Combinational Logic in Minispec

Combinational circuits, also known as Boolean circuits, are a basic building block of digital logic. A combinational circuit has digital inputs and outputs; each output is a Boolean function of its current inputs (i.e., the circuit has no memory or state); and each output is guaranteed to reach a stable digital value after a bounded *propagation delay* from the time at which the inputs reach valid digital values.

In lecture, we have seen how to describe the behavior of combinational circuits using Boolean formulas or truth tables, and how to implement these circuits using gates.

In this tutorial, you will learn how to design combinational circuits in Minispec. We will cover the following topics:

These topics build on each other and present increasingly advanced language features. In particular, topics 1 and 2 suffice to build any combinational circuit; every other feature is a convenience.

This tutorial uses interactive examples. You can execute each code snippet (cell) by pressing *Shift+Enter*. The executable examples consist of Boolean functions and one of two special commands, called *magics*: `%%eval` to evaluate a function 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://6191.mit.edu/_static/fall23/resources/references/minispec_reference.pdf for more details).

1. The Bool type and Boolean operations

Values of type `Bool` can take one of two values, `True` or `False` . `Bool` values support the three basic Boolean algebra operations:

- `!` : Boolean NOT
- `&&` : Boolean AND
- `||` : Boolean OR

For example,

```
Bool a = True;
Bool b = False;

Bool x = !a;      // False since a == True
Bool y = a && b; // False since b == False
Bool z = a || b; // True since a == True
```

`Bool` values also support equality operations:

- `==` : Equals
- `!=` : Not equals

For example,

```
Bool a = True;
Bool b = False;

Bool e = a == b; // False
Bool n = a != b; // True
```

Note that, for Boolean values, `!=` is equivalent to Boolean XOR, and `==` is equivalent to Boolean XNOR.

2. Functions

In Minispec, *functions* specify combinational circuits. Each function produces an *output result* that depends only on the values of its *input arguments*.

Definition: Functions can be defined using the following syntax:

```
function RetType fname(Type1 arg1, ..., TypeN argN);
stmt1
...
stmtM
endfunction
```

where `fname` is the function's name; `RetType` is the type of its return value; `arg_i` are the names of its input arguments, with types `Type_i` ; and `stmt_i` are statements.

Functions must return a result by using **return statements**. For example, the function below shows a function containing a single return statement.

In [ ]:
```
function Bool and2(Bool a, Bool b);
   return a && b;
endfunction
```

You can evaluate a function with the `%%eval` magic. For example,

In [ ]:
```
%%eval and2(False, True)
%%eval and2(True, True)
```

More complex functions can use multiple statements. For example, the `majority` function below returns `True` iff at least two out of its three inputs are `True` . This function uses three assignment statements before its return statement.

In [ ]:
```
function Bool majority(Bool a, Bool b, Bool c);
   Bool ab = a && b;
   Bool ac = a && c;
   Bool bc = b && c;
   return ab || ac || bc;
endfunction

%%eval majority(True, False, False)
%%eval majority(False, True, True)
```

Functions can invoke other functions:

In [ ]:
```
function Bool and4(Bool a, Bool b, Bool c, Bool d);
   Bool ab = and2(a, b);
   Bool cd = and2(c, d);
   return and2(ab, cd);
endfunction

%%eval and4(True, True, True, True)
%%eval and4(True, False, True, True)
```

However, a function may not invoke itself, i.e., **recursion is not allowed**.

Because functions are often short, Minispec has a **shorthand syntax** to declare single-statement functions more succinctly. For example, the `and4` implementation below is equivalent to the one above. Instead of a `return` statement followed by `endfunction` , the expression that computes the output value follows the `=` sign.

In [ ]:
```
function Bool and4(Bool a, Bool b, Bool c, Bool d) = a && b && c && d;
```

3. Manual synthesis of functions into combinational circuits

Minispec functions can *always* be synthesized to combinational circuits. This is because functions have two key properties:

1. *Pure*: Functions compute the output based only on the values of its input arguments. They cannot use or alter any state or variables outside the function.
2. *Acyclic*: Functions have no cycles, i.e., they cannot "jump back" to a prior point of execution. They are thus always guaranteed to terminate in a bounded number of steps, and that number is independent of the particular values of the input arguments. This is why recursion is not allowed.

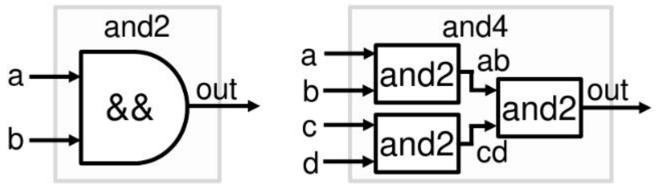
In practice, Minispec functions are synthesized by composing smaller building blocks. This exploits that combinational circuits compose easily: a circuit composed of multiple combinational devices connected with wires is also combinational if each input of the constituent devices is tied to a single output or a constant value, and if the network is acyclic, i.e., it has no directed cycles (which may create feedback loops).

Since Minispec functions are acyclic, one can synthesize a function by synthesizing a each syntax element (expression, statement, etc.) and connecting them. Though the details of synthesis depend on the compiler and tools used, it is useful to roughly understand how syntax elements are translated and composed. With what we have seen so far, we need only three rules:

1. Each complex expression is broken down into a tree of basic operations (e.g., operators or function calls), then each operation is synthesized, and finally their inputs and outputs are wired together in the same tree fashion.
2. All operators (e.g., `&&`, `||`, etc.) can be implemented using combinational circuits. Each operator is synthesized as a separate circuit.
3. Functions are **inlined**: each function call instantiates a separate circuit implementing the function, which is then synthesized (since there's no recursion, inlining functions always terminates).

The last rule is particularly important to keep in mind, because it is completely different from how software procedures are commonly implemented.

Using these rules, we can manually synthesize all the functions we have seen so far. For example, the figure below shows the manual synthesis of the `and2` and `and4` circuits.



We will introduce more language elements later, and explain how they are synthesized then. For a full description, see Section 8.1 of the Minispec reference (https://6191.mit.edu/_static/fall23/resources/references/minispec_reference.pdf).

4. The Bit#(n) type and operations

While `Bool` is technically the only type we need to write combinational circuits, working with many single-bit values gets tedious quickly. For example, if we wanted to code a function that operated on a 32-bit number, we'd need to pass 32 `Bool` arguments! Even worse, since each function can produce only one output result, implementing circuits with several output wires would require a separate function per output wire.

To address this problem, Minispec provides more complex types that encapsulate a collection of bits. The most basic of these is `Bit#(n)`, which encodes an `n`-bit value (`n` must be a non-negative number). Most types (all except `Integer`, which we will see later) are represented in hardware using a fixed number of bits.

The first order of business is how to assign fixed values (e.g., 0, 1, 42, etc.) to `Bit#(n)` variables. This is the role of number literals.

Number literals: Number literals encode numeric values. They can be **sized** or **unsized**. Sized literals encode their bit-width, i.e., the number of bits they take. By contrast, unsized literals have no explicit bit-width. Literals can be specified in decimal, hexadecimal, and binary bases.

A sized number literal always consists of three elements:

1. *Bit-width*, written as a decimal number.
2. *Base*: `'d` for decimal, `'h` for hexadecimal, and `'b` for binary.
3. *Value*, written using digits in the specified base (0–9 for decimal, 0–9 and a–f for hex, and 0 or 1 for binary).

An unsized number literal has no bit-width, and consists of an optional base and value. If the base is not given, the value is interpreted as decimal. To make long values more readable, number literals allow using underscore characters (_) to separate value digits.

The examples below showcase these rules:

```
// Sized number literals
// 4-bit decimal value
10 4'd10 // in decimal
4'b1010 // in binary
4'ha // in hex
4'hA// hex digits are case-insensitive
// 8-bit decimal value 10
8'b00001010
8'b1010
8'h0a

// Unsized number literals
'd10 // decimal 10
10 // also decimal 10
'b1010
'habcdef

// Underscores (_) can separate digits
16'b1010_0110_1101_0010
```

Sized and unsized literals can both be used in expressions involving `Bit#(n)` variables. The examples below illustrate their pros and cons. Sized literals enforce their bit-width, and will cause a compile-time error on a width mismatch. By contrast, unsized literals have their bit-width deduced by the compiler. This makes code more succinct and can help express long values, but the compiler won’t catch mistakes that stem from wrong assumptions about bit-widths.

```
Bit#(4) x = 4'd10; // OK, as both x and the literal are 4 bits
Bit#(5) y = 4'd10; // Error due to mismatched bit-widths, y is 5 bits
```

```
Bit#(4) z = 10;    // OK, 10 inferred to be 4 bits
Bit#(4) w = 1000;  // Error, decimal 1000 inferred to be 4 bits but doesn't fit in 4 bits
```

Bitwise logical operations:
`Bit#(n)` supports bitwise inversion/NOT (`~`), AND (`&`), OR (`|`), XOR (`^`), and XNOR(`~^` or `^~`).
Bitwise logical operations take `Bit#(n)` inputs and produce a `Bit#(n)` output, where the operations apply to each bit of the inputs.
For example:

```
Bit#(4) a = 4'b0011;
Bit#(4) b = 4'b0101;

Bit#(4) x = ~a;   // 4'b1100
Bit#(4) y = a & b; // 4'b0001
Bit#(4) z = a ^ b; // 4'b0110
```

Bit selection: Given a `Bit#(n)` variable `x` `x[i]` selects the i bit of `x` . `x[i]` has type `Bit#(1)`.
Bits are enumerated right-to-left, i.e., starting from the least-significant bit. For example, given `Bit#(4) x = 4'b1010` , then `x[0] = 0` (the least-significant or rightmost bit), `x[1] = 1` , `x[2] = 0` , and `x[3] = 1` (the most-significant or leftmost bit).
This is consistent with how digits are always enumerated in a number (according to significance, so `x` 's value is $\sum_{i=0}^{n-1} 2^i x[i]$), but it is different from how vectors are indexed.

Given a `Bit#(n)` variable `x` , `x[i:j]` , with i ≥ j, selects the range of bits of `x` starting at `x[i]` and ending at `x[j]` , both inclusive.
For example, given `Bit#(4) x = 4'b1010` , `x[2:1]` is the `Bit#(2)` value `2'b01` .

Example 1: Parity circuit. The *parity* of an n-bit value is 0 if the value has an even number of ones, or 1 if the value has an odd number of ones. Parity is a useful way to detect errors: if a single bit flips in a value, its parity changes, so systems often store and transmit the parity of all values and check them to see whether a bit has flipped. Moreover, computing the parity is very simple: it's the XOR of all the bits.

The `parity8` function below computes the parity of an 8-bit argument. `parity8` is built by combining smaller circuits (functions) that compute the parity of 4-bit and 2-bit arguments:

```
In [ ]: function Bit#(1) parity2(Bit#(2) x) = x[1] ^ x[0];
        function Bit#(1) parity4(Bit#(4) x) = parity2(x[3:2]) ^ parity2(x[1:0]);
        function Bit#(1) parity8(Bit#(8) x) = parity4(x[7:4]) ^ parity4(x[3:0]);
        %%eval parity8(8'b0100_0110)
        %%eval parity8(8'b0101_0110)
```

The example above uses *function composition* to build `parity8` out of several simple functions. We recommend you structure your design into collections of simple and reusable functions.

That said, the example above does not use the fact that XOR (`^`) is a bitwise operator. To show this, here is an alternative implementation that does so, by XOR-ing the upper and lower parts of the value until reducing the value to a `Bit#(1)` :

In [ ]:
```
 function Bit#(1) parity8_alt(Bit#(8) x);
    Bit#(4) r4 = x[7:4] ^  x[3:0];
    Bit#(2) r2 = r4[3:2] ^ r4[1:0];
    Bit#(1) r1 =  r2[1] ^   r2[0];
    return r1;
endfunction
%%eval parity8_alt(8'b0100_0110)
%%eval parity8_alt(8'b0101_0110)
```

Bit concatenation: Given two or more `Bit#()` values `x1 , ..., xk`, `{x1 , ..., xk}` concatenates these values. The result has type `Bit#(s)` , where `s` is the sum of the bit-widths of all concatenated elements. For example:

```
Bit#(2) a = 2’b11;
Bit#(4) b = 4’b1001;
Bit#(3) c = 3’b010;
Bit#(9) x = {a, b, c}; // 9’b111001010
Bit#(5) y = {a, c};               // 5’b11010
Bit#(3) z = {1’b1, a}; // 3’b111
```

Example 2: Ripple-carry adder. A ripple-carry adder is a simple (but slow) adder that implements the elementary addition method (i.e., it adds numbers the way we saw in Lecture 1 https://6191.mit.edu/web/_static/fall23/resources/lectures/L01.pdfof 6.191).

An n-bit ripple-carry adder takes two n-bit inputs and a 1-bit *carry-in* input and returns an (n+1)-bit output with the sum of all inputs. The most-significant bit of the output is called the *carry-out* bit. This allows constructing a ripple-carry adder as a concatenation of 1-bit adders, also called *full adders*, where the carry-out of the h adder becomes the carry-in of the adder.

The `rca4` function below implements a 4-bit ripple-carry adder using bit selection, concatenation, and bitwise logical operators:

In [ ]:
```
function Bit#(2) fullAdder(Bit#(1) a, Bit#(1) b, Bit#(1) carryIn);
    Bit#(1) sum = a ^ b ^ carryIn;
    Bit#(1) carryOut = (a & b) | (carryIn & a ) | (carryIn & b);    // majority (at least two 1s)
    return {carryOut, sum};
endfunction

function Bit#(3) rca2(Bit#(2) a, Bit#(2) b, Bit#(1) carryIn);
    Bit#(2) lower = fullAdder(a[0], b[0], carryIn);
    Bit#(2)upper = fullAdder(a[1], b[1], lower[1]); // carry-out of previous full adder used as carry-in
    return {upper, lower[0]};
endfunction

function Bit#(5) rca4(Bit#(4) a, Bit#(4) b, Bit#(1) carryIn);
    Bit#(3) lower = rca2(a[1:0], b[1:0], carryIn);
    Bit#(3) upper = rca2(a[3:2], b[3:2], lower[2]);
    return {upper, lower[1:0]};
endfunction

// Expected 5+3+1 = 9
%%eval rca4(5,3,1)
```

Other operators: `Bit#(n)` supports more complex built-in operators, including arithmetic operators (`+` , `-` , `*` , ...) and relational operators (`>` , `>=` , etc.), These operations treat `Bit#(n)` values as *unsigned integers*.

Though complex operators are convenient in large designs (they express a lot of logic!), they can obscure implementation details. For this reason, we discourage their use when first learning Minispec, and their use is not permitted in the first few labs on digital design. We will lift this restriction later.

To see these operators and further details on the ones we have introduced, see Section 6.1 of the Minispec reference (https://6191.mit.edu/_static/fall23/resources/references/minispec_reference.pdf).

Example 3: Majority, again (or why we'll pass on complex operators for now). The function below implements the `majority` function we saw above, but using a single `Bit#(3)` argument instead of three `Bool` arguments, and a `Bit#(1)` output instead of a `Bool` output.

```
In [ ]: function Bit#(1) majority(Bit#(3) x) = (x[0] & x[1]) | (x[0] & x[2]) | (x[1] & x[2]);
        %%eval majority(3'b101)
        %%eval majority(3'b001)
```

From looking at the operators used, it is pretty clear that this function can be implemented using three 2-input AND gates and two 2-input OR gates.

Now, consider an alternative implementation that uses complex operators:

```
In [ ]:  function Bit#(1) majority_alt(Bit#(3) x);
            Bit#(2) sum = {1'b0, x[0]} + {1'b0, x[1]} + {1'b0, x[2]};
            return (sum >= 2)? 1'b1 : 1'b0;
        endfunction
        %%eval majority_alt(3'b101)
        %%eval majority_alt(3'b001)
```

This implementation is arguably a more direct description of the majority function: it sums the bits of the input and returns 1 if over half of them are 1. Moreover, this implementation is **much** easier to extend to more bits (e.g., to compute the majority of a `Bit#(7)` input instead of a `Bit#(3)`, we could add the 7 bits and compare with 4). But it's harder to see how this function would be implemented: synthesizing this function by hand requires knowing how addition and comparisons are implemented. (It also requires knowing more syntax.)

Thus, until we see the underlying implementation of more complex operators in labs, we will refrain from using these operators.

5. Types and type inference

Minispec is a *strongly typed language* with *static type checking*: every variable and expression has a type, and that type must be known statically, i.e., at compile time. Variables must be assigned values with compatible types.

We have already seen two types, `Bool` and `Bit#(n)`, and have informally seen strong typing and static type checking at work. For instance, so far we have specified the type of every variable (e.g., `Bool x = True;`), and seen how every variable has a fixed type: whereas in languages like Python one can write `x = True` and later `x = 2`, in Minispec you cannot have the same variable take `Bool` and `Bit#(n)` values.

Minispec has other built-in types and the ability to define your own types. We will introduce these concepts later.

Type conversions are explicit: Unlike in some languages, in Minispec almost all conversions between values of different types are explicit. For example, it is illegal to assign a `Bool` value to a `Bit#(1)` variable (or vice versa), even though both take one bit to represent. Similarly, it is illegal to assign `Bit#(4)` value to a `Bit#(8)` variable, because they have a different number of bits and *implicit bit-width extensions never happen*. This makes code more verbose but reduces mistakes, as shown below:

```
Bool b = True;
Bit#(1) x = b;                  // Error, cannot convert from Bool to Bit#(1)
Bit#(1) y = b? 1 : 0;           // OK, uses ternary operator to convert explicitly

Bit#(4) i = 12;
Bit#(8) j = i;                  // Error, mismatched bit-widths 4 and 8
Bit#(8) k = {0, i};             // OK, uses concatenation to zero-extend i
```

The one exception to this rule is the `Integer` type, which we will see in Section 8. `Integer` values work just like unsized literals and can be assigned to `Bit#(n)` variables without explicit conversion. For example:

```
Integer n = 3 * 4; // n=12
Bit#(8) x = n;                // OK
Bit#(n) y = n * n; // OK, Bit#(12) y = 144;
```

Type inference: Minispec allows omitting a variable's type by using the `let` keyword. The compiler will infer the variable's type from the expression assigned to the variable. For example:

```
Bit#(4) x = 1;
let y = x;          // Bit#(4)
let z = {x, x};     // Bit#(8)
let w = 2'b11;      // Bit#(2)
let n = 42;         // Integer
```

Truncation and extension of Bit#(n) values: Since Minispec does not implicitly extend or truncate values, it provides three built-in functions to do length conversions:

- `truncate` truncates the most-significant bits of its argument to match the bit-width of a narrower destination.
- `zeroExtend` adds zeros to the left of the argument to match the bit-width of a wider destination.
- `signExtend` extends the argument by replicating its most significant bit to match the bit-width of a wider destination. For example:

```
Bit#(4) a = 4'b1001;
Bit#(2) x = truncate(a);                // 2'b01
Bit#(6) y = zeroExtend(a);              // 6'b001001
Bit#(6) z = signExtend(a);              // 6'b111001
Bit#(6) w = signExtend(x);              // 6'b000001
```

If a `Bit#(n)` value encodes an unsigned integer, `zeroExtend` preserves its value; if the variable encodes a signed integer in two's complement representation, `signExtend` preserves its value.

6. Conditional expressions

Conditional expressions allow selecting between two or more values depending on another value.

Conditional operator: The conditional or ternary operator selects between two values based on a `Bool` value. Its syntax is:

```
condExpr? trueExpr : falseExpr
```

where `condExpr` is a `Bool` expression, and `trueExpr` and `falseExpr` are expressions of the same type. If `condExpr` is `True` , the expression evaluates to `trueExpr` ; otherwise, it evaluates to `falseExpr` .

Each conditional operator is synthesized as a **multiplexer** or mux, specifically a 2-to-1 mux. The code below implements a multiplexer of
`Bit#(4)` values:

```
In [ ]: function Bit#(4) mux4(Bit#(4) a, Bit#(4) b,
        Bool s) = s? a : b; %%eval mux4(2, 7, True)
        %%eval mux4(2, 7, False)
```

Case expression: The case expression allows selecting among multiple values depending on a value. For example, the function below uses a case expression to compute whether its 4-bit input is prime. The case expression enumerates all 4-bit values that are prime; if so, the case expression evaluates to `True` . If none of the values match, the case expression evaluates to the `default` value, `False` :

In [ ]:
```
function Bool isPrime(Bit#(4) x) = case (x)
   1 : True;
   2 : True;
   3 : True;
   5 : True;
   7 : True;
   11: True;
   13: True;
   default: False;
endcase;
%%eval isPrime(13)
%%eval isPrime(10)
```

The case expression must always evaluate to a value, so it can omit the default item only when it enumerates all possible values of the expression being compared (e.g., of `x` in the above example).

Case expressions synthesize to **N-to-1 muxes**. For example, the function above can be synthesized with a 16-to-1 1-bit mux, where x is used as the select input, inputs 1, 2, 3, 5, 7, 11, and 13 are tied to 1, and all other others are tied to 0. (As we will see in Section 7, in practice, synthesis tools will use optimizations to simplify the implementation, but conceptually it is a mux).

No "conditional execution": Many software programming languages provide conditional and case expressions or similar constructs, and their implementation typically uses *conditional execution*, i.e., evaluating only the expression in the path of interest.

For example, consider the following conditional expression in C: `r = c ? foo(x) : bar(y)` (in Python this would be `r = foo(x) if c else bar(y)`). When compiled to assembly, this code will evaluate `c` , then either call `foo(x)` if `x` is `True` , or call `bar(y)` if `x` is `False` , and assign the result to variable `r` . In other words, the code will call **either** `foo(x)` or `bar(y)` , but not both. This avoids wasted work because only one result is needed, and `foo` and `bar` may be long functions.

This is **not at all** how conditional expressions are synthesized in Minispec, because combinational logic does not allow for conditional execution. Remember: functions describe combinational circuits, functions are inlined when called, and conditional expressions translate to muxes. Thus, synthesizing `r = c? foo(x) : bar(y)` will (1) synthesize `foo(x)` , i.e., creating an instance of the circuit implementing function `foo` with input `x` ; (2) synthesize `bar(y)` ; and (3) use a 2-to-1 mux to select between the outputs of `foo(x)` and `bar(y)` . Note how the circuit will evaluate **both** `foo(x)` and `bar(y)` .

"Conditional execution" is a common source of confusion, so this bears repeating: **there is no conditional execution with combinational logic**. This is important because, when thinking about the area and time a circuit will take, you must consider that **all** paths will be evaluated, not just one.

7. Automatic synthesis of functions into combinational circuits

In Section 2 we saw how to synthesize functions manually, with pen and paper. But this gets tedious quickly, so we provide tools to automatically synthesize Minispec code into gate-level circuits.

In Jupyter, functions can be synthesized using the `%%synth` magic. For example, let's synthesize the `and4` function from Section 2 and visualize the resulting circuit:

In [ ]:
```
%%synth and4 -v
```

The result of `%%synth` above has four parts:

1. A *summary* of the characteristics of the synthesized circuit: number of gates, area, and *critical-path delay* (i.e., the longest propagation delay from any input to any output). For example, the `and4` circuit has 3 gates that take 2.39 square microns (um^2) and a critical-path delay of 23.53 picoseconds (ps).
2. A *critical path report*, showing the input and output wires involved in the critical path and the delay accumulated through individual gates in the path between them.
3. An *area report*, breaking down the area consumed by each type of gate.
4. If the `-v` (view) flag is given, a drawing of the circuit.

`%%synth` performs **synthesis optimizations** to achieve a circuit implementation with low delay and area. This means that, in general, you will not see the same gates being used in the implementation as the ones the function specifies. For example, looking at the function code, you may have expected a circuit using 3 2-input AND gates; however, `synth` uses two 2-input NAND gates and one 2-input NOR gate. In general, you'll see inverting gates being used more often, as they are more efficient in current fabrication technology (CMOS).

`%%synth` has more options (e.g., to control what library of gates to use, whether to optimize for delay or area, etc.). Run `%%synth -h` for a list of options. `%%synth` will be used in labs but we will not describe it further in this tutorial.

You can try synthesizing a few more of our previous functions:

In [ ]:
```
 %%synth rca2 -v
// The extended library has more gates
%%synth rca2 -l extended -v
```

In [ ]:
```
 // By default synth seeks to minimize delay, then area.
// If you give it a high delay target (1000 ps below), it will seek to minimize area.
%%synth parity8 -l extended -d 1000 -v
```

8. Parametric functions, Integers, and static elaboration

Minispec provides *parametric types*, such as `Bit#(n)`, that take one or more *parameters*, such as n in this case. These parameters must be known at compile time, and when specified, yield a concrete type that can be implemented in hardware (such as `Bit#(4)`, a 4-bit value).

Minispec also allows defining *parametric functions*. These enable writing generic circuit descriptions that are then concretized as needed.

For example, so far we have implemented functions specialized to particular bit-widths, like `parity2`, `parity4`, and `parity8` for 2-, 4-, and 8-bit inputs, respectively. Instead, we can write a *single* `parity#(n)` function that computes the parity of an n-bit input. Then, users of the function will call it with particular values of n, instantiating parity circuits of different widths. Let's go ahead and do that:

In [ ]:
```
function Bit#(1) parity#(Integer n)(Bit#(n) x) =
    (n == 1)? x : x[n-1] ^ parity#(n-1)(x[n-2:0]);

%%eval parity#(8)(8'b0100_0110)
%%eval parity#(128)(-1)
```

There is a lot going on here! First, the function name is followed by `#(Integer n)`, denoting that this function has an `Integer` parameter called `n`. `Integer` is a special type that is *evaluated at compile time*. `n` is used within the function, including to select bits, in a conditional expression, etc. Second, the body of the function does one of two things: if `n` is 1, it returns `x`, which in this case is a `Bit#(1)` value; otherwise, it XORs the most-significant bit of `x` with the parity of its remaining bits, which it computes by calling `parity#(n-1)`.

This example conspicuously violates two of the rules we've established so far. First, we said *no recursion* (Section 2), but `parity#(n)` is calling `parity#(n-1)`! Second, we said *no conditional execution* (Section 6), but if `parity#(1)` instantiated both branches of the conditional, it would call `parity#(0)`, which would call `parity#(-1)`, and so on, so recursion would never end!

The reason this code works is that **parameters are always evaluated a compile-time**: they are simply a facility to make the code more expressive, but no `Integer` variable will ever make it to hardware. The two restrictions above exist because we want functions to be implementable in combinational logic, but if we are doing this evaluation at compile-time, these restrictions are unnecessary, and so they are lifted.

More concretely, given the `parity#(n)` code above, a call to `parity#(4)` will make the compiler produce the following code:

```
In [ ]: function Bit#(1) parity#(4)(Bit#(4) x) = x[3] ^ parity#(3)(x[2:0]);
        function Bit#(1) parity#(3)(Bit#(3) x) = x[2] ^ parity#(2)(x[1:0]);
        function Bit#(1) parity#(2)(Bit#(2) x) = x[1] ^ parity#(1)(x[0:0]);
        function Bit#(1) parity#(1)(Bit#(1) x) = x;

         %%eval parity#(4)(4'b1010)
         %%eval parity#(4)(4'b1011)
```

This compile-time evaluation process is known as static elaboration. Note how the above code, after expanded, follows the rules we saw before: it is not recursive **on the arguments** (only on the parameter `n`), and it terminates because the conditional statement is gone (it was conditional on a parameter, so it was evaluated at compile-time).

The rest of this section gives more detail on Integers and static elaboration semantics:

Integer type: `Integer` represents an integer value (i.e., a negative or non-negative number) with an *unbounded number of bits*. `Integer` supports the same operators as `Bit#(n)` (including logical, arithmetic, relational, and bit-reductions). However, since `Integer` is signed, arithmetic and relational operators have *signed semantics*.
`Integer` is the only type in Minispec that is not synthesizable to hardware. It is used exclusively at compile time: all `Integer` expressions must be evaluable by the compiler.

The internal representation of `Integer` values is irrelevant: they have an unbounded number of bits, so you needn't worry about whether they use two's complement or some other representation.

Parameters can be `Integer` values or other types. So far we have only seen `Integer` parameters (e.g., `Bit#(n)` and `parity#(n))`. As an example of a parametric type with a type parameter, `Vector#(n, T)` represents an n-element vector with elements of type `T`. For instance, the function below sums the elements of 4-element vector of 32-bit values:

```
In [ ]: function Bit#(32) sum(Vector#(4,
            Bit#(32)) elems) = elems[0] +
            elems[1] + elems[2] + elems[3];

         %%synth sum
```

Parametric functions are defined by specifying the parameters after the function name, enclosed in `#(...)`. Each parameter definition can be:

- `Integer intParam` to denote an `Integer` parameter with name `intParam`, which must be lowercase.
- `type TypeParam` to denote a type parameter with name `TypeParam`, which must be uppercase.
- A specific Integer literal or type, which allows writing parametric definitions that are already specialized to particular parameter values (like the `parity#(4)`, `parity#(3)`, etc. code above). Integer and type parameter names can be used anywhere in the parametric definition, including in the return type of the function.

Parametric definitions can be specialized. The code can specify both a general parametric definition with Integer and type parameters, and one or more specialized definitions with fixed Integers and types. On a match, a specialized definition takes precedence over the general one.

For example, the code below implements an n-bit ripple-carry adder `rca#(n)` using a specialized definition and a general one:

In [ ]:
```
// base case (specialized)
function Bit#(2) rca#(1)(Bit#(1) a, Bit#(1) b, Bit#(1) cin);
    let cout = (a & b) | (a & cin) | (b & cin);
    let sum = a ^ b ^ cin;
    return {cout, sum};
endfunction

// general case (used when n != 1)
function Bit#(n+1) rca#(Integer n)(Bit#(n) a, Bit#(n) b, Bit#(1) cin);
    let lower = rca#(n-1)(a[n-2:0], b[n-2:0], cin);
    let upper = rca#(1)(a[n-1], b[n-1], lower[n-1]);
    return {upper, lower[n-2:0]};
endfunction

%%eval rca#(32)(10000, 99999, 1)

// Synthesize rca#(32). If you try different sizes,
// you'll see that delay grows linearly with number of bits
%%synth rca#(32) -l multisize
```

Static elaboration: Parametrics require clear guarantees on what computations and simplifications the compiler performs at compile time.
We refer to this process as static elaboration. Minispec guarantees the following static elaboration behavior:

1. All `Integer` expressions and variables are elaborated, i.e., turned into concrete values at compile time. An `Integer` expression or variable that cannot be elaborated causes a compiler error.
2. `Bool` expressions and variables are elaborated if they depend only on `Bool` constants and `Integer` values.
3. All conditional expressions (ternary, case) and control-flow statements (if-else, case, see Section 10 whose predicate is an elaborated `Bool` or `Integer` expression are elaborated to include only the branch that they execute, eliminating all others.
4. Parametrics are instantiated and elaborated lazily, as the compiler finds code that uses them.

The example below shows some errors that happen when one tries to assign values to `Integer` variables that are unknown at compile-time:

In [ ]:
```
function Bool integerFail(Bit#(1) a, Bool b);
    // Uncomment these to see the compiler errors
    //Integer i = a;
    //Integer j = (a == 0)? 0 : 1;
    //Integer k = b? 100 : 1000 + j;
    return True;
endfunction
```

This concludes all you need to know on parametric functions. Further details are available in Section 10 of the Minispec reference (https://6191.mit.edu/web/_static/fall23/resources/references/minispec_reference.pdf).

9. User-defined types

Minispec supports three kinds of user-defined types: type synonyms, structs, and enums.

Type synonyms allow giving a different name to an existing type. Their syntax is

```
typedef Type NewType;
```

where `Type` is any existing type, and `NewType` is the new type's name. A type and its synonym can be used interchangeably. For example:

In [ ]:
```
typedef Bit#(8) Byte;
function Byte invert(Byte x) = ~x;
%%eval invert(8'd1)
```

Structs are composite types: they represent a group of members of different types.
Struct definitions use the following syntax:

```
typedef struct {
Type1 member1;
Type2 member2;
...
TypeN memberN;
} StructType;
```

where `StructType` is the (new) type for the struct, `member_i` are the (lowercase) names of its members, and `Type_i` are the (uppercase) types of its members.

For example, the code below defines a 24-bit pixel with 8-bit red, green, and blue components (following the typical red-green-blue representation for on-screen colors):

In [ ]:
```
 typedef struct {
    Byte red;
    Byte green;
    Byte blue;
} Pixel;
```

Struct values can be constructed with the following syntax:

```
StructType { member1 : expr1, ..., memberN : exprN }
```

where `StructType` is the struct's type name, `member_i` are the struct members' names, and `expr_i` denote the values that the members should take.

For example, the code below defines the `cyan` color constant (no red, full green and blue):

In [ ]:
```
Pixel cyan = Pixel{ red : 0, green :

255, blue : 255 }; %%eval cyan
```

Struct members can be accessed as follows: given value `s` of a struct type that has a member with name `m`, the expression `s.m` yields the value of member `m`. For example:

In [ ]:
```
 %%eval cyan.red + cyan.blue
```

Enums or *enumerations* represent a set of unique symbolic constants, called *labels*. Enums can be defined using the following basic syntax:

```
typedef enum { Label1, ..., LabelN } EnumType;
```

where `EnumType` is the enum's type name, and `Label_i` are the names of the labels. Labels must be uppercase, and can be repeated across enum definitions. A value of type `EnumType` can take one of these labels.

The compiler internally represents an enum with possible labels as a -bit value. With the syntax above, `Label_i` will take ⌈ 2 ⌉ numeric value `i-1` (i.e., labels take consecutive values starting from 0). It is possible to assign the numeric value of each label explicitly using the following syntax:

```
typedef enum { Label1 = val1, ..., LabelN = valN } EnumType;
```

where `val_i` are the distinct numeric values of the labels.

For example:

```
In [ ]:  typedef enum { Ready, Busy, Error } State;
        State state = Ready;
        typedef enum { Red = 0, Blue = 2, Green = 1 } PixelChannel;
```

Example: Comparators. Comparators illustrate the benefits of structs and enums in defining combinational logic. First, structs come in handy when we want functions to return more than one named output value. For example, consider the code for a comparator of unsigned n-bit integers. Given inputs `a` and `b`, the `compare#(n)` function below divides them in upper and lower portions, and builds the comparator recursively: `a` is less than `b` if `a`'s upper part is less than `b`'s or their upper parts are equal and `a`'s lower part is smaller.

To clarify the intent of the code, `compare#(n)` returns a struct with two `Bool` members, `lessThan` and `equal`. While the function could return a `Bit#(2)` value, we'd have to remember which bit represents which condition. Using a struct makes the code much more descriptive:

```
In [ ]:  typedef struct {
            Bool lessThan;
            Bool equal;
        } CompareResult;

        function CompareResult compare#(Integer w)(Bit#(w) a, Bit#(w) b);
            CompareResult upper = compare#(w-w/2)(a[w-1:w/2], b[w-1:w/2]);
            CompareResult lower = compare#(w/2)(a[w/2-1:0], b[w/2-1:0]);
            return CompareResult {
                equal : upper.equal && lower.equal,
                lessThan : upper.lessThan || (upper.equal && lower.lessThan)
            };
        endfunction

        // Base case
        function CompareResult compare#(1)(Bit#(1) a, Bit#(1) b);
            return CompareResult { equal: (a ^ b) == 0, lessThan: (~a & b) == 1 };
        endfunction

        %%eval compare#(16)(2500, 3700)
        %%eval compare#(16)(4500, 3700)
        %%eval compare#(7)(42, 42)
```

A drawback of the above `compare#(n)` implementation is that `CompareResult` admits a value that makes no sense: a number cannot be both less than and equal to another number. The alternative implementation `cmp#(n)` below uses an enum with three labels instead to avoid this issue:

```
In [ ]:  typedef enum { LessThan, Equal, GreaterThan } CmpRes;

        function CmpRes cmp#(Integer w)(Bit#(w) a, Bit#(w) b);
            CmpRes upper = cmp#(w-w/2)(a[w-1:w/2], b[w-1:w/2]);
            CmpRes lower = cmp#(w/2)(a[w/2-1:0], b[w/2-1:0]);
            return (upper == Equal && lower == Equal)? Equal :
                   (upper == LessThan || (upper == Equal && lower == LessThan))? LessThan : GreaterThan;
        endfunction

        // Base case
        function CmpRes cmp#(1)(Bit#(1) a, Bit#(1) b);
            return case ({a, b})
                2'b00: Equal;
                2'b01: LessThan;
                2'b10: GreaterThan;
                2'b11: Equal;
            endcase;
        endfunction
        %%eval cmp#(16)(2500, 3700)
        %%eval cmp#(16)(4500, 3700)
        %%eval cmp#(7)(42, 42)
```

10. Higher-level constructs and advanced features

Finally, Minispec incorporates several constructs, such as if-else statements and for loops, that are familiar in other programming languages but are not trivial to synthesize to combinational logic.

We present these constructs last for two reasons. First, they are the least necessary to know: you can design combinational circuits without
them. Second, *they can be confusing*: their implementation in combinational logic is quite different from the equivalent constructs in programming languages and it is not trivial. Thus, it is crucial that you develop a clear understanding of what circuit each of these these constructs generates, and in this section we present the implementation of each construct together with its description.

Variable assignments

So far, we have only seen variable declarations and assignments of the form:

```
Type varName = expression;
```

This declares a new variable of type `Type`, named `varName`, and assigns it the value given by `expression`.

Once declared, the previous examples never reassigned the value of a variable. But a variable can be assigned to multiple times. Each assignment changes the value bound to the variable. Every statement following the assignment sees the new value. For example:

In [ ]:
```
 function Bit#(1) parity4(Bit#(4) x);
    Bit#(1) result;
    result = x[0] ^ x[1];
    result = result ^ x[2];
    result = result ^ x[3];
    return result;
endfunction

%%eval parity4(4'b0111)
```

It is also possible, though not recommended, to reassign individual bits, bit ranges, or struct members of a variable. For example:

In [ ]:
```
 function Bit#(1) parity4(Bit#(4) x);
    x[1:0] = x[3:2] ^ x[1:0];
    x[0] = x[1] ^ x[0];
    return x[0];
endfunction

%%eval parity4(4'b0111)
```

Structs, `Bit#(n)` variables, and other compound types can be declared uninitialized and then initialized element by element. However, it is illegal to use a `Bit#(n)` variable that is partially initialized, even if the particular bits being accessed have been initialized. To avoid this problem, you can initialize the entire `Bit#(n)` variable (e.g., to `0`), then reassign the individual bits. Structs do not suffer from this limitation.

Synthesis: Variables require no logic. Each variable assignment is simply naming the value (i.e., wires) of a particular expression, so that it can be used somewhere else.

Note that, unlike in software programming languages, **variables are not state**. They are not stored in registers or memory, they are just names for wires in a combinational circuit.

Begin-end statements

A begin-end statement denotes a block of code. It allows combining multiple statements into a single statement. Begin-end blocks can be used anywhere a statement is required, and are often used with control-flow statements `if` and `for`, which we will see next. Its syntax is:

```
begin stmt1 ... stmtN end
```

where `stmt_i` are statements.

Each begin-end block initiates a new lexical context, which supports local variable declarations. Variables are *lexically scoped*: a variable may only be used inside the block of code it is defined in. Thus, a variable declared within a begin-end block cannot be used outside the block.

If statements

If statements have the following syntax:

```
if (condExpr) trueStmt [ else falseStmt ]
```

where `condExpr` is a `Bool` expression, and `trueStmt` and `falseStmt` are statements. If `condExpr` evaluates to True, then `trueStmt` takes effect. Otherwise, if the optional else clause is present, `falseStmt` takes effect. For example:

In [ ]:
```
 function Bit#(4) max4(Bit#(4) a, Bit#(4) b);
    Bit#(4) result = b;
    if (a > b) result = a;
    return result;
endfunction

// Alternative implementation using
if-else function Bit#(4)
max4_alt(Bit#(4) a, Bit#(4) b);
    Bit#(4) result;
    if (a >
    b)
    result
    = a;
    else
    result
    = b;
    return
    result;
endfunction

%%eval max4(3, 8)
%%eval max4_alt(3, 8)
```

As shown in the example below, if statements often use begin-end blocks, and multiple if-else statements can be chained (`if (cond1) ... else if (cond2) ...`):

In [ ]:
```
 function Bit#(2) ifExample(Bit#(2) x, Bit#(2) y);
    Bit#(2) z;
    if (x > 2) z = x;
    else if (y > 2) z = y;
    else begin
        let w = x + y;
        z = w + 1;
    end
    return z;
endfunction

// First branch (if)
%%eval ifExample(3, 1)
// Second branch (else if)
%%eval ifExample(1, 3)
// Third branch (else)
%%eval ifExample(1, 0)
```

Finally, return statements can be directly present in if-else branches. For example:

In [ ]:
```
 function Bit#(4) max4_ret(Bit#(4) a, Bit#(4) b);
    if (a > b) return a;
    else return b;
endfunction

%%eval max4_ret(3, 8)
```

Synthesis: Just like conditional expressions, if statements are translated to multiplexers, but their synthesis is less obvious.

All expressions within the if statement are synthesized, then each variable assigned to within the if statement is followed by a multiplexer to select between the value assigned within the if branch (if the predicate is true) and the value before the if branch (if the predicate is false).

If-else statements are similar: the if and else branches are both synthesized. For variables assigned to in both branches, the multiplexer selects between the value in the if branch and the value in the else branch; for variables assigned to in only one of the branches, the multiplexer selects between the value within the branch and the old value.

It bears repeating that if statements do not change the basic fact that **there is no conditional execution in combinational circuits**, as we saw with conditional expressions. All branches of the if statement are synthesized and evaluated, even though at most one branch takes effect.

Another way to see this is that if statements can be systematically rewritten to conditional expressions. For example, here are the rewritten `max4` implementations following the procedure above:

In [ ]:
```
 function Bit#(4) max4_rewritten(Bit#(4) a, Bit#(4) b);
    Bit#(4) result = b;
    // The next two lines are a rewrite of if (a > b) result = a;
    Bool ifCond = (a > b);
    result = ifCond? a : result;
    return result;
endfunction

// Alternative implementation using if-else
function Bit#(4) max4_alt_rewritten(Bit#(4) a, Bit#(4) b);
    Bit#(4) result;
    // The next two lines are a rewrite of if (a > b) result = a; else result = b;
    Bool ifCond = (a > b);
    result = ifCond? a : b;
    return result;
endfunction

%%eval max4_rewritten(3, 8)
%%eval max4_alt_rewritten(3, 8)
```

Finally, when a function has multiple return statements in if-else branches (as in `max4_ret` above), the output value is selected using multiplexers in the same way as is done in an if-else statement.

Case statements

Case *statements* have a similar syntax to case *expressions*:

```
case (compExpr)
value1 : stmt1;
value2 : stmt2;
...
    [ default : defaultStmt; ]
    endcase
```

A case statement tests `compExpr` against the `value_i` values, and on a match with `value_i` , `stmt_i` takes effect. If there are no matches and the optional default label is specified, the optional `defaultStmt` takes effect. Unlike case expressions, case statements need not enumerate all values or specify a default statement. If there are no matches and no default, none of the statements takes effect.

For example, here is the `cmp#(1)` function from Section 9 using a case statement instead of a case expression:

```
In [ ]: function CmpRes cmp_case#(1)(Bit#(1) a, Bit#(1) b);
            CmpRes res = Equal;
            case ({a, b})
                2'b01: res = LessThan;
                2'b10: res = GreaterThan;
            endcase
            return res;
        endfunction

        %%eval cmp_case#(1)(0, 0)
        %%eval cmp_case#(1)(0, 1)
        %%eval cmp_case#(1)(1, 0)
```

Synthesis: Case statements are synthesized similarly to if statements: all branches are synthesized, and a multiplexer is used for each value assigned within the case statement to select the value from the right branch.

For loops

For loop statements allow compactly expressing a sequence of similar statements. They have the usual syntax:

```
for (Integer iVar = initExpr; testExpr; iVar = updExpr) stmt;
```

where `iVar` is the name of the `Integer` induction variable; `initExpr` is the induction variable's initial value; `testExpr` is a `Bool` expression that denotes whether to stop iterating; `updExpr` is evaluated after each iteration to update `iVar`; and `stmt` is the statement executed on each iteration. For example:

```
Bit#(6) w;
for (Integer i=0; i<6; i=i+1)
    w[i] = x[i % 2];
```

For loops are **not** like general loops in programming languages. Whereas general loops may iterate on values unknown at compile time and may have an unknown number of iterations, Minispec for loops have a known iteration count and are unrolled at compile time. Note how the induction variable **must** be an `Integer`, which forces the iteration count to be evaluable at compile time. These restrictions make for loops implementable with combinational logic.

Synthesis: For loops are *unrolled at compile time*, i.e., expanded into a fixed sequence of iterations. Then, each iteration is synthesized as usual.

For example, the above loop synthesizes to:

```
w[0] = z[0];
w[1] = z[1];
w[2] = z[0];
w[3] = z[1];
w[4] = z[0];
w[5] = z[1];
```

Example 1: Ripple-carry adder. The function below implements an n-bit ripple-carry adder using a for loop instead of recursion, as we used to do before.

In [ ]:
```
function Bit#(n+1) rca_loop#(Integer n)(Bit#(n) a, Bit#(n) b, Bit#(1) carryIn);
    Bit#(n+1) result = zeroExtend(carryIn);
    for (Integer i = 0; i < n; i = i + 1)
        result[i+1:i] = fullAdder(a[i], b[i], result[i]);
    return result;
endfunction

%%eval rca_loop#(16)(42, 86, 0)

// Can you see the carry chain in the 4-bit adder below?
%%synth rca_loop#(4) -O0 -l extended -v
```

This adder function produces the same circuit as our previous recursive definition: both have a carry chain. However, beware that when functions have a natural tree implementation, for loops tend to force a slower, chained implementation, as we will see in the next example.

Example 2: Comparator. The function below implements an n-bit comparator using a for loop.

In [ ]:
```
function CmpRes cmp_loop#(Integer n)(Bit#(n) a, Bit#(n) b);
    CmpRes result = Equal;
    for (Integer i = n -1 ; i >= 0; i = i - 1) begin
        if (result == Equal && a[i] == 0 && b[i] == 1) result = LessThan; if
        (result == Equal && a[i] == 1 && b[i] == 0) result = GreaterThan;
    end
    return result;
endfunction

%%eval cmp_loop#(32)(42, 57)
%%eval cmp_loop#(32)(42, 42)
```

Though shorter than our previous recursive implementation, this version produces a different, slower circuit: the loop version synthesizes to a long chain of multiplexers on `result` (one per if statement, two per iteration), so the number of logic levels grows linearly with the number of bits. By contrast, our previous `cmp#(n)` implementation synthesizes to a tree of function calls.

You can synthesize both versions to see which is faster:

In [ ]:
```
 // Without logic optimization
%%synth cmp_loop#(8) -O0
%%synth cmp#(8) -O0
```

In [ ]:
```
 // With logic optimization (default)
%%synth cmp_loop#(8) -O1
%%synth cmp#(8) -O1
```

As you can see, in this case `%%synth` 's logic optimizations are smart enough to produce similar circuits, but without optimization (-O0 flag), the loop version is significantly slower.

NOTE: You may experience very long synthesis times when synthesizing `cmp_loop#(n)` for large values of `n` (e.g., 32). This is because the tools have trouble with very long chains of if statements that assign to the same variable. For this reason, please avoid using long chains of if statements in your code.

Don't care values

A question mark (`?`) denotes a special *don't-care* value. Don't-care values can be assigned to variables of any type. A don't-care denotes that the value is irrelevant; the compiler is free to pick any value for it. This flexibility often lets the compiler produce better circuits.

For example:

```
In [ ]:  function Bit#(4) dontCareExample(Bit#(4) a, Bool s) =
            s? ? : ~a;

         %%synth dontCareExample -v
```

In `s? ? : ~a`, to minimize the amount of logic, the compiler picks `~a` for the don't-care value `?`. This leaves `s? ~a : ~a`, which can be simplified to `~a`. As you can see, the synthesized circuit ignores `s` and is just `~a`.

Conclusion

Putting all these features together, we have seen how Minispec lets you design combinational circuits with programming constructs that you are already familiar with: functions, function composition, expressions, variables, conditional statements and loops, etc.

The result is that Minispec code looks similar to that of a serial, software programming language. This makes the code readable and easily understandable, even though it is describing an inherently parallel circuit. However, to design good hardware, it is **crucial to understand how these constructs are synthesized to combinational logic**. For example, functions are always inlined, for loops are always unrolled, and there is no conditional execution (the expressions in all paths of conditional expressions and statements are evaluated). In other words: **never forget that you are describing hardware**, not software.

This tutorial has covered combinational logic only. The (upcoming) sequential logic tutorial builds on this tutorial to show you how to design sequential (i.e., stateful) circuits.

This tutorial omits some of the finer details of the Minispec language; to dig deeper, please check the Minispec reference (https://6191.mit.edu/_static/fall23/resources/references/minispec_reference.pdf) for the full semantics.