RTL Design

This lecture describes an approach to logic design called Register Transfer Level (RTL) design. This is the approach used for the design of logic circuits ranging in complexity from digital clocks to microprocessors.
The steps involved include partitioning the data structures required by a device into the registers, selecting the operations that need to be performed on the data in those registers, and planning the exact sequence in which those operations are to be done.
After this lecture you should be able to:

- *classify a given piece of VHDL code as a behavioural, structural, or dataflow description*
- *identify the registers and combinational logic functions required to implement a particular algorithm*
- *partition the algorithm into a sequence of register transfers*
- *write synthesizeable VHDL code to implement an RTL processor*

Design Strategies

There are a number of strategies that are useful in dealing with complex designs.

One strategy is to design at the highest (most abstract) level possible with the tools available. For example, using a behavioural design style with an HDL instead of a structural style with schematics will make it easier to write, read, document, debug and transfer your design.

Another design strategy is to use hierarchical decomposition. The device being designed should be decomposed into a number of modules (VHDL entities) that interface through well-defined interfaces (VHDL ports). The internal structure of these modules should not be visible from outside the module. Each of these modules should then be further subdivided in other modules. The decomposition process should be repeated until the remaining modules are simple enough to be easily written and tested. This decomposition makes it easier to test the modules, allows modules to be re-used and allows more than one person to work on the same project at the same time.

It's also a good idea to keep the design as portable as possible. Avoid using language features that are specific to a particular manufacturer or target technolgy unless they are necessary to meet other requirements. This will make it possible to use different manufacturing processes and different devices with a minimum of redesign.

Structural Design

Structural design is the oldest digital logic design method. In this method the user does all the work. The user selects the low-level components and decides exactly how they are to be connected together. The parity generator described in the previous lecture is an example of structural design.

A structural design can be represented as a parts list and a list of the connections between the pins on the components (for example: "pin 12 on chip 3 is connected to pin 5 on chip 7"). This representation of a circuit is called a *netlist*.

Schematic capture is the most common structural design method. The designer works with a program similar to a drawing program that allows components to be inserted into the design and connected to other components.

Behavioural Design

At the other extreme, a behavioural design is meant to demonstrate the function of a device without being concerned about the implementation. Thus a behavioural design may include operations (such as propagation delays) that cannot be synthesized at all or that would be difficult to synthesize (such as integer division).

However, every design should start with a behavioural description. The behavioral description can be simulated and used to verify that all of the

required aspects of the design have been identified. Often the output of a behavioural description can be compared to the output of a structural or RTL description to check for errors.

As a simple example, consider a device that needs to add four numbers. In VHDL we can simply write:

```
c <= a + b + c + d ;
```

This particular description is simple enough that it can probably be synthesized. However, the resulting circuit will be a fairly large combinational circuit including three adder circuits.

RTL Design

We can also compute the sum of the four values using the following sequence of steps:

```
c <= 0 ;
c <= c + a ;
c <= c + b ;
c <= c + c ;
c <= c + d ;
```

where each signal assignment is executed sequentially. Now we only need one adder circuit but the process requires five steps and likely will take five times as long. This approach is quite common and the design method that has been developed to design such circuits is called Register Transfer Level (RTL) or "dataflow" design.

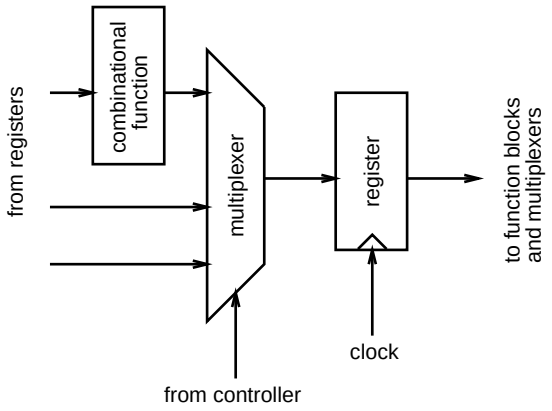
RTL design is well suited for the design of special-purpose processors such as disk drive controllers, video display cards, network adapter cards, etc.

The first step in an RTL design is to define the operation of a device as an algorithm. This algorithm is then broken down into (1) a set of registers and combinational function blocks (e.g. adders) called the *datapath* and (2) a state machine, called the *controller* that controls the transfer of data through the function blocks and between the registers.

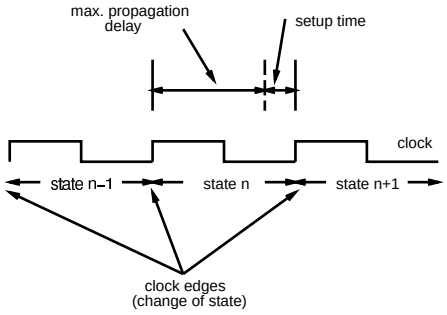
Producing an RTL design is similar to writing a computer program. Selecting the flow of data in the datapath is similar to writing expressions involving the variables (registers) and operators (combinational function blocks) in a conventional programming language. Designing the controller state machine is similar to defining the flow of control within the program (if/then/else, while-loops, etc).

When doing RTL design with VHDL the design of the datapath (the instantiation of registers and the design of multiplexers and combinational functions) is done by the synthesizer. However, the designer must design the state machine and decide on which register transfers are performed in which clock cycle.

The diagram below shows a part of a typical datapath. The type and number of registers, combinational functions and multiplexers will be determined by the application.



As in any synchronous circuit, the registers in the datapath and the registers in the state machine use the same clock. Thus they will both change state at the same time. The datapath registers therefore load the values "computed" during one state at the end of that state (which is also the start of the next state). To ensure proper operation the worst-case propagation delay through the multiplexers and combinational function blocks must be less than the clock period minus the register's setup time.



RTL designs can trade datapath complexity (e.g. using more adders and thus using more chip area) against speed (e.g. having more adders means fewer operations are required to obtain the result).

RTL Design Example

To show how an RTL design is described in VHDL and to clarify the concepts involved, we will design a four-input addder. This design will also demonstrate how to create packages of components that can be re-used.

The first design unit is a package that defines a new type, `num`, for eight-bit unsigned numbers and an enumerated type, `states`, with 6 possible values. `num`s are defined as a subtype of the `unsigned` type.

```
-- RTL design of 4-input summer

-- subtype used in design

library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;

package averager_types is
   subtype num is unsigned (7 downto 0) ;
   type states is (clr, add_a, add_b, add_c,
      add_d, hold) ;
end averager_types ;
```

The first entity defines the datapath. In this case the four numbers to be added are available as inputs to the entity and there is one output for the sum. The datapath includes one register with synchronous load and clear functions, an 8-bit adder and a multiplexer that selects one of the four inputs as the value to be added to the current value of the register.

The inputs to the datapath from the controller are a 2-bit selector for the multiplexer and two control signals to load or clear (set to 0) the register.

```
-- datapath

library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.averager_types.all ;

entity datapath is
   port (
   a, b, c, d : in num ;
   sum : out num ;
   sel : in std_logic_vector (1 downto 0) ;
   load, clear, clk : in std_logic
   ) ;
end datapath ;

architecture rtl of datapath is
   signal mux_out, sum_out, reg_out : num ;
begin

   -- multiplexer selects input to be added
   mux_out <=
      a when sel = "00" else
      b when sel = "01" else
      c when sel = "10" else
      d ;

   -- adder adds selected input and register
   sum_out <= mux_out + reg_out ;

   -- output is register output
   sum <= reg_out ;

   -- register with load and clear functions
   process(d,load,clear,clk)
   begin
      if clk'event and clk = '1' then
         if clear = '1' then
            reg_out <= conv_unsigned(0,reg_out'length) ;
         elsif load = '1' then
            reg_out <= sum_out ;
         end if ;
      end if ;
   end process ;
end rtl ;
```

The controller is a state machine. It's outputs are used to control the datapath and its inputs are used to control the state machine's state transitions. In this case the only input is an `update` signal that tells our device to re-compute the sum (presumably because one or more of the inputs has changed).

This particular state machine sits at the "hold" state until the update signal is true. It then sequences through the other five states and the stops at the hold state again. The other five states are used to clear the register and to add the four inputs to the current value of the register.

```
-- controller

library ieee ;
use ieee.std_logic_1164.all ;
use work.averager_types.all ;

entity controller is
   port (
   update : in std_logic ;
   sel : out std_logic_vector (1 downto 0) ;
   load, clear : out std_logic ;
   clk : in std_logic
   ) ;
end controller ;

architecture rtl of controller is
   signal s, ns : states ;
   signal tmp : std_logic_vector (3 downto 0) ;
begin

   -- state register
   process(ns,clk)
   begin
      if clk'event and clk = '1' then
```

```
         s <= ns ;
      end if ;
   end process ;

  -- state sequencer
   process(s,update)
   begin
      if update = '1' and s = hold then
         ns <= clr ;
      else
         case s is
            when clr    => ns <= add_a ;
            when add_a  => ns <= add_b ;
            when add_b  => ns <= add_c ;
            when add_c  => ns <= add_d ;
            when add_d  => ns <= hold ;
            when hold   => ns <= hold ;
         end case ;
      end if ;
   end process ;

   -- controller outputs
   with s select tmp <=
      "0001" when clr,
      "0010" when add_a,
      "0110" when add_b,
      "1010" when add_c,
      "1110" when add_d,
      "0000" when hold ;

   sel   <= tmp(3 downto 2) ;
   load  <= tmp(1) ;
   clear <= tmp(0) ;

end rtl ;
```

The next piece of code is an example of how the datapath and the controller entities can be placed in a package, `averager_components`, as components. This would not normally be worthwhile for such a simple design.

```
-- package for datapath and controller

library ieee ;
use ieee.std_logic_1164.all ;
use work.averager_types.all ;

package averager_components is

component datapath
   port (
   a, b, c, d : in num ;
   sum : out num ;
   sel : in std_logic_vector (1 downto 0) ;
   load, clear, clk : in std_logic
   ) ;
end component ;

component controller
   port (
   update : in std_logic ;
   sel : out std_logic_vector (1 downto 0) ;
   load, clear : out std_logic ;
   clk : in std_logic
   ) ;
end component ;

end averager_components ;
```

The top-level `averager` entity instantiates the two components and interconnects them. The two declarations in the architecture starting with `for` are configuration declarations. They specify which architectures are to be used for each entity[1]

```
-- averager

library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.averager_types.all ;
use work.averager_components.all ;

entity averager is port (
   a, b, c, d : in num ;
   sum : out num ;
   update, clk : in std_logic ) ;
end averager ;

architecture rtl of averager is
   signal sel : std_logic_vector (1 downto 0) ;
   signal load, clear : std_logic ;

   for all : datapath use entity work.datapath (rtl) ;
   for all : controller use entity work.controller (rtl) ;

begin
   d1: datapath port map ( a, b, c, d, sum, sel, load,
                           clear, clk ) ;
   c1: controller port map ( update, sel, load,
                             clear, clk ) ;
end rtl ;
```

[1] They should default to the most recently analyzed architectures but Synopsys' `vhdlsim` simulator does not seem to understand this.