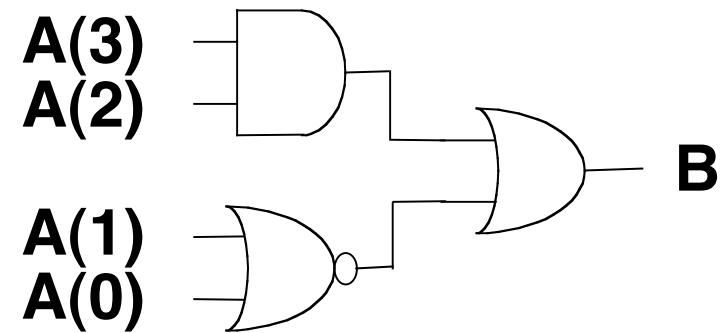


'Real World' VHDL

Introduction

- Previous lecture(s) discussed the digital design flow and also the basic concepts of VHDL
 - Known by students as the ‘boring stuff’!!
- Remaining lectures on VHDL will build on the previous material to introduce all the language features used to implement real designs
 - Only a small subset of the VHDL language is used to build real systems

But first...



```
entity COMB_LOGIC is
    port ( A: in STD_LOGIC_VECTOR (3 downto 0);
          B: out STD_LOGIC );
end COMB_LOGIC ;
```

Behavioural VHDL

```
architecture MY_BEHAVIOUR of COMB_LOGIC is
  signal C1, C2: STD_LOGIC;
begin
  process (A)
  begin
    C1  <= A(4) and A(3);
    C2  <= A(1) nor A(0);
    B   <= C1 or C2;
  end process;
end COMB_LOGIC ;
```

**The order of these is
critically important !**



Structural VHDL

```
architecture MY_STRUCT of COMB_LOGIC is
```

```
signal C1, C2: STD_LOGIC;
```

```
component AND port (I0, I1: in STD_LOGIC; O: out STD_LOGIC); end component;
```

```
component OR port (I0, I1: in STD_LOGIC; O: out STD_LOGIC); end component;
```

```
component NOR port (I0, I1: in STD_LOGIC; O: out STD_LOGIC); end component;
```

```
begin
```

```
    U1: AND port map (A(3), A(2), C1);
```

```
    U2: NOR port map (A(1), A(0), C2);
```

```
    U3: OR port map (C1, C2, B);
```

```
end COMB_LOGIC ;
```



**Order unimportant,
concurrent assignment**

Dataflow VHDL - a,b,...

```
architecture MY_DATAFLOW of COMB_LOGIC is
```

```
signal C1, C2: STD_LOGIC;
```

```
begin
```

```
    C1  <= A(4) and A(3);
```

```
    C2  <= A(1) nor A(0);
```

```
    B   <= C1 or C2;
```

```
end COMB_LOGIC ;
```

```
begin
```

```
    with A select
```

```
        B <= '1' when "0000",
```

```
        '1' when "0100" | "1000",
```

```
        '1' when "1100" | "1101",
```

```
        '1' when "1110" | "1111",
```

```
        '0' when others'
```

```
end COMB_LOGIC ;
```

Dataflow VHDL - ...c

```
architecture MY_DATAFLOW of COMB_LOGIC is
```

```
signal C1, C2: STD_LOGIC;
```

```
begin
```

```
    C1 <= '1' when A(4)='1' and A(3)='1' else '0' ;
```

```
    C2 <= '1' when A(1)='0' and A(0)='0' else '0' ;
```

```
    B <= C1 or C2;
```

```
end COMB_LOGIC ;
```

Real World VHDL

- Previous VHDL examples have shown (simple) examples of gate level designs
- This is really the basis behind any system from simple to highly complex
- However, working at gate level gets complicated- so VHDL has a rich syntax to allow us to model things much nearer the behaviour of the system
 - Just as we don't write software in assembly code !
- But, how do we translate this into gates?
 - This is termed *synthesis* and is the clever part !

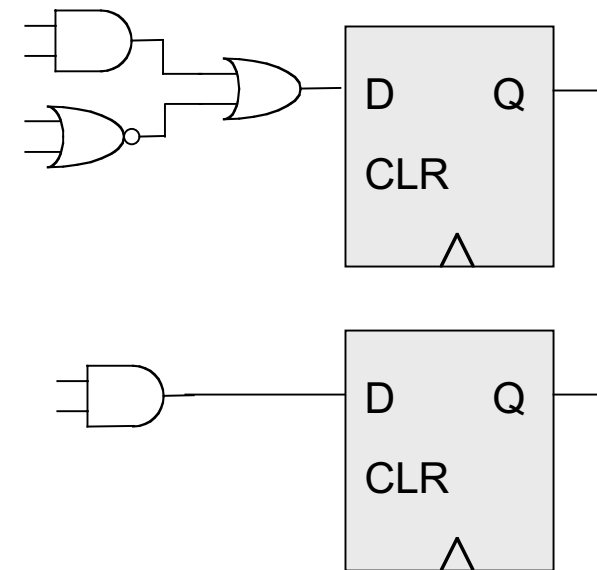
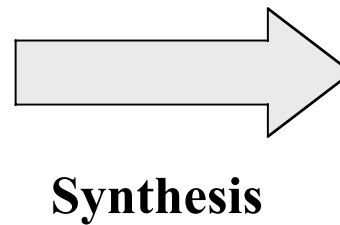
Register Transfer Level (RTL)

- We will talk about Register Transfer Level synthesis
- So, what does this mean ??
 - We are going to design our system in terms of registers (Flip-Flops) and the transfer functions between these registers (Combinatorial Logic)
 - We will describe the internal architecture of the system using control flows and registers (like a state machine etc.)
- Consequently, we will have to decompose systems to primitive state machines, dataflows, counters, etc.
- More on RTL later

Logic Synthesis- I

- Synthesis takes some high-level coding of a system (e.g. VHDL code) and automatically generates a gate implementation for this system

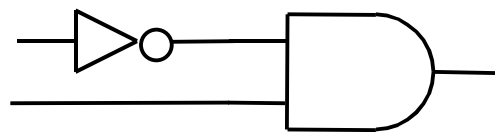
```
architecture MYARCH of ~~~~~ is
begin
    ~~~~~
    ~~~~~
end MYARCH;
```



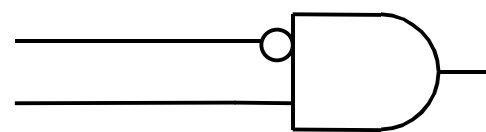
- May be undertaken in several stages- but most tools nowadays have a single flow

Logic Synthesis- II

- For each target architecture, a *synthesis library* exists
- For example, each semiconductor fabrication process will have its own synthesis technology library
 - This details all the primitive cells that exist in the technology
 - For example, AND gate, OR gate, Multiplexer etc.
- Performs optimisation given cells that exist in the synthesis library (see later examples)



2 Logic Gates



1 Logic Gate

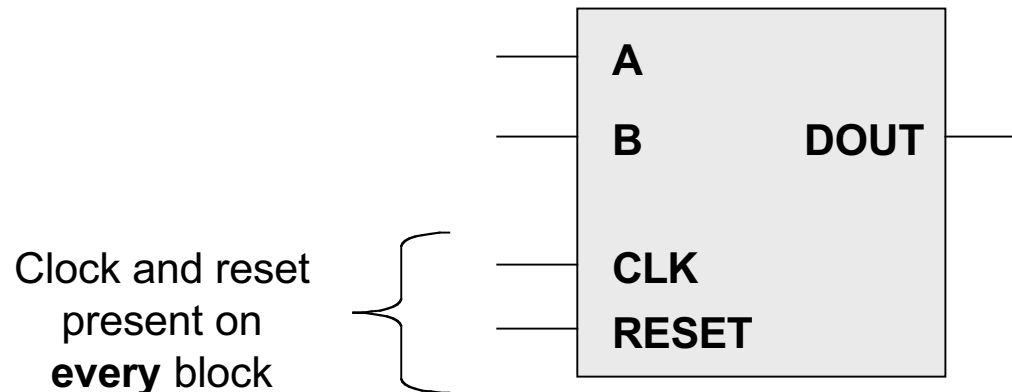
Synchronous Design Methodology- I

- All blocks in our design have a common clock and reset
 - See **schematic 1** for an example
 - Typically reset is asynchronous (active low or high)
 - System only responds to inputs changing on clock
- Has the following advantages
 - The system starts up in a know state
 - Changes in system inputs are localised.
 - Facilitates testing of the final IC (see later lecture on testing)
- **We will use this for all designs !!!**

Synchronous Design Methodology- II

- What does the VHDL look like ?

```
entity CLKDSYS is  
  port (A,B: in std_logic;           -- system inputs  
        DOUT: out std_logic;         -- system outputs  
        CLK,RESET: in std_logic);   -- clock and reset  
end CLKDSYS;
```



Synchronous Design Methodology- III

- Architecture definition

```
architecture TRIAL of CLKDSYS is
```

```
begin
```

```
process (CLK, RESET)
```

```
begin
```

```
if RESET='1' then -- reset (asynchronous)
```

```
    C <= '0' ; -- reset outputs to default state (1 or 0)
```

```
else if CLK' EVENT and CLK='1' then -- rising edge of clock
```

```
    DOUT <= A or B;
```

```
end if;
```

```
end process;
```

```
end TRIAL;
```

Only 1 Process !!



Reset phase



All outputs etc. only change on clock edge

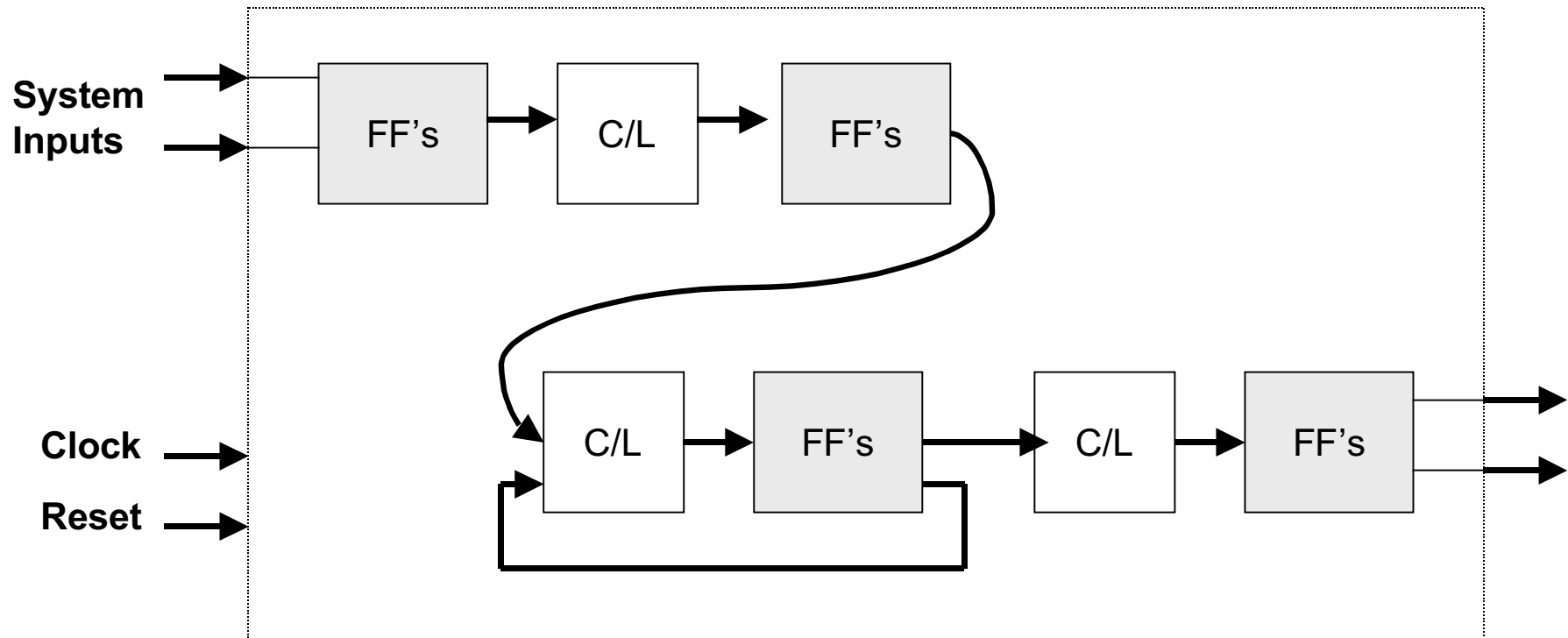


Compare with DD2 code

```
architecture BEHAVIOUR of SIMPLE_COUNTER is
  signal Count_Present, Count_Next: unsigned (2 downto 0);
begin
  process (CLK, RESET)
begin
  if RESET='1' then Count_Present <= "000"; \-- reset
  else if Rising_edge(CLK) then -- rising edge of clock
    Count_Present <= Count_Next;
  end if;
end process;
  process (Count_Present)
begin
    work out what the value of Count_Next should be;
end process;
end SIMPLE_COUNTER;
```

Synchronous Design Methodology- IV

- Using this approach, after synthesis of (any) design resultant system as follows:
 - C/L = Combinatorial Logic (And, Or, etc), FF's= Flip-Flops




Control Flow- I

- We can use IF-THEN-ELSE as a **prioritised** selector
- General form:

```
if <condition> then  
    <statements>;  
elsif <condition>  
    <statements>;  
else  
    <statements>;  
end if;
```

There can be multiple
elsif statements



Control Flow- II

- Here's an example of a 2-1 priority encoder (synch.):

```
architecture BEHAVIOR of PENCODER is
```

```
  process (CLK, RESET)
```

```
  begin
```

```
    if RESET='1' then
```

```
      DATAOUT <= '0' ;
```

```
    elsif CLK' event and CLK='1' then
```

```
      if DATAIN(0) = '1' then
```

```
        DATAOUT <= '0' ;
```

```
      elsif DATAIN(1) = '1' then
```

```
        DATAOUT <= '1' ;
```

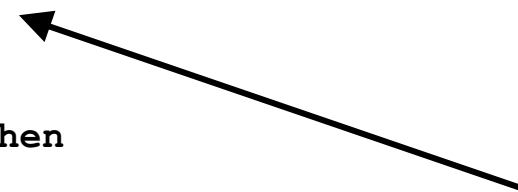
```
      end if;
```

```
    end if;
```

```
  end process;
```

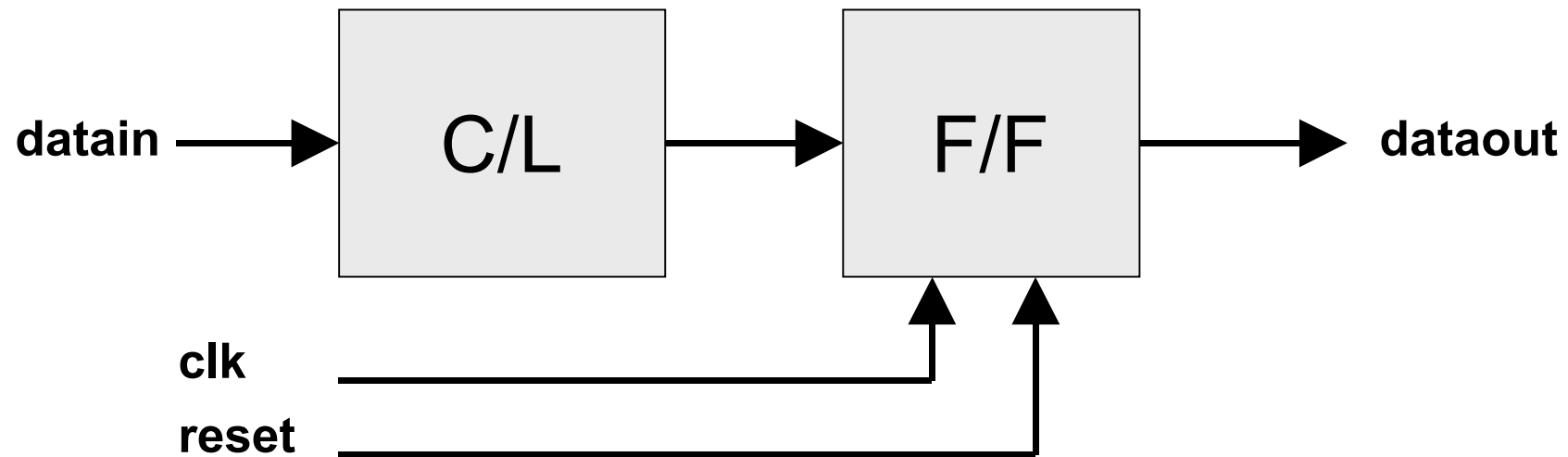
```
end BEHAVIOR;
```

DATAIN(0) has
higher priority than
DATAIN(1)



Control Flow- III

- Resultant system (see handout) after synthesis (targeting a Xilinx 4000E FPGA)
 - Exercise- derive the truth table and check it works !

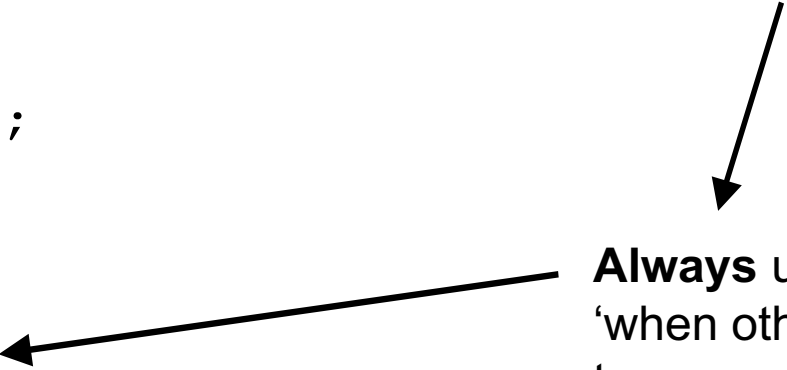


Control Flow- IV

- We can use CASE as a equal priority selector
 - So, inputs are *mutually exclusive*
- General form:

```
case <expression> is  
  when <choice> =>  
    <statements>;  
  when <choice> =>  
  ...  
  when others =>  
    <statements>;  
end case;
```

Be careful here !



Always use
'when others =>'
to mop up any other
conditions

Control Flow- V

- Here's an example of a multiplexer

```
architecture BEHAVIOR of MUX is
  process (SEL, DATAIN)
  begin
    case SEL is
      when '0' =>
        DATAOUT <= DATAIN(0);
      when others =>
        DATAOUT <= DATAIN(1);
    end case;
  end process;
end BEHAVIOR;
```

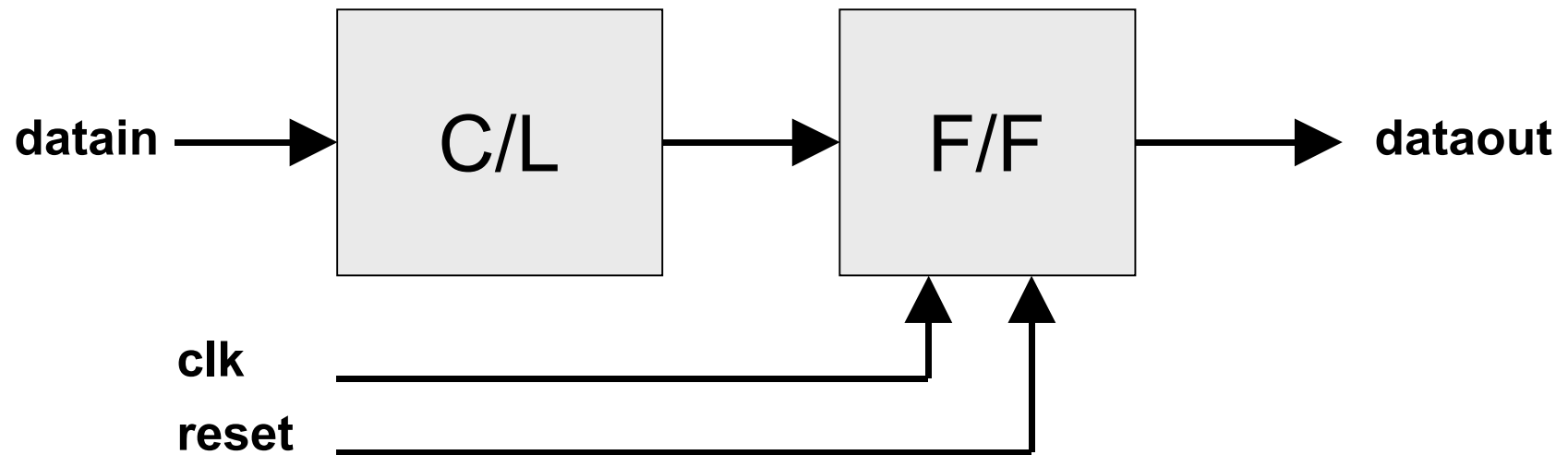
} Both possibilities
of 'SEL' have
equal priority

Control Flow- VI

- The choice of *if-then-else* Vs. *case* has an effect on the resultant generated circuit
 - If you select if-then-else then a priority encoder will be inferred which will add extra gates to the design !
- Choice of statement
 - Use IF-THEN-ELSE if there is the possibility of more than one of the input conditions happening (and hence specify a priority) or if there is on a single selection
 - Use CASE if only one of the possible inputs could happen (mutually exclusive)- this will almost always generate the most efficient circuit

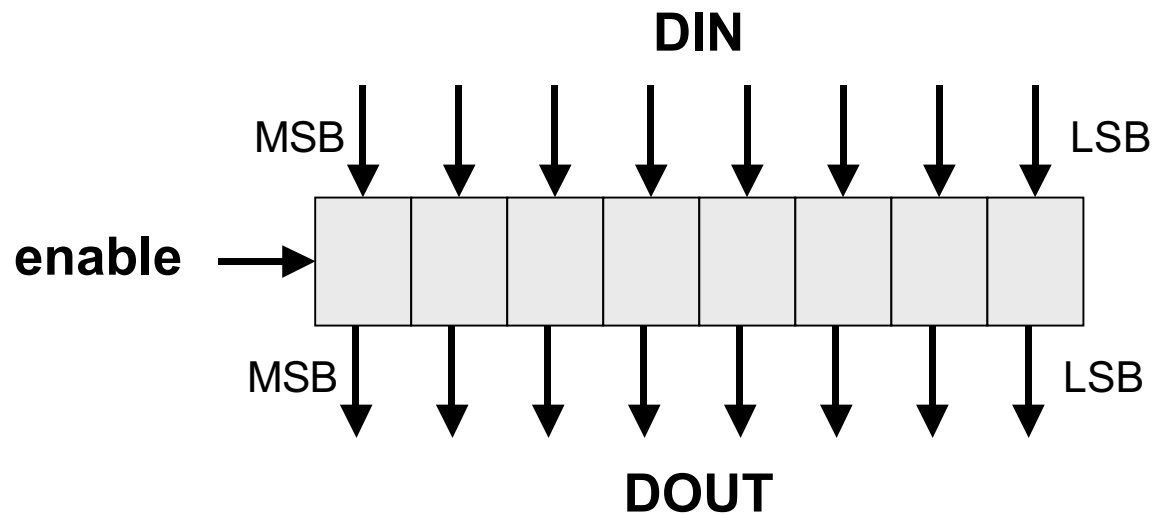
Summary

- Synchronous Design Methodology
 - RTL (Register Transfer Logic) style of VHDL coding
- Clocked/latched (so far combinatorial) systems
 - IF...THEN...ELSE
 - CASE...



A Simple Register Example- I

- Want a clocked 8 Bit register, with enable
 - So, when enable='1' **DOUT** follows **DIN** on rising clock edge
 - When enable='0' no change in **DOUT**



A Simple Register Example- I

- Want a clocked 8 Bit register, with enable
 - So, when enable='1' **DOUT** follows **DIN** on rising clock edge
 - When enable='0' no change in **DOUT**
- Here's the entity

```
entity SIMPREG is
  port (DIN: in std_logic_vector (7 downto 0);      -- system inputs
        DOUT: out std_logic_vector (7 downto 0);   -- system outputs
        ENABLE: in std_logic;                      -- load enable
        CLK,RESET: in std_logic);                 -- clock and reset
end SIMPREG;
```

- Program listing in sheet 'SIMPREG.VHD'

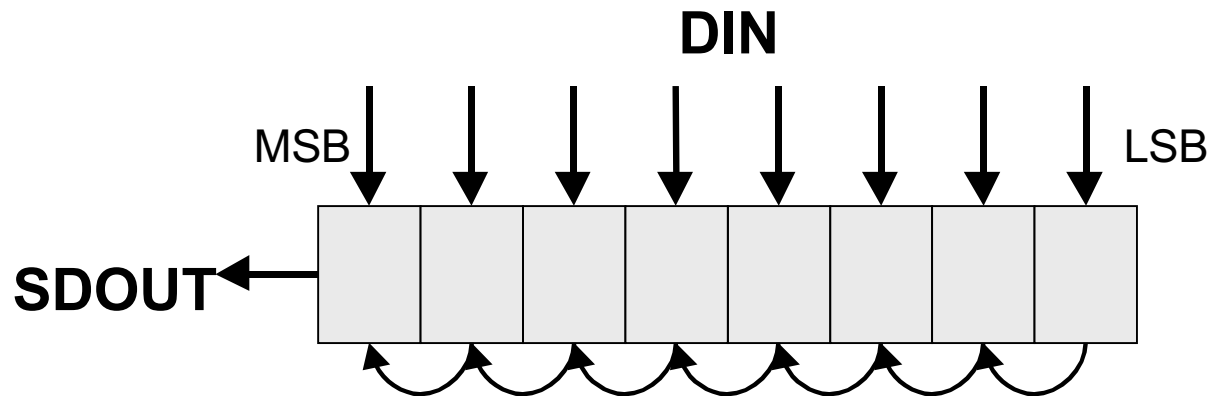
A Simple Register Example- II

- What about the architecture ?

```
architecture SIMPLE of SIMPLEREG is
begin
  process (CLK,RESET)
    begin
      if RESET='1' then -- reset (asynchronous)
        DOUT <= "00000000";
      else if CLK' EVENT and CLK='1' then -- rising edge of clock
        if LENABLE='1' then -- if enable input received
          DOUT <= DIN;
        else
          null; -- do nothing
        end if;
      end if;
    end process;
end SIMPLE;
```

Parallel to Serial Example- I

- Can extend our simple register to make a shift register
- FOR loop can be used to make the shifting process easier



- Have to include an internal signal to hold the register status
 - Will explain this during example

Parallel to Serial Example - II

- Entity declaration
- Have 2 pins called 'mode' to determine the register mode
 - 00= No operation
 - 01= Load
 - 10= Shift left
 - 11= No operation
- Also, a set of pins to load the data (DIN)
- And a pin to shift the data out (SDOUT)
- Program listing in sheet 'PAR2SER.VHD'

Parallel to Serial Example - III

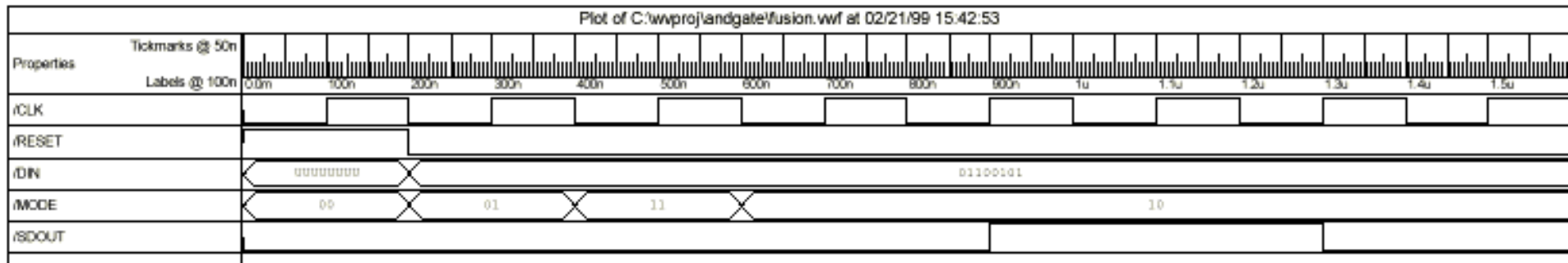
- Resultant (partial) code as follows (see program code handout for complete details)

```
if RESET='1' then
    SDOUT <= '0';
    IDATA <= "00000000"           -- IDATA is internal signal

elsif CLK`event and CLK='1' then
    case MODE is
        when "00" =>
            null;
        when "01" =>
            IDATA <= DIN;
        when "10" =>
            SDOUT <= IDATA(7);
            for mloop in 6 downto 0 loop
                IDATA( mloop+1) <= IDATA( mloop);
            end loop;
        when others =>
            null;
    end case;
```

Parallel to Serial Example- IV

- Simulation output:





VHDL- Concluding Remarks

- Have introduced all the main points of writing VHDL code for logic synthesis
- Only a limited range of the language features are suitable for synthesis
 - Standards are currently being developed to define the actual range !
- The final output is only as good as the designer who wrote the code !
 - Just as a programmer can write inefficient ‘C’ code, a VHDL programmer can write bad VHDL code which can have drastic effects (e.g. 1000% more Flip-Flops than are really necessary)

