RTL Design with VHDL

This chapter covers some features of VHDL that are useful for logic synthesis. You should learn to:

- *make library packages visible*
- *declare components in architectures and packages*
- *declare constants*
- *instantiate components into an architecture*
- *declare std_logic, std_logic_vector, signed and unsigned signals*
- *declare enumerated types and subtypes of array types in architectures and packages*
- *declare and use entities with generics*
- *use conditional signal assignments*
- *convert between std_logic_vector, unsigned and integer types*
- *instantiate tri-state outputs*
- *create RAM and ROM memories*
- *classify a VHDL description as a behavioral, structural, or dataflow (RTL) description*

We will also learn an approach to logic design called Register Transfer Level (RTL) or "dataflow" design. This is the method currently used for the design of complex logic circuits such as microprocessors.
You should be able to:

- *select a sufficient set of registers and logic/arithmetic functions required to implement an algorithm*
- *convert the algorithm into a sequence of register transfers through logic/arithmetic functions*
- *write synthesizeable VHDL RTL code to implement the algorithm*

We also cover three topics related to the design of interfaces to logic circuits: metastability, input synchronization and glitches. You should be able to: identify circuits where metastable behaviour is possible; compute the mean time between metastable outputs; identify circuits that could fail due to asynchronous inputs; add synchronizer flip-flops to reduce the probability of metastability; remove race conditions by registering inputs; and use registered outputs to eliminate glitches.

Reserved Words

Table **??** lists the 97 reserved words that cannot be used as VHDL identifiers.

Libraries, Packages and Components

When designing complex logic circuits it helps to decompose a design into simpler parts. Each of these parts can be written and tested separately, perhaps by different people. If the parts are sufficiently general then it's often possible to re-use them in future projects. In VHDL, design re-use is done by using "components." A component can be a general-purpose building-block (e.g. an adder or a counter), or it can be sub-system of your design.

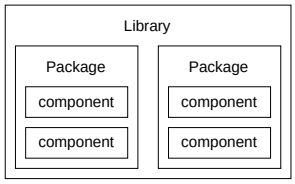
Before we use a component, we first need to declare it. A component declaration is very similar to an entity declaration — it defines the input and output signals, not the functionality.

In order to avoid declaring each component in every architecture where it is used, we typically place component declarations in "packages." A package typically contains a set of component declarations for a particular application. Packages are themselves

abs access after alias all and architecture array assert attribute begin block body buffer bus case component configuration constant disconnect downto else elsif end entity exit file for function generate generic group guarded if impure in inertial inout is label library linkage literal loop map mod nand new next nor not null of on open or others out package port postponed procedure process pure range record register reject rem report return rol ror select severity signal shared sla sll sra srl subtype then to transport type unaffected units until use variable wait when while with xnor xor

Table 1: VHDL reserved words.

stored in "libraries":



In the Synopsys Design Compiler[1] and Max+PlusII VHDL implementations, a library is a directory and each package is a file in that directory. The package file is a database containing information about the components in the package (the component inputs, outputs, types, etc).

To use a component in a design, we use `library` statements to specify the libraries to be searched and a `use` statement for each package we need to use. The two most commonly used libraries are called `IEEE` and `WORK`.

The `WORK` library is always available without having to use a library statement. In Design Compiler the `WORK` library is a subdirectory of the current directory called `WORK` while in Max+PlusII it is the current project directory.

`library` and `use` statements must be used before *each* design unit (entity or architecture) that uses those packages[2]. For example, if you wanted to use the `numeric_bit` package in the `ieee` library you would use:

```
library ieee ;
use ieee.numeric_bit.all ;
```

and if you wanted to use the `dsp` package in the `WORK` library you would use:

```
use work.dsp.all ;
```

Exercise 35: Why is there no library statement in the second example?

Note that a component defines an interface to another device. That device may not have been designed with VHDL so there may not necessarily be a corresponding entity declaration.

Creating Components

A component declaration is similar to an entity declaration and defines the input and output signals.

Component declarations can be placed in an architecture before the `begin`. But it's usually more convenient to put component declarations within a `package` declaration. When we compile (or "analyze") the package declaration the information about the components in the package is saved in a file in the WORK library. The components in the packages can then be used in an architecture (in that same file or in other files) by using the appropriate `use` statements.

For example, the following code declares a package called `flipflops`. This package contains only one component, `rs`, with inputs `r` and `s` and an output `q`:

```
package flipflops is
   component rs
      port ( r, s : in bit ; q : out bit ) ;
   end component ;
end flipflops ;
```

Exercise 36: If this code was stored in a file called `ff.vhd`, how many files would be created? What would they contain? Where would they be placed?

Component Instantiation

Once a component has been declared, it can be used ("instantiated") in an architecture. A component instantiation describes how the component is "hooked

[1] The logic synthesizer used to create the schematics in these lecture notes.

[2] An exception: when an architecture immediately follows its entity you need not repeat the `library` and `use` statements.

up" to the other signals in the architecture. It is a *concurrent* statement (as is a selected assignment statement).

The following example shows how three 2-input exclusive-or gates can be used to build a 4-input parity-check circuit using component instantiation. This type of description is called *structural* VHDL because we are defining the structure rather than the behaviour of the circuit.

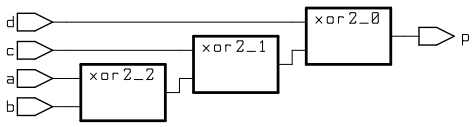
In this case we have put the component declaration into the file `mypackage.vhd`. The `xor_pkg` contains the `xor2` component (although a typical package defines more than one component):

```
-- define an xor2 component in a package

package xor_pkg is
   component xor2
      port ( a, b : in bit ; x : out bit ) ;
   end component ;
end xor_pkg ;
```

A second file, `parity.vhd`, describes the `parity` entity that uses the `xor2` component:

```
-- parity function built from xor gates

use work.xor_pkg.all ;

entity parity is
   port ( a, b, c, d : in bit ; p : out bit ) ;
end parity ;

architecture rtl of parity is
   -- internal signals
   signal x, y : bit  ;
begin
   x1: xor2 port map ( a, b, x ) ;
   x2: xor2 port map ( c, x, y ) ;
   x3: xor2 port map ( d, y, p ) ;
end rtl ;
```

The resulting top-level schematic for the `parity` entity is:



Exercise 37: Label the connections within the parity generator schematic with the signal names used in the architecture.

When the `parity.vhd` file is analyzed ("compiled"), the synthesizer will search the (`WORK`) directory for the `xor_pkg` package.

We could also have put the `xor_pkg` package declaration in the `parity.vhd` file (the package file would then be recreated every time we analyzed `parity.vhd`).

Although components don't necessarily have to be created using VHDL, we could have done so by using the following entity/architecture pair in file called `xor2.vhd`:

```
-- xor gate

entity xor2 is
   port ( a, b : in bit ; x : out bit ) ;
end xor2 ;

architecture rtl of xor2 is
begin
   x <= a xor b ;
end rtl ;
```

VHDL versus C Terminology

The following comparison shows some rough equivalents between the VHDL concepts described above and C programming[3].

VHDL	C
analyze	compile
elaborate	link
component	function
instantiate	call
`use`	`#include`
package	DLL
library	directory

`std_logic` Packages

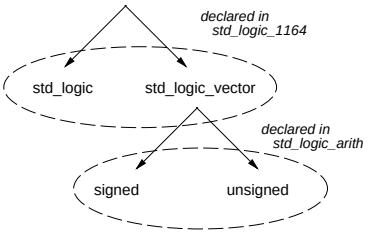
The `IEEE` library contains two useful packages. These packages define alternatives to the `bit` and `bit_vector` types for logic design.

The first package, `std_logic_1164`, defines the types `std_logic` (similar to `bit`) and `std_logic_vector` (similar to `bit_vector`). The advantage of the `std_logic` types is that they can have values other than '0' and '1'. For example, an `std_logic` signal can also have undefined ('X') and high-impedance values ('Z'). The `std_logic_1164` package also redefines ("overloads") the standard

[3]The effect of a call is rather different than a component instantiation: in VHDL we get an extra copy of the component each time it is used. In C we get only one copy of a function no matter how many times it is called.

boolean operators (`and`, `or`, `not`, etc.) so that they work with `std_logic` signals.

The second package, `std_logic_arith`[4] defines the types `signed` and `unsigned`. These are subtypes of `std_logic_vector` with overloaded operators that allow them to be used both as vectors of logic values and as as binary numbers (in signed two's complement or unsigned representations). The hierarchy of these logic types could be drawn as follows:

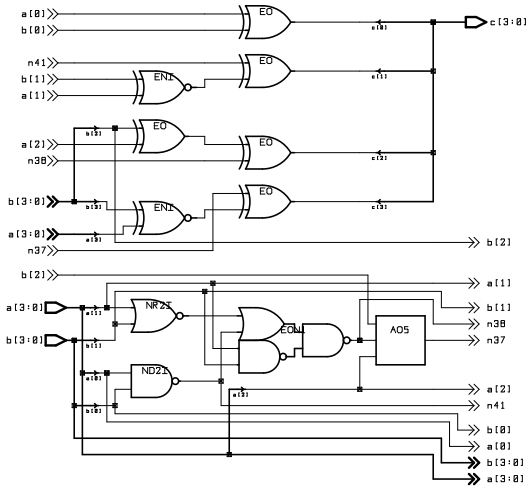


The standard arithmetic operators (`+`, `-`, `*`, `/`, `**`, `>`, `<`, `<=`, `>=`, `=`, `/=`) can be applied to signals of type `signed` or `unsigned`. Note that it may not be practical or possible to synthesize complex operators such as multiplication, division or exponentiation.

For example, we could generate the combinational logic to build a 4-bit adder using the following architecture:

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

entity adder4 is
   port (
   a, b : in unsigned (3 downto 0) ;
   c : out unsigned (3 downto 0) ) ;
end adder4 ;

architecture rtl of adder4 is
begin
   c <= a + b ;
end rtl ;
```

The resulting (rather messy) schematic is:



Constants

You can declare symbolic constants in the same way as signals. For example:

```
constant zero_bits : unsigned (3 downto 0) := "0000" ;
```

A constant declared in a package is available to all design units (packages, entities and architectures) that `use` that package. You should use symbolic constants for any values that are likely to change or if it makes your code easier to read or easier to modify.

Integers

VHDL also includes an `integer` type which is useful for specifying small constants (e.g. `next_x <= x + 1 ;`). However, *signals* should be declared `std_logic` or one of its subtypes, *not* `integer`. Declarations sometimes use the `natural` (values $>= 0$), and `positive` (values > 0) types. Integer constants can be specified in non-decimal base. For example, the value 2000 hex can be specified as: `16#2000#`.

Type Conversion Functions

VHDL is a strongly-typed language – each operator must be supplied arguments of exactly the right type or the synthesizer will give an error message. Although many functions and operators (e.g. `and`)

[4]The IEEE standard is really `numeric_std` but it's not widely used.

are overloaded so that you can use the same function/operator with more than one type, in many cases you will need to use type conversion functions.

The following type conversion functions are found in the the `std_logic_1164` package in the `ieee` library:

from	to	function
lv	bv	`to_bitvector(x)`
bv	lv	`to_stdlogicvector(x)`

The abbreviations `bv`, `lv`, `un` and `in` are used for `bit_vector`, `std_logic_vector`, `unsigned` and `integer` respectively.

The following type conversion functions are found in the the `std_logic_arith` package in the `ieee` library.

from	to	function
lv	un	`unsigned(x)`
un	lv	`std_logic_vector(x)`
un	in	`conv_integer(x)`
in	un	`conv_unsigned(x,len)`
in	lv	`conv_std_logic_vector(x,len)`

Functions in the `std_logic_arith` package "overload" most of the arithmetic and comparison operators (e.g. `+`, `=`) so that they take integer as well as `unsigned` operands.

Note that when converting an `integer` you must explicitly specify the number of bits in the result (`len`).

For example:

```
constant awidth : integer := 24 ;
constant dwidth : integer :=  8 ;
constant r1addr : std_logic_vector (awidth-1 downto 0)
            := to_stdlogicvector(X"1A_0002") ;
signal abus : unsigned (awidth-1 downto 0) ;
signal r1, d : std_logic_vector (dwidth-1 downto 0) ;
...
r1 <=
   d when abus = unsigned(r1addr) else
   "00000000" ;
```

Exercise 38: What is the type of the constant `X"1A_0002"`? What is the purpose of the `unsigned()` function in the last line of the above example? What conversion function(s) would you need to use if `r1addr` was declared to be of type `bit_vector`?

Type Declarations

It's often useful to make up new types for a project. We can do this in VHDL by using type declarations. The most common uses for defining new types are to create signals of a given width (i.e. a bus) and to declare types that can only have one of a set of possible values (called enumeration types).

Type declarations are often placed in packages to make them available to multiple design units. The following example shows a package called `dsp_types` that declares two new types:

```
package dsp_types is
   type mode is (slow, medium, fast) ;
   subtype word is std_logic_vector (15 downto 0) ;
end dsp_types ;
```

Note that we need to use a `subtype` declaration in the second example because the `std_logic_vector` type is already defined.

Exercise 39: Write a declaration for a signal that controls whether the value in a register should be loaded, incremented, decremented, or held. Write the declaration for an 8-bit signal type called `byte`.

Generics

An entity can be declared with a bus or register size that is left undefined until the component is used ("instantiated") by adding a *generic* clause in its entity and component declarations. For example, a register with negated outputs could be declared in the file `nregister.vhd` as:

```
-- register with negated output

entity nregister is
   generic ( width : integer ) ;
   port ( d : in  bit_vector (width-1 downto 0) ;
          q : out bit_vector (width-1 downto 0) ;
        clk : in bit ) ;
end nregister ;

architecture rtl of nregister is
   signal tmp : bit_vector(width-1 downto 0) ;
begin
   process(clk)
   begin
      if clk'event and clk='1' then
         tmp <= d ;
      end if ;
   end process ;
   q <= not tmp ;
end ;
```

you might declare the `nregister` component in a package as:

```
package registers is
   component nregister
      generic ( width : integer ) ;
      port ( d : in  bit_vector (width-1 downto 0) ;
             q : out bit_vector (width-1 downto 0) ;
           clk : in bit ) ;
   end component ;
end registers ;
```

and then use it in another architecture as follows:

```
use work.registers.all ;
...
   r1: nregister
      generic map ( 8 )
      port map ( din, dout, clk ) ;
...
```

You should use generics if your component might have to be instantiated with various signal widths.

Attributes

Each signal has a number of properties associated with it which can be extracted and used in expressions by using VHDL's *attributes*. For example, the number of elements in an array `x` is given by `x'length`. Other useful attributes are `left`, `right`, `high`, `low` which extract the appropriate index limits and `range` which extracts the index range.

Conditional Assignment

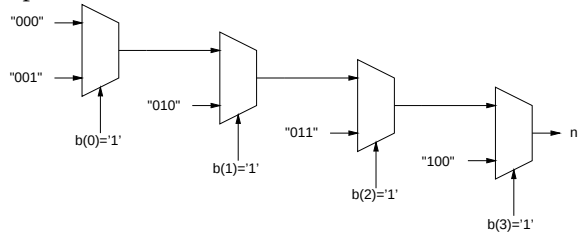
In the same way that a selected assignment statement models a case statement in a sequential programming language, a conditional assignment statement models an if/else statement. Like the selected assignment statement, it is also a *concurrent* statement.

For example, the following circuit outputs the position of the left-most '1' bit in the input:

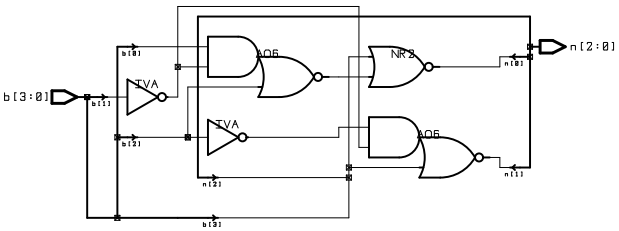
```
library ieee ;
use ieee.std_logic_1164.all ;

entity nbits is port (
   b : in std_logic_vector (3 downto 0) ;
   n : out std_logic_vector (2 downto 0) ) ;
end nbits ;

architecture rtl of nbits is
begin
   n <=
      "100" when b(3) = '1' else
      "011" when b(2) = '1' else
      "010" when b(1) = '1' else
      "001" when b(0) = '1' else
      "000" ;
end rtl ;
```

Note that the conditions are tested in the order that they appear in the statement and only the first value whose controlling expression is true is assigned.

In the same way that we can view a selected assignment statement as the VHDL model for a ROM or lookup table, a conditional assignment statement can be viewed the VHDL description of a tree of multiplexers. For example, the structure of the example above could be drawn as:



Synthesizing the above description results in:



Exercise 40: Write a conditional assignment that models a 2-to-1 multiplexer. Use an array `x` as the input, a signal `sel` to select the input and a signal `y` as the output. Repeat for a 4-to-1 multiplexer (`sel` is now an array).

The choice of selected or conditional assignments can affect the logic that is generated. A conditional assignment implies an ordered sequence of two-way decisions which results in the multiplexer tree as shown above. A selected assignment implies a logic circuit that evaluates all possible inputs simultaneously. This implies a single-stage sum-of-products (or equivalent) circuit. The circuit generated by a selected assignment will typically require less logic but will incur a longer propagation delay.

However the logic synthesizer may need to optimize the original circuit to meet either speed or space constraints. The final circuit may not match either of the above models.

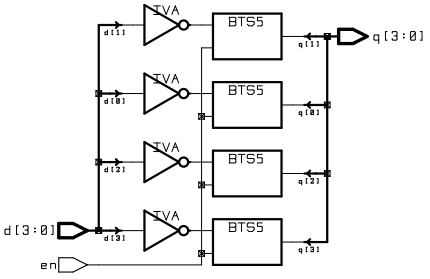
Tri-State Buses

A tri-state output can be set to high and low logic levels as well as to a third state: high-impedance (‘Z’). This type of output is used where different devices’ outputs are connected together and drive a common bus (hopefully at different times!). To specify that an output should be set to the high-impedance state, we use a signal of type `std_logic` and assign it a value of ’Z’.

The following example shows an implementation of a 4-bit buffer with an enable output. When the enable is not asserted the output is in high-impedance mode :

```
library ieee ;
use ieee.std_logic_1164.all ;

entity tbuf is port (
   d : in std_logic_vector (3 downto 0) ;
   q : out std_logic_vector (3 downto 0) ;
   en : in std_logic
   ) ;
end tbuf ;

architecture rtl of tbuf is
begin
   q <=
      d when en = ’1’ else
      "ZZZZ" ;
end rtl ;
```

The resulting schematic for the `tbuf` is:



Tri-state outputs are used primarily to implement bidirectional bus signals. Bidirectional buses are declared of type `inout` rather than `in` or `out` and their values can be both ‘read’ and ‘written’ within the architecture (unlike signals of type `out`). When the bus is to act as an input, the bidirectional bus signals are driven to the high-impedance state and in this case it’s the value of other signals that determine the signal’s value.

The tri-state enable is usually controlled by an address decoder or other enable input.

Memory Models

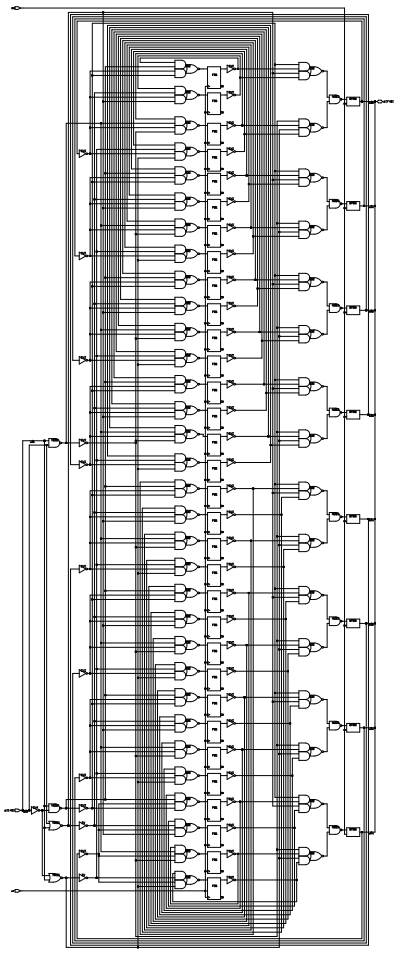
VHDL also allows the use of arrays with signal indices to model random-access memory (RAM). The following example demonstrates the use of VHDL arrays as well as bi-directional buses. We must use the type-conversion function `conv_integer` because the address input, `a`, is of type `unsigned` while the array index must be of type `integer`.

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

entity ram is   port (
   -- bi-directional data signal
   d : inout std_logic_vector (7 downto 0) ;
   -- address input
   a : in unsigned (1 downto 0) ;
   -- output enable and write strobe (clock)
   oe, wr : in std_logic ) ;
end ram ;
```

```
architecture rtl of ram is
   subtype byte is std_logic_vector (7 downto 0) ;
   type byte_array is array (0 to 3) of byte ;
   signal ram : byte_array ;
begin
   -- output value is the indexed array element
   d <=
      ram(conv_integer(a)) when oe = ’1’ else
      "ZZZZZZZZ" ;

    -- register the indexed array element
   process(wr)
   begin
      if wr’event and wr = ’1’ then
         ram(conv_integer(a)) <= d ;
      end if ;
   end process ;
end rtl ;
```

Exercise 41: Modify the design above to create a 16-element, 4-bit wide RAM with separate input and output signals. How could you model a ROM?

The result of synthesizing this description is:



For many implementation technologies (FPGAs, gate arrays, or standard-cell ASICs) there are usually vendor-specific ways of implementing memory arrays that give better results. However, using a VHDL-only model with "random logic" as shown above is more portable and may be practical for small memories such as CPU "register files."

Exercise 42: Why is portability desirable?

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

Behavioral Design

At the other extreme, a behavioral design is meant to demonstrate the functional behaviour of a device without concerning itself about implementation details. Thus a behavioral design may include operations such as integer division or behaviour such as propagation delays that are difficult or impossible to synthesize.

However, every design should start with a behavioral description. The behavioral description can be simulated and used to verify that all of the required aspects of the design have been identified. The output of a behavioral description can be compared to the output of a structural or RTL description to check for errors.

Exercise 44: A VHDL description contains non-synthesizeable constructs such as propagation delays. Is it a behavioural or structural description?

RTL Design

Register Transfer Level, or RTL[5] 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.

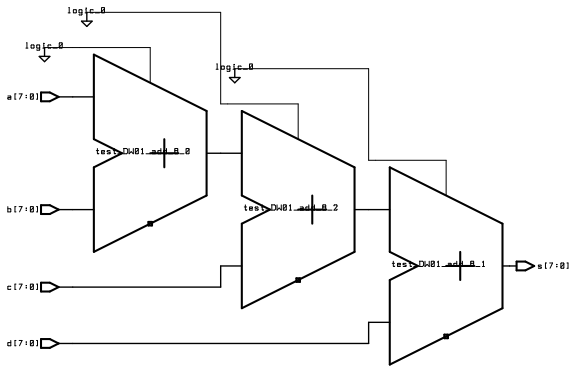
Producing an RTL design is similar to writing a computer program in a conventional programming language. Choosing registers is the same as choosing variables. Designing the flow of data in the "datapath" is analogous to writing expressions involving the variables (registers) and operators (combinational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

[5]The "L" in RTL sometimes stands for "Language" or "Logic" – all refer to the same method of designing complex logic circuits.

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.

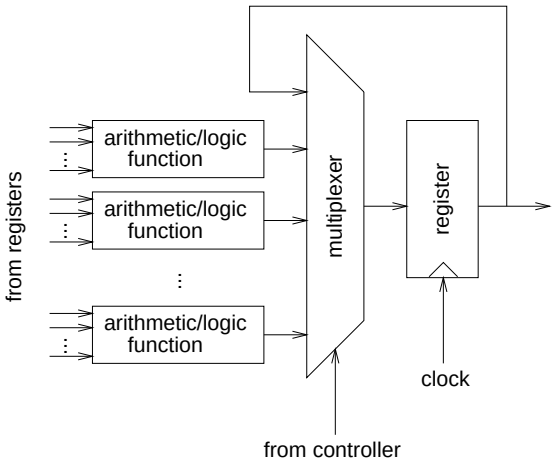
Although this approach only needs one adder, the process requires more steps and will take longer. Circuits that divide up a computation into a sequence of arithmetic and logic operations are quite common and this type of design is called Register Transfer Level (RTL) or "dataflow" design.

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

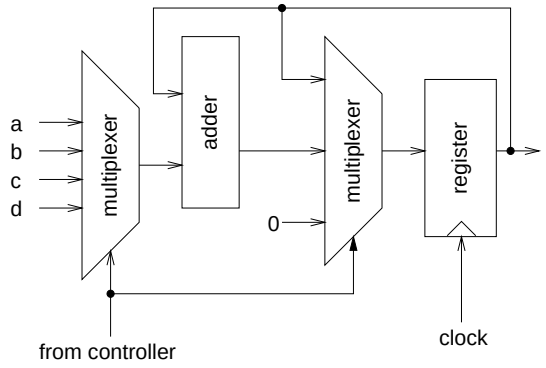
The diagram below shows a generic component in the datapath. Each RTL design will be composed of one of the following building blocks for each register. The structure allows the contents of each register to be updated at the end of each clock period with a value selected by the controller. The widths 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 45: 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. `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 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 46: 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 ;
```

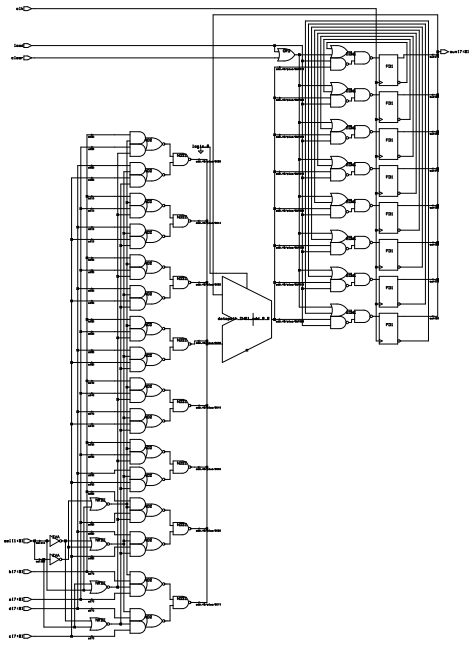
The next section of code is an example of how the datapath and the controller entities can be placed in a package, `averager_components`, as components. In practice the datapath and controller component declarations would probably have been placed in the top-level architecture since they are not likely to be re-used in other designs.

```
-- 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.

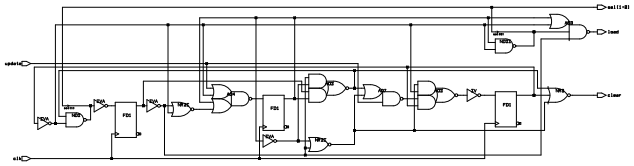
```
-- averager

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:

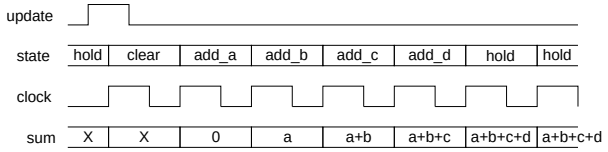


The register flip-flops are at the upper right, the adder is in the middle and the input multiplexer is at the lower left.

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.



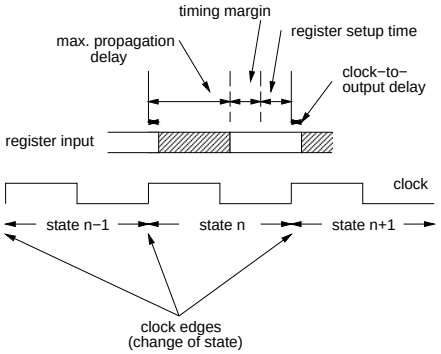
RTL Timing Analysis

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.

Behavioural Synthesis

It is possible to work at even higher levels of abstraction than RTL when design time is more important than cost. Advanced synthesis programs (for example, Synopsys' Behavioral Compiler) can convert a behavioral description of an algorithm into an RTL description. The compiler does this by automatically allocating registers and partitioning the processing over as many clock cycles as are required to meet high-level processing time requirements.

Metastability

Introduction

The proper operation of a clocked flip-flop depends on the input being stable for a certain period of time before (the setup time) and after (the hold time) the clock edge. If the setup and hold time requirements are met, the correct output will appear at a valid output level (between V_{OL} and V_{OH}) at the flip-flop output after a maximum delay of t_{CO} (the clock-to-output delay). However, if these setup and hold time requirements are not met then the output of the flip-flop may take *much* longer than t_{CO} to reach a valid logic level. This is called *metastable* behaviour or *metastability*.

An invalid logic level at the output of the flip-flop may be interpreted by some logic gates as a '1' and by others as a '0'. This leads to unpredictable and usually incorrect behaviour of the circuit.

In the synchronous circuits we have studied thus far we have been able to prevent metastability by clocking all flip-flops from the same clock and ensuring that the maximum propagation delay of any combinational logic path is less than the clock period minus the flip-flop setup time and clock-to-output delay.

However, when inputs to a synchronous circuit are not synchronized to the clock, it is *impossible* to ensure that the setup and hold times will be met. This will eventually lead to the incorrect behaviour of the device. It is important to realize that all practical logic circuits will eventually fail due to metastability. However, the designer should try to ensure that these failures happen very infrequently (e.g. once per 10^3 or 10^6 years of operation) so that other causes of failure predominate.

Computing MTBF

The average time between metastable outputs (mean time between failures or 'MTBF') is given by the formula:

$$\text{MTBF} = \frac{e^{C_2 t_{MET}}}{C_1 f_{clk} f_{data}}$$

where C_1 and C_2 are constants that depend on the technology used to build the flip-flop, t_{met} is the duration of the metastable output, and f_{clk} and f_{data} are the frequencies of the synchronous clock and the asynchronous input respectively.

Let's compute the MTBF assuming we used the lab FPGA board's internal oscillator as a clock to register the PC-104 bus IOR* signal. Since the clock and the signal being registered are coming from different oscillators the input is asynchronous. The clock frequency, f_{clk} is 25.175 MHz. The exact frequency of IOR* will depend on the program being executed, but let's assume a value of $f_{CLK2}/4 = 8.333/4 = 2.08$ MHz. For the Altera Flex10K family $C_1 = 1 \times 10^{-13}$ and $C_2 = 1.3 \times 10^{10}$. For correct operation of our circuit the settling time of the flip-flop output, the metastable time t_{MET}, must be less than the clock period minus the maximum propagation delays through the combinational logic elements minus the setup times of the other flip-flops in the circuit. The setup time of the -4 speed grade 10K20 input flip-flops, t_{IOSU}, is is 3.2 ns, thus $t_{MET} = t_{clk} - t_{PD} - t_{IOSU}$. If we assume t_{PD} is, for example, 30 ns, $t_{MET} = 39.7 - 30 - 3.2 = 6.5ns$ and the MTBF is:

$$\text{MTBF} = \frac{e^{1.3 \times 10^{10} \times 6.5 \times 10^{-9}}}{1 \times 10^{-13} \times 25.175 \times 10^6 \times 2.08 \times 10^6} \text{s}$$

which about 10^{33} seconds (a very long time).

Reducing Metastability

The simplest approach is to slow down the clock since this provides a longer time for the output of the flip-flop to reach a stable output value. Because the MTBF increases exponentially with t_{MET} a small reduction in clock frequency will often be enough to increase the MTBF to an acceptable value. However, in other cases this approach will be unacceptable because the resulting clock rate will be too slow.

Another approach is to use flip-flops with shorter setup and hold times (and correspondingly smaller C_1 and larger C_2 values). Whenever possible, these "metastable-hardened" flip-flops should be used on asynchronous inputs.

If this does not result in the desired degree of reliability it is possible to use two or more flip-flops in series. In this case the output of the second flip-flop will only be metastable if *both* flip-flop outputs were metastable. The disadvantage of this approach is that the input will now be delayed by one to two clock periods (instead of zero to one clock periods).

Input Synchronization

Inputs typically affect the results loaded into more than one flip-flop. For example, an input that controls state transitions in a state machine affects the various flip-flops that hold the encoded state. If an asynchronous input changes shortly before a clock edge, it is possible that the outputs of the combinational logic will not have reached their correct values when the flip-flops are loaded. This will almost certainly lead to inconsistent and incorrect behaviour. A circuit that exhibits unpredictable behaviour as a result of the timing of its inputs is said to have a *race condition.*

Such problems can be avoided by registering each asynchronous input using a single (preferably metastable-hardened) flip-flop and using the output of this flip-flop output to drive the rest of the logic. This results in a delay of up to 1 clock period before the circuit can respond to the changed input. Usually this is an acceptable trade-off for improved reliability.

Exercise 47: Draw the schematic of an input synchronizer.

As a general rule, **always synchronize (register) asynchronous inputs.**

Glitches

Glitches are short temporary changes in outputs that are caused by different propagation delays in a circuit. There are two reasons why glitches are undesirable.

The first set of problems is related to noise and power. Since glitches are short pulses much of their energy is at high frequencies and this power couples easily onto adjacent conductors. This induces

noise into other circuits and reduces their noise immunity. Glitches also cause power supply current spikes which result in voltage transients on the power supply lines. Another problem with glitches is that in CMOS logic families current consumption is proportional to the number of transistor switchings and glitches lead to increased current consumption.

The second set of problems arises when the digital output of one circuit is used as a clock in another circuit (e.g. to drive a counter or register). In this case glitches cause undesired clock edges (similar to switch bounce). In synchronous (single-clock) circuits these glitches are not a problem.

Glitches can be reduced by modifying the design of the combinational logic. However, this usually introduces additional logic. Glitches on signals that are confined to short paths within a circuit or inside a chip are usually tolerated. However, when outputs are brought off a chip, board or system (e.g. onto a bus) it is good practice to eliminate glitches.

The simplest way to eliminate glitches is to use a registered output signal. The output of a flip-flop changes only once, on the clock edge, and thus eliminates any glitches on its input. There are two ways to register outputs. Often it is possible to use register outputs directly such as when an output is already in a data register or when the signals are state machine state registers. The second method is to pass the signal through an additional flip-flop before it is output. The disadvantage of this method is that the output will be delayed by up to one clock period.

As a general rule, **always register outputs.**