SystemVerilog Guide

Zachary Yedidia

October 19, 2020

Contents

1 Introduction

This is a guide and reference for learning SystemVerilog, the hardware description language we will use to build circuits in CS141. This guide will help you to implement in SystemVerilog the various circuit components and techniques in digital design you learn through lecture. As we cover the fundamentals of digital design in the first half of the course, the teaching staff will hold sections to cover SystemVerilog and the CAD tools we use to program the FPGA boards. This guide will serve as notes for those sections. The first 7 sections cover usage of SystemVerilog for modeling combinational and sequential circuits, and the last 2 sections discuss the FPGA board we will use for the course and the CAD tools and development environment.

SystemVerilog is a language for describing and simulating digital systems. We can use SystemVerilog to describe a model of a digital circuit as logic gates, and then use it to simulate how signals will propagate through the system.

It is important to note that there are many differences between SystemVerilog and a traditional programming language. When using SystemVerilog, your code will be compiled to static logic gates and even though there is an order to the code, it will not be executed sequentially. It is often helpful to have an underlying circuit in mind when writing SystemVerilog code.

SystemVerilog is a very large language, with many features for both logic design and formal verification. We will focus on using the subset of SystemVerilog that can be actually synthesized into circuitry. Rigorous testing methodology using formal verification or other techniques is outside the scope of this guide. However we will touch on the basics of writing testbenches.

Hardware description languages like SystemVerilog often rely on the use of various idioms that synthesize to different hardware elements. This guide is meant as an introduction to SystemVerilog and the idioms it provides for creating digital circuits.

2 A Brief History

The Verilog hardware description language was written in the early 1980s for use at Gateway Design Automation. While it began as proprietary software, it was made open-source in 1989, and the first IEEE standard was released in 1995. Verilog was then updated by the IEEE multiple times, first in 2001, and then in 2005. The 2001 version of Verilog is the most widely used version and is usually referred to simply as "Verilog." By 2001 it became clear that Verilog needed an update to accomodate the increasing complexity in digital design. The IEEE began designing a substantial set of enhancements under the name *SystemVerilog*. These enhancements were useful both for modeling circuits and for the verification of those circuits. In 2005, a new standard of Verilog was created, and at the same time the enhancements were released and documented in the 2005 SystemVerilog standard.

Following 2005, work began at the IEEE to merge the two standards into one language. The subsequent language took on the name *SystemVerilog*, and was released in 2009. At this point Verilog was completely replaced by SystemVerilog. SystemVerilog received an update to the standard in 2012, and a minor update in 2017. Some advanced features of SystemVerilog remain unimplemented in various commercial SystemVerilog compilers. In this guide we will use the 2017 standard and stick to simple modeling features that are universally supported across HDL compilers.

3 Gate-level Combinational Modeling

SystemVerilog supports the modeling and design of digital circuits at different abstraction levels. We will begin with the simplest abstraction level: the gate (or structural) level.

3.1 Modules

In SystemVerilog, the *module* is the basic building block. Every module defines a set of input and output *ports* which specify the input and output signals of the circuit. After the port definitions come any internal signal definitions, finally followed by concurrent statements which specify the logic of the circuit.

We will begin with a simple combinational circuit: a 2-1 multiplexer. This logic circuit takes three inputs, a, b, and sel, and produces one output f. If sel is low, then $f = a$, otherwise $f = b$. We can express this using boolean logic as the expression $f = a \cdot \overline{sel} + b \cdot sel$. In SystemVerilog, this could be described with the following code:

```
module mux
    (
        input logic a, b, sel,
        output logic f
    );

    logic n_sel, f1, f2;

    and g1 (f1, a, n_sel);
    and g2 (f2, b, sel);
    or g3 (f, f1, f2);
    not g4 (n_sel, sel);
endmodule
```

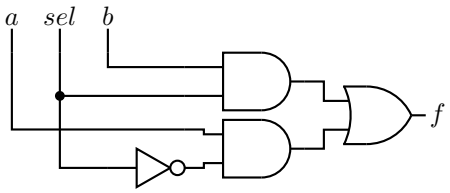


Figure 1: 2-1 multiplexer diagram

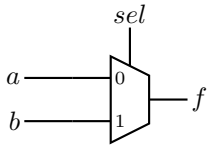


Figure 2: 2-1 multiplexer symbol

3.1.1 The logic data type

In this example, the inputs and outputs are declared as single bit *logic* data types. The *logic* data type is *4-state* and means the signal can store four values: 0, 1, X, or Z. The X value indicates a "don't care," or uninitialized value, and Z indicates a high impedance or "floating" value. This means that there is nothing driving the wire and should be avoided. The X and Z types are only relevant in simulation and help to catch design errors in the circuit: X values often indicate a value that was erroneously uninitialized, and Z values should be avoided in almost all cases because they mean that the wire's value was not properly specified and thus in hardware it could be anything.

Other data types exist in SystemVerilog, some synthesizable and some not. The only notable ones are `int` and `tri` (or `wire`).[1] The `int` data type indicates a 32-bit integer value which does not support X or Z values. It should not be used for synthesis but can be useful for loop iterators or integer values in testbenches. The `tri`/`wire` type can be driven by multiple signals at once and is useful for modeling tri-state gates or buses.

[1] `wire` and `logic` actually specify different characteristics of a signal. Using `wire` or `variable` specifies the *type* of the value (wires may have multiple drivers but unlike variables may not store values), and using `logic` specifies the *data type* (indicating two-state or four-state). If only `logic` is written, inputs are inferred to `wire logic` and the outputs to `variable logic`.

> Style guideline: use the `logic` data type for synthesis unless it is absolutely necessary to use a different data type.

3.1.2 Module instantiation

Following the port and internal signal declarations, we instantiate four primitive logic gates. SystemVerilog provides primitive logic gates as default modules. Each statement is *concurrent* in that it would not matter in what order the modules are instantiated. The result is a hardware circuit, and the order in which module instantiation statements are made does not matter. It is important to understand this in order to build working circuits with SystemVerilog.

The syntax for instantiating a module is

```
module_name instance_name (arg1, arg2, ...)
```

The instance name is helpful when debugging if you have multiple instances of the same module. Arguments can be provided in two different ways: "pass by position," and "pass by name".

In the multiplexer example above, the arguments are passed by position. This is the standard C style of passing arguments, where each signal is passed into corresponding port in the port ordering at the definition of the module. The primitive gates always provide the output as the first port, followed by the inputs.

To instantiate our mux with $a = X$, $b = Y$, $sel = Z$, and $f = W$, we would write

```
mux mux_unit (X, Y, Z, W)
```

In general, pass by position should be avoided in favor of pass by name. If the module is updated to include new ports, then all instantiations must be updated, and if you forget to update one instance, then the design error will be very difficult to find.

Instead, pass by name should be used, where each port value is assigned to the port name explicitly, as shown below.

```
mux mux_unit (.a(X), .b(Y), .sel(Z), .f(W))
```

Now we can reorder the ports however we want without needing to update instantiations. The arguments are tied by name to the proper ports.

If the port name and the signal name being passed into that port have the same name then the parentheses may be omitted. For example if X was renamed to a and Y was renamed to b we could perform the same operation as:

```
mux mux_unit (.a, .b, .sel(Z), .f(W))
```

To automatically perform this matching the `.*` syntax may be used. However, it is then hard to know exactly what ports the module defines and what signals are being passed in.

```
mux mux_unit (.*, .sel(Z), .f(W))
```

> Style guideline: always use pass by name and avoid using `.*` so that arguments are explicitly named.

3.1.3 Vectors

We can now use instances of our 1-bit 2-1 multiplexer to create a 2-bit 2-1 multiplexer. This circuit should take two 2-bit values and a 1-bit select signal and assign the 2-bit output to the correct input value depending on the select signal. Everything is the same except we are now passing 2-bit values.

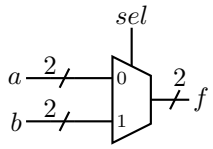


Figure 3: 2-bit 2-1 multiplexer symbol

The SystemVerilog code for this circuit is shown below.

```
module two_bit_mux
    (
        input logic [1:0] a, b,
        input logic sel,
        output logic [1:0] f
    );

    mux mux_1 (.f(f[0]), .a(a[0]), .b(b[0]), .sel);
    mux mux_2 (.f(f[1]), .a(a[1]), .b(b[1]), .sel);
endmodule
```

We now declare our signals as *vectors* using the `[N:0]` notation. This defines the valid indices (including 0) for our vector. Thus the declaration `logic [1:0]` creates a vector of 2 bits, and `logic [N-1:0]` will create one of length `N`. We can access the ith element by indexing into the vector at i with the `[i]` syntax, and we can access bit slices with `[i:j]` (inclusive).

Exercise

Using the 1-bit 2-1 mux from earlier, write a SystemVerilog module to model the 1-bit 4-to-1 multiplexer shown in Figure 4.

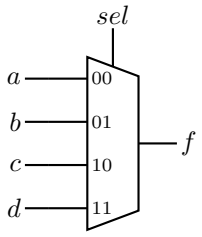


Figure 4: 4-1 multiplexer

3.2 Literals

SystemVerilog supports "integer literals" which are constant whole numbers with no fractional element. They can be expressed in a number of different ways. For simulation or synthesis, the SystemVerilog compiler must know (or assume) a number of characteristics about each integer literal, regarding its size, signedness, or base.

Values with no qualifying information, such as the value `5` are interpreted as 32-bit, signed, and decimal.

3.2.1 Binary, decimal, and hexadecimal literals

The base of a value may be specified by an apostrophe followed by a character specifying the base: `d` for decimal, `h` for hexadecimal, `b` for binary, and `o` for octal. For example, the value `'b0101` would be

interpreted as a 32-bit unsigned binary value. Note that the default is unsigned which is different from the plain integer literal.

A literal with a base may also contain `x` or `z` value, which will fill the digit with that value (`'hx` is `'bxxxx`).

3.2.2 Signed literals

By default, an integer literal with a base is unsigned. An `s` may be added after the apostrophe to make the value signed, such as `'sh7FF`. The signedness of a value will change how it behaves for certain operations that will be discussed later.

3.2.3 Sized literals

The size, or bit width, of an integer literal may also be specified by prepending a number before the apostrophe. This overrides the default bit width of 32 bits, and should be used in almost all cases when designing hardware. Below are some examples of integer literals with full qualifications:

Value	Base	Signedness	Size
`16'd5`	Decimal	Unsigned	16 bits
`4'sb0101`	Binary	Signed	4 bits
`12'h2FF`	Hexadecimal	Unsigned	12 bits

3.2.4 Size mismatch rules

SystemVerilog permits specifying a larger or smaller bit width than the number of bits required to represent the value. In such a case the upper bits will either be zero-extended (not sign-extended!) or truncated.

> Style guideline: specify the size and base of every literal. Only use the binary and hexadecimal bases (and in some cases decimal) as these have the most meaning in digital design.

3.3 Parameters

Parameters provide a way to define a constant value in a module that can be determined at compile-time. There are two types of parameters:

- `parameter` – the value of the constant can be specified when the module is instantiated.
- `localparam` – the value of the constant is internal to the current module.

Parameters are most useful for generalizing the bit width of a module. For example, you can write a general module for addition using a parameter, and each instance can then specify how many bits the adder should support. Parameters should be declared with a default value before the port definition, and may then be referenced anywhere in the module.

A local parameter should be declared within the module, and, as the name suggests, is local to the module. The following example showcases both types of parameters.

```
module n_bit_alu
    #(
        parameter N = 32
    )
    (
```

```
        input logic [N-1:0] x, y,
        input logic [2:0] opcode,
        output logic [N-1:0] s,
        output logic cout
    )

    localparam op_add = 3'b001,
               op_sub = 3'b010,
               op_cmpr = 3'b100;
    // ...
endmodule
```

By default when this module is instantiated `N` will be 32. However we can override this with the following syntax (for example for an 8 bit alu):

```
n_bit_alu #(.N(8)) alu_unit (...)
```

Modules may have multiple parameters which is why we also use pass by name for them (pass by position is also supported). Parameters are powerful because they allow us to generalize the modules we write to arbitrary sizes.

Certain literals can be given a size that is determined by context. This is useful for giving certain signals values even if they have parameterized size.

An unsized single-bit value like `'0` is a constant that should expand to `N` binary zeros as inferred by the context. The following literals support this form:

`'0`, `'1`, `'x`, `'z`

As we will see in the next section, SystemVerilog supports assignment, so as an example we could use `'0` like so

```
logic [N-1:0] x;
assign x = '0;
```

4 RT-level Combinational Modeling

The previous section discussed building modules up from simple logic cells. In this section we examine a more powerful abstraction level called the *register-transfer* level, where the HDL description of components such as adders, comparators, or decoders becomes much simpler to express.

4.1 Continuous Assignments

So far we have only seen how to build circuits starting with primitive logic gates and connecting wires from there. SystemVerilog provides a much more powerful tool for building circuits with the *continuous assignment*. A continuous assignment binds an expression to a signal. For example, instead of instantiating the primitive module for an AND gate, we could use a continuous assignment.

```
assign c = a & b;
```

Whenever any value in the expression on the right-hand side changes, the left-hand side is updated. This is not a one-time assignment but a *continuous* one. Note that continuous assignments, like module instantiations, are concurrent statements. This means that the following two pieces of code are equivalent.

```
logic a, b, c, d, e;

assign c = a & b
assign e = c | d
```

```
logic a, b, c, d, e;

assign e = c | d
assign c = a & b
```

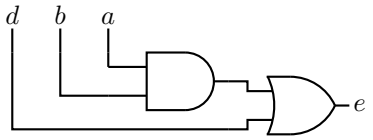


Figure 5: Corresponding circuit

There are many operators that are supported for expressions (not only limited to continuous assignments as will be discussed later) which are explained below.

4.1.1 Conditional operator

Conditional assignment is shorthand for performing the mux operation from earlier. It has the same syntax and behavior as the C ternary operator. Here is an equivalent implementation of the mux from earlier.

```
module mux
    (
        input logic a, b, sel,
        output logic f
    );

    assign f = sel ? b : a;
endmodule
```

This operator will work with arbitrary vector sizes, and is therefore much more extensible compared to the method of using module instantiation from before.

4.1.2 Bitwise operators

A bitwise operator takes one or two vectors as input and performs some bitwise operation on them, returning a vector of the same length. For example for the two vectors `a = 01101001` and `b = 01010011` the result of `a & b` will be `01000001`. The AND operation is performed between each bit of each vector such that $c_i = a_i \cdot b_i$. In other words the following two code blocks are equivalent:

```
assign c[0] = a[0] & b[0];
assign c[1] = a[1] & b[1];
assign c[2] = a[2] & b[2];
assign c[3] = a[3] & b[3];
```

```
assign c = a & b;
```

SystemVerilog supports the following bitwise operators.

Operator	Function
`a & b`	bitwise and
`a | b`	bitwise or
`a ^ b`	bitwise xor
`~a`	bitwise not

Notice how we can now model NAND and NOR as `~(a & b)` and `~(a | b)`.

4.1.3 Logical operators

Logical operators are similar to bitwise operators except they deal only with single bit values (true or false). A bitwise operator with single-bit operands performs the same function as the logical operators. However logical operators should not be used with multi-bit vectors.

Operator	Function
`a && b`	logical and
`a || b`	logical or
`!a`	logical not

Style guideline: use logical operators for scalar values that represent a true/false signal.

SystemVerilog also supports the following two logical operators for *simulation only*.

Operator	Function
`a -> b`	logical implication
`a <-> b`	logical equivalence

These operators cannot be synthesized and some simulators do not support them either. They should not be used.

4.1.4 Reduction operators

Reduction operators take a single multi-bit vector and output a single bit value. For example `&a` will take all the bits in the vector `a` and pass them into a single AND gate. Thus `&a` will only be one if `a` is

all ones.

```
assign b = a[0] & a[1] &
           a[2] & a[3];
```

```
assign b = &a;
```

SystemVerilog supports the following reduction operators.

Operator	Function
`&a`	reduction and
`|a`	reduction or
`^a`	reduction xor
`~^a`	reduction xnor
`~|a`	reduction nor
`~&a`	reduction nand

4.1.5 Arithmetic operators

So far the operators we have discussed very clearly map to some gate-level circuit and are therefore mostly shorthand for what we already know. Now we will look at arithmetic operators which SystemVerilog supports for synthesis. These are much more powerful because creating an adder is not trivial, and yet you can do so in SystemVerilog with a single continuous assignment.

The arithmetic operators supported by SystemVerilog are shown in the table below

Operator	Function
`a + b`	add
`a - b`	subtract
`-a`	unary minus
`a * b`	multiply
`a / b`	divide
`a % b`	modulus
`a**b`	power

Some of these operators, such as modulus, multiply, divide and power, require complex circuits to implement so care should be taken to ensure that the target FPGA or ASIC will support the operation. Often FPGAs will include macro cells for some of these operations, but you can check utilization reports to understand exactly how the operation is being synthesized.

4.1.6 Shift operators

SystemVerilog also has support for the standard shift operators, shown below.

Operator	Function
`a << b`	shift left logical
`a >> b`	shift right logical
`a >>> b`	shift right arithmetic

The arithmetical right shift preserves the sign of the value by copying in the sign bit into the newly available slots from the shift. The logical shift right extends with zeros, and the logical shift left does as well.

4.1.7 Comparison operators

Many comparison operators are also supported. These are explained below.

Operator	Function
`a == b`	equal
`a != b`	not equal
`a < b`	less than
`a <= b`	less than or equal
`a > b`	greater than
`a >= b`	greater than or equal

All of these operators are self explanatory. Be careful about the signedness of operands when using the relational operators (`>`, `<`...) because this will affect the output.

For equality checking, the operators `==` and `!=` only compare the values of 0 and 1 for the two inputs, and they do not check `X` or `Z`. This is because at the hardware level `X` and `Z` cannot truly be compared across signals. However, in a testbench it can be useful to compare all four states to check but bugs with uninitialized or unconnected values. For this purpose, SystemVerilog provides the following two operators.

Operator	Function
`a === b`	`a` equals `b` including `X` and `Z` values
`a !== b`	`a` not equals `b` including `X` and `Z` values

> Style guideline: the `===` and `!==` are not synthesizable. Use `==` and `!=` in synthesis and `===` and `!==` for testing.

Alternatively, we can treat `X` and `Z` values as "don't care" values in equality checking. This behavior is supported by the following operators which *are* synthesizable.

Operator	Function
`a ==? b`	equal with `X` and `Z` as "don't cares"
`a !=? b`	not equal with `X` and `Z` as "don't cares"

4.1.8 Concatenate and replicate operators

The concatenate and replicate operators are useful for extending a vector to a larger size. We may either concatenate two vectors together, or append a replicated vector (or single bit) to another vector.

Operator	Function
`{a,b}`	concatenate vector `a` with vector `b`
`{n{a,b}}`	concatenate vector `a` with vector `b` and replicate the result `n` times

For replication, `n` must be a literal value and cannot be a parameter. With replication, you may also provide a single vector to be replicated with the syntax `n{a}`.

These operators are very powerful. For example, we can easily implement sign-extension with the replication operator:

```
logic [31:0] sign_ext;
assign sign_ext = {{16{a[15]}}, a};
```

This sign extends a 16-bit value to a 32-bit value by replicating the most significant bit 16 times and then concatenating it with the original vector.

4.1.9 More operators

SystemVerilog supports even more operators (such as the pack and unpack operators) but we won't use them and they will not be covered here.

4.2 Generate

In some cases it might be useful to instantiate a configurable number of modules where the number is defined by a parameter. The `generate` statement is useful for creating this replicated structure. It has the following general syntax. The `generate` statement can be used to replicate both module instantiation and continuous assignments.

```
generate
    genvar [index_variables];
    for (...; ...; ...) begin [: optional label]
        ... concurrent_constructs ...;
    end
endgenerate
```

Consider the use of generate for building an N-bit comparator. Of course, we can use the XNOR reduction operator `~^`, but as an example we could also achieve this using generate and module instantiation.

```
module eq_n
    #(parameter N=4)
    (
        input logic [N-1:0] a, b,
        output logic eq
    );

    logic [N-1:0] tmp;

    generate
        genvar i;
        for (i = 0; i < N; i = i + 1)
            xnor gen_u (tmp[i], a[i], b[i]);
    endgenerate

    assign eq = &tmp;
endmodule
```

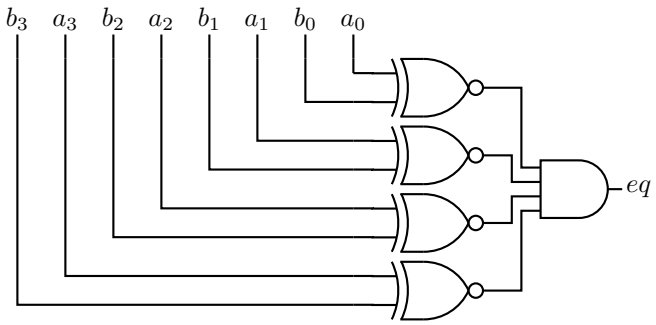


Figure 6: Corresponding circuit for $N = 4$

Think of generate as automating and generalizing (to a parameter `N`) copy-paste of module instantiation or continuous assignment.

4.3 Always block for combinational design

The always block is a powerful tool for modeling circuits with SystemVerilog. It is the central block for designing sequential logic, but can also be used for designing combinational logic, as will be discussed in this section. We will examine the **`always_comb`** statement in this section and later when designing sequential circuits, the general form of the always block as well as the **`always_ff`** and **`always_latch`** variations.

The **`always_comb`** statement takes the form of

```
always_comb begin
    [optional local variable declaration]
    ... procedural statement ...
end
```

It can be thought of as continuously looping and executing the internal statements whenever any of the input signals change. The **`always_comb`** block is similar to a multi-line version of the continuous assign statement. The **`begin`** and **`end`** keywords are optional if the block only contains one procedural statement (this is also the case for other constructs that use **`begin`** and **`end`**). Note that a single procedural statement may be multiple lines long, for example a case statement counts as a single procedural statement even though it may contain subblocks of procedural statements within it. Be careful when omitting the **`begin`** and **`end`** keywords because you must not forget to add them back if you decide to add more procedural statements to the block.

There are many possible procedural statements supported by SystemVerilog that may be used within an always block. Many of them do not have a clear physical counterpart and cannot be synthesized. Our focus will be limited to the following synthesizable statements:

- Blocking assignment
- Non-blocking assignment (discussed in the sequential section)
- If statement
- Case statement

Note that it is possible that an always block using only these procedural statements still may not be synthesizable.

Procedural statements are only valid inside procedural blocks – initial or always blocks, and are "executed" sequentially. The code is synthesized into hardware which must model the sequential semantics of the procedural statements even if hardware itself is not executed like a sequential program. In some cases it is not possible to convert the sequential semantics into hardware, resulting in non-synthesizable always blocks. It is very important to note the difference between the *semantics* of the program (the procedural description) and the actual synthesis of it into hardware.

A simple add-or-subtract module is shown below using `always_comb`

```
module add_or_sub
    #(parameter N = 4)
    (
        input logic [N-1:0] x, y,
        input logic add,
        output logic [N-1:0] z;
    );

    always_comb
        if (add)
            z = x + y;
        else
            z = x - y;
endmodule
```

Whenever `x` or `y` changes, the `always_comb` block is re-evaluated and a new value is assigned to `z`. In this simple case we could also use a single continuous assign statement to achieve the same effect:

```
assign z = add ? x + y : x - y;
```

> Style guideline: use `always_comb` when designing combinational logic with an always block. Do not use the general purpose `always` statement.

We are trying to design combinational logic, which is boolean logic where the output of a function is solely dependent on its input (there is no internal memory that could affect the output). With an always block we are not always guaranteed that every input has a well-defined output. Consider the previous module where the `else` block is removed:

```
always_comb
    if (add)
        z = x + y;
```

What is the output of the circuit if `add` is low? It is `z`, but the value of `z` depends on the values of `x` and `y` at the last time that `add` was high. It is possible that `x` and `y` have changed since then, but `z` must remember its value. This is not combinational behavior and is incorrect (in fact this infers to a latch, as will be examined later). Since `always_comb` is used the synthesis compiler should give a warning. To model combinational logic properly every output must be given a value for every possible procedural path that could be taken. In this case, for the path where `add` is low, `z` (the output) is not assigned, and thus this is not combinational.

> Requirement for combinational logic: every output must be given a value for every path in an `always_comb` block.

4.3.1 Blocking assignment

The syntax for the blocking assignment is shown below.

```
[variable_name] = [expression]; // blocking assignment
```

In the blocking assignment, the value of the expression is evaluated and immediately assigned to the variable thus *blocking* execution of further procedural statements. These are the same semantics as variable assignment in C.

The blocking assignment is used for combinational circuit design and the non-blocking assignment is used for sequential circuit design. The non-blocking assignment will be discussed in more depth in the sequential design section, and for now we will only use the blocking assignment.

> Style guideline: use blocking assignment only for combinational circuit design.

For example, due to the semantics of blocking assignment, the following code block synthesizes to a shift by four module:

```
always_comb begin
    logic tmp; // local variable
    tmp = in << 1;
    tmp = tmp << 1;
    tmp = tmp << 2;
    out = tmp;
end
```

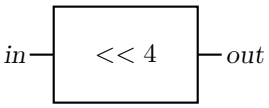


Figure 7: Shift by 4 circuit

Each previous line has an effect on the current value of `tmp`, thus allowing it to be updated after each statement. Blocking statements do not behave concurrently. This example is relatively simple so the compiler may see that this may be implemented as simply `out = in << 4`, but converting sequential semantics to hardware like this is not always possible.

4.3.2 If statements

The if statement acts much like an if statement in any other programming language. The general syntax is

```
if ([boolean]) begin
    ... procedural statement ...
end else begin
    ... procedural statement ...
end
```

The top branch of the if statement is evaluated if `[boolean]` is true, and the else branch otherwise. The else branch may also be omitted.

Let's now examine how such a statement can be synthesized. The if statement can come in multiple forms since the `else` statement is optional and may also contain further if statements (in the form of `else if` for example). Synthesis often depends on context but in general an if statement will either synthesize to a multiplexer or a priority routing network.

Simple if-else: If we have a simple if-else block where the same variable is assigned to in each branch then this can be synthesized as a multiplexer:

```
always_comb begin
    if (sel)
        f = a;
    else
        f = b;
end
```

Chained if-else-if: If a chain of if statements is made then this cannot be implemented as a simple multiplexer since the boolean expressions must be individually evaluated and the first one that is true must be selected (multiple may be true). This simply means that a series of multiplexers are used to ensure the correct order of evaluation (called a priority routing network).

If some outputs are not assigned for certain paths then some variables need to retain their value and the always block is no longer combinational. In such cases if statements with no else will synthesize to latches and if-else statements will synthesize to flip flops. We will see this in further detail in the sequential logic section.

4.3.3 Case statements

A case statement in SystemVerilog is very similar to a switch statement in C. The general syntax is

```
case [expr]
    [item]: begin
        [procedural statement]
        [procedural statement]
        ...
    end
    [item]: begin
        [procedural statement]
        [procedural statement]
        ...
    end
    [item]: begin
        [procedural statement]
        [procedural statement]
        ...
    end
endcase
```

The case statement evaluates the case expression and compares it against each item. If there is a match, the procedural statements in the matching block are evaluated. A `default` case may be specified as well, which will match if no earlier items match the case expression. This is extremely important for making sure that a case statement synthesizes to combinational logic, as in combinational logic, all control paths must be accounted for. The `begin` and `end` keywords may be omitted if there is only one procedural statement in the block. Note that if multiple case items match the expression, only the first one is evaluated. In such a case, a priority routing network will be inferred, much like the chained if-else-if case from above.

As an example, let's use a case statement to model an N-bit 4-1 mux. Modeling a 4-1 mux was an exercise from an earlier section which required using structural SystemVerilog. Now that we know about RT-level modeling, creating a 4-1 mux will be much easier.

```
module mux4_1
    #(
        parameter N = 2
    )
    (
        input logic [N-1:0] a, b, c, d,
        input logic [1:0] sel,
        output logic [N-1:0] y
    );

    always_comb begin
        case (sel)
            2'b00: y = a;
            2'b01: y = b;
            2'b10: y = c;
            2'b11: y = d;
        endcase
```

```
    end
endmodule
```

The SystemVerilog synthesizer is able to infer two things from this case statement:

- For every possible value of `sel`, there is an assignment to `y`, therefore the module is combinational
- All case items are mutually exclusive (it is not possible for `sel` to match multiple case items), therefore the module can be synthesized to a multiplexer rather than a priority routing network.

4.3.4 Unique case

In some cases, the case statement may be written such that the compiler cannot infer the two facts above, but the programmer is able to guarantee that they are true. In this case, the `unique` keyword can be used.

For example, the following 3-1 mux is valid.

```
always_comb begin
    unique case (sel)
        2'b00: y = a;
        2'b01: y = b;
        2'b10: y = c;
    endcase
end
```

The `unique` modifier informs the synthesis compiler that the case statement is in fact complete, and that all the cases may be evaluated in parallel (mutually exclusive). If `sel` takes on the value `2'b11`, anything may be assigned to `y`.

For simulation, using the `unique` modifier will enable runtime checks to make sure the guarantees are not violated.

4.3.5 Case inside

The `inside` keyword allows a case statement to match wildcard values. For example, the following circuit will assign `y = a` if the MSB of `sel` is set, `y = b` if the upper two bits of `sel` are `01`, etc.

```
always_comb begin
    case (sel) inside
        4'b1???: y = a;
        4'b01??: y = b;
        4'b001?: y = c;
        4'b0001: y = d;
        default: y = '0;
    endcase
end
```

> Style guideline: in original Verilog, this behavior was implemented with the `casex` and `casez` statements. These are now obsolete, and `case ... inside` should be used instead.

5 Modeling Sequential Circuits

We have seen how to build combinational circuits using SystemVerilog, where the output of the circuit depends only on its current inputs. Now we'll look at building sequential circuits, which are circuits with memory, where the output now depends on both the inputs and some internal state. Modern development of sequential circuits follows *synchronous design methodology*. This means that a common clock signal controls all global storage elements and data is stored in these elements only at the rising or falling edge of the clock signal. This allows us to separate the storage components from the rest of the circuit and greatly simplifies development. This methodology is important for designing and verifying large and complex digital systems.

In this section we will examine how the always block can be used to model latches, flip flops, and more complex circuits with state.

5.1 Always block

We will first examine the semantics of the general always block and then discuss how it may be used to synthesize hardware.

The general syntax for an always block consists of a *sensitivity list* followed by a sequence of *procedural statements*.

```
always @(... sensitivity list ...) begin [: optional name]
    [optional local variable declaration];

    ... procedural statement ...;
end
```

The sensitivity list is a list of signals to which the always block responds. When any of these signals change value, the always block executes. For example, in the following code block, the value of `sum` is updated whenever `a` or `b` changes.

```
always @(a, b) begin
    sum = a + b;
end
```

An always block with no sensitivity list will continuously execute acting as an infinite loop (this is not synthesizable). For hardware synthesis, we are especially interested in two special cases for the sensitivity list.

The first special case of the sensitivity list uses the `posedge` or `negedge` keyword. When this keyword is placed in front of a signal the always block only responds on the specified edge transition of that signal. This is critical for modeling sequential circuits such as flip flops that must respond to a rising clock edge.

The second special case is the `*` syntax. If the sensitivity list simply consists of a `*` this means that it will respond to all input signals that are accessed within it. This form accurately models combinational logic, as outputs of a combinational circuit should be updated immediately if any of the inputs are changed (outputs are functions only of their inputs). The syntax `always @(*)` may be abbreviated as `always @*`. In SystemVerilog the syntax `always_comb` was added specifically for this case. It is equivalent to `always @*`[2] and should be used instead, as `always_comb` gives the synthesis compiler more information about your intentions allowing the compiler to give useful warnings if your always block is not combinational.

[2]The `always_comb` and `always @*` notations are actually slightly different as `always_comb` fixes an issue where the `@*` notation might not include in the sensitivity list variables that are used in synthesizable functions (a topic not discussed).

5.1.1 Non-blocking assignment

The syntax of the non-blocking assignment is

```
[variable_name] <= [expression]; // non-blocking assignment
```

In a non-blocking assignment, the evaluated expression is assigned at the end of the always block[3], thus it does not block the execution of further statements. It may be better to think of the non-blocking assignment as "deferred" and the blocking assignment as "immediate." Note that two non-blocking assignments to the same variable will cause a race condition and should be avoided.

> Style guideline: use non-blocking assignment for sequential circuit design.

We will examine why non-blocking assignment must be used for sequential circuit design in the next section.

5.2 The D FF and register

The most fundamental data storage component is the D flip flop. The `d` signal is sampled on the rising edge of the clock and stored to the flip flop. A D FF may also contain a reset signal which resets the storage to 0 and depending on the implementation may be independent of the clock signal (asynchronous). The function table of the D FF is shown below.

clk	q*
0	q
1	q
↑	d

We wish to assign `d` to `q` only on the rising edge of the clock so we can model this in SystemVerilog with an always block that is triggered by `posedge clk`, and assigns `q <= d`. SystemVerilog provides the `always_ff` keyword, which tells the synthesis compiler your intention to make a flip flop, meaning warnings will be generated if the always block does not correctly model a flip flop. The `always_ff` keyword should be used instead of the plain `always`.

```
module d_ff
    (
        input logic clk,
        input logic d,
        output logic q
    );

    always_ff @(posedge clk)
        q <= d;
endmodule
```

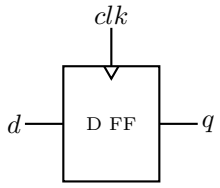


Figure 8: D Flip Flop

A D FF stores one bit, and a collection of D FFs can be put together, called a *register*, to store multiple bits.

[3]The result of the expression is actually assigned during the non-blocking assignment (NBA) event region, meaning that all non-blocking assignments from all always blocks for a time unit are pooled and performed at the same time .

> Style guideline: use `always_ff` for designing flip flops. Do not use the general purpose `always` statement.

5.2.1 More on blocking versus non-blocking

Now one might ask why we had to use a non-blocking assignment. Wouldn't it be the same with a blocking assignment? The answer is no and the reason is because in sequential circuits we want to determine the state of the system after the clock edge from the state before the clock edge.

Suppose we implemented our flip flop with

```
always_ff @(posedge clk)
    q = d;
```

This block is synthesizable and although the code works properly for a single FF, there will be problems in a more complex system with interacting flip flops.

Consider a case where we would like two flip flops (or registers) to swap data every clock cycle. The desired circuit is shown below.

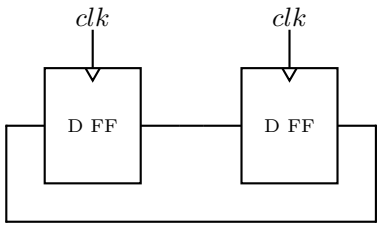


Figure 9: Two D FFs which swap data every rising clock edge

With our faulty flip flop we would try to implement this circuit with

```
always_ff @(posedge clk)
    a = b;

always_ff @(posedge clk)
    b = a;
```

At the rising edge of the clock, both always blocks are activated. It is undefined which always block executes first, but whichever one happens first will perform its blocking assignment before the other. When the second flip flop tries to store a value, the other flip flop will have already updated and the input will be incorrect. In fact, the second flip flop's currently stored value will not change! This code does not describe the swapping behavior and leads to a race condition where it will behave differently depending on which always block is executed first.

With a non-blocking assignment, both assignments would evaluate immediately but wait to perform the storage of the left-hand side until the end of both always blocks, where both assignments will happen at once, giving the intended swapping behavior.

5.2.2 Variations of the D FF

We often want a little more control over our flip flops, such as a reset signal which causes the flip flop to store predefined value instead of the input, or an enable signal such that the D FF can only sample an input when the signal is high. In the case of reset, designs can use either synchronous reset, where the flip flop is reset at the next clock edge, or asynchronous reset where the flip flop is reset immediately.

Asynchronous resets tend to be used in ASICs and synchronous resets in FPGAs. Either can be created from the vanilla flip flop through the use of clever AND gates, though target boards will often have them implemented as specific logic cells.

```
module d_ff_rst
    (
        input logic clk, rst,
        input logic d,
        output logic q
    );

    always @(posedge clk, posedge rst) begin
        if (rst)
            q <= '0;
        else
            q <= d;
    end
endmodule
```

The above SystemVerilog code implements a flip flop with asynchronous reset. This is due to the `posedge rst` in the sensitivity list. Remove that from the sensitivity list and it becomes synchronous.

It is also simple to implement a D FF with an enable signal. The only change is that we do not sample data if the `en` signal is low.

```
module d_ff_en
    (
        input logic clk, rst, en,
        input logic d,
        output logic q
    );

    always_ff @(posedge clk,
                posedge rst) begin
        if (rst)
            q <= '0;
        else if (en)
            q <= d;
    end
endmodule
```

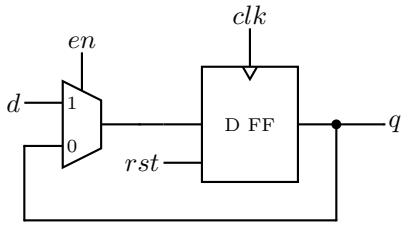


Figure 10: D FF with synchronous enable.

All the implementations thus far have been for flip flops, but it is very easy to extend them to any size register. The change is simply to make `d` and `q` vectors of the desired bit-width.

5.3 The D Latch

The D Latch is a less commonly used sequential circuit element and its use is a matter of debate. We will generally avoid using latches, however they are useful to know about especially because they are easily inferred by the synthesis compiler due to design errors.

Latches, like flip flops, have d and clk as inputs and output a q. However, they are not edge-triggered. Instead they are *transparent* when clk is high meaning that q is given d's value, and *opaque* otherwise meaning q remembers its value. The symbol for the D latch is shown below – note the lack of the triangle on the clock input means that the input is not edge-triggered.

We can model this behavior in SystemVerilog using an always block.

```
always @(clk, d)
    if (clk)
        q <= d;
```

Note that the sensitivity list for implementing a latch is the same as for combinational logic. However since q is not assigned when clk is low, it retains its value and thus a latch is inferred. This is problematic because synthesis compilers can't know if a latch was intentional or was part of a misdesigned combinational block. For this reason, instead of the general always block, the `always_latch` statement should be used (and `always_comb` for combinational modeling). As with `always_comb`, using `always_latch` automatically selects the sensitivity list for a latch (which should include every used variable).

```
module d_latch
    (
        input logic clk,
        input logic d,
        output logic q
    );

    always_latch
        if (clk)
            q <= d;
endmodule
```

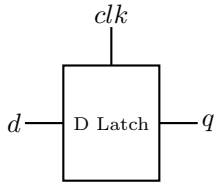


Figure 11: D Latch

> Style guideline: use `always_latch` when modeling latches. Do not use the general purpose `always` statement. Only use latches in your designs when necessary.

Exercise: What circuit does the following code synthesize to?

```
always @(clk, a, b)
    if (clk)
        q0 <= a & b;
    else
        q1 <= a | b;
```

5.4 General sequential circuit design

In general when designing sequential circuits we want to separate the sequential part from the rest of the design. We keep some internal state implemented as a register and then by wrapping the register's inputs and outputs with combinational logic we can implement more sophisticated circuits. In this section we introduce the fundamentals for building circuits that have internal state. The following concepts are expanded upon in the next section into the more powerful and general idea of *finite state machines*.

A general block diagram for building sequential circuits is shown below. It consists of the following parts:

- A *state register* which stores the internal state of the circuit.
- The *next-state logic* which is a combinational circuit that determines what the next state of the system should be given the current inputs and the current state.
- The *output logic* which is a combinational circuit that determines what the output of the system should be given the current inputs and the current state.

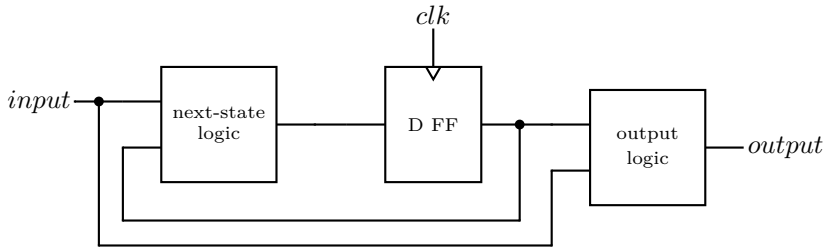


Figure 12: General sequential circuit block diagram

Our code development will follow this basic block diagram. We will see in the next section that this is actually an implementation of a *Mealy machine*.

5.5 Example: Shift register

A shift register is a register with a single bit input. On each rising clock edge the contents of the register are shifted over and the input bit is placed in the newly available slot. This is useful for performing *serial-to-parallel* conversion where a multi-bit value is sent one bit at a time and is loaded into the register and the individual bits can then be operated on in parallel.

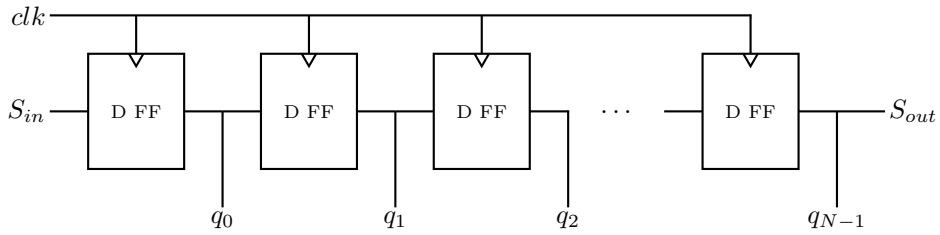


Figure 13: Shift register

Shift registers can also be modified to perform *parallel-to-serial* conversion by adding an N-bit input and a `load` control signal. When the load signal is high, the data is loaded (parallel load), otherwise the contents are shifted as usual. The last bit of the register can be read each clock cycle, effectively converting the parallel data into serial data.

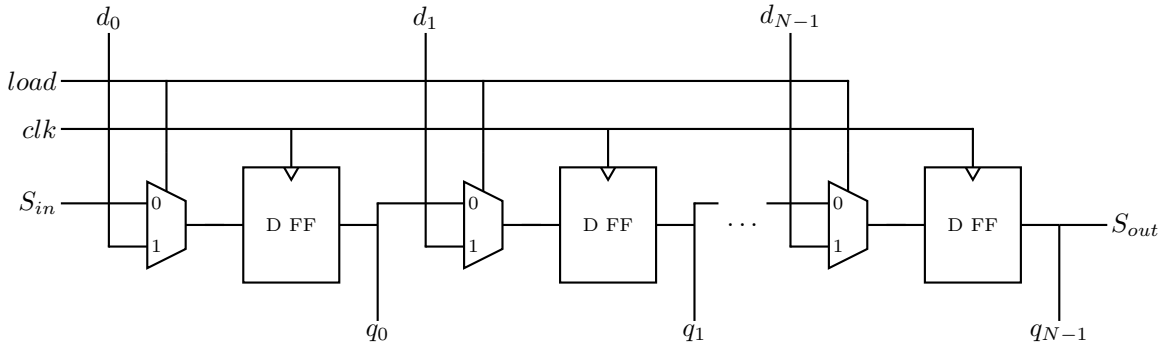


Figure 14: Shift register with parallel load

Let's write the SystemVerilog code to model a shift register with support for pausing, shifting, or performing a parallel load. The function table and SystemVerilog is shown below.

ctrl	Operation
00	pause
01	shift contents and load s_in
10	load d
11	-

```
module shift_reg
    #(parameter N = 8)
    (
        input logic clk, rst, s_in,
        input logic [1:0] ctrl,
        input logic [N-1:0] d,
        output logic [N-1:0] q,
        output logic s_out
    );

    logic [N-1:0] r_reg, r_next;

    // state register
    always_ff @(posedge clk, posedge rst)
        if (rst)
            r_reg <= '0;
        else
            r_reg <= r_next;

    // next-state logic
    always_comb
        unique case (ctrl)
            2'b00: r_next = r_reg;
            2'b01: r_next = {s_in, r_reg[N-1:1]};
            2'b10: r_next = d;
        endcase

    // output logic
    assign q = r_reg;
    assign s_out = r_reg[0];
endmodule
```

Exercise: A universal binary counter is a register with certain operations defined on its contents. It may count up, down, pause, load data, or clear its data, according to control signals. The control signals define which operation it should perform on the next rising edge of the clock. The behavior can be summarized by the following function table:

syn_clr	load	en	up	q*	Operation
1	-	-	-	0	synchronous clear
0	1	-	-	d	load
0	0	1	1	q+1	count up
0	0	1	0	q-1	count down
0	0	0	-	q	pause

The universal binary counter has 7 inputs: `clk`, `rst`, `syn_clr`, `load`, `en`, `up`, and `d` (`N` bits), and 1 output `q` (`N` bits).

1. Write SystemVerilog code to model this behavior

2. Verify its correctness with a testbench.

6 Modeling Finite State Machines

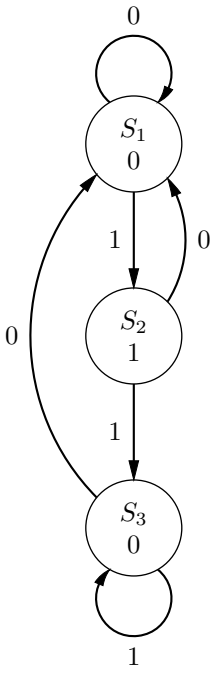
A *finite state machine* (*FSM*) is a model of computation which can be implemented in hardware as a sequential circuit. The machine has a set of inputs and outputs as well and can be in one of a set of *states* at a time. The current state of the machine and the current input are associated with a *transition* which determines the next state of the machine. The output is determined either by the current state or the combinational of current state and input. If the output is determined solely by the current state the FSM is called a *Moore machine*. If the output is determined by both the current state and the current inputs the FSM is called a *Mealy machine*.

In practice FSMs primarily serve as controllers of digital systems. For example an FSM is used to control the operation of a CPU datapath, which is composed of combinational logic and registers to store intermediate computations.

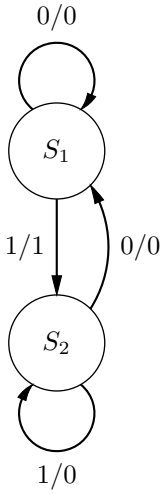
6.1 State diagrams

We use *state diagrams* to specify the behavior of an FSM. The state diagram is composed of *nodes* (states) and *transitional arcs* (state transitions). For a Moore machine, output values are placed inside the state node they are associated with (the output only depends on the state). For a Mealy machine the output is placed along the transitional arc (the output depends both on the state and the input).

Let's now construct a finite state machine that detects rising edges. It will have an input signal that varies with time, and when there is a transition in the signal from low to high, the FSM should output a pulse for one clock cycle. We can draw the state transition diagram for both Moore and Mealy implementations of the FSM.



(a) Moore implementation



(b) Mealy implementation

Figure 15: FSM implementation of a rising-edge detector

In the Moore machine implementation each time a rising edge is detected (a transition from 0 to 1) the FSM goes to the S_2 state for one unit of time and during that state the output is one. For the Mealy machine we are able to express the behavior using only two states: each time there is a transition from S_1 to S_2 we output a 1 along the transition. It is often the case that Mealy machines require fewer states then Moore machines to express the same behavior. However Mealy machines are often more

difficult to think about and as we'll see soon Mealy machine implementations are asynchronous because their output may change regardless of the clock. This means that generally a Mealy output will occur one clock cycle earlier than a Moore output.

6.2 General FSM circuit design

The general block diagram of an FSM is the same as the block diagram we saw last time for the sequential circuit except for two main differences. First, we now differentiate between "Moore" and "Mealy" output. Second, the next-state logic will be more complex as we now support arbitrary transitions between states. Previously, in the simple case of the counter, we just performed the same operation every time to the value stored in the state register.

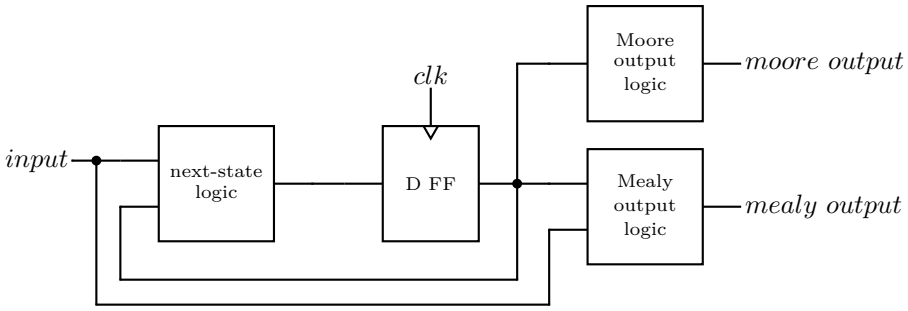


Figure 16: General FSM circuit block diagram

6.3 Enumerations

We will now implement the rising edge detector FSM with SystemVerilog. One useful SystemVerilog feature for encoding state bits is the *enumeration*. An enumeration is declared with a list of values and a name and defines a new type which can hold the given values. An enumeration for the states from our Moore state machine would be

```
typedef enum {s1, s2, s3} state_t;
```

We will not use user-defined types in SystemVerilog very much, but in the few cases we do the type names should end with the `_t` suffix. SystemVerilog will automatically assign some encoding to the enumeration, such as a one-hot encoding or a standard encoding. This decision is made to be optimal for your circuit. The programmer may also specify the exact bits for each value, for example to force a one-hot encoding:

```
typedef enum {s1=3'b100, s2=3'b010, s3=3'b001} state_t;
```

We will generally avoid doing this and let the synthesis compiler choose what is best.

6.4 FSM code development

With the FSM block diagram in mind we can implement the edge detector in SystemVerilog. After defining the state type with an enum, we declare two signals: `state_reg` and `state_next`. The `state_reg` stores the current state of the FSM and it is a register. The `state_next` signal is determined by combinational logic and is the input to the `state_reg` register.

```
typedef enum {s1, s2, s3} state_t;
state_t state_reg, state_next;

// state register
```

```
always_ff @(posedge clk, posedge rst)
    if (rst)
        state_reg <= zero;
    else
        state_reg <= state_next;
```

Now we just have to determine `state_next` and the output using combinational logic, and we can do so using an `always_comb` block where we check what state we are in and what the input is and set `state_next` accordingly. For a Moore machine, setting the output can be done once we have checked the current state, and for a Mealy machine this is done after checking the current state and the current input. The full Moore machine implementation is shown below.

```
module edge_detect_moore
    (
        input logic clk, rst,
        input logic in,
        output logic out
    );

    typedef enum {s1, s2, s3} state_t;
    state_t state_reg, state_next;

    always_ff @(posedge clk, posedge rst)
        if (rst)
            state_reg <= s1;
        else
            state_reg <= state_next;

    always_comb begin
        state_next = state_reg;  // default state: the same
        out = 1'b0;              // default output: 0
        unique case (state_reg)
            s1:
                if (in)
                    state_next = s2;
            s2: begin
                out = 1'b1;
                if (in)
                    state_next = s3;
                else
                    state_next = s1;
            end
            s3:
                if (~in)
                    state_next = s1;
        endcase
    end
endmodule
```

Note that default values are given to `out` and `state_next` in the `always_comb` block. This is a good trick to reduce the amount of code to write. Every signal is assigned for every path (so the block is combinational) but now we only have to assign to `out` and `state_next` when their value should be different from the default. Using blocking assignment allows overwriting like this.

Exercise: Write the SystemVerilog code for the Mealy machine implementation of the rising edge detector.

6.5 Mealy versus Moore

Now that we have hardware implementations of Moore and Mealy machines for the rising edge detector we can take a closer look at the differences in timing between the two. The main difference in timing is that the Moore machine output only changes on the rising edge of the clock while the Mealy machine output may change at any time. This means that the Moore output may change up to one clock cycle after a Mealy output for the same FSM. This difference is illustrated in the timing diagram below.

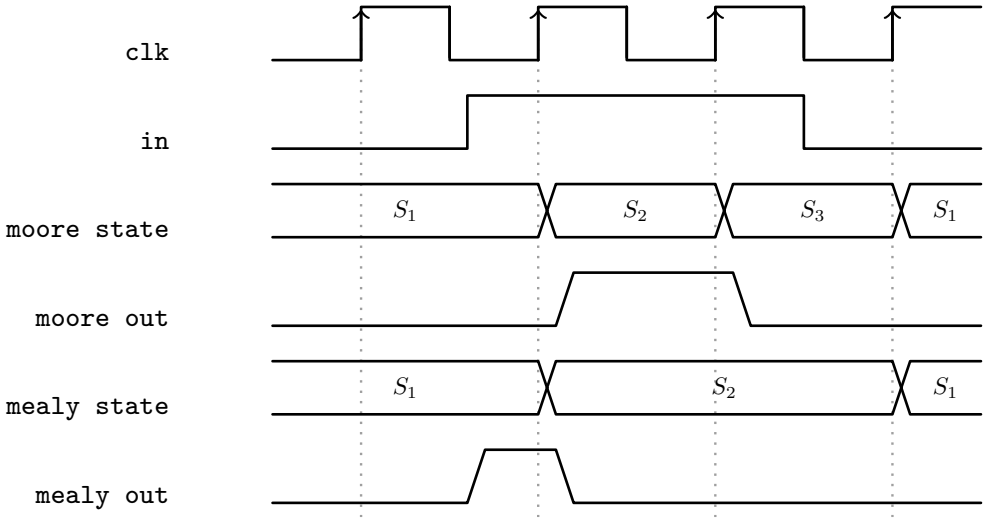


Figure 17: FSM timing diagram

The Mealy output changes immediately and it requires fewer states, but the width of its pulse depends on when the rising edge occurred with respect to the clock signal which is not ideal. The choice of Moore versus Mealy should be made within the context of the design, but generally Moore machines are more commonly used.

Exercise: Suppose there is an intersection between street A and street B. The inputs T_A and T_B specify if there is traffic on streets A and B respectively, and the outputs of the FSM L_A and L_B give the values for the lights on each street (red, yellow, or green). Initially, the light on street A is green and the light on street B is red. L_A remains green until there is no more traffic, at which point it should become yellow for 5 seconds and then go to red, while L_B goes to green. L_B should then remain green until there is no more traffic on street B, at which point the lights should swap again.

1. Draw the state transition diagram for a Moore machine implementation of this FSM.
2. Implement the FSM in SystemVerilog and verify its operation on an FPGA. What extra considerations need to be taken into account to ensure that the yellow light lasts 5 seconds?

7 Modeling Memory

It is often necessary for circuits to have access to more complex forms of storage than simply flip flops and registers. In this section we will discuss how to model register files which are addressable arrays of registers, as well as different kinds of memories which utilize the denser block SRAM macro cells on the FPGA.

7.1 Register file

A register file is simply a collection of registers with a wrapping circuit that allows reading or writing certain registers according to a set of addresses. Fundamentally, memory in SystemVerilog can be modeled with a two dimensional array declared as

```
logic [N-1:0] mem [0:M-1];
```

This declares a variable `mem` which is a `M`-by-`N` array of data. In certain situations such as for modeling memory, SystemVerilog has support for *dynamic indexing* where the index into an array may be a signal (i.e. not known at compile-time). This makes implementing memory very simple.

FIGURE

```
// 32x32 register file by default
module reg_file
    #(
        parameter DATA_N = 32,
                  SIZE   = 32
    )
    (
        input logic clk, wr_en,
        input logic [$clog2(SIZE)-1:0] w_addr, r0_addr, r1_addr,
        input logic [DATA_N-1:0] w_data
        output logic [DATA_N-1] r0_data, r1_data
    )

    logic [DATA_N-1:0] regs [0:SIZE-1];

    always_ff @(posedge clk)
        if (wr_en)
            regs[w_addr] <= w_data;

    assign r0_data = regs[r0_addr];
    assign r1_data = regs[r1_addr];
endmodule
```

This code implements a register file with one write port and two read ports. When `wr_en` is high, `w_data` is written to the register specified by `w_addr`. In addition, `r0_data` and `r1_data` will contain the data of the two registers specified by the read addresses `r0_addr` and `r1_addr`. Note that the dynamic indexing will need to infer decoding and multiplexing logic. This is not trivial and should only be used when the synthesis compiler is able to recognize the pattern and infer it correctly. We also assign outputs *asynchronously* so the output read data will change as soon as the input read address changes, ignoring clock edges. This is a key difference between a register file and RAM (RAM must be synchronous).

7.2 RAM

Flip flops and registers are very fast and flexible but are not very dense compared to other forms of memory such as SRAM or DRAM. When dense storage is needed it is far better to use RAM modules. FPGA devices often provide "block RAM" which is an embedded SRAM memory implemented in a macro cell. For small memory implementations logic cells organized as distributed SRAM may also be used instead of the larger block RAM. Using these macro cells involves writing SystemVerilog code according to a template that is correctly inferred by the synthesis compiler. In order to make sure that these templates are correctly inferred, RAM implementations should not be mixed with the rest of the design and should instead be implemented in a module on its own and then structurally instantiated where necessary.

7.2.1 Single-port RAM

The simplest implementation of a memory is a synchronous single-port RAM. In this template we have a single port that can be used to read or write.

```
module single_port_ram
    #(
        parameter DATA_N = 32,
                  SIZE   = 128
    )
    (
        input logic clk,
        input logic wr_en,
        input logic [$clog2(SIZE)-1:0] addr,
        input logic [DATA_N-1:0] w_data,
        output logic [DATA_N-1:0] r_data,
    );

    logic [DATA_N-1:0] ram [0:SIZE-1];

    always_ff @(posedge clk) begin
        if (wr_en)
            ram[addr] <= w_data;

        r_data <= ram[addr];
    end
endmodule
```

The code above implements a single-port memory with 128 blocks of 32 bits each. Note that unlike in the register file the `r_data` signal is assigned inside the always block. This makes the memory *synchronous* as the output is only changed on a clock edge, and therefore the synthesis compiler can compile this design to denser RAM modules.

7.2.2 Dual-port RAM

A more general RAM is shown below. This is a synchronous dual-port memory. This is the same as earlier except now the RAM has two ports, which can each be used to read or write.

```
module dual_port_ram
    #(
        parameter DATA_N = 32,
                  SIZE   = 128
    )
    (
```

```
        input logic clk,
        input logic we0, we1,
        input logic [$clog2(SIZE)-1:0] addr0, addr1,
        input logic [DATA_N-1:0] w0_data, w1_data,
        output logic [DATA_N-1:0] r0_data, r1_data
    );

    logic [DATA_N:0] ram [0:SIZE-1];

    always_ff @(posedge clk) begin
        if (wr0)
            ram[addr] <= w0_data;

        r0_data <= ram[addr];
    end

    always_ff @(posedge clk) begin
        if (wr1)
            ram[addr] <= w1_data;

        r1_data <= ram[addr];
    end
endmodule
```

7.2.3 Pre-loading data

If desired, data can be pre-loaded into the RAM using the `$readmemh` or `$readmemb` functions. These functions are system tasks which read a *pattern file* at compile-time and embed the data into the bitstream to be loaded into the memory when the system starts up. The pattern file is written as a text file with each line containing the data for the specified address. The `$readmemh` and `$readmemb` tasks read hexadecimal and binary data respectively. An `@` token followed by a hexadecimal address may precede a line and instructs the following patterns to be loaded starting at that address. An example of a hexadecimal pattern file is

```
00642824
@A7 // load the next pattern at 0xA7
20020005
2003000c
2067fff7
// ...
ac020054
```

Both readmem tasks take the same arguments:

- `$readmemh("file", array, start_addr, end_addr)`: read the hexadecimal pattern file `file` into the memory called `array` starting at `start_addr` and ending at `end_addr`. The address arguments are optional.
- `$readmemb("file", array, start_addr, end_addr)`: read the binary pattern file `file` into the memory called `array` starting at `start_addr` and ending at `end_addr`. The address arguments are optional.

To actually perform the read, we place the function call in an `initial` block within the module:

```
initial
    $readmemh("mem.txt", mem);
```

Initial blocks are usually not synthesizable but in this case synthesis compilers will recognize the template and infer the corresponding load into the memory correctly.

7.3 Additional considerations

It is impressively easy to model memory with SystemVerilog because the synthesis compiler is able to recognize templates for us. However, care must be taken to make sure synthesis is correct. For example creating a tri-port memory is not as simple as adding another port. The memories on the FPGA device are dual-port so the synthesis compiler cannot map the design to the memories like it can for the dual or single port models. Additionally, BRAM modules have a minimum capacity of 16K and thus will not be inferred unless the memory is large enough (normal logic cells will be used instead). Usually the synthesis compiler reports can be checked to see how the memory was inferred.

In simulation, a memory array may also take up twice as much virtual memory as in hardware because the `logic` data type is four-state. In certain cases where large memories need to be simulated it is helpful to use the `bit` data type instead which is two-state. Be careful when using the `bit` data type though as uninitialized values will be `0` instead of `X` and this may hide bugs.

8 FPGA Devices

The information here is only provided so you understand a little more what kind of technology you are working with when using an FPGA. You do not need to understand it in any depth in order to write hardware for FPGAs and only a small overview is given here.

An *FPGA* or *field-programmable gate array* is an integrated circuit (IC) which is useful because it can be reconfigured without needing to refabricate the device (programmable "in the field"). Manufacturing a chip is very time consuming and expensive so it is important to make sure that the design will work before doing so. Simulation software is powerful but implementation on an FPGA shows that the design works in real hardware. Additionally, while an FPGA incurs some overhead in processing power compared to a custom IC, FPGAs can often be used to greatly speed up certain workloads at a fraction of the cost of manufacturing a chip.

FPGA devices are made up of *logic cells*, *macro cells*, and *programmable switches*.

- A logic cell usually contains a small combinational circuit called a *LUT* (*lookup table*) and a D flip flop. The LUT is a small memory which can be used to implement any n-input combinational logic function The output of the LUT can then be the output of the logic cell, or may be sampled by a D flip flop to implement sequential circuit elements. There is also a small 1-bit memory cell to determine the select signal for the multiplexer. When a program is loaded onto the FPGA, the correct memories are programmed by sending the data along the global data bus and correctly controlling the write enable (wr) signals for all memories.

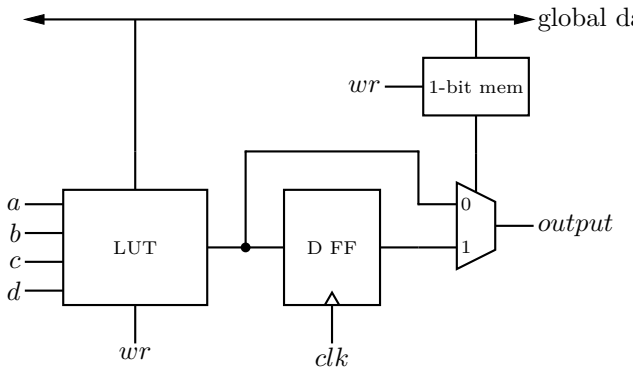


Figure 18: FPGA logic cell

- Using purely logic cells is possible but a lot of overhead is incurred compared to creating application specific (and non-reconfigurable) hardware. To ease this overhead, circuits for common operations are usually provided in the form of macro cells. These are designed at the transistor level and are meant to complement the logic cells. Depending on the FPGA device, these may include memory blocks, multipliers, clock management circuits, and more. Some FPGA devices even come with prefabricated CPUs.
- The programmable switches connect all the different cells in the FPGA according to the programmed design. All the cells are connected in a grid with programmable switches at each junction.

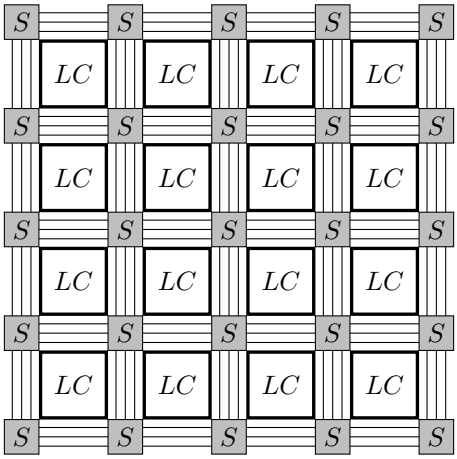


Figure 19: FPGA layout where S means switch and LC means logic cell

Each logic cell is connected to the neighboring wires by multiplexers on the left side to determine the inputs (a, b, c, d) and tri-state gates on the right so that the proper wires can be used to route the output of the cell.

9 Development for the Nexys A7

This section provides information for actually applying the SystemVerilog concepts learned previously to run hardware on an FPGA device. This guide is specifically geared towards the Nexys A7 FPGA board because it is a good beginner board with a variety of peripherals for more advanced projects. The information in this section should also be applicable to other FPGA boards although the development flow outlined below is tighly coupled with Xilinx Vivado so using non-Xilinx boards would require significant modification.

9.1 Overview of the Nexys A7

The Nexys A7 is a basic general-purpose FPGA board designed for use in education. It features a Xilinx Artix-7 FPGA unit surrounded by many peripherals such as DDR memory, an ethernet port, a VGA port, a temperature sensor, and accelerometer, a microphone, and more. The board also has a great built-in eight-digit seven-segment display. This, along with the 16 switches and 16 LEDs, make it easy to view feedback from simple designs. The Nexys A7 also supports soft core processors such as the Xilinx MicroBlaze, and with all these built-in peripherals this makes building a simple SoC design easier. The board can be powered by USB (no need to find an outlet) and can run at clock speeds of 450 Mhz.

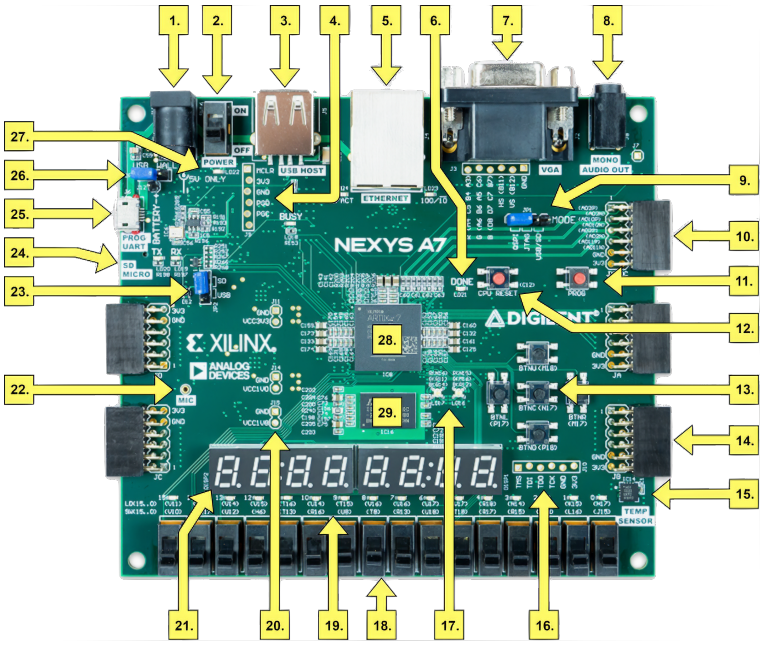


Figure 20: Nexys A7 features

Number	Component	Number	Component
1	Power jack	16	JTAG port for (optional) external cable
2	Power switch	17	Tri-color (RGB) LEDs
3	USB host connector	18	Slide switches (16)
4	PIC24 programming port	19	LEDs (16)
5	Ethernet connector	20	Power supply test point(s)
6	FPGA programming done LED	21	Eight digit seven-segment display
7	VGA connector	22	Microphone
8	Audio connector	23	External configuration jumper (SD / USB)
9	Programming mode jumper	24	MicroSD card slot
10	Analog signal Pmod port (XADC)	25	Shared UART/ JTAG USB port

Number	Component	Number	Component
11	FPGA configuration reset button	26	Power select jumper and battery header
12	CPU reset button (soft cores)	27	Power LED
13	Five pushbuttons	28	Xilinx Artix-7 FPGA
14	Pmod ports	29	DDR2 memory
15	Temperature sensor		

The Artix-7 FPGA has the following features:

Component	Nexys A7-100T
Look-up tables (LUTs)	63,400
Flip-flops	126,800
Block RAM	1,188 Kb
DSP slices	240
Clock management tiles	6

For more information, visit the Digilent Nexys A7 Reference Manual here.

9.2 Top modules and constraint files

The top module of a SystemVerilog project is the module that will be instantiated onto the FPGA board. When the module is instantiated the input and output ports are connected according the mappings specified by the *constraint file*.

Constraint files map specific I/O pins on the FPGA board to SystemVerilog variable names. Constraint files are usually provided by the FPGA distributor or can be found online. For example, the following excerpt from the Nexys A7 constraint file maps a 100 Mhz clock to the `clk` port of the top module, as well as some of the board's LEDs and switches:

```
// clock
set_property -dict {PACKAGE_PIN E3 IOSTANDARD LVCMOS33}
    [get_ports clk]
create_clock -period 10.000 -name sys_clk_pin -waveform {0.000 5.000}
    -add [get_ports clk]

// four switches
set_property -dict {PACKAGE_PIN J15 IOSTANDARD LVCMOS33}
    [get_ports {sw[0]}]
set_property -dict {PACKAGE_PIN L16 IOSTANDARD LVCMOS33}
    [get_ports {sw[1]}]
set_property -dict {PACKAGE_PIN M13 IOSTANDARD LVCMOS33}
    [get_ports {sw[2]}]
set_property -dict {PACKAGE_PIN R15 IOSTANDARD LVCMOS33}
    [get_ports {sw[3]}]

// four LEDs
set_property -dict {PACKAGE_PIN H17 IOSTANDARD LVCMOS33}
    [get_ports {led[0]}]
set_property -dict {PACKAGE_PIN K15 IOSTANDARD LVCMOS33}
    [get_ports {led[1]}]
set_property -dict {PACKAGE_PIN J13 IOSTANDARD LVCMOS33}
    [get_ports {led[2]}]
set_property -dict {PACKAGE_PIN N14 IOSTANDARD LVCMOS33}
    [get_ports {led[3]}]
```

In the top module we define

```
module design_top
    (
        input logic clk,
        input logic [3:0] sw,
        output logic [3:0] led
    );

    // instantiate design using clk, sw, and led...
endmodule
```

Make sure to select the top module with the "set as top" option in Vivado so that it is used for synthesis. The filename should be bold in the navigator view when this is the case. Additionally, in the simulation sources, make sure to select your testbench as the top module because that is the file you would like to use for behavioral simulation (the synthesis top module should not be used for obvious reasons).

9.3 Vivado project organization

Vivado is very useful because it provides an integrated development environment for working on a SystemVerilog project, simulating the code, synthesizing and uploading to the FPGA all with a fairly intuitive GUI. However it is notoriously difficult to use with version control. Vivado supports two design flows

- *Project mode*: Vivado controls the project files, automatically creates reports during synthesis, displays simulation results easily, and provides the flow navigator window which allows the programmer to easily switch between synthesis, simulation, and the hardware manager for programming the board. Unfortunately since Vivado controls the files, it is difficult to set up version control for this mode.
- *Non-project mode*: Vivado commands for synthesis, simulation, and programming are executed via TCL manually at the command line. This gives the programmer full control over the files and allows for direct access to the project files and easy version control. However, everything must be done manually which can be difficult and warnings and errors are hard to spot at the command line. Logic simulation is especially difficult to run from the command line, and Vivado must be booted up on every simulation run.

We will attempt to use the best of both worlds by using TCL scripts to generate projects on the fly from SystemVerilog files in version control. The projects can be placed in the `.gitignore` and regenerated when the repository is cloned. Synthesis and simulation can be accessed from inside project mode and the files can be edited in the Vivado IDE or in another text editor. A project will contain the following files

- The `constraint` directory contains a `.xdc` file which is the constraint file for the project.
- The `hdl` directory contains all the SystemVerilog source files
- The `tcl` directory contains TCL scripts for generating, refreshing, and opening the Vivado project associated with the rest of the source code.
  - `make.tcl`: generates the Vivado project in the `build` directory and adds all hdl and constraint files as relative files (source files are not copied into the project).
  - `refresh.tcl`: refreshes the project files if any new SystemVerilog files are added to the `hdl` directory.
  - `open.tcl`: opens the project with the Vivado IDE.
  - `synth.tcl`: synthesizes the design from the command line, and creates a bitstream and utilization report. These files are placed in `synth_output/`.
  - `upload.tcl`: uploads the bitstream in `synth_output/` (generated by `synth.tcl`) to the FPGA device.

- One bash script: `tcl.sh` which will run the corresponding TCL scripts with Vivado in batch TCL mode. It must be provided one argument, which is the name of the TCL script to run (without the extension). For example `./tcl.sh make` will run the `make.tcl` script.

For example the mux example project has the following structure:

```
constraint/
    Nexys_A7.xdc
hdl/
    mux_top.sv
    mux.sv
    mux_tb.sv
tcl/
    make.tcl
    open.tcl
    refresh.tcl
    synth.tcl
    upload.tcl
tcl.sh
build/ # This directory is created by running ./tcl.sh make
```

The `build/` directory (and the `.Xil/` directory) should be placed in the `.gitignore` for the repository. All other files should be committed to Git.

Files can be edited at the command line or in Vivado. Once a project is generated it can be opened with `./tcl.sh open` and from there can be simulated, synthesized, or uploaded to the FPGA board. If you would like to delete the project simply delete the `build/` directory.

9.4 Editing code with Vivado

After initially downloading the project zip file you should see a structure like the one shown above for the mux project (without the `build/` directory). You may choose to edit the HDL source files with a text editor rather than Vivado, though the following instructions will be useful regardless as Vivado is necessary for simulating. To create a Vivado project run the `tcl.sh` script at the command line

```
./tcl.sh make
```

This should generate a `build/` directory containing a Xilinx Vivado project where the files in it are linked to the files in the `hdl` folder (and the constraint file). Now that the project is created you will not need to run the `make` script again. If you ever create new files in the `hdl` directory, to add them to the project use the `refresh` script by running `./tcl.sh refresh`. You can also remove the `build` folder and regenerate the project with the `make` script. To remove files from the project delete them from inside Vivado and then remove them from `hdl/` using the command line.

To open the project run

```
./tcl.sh open
```

Vivado will open in the background (it can be quite slow so give it time) and you should see the following:

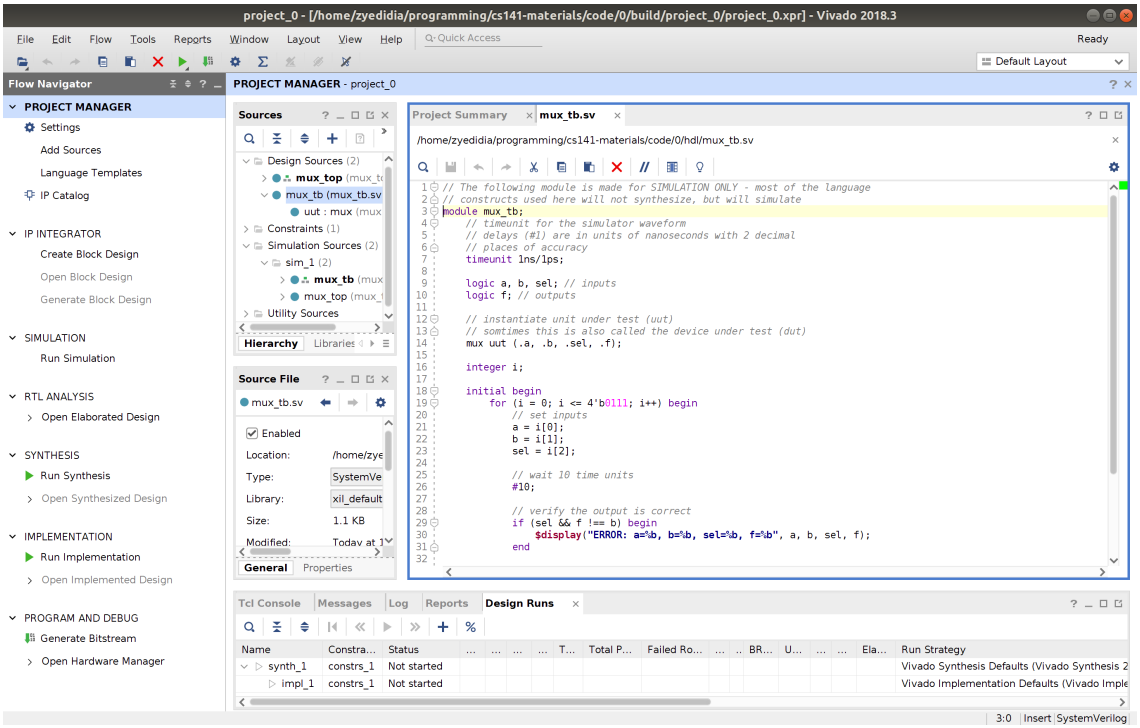


Figure 21: Vivado development window

In the sources panel you should see two folders: "Design Sources" and "Simulation Sources." These should contain the same SystemVerilog files, however you will want their top modules to be different. The top module is the module contained in the bolded file. In the picture above, `mux_top` is the top module for design and `mux_tb` is the top module for simulation. This is correct since `*_top` should be used for synthesis ("Design Sources") and `*_tb` should be used for simulation (when you click "Simulate" the top module in "Simulation Sources" will be run). It is likely that the incorrect files are the top modules when you first open the project. To fix this right click the file you want to be the top module and select the "Set as Top" option.

Double click files in the "Sources" panel to open them in the editor. Errors and warnings should be shown in the editor as you type or displayed in the console at the bottom of the screen.

9.5 Simulation

To run simulation, click on "Run Simulation" in the flow navigator on the left side. Choose "Run Behavioral Simulation." This will simulate your design for 10 microseconds or until a `$finish` statement is reached in the testbench. If a `$finish` statement is reached, the code and line where it is located will be shown. Otherwise a waveform at the current time will be displayed. Any messages that were displayed using `$display` or `$monitor` will be shown in the console at the bottom of the screen. Hopefully you don't see any errors displayed there for the example project. To view the waveform after a `$finish` statement has been reached, select the "Untitled" tab. Select "Zoom fit" to fit the entire waveform to the view.

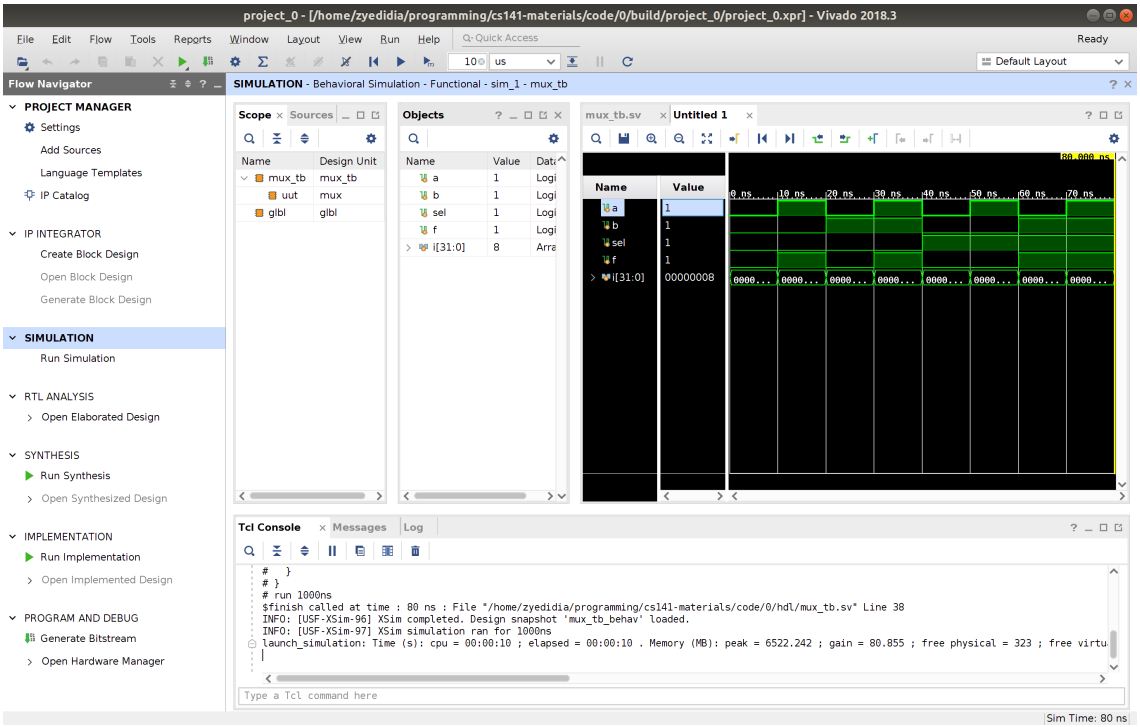


Figure 22: Simulator waveform viewer

You can also select a signal in the waveform viewer (such as `a` which is selected in the picture) and press the arrow keys to navigate to every place where that signal changes. Notice that we are only able to view signals that are exposed to the testbench. The internal mux signals `f1`, `f2`, and `n_sel` are not viewable. To view internal signals, select the module to inspect from the "Scope" panel on the left. We want to view signals internal to `uut` (here is where module instantiation names are useful) so select it after dropping down `mux_tb`. In the "Objects" panel you should now see the internal signals. To view them in the waveform viewer select them and drag them to the right. Then click "Relaunch simulation" which is the rightmost option in the upper toolbar (it is an arrow pointing clockwise in a circle). The waveform should then be populated with the selected internal signals.

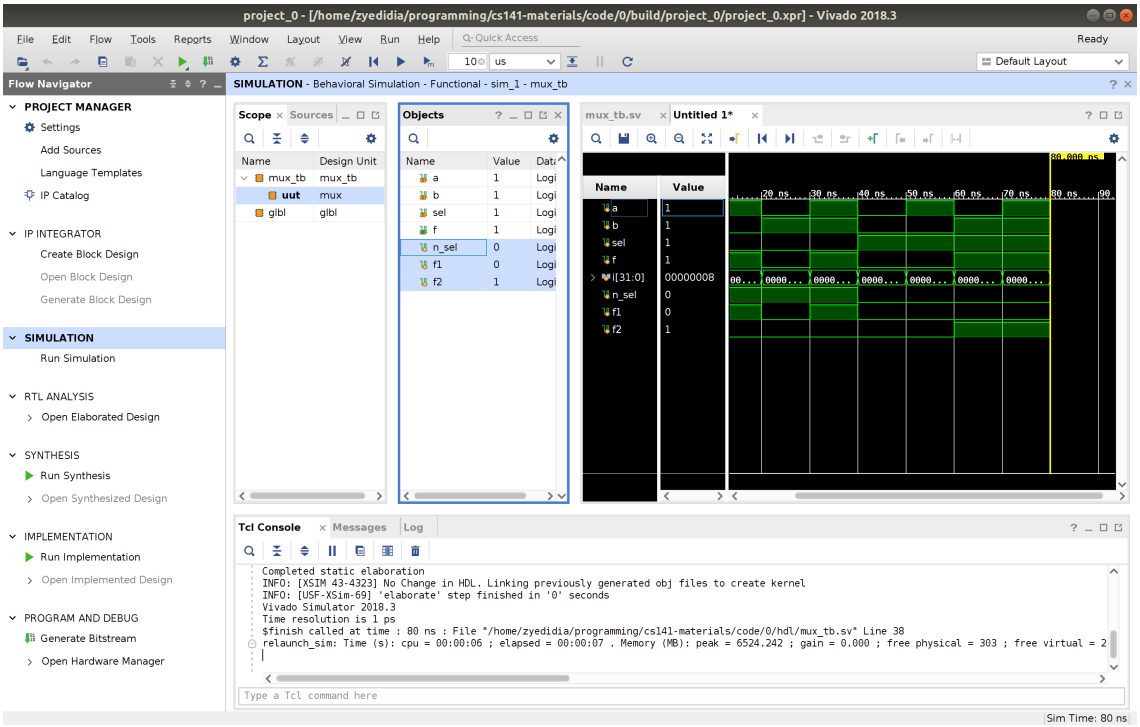


Figure 23: Simulator waveform viewer with internal signals

Feel free to play around and get used to the different options. You can also close the "Scopes" and "Objects" panels to give more space for the waveform. You can always get them back by navigating to `Layout->Default Layout`. You may close the simulator with the blue X when you are done, or you can leave the simulator open and click "Relaunch simulation" every time you change your SystemVerilog code and want to re-simulate.

9.6 Synthesis and uploading to the FPGA

To synthesize your design click "Generate Bitstream" in the flow navigator. This will probably ask you to run synthesis and implementation as well and you should select "Yes." In Vivado synthesis has a very specific meaning within the overall chip creation steps. When I say "synthesis" I mean generally turning the code into hardware, but for Vivado "synthesis" means specifically turning the SystemVerilog code into a list of logic gates. Then Vivado must perform many more steps to actually turn the list of logic gates into hardware (the logic gates need to be placed into certain logic cells and the wires between them routed correctly, among other steps). Finally the bitstream must be generated which can be uploaded to the FPGA to actually set up the design in hardware.

Generating the bitstream (and running all the previous steps) should take about 1-2 minutes and you can track progress in the console area under "Design Runs" or in the top right corner of Vivado. Opening the "Project Summary" tab will also show various reports about how the design was synthesized.

Once the bitstream is generated, you can upload to the FPGA by opening the hardware manager. Make sure that the FPGA is turned on and connected by USB to your computer (you should plug the cable into the "Prog/UART" microUSB slot on the board). Also make sure that jumper JP1 (component 9 on the diagram from earlier) on the FPGA board is in JTAG mode[4]:

[4]boards we hand out will likely already be in JTAG mode, but brand new boards generally come in SPI Flash mode

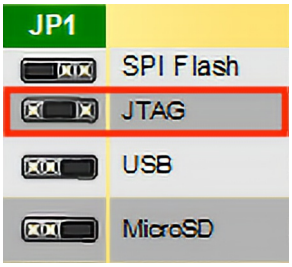


Figure 24: JP1 must be in JTAG mode to program the Nexys A7

Now in Vivado click "Open Hardware Manager" and select "Open Target" and then "Auto Connect." Then you can click "Program Device" and the design should be viewable on the board.

9.7 Synthesis with TCL scripts

In the TCL directory we also provide scripts for synthesizing and uploading the bitstream at the command line. The synthesis script can also be edited to generate timing, power, and utilization reports (by default it only generates a utilization report). These scripts are called `synth.tcl` and `upload.tcl`. The `synth.tcl` script will generate an additional directory called `synth_output` which will contain the generated bitstream and any reports or design checkpoints that were created. This directory should be ignored in version control. These tcl scripts can be run with the `tcl.sh` script

```
./tcl.sh synth
```

After running this command you should see `mux.bit` and `post_route_util.rpt` in the `synth_output` folder.

To upload to the board, you can then run

```
./tcl.sh upload
```

Note that the `upload.tcl` script requires that the bitstream be located in `synth_output/` so make sure that you ran synthesis using `synth.tcl` and not in the Vivado IDE.

10 Sources and further reading

1. *FPGA Prototyping by SystemVerilog Examples* by Pong Chu.

This guide is most inspired by this excellent textbook by Pong Chu. Chu expands on many of the concepts introduced in this guide by providing larger design projects. In the second part of his book, Chu goes beyond the concepts learned here to design a full SoC (System on Chip) system for the Nexys A7 that uses the Xilinx MicroBlaze soft-core processor to control many of the peripheral devices. Chu provides the SystemVerilog components and C++ drivers to control them for many subsystems (video, audio, UART, temperature sensor, ...), along with great ideas for further projects. Many of the examples in this guide are adopted from this source, especially for the FSM and memory sections.

2. *RTL Modeling with SystemVerilog for Simulation and Synthesis* by Stuart Sutherland.

Stuart Sutherland provides a very in-depth view of SystemVerilog's features for RTL synthesis. He goes into much more detail than this guide in features such as synthesizeable functions, user-defined types such as structs, packing and unpacking and more. SystemVerilog is a very large language with many features and quircks and Sutherland delves into great detail about the language. His book serves excellently as a SystemVerilog reference. Many of the style guidelines are drawn from this source along with many of the SystemVerilog details.

3. *Logic Design and Verification Using SystemVerilog* by Donald Thomas.

This book provides a nice overview of SystemVerilog and then dives into more of the testbench and simulation features of the language. If you are interested in more of SystemVerilog's testbench and verification features then this book is worth looking into. I would also recommend *SystemVerilog for Verification* by Chris Spear, though I haven't read it myself.

4. *Digital Design and Computer Architecture* by David Harris and Sarah Harris

Unlike the Harris&Harris book, this guide doesn't touch much on the theory of digital design, mostly sticking to how to apply digital design principles to real hardware using SystemVerilog. DDCA provides a great introduction to digital design, delving into topics like arithmetic circuits, sequential timing, boolean algebra and more. In the second half it also provides a nice overview of computer architecture, outlining how to implement a simple MIPS processor. SystemVerilog examples are used along the way, though the HDL is not the focus of the book.

5. *CMOS VLSI Design* by Neil Weste and David Harris

This book really goes into the details of digital design and how integrated circuits are actually fabricated. In the first part It discusses transistor details such as non-idealities as well as how they are fabricated (layout and lithography). More accurate models for measuring delay, power, and area are also discussed. In the later chapters more advanced arithmetic, sequential, memory, and special-purpose circuits are examined. Some of the FPGA information in this guide is drawn from this source.

6. The IEEE SystemVerilog specification.

The SystemVerilog specification provides all the detail you might want about SystemVerilog though it can be difficult to read. The book above by Stuart Sutherland should provide enough information for most use-cases and is a much easier read with better explanations.