Sequential Logic in Minispec

Sequential logic is digital logic with *state*, i.e., memory. Unlike combinational circuits, where outputs depend only on the current inputs, in sequential circuits outputs are a function of both the current inputs and the state.

The most common type of sequential circuits are *single-clock synchronous sequential circuits*. In these circuits, state is maintained in *registers* that all share the same periodic *clock signal*. Registers update their contents simultaneously, at the rising edge of the clock. This allows discretizing time into cycles and abstracting sequential circuits as *finite state machines (FSMs)*.

In lecture, we have seen how to describe the behavior of FSMs using truth tables or state-transition diagrams, and how to implement them as sequential circuits with registers (flip-flops) and logic gates.

In this tutorial, you will learn how to design sequential circuits in Minispec, which are implemented as *modules*. We will cover the following topics:

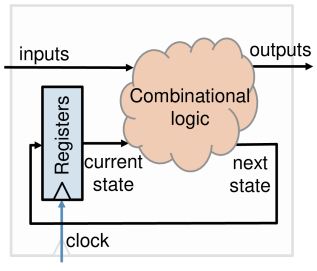
This tutorial is structured to get you started quickly: Section 2 presents a sequence of simple examples that illustrate the main points of describing, composing, synthesizing, and simulating sequential circuits. The later sections expand on the finer points of each topic, presenting the syntax and semantics of modules in detail.

This tutorial assumes you have completed the combinational logic tutorial, as the syntax builds on that of combinational circuits.

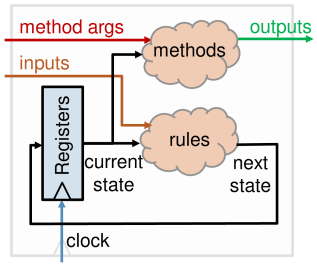
This tutorial uses interactive examples. You can execute each code snippet (cell) by pressing *Shift+Enter*. The executable examples consist of modules and one of two special commands, called *magics*: `%%sim` to simulate a module and `%%synth` to synthesize it into a circuit. (These magics are not part of the Minispec language, but separate commands integrated in Jupyter; see Section 13 of the Minispec reference (https://6004.mit.edu/web/_static/fall19/resources/references/minispec_reference.pdf) for more details).

1. Modules implement FSMs

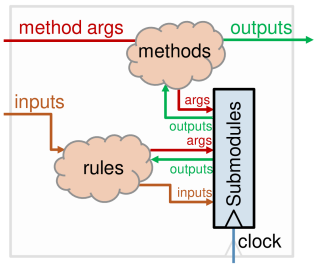
Minispec *modules* implement FSMs. The figure below (left) shows the canonical representation of an FSM. On each cycle, the FSM stores a particular *state* (in registers) and takes some *inputs*. Combinational logic within the FSM uses the state and inputs to compute the *next state* of the FSM and its *outputs* for the current cycle. At the end of the cycle (i.e., in the next rising clock edge), registers update their values, placing the FSM into the next state.



Finite-State Machine



Basic module



General module

Minispec modules have four key elements that correspond to the components of FSMs:

1. **Submodules**, which can be registers or other user-defined modules to allow composition of modules.
2. **Methods**, which implement combinational logic to produce outputs given some input arguments and the current state.

3. **Rules**, which implement combinational logic to produce the next state given some external inputs and the current state.
4. **Inputs**, which represent external inputs controlled by the enclosing module.

The figures above show two modules: a basic module with registers but no other submodules (middle figure), and a general module that includes submodules other than registers (right figure). Comparing the left and middle figures, we see that the key difference between modules and FSMs is the distinction between *inputs* and *method arguments*: whereas combinational logic in FSMs can use any input to produce both the next state and the output, module methods use arguments to produce outputs, separate from the inputs used by rules. As we will see later, this distinction makes modules easy to compose: whereas building an FSM by composing smaller FSMs can cause combinational cycles, a module can be composed from other submodules without combinational cycles, regardless of the implementation of the submodules.

2. Introduction to modules by example

To introduce the main syntax elements of modules, consider the example below, which implements a two-bit counter.

In [ ]:
```
1 module TwoBitCounter;
2     Reg#(Bit#(2)) count(0);
3     method Bit#(2) getCount = count;
4     input Bool enable;
5     rule increment;
6         if (enable)
7             count <= count + 1;
8     endrule
9 endmodule
```

There is a lot of new syntax packed in this example. Let's go line by line:

- *Line 1* starts the definition of a new module *type* called `TwoBitCounter`. This module can then be *instantiated* as a submodule in other modules or used as the top-level module (i.e., synthesized on its own). Like type names, module names must be uppercase.
- *Line 2* instantiates a submodule, a 2-bit register named `count`. This submodule is a 2-bit register because its type is `Reg#(Bit#(2))`. In general, `Reg#(T)` denotes a register of values of type `T`. Also, `Reg#(T)` takes its initial value as an *argument*, `0` in this case. Arguments are specified in parentheses after the submodule name.
- *Line 3* declares a method named `getCount` that returns the current value of the `count` register. In general, writing the name of a register (e.g., `count`) yields its current value. Also, in general methods can have arguments, though in this case `getCount` doesn't have any.
- *Line 4* declares a `Bool` input called `enable`.
- Finally, *lines 5-8* declare a rule called `increment`. Rules specify the combinational logic in charge of updating state, i.e., implementing side-effects. Each rule *automatically executes (i.e., fires) every cycle*. The `increment` rule increments `count` if the `enable` input is `True`. To write to the `count` register, the rule uses a *register assignment operator* `<=`. Register assignments have some crucial differences from normal variable assignments (using `=`), mainly in that *the assigned value is not immediately visible*, as registers are updated at the end of the cycle. We will see all the differences in detail later.

Note how the `TwoBitCounter` module is a particular example of the *basic module* figure in Section 1 above: it uses a register but no other submodules. Before continuing, can you manually synthesize the entire circuit from the code above? Try implementing the rule and method blobs of combinational logic.

Synthesizing modules: You can automatically synthesize the `TwoBitCounter` module into a synchronous sequential circuit:

In [ ]:
```
1 %%synth TwoBitCounter -v -l extended
```

This synthesized module has all the elements you'd expect in a 2-bit counter FSM:

- The two bits of state (`count`) are held in two D flip-flops. Both flip-flops are driven by the same clock signal (`CLK`).
- The `getCount` method translates to the two-bit output `getCount`, which is simply the 2-bit value stored in both flip-flops.
- There is a single-bit `enable` input the corresponds to our `Bool` input `enable`.
- Finally, there is some combinational logic that computes the next value of the output: if the reset signal `RST_N` input is 0, both flip-flops take `0` as their next value (implementing the initial value `0` of `count`); and if `RST_N` is `1` (meaning the circuit is not being reset to the initial state), then the flip-flops take either their current value (if `enable` is `False`) or the incremented value of `count` (if `enable` is `True`).

Note how the FSM above has two inputs that are **implicit** in the Minispec code: the clock (`CLK`) and reset (`RST_N`) signals. The Minispec tools automatically connect all registers to `CLK` and generate initialization logic for all registers using `RST_N`.

Composing modules: Now that we have a two-bit counter module, we can use it in other modules. For example, the `FourBitCounter` module below implements a four-bit counter:

In [ ]:
```
module FourBitCounter;
    TwoBitCounter lower;
    TwoBitCounter upper;

    method Bit#(4) getCount;
        return {upper.getCount, lower.getCount};
    endmethod

    input Bool enable;

    rule increment;
        lower.enable = enable;
        upper.enable = enable && (lower.getCount == 3);
    endrule
endmodule
```

In this example, `FourBitCounter` consists of two `TwoBitCounter` submodules, but does not directly instantiate any register submodules (it is an example of the *general module* figure in Section 1 above). Instead, `FourBitCounter`'s `increment` rule sets the `enable` inputs of its `lower` and `upper` submodules: it increments lower every time `enable` is true, and increments `upper` when `enable` is true and `lower` has reached its maximum value (i.e., it's about to wrap around to zero). With this logic, the concatenation of the upper and lower counts is a 4-bit counter that grows by 1 on every cycle in which `enable` is `True`.

To set the input of a submodule, the `increment` rule simply assigns it a value (e.g., `lower.enable = enable;` sets the `enable` input of the `lower` submodule).

In general, rules are in charge of both updating registers and setting input values. **Only rules can have these side-effects**; methods, like functions, have no side effects, and they may only read registers or call other methods from submodules.

Finally, when modules are composed hierarchically like in this case, the execution semantics are very simple: the system behaves as if, on each cycle, **rules fire sequentially, outside-in**. Specifically, in this case, `FourBitCounter`'s `increment` rule fires, setting the inputs for its submodules. Then, the `increment` rules for the `lower` and `upper` submodules fire. Though more complex types of composition are possible, in 6.004 we will always use hierarchical composition.

Simulating and debugging modules: Last but not least, modules can be simulated over multiple cycles to analyze and test their behavior. You can simulate a module with the `%%sim` magic, like so:

```
%%sim FourBitCounter
```

However, if you tried to run this, you would get no useful output, for two reasons. First, `FourBitCounter` has an input that we are not setting to anything. Second, simulating a module drives its internal state, but does not print any outputs.

To solve both problems, we can write a **testbench module** that (1) instantiates the module we want to test as a submodule, (2) feeds it a sequence of inputs, (3) prints outputs or quantities of interest, and (4) terminates the simulation when all tests are done.

The example below shows a test for the four-bit counter. This example uses two *system functions*, `$display` to print output and `$finish` to terminate the simulation. Section 7 describes these system functions in detail.

In [ ]:
```
module FourBitCounterTest;
    FourBitCounter counter;
    Reg#(Bit#(6)) cycle(0);

    rule test;
        // Increment only on odd cycles
        counter.enable = (cycle[0] == 1);

        // Print the current count
        $display("[cycle %d] counter.getCount = %d", cycle, counter.getCount);

        // Test that the count is what we expect
        if (counter.getCount != cycle[4:1]) begin
            $display("FAILED: Wrong count");
            $finish;
        end

        // Increment the cycle counter, and terminate after a full loop
        cycle <= cycle + 1;
        if (cycle >= 32) begin
            $display("PASSED");
            $finish;
        end
    endrule
endmodule
```

We can then simulate the testbench module:

In [ ]:
```
%%sim FourBitCounterTest
```

These examples cover the main aspects of modules. The remainder of the tutorial goes in more depth into each of these aspects and presents additional examples.

3. Module syntax and elements

Definition: Modules are defined using the following syntax:

```
module ModType [#(param1, paramK)][(Type1 arg1, ..., TypeN argN)];
  <submodule, method, input, rule, constant decls>
endmodule
```

Each definition specifies a new module type, `ModType`, which can then be instantiated as a submodule in other modules or as the top-level module.

Modules meant to be submodules can have optional *module arguments*, `arg_i`, with types `Type_i`. These arguments can be constant values (e.g., used to set initial values), or other modules.

Modules can be parametric. Parametric module definitions use exactly the same syntax as parametric functions, with parameters specified after the module name.

The body of the module can declare submodules, methods, inputs, rules, and constants, whose syntax is explained below.

Example: n-bit Counter with initial value. The example below generalizes the previous counter modules to show module parameters and arguments in action. The `Counter#(n)` module below is built by composing smaller counters; the base case `Counter#(1)` stops the recursion. `Counter#(n)` takes a single `Bit#(n)` argument, the counter's initial value.

In [ ]:

```
// Base case
module Counter#(1)(Bit#(1) initialValue);
    Reg#(Bit#(1)) count(initialValue);
    method Bit#(1) getCount = count;
    input Bool enable;
    rule increment;
        if (enable) count <= ~count;  // that's one-bit +1
    endrule
endmodule

module Counter#(Integer n)(Bit#(n) initialValue);
    Counter#(n/2) lower(initialValue[n/2-1:0]);
    Counter#(n-n/2) upper(initialValue[n-1:n/2]);
    method Bit#(n) getCount = {upper.getCount, lower.getCount};
    input Bool enable;
    rule increment;
        lower.enable = enable;
        upper.enable = enable && (lower.getCount == -1); // -1 == all ones, about to wrap around
    endrule
endmodule

// Simple test for a 7-bit counter with a non-zero initial value
module CounterTest;
    Counter#(7) counter(29);
    Reg#(Bit#(6)) cycle(0);
    rule test;
        counter.enable = (cycle[0] == 1);
        $display("[cycle %d] counter.getCount = %d", cycle, counter.getCount);
        cycle <= cycle + 1;
        if (cycle >= 8) $finish;
    endrule
endmodule

%%sim CounterTest
```

Submodule declarations

Submodule declarations use the syntax:

```
SubmodType submodName [(arg1, ..., argN)];
```

where `SubmodType` is the type of the submodule being instantiated, `submodName` is the name of this specific submodule instance, and `arg_i` are the (optional) arguments to the submodule.

Methods

Methods are nearly identical to functions: they specify combinational logic that produces an output and have no side effects. They can call methods in submodules or read register values, but they cannot set the inputs of submodules or write to any register (these are side effects, which only rules can have).

Methods use a syntax nearly identical to functions:

```
method RetType mname(Type1 arg1, ..., TypeN argN);
    stmt1
    ...
    stmtN
endmethod
```

where `mname` is the method's name; `RetType` is the type of its return value; `arg_i` are the names of its arguments, with types `Type_i`; and `stmt_i` are statements. Methods also support the same shorthand syntax as functions:

```
method RetType mname(Type1 arg1, ..., TypeN argN) = expr;
```

Methods may have no arguments. Unlike functions, methods *cannot be parametric* (i.e., they cannot define their own parameters, though they can use the parameters of its module, as shown in the `Counter#(n)` example above).

Use: A method may be called only from the methods or rules of its enclosing module. The syntax for a method call is `submoduleName.methodName(arg1, ..., argN)`. A module cannot call its own methods.

Example: Methods with arguments. The examples we have seen so far do not use methods with arguments. The example below shows an alternative `Counter#(n)` implementation with a method, `countIs`, that returns whether the current count matches a particular value, passed as an argument to the method. Note how the synthesized circuit has inputs for both the enable and the `countIs` argument, `countIs_x`.

In [ ]:
```
module Counter#(Integer n);
    Reg#(Bit#(n)) count(0);
    method Bit#(n) getCount = count;
    method Bool countIs(Bit#(n) x) = count == x;
    input Bool enable;
    rule increment;
        if (enable) count <= count + 1;
    endrule
endmodule

// synth options chosen to reduce number of gates
%%synth Counter#(2) -v -l extended -d 1000
```

Inputs

Inputs specify external inputs that are controlled by the rule(s) of an enclosing module. Their syntax is:

```
input Type name [default = defaultExpr];
```

where `Type` is the input's name, `name` is its name, and the optional `defaultExpr` specifies a default value for the input.

Use: Inputs can be read within the module just like variables, but cannot be set.

Inputs can be set within a rule of the enclosing module, with syntax submoduleName.inputName = expr;, like a normal assignment. If the input does not have a default value, the enclosing module must set the input every cycle. If the input has a default value, then setting the input is optional.

An input can only be set *once*. Trying to assign to the input multiple times within a cycle will cause a compiler error.

Example: Inputs with default values. The example below implements an n-bit *delta counter*, which can be incremented by a variable amount every cycle. The `delta` input specifies this increment. By default, `delta` is `0`. This way, modules instantiating `DeltaCounter` need not set `delta` every cycle. (See what happens with `DeltaCounterTest` when you remove the default value of `delta`).

In [ ]:
```
module DeltaCounter#(Integer n);
    Reg#(Bit#(n)) count(0);
    method Bit#(n) getCount = count;

    input Bit#(n) delta default = 0;

    rule tick;
        count <= count + delta;
    endrule
endmodule

module DeltaCounterTest;
    DeltaCounter#(8) counter;
    Reg#(Bit#(8)) cycle(0);
    rule test;
        if (cycle[0] == 1)
            counter.delta = cycle;
        $display("[cycle %d] counter.getCount = %d", cycle, counter.getCount);
        cycle <= cycle + 1;
        if (cycle >= 8) $finish;
    endrule
endmodule

%%sim DeltaCounterTest
```

Rules

Rules specify combinational logic that updates the state of the module, i.e., they implement side-effects. Specifically, rules set the values to be written in registers at the end of the cycle and the inputs of submodules. Rules use the following syntax:

```
rule ruleName;
    stmt1
    ...
    stmtN
endrule
```

Rules *fire (i.e., automatically execute) every cycle*. A module may have multiple rules, but these rules cannot have overlapping side-effects (i.e., they must update disjoint registers and inputs). Any such overlap will cause a compiler error.

4. Registers

Registers are the most basic module. `Reg#(T)` stores a value of type `T`. `T` can be any type that can be represented as bits.

Declaration: Modules can declare registers with the usual syntax for submodules. `Reg#(T)` takes an initial value, so its declaration is `Reg#(T) regName(initialValue);`.

Initial values allow registers to start set to known values. This requires some additional *reset* circuitry, as we saw in the `FourBitCounter` example. If it's not necessary to have an initial value, this circuitry can be avoided by using `RegU#(T)`, a variant of Reg#(T) that starts on an unknown value. `RegU#(T)` declarations do not take an initial value: `RegU#(T) regName;`.

Reads: Registers can be read from anywhere in the module. Simply using the name of the register yields its value.

Writes: Registers can be written from rules using the following syntax:

```
regName <= expr;
```

where `regName` is the register's name and `expr` is the value to be be written to it. Note how register writes are not normal assignments: they use `<=` instead of `=` and have different semantics in two key aspects:

1. *Register writes do not take place until the end of the cycle*. Reading a register value in the same rule and after a register write statement will yield the value of the register in the current cycle, not the value set by the register write.
2. *Registers can be written only once*. In each cycle, a register may be written at most once. A rule that writes the same register multiple times will cause a compiler error. Two rules that may write to the same register will cause a compiler error. Registers need not be written every cycle; if not written, a register retains its previous value.

Example: Accumulator. The example below shows a multi-cycle accumulator that illustrates the use of `Reg` and `RegU` and register assignment semantics. `Accumulator` takes a stream of input values, one per cycle, and stores their running sum. Together with each value, the accumulator takes a command (`cmd`), which can be either `Add` or `Set`. `Add` causes the current value to be added to the running sum, while `Set` discards the running sum and sets the accumulator to the given value.

To avoid the reset logic for the `runningSum` register, the accumulator uses a `RegU` to store it, plus a `Bool` register `valid` initialized to `False`. On the first cycle, `valid` transitions from `False` to `True` and `runningSum` always captures the first value given, discarding its (garbage) initial value. Finally, note how *register writes do not take place until the end of the cycle*: in cycle 0, the `tick` rule writes `valid <= True;`, but the expression `(!valid || in.cmd == Set)` in the next line *still sees valid is set to False*.

In [ ]:

```
typedef enum { Add, Set } AccumulatorCmd;
typedef struct {
    AccumulatorCmd cmd;
    Bit#(16) value;
} AccumulatorInput;

module Accumulator;
    Reg#(Bool) valid(False);
    RegU#(Bit#(16)) runningSum;
    input AccumulatorInput in default = AccumulatorInput { cmd: Add, value : 0 };
    method Bool isValid = valid;
    method Bit#(16) sum = runningSum;
    rule tick;
        valid <= True;
        runningSum <= (!valid || in.cmd == Set)? in.value : runningSum + in.value;
    endrule
endmodule

module AccumulatorTest;
    Accumulator acc;
    Reg#(Bit#(16)) cycle(0);
    rule test;
        if (cycle[0] == 0)
            acc.in = AccumulatorInput { cmd: Add, value : cycle + 10 };
        else if (cycle[2:1] == 2'b11)
            acc.in = AccumulatorInput { cmd: Set, value : cycle };
        $display("[cycle %d] acc.sum = %d acc.isValid = %d", cycle, acc.sum, acc.isValid);
        cycle <= cycle + 1;
        if (cycle >= 16) $finish;
    endrule
endmodule

%%sim AccumulatorTest
```

5. Semantics of hierarchically nested modules

Assume we impose two conditions on a design. First, modules follow a strict hierarchy, i.e., each module interacts only with the submodules it instantiates. Second, no method reads module inputs.

Under these conditions, Minispec guarantees that there are no combinational cycles and gives very simple semantics: the system behaves as if, on each cycle, **rules fire sequentially, outside-in**: first, the rule in the top-level module fires, then the rules in all its submodules, and so on.

Because each module's rule calls methods in its submodules and sets the inputs to its submodules, this order guarantees that all inputs are set by the time a rule executes. Moreover, the effects of rules in submodules cannot be observed by their enclosing modules: *data flows inside-out only through methods, and data flows outside-in through inputs and rules*.

In 6.004, we will **only** use and require you to understand module composition under the above conditions, i.e., Minispec's simple module semantics. For advanced designs, it may be helpful to break these conditions; if you're curious, Section 9.4 of the Minispec reference (https://6004.mit.edu/web/_static/fall19/resources/references/minispec_reference.pdf) explains why this may be desirable and gives Minispec's general semantics of modules, but **you do not need to know them**.

Why use modules instead of composing FSMs directly? In Section 1, we said that modules add structure over FSMs by splitting module arguments and inputs, and that this *makes hierarchical composition easy*. Specifically, the separation between rule inputs and method arguments allows modules to instantiate and use arbitrary submodules **while avoiding combinational cycles**, i.e., ensuring that combinational logic remains acyclic.

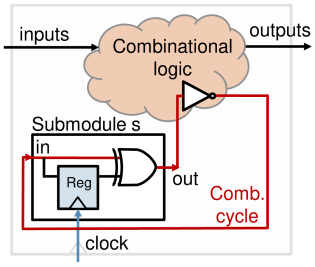
It is useful to understand why this is the case, because this is the key conceptual difference between Minispec and other hardware description languages (HDLs). To see why rules and methods enable composition while avoiding combinational cycles, consider a different approach where we composed different FSMs directly, with the combinational logic in a module setting the inputs and using the outputs of its submodules (Verilog and other HDLs follow this approach).

Unfortunately, this can cause combinational loops like the one shown in the figure to the right: the outer module sets `s.in = !s.out;` and submodule `s` has a combinational path from `in` to `out`, causing a combinational feedback loop. We cannot prevent loops by disallowing modules from setting submodule inputs based on submodule outputs, because this is often

necessary. For example, a module may need to check whether a submodule is ready to start processing a new value (e.g., through a ready output) before giving it the value through an input.

Thus, composing FSMs this way requires *the specific combination of a module and its submodules to yield acyclic combinational logic*. But this condition is brittle, requiring discipline from the designer, and it is implementation-dependent: changing the implementation of a submodule may introduce a combinational cycle in a previously correct circuit. *Therefore, this is a poor abstraction.*

Methods avoid these problems: with methods, *input-to-output combinational paths in a module happen only between method arguments and method outputs*. Moreover, method calls *force the arguments to be available before the output is available*. Thus, a module cannot perform a sequence of method calls to its submodules that results in a cycle. As a result, modules can safely call methods from submodules without knowing their implementation details; *only their interface matters.*



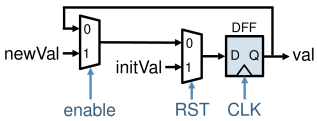
Example showing how composing FSMs by wiring their inputs and outputs can introduce a combinational cycle.

6. Synthesis of modules into sequential circuits

Sequential synthesis is a straightforward extension of combinational synthesis.

As we have seen before, sequential circuits have two **implicit inputs** that are not present in Minispec code: the clock signal (CLK) and the reset signal (RST or RST_N).

Registers are synthesized as collections of 1-bit D flip-flops (DFFs). All registers use CLK as their clock. Registers with an initial value (i.e., `Reg#(T)`) include reset circuitry that sets its value to the initial value when the circuit powers up. Because flip-flops hold an arbitrary value when first powered, this is accomplished by the RST signal: the RST signal is 1 for a few cycles after power-up, letting registers write their initial values with the reset circuitry shown in the figure to the right. Registers with no initial value (i.e., `RegU#(T)`) have no reset circuitry. RST_N is simply an inverted (negated) RST signal: it is 0 for a few cycles after power-up. Depending on the standard cell library, RST or RST_N will be used.



Example synthesized register with reset and enable circuits.

Registers are always written to by a single rule, but may not be updated every cycle. When not updated every cycle, the register includes *write-enable* circuitry to optionally retain its old value, as shown in the figure above.

Inputs without a default value are simply wires. Inputs with a default value translate to a multiplexer that chooses between the input value, if any is set, and the default value, if none is set.

Rules are synthesized as normal combinational logic. They produce the values for all registers and inputs they set. When rules conditionally set registers or inputs, they also generate the corresponding enable signals so that, when not set, registers retain their old value and inputs use their default value.

Methods are synthesized like functions. A method can be called multiple times. If the method has no arguments (i.e., it always returns the same value on a given cycle), all callers share the same output value. If the method has arguments, each call instantiates a new copy of the method.

7. System functions for testing and debugging

As we have seen before, testbench modules use special functions, called *system functions*, to control simulation and print data. System functions are not synthesizable to hardware, and are only used when simulating a module. All system functions begin with a dollar sign (`$`). System functions *may only be used within module rules*, as they have side effects. Calls to a system function from a function or method will cause a compiler error.

The two main system functions are `$finish` and `$display`.

`$finish` terminates the simulation. It takes no arguments.

`$display` prints strings to standard output. It takes a variable number of arguments, which can be strings or other values.

As shown in the example below, every value that is not a string will be interpreted as an n-bit value and printed as a decimal number by default. To print numbers in other bases, `$display` can use the same syntax as the printf function in C or the print function in Python, using format strings with `%b` for binary, `%d` for decimal, and `%h` or `%x` for hexadecimal values.

Nevertheless, displaying complex types like structs as one long number is inconvenient. The `fshow` function automatically formats complex types, as shown in the example below. `fshow` can be used on values of any type.

In [ ]:
```
typedef struct { Bit#(8) red; Bit#(8) green; Bit#(8) blue; } Pixel;

module DisplayExample;
    rule test;
        $display("Hello world!");         // Prints "Hello world!"
        $display("Hello", " ", "world!"); // Prints "Hello world!"

        // Printing non-string values
        Bit#(8) x = 42;
        $display("x in decimal is ", x);  // Prints "x in decimal is 42"

        // Using printf-style formatting
        $display("0b%b == %d == 0h%h", x, x, x); // Prints "0b00101010 == 42 == 0x2a"

        // Using fshow
        Pixel cyan = Pixel{ red : 0, green : 255, blue : 255 };
        $display(cyan);                  // Prints " 65535"
        $display(fshow(cyan));   // Prints "Pixel { red: 'h00, green: 'hff, blue: 'hff }"

        $finish;
    endrule
endmodule

%%sim DisplayExample
```

8. Building good module interfaces

So far, we have seen how to write sequential circuits with a given set of methods and inputs, i.e., a pre-specified interface. However, there are many ways to design a module's interface, i.e., how it interacts with the outside world. A good interface will make the module simple to implement and easy to use, whereas a bad interface can make the module brittle and introduce frequent errors.

Group related inputs and outputs

In this section, we will see some design strategies and introduce a new type that will keep your modules simple and make them easy to use correctly by other modules. All we will see is several manifestations of the following basic principle: **minimize the number of inputs and methods** by **grouping related inputs and outputs**.

In other words, any time your module may need to take several inputs together, or produce multiple related outputs that the module's user should access together, those different pieces of data should be grouped into **a single input or method**, instead of transferred using several inputs and methods. This will often require *using composite types like structs* for inputs and method outputs.

We have already seen this principle at work! In the accumulator example from Section 4, the Accumulator module used a single input to convey both an input and a command:

```
typedef enum { Add, Set } AccumulatorCmd;
typedef struct {
    AccumulatorCmd cmd;
    Bit#(16) value;
} AccumulatorInput;

module Accumulator;
    Reg#(Bool) valid(False);
    RegU#(Bit#(16)) runningSum;

    input AccumulatorInput in default = AccumulatorInput { cmd: Add, value : 0 };

    method Bool isValid = valid;
    method Bit#(16) sum = runningSum;
    ...
```

```
    ...
```

This approach requires declaring a custom `AccumulatorStruct`. But the benefits are well worth it! Consider the alternative approach where we define two inputs instead:

```
    input Cmd inCmd default = Add;
    input Bit#(16) inValue default = 0;
```

This alternative approach would make `Accumulator` **harder to use**: the module user could forget to set one input, or set inputs on different cycles, or misread the defaults, introducing errors. In this case, we want to do something specific with the value we're passing, so the command and the value should be grouped together.

Though `Accumulator` is using the principle of grouping related inputs and methods, it is not using it enough: it has two methods for two pieces of related information, the running sum and whether the sum is valid. This is error-prone, because the module's user could use the output from the `sum()` method without checking the output of `isValid()`, . Instead, we should have a single method that returns **both** whether the sum is valid, and, if so, its value.

Because this is a very common need, Minispec already includes a built-in type that represents a valid or invalid value: `Maybe#(T)`. Let's see how it works.

The Maybe#(T) type

`Maybe#(T)` represents an *optional value* of type `T`. A `Maybe#(T)` can be either `Valid` if it holds a value of type `T`, or `Invalid` if it does not hold a value. `Maybe#(T)` is especially useful for modules, which often do not have valid inputs or outputs every cycle.

Creating Maybe#(T) values: Given a value `v` of type `T`, `Valid(v)` is a valid `Maybe#(T)` that holds `v`. The literal `Invalid` can be assigned to any `Maybe#(T)` variable to make it invalid.

Checking for validity: The built-in function `isValid` returns `True` if its argument is `Valid`, and `False` if it is `Invalid`.

Unpacking Maybe#(T)'s optional value: The built-in function `fromMaybe` allows extracting the value of a valid Maybe value. Its signature is `T fromMaybe(T defaultValue, Maybe#(T) x)`. If `x` is `Valid`, `fromMaybe` returns `x`'s value; if `x` is `Invalid`, `fromMaybe` returns `defaultValue`.

Example: Accumulator using Maybe#(T). The example below rewrites the `Accumulator` module to use Maybe types:

In [ ]:

```
typedef enum { Add, Set } AccumulatorCmd;
typedef struct {
    AccumulatorCmd cmd;
    Bit#(16) value;
} AccumulatorInput;

module AccumulatorMaybe;
    Reg#(Maybe#(Bit#(16))) runningSum(Invalid);
    input AccumulatorInput in default = AccumulatorInput { cmd: Add, value : 0 };
    method Maybe#(Bit#(16)) sum = runningSum;
    rule tick;
        runningSum <= (!isValid(runningSum) || in.cmd == Set)?
            Valid(in.value) :
            Valid(fromMaybe(?, runningSum) + in.value);
    endrule
endmodule

module AccumulatorMaybeTest;
    AccumulatorMaybe acc;
    Reg#(Bit#(16)) cycle(0);
    rule test;
        if (cycle[0] == 0)
            acc.in = AccumulatorInput { cmd: Add, value : cycle + 10 };
        else if (cycle[2:1] == 2'b11)
            acc.in = AccumulatorInput { cmd: Set, value : cycle };
        $display("[cycle %d] acc.sum = ", cycle, fshow(acc.sum));
        cycle <= cycle + 1;
        if (cycle >= 16) $finish;
    endrule
endmodule

%%sim AccumulatorMaybeTest
```

Note how the module now has a single method that returns a valid or invalid output. This makes it nearly impossible for the module user to use an invalid output: to get to the value returned by `sum`, the user must first unpack it using `fromMaybe`.

Multi-cycle computations

It is common for sequential circuits to implement computations that take multiple cycles: the circuit takes an input at a given cycle, then spends multiple cycles, potentially a variable number of them, producing the output, which it then makes available through an output.

Maybe types are a helpful way to implement the inputs and outputs for these modules in a robust way. Let's see this though an example.

Example: GCD. We want to design a circuit to find the greatest common divisor of two numbers, a and b, using Euclid's algorithm. The code below shows a Python implementation of Euclid's algorithm:

```
def gcd(a, b):
    x = a
    y = b
    while x != 0:
        if x >= y:
            x = x - y  # subtract
        else:
            (x, y) = (y, x)  # swap
    # when x is 0, y has the gcd
    return y
```

The module below implements a sequential circuit that performs one iteration of the `while` loop above each cycle, either subtracting or swapping `x` and `y`. Thus, this circuit takes a variable number of cycles to produce an output (note how there's no efficient way to do this with combinational logic).

In [ ]:
```
typedef struct {Bit#(16) a; Bit#(16) b;} GCDArgs;

module GCD;
    Reg#(Bit#(16)) x(1);
    Reg#(Bit#(16)) y(0);

    input Maybe#(GCDArgs) in default = Invalid;

    rule gcd;
        if (isValid(in)) begin
            let args = fromMaybe(?, in);
            x <= args.a;
            y <= args.b;
        end else if (x != 0) begin
            if (x >= y) begin
                x <= x - y;  // subtract
            end else begin
                // swap (note assignments don't take effect till next cycle)
                x <= y;
                y <= x;
            end
        end
    endrule

    method Maybe#(Bit#(16)) result =
        (x == 0)? Valid(y) : Invalid;
endmodule
```

Note how the `GCD` module uses a single Maybe input and a single Maybe output. If the input is valid, the circuit begins a single computation by loading the arguments a and b into x and y. Otherwise, the circuit performs one step of Euclid's algorithm until x reaches 0, at which point y has the result. The module outputs this result safely by using a Maybe type: the `result` method returns `Valid(y)` only when y has a valid result, and `Invalid` otherwise.

This circuit is straightforward to use: the enclosing module starts a GCD computation by setting a `Valid` input, then waits until the output becomes `Valid`. At that point, the enclosing module can set a new `Valid` input to begin the next computation. For example:

In [ ]:

```
module GCDTest;
    GCD gcd;
    Reg#(Bit#(16)) cycle(0);
    Reg#(Bit#(4)) numResults(0);
    rule test;
        cycle <= cycle + 1;
        if (cycle == 0) begin
            let args = GCDArgs{a: 10, b: 15};
            gcd.in = Valid(args);
            $display("[cycle %d] Initial GCD arguments: %d, %d", cycle, args.a, args.b);
        end else if (isValid(gcd.result)) begin
            let result = fromMaybe(?, gcd.result);
            $display("[cycle %d] GCD result: %d", cycle, result);

            numResults <= numResults + 1;
            if (numResults >= 4) begin
                $display("[cycle %d] Finished", cycle);
                $finish;
            end else begin
                let args = GCDArgs{a: cycle, b: 2 * cycle + 9};
                $display("[cycle %d] Next GCD arguments: %d, %d", cycle, args.a, args.b);
                gcd.in = Valid(args);
            end
        end
    endrule
endmodule

%%sim GCDTest
```

Note how the GCD module takes a variable number of cycles to process each input (from 6 to 45-29=16).

In GCD, the values of x and y are sufficient to determine the state of the circuit (e.g., whether it has a valid input). In other cases, it may be necessary to have additional registers to track the state (e.g., how many steps are there to completion). But regardless of the internal implementation, Maybe inputs and outputs can always be used to build a simple interface for modules performing multi-cycle computations.

Conclusion

We have seen how Minispec makes sequential circuits composable by describing them as modules. Modules can be composed hierarchically without introducing combinational cycles, and their composition yields simple semantics: the system behaves as if rules fire sequentially, outside-in.

Although these semantics make modules easy to understand, it is important to remember that the system is intrinsically parallel, with potentially many computations and register assignments happening concurrently in a single cycle. As with combinational logic, you should never forget that you're describing hardware, not software.

This tutorial omits some of the finer details of the Minispec language; to dig deeper, please check the Minispec reference (https://6004.mit.edu/web/_static/fall19/resources/references/minispec_reference.pdf) for the full semantics.