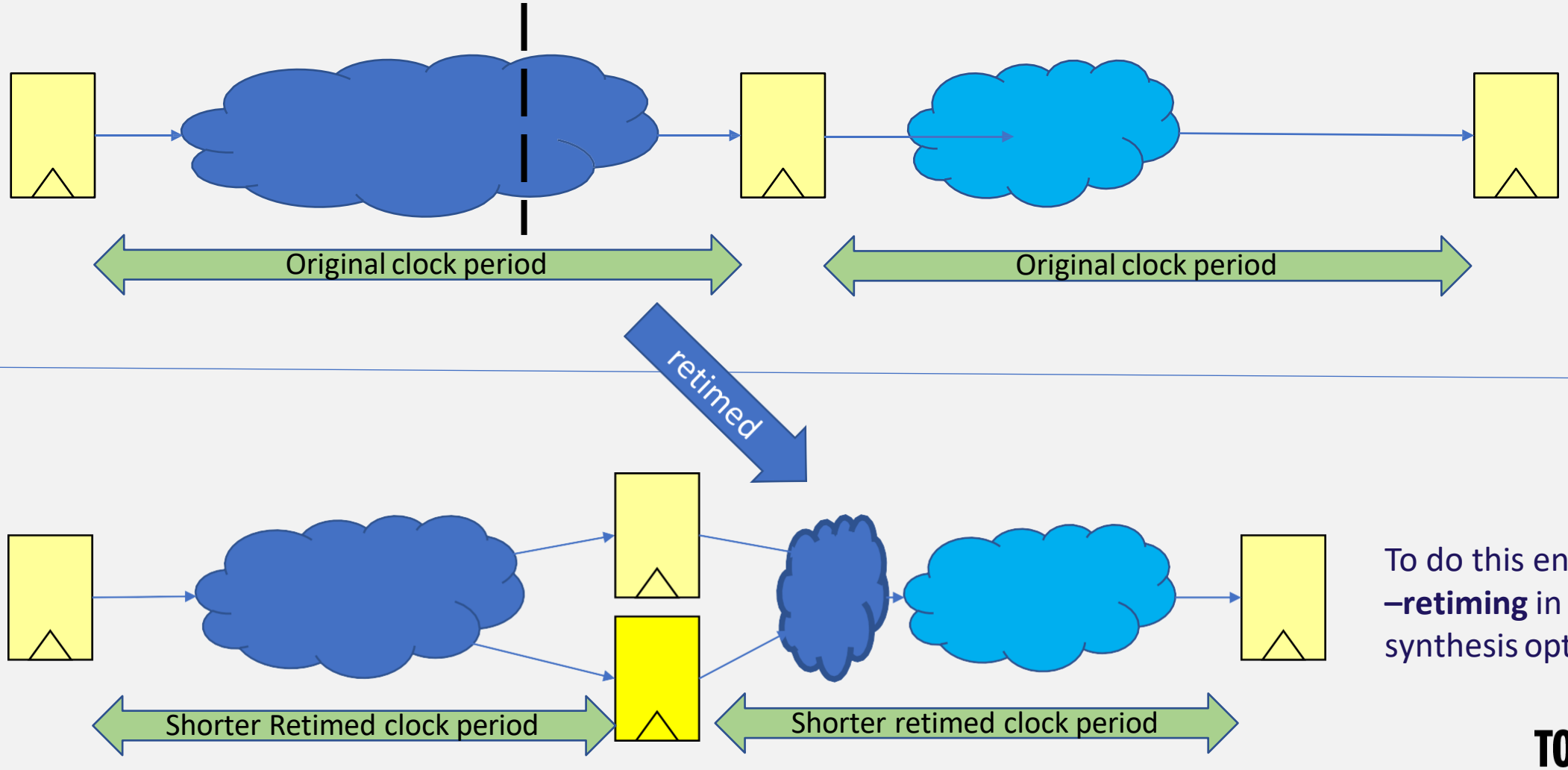
```
# An example XDC file

# constrain clock
set_property -dict {PACKAGE_PIN E3 IOSTANDARD LVCMOS33} [get_ports CLK100MHZ]
create_clock -period 10.000 -name sys_clk_pin
                -waveform {0.000 5.000} -add [get_ports CLK100MHZ]

# fix I/O to specific pins and signal levels
set_property -dict {PACKAGE_PIN H17 IOSTANDARD LVCMOS33} [get_ports {LED[0]}]
set_property -dict {PACKAGE_PIN K15 IOSTANDARD LVCMOS33} [get_ports {LED[1]}]
set_property -dict {PACKAGE_PIN J13 IOSTANDARD LVCMOS33} [get_ports {LED[2]}]

# set output drive strength if needed
set_property DRIVE 4 [get_ports { PWM[*]}];

# allow for specific multicycle paths if needed (here 12 cycle), names from timing report
set_multicycle_path 12 -setup -from [get_pins {slider/areg[*]/C}]
                              -to   [get_pins {slider/breg[*]/D}]
set_multicycle_path 11 -hold  -from [get_pins {slider/areg[*]/C}]
                              -to   [get_pins {slider/breg[*]/D}]
```

Vivado Constraints Guide: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug903-vivado-using-constraints.pdf

Critical path mitigation

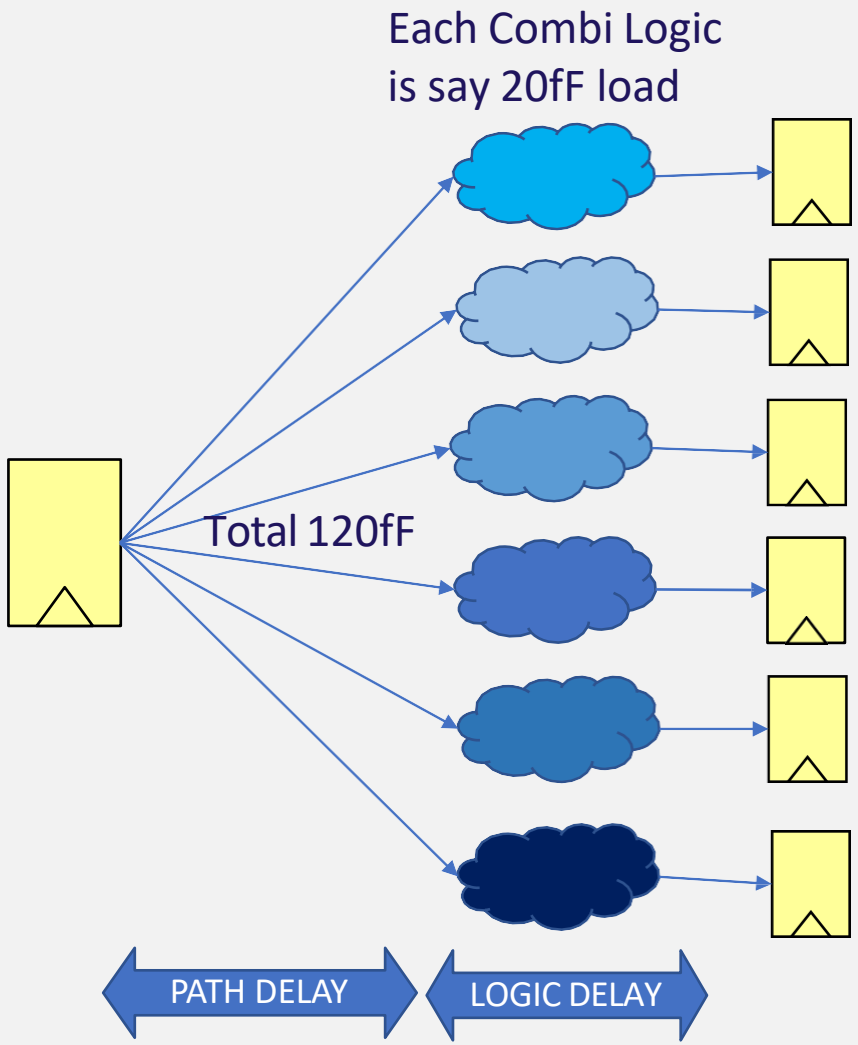
- In many designs there will be a very few areas of the design which have relatively slow paths. If unchecked this forms the critical path and thus sets the maximum possible clock rate for the design.
- There are a number of options available to fix these paths:
- Register balancing/duplication (retiming) – if enabled this is done automatically by the tools by moving 'slow' functions either side of registers or adding duplicate registers to break excessive fan out. However, there needs to be sufficient registers / cycles in the datapath for the tool to have freedom to operate. It wont change the overall number of cycles in the datapath.
- Multi-cycle paths – if the result of some combinatorial logic isn't required in the next clock cycle after its input was defined, the designer can tell the tool which cycle it is required by thus it has multiple-clock cycles to stabilise to its final (valid) value before being clocked in.

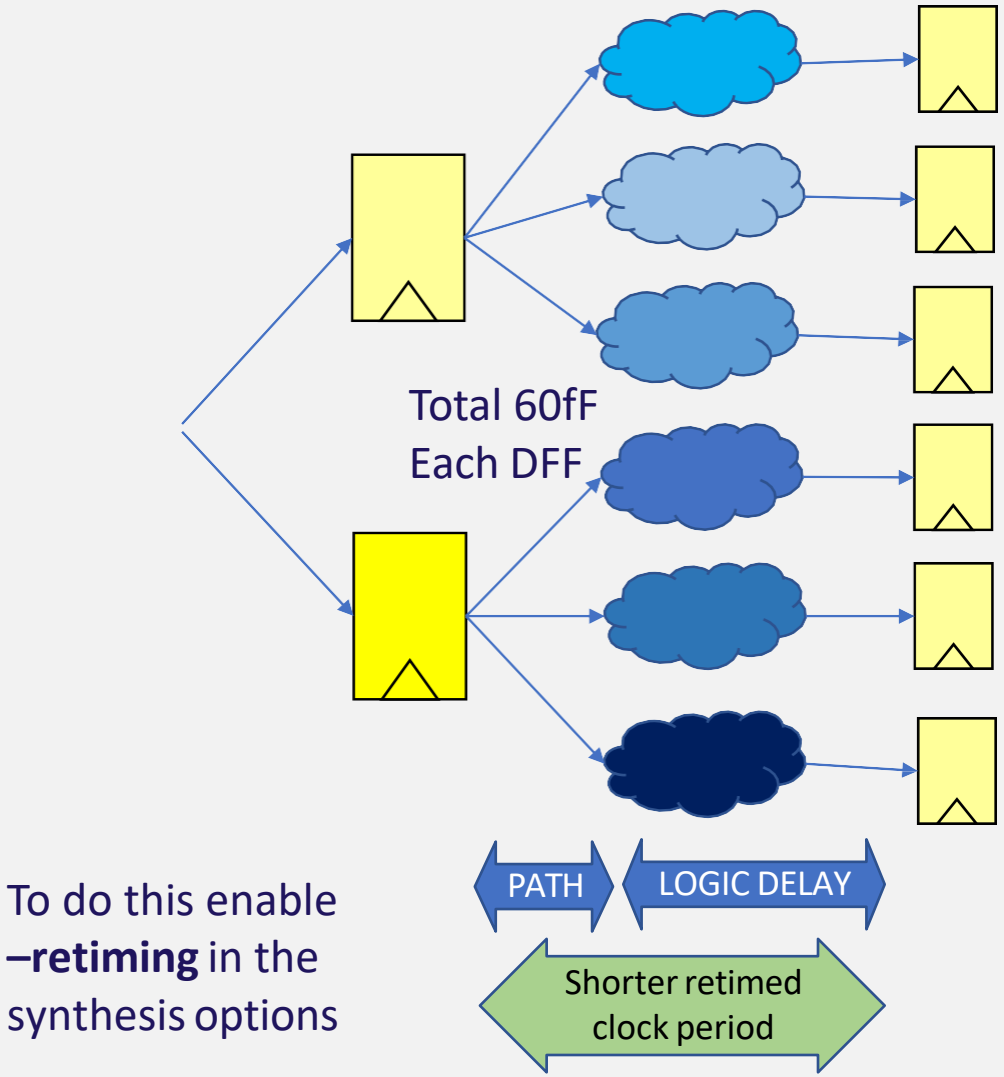
Register Balancing Example



To do this enable **–retiming** in the synthesis options

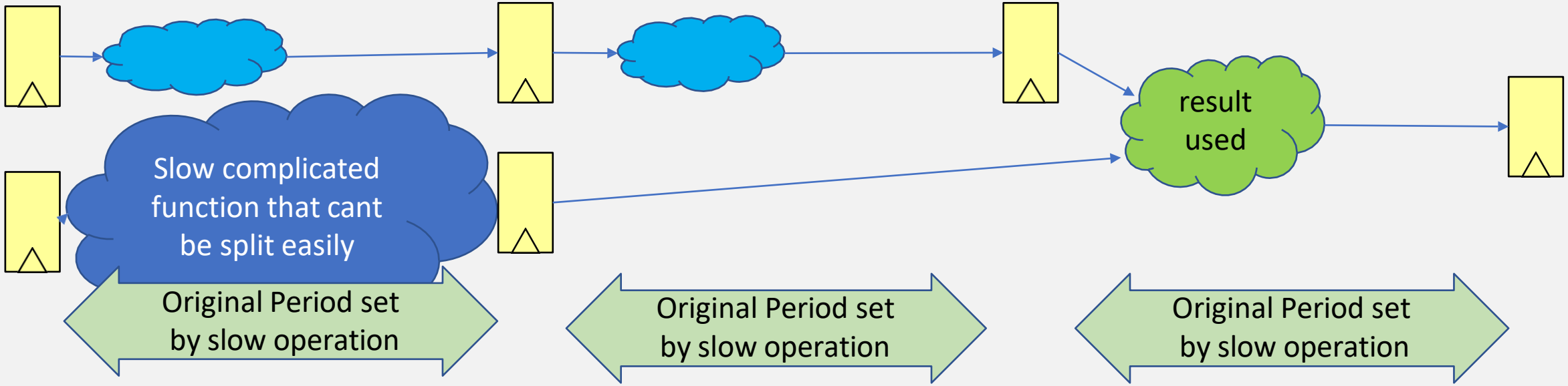
Register Duplication Example

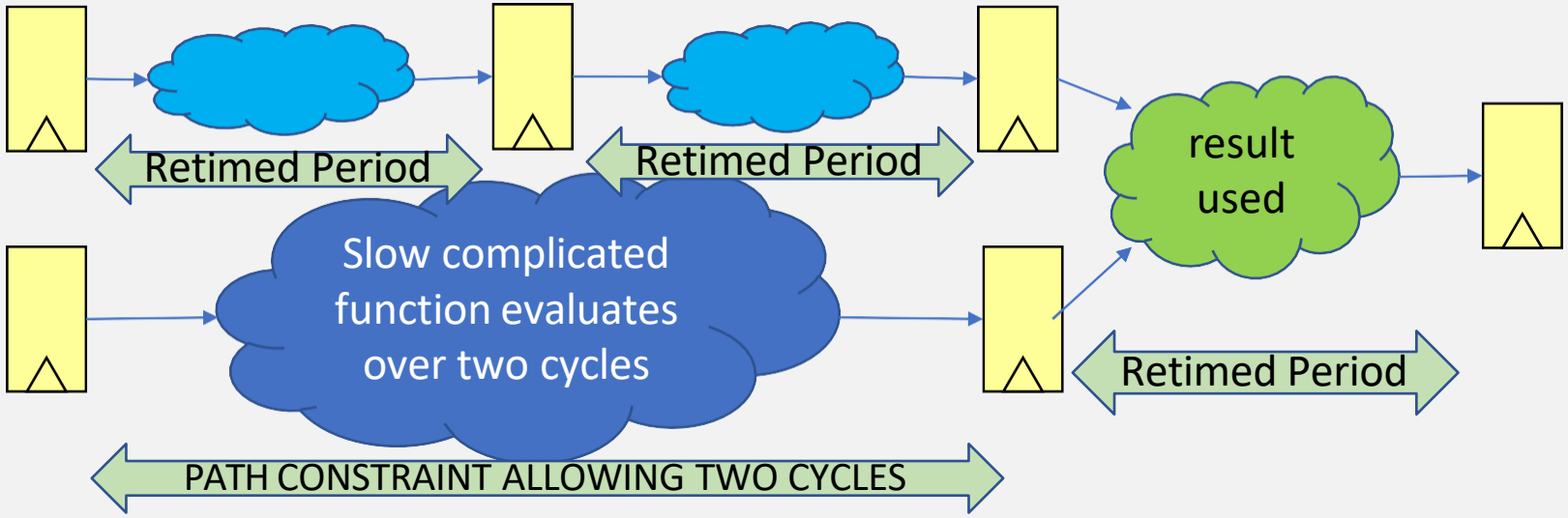




To do this enable **–retiming** in the synthesis options

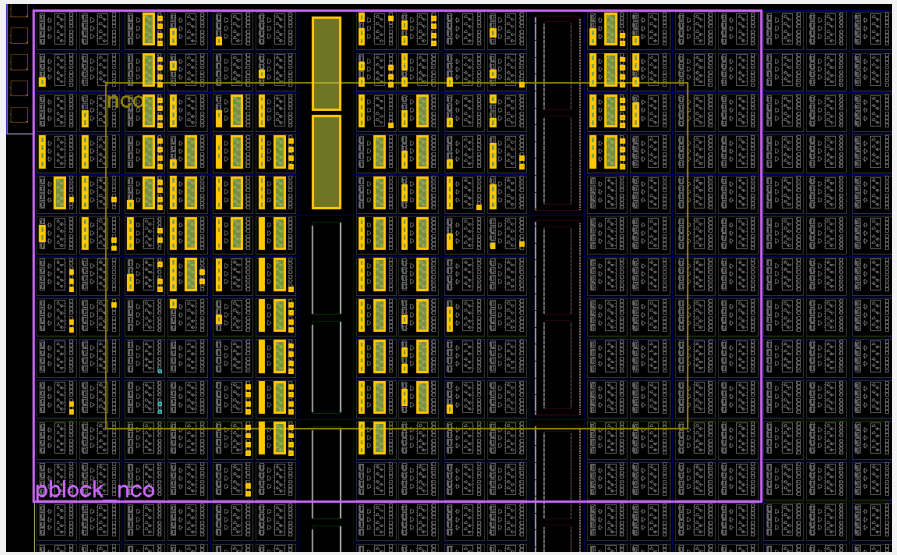
Multi Cycle Path Example





This is done by adding a **set_multicycle_path** statement in the XDC file

Floor planning



- With large designs the execution time for the tool to carry out the placement and routing would be excessive as it is a NP hard problem.
- We simply this by breaking down our design along its hierarchy by setting area constraints on various modules so the P&R has a set of much simpler problems to solve.
- In Vivado open the implemented design, then from the netlist window choose a module within your design and “floorplan” then “Create P block”. You can then draw a rectangle to constrain the locations for this module.

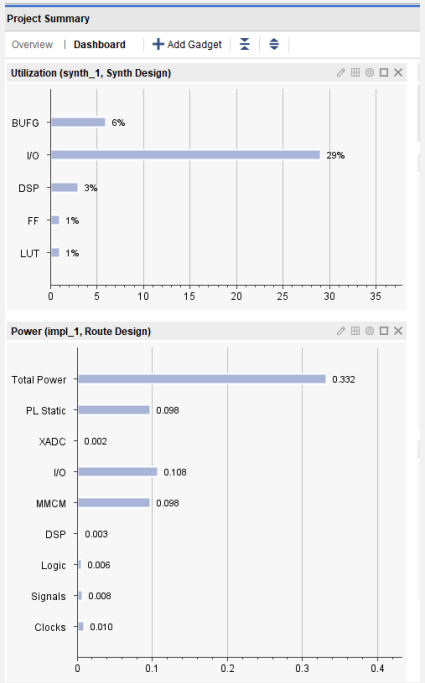
Vivado Constraints Wizard

- This tool (under synthesis menu) will help the designer describe the complete set of **timing constraints** for their design and pick up any which are missing. It will create the XDC for you
- Although, it is still up to the designer to put in correct / valid information. Don't guess, please refer to external component datasheets where needed.
- This is particularly important when internally multiplying up an external crystal fed clock to a higher frequency, otherwise the PLL in the FPGA can go unstable with unexpected clock jitter.

Evaluate your Design

University of Sheffield

Power Performance Area (PPA)



- When evaluate a piece of hardware, we usually see if we have a good PPA trade-off.
- Power – Can be found in Open Implemented Design -> Report Power
- Area – Can be found in Open Implemented Design -> Report Ultilization
- Performance – This is bit tricky
  - Throughput & Latency
- You need to find out the max clock frequency you can have by finding out the time of covering the critical path. For example, if the critical path takes at least 20ns, you obviously cannot use 10ns (100MHz) clock frequency but have to use 50MHz
- To find out this, a post Simulation is always recommended as it takes all the physical conditions into the simulation.
- Then you will be able to know how many outputs you have within 1 second if you keep feeding inputs. This is called the **throughput.**
- The time you wait from your first input to the first output, is called the latency

TOP 100 UNIVERSITY

University of Sheffield

Power Performance Area (PPA)

- Based on the requirements of the application or your design goal, it's always good to know if the following 2 factors meets the design specs.
  - Performance/Area Efficiency – How much speed you have within 1 LUT
  - Performance/Power Efficiency – How much speed you have within 1 W
- If we look at a publication in the top journal, you will know the commonly used evaluation points
  - Latency
  - Area
  - Power
  - Throughput
  - Energy Efficiency
  - Area Efficiency
  - Max Frequency

COMPARISON WITH PREVIOUS WORKS BASED ON ASICS

Design	Node (nm)	Frequency (MHz)	Latency (ns)	Area (mm^2)	Voltage (V)	Power (mV)	Throughput (GFLOPS)				Energy Efficiency (GFLOPS/W)			
							FP64	FP32	FP16	INT8	FP64	FP32	FP16	INT8
ARAB [17]	90	357	8.4	0.082	1.0	60.0	-	1.43	1.43	-	-	11.41	11.41	-
arXiv [27]	28	1360	4.41	0.018	0.8	18.38	1.36	1.36	-	-	43.70	110.00	-	-
TCOM [22]	90	667	4.5	0.795	1.0	43.8	0.67	1.33	2.67	-	30.44	60.88	121.77	-
TVLSI [28]	22	923	3.25	0.049	0.8	57.41	0.92	1.85	3.69	-	**74.83**	199.70	497.67	-
DATE [12]	28	1351	2.96	0.0126	1.0	38.93	1.35	6.76	27.0	-	32.41	173.25	748.67	-
TCAS [13]	28	1351	2.96	0.0184	1.0	51.4	**2.70**	**10.8**	**43.2**	-	52.8	216.7	814.4	-
TVLSI [15]	28	1429	**2.8**	0.013	1.0	29.3	0.71	3.57	14.29	-	48.76	**243.78**	**975.13**	-
TCAS [11]	28	**1471**	-	**0.010**	1.0	**15.86**	-	2.94*	13.24**	27.94	-	185.41*	834.35**	1761.41
Prop.	28	971	11.33	0.0276	1.0	39.0	-	7.76	23.28	**69.84**	-	199.15	597.5	**1792.4**

* [11] supports TP32 format, the corresponding Throughput is 13.24 GFLOPS, and Energy Efficiency is 834.35 GFLOPS/W.

** [11] supports BP16 format, the corresponding Throughput is 27.94 GFLOPS, and Energy Efficiency is 1761.41 GFLOPS/W.

TOP 100 UNIVERSITY

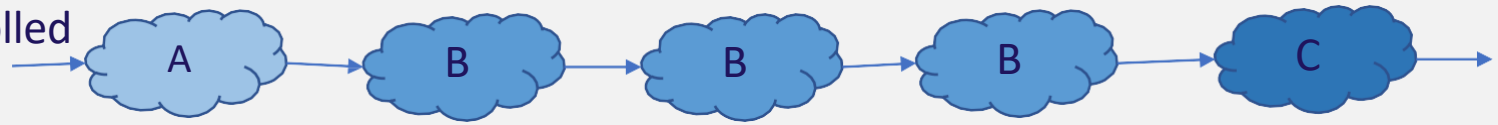
Algorithmic improvements

- The tools don't do this – yet – or likely in the near future so us engineers are still in a job for a while ;-)
- Need to quickly assess how a candidate design performs vs our customer's design constraints
- Assess different alternatives and supply the best one
- Ultimately the decision is an economic one: time to market, customer needs, design team costs, managing risk, etc…
- Typically the designer is comparing performing the required operations sequentially over a number of cycles with a datapath versus performance gains from using more area / parallel operation

Algorithmic Trade offs

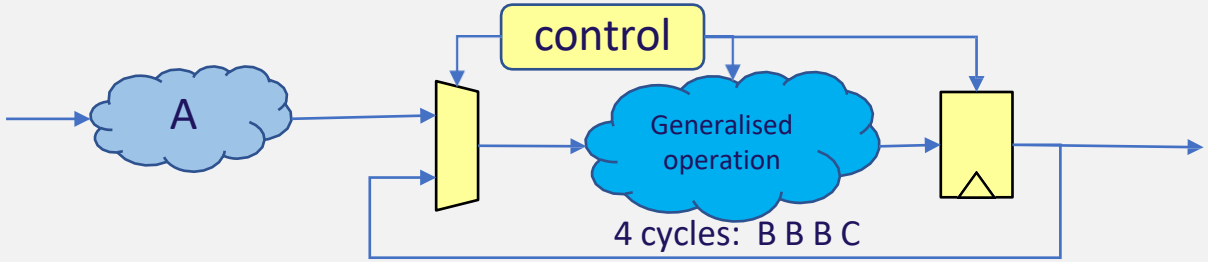
Set of simpler operations performed sequentially, some are repeated

Unrolled



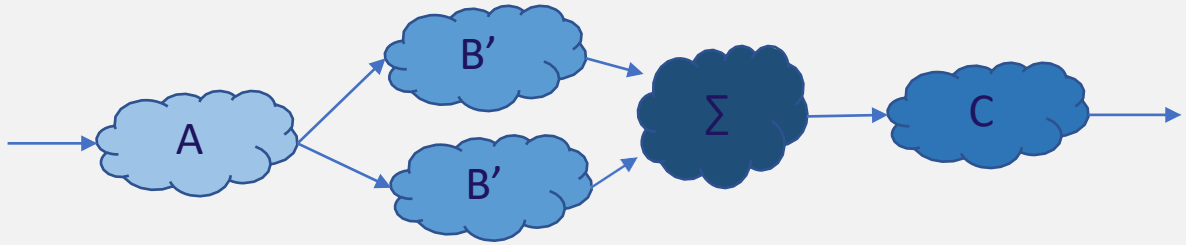
The unrolled baseline version of the design. All operations wired together one after each other. Slowest and large area

Iterative



If possible write a generalised datapath to do operations, here B&C iterate over a number of cycles, here 4 to give required result. Low area moderate speed

Parallel



Look at redefining operator to break dependency so operations can be done in parallel. Example shows redefined B' in parallel. The design can then be further pipelined or iterated.

Pipelined

Registers between operations allowing shorter clock period and higher throughput with new data every clock cycle. Improves throughput at expense of latency