

RTL Design with VHDL

This chapter covers an approach to logic design called Register Transfer Level (RTL) design. This is the method used for the design of complex logic circuits such as microprocessors.

You should be able to:

identify the registers and logic/arithmetic functions required to implement a particular algorithm; partition this algorithm into a sequence of these operations and register transfers; write synthesizable VHDL RTL code to implement the algorithm.

Design Strategies

There are a number of strategies that are useful when designing complex logic circuits. You may recognize similar strategies that are used in computer programming.

One strategy is to design at the most abstract (“highest”) level possible with the tools available. For example, using a behavioral design style with VHDL instead of a structural style (e.g. schematics) will make it easier to write, read, document, and debug your design.

Another design strategy is hierarchical decomposition. The device being designed should be decomposed into a number of modules (represented as 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 into 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 easy to test the modules individually, 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 technology 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 designer does all the work. The designer selects the low-level components and decides exactly how they are to be connected. The parity generator described previously 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.

Exercise 1: What would be the most common type of statement in a structural VHDL description?

Behavioral Design

A behavioral design is meant to demonstrate the behaviour of a device without necessarily including details of the implementation.

If the behavioural description is written in a certain style it may be possible for a program called a logic synthesizer to map it to a hardware implementation.

In other cases the behavioral description may not be synthesizable and is used only used to verify other steps of the design process.

Exercise 2: A VHDL description contains non-synthesizable constructs such as propagation delays. Is it a behavioural or structural description?

RTL Design

Register Transfer Level, or RTL¹ design lies between a purely behavioral description of the desired circuit and a purely structural one. An RTL description describes a circuit's registers and the sequence of transfers between these registers but does not describe the hardware used to carry out these operations.

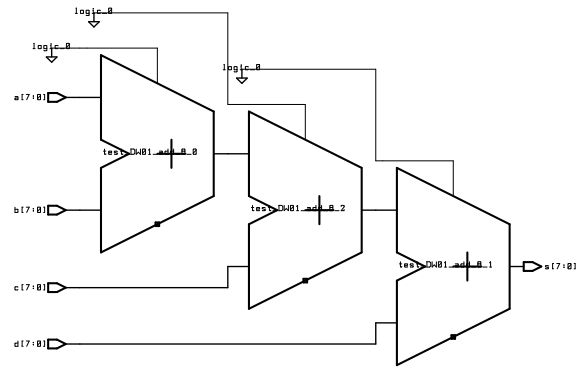
The steps in RTL design are: (1) determine the number and sizes of registers needed to hold the data used by the device, (2) determine the logic and arithmetic operations that need to be performed on these register contents, and (3) design a state machine whose outputs control how the register contents are updated in order to obtain the desired results.

Producing an RTL design is similar to writing a computer program in a conventional programming language. Choosing registers is the same as choosing variables. Designing the flow of data in the “datapath” is analogous to writing expressions involving the variables (registers) and operators (combinational functions). Designing the controller state machine is similar to deciding on the flow of control within the program (if/then/else, while-loops, etc).

As a simple example, consider a device that needs to add four numbers. In VHDL, given signals of the correct type, we can simply write:

```
s <= ( ( a + b ) + c ) + d ;
```

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



A behavioral description, not being concerned with implementation details, would be complete at this point.

However, if we were concerned about the cost of the implementation we might decide to break down the computation into a sequence of steps, each one involving only a single addition:

```
s = 0
s = s + a
s = s + b
s = s + c
s = s + d
```

where each operation is executed sequentially. The logic required is now one adder, a register to hold the value of *s* in-between operations, a multiplexer to select the input to be added, and a circuit to clear *s* at the start of the computation.

Although this approach only needs one adder, the process requires more steps and will take longer. Circuits that divide up a computation into a sequence of arithmetic and logic operations are quite common and this type of design is called Register Transfer Level (RTL) or “dataflow” design.

An RTL design is composed of (1) registers and combinational function blocks (e.g. adders and multiplexers) called the *datapath* and (2) a finite state machine, called the *controller* that controls the transfer of data through the function blocks and between the registers.

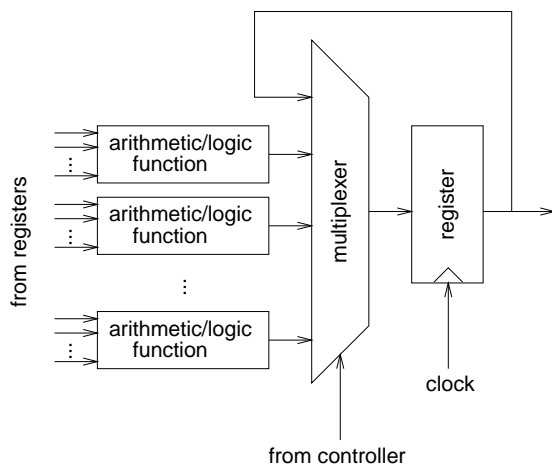
In RTL design the gate-level design and optimization of the datapath (registers, multiplexers, and combinational functions) is done by the synthesizer.

¹The “L” in RTL sometimes stands for “Language” or “Logic” – all refer to the same method of designing complex logic circuits.

However, the designer must design the state machine and decide which register transfers are performed in which state.

The RTL designer can trade off datapath complexity (e.g. using more adders and thus using more chip area) against speed (e.g. having more adders means fewer steps are required to obtain the result). RTL design is well suited for the design of CPUs and special-purpose processors such as disk drive controllers, video display cards, network adapter cards, etc. It gives the designer great flexibility in choosing between processing speed and circuit complexity.

The diagram below shows a generic component in the datapath. Each RTL design will be composed of one of the following building blocks for each register. The structure allows the contents of each register to be updated at the end of each clock period with a value selected by the controller. The widths of the registers, the types of combinational functions and their inputs will be determined by the application. A typical design will include many of these components.

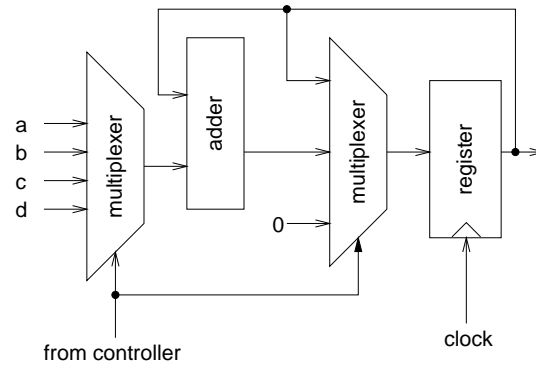


RTL Design Example

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

The datapath shown below can load the register at the start of each clock cycle with one of: zero, the current value of the register, or the sum of the register and one of the four inputs. It includes one 8-bit register, an 8-bit adder and a multiplexer that selects one

of the four possible inputs as the value to be added to the current value of the register.



Exercise 3: Other datapaths could compute the same result. Draw the block diagram of a datapath capable of computing the sum of the four numbers in three clock cycles.

The first design unit is a package that defines a new type, `num`, for eight-bit unsigned numbers and an enumerated type, `states`, with six possible values. `nums` 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 current sum.

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_reg, next_sum_reg : num ;
    constant sum_zero : num :=
        conv_unsigned(0,next_sum_reg'length) ;
begin

    -- mux to select input to add
    with sel select mux_out <=
        a when "00",
        b when "01",
        c when "10",
        d when others ;

    -- mux to select register input
    next_sum_reg <=
        sum_reg + mux_out when load = '1' else
        sum_zero when clear = '1' else
        sum_reg ;

    -- register sum
    process(clk)
    begin
        if clk'event and clk = '1' then
            sum_reg <= next_sum_reg ;
        end if ;
    end process ;

    -- entity output is register output
    sum <= sum_reg ;

end rtl ;

```

Exercise 4: Label the block diagram above with the bus widths and signal names used in the entity.

What would happen if both `clear` and `load` inputs were asserted? Why do we need to define both `sum_reg` and `sum` signals?

How many clock cycles will it take to compute the sum of the four inputs?

The RTL design's controller is a state machine whose outputs control the multiplexers in the datapath. The controller's inputs are signals that control the controller's state transitions. In this case the only input is an update signal that tells our device to recompute 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 then 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, holdns, ns : states ;
    signal tmp : std_logic_vector (3 downto 0) ;
begin

    -- select next state
    with s select ns <=
        add_a when clr,
        add_b when add_a,
        add_c when add_b,
        add_d when add_c,
        hold when add_d,
        holdns when others ; -- hold

    -- next state if in hold state
    holdns <=
        clr when update = '1' else
        hold ;

    -- state register
    process(clk)
    begin
        if clk'event and clk = '1' then
            s <= ns ;
        end if ;
    end process ;

    -- controller outputs
    with s select sel <=
        "00" when add_a,
        "01" when add_b,
        "10" when add_c,
        "11" when others ;

    load <= '0' when s = clr or s = hold else '1' ;

    clear <= '1' when s = clr else '0' ;

end rtl ;

-- 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.

```

-- averager

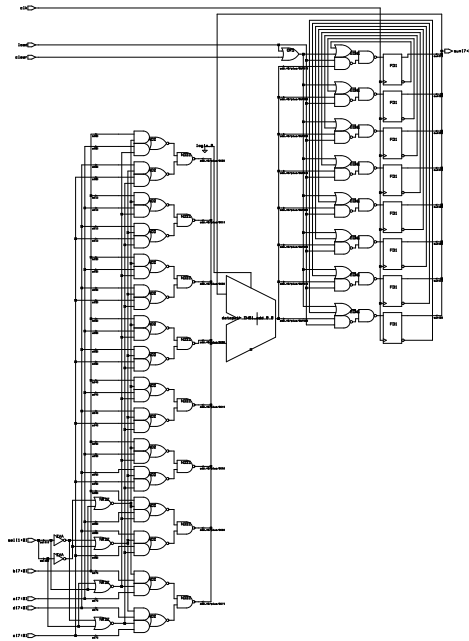
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.all ;
use work.averager_types.all ;
use work.averager_components.all ;

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 ;
  -- other declarations (e.g. components) here
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 ;

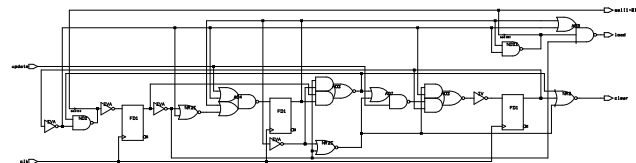
```

The result of the synthesizing the datapath is:

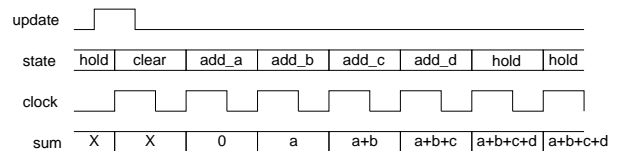


The register flip-flops are at the upper right, the adder is in the middle and the input multiplexer is at the lower left.

The result of the synthesizing the controller is:



The following timing diagram shows the datapath output and the controller state over one computation. Note that the state and output transitions take place on the rising edge of the clock. Also note that the output is updated at the *end* of the state in which a particular operation is performed.



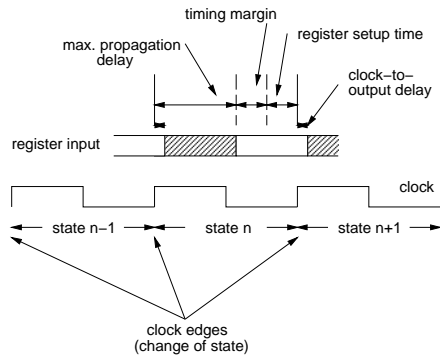
RTL Timing Analysis

As usual, the datapath should be designed as a synchronous sequential circuit that uses the same clock for all registers. All register contents thus change at the same time. The controller also uses the same clock as the datapath.

The result is that each datapath register loads the values “computed” during one state at the end of that state (which is then the start of the computation for the next state).

We can guarantee that the correct results will be loaded into registers if the longest propagation delay (t_{PD}) through any path through the combinational logic that lies between register outputs and inputs is less than the clock period (t_{clock}) minus the registers’ setup time (t_s) and clock-to-output (t_{CO}) delays:

$$t_{PD} < t_{\text{clock}} - t_s - t_{CO}$$



Using a single clock means we only need to compute the delay through *combinational* logic blocks which is much simpler than having to predict the effect of propagation delays on clock signals. This is why almost all large-scale digital circuits are synchronous designs.

Synthesis tools can be asked to synthesize logic that operates at a particular clock period. The synthesizer is supplied with the propagation delay specifications for the combinational logic components available in the particular technology being used and it will then try to arrange the logic so that the longest propagation delay between any register output and any register input is less than the clock period (minus setup and clock-to-output delays). This ensures that the circuit will work properly at the specified clock rate.